Comparison of Different Genetic Algorithm Crossover Techniques for the Traveling Salesman Problem

*Sean Bird, Joseph Crandall, Joseph Haaga*

**TSP Overview**

The Traveling salesman problem is a NP-hard problem in combinatorial optimization. It is possible that the worst case running time for any algorithm for the TSP increases at a superpolynomial rate with the number of cites. The traveling salesman problem consists of a salesman[[1]](#footnote-0) and a set of cites. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the salesman wants to minimize the total trip length.

TSP is focused on optimization. In this context a better solution often means a solution that is cheaper. TSP is a mathematical problem, most easily expressed as a graph describing the locations of a set of nodes. Simply trying all solutions is trivial, as for N cities you have (N-1) factorial possibilities. This means that for only 10 cities there are over 360 thousand combinations to try (since the start city is defined, there can be permutations on the remaining nine - 9\*8\*7...etc.).

**Genetic Algorithm Overview**

A genetic algorithm is a search heuristic that mimics the process of natural selectio. The heuristic is used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms, which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection and crossover. Each candidate solution has a set of problems which can be mutated and altered.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation the fitness of every individual in the population is evaluated and the fitness. The fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual’s genome is modified to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. The algorithm terminates when either a maximum number of generations has been produced or a satisfactory fitness level has been reached for the population. A typical genetic algorithm requires, a genetic representation of the solution domain ,and a fitness function to evaluate the solution domain.

The standard representation of each candidate solution is an array of bits. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations.

A Genetic Algorithm Basic Process consists of the following steps. Initialization of initial population of given size. Evaluation of each population member according to desired fitness ranking. Selection and discarding of lower fitness members, and breeding of the most fit. Crossover breeding of new members by combining the fittest together.Mutation of percentage of new generation to create required variability of the members, otherwise every combination of solutions created would already be in the population. Repeat, with each new generation of members, the process runs selected more desirable traits start again until a set termination condition is met and the program stops.

**Mutation Method**

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. The classical example of a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be changed from its original state. To implement a mutation a random variable is generated which tells whether or not a particular bit will be modified. When the gene encoding is restrictive as in permutation problems, mutations are swaps, inversions,and scrambles.The purpose of mutation in GAs is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. For different genome types different mutation methods are available. The Fit bit mutation operator takes the chosen genome and inverts the bits. The boundary mutation operator replaces the genome with either a lower or upper bound randomly. This can be used for either integer or float genes. The Non-Uniform operator increase the probability of mutation that will go to 0 with the next generation, which keeps the population from stagnating in the early stages of the evolution. The Uniform operator replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for the gene. This mutation operator can only be used for integer and floating point genes. The Gaussian operator adds a unit Gaussian distribution random value to the chosen gene. If it falls outside of the user-specified lower or upper bounds for that gene, the new gene value is clipped. The mutation operator can only be used for integer and floating point genes

The TSP Mutation Method Consists of the following processes. The mutation method should only be capable of shuffling the route, as adding or removing locations from the route would risk creating an invalid solution. We utilize simple swap mutation, as such. Because swap mutation only swaps pre existing values, it will never create a list which contains missing or duplicate values compared to the original. We initially used a low mutation rate of 0.015 to provide the best, consistent results, but experimented with varying values.

**Genetic Algorithm Implementation (Java) and 2-Point/Ordered Crossover**

We began our approach of creating a simulated world for our TSP problem. This includes various setup Java classes such as *City.java*, *Tour.java*, and a *population.java.* These classes had helpful methods which includes adding cities, getting total number of cities, calculating total tour distance, and our error checking method of determining if a city is already present in a given tour. With our city class and respective population class created, we can begin creating our city grids and forming a general population of candidate solutions to the TSP. Our GA methods are within our *GA.java* class, including our evolvePopulation, crossover, mutate and fittest selection methods.

Our evolve Population method loops over the populations set size, selecting two parent individuals at random, then creates a child individual by crossbreeding them with a given crossover method. Our swap mutation method simply generates a random number, checks to see if it is less than our set mutation rate, and performs the swap mutation for two randomly chosen indices within the given tour. Information on our crossover method will follow in the next paragraph.

Our approach to a 2-Point Crossover technique is as follows. Begin by substring selections; cutting a single substring of varying size from one parent. Next, substring exchange; exchange selected substring to produce proto-child. Then, exchange remaining cities from second parent; iterate through second parent’s tour, moving cities currently not existent in the offspring into the offspring, creating our valid child tours. Example: Let P1: 1 2 5 6 4 3 8 7 | P2: 1 4 2 3 6 5 7 8. Suppose the randomly chosen cut points are as follows: P1: 1 2 │ 5 6 4 │ 3 8 7. The proto child will then be: 1 2 │ 5 6 4 │ 3 7 8 respectfully. As our method iterates through our second parents tour list, it places 1 into our offspring. Once it gets to the second parents second index, it detects that the city 4 is already placed in the offspring. It move on to the next index in the parent, and repeats the process for all available locations within our offspring tour.

As stated in our presentation slides, we opted for a basic swap mutation for our mutation algorithm. This simply implies two locations in the route are selected at random, then their positions are simply swapped. We started with a rather small and minute mutation rate, 0.015. This gave us a good indication of how our crossover method provided most of the variation in making more fit offspring, but that a small mutation variable still existed and created minor changes. When we increased that value, to 0.10, we found similar consistent results, with no significant variation in optimal tour distance after 20 runs of population size 1000. However, when we increased mutation rate again to 0.25, we found the optimal distance drop over these 20 runs. We can only assume that mutation rate was too high, and caused too many unnecessary swaps in our population, gradually reducing the GA’s effectiveness in finding the most optimal solution.

One consideration we had while constructing our GA was the implementation of an Elitism feature. We knew that when creating a new population by crossover and mutation, there is a rather big chance what we would lose the best individual (most optimal solution). If Elitism is enabled (On by def.), it copies the best individual from the current population using our getFittest method, and adds to the new population by default. Elitism can very rapidly increase performance of our GA, because it prevents the loss of the potentially best solution. We choose to only pull one best solution from existing population, instead of opting for an Elitism rate that would vary inversely with our population size. (631 words, continuing 2mrw…)

**Partially-Matched Crossover**

Lorem ipsum…

Partially-matched crossover is a recombination technique inspired by 2-point crossover, with chromosome order-preserving characteristic that enables us to apply it to the Travelling Salesman Problem. PMX preserves chromosome order by employing a ‘legalization’ process that observes existing mapping relations and preserves them in the offspring. This mapping step has earned PMX the alternative title ‘partially-mapped crossover’.

First, a random substring of the parent chromosomes is selected and swapped. Then, mapping relationships are determined by observing the position-wise swaps that occurred during the first step. Lastly, these mapping relationships are used in the ‘legalization’ process to protect chromosome order.

Legalization involves iterating through the alleles in each candidate solution that were *not* inside the selected substring, and adjusting them according to the relationship mappings. Without this legalization step, the offspring solutions would fall victim to a two-part issue preventing them from solving the TSP; they would ignore certain cities (graph nodes) in favor of revisiting others. These problems manifest as graph cycles, with candidate solutions repeating certain nodes. Since the length of the candidate solution is equal to the number of nodes in the graph (by definition of TSP), these non-legalized solutions can not possibly solve the problem.

**Results & Analysis**

For the comparative analysis, we leveraged TSP boilerplate from a well-known evolutionary algorithms package called DEAP, Distributed Evolutionary Algorithms in Python. Using Jupyter Notebooks to iteratively build experiments, we were able to run and compare trials of PMX and OX methods with varying mutation rates. Descriptive statistics and the matplotlib graphic library were used to visualize results and aid in analysis.

Gauging the performance of both crossover methods required a bit of creativity. Each simulation ran for a predetermined number of generations, and prior-knowledge of the optimal tour for the 17, 24 and 120 city problem sets allowed us to compare how quickly—if at all—a particular instance of a genetic algorithm was able to converge upon the true solution.

|  |  |
| --- | --- |
| 17cityOrderedCx1_5 | 17cityPartialCx1_5 |
| 17cityOrderedCx10_0 | 17cityPartialCx10_0 |
| 17cityOrderedCx25_0 | 17cityPartialCx25_0 |

Results for 17-city example input, gr17.json, after 40 generations

The red line parallel to the x axis indicates the best possible solution for the graph input, and an ideally-performant genetic operator reaches this line in the shortest amount of time [e.g. number of generations] it can manage. Since chromosome selection methods often rely on random number generators, the overall behavior of the algorithm is of greater concern than the final score output by the best candidate solution of the *n*th generation. Therefore, analysis will focus heavily on convergence to a ‘neighborhood’, where the derivative of the ‘min’ is approximately 0.

The independent variable, generation, is plotted along the x-axis, with the y-axis representing the range of scores achieved by the candidate solutions belonging to a given generation.This ‘score’ performance metric is the total distance travelled for a given candidate solution tour, and the best solution minimizes this score as much as possible.

Over the course of ‘evolution’, each instance of the genetic algorithm trends down towards the optimal, minimized total tour length value. However, brief inspection of the results reveals a stark contrast between the two crossover methods’ performance. This marked difference between each crossover method’s performance is consistent across the range of mutation rate values, with higher mutation rates reliably preventing the algorithm from getting closer to the ideal solution in as few generations as trials with lower mutation rates.

|  |  |
| --- | --- |
| comparison1_5 | comparison10 |

Ordered crossover (yellow) outperforms partially-matched (red)

Plotting the range of fitness/evaluation function scores for OX (yellow area) against PMX (red area) reveals ordered crossover consistently outperforming partially-matched crossover throughout the course of evolution. This trend appears visibly in the left-hand chart under a 1.5% mutation rate, and less-pronounced in the right-hand chart under a 10% mutation rate.

The 1.5% mutation rate results demonstrate another interesting trend; a sudden reconciliation of PMX’s candidate solutions from the 23rd through the 25th generations. With the worst-performing solutions suddenly performing much better, we look to standard deviation to confirm the underlying behavior.

|  |  |
| --- | --- |
| std1_5 | std10 |

Lo and behold, the standard deviation graph reveals a sudden, sharp decrease shortly after the 20th generation. Since standard deviation is a measure of the 'spreadness' of data, this sharp decrease suggests the various candidate solutions generated by PMX suddenly begin to value homogeneity, producing offspring very similar to one another [performance-wise].

Considering how PMX works, with mapping-relationships inside the selected chromosome substrings informing the ‘legalization’ process of offspring (maintaining ordered chromosomes is a requirement for solving TSP), we predict this sudden improvement is due to a series of two or three wildly-successful mutations [due to ‘legalization’].

One interesting takeaway from this observation is that, although OX originally outperformed PMX and maintained a better ‘top score’ than PMX throughout the course of evolution, there came a time (generation #20 and beyond) where any random candidate solution selected from the PMX population likely performed better than any random candidate solution from OX. Although standard logistical applications of the TSP (routing for a single delivery person) would likely choose the single ‘best’ solution after the *n*th generation, certain uncommon applications in dynamic systems, swarm intelligence and ensemble learning might require multiple different solutions that all perform ‘well enough’.

While mutation provides chance by adding alleles previously lost from the population, crossover provides the fundamental “convergence operation which is intended to pull the population towards a local minimum/maximum”.[[2]](#footnote-1) Our results support the notion that different crossover methods exhibit different strategic behaviors with respect to how their offspring are ‘raised’, with implications for each real-world application of TSP requiring solution by GA. Future research efforts will focus on exploring ideal crossover strategies for these different applications of TSP, including computer wiring, vehicle routing and job scheduling.[[3]](#footnote-2)

1. http://www.csd.uoc.gr/~hy583/papers/ch11.pdf [↑](#footnote-ref-0)
2. Wilson, Lucas A. "What Is the Role of Mutation and Crossover Probability in Genetic Algorithms?" Technical Q/A forum. *What Is the Role of Mutation and Crossover Probability in Genetic Algorithms?* ResearchGate, 1 Oct. 2014. Web. 4 May 2016. <https://www.researchgate.net/post/What\_is\_the\_role\_of\_mutation\_and\_crossover\_probability\_in\_Genetic\_algorithms>. [↑](#footnote-ref-1)
3. Lenstra, J. K., and A H G. Rinnooy Kan. "Some Simple Applications of the Travelling Salesman Problem." *Operational Research Quarterly* I 26.4 (1975): n. pag. Web. <http://oai.cwi.nl/oai/asset/21825/21825A.pdf>. [↑](#footnote-ref-2)