Solving Travelling Salesman Problems with Genetic Algorithms

Samuel Jackson, University of Aberystwyth

I. Introduction

A genetic algorithm (GA) is a search and optimisation method frequently used to find approximate solutions in challenging problem domains. Genetic algorithms are inspired by the biological concept of natural selection. This places genetic algorithms under the category of biologically inspired approaches to optimisation; along with genetic programming, ant colony optimisation, and particle swarm optimisation. Many traditional optimisation techniques rely on the calculation of derivatives and often requires good knowledge of the search space. GAs on the other hand only require a measure of solution quality, making them well suited to difficult optimisation problems where traditional techniques would otherwise fail.

In a genetic algorithm solutions to a problem are encoded as chromosomes. A chromosome in GA terminology is usually an array of binary, integer, or real numbers but other representations are possible. For example, an array of real numbers might represent the encoding of coefficients of a polynomial in a curve fitting problem. A gene in GA terminology is a single atomic component of a chromosome. In the previous curve fitting example a gene would be a single coefficient.

A GA proceeds by creating an initial population of randomly generated solutions. From this population a subset of candidates are selected which are used to generate the next population. New solutions are generated from this subset using genetic operators. Their are two fundamental types of genetic operators: crossover and selection. Crossover creates new chromosomes from two or more parent chromosomes by combining portions from each of the parent chromosomes together in some way. Crossover aims to preserve some information about what makes decent solutions between generations. Mutation randomly modifies a chromosome by altering one or more of its genes. The mutation operator aims to encourage more exploration of the search space to avoid local minima. Finally, each chromosome in the new population is evaluated according to its fitness. The fitness of a chromosome is how well the solution encoded by the chromosome solves the problem. The fitness function is the entirely dependant on the problem domain. In the curve fitting example above the fitness function could be the mean squared error between a dataset and the polynomial represented by a particular chromosome.

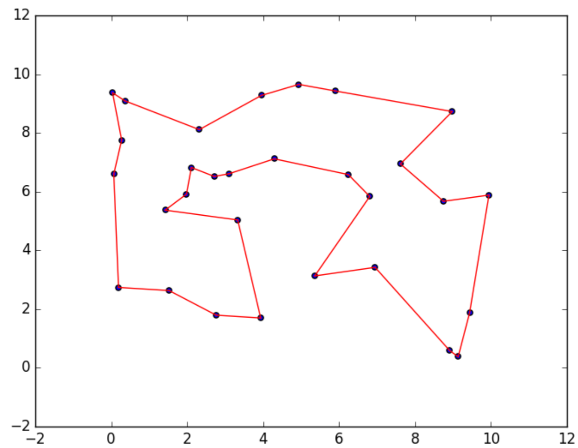

Fig. 1. Example of a solution to a TSP with 30 cities distributed uniformly at random. Each of the blue points represents a city. Each of the red lines indicates which city to travel to next. The solution shown is most likely optimal.

The travelling salesman problem (TSP) is a classic mathematical problem well suited for the application of GAs. The premise of the TSP is as follows:

> *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*

The problem is renowned for being simple to grasp but computationally intensive to calculate an exact solution. The TSP has been shown to be NP-hard and a brute force solution requires O(n!) time. Algorithms with factorial time complexity become unworkable with anything other than very small datasets. GAs easily lend themselves to the travelling salesman problem. A GA makes no guarantees about finding an optimal solution to a TSP, but can be used to find an approximate solution in a reasonable amount of time given decent parameters. It is able to do this faster than with a basic brute force search because a GA will only examine a subset of solutions in the search space that may or may not include the optimum result, but should with good parameters converge towards the optimum solution.

II. Program Description

My solution to this assignment is implemented as a Python package with a basic command line interface. The installation of the package and the operation of the command line interface are described in detail in appendices A and B respectively.

The main implementation of the GA algorithm is within the sub-package *tspsolver.ga*. This package contains a separate module for each of the components of the genetic algorithm. It also includes the simulator module, which is responsible for composing each of the components and running the genetic algorithm itself.

Each of the component modules contains an abstract base class for the particular type of component and several subclasses which provide concrete implementations of specific types of that component. For example, the crossover module contains a class *AbstractCrossoverOperator* which implements operations common to all crossover operators and provides an abstract method *_crossover_for_chromosomes* which all subclasses must implement in order to derive the class. Provided that each of the components implement the specified interface, the *Simulator* class will be able to setup and use the component without knowledge of the implementations details of the concrete component. In software engineering terms this architecture is known as the "strategy" design pattern. The strategy enables the behaviour of the algorithm to be dynamically selected at runtime.

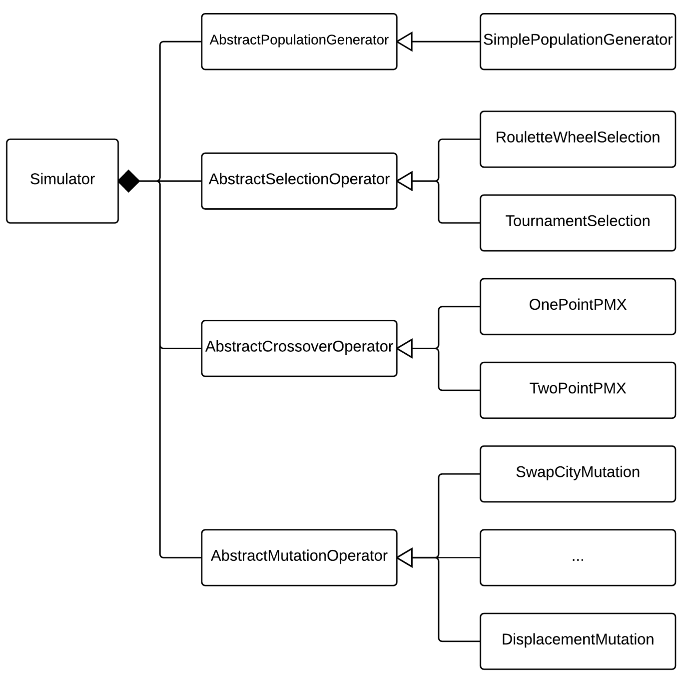

Fig. 2. Class diagram for the simulator class and its components using the strategy pattern. The simulator is composed with one of each type of component. The abstract classes provide and common interface for the components. Concrete implementations of the components derive from the relevant base class

My implementation takes this slightly further in that the *Simulator* class which composes the components of the GA together actually dynamically loads the correct components using Python's reflection capabilities according to user supplied string parameters.

Once instantiated the *Simulator* takes a 2D matrix with two columns (x and y coordinates) and n rows (equal to the number of cities) as a parameter to the evolve method. The evolve method runs converts the datasets into a distance matrix then approximates a solution using the genetic algorithm.

In addition to the main *tspsolver.ga* sub-package there are four other modules. These are really just supporting code for the running the simulator and producing the analysis. These modules are:

- **tsp_generator** - Implements a small class for generating TSP datasets with uniformly random cities.
- **command** - Implements the command line interface for running the simulator and includes a couple of file loading and saving routines.
- **plotting** - Defines some custom plotting functions for producing the graphs used in this report.
- **tuning** - A basic class for using a grid search to automatically select GA parameters.

III. Representation Discussion

For my experiments I chose to use a path based representation for chromosomes. In a path based representation each chromosome is represented as an array of integers. The value of each integer element represents a single city in the dataset. In my program this integer corresponds to the i^{th} row in the $2 \times n$ dataset. Each integer's position in the array indicates the order in which it will be visited. For example: in the array $[3, 4, 1, 5, 2]$ the first city to be visited would be city 3. For city 3 the tour then moves to city 4 and from city 4 to city 1.

Obviously since each city in the tour should only be visited once any valid solution should only contain a single occurrence of each city. Likewise because all cities must be visited exactly once each of cities in the dataset must be included in any valid solution. Therefore all valid solutions must be of size n where n is the number of cities in the dataset. In this representation format valid solutions are simply permutations of the enumeration of every city in the dataset.

While many different representations of the TSP are possible [1], a path based representation remains the most natural. It is both intuitive for the beginner to understand (of which I am one personally) and has a large array of applicable genetic operators which can be utilised. Other approaches are possible as shown in ref. [1], but these are shown to either have poor results or have peculiarities in their representations. For example in a binary representation there are some seemingly valid encodings for which their are no valid cities.

However, this does not mean that a path based representation is without its downsides. The primary issues with encoding a set of cities as a chromosome for a TSP is ensuring that the genetic operators used will produce valid tours. As mentioned in the preceding paragraphs, a valid tour needs to be a permutation of the enumeration of all cities in the dataset. It is easy to see how traditional naive genetic operators would inevitably create invalid tours by producing a solution that visits the same city twice and excludes one or more cities.

IV. Genetic Operators Discussion

V. Experiments Performed

VI. Discussion and Analysis

VII. Conclusions and Future Work

Appendix A
Installation of Program

Appendix B
Command Line Interface

This appendix provides an overview of the operation of the CLI for the application.

References

[1] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.