Solving Large Travelling Salesman Problems with Small Populations

Rong Yang
Department of Computer Science, University of Bristol, Bristol BS8 1UB, U.K.

Abstract

A new genetic algorithm for the solution of the travelling salesman problem is presented in this paper. The approach is to introduce several knowledge-augmented genetic operators which guide the genetic algorithm more directly towards better quality of the population but are not trapped in local optima prematurely. The algorithm applies a greedy crossover and two advanced mutation operations based on the 2-opt and 3-opt heuristics. One of our particular interests is to investigate whether small populations are adequate for solving large problems. We also want to see how the quality of the initial population and the quality of the final solution are related, especially when the population is small. For this purpose, we designed a selective initialization scheme to generate a better initial population. The algorithm has been implemented in C and tested on several sets of data. The largest data instance used is 2392 cities (i.e., pr2392). The actual population size used is only 32. For small sets of data (less than 100 cities), our algorithm can find the optimal solutions. For large data sets, the quality of the best solutions found is about 1.3-2.3% worse than optimal. The empirical results show that combining the knowledge from the heuristic method and the genetic algorithm is a good approach for solving the large travelling salesman problem. By incorporating the heuristic method, we can develop greedy genetic operators to improve the efficiency of genetic algorithms. Moreover, it makes small population sizes sufficient to solve large problems. By incorporating the genetic algorithm technique, we can escape from local optima in many cases so that much better results can be obtained than with heuristic methods alone.

1 Introduction

The travelling salesman problem (TSP) is the task of finding the shortest possible tour through a given set of cities. It is a typical optimization problem having many practical applications such as routing, scheduling, wiring, etc. In the standard TSP, the distance from city *i* to city *j* is the same as the distance from city *j* to city *i*. There is also the asymmetric version of this problem and some other variations, which can be theoretically transformed to the standard version [11]. In this work, we concentrate on the standard TSP.

There are two alternative approaches to solving the problem. One is to find the solution and prove its optimality. This involves a huge amount of computation because the TSP is NP-complete. For example, the optimal solution of the 3038-cities instance was proved a few years ago by using one of the integer programming techniques, branch and cut. The program required one and a half years of computer time [8]. Another approach is to find an approximate solution within a reasonably short time. The study of algorithms to achieve this practical goal has been carried out by applying many different methods from many areas such as heuristic methods [2, 7], simulated annealing [1, 5], tabu search [6], neural networks [9], and genetic algorithms [4, 14, 13].

In this paper, we present a knowledge-augmented genetic algorithm to solve the TSP. It combines heuristic techniques with genetic algorithms. The mechanics of genetic algorithms are used as the basic framework, and ideas from heuristic methods are used to design the genetic operators. The results show that the algorithm is competitive with conventional heuristic approaches.

The rest of the paper is organized as follows. The next section introduces the knowledge-augmented operators used in our algorithm. Section 3 gives an outline of the algorithm. Section 4 presents the results of experiments. The final section contains some concluding remarks.

2 Design of knowledge-augmented genetic operators for TSP

The basic genetic operators are *reproduction*, *crossover*, and *mutation*. Our reproduction operator is a standard method called *tournament selection*, and is therefore not discussed here. We only describe non-standard operators: the crossover and mutation operators, which both use domain-specific knowledge. We also introduce a special operator which generates an initial population in a systematic way instead of by a purely random selection.

In our algorithm, a *normalized path representation* is used to represent a tour. That is, a list (1,2,3,4) means the tour going from city 1 to city 2, 3, 4, and back to city 1. With the traditional path representation, different lists such as (4,1,2,3), (3,4,1,2), and (2,3,4,1) all represent the same tour as (1,2,3,4). To improve coding efficiency, we normalize the path list so that the first city in the list is always 1.

2.1 Selective initialization

Traditionally the initial population is generated by a purely random selection procedure. Thus, a reasonably large population is needed to cover the solution space. In this work, we aimed to investigate an interesting issue raised by C. Reeves [10]: that is, whether we can use very small populations for some applications.

For large TSP applications, if we follow the common practice of having a size between n and $2n$ where n is the tour length, we would need extremely large populations. This would require huge amounts of computation time. Our solution is to generate a small but representative initial population. We use an idea summarized in Reinelt's book [11]: most of the possible edges in a complete graph will not occur in short tours because they are too long. It is reasonable to concentrate on a much smaller subgraph instead of exploiting the whole graph.

A **k-nearest-neighbour subgraph** is defined as a graph containing all edges $<c_i, c_j>$ where c_i is among the k nearest neighbours of c_j, or c_j is among the k nearest neighbours of c_i.

In our selective initialization, we give a higher priority to edges belonging to the k-nearest-neighbour subgraph. That is, from a city c, we first try to randomly select a next city from c's k nearest neighbours. If all cities of c's k nearest neighbours have already been used, then we make a random choice from all unused cities.

A list of k nearest neighbours for each city is generated in advance. We have relaxed the definition of the k-nearest-neighbour subgraph to a looser but more practical one. That is, if c's k+1 nearest neighbour is as near, or almost as near, as the k nearest neighbour, we include it in the k-nearest-neighbour list. Then we carry on performing the same checking on k+2 until a "large" distance increment is found.

In Section 4.3 we compare the results obtained by using this selective initialization and the random initialization.

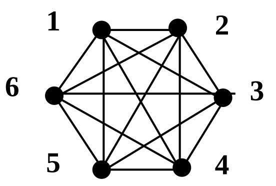
2.2 Greedy crossover

In the design of the crossover operator, we first absorbed ideas from two previous studies. One was by Grefenstetts *et al*. [3], who used a heuristic crossover which constructs an offspring by choosing the better of two parental edges. Another was by Starkweather *et al*., who designed the edge recombination operator [14]. One of their findings was that it is very important to preserve common edges between the two parents [13]. After studying the above work, we developed the following crossover operator, which is different from the previous two algorithms but combines the good ideas from both.

Given two parent tours in the normalized path representation, **P1** and **P2**, the first offspring is constructed as follows: start with a random city c, then check whether either the edge leading to c or from c (i.e., the edge $<c, c_{right}>$ or $<c_{left}, c>$) is used in both **P1** and **P2**. If so, the common edge is chosen. Otherwise, we compare c's right side edge in each of **P1** and **P2**. The shorter one is chosen, unless it would introduce a cycle, in which case the longer one is chosen. If the longer one would also introduce a cycle, then we extend the tour by a carefully selected edge (details later). The second offspring will be constructed in a similar way, but we compare c's two left side edges instead of its right side edges.

Distance Table

1	0					
2	10	0				
3	17	10	0			
4	20	17	10	0		
5	17	20	17	10	0	
6	10	17	20	17	10	0
cities	1	2	3	4	5	6



parent 1: 1 3 4 2 5 6
parent 2: 1 2 3 6 5 4

Offspring 1: (if randomly starting from city 6)

6 5 4 2 3 1 $\longrightarrow$ **1 3 2 4 5 6**

Offspring 2: (if randomly starting from city 3)

3 2 1 6 5 4 $\longrightarrow$ **1 6 5 4 3 2**

Figure 1: An example of greedy crossover

Figure 1 shows a simple example of our crossover operation. Assume that we have two parents as shown in Figure 1. To generate the first offspring, we randomly start from city 6. The two right side edges are <6,1> in **P1** and <6,5> in **P2**. The distance table shows that <6,1> and <6,5> have exactly the same length. In this situation, the edge in **P1** would normally be selected first for offspring 1; the edge in **P2** for offspring 2. However, in our example, edge <6,5> is a common edge between the two parents, so <6,5> is chosen. Next, from city 5, we have two right side edges: <5,6> and <5,4>. This time, we choose <5,4> because the common edge <5,6> would create a cycle. From city 4, the edges are <4,2> and <4,1>: the shorter one <4,2> is chosen. Then from city 2, there are <2,5> and <2,3>: the shorter one <2,3> is chosen. Finally, we only have city 1 left, which has to be the last one in the tour. The second offspring is generated starting from a different random city, 3. This time, the left edges are compared. Luckily, all of the best edges in **P1** and **P2** have been inherited by offspring 2.

The criterion of selecting a new edge to prevent a cycle is based on the same idea as used in our selective initialization. That is, edges not belonging to the k-nearest-neighbour subgraph are most likely to be excluded from the optimal tour. Thus, we try to select from the k-nearest-neighbour list first; only if they are all used do we select randomly from the other, longer, edges. The actual selection rule implemented is a bit more sophisticated in how to choose from the k-nearest-neighbour list. Assuming that c is the current city, we examine all unused cities in c's k-nearest-neighbour list, then choose the one which has the fewest k-nearest-neighbour cities available. The idea behind this is that the fewer k-nearest-neighbours c has, the more likely it is to become an isolated city in the k-nearest-neighbour subgraph. So we should choose it, to avoid the danger of using a non-k-nearest-neighbour city.

We have done experimental work to assess the difference between using our rule and the random rule to extend a new edge. Note that the random rule has been applied in both [3] and [14]. Detailed results will be discussed in Section 4.3.

2.3 Mutations based on 2-opt and 3-opt heuristics

Mutation has been regarded as an important operator in increasing the genetic diversity of populations. In particular, as we only intend to use small populations, it is very important to have good mutations to prevent premature convergence on local optima.

We designed two mutations, which are based on 2-opt and 3-opt heuristics, respectively. Both 2-opt and 3-opt heuristics have been deeply studied in the area of heuristic search to solve the TSP, producing impressive solutions [11].

2.3.1 2-opt mutation

The 2-opt heuristic has been combined with genetic algorithms as a specialized genetic operator by Homaifar *et al*. [4]. In their algorithm, a limited segment length N is imposed. The 2-opt exchange is then tried on all pairs of edges which can fall within a given segment of length N. The actual exchange is performed as long as it can improve the cost of the tour; otherwise the tour is unchanged. Many 2-opt changes might occur in one operation, as the operation continues until all possible pairs have been tried. On the other hand, there is no exchange at all if the 2-opt heuristic does not work within segments of length N.

In our implementation, the 2-opt heuristic is applied in a different way. First, we want to introduce a random factor to the operator. That is, if it is applied to two identical tours, we prefer to have two different results than two identical ones. Second, we favour traditional mutation: that is, to make exactly one exchange in one operation. Our final consideration is that a good 2-opt exchange does not necessarily fall within a limited segment. Therefore our 2-opt mutation is designed as follows.

We randomly select a number of edges. For every random edge, we examine its 2-opt exchange with all the other edges in the tour. Finally, we select the best exchange among all examined pairs. The exchange is performed whether or not it makes an improvement. The total number of random edges selected is an adjustable parameter, called max_try, which is currently set to 5.

2.3.2 3-opt mutation

According to experimental results from the area of heuristic methods, the 3-opt heuristic is a more powerful and flexible method than the 2-opt heuristic. Generally it produces better solutions, though the computation cost is much higher.

In our early experiments, we found that using the above 2-opt mutation and our greedy crossover can find optimal solutions for small data sets. However, when we moved to large data instances, it produced solutions of 4-8% worse than optimal, on average. A further improvement is certainly desirable. In Homaifar's work [4], which combined the 2-opt heuristic, the largest data set studied was the 318-city problem. Therefore, it is not certain if their approach can scale up. We believed that we needed a more powerful mutation than simple 2-opt exchange, so we developed another mutation operator based on the so-called 3-opt move.

A 3-opt move is to remove 3 existing edges and add 3 new edges. Let us name the 3 removing edges r_1, r_2, and r_3 and name the 3 replacing new edges n_1, n_2, and n_3. The following steps describe how our 3-opt mutation determines these removed edges and new edges. A simple example is also illustrated in Figure 2.

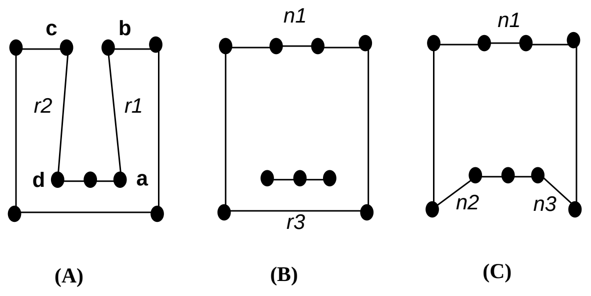


Figure 2: An example of 3-opt mutation

Step 1: The first removed edge r_1 is selected randomly. In the example, we assume r_1 = <a,b> is chosen (Figure 2 (A)).

Step 2: From city b (or a), we randomly select a neighbour from the k-nearest-neighbour subgraph. Assume that we select c. Then, the edge r_2 = <c,d> is determined to be the second removed edge since d is the city leading to city c if travelling in the direction of <b,a>. Meanwhile, the edge n_1 = <b,c> is determined to be the newly added edge (Figure 2 (B)). Note that, after adding n_1 and removing r_1 and r_2, the original tour has been separated into a cycle and a segment.

Step 3: Now we need to decide how to insert the segment from a to d into the newly created cycle. We try all the possibilities of insertion and select one with the smallest cost. Figure 2 (C) shows the final tour after the 3-opt mutation where the edge r_3 is the inserting point: two new edges, n_2 and n_3, are added.

3 Outline of the algorithm

The following is a summary of our algorithm:

Step 1: Initializing the population by using the k-nearest-neighbour selective method described in Section 2.1. k is set to 10 for data sets which contain less than 600 cities and to 20 for larger data sets.

Step 2: Taking all tours pair by pair from the population and performing either crossover or mutation. We first check whether the pair of tours are identical. If not, the greedy crossover described in Section 2.2 is applied. Otherwise, one of the tours does a 2-opt mutation and the other does a 3-opt mutation.

Step 3: Reproducing the new generation by using tournament selection: randomly pick two tours from the population and let the cheaper one move to the next generation. An elitist selection rule is also applied. That is, if the best tour in the new generation is worse than the previous one, we preserve the previous best tour by passing it to the new generation.

Step 4: If the best tour so far has not been improved for more than N generations, stop the computation; else go to step 2. N is currently set to 1000.

Note that, in the above algorithm, mutation is not performed regularly as in conventional genetic algorithms. It is performed when we incidentally select two identical tours for crossover. As we only use small populations, the current best tour can be spread quickly. Therefore it frequently happens that two identical tours are picked. Instead of selecting a different tour, we feel that it is better to do mutation because, in this way, the heuristic-based mutations can be concentrated on improving better tours rather than whole populations.

4 Empirical results

We have implemented our algorithm in C and tested it on the following six sets of data: gr17, dan42, st70, pa561, pr1002, and pr2392, which are all from the TSPLIB [12]. For the first small sets of data, our algorithm can find all optimal solutions in less than a minute. For the other larger sets of data, we can reach reasonably good quality solutions. In the following subsections, we give the detailed empirical results for the 3 large sets of data. The machine used for testing is a Sun UltraSPARC.

4.1 Overall performance

We tested 10 runs (using different seeds) on each of the 3 sets of data. The best results from each data are summarized in Figure 3. The first two rows in Figure 3 show the computation effort, and the next 3 rows show how good the solution is. The figure shown in the *quality* row is the result of calculating

$$\left(\frac{cost\ of\ solution\ found}{known\ optimum\ cost} - 1\right) * 100$$

This indicates by what percentage the solution exceeds the optimal solution.

DATA SETS	pa561	pr1002	pr2392
total generations needed	3024	5063	9659
time spent (minutes)	1.6	29.5	130
best solution found by us	2800	265096	386856
known optimum	2763	259045	378032
quality of our solution	1.34	2.33	2.33

Figure 3: Overall performance of our algorithm

The next table (Figure 4) gives the quality achieved by heuristic methods, which have been widely regarded as a highly practical approach to solving the TSP. We selected the best result from several variants of 2-opt, 3-opt and Lin-Kernighan heuristic algorithms, published in [11]. The result of the 2-opt heuristic is actually from an algorithm which combines 2-opt exchange and node insertion; the quality would be much lower using 2-opt alone. As we can see, the results of our algorithm are well above the 2-opt and 3-opt heuristic methods, and are competitive with the Lin-Kernighan method.

DATA SETS	pa561	pr1002	pr2392
2-opt + node insertion	NA	5.17	6.36
3-opt	NA	3.09	3.35
Lin-Kernighan	NA	1.17	2.02

Figure 4: The best quality achieved by heuristic methods

4.2 Stability analysis

DATA SETS	pa561	pr1002	pr2392
minimum	1.34	2.33	2.33
maximum	2.84	4.41	4.27
average	2.06	3.32	3.53
span	1.50	2.08	1.94
deviation	0.44	0.54	0.60

Figure 5: Stability analysis of our algorithm

Another important measure is the stability of the algorithm. The table in Figure 5 shows the stability analysis of our algorithm. The results are taken from 10 random runs for each set of data. Each column corresponds to one data instance and gives the best quality, the worst quality, the average quality, the span between best and worst, and the standard deviation. The results show that our algorithm is very stable. The span values are much smaller than the results from heuristic methods, published in [11]. Moreover, the standard deviations are very small, so we can expect to achieve average performance easily.

4.3 Results of not using k-nearest-neighbour knowledge

There are two places in our algorithm where we rely heavily on the k-nearest-neighbour knowledge:

- Scheme 1: in initialization, we first select random edges from the k-nearest-neighbour subgraph;
- Scheme 2: in crossover, when selecting a new city to prevent cycles, we give higher priority to k-nearest-neighbours.

To verify how the k-nearest-neighbour knowledge works, we produced four versions of the algorithm.

version 1: uses neither scheme 1 or 2;

version 2: uses scheme 2 but not scheme 1;

version 3: uses scheme 1 but not scheme 2;

version 4: uses both scheme 1 and 2 (i.e., the original version).

Figure 6 shows the results from the above four versions. The data instance used is pa561. The x axis represents generations; the y axis represents the cost of the best tour found. Thus, the four curves in the graph show how the population best improves through the generations. The initial cost of versions 1 and 2 (which do not use selective initialization) is extremely large, around 35000, while the initial cost produced by the versions using selective initialization is only about 7000.

It is clear that version 4 produces the best result. It is interesting to point out that, despite not using selective initialization, versions 1 and 2 can quickly improve their population best from about 35000 to just over 3000 within the first 100 generations. This shows that our greedy crossover is very efficient in improving poor quality tours. We also notice that version 3, which used selective initialization but randomly selected long edges in crossover, is trapped in a local optimum much earlier than the other versions.

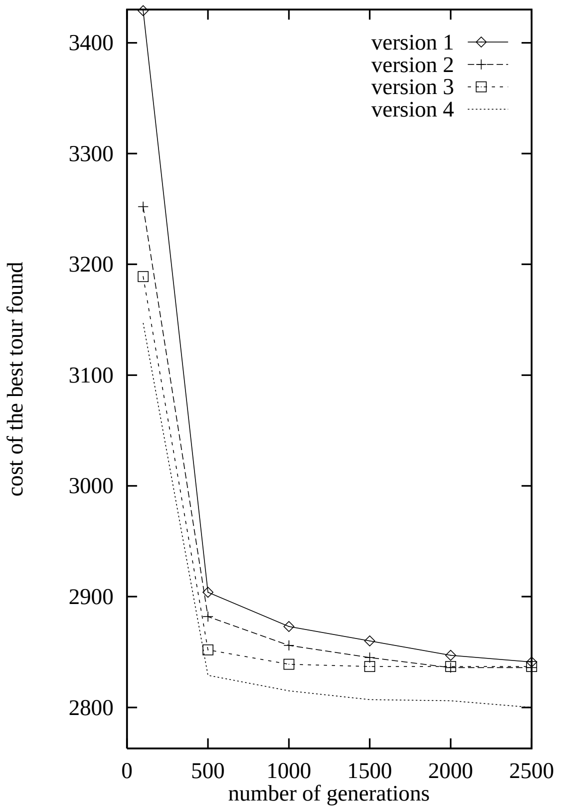


Figure 6: Using k-nearest-neighbour knowledge

4.4 Results on varying mutation scheme

The last experiment we did is to compare the effectiveness of the 2-opt and 3-opt mutations. Our algorithm combines both 2-opt and 3-opt mutations; what if we only used one of them? Figure 7 shows the difference in performance between using 2-opt or 3-opt mutation alone and combining both.

It shows that using 2-opt mutation alone produced the poorest result of the three. Using 3-opt mutation alone achieved better quality than using 2-opt mutation, but more computation time is required. When we combine both, a much better quality is achieved without an increase in computation time.

DATA SETS	pa561	pr1002	pr2392
Only use 2-opt mutation			
time spent (minutes)	15	17	71
quality achieved	3.22	5.73	4.72
Only use 3-opt mutation			
time spent (minutes)	19	25.7	150
quality achieved	2.57	2.79	4.52
Use both 2-opt and 3-opt mutations			
time spent (minutes)	1.6	29.5	130
quality achieved	1.34	2.33	2.33

Figure 7: Results of testing 2-opt and 3-opt mutations

5 Conclusion

In this work, we designed and developed a new genetic algorithm for the TSP. The algorithm applies a greedy crossover and two advanced mutation operations based on the 2-opt and 3-opt heuristics. A selective initialization operator is also proposed. The algorithm can find solutions 1.3-2.3% worse than optimal. The results show that combining the knowledge from heuristic methods and genetic algorithms is a promising approach for solving the large TSP.

Two concluding remarks are as follows.

- From the point of view of genetic algorithms, by incorporating the heuristic method, we can develop greedy genetic operators to improve the efficiency of genetic algorithms. Moreover, it makes small population sizes sufficient to solve large problems.
- From the point of view of heuristic methods, by incorporating the genetic algorithms technique, we can escape from local optima in many cases, so that much better results can be obtained than by using heuristic methods alone. We can also achieve very high stability.

There are two interesting directions for future work. One is to further improve the algorithm by introducing a simplified form of the Lin-Kernighan heuristic. We have already experienced a great improvement by adding the mutation based on the 3-opt move. As the Lin-Kernighan heuristic is much more powerful than the 3-opt heuristic, we can expect to achieve a further improvement in the quality of the solution. Another direction is to parallelize the algorithm. We would like to investigate whether the algorithm can be vectorized efficiently. We are also interested in modifying the algorithm to a coarse-grained parallel genetic algorithm.

Acknowledgements

The work is supported by the Information-technology Promotion Agency (Japan) through an international research collaboration. The author is very grateful to her sponsors in Mitsubishi Research Institute. Many thanks also go to Steve Gregory for his comments on the paper.

References

[1] E. H. L. Aarts and J Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley and Sons, 1989.

[2] J. L. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, (4):387–411, 1992.

[3] J. J. Grefenstetts, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic Algorithms for the Traveling Salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 160–168, 1985.

[4] A. Homaifar, S. Guan, and G. E. Liepins. A New Approach on the Traveling Salesman Problem by Genetic Algorithms. In *Proceedings of the Fifth International Conference on Genetic Alorithms*, pages 460–466, 1993.

[5] D. S. Johnson. Local Optimization and the Traveling Saleman Problem. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming*, pages 446–461, Springer-Verlag, Lecture Notes in Computer Science 442, 1990.

[6] J. Knox. *The Application of TABU Search to the Symmetric Traveling Salesman Problems*. PhD dissertation, University of Colorado, 1989.

[7] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for Traveling Salesman Problem. *Operation Research*, (21):498–516, 1973.

[8] CRPC Newsletter. *World-Record Traveling Salesman Problem for 3038 Cities Solved*. Web Page, http://www.crpc.rice.edu/CRPC/newsletters/jan93/news.tsp.html, 1993.

[9] J. Y. Potvin. The Traveling Salesman Problem: A Neural Network Perspective. *ORSA Journal on Computing*, (5):328–348, 1993.

[10] C. R. Reeves. Using Genetic Algorithms with Small Populations. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 92–99, 1993.

[11] Gerhard Reinelt. *The Traveling Salesman*. Springer-Verlag, 1994.

[12] Gerhard Reinelt. *TSPLIB*. Web Page, http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html, 1991.

[13] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A Comparison of Genetic Sequencing Operators. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76, 1991.

[14] D. Whitley, T. Starkweather, and D. Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, 1990.

This paper appears in the Proceedings of GALESIA97, Genetic ALgorithms in Engineering Systems: Innovations and Applications, Sept 1997