

# GAs To Solve The Travelling Salesman Problem

JACOB HOUSE, NABIL MIRI, OMAR MOHAMED, AND HASSAN EL-KHATIB

Memorial University of Newfoundland, Faculty of Science, Department of Computer Science

Compiled December 8, 2018

Submitted December 10, 2018

This report investigates the *Travelling Salesman Problem* through the use of a genetic algorithm (GA) employing the *inver-over* crossover operator, a combination of scramble and swap mutations, and a combination of  $\mu + \lambda$  and replacement survival selection operators.

## 1. PROBLEM SPECIFICATIONS

Wolfram MathWorld defines the travelling salesman problem as “a problem in graph theory requiring the most efficient (*i.e.*, least total distance) Hamiltonian cycle a salesman can take through each of  $n$  cities,” for which “no general method of solution is known.” [4] The website also remarks that the problem is assigned to the class of NP-hard (non-deterministic polynomial time) problems.

Given three datasets, for territories Western Sahara, Uruguay, and Canada of size 29, 734, and 4,663 cities, respectively, the objective has been to design a genetic algorithm from scratch in the Python programming language to solve the TSP for each locale using some advanced technique(s).

The first advanced technique employed in the algorithm is the *inver-over* hybrid crossover/mutation operator.

## 2. THE INVER-OVER OPERATOR

The *inver-over* operator can be regarded as either a crossover or mutation operator because it takes information from other individuals in the GA’s mating pool, yet bases a single offspring off of a single primary parent, much the same way a mutation operator is unary, performing mutation on a single individual [3]. Throughout this report, the *inver-over* operator

will be referenced as a crossover operator as this was the *inver-over* algorithm’s place in the GA developed.

### A. Usefulness

As *inver-over* embodies aspects of both crossover and mutation operators, the operator is designed to provide middle-ground to algorithms relying primarily on crossover for variation (as this is computationally expensive) and those relying on mutation for variety, since this is often ineffective in escaping local minima<sup>1</sup> [3].

### B. Representation

A set  $P$  of individuals of length  $n$ , each represented as a sequence

$$i_k = \langle C_{c_k(0)}, C_{c_k(1)}, C_{c_k(2)}, \dots, C_{c_k(n-1)} \rangle,$$

where  $c_k: \{0, 1, \dots, n-1\} \rightarrow \{1, 2, \dots, n\}$  is a bijection between indices of  $i_k$ ,  $0 \leq k < \text{card}(P)$ , and city indices as provided in the data file, is used to denote the population. Let  $M \subsetneq P$  be the mating pool containing individuals  $i_{m(0)}, i_{m(1)}, \dots, i_{m(\text{card}(M)-1)}$  with  $m: \{0, \dots, \text{card}(M)-1\} \rightarrow \{0, \dots, \text{card}(P)-1\}$  an injective function mapping individuals in the mating pool  $M$  to their equal ‘self’ in the population  $P$ .

### C. Algorithm

The algorithm (Algorithm 1) used mirrors that depicted in the article by Tao and Michalewicz [3].

<sup>1</sup>Local minima occur when the natural selection in the GA narrows the gene pool towards a ostensibly optimal solution and eliminates individuals that otherwise would evolve to become the true optimal solution.

**Algorithm 1.** The inver-over operator**Require:**

- ▷  $M$  be the mating pool
- ▷  $0 \leq i < \text{card}(M)$  be the index of the initial parent
- ▷  $n$  be the number of cities in the tour

**Ensure:**

- ▷ New offspring individual created

```

1: var  $child := M[i]$                                 ▷ Copy the parent
2: var  $unused := \{x \in \mathbb{Z} \mid 0 \leq x < n - 1\}$ 
3: var  $p := \frac{1}{2}$ 
4: var  $c := \text{rand}(unused)$                             ▷ Select randomly
5:  $unused := unused \setminus \{c\}$ 
6: while  $\text{card}(unused) > 0$  do
7:   if  $\text{rand}\{x \in \mathbb{R} \mid 0 \leq x < 1\} < p$  then
8:      $c' := \text{rand}(unused)$ 
9:   else
10:     $newPar := \text{rand}(M)$ 
11:     $newParC := \text{where}(newPar = child[c])$ 
                                         ▷ Index  $j$  in  $newPar$ 
12:     $c' := newParentC + 1$ 
13:     $unused := unused \setminus \{c'\}$ 
14:    if  $child[c \pm 1] = child[c']$  then
15:      break from the while loop
16:     $child[c : c'] := child[c' : c]$                 ▷ Invert
17:     $c := c'$ 
18: if  $\text{fitness}(child) \geq \text{fitness}(M[i])$  then
19:   return  $child$ 
20: else
21:   return  $M[i]$ 

```

Here  $M[i]$  is the primary parent from which the child is based. One can observe that all changes are then made to this individual, like a mutation, yet they involve other individuals, denoted  $newPar$  in the mating pool.

We borrow the following example of a single iteration of the algorithm from Tao and Michalewicz.

1. Let  $child = \langle 2, 3, 9, 4, 1, 5, 8, 6, 7 \rangle$  and the current city index  $c$  is 1 so  $child[c] = 3$ .
2. (a) Suppose the random number generated by  $\text{rand}\{x \in \mathbb{R} \mid 0 \leq x < 1\}$  does not exceed  $p$ . Another city index  $c'$  from the child is selected, say  $c' = 6$  so  $child[c'] = 8$ . The section of  $child$  after indices  $c$  and  $c'$  is inverted, leaving  $child = \langle 2, 3, 8, 5, 1, 4, 6, 7 \rangle$ .

- (b) Otherwise, another individual is (randomly) selected from the mating pool to become the new parent. Let this be  $\langle 1, 6, 4, 3, 5, 7, 9, 2, 8 \rangle$ . Here the city next to  $child[c] = 3$  is city 5. So, the segment of  $child$  to invert is that which starts after city 3 and terminates after city 5. This leaves  $child = \langle 2, 3, 5, 1, 4, 9, 6, 7 \rangle$ .

As in Tao and Michalewicz, we remark that in either case the resulting string is intermediate in the sense that the above inversion operator is applied several times before an offspring is evaluated. After a number of iterations, suppose  $child = \langle 9, 3, 6, 8, 5, 1, 4, 2, 7 \rangle$  and  $c = 2$  so  $child[c] = 6$ .

3. (a) If  $\text{rand}\{x \in \mathbb{R} \mid 0 \leq x < 1\}$  is greater than  $p$ , the city following  $child[c] = 6$  is selected from a randomly chosen individual in the mating pool  $M$ . Let this city be city 8. As 8 follows 6 in  $child$ , the algorithm terminates.
- (b) Otherwise, a randomly selected city is selected. This may also be 8, in which case the algorithm also terminates. If it is not 8, the algorithm continues as described in (2).

### 3. OUR ALGORITHM

#### 4. OPTIMIZATION

Before any benchmarking was done, an effort was made to use the NumPy [2] module's interfaces as much as possible over raw Python data types. For example, when possible `numpy.ndarray` was chosen over Python's `list` data structure. The SciPy<sup>2</sup> documentation states:

...the fact that [Python `lists`] can contain objects of differing types mean that Python must store type information for every element, and must execute type dispatching code when operating on each element. This also means that very few list operations can be carried out by efficient C loops — each iteration would require type checks and other Python API bookkeeping. [1]

<sup>2</sup>NumPy is part of the SciPy Stack.

As the problem being solved requires extensive accessing and mutating of arrays containing a single data type (*i.e.*, real numbers), NumPy provided a very easily-implemented speed boost.

## 5. RESULTS

## 6. CONCLUSION

## REFERENCES

- [1] *Frequently Asked Questions*. URL: <https://www.scipy.org/scipylib/faq.html#what-advantages-do-numpy-arrays-offer-over-nested-python-lists>.
- [2] Travis E. Oliphant. *Guide to NumPy*. Continuum Press, 2015.
- [3] G. Tao and Z. Michalewicz. "Inver-over operator for the TSP". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1498 (1998), pp. 803–812. ISSN: 03029743.
- [4] Eric W. Weisstein. *Traveling Salesman Problem*. URL: <http://mathworld.wolfram.com/TravelingSalesmanProblem.html>.