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 10, 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." [**tsp_wolfram_alpha**] 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 [**p44**]. 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.

2.1. 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[1] [**p44**].

2.2. Representation

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

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

where $c_k \colon \{0, 1, \ldots, n-1\} \to \{1, 2, \ldots, n\}$ is a bijection between indices of i_k, $0 \leqslant k < \operatorname{card}(P)$, 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 $i_{m(0)}, i_{m(1)}, \ldots, i_{m(\operatorname{card}(M)-1)}$ with $m \colon \{0, \ldots, \operatorname{card}(M)-1\} \to \{0, \ldots, \operatorname{card}(P)-1\}$ an injective function mapping individuals in the mating pool M to their equal 'self' in the population P.

Fitness of an individual i_k, denoted by $\operatorname{fitness}(i_k)$ in the Algorithm 1, is then determined by the formula in Equation (1).

$$\operatorname{fitness}(i_m) = \sum_{j=0}^{n-1} \left\| \overrightarrow{C_j\, C_{j+1}} \right\|, \qquad \textbf{(1)}$$

[1]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.

where

$$\left\|\overrightarrow{C_j\,C_{j+1}}\right\| = \sqrt{(x_{C_{j+1}} - x_{C_j})^2 + (y_{C_{j+1}} - y_{C_j})^2}$$

is the Eucliean distance between cities C_j and C_{j+1}.[2] Hence a lower fitness score is better.

2.3. Algorithm

The algorithm (Algorithm 1) used mirrors that depicted in the article by Tao and Michalewicz [**p44**].

Algorithm 1. The inver-over operator

```
Require:
    ▷ $M$ be the mating pool
    ▷ $0 \leqslant i < \mathrm{card}(M)$ 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 $unused := \{x \in \mathbb{Z} \mid 0 \leqslant x < n - 1\}$
3:  var $p := \frac{1}{2}$
4:  var $c := \mathrm{rand}(unused)$                       ▷ Select randomly
5:  $unused := unused \setminus \{c\}$
6:  while $\mathrm{card}(unused) > 0$ do
7:      if $\mathrm{rand}\{x \in \mathbb{R} \mid 0 \leqslant x < 1\} < p$ then
8:          $c' := \mathrm{rand}(unused)$
9:      else
10:         $newPar := \mathrm{rand}(M)$
11:         $newParC := \mathrm{where}(newPar = child[c])$
                                                   ▷ 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 $\mathrm{fitness}(child) \geqslant \mathrm{fitness}(M[i])$ 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 $\mathrm{rand}\{x \in \mathbb{R} \mid 0 \leqslant x < 1\}$ 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' (*i.e.,* $\langle 9, 4, 1, 5, 8\rangle$) is inverted, leaving $child = \langle 2, 3, 8, 5, 1, 4, 9, 6, 7\rangle$.
   (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 (*i.e.,* $\langle 9, 4, 1, 5\rangle$). This leaves $child = \langle 2, 3, 5, 1, 4, 9, 8, 6, 7\rangle$.

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 $\mathrm{rand}\{x \in \mathbb{R} \mid 0 \leqslant x < 1\}$ 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 chosen. 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. OPTIMIZATION

3.1. The NumPy Module

Before any benchmarking was done, an effort was made to use the NumPy [**numpy**] module's inter-

[2] Take indices j modulo n so when $j = n-1$, $(n-1)+1 \equiv 0 \pmod{n}$ and we compute the distance from the last city back to the first.

faces 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[3] 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. [**scipy_docs**]

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.

3.2. Cheaply Computing Distance

Suppose that the distance between cities C_j and C_{j+1} is not less than 1 and the same applies for cities C_k and C_{k+1}. That is,

$$\begin{aligned} 1 &\leqslant \left\|\overrightarrow{C_j\,C_{j+1}}\right\| \\ &= \sqrt{(x_{C_{j+1}} - x_{C_j})^2 + (y_{C_{j+1}} - y_{C_j})^2} \end{aligned}$$

and

$$\begin{aligned} 1 &\leqslant \left\|\overrightarrow{C_k\,C_{k+1}}\right\| \\ &= \sqrt{(x_{C_{k+1}} - x_{C_k})^2 + (y_{C_{k+1}} - y_{C_k})^2} \end{aligned}$$

Then $\left\|\overrightarrow{C_j\,C_{j+1}}\right\| < \left\|\overrightarrow{C_k\,C_{k+1}}\right\|$ if, and only if, $\left\|\overrightarrow{C_j\,C_{j+1}}\right\|^2 < \left\|\overrightarrow{C_j\,C_{j+1}}\right\|^2$. Hence, by adding an `assert` statement to our code to ensure that

$$(x_{C_{j+1}} - x_{C_j})^2 + (y_{C_{j+1}} - y_{C_j})^2 \geqslant 1,$$

we can omit the expensive square root operation $\frac{1}{2}(n^2 - n)$ times[4] and still maintain accurate distance comparisons and fitness scores.

After the final iteration of the GA, the true distance must still be calculated using Equation (1).

[3]NumPy is part of the SciPy Stack.

[4]If distances are computed only once (see sub-section 3.4). Otherwise the number of expensive calculations may be higher.

3.3. Maximizing Hardware Utilization

At the time of writing, nearly all modern computers utilize multiple processor cores (or in some cases multiple processors, each with multiple cores) to accomplish tasks[5]. Natively, the Python language uses only *one* core, leaving the rest underutilized.

Immediately, multi-threaded processing using Python's `multiprocessing.dummy` module was investigated to take full advantage of the computer's hardware. Multithreading was chosen over multiprocessing as this does not impose the requirement that functions be serializable (referred to by the term *pickleable* in Python). This allows local helper functions and lambda expressions to be used. This resulted in *worse* performance because, as QuantStart explains,

> ...the Python interpreter is not thread safe. This means that there is a globally enforced lock when trying to safely access Python objects from within threads. At any one time only a single thread can acquire a lock for a Python object or C API. The interpreter will reacquire this lock for every 100 bytecodes of Python instructions and around (potentially) blocking I/O operations. Because of this lock CPU-bound code will see no gain in performance when using the Threading library... [**quantstart**]

Instead, multi*processing* was used to create several Python instances when necessary, each with their own lock that does not interfere with the other processes. Unfortunately, this meant a large amount of code needed to be restructured to ensure that all functions and the data passed to them could be pickled. Implementation was done using the `Pool` class and the `Process` class from Python's `multiprocessing` module. A caveat of this is that separate `multiprocessing` processes cannot share memory without a shared data structure [**wang**]. As NumPy arrays were the preferred data structure, several functions needed to be refactored to return the result of a computation (which then is inserted in some data structure in the main program)

[5]Specifications of several of Memorial's LabNet computers used for computation are included in Appendix B.3.

rather than modifying the data structure within the function.

These changes shortened execution of the algorithm for the Uruguay dataset from nearly 5.5 hours to less than 1.5 hours on a computer with a quad-core processor. Monitoring of Windows' `taskmgr.exe` and Unix's `htop` programs revealed that CPU utilization, while spread evenly among processor cores, still was not consistently near 100%, as one may expect.

Attempting program execution on several different computers and servers led to the conclusion that the limiting factor at this point was not the CPU but rather the speed of the RAM in the computer[6].

We also remark that nodes with less cores at higher speeds (*i.e.,* `en2048ece04.engr.mun.ca`, which has a quad-core 3.2GHz processor) fared much better than nodes with many cores at slightly lower speeds (*i.e.,* `reliant.cs.mun.ca`, which has two dual-threaded quad-core processors each clocking 2.67GHz).

3.4. Pre-computing All Distances

Initially, distance between cities C_j and C_{j+1} was computed each time it was needed to score the fitness of an individual. Even with the 'cheap' computation technique described in sub-section 3.2, this resulted in significant needless computation as some values inevitably would be computed more than once. To avoid duplicate calculations, there were two possible approaches to take:

(i) Develop a data structure to compute distances as needed and store the values for future reference

(ii) Pre-compute all distances

The first approach avoids the cost of pre-computing distances that are never used by using a just-in-time approach and then storing the computed value. A major drawback of this is that is nullifies any advantage of multiprocessing since this approach computes small sets of distances very sporadically instead of all of the distances at one time. For this reason, the second option was chosen.

[6]Due to the size of the data being stored in memory, CPU L1, L2, and L3 caches (see Appendix B.3) were all too small to store the entire data structure; hence it needed to be stored in RAM.

4. ALGORITHM & IMPLEMENTATION

4.1. Program Entry and Preprocessing

The program entry point is in `main.py`; other local modules used in the algorithm are contained in the following directory structure.

```
main.py
crossover/
    __init__.py
    inver_over.py
fitness/
    __init__.py
    distance.py
    fitness.py
ga_helper/
    __init__.py
    offspring.py
timing/
    __init__.py
mutation/
    __init__.py
    scramble.py
    swap.py
population/
    __init__.py
    candidates.py
    initialization.py
selection/
    __init__.py
    parent.py
    survival.py
```

Initialization is done in `main.py` through a call to `initialization.init_file()` in the `population` module, which takes a path to a file and returns a `numpy.ndarray` object containing the x- and y-coordinates of the cities in the data file.

```
def init_file(populationFilePath):
    if not os.path.isfile(populationFilePath):
        raise FileNotFoundError
    return np.loadtxt(
        populationFilePath,
        delimiter=' ',
        usecols=(1, 2))
```

The array returned from this function is then passed to the `Distances()` class constructor in the `fitness.distance` module and the `gen_all()` method is called on this object to generate all $\frac{1}{2}(n^2 - n)$ distances.

```
def gen_all(self):
    city_pairs = (
        (start_idx, end_idx)
        for start_idx in ↩
          range(self._cities.shape[0])
        for end_idx in range(
            start_idx + 1,
            self._cities.shape[0]))
    with mp.Pool() as pool:
        [self._add_edge(*data_point)
            for data_point in pool.starmap(
                self._add_edge_map_helper,
                city_pairs)]
```

The `_weighted_adjacency` instance variable of the object is then passed to the main function; this way `main()` can run multiple times without reloading and recomputing the same data.

4.2. GA Initialization

Immediately in `main()`, variables such as generation limit G, candidate pool size CP, mating pool size MP, and so on are defined.

Candidates are then chosen by choosing CP enumerations from the set of permutations of $\{0, 1, \ldots, n-1\}$, where n is the number of cities. Following this part of the algorithm's initialization, the `adjacent_distance()` function is invoked on for the population. This function calls `single_cand_adjacent_distance`,

```
def single_cand_adjacent_distance(distances, ↩
    city_idx, true_distance=False):
    dist = []
    if true_distance:
        for i in range(city_idx.shape[0]):
            start_idx = city_idx[i]
            end_idx = city_idx[(i+1) % ↩
                city_idx.shape[0]]
            d = distances[start_idx][end_idx] ↩
                ** 0.5
            dist.append(d)
    else:
        for i in range(city_idx.shape[0]):
            start_idx = city_idx[i]
            end_idx = city_idx[(i+1) % ↩
                city_idx.shape[0]]
            d = distances[start_idx][end_idx]
            dist.append(d)
    return np.array(dist)
```

which uses the adjacency matrix constructed in preprocessing to form an array of pairwise distances between each adjacent set of cities in the individual. Note that `end_idx` is defined as `city_idx[(i+1) % city_idx.shape[0]]`, *i.e.,* the index of `city_idx` being accessed is found modulo `city_idx.shape[0]` $= n$ so this computes distance from the last city on the tour back to the first. From the returned sequences of distances, we compute overall fitness of each individual in the pool (using Equation (1) with the optimization discussed in sub-section 3.2).

4.3. GA Operators

Within a loop that iterates G times, the first operation performed is parent selection using the multi-pointer selection (MPS) operator. MPS randomly selects an individual from one of MP evenly spaced segments from the candidate pool and adds it to the parent pool. The operator then chooses $MP - 1$ other individuals at regular intervals of CP/MP. MPS was primarily chosen so as to minimize the risk of elitist selection emerging in our algorithm (*i.e.,* selecting only the fittest candidates), which leads to a high chance of the algorithm peaking at a local minima extremely early.

[Need to discuss inver-over's place here.]

Initial implementations of the algorithm used scramble as a mutation operator and $\mu+\lambda$ survival selection. Preliminary tests showed that the randomness introduced by the swap mutation could be harmful to a fit individual more often than it was beneficial.

Instead, a system was developed that considers the previous two generations' top fitnesses when deciding which operator(s) to use for the next generation. This is done by comparing variables `current_best_fitness` and `previous_best_fitness`; if optimal generational fitness is not improving then introduce more randomness, otherwise maintain moderate randomness.

Varied amounts of randomness were selected by adding swap mutation — which simply swaps two cities on the tour — instead of just scramble. In such cases where generational fitness was not improving, scramble mutation was favoured $4:1$, the mutation rate was increased, crossover rate was decreased $1:2$, severity of mutation[7] was increased fourfold, and survivor selection was done favouring the replacement operator, rather than $\mu+\lambda$. Replacement was favoured $3:1$ here because the new (more random) individuals may initially score lower (*i.e.,* higher Hamiltonian cycle distance) than existing individuals, however their varied genetic make-up could allow them to escape local minima more effectively. Such a scenario with $\mu+\lambda$ would result in many or all of the newer, less fit individuals immediately being removed from the population. When generational fitness is improving, the two mutation types are favoured evenly, mutation rate is lowered back to 10%, crossover rate is reset to 90%, and survival selection favours $\mu+\lambda$ 75% of the time.

4.3.1. *Severity in the Scramble Mutation*

As the scramble mutation is now being used when randomness is desirable, the decision to capitalize on this operator's strength was made. The implementation of this involves a mutation factor $f_m \in (0, 1)$.

[7] See sub-sub-section 4.3.1.

```
def scramble_individual(individual, 
    mutation_factor=0.25):
    mutant = individual
    n = mutant.shape[0]
    mutation_thresold = mutation_factor * n

    start_idx = end_idx = 0
    while end_idx-start_idx < mutation_thresold:
        start_idx = np.random.randint(0, n - 3)
        end_idx = 
            np.random.randint(start_idx+2, n-1)
    np.random.shuffle(mutant[start_idx:end_idx])
    return mutant
```

A mutation threshold is then defined by $t_m := f_m \cdot n$. Two indices, `start_idx` and `end_idx`, both between 0 and $n-1$, are selected at random from the individual. If `end_idx` $-$ `start_idx` $< t_m$ then the algorithm chooses new indices until this requirement is satisfied.

Consequently, when high entropy is required a mutation factor can be chosen to enforce this (*e.g.*, if $f_m = 0.4$ then no less than 40% of the individual will be scrambled).

5. RESULTS

6. CONCLUSION

A. SOURCE CODE

Source code and log files used in this report are available to the public from the project's GitHub repository, located at https://github.com/jwfh/cs3201-final-project.

B. COMPUTER SPECIFICATIONS

As discussed in Section 3.3, hardware limitations of the computers used for algorithm execution were something that needed to be considered and worked with in this project. Therefore, to facilitate an accurate and comprehensive report, hardware specifications for all terminals used are included in this section.

B.1. Hosts Used

A number of hosts belonging to Memorial's LabNet network were used to run the GA developed for this report. These hosts are listed below.

```
devastator.cs.mun.ca
en2048ece03.engr.mun.ca
en2048ece04.engr.mun.ca
en2048ece05.engr.mun.ca
en2048ece07.engr.mun.ca
en2048ece12.engr.mun.ca
laserbeak.cs.mun.ca
overkill.cs.mun.ca
pounce.cs.mun.ca
starscream.cs.mun.ca
```

Several other nodes were tested and not used due to lesser hardware capabilities.

```
excalibur.cs.mun.ca
excelsior.cs.mun.ca
intrepid.cs.mun.ca
reliant.cs.mun.ca
valiant.cs.mun.ca
```

B.2. Processor Speed

Specifications of the processor(s) in all nodes used for testing are included in Table 1 on page 7.

B.3. Memory Specifications

Due to conjecture made in Section 3.3 that one of the limiting factors of the program execution time was random access memory (RAM) speed, an effort was made to retrieve these specifications for the report. Unfortunately, due to system restrictions placed on LabNet computers, standard users do not have `sudo` privileges. As a result, access to the `dmidecode` and `lshw` commands needed to read information about system hardware is restricted.

Hostname	Sockets	Cores per Socket	Threads per Core	CPU max. MHz	Cache		
					L1	L2	L3
devastator	1	4	1	3800	32K	256K	6144K
en2048ece03	1	4	1	3600	32K	256K	6144K
en2048ece04	1	4	1	3600	32K	256K	6144K
en2048ece05	1	4	2	4000	32K	256K	8192K
en2048ece07	1	4	2	4000	32K	256K	8192K
en2048ece12	1	4	2	4000	32K	256K	8192K
excalibur	2	4	2	2668	32K	256K	8192K
excelsior	2	4	2	2668	32K	256K	8192K
laserbeak	1	4	1	3800	32K	256K	6144K
intrepid	2	4	2	2668	32K	256K	8192K
overkill	1	4	1	3800	32K	256K	6144K
pounce	1	4	1	3800	32K	256K	6144K
reliant	2	4	2	2668	32K	256K	8192K
starscream	1	4	1	3800	32K	256K	6144K
valiant	2	4	2	2668	32K	256K	8192K

Table 1. Computer CPU Specifications