Perturbation Method For Probabilistic Search For The Traveling Salesperson Problem

James P. Cohoon, John E. Karro, Worthy N. Martin, William D. Niebel[1]
University of Virginia, Department of Computer Science
Charlottesville, VA 22903 USA

Klaus Nagel[2]
Siemens AG, Corporate Department Technology
Munich, D-81730, Germany

Abstract

The Traveling Salesperson Problem (TSP), is an NP-complete combinatorial optimization problem of substantial importance in many scheduling applications. Here we show the viability of SPAN, a hybrid approach to solving the TSP that incorporates a perturbation method applied to a classic heuristic in the overall context of a probabilistic search control strategy. In particular, the heuristic for the TSP is based on the minimal spanning tree of the city locations, the perturbation method is a simple modification of the city locations, and the control strategy is a genetic algorithm (GA). The crucial concept here is that the perturbation of the problem (since the city locations specify the problem instance) allows variant solutions (to the perturbed problem) to be generated by the heuristic and applied to the original problem, thus providing the GA with capabilities for both exploration and exploitation in its search process. We demonstrate that SPAN outperforms, with regard to solution quality, one of the best GA systems reported in the literature.

Keywords: Genetic algorithms, Traveling salesperson problem, probabilistic search, minimum spanning tree.

1. Introduction

The Traveling Salesperson Problem (TSP), a classic combinatorial optimization problem, is generally described as follows. The input is a set of *n* cities and a salesperson. The salesperson starts from one of the *n* cities and proceeds to each of the other cities, then completes the tour with a return to the original city. The goal is to find a tour for the salesperson that minimizes the total distance traveled. The problem clearly has practical application.

For our purposes, the input to TSP is a complete graph (*V,E*) whose vertices (cities) are embedded in the Euclidean plane. The fact that we have Euclidean distances, and thus the triangle inequality, implies that it is not worthwhile to visit any intermediary city more than once. Hence, a solution to TSP is essentially a permutation of the cities.

TSP is NP-complete [3]. Thus it is unlikely that there is a polynomial time solution for the problem. TSP researchers have concentrated on developing exact algorithms for special cases [4], heuristics [12], randomized algorithms [16], and probabilistic search [5, 9].

In subsequent sections, we consider a simple but effective TSP heuristic, provide an overview of the genetic algorithm paradigm, present the details of our genetic algorithm SPAN, compare the SPAN method with a state-of-the-art genetic algorithm, and draw conclusions.

For a detailed history of the TSP see the monograph of Lawler, Lenstra, Rinnooy Kan, and Shmoys [11]. For an examination of TSP genetic algorithms see Potvin's survey [14].

2. A simple heuristic with bounded performance

Although the primary area of investigation that we consider here is probabilistic search and in particular, a genetic algorithm, it is first necessary to consider one of the classic effective heuristics that uses a minimum spanning tree as a basis for generating a tour [18].

Heuristic H(*V, E*) {
    Construct a minimum spanning tree *T* for graph (*V, E*)
    Perform a depth-first traversal of *T* to obtain *T'*, a tour with vertex repetition

1. Correspondence: Email: cohoon@virginia.edu, karro@virginia.edu, wnm@virginia.edu, niebel@cs.virginia.edu; WWW http://www.cs.virginia.edu.
2. Correspondence: Email klaus.nagel@zfe.siemens.de; WWW http://w1.siemens.de.

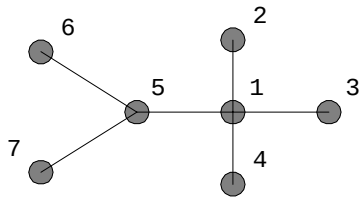
Remove subpaths with repeated vertices to yield a tour T''
return T''
}

A *minimum spanning tree* (MST) for a graph is a tree of minimum total length for which the vertex set is the same as that of the original graph and the edge set is a subset of the original graph [10, 15]. Because an MST is a relaxation of the TSP spanning requirements, the length of a MST can be no greater than the length of an optimal Traveling Salesperson Tour (TST). The depth-first transversal of T yields T', a "tour with vertex repetition", by including each edge of the MST twice. Thus, the length of T' is at most twice the optimal TST length. Removing repeated vertices from T' to yield a tour T'' removes all repeated edges and adds new edges. However, the length of any new edge is no longer than the subpath that it replaced, thus the length of T'' is no more than twice that of the optimal TST.

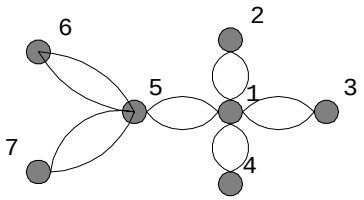
The claim that the length of T'' is no greater than that of T' follows simply from the triangle inequality, since the reduction always replaces a subpath in T' by a single new edge. In the reduction a subpath $p_{st}=v_s r_1 r_2 ... r_n v_t$ is replaced by a single edge (v_s,v_t). Since the vertices are in the Euclidean plane, the triangle inequality assures that the length of the added edge is no greater than that of the subpath.

The MST for a set of seven vertices is given in Figure 1(a). In this figure, the vertices are labeled according to a depth-first transversal [1], where the starting vertex was arbitrarily selected. The tour with vertex repetition derived from the MST is given in Figure 1(b). The repetition-free derived TST is given in Figure 1(c). Observe that this repetition-free tour is nonoptimal.

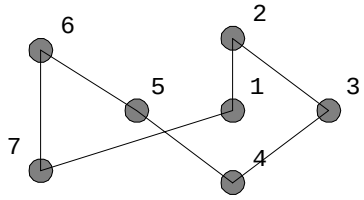
Figure 1. MST and derived TSTS.



(a) MST



(b) TST with repeated vertices



(c) Repetition-free TST

An analysis of heuristic H indicates that it can be performed in $O(n^2)$ time. The dominant step is generating the MST, which can be done in $O(n^2)$ [10, 15]. The other two steps can be done in linear time $O(n)$ [1, 11]. The subpath removal can in fact be done simultaneously while doing the depth-first traversal by not generating back-edges to already visited vertices and instead generating an edge to the next newly visited vertex.

We shall use heuristic H in the course of our genetic algorithm.

3. Genetic Algorithm

Genetic algorithms (GAs) were first proposed by Holland as a technique for adaptive learning [8]. The method has since then been more widely applied. In particular, there have been many GAs for TSP [2, 6, 7, 13, 19, 20, 21].

A genetic algorithm iteratively manipulates a population of strings, where a *string* is a representation of a problem solution. The initial population is a collection of random solutions. Associated with each string is a measure of its *fitness*. (When using a fitness measure to compare two solutions, the better solution has higher fitness.) As TSP is a combinatorial minimization problem, a common method for associating a fitness with a TSP solution π is to use the value *MaxPopScore* - *cost*(π),

where *MaxPopScore* is the length of the worst tour in the current population and *cost*() is a function that returns the length of its tour parameter.

There are two operators for manipulating strings: the *crossover* and *mutation* operators. The crossover operator takes two strings (parents) to produce a new string (offspring). Parent selection is done probabilistically, with strings with higher fitness more likely to be selected than strings with lower fitness. The standard method for associating a probability with a fitness is f/F, where F is the sum of fitnesses for the strings in the population.

The mutation operator modifies a string. The modification is typically applied to a small component of the string. For example, for a string representing a TST, two adjacent cities in the TST have their tour locations swapped. The selection of which strings to mutate is done randomly without regard to fitness, i.e., with uniform distribution.

3.1 A basic GA

A basic genetic algorithm is given in Listing 1. The algorithm indicates that the method iterates for N_G generations. For each iteration, N_C crossovers and N_M mutations are performed. Function *choose*() is the string selection function. Its first parameter is the set from which the choice is to be made; its second parameter indicates how the choice is to be made (i.e., probabilistically by *fitness*, probabilistically by *inverse fitness*, and *uniformly* random).

Listing 1. A basic genetic algorithm.

```
Algorithm
    Construct initial population P
    Compute fitnesses for strings in P
    best = min{score(p) | p in P}
    for i = 1 to the number of generations N_G {
        Offspring = empty set
        for j = 1 to number of crossovers N_C {
            p_1 = choose(P, fitness)
            p_2 = choose(P – {p_1}, fitness)
            c = crossover(p_1, p_2)
            if score(c) < best {
                best = score(c)
                solution = c
            }
            Offspring = Offspring + {c}
        }
        P = P + Offspring
        Compute fitnesses for strings in P
        for k = 1 to N_C {
            p = choose(P, –fitness)
            P = P – {p}
        }
        for l = 1 to number of mutations N_M {
            p = choose(P, uniformly random)
            q = mutate(p)
            if score(q) < best {
                best = score(q)
                solution = q
            }
            P = P – {p} + {q}
        }
        Compute fitnesses for strings in P
    }
    return solution
}
```

3.2 Current TSP crossover operators

A problem with conventional genetic algorithms for TSP is a meaningful crossover operator; that is, an operator that produces the offspring's tour out of elements (*building blocks*) from its parents p_1 and p_2. Let π be the tour represented by p_1 and let τ be the tour represented by p_2. Some of the more popular TSP crossovers are variants of the following methods (for a more complete list see Potvin [14]):

- *Basic one-point crossover* [14]: Without loss of generality suppose the TSTs of parents p_1 and p_2 both begin with the same city. The operator first randomly chooses a position i. The offspring tour begins with subpermutation $\pi_1 \ldots \pi_i$ of parent p_1. The tour is completed by visiting the remaining cities in the order that they occur in p_2's tour τ. With this operator, the offspring significantly resembles parent p_1, but its resemblance to parent p_2 is usually to a far lesser degree. Because of this situation, many GAs using this operator will produce two offspring for each pair of parents, where the parents alternately take on the role of p_1 and p_2.
- *Ordinal representation* [7]: For this operator, the value of a string's ith tour entry is the index into a ordered list of cities not referenced by indices 1, 2, … i–1. For example, if the third tour entry has value 4 and the cities not used in the first two tour locations are $\{c_1, c_3, c_4, c_6, c_7\}$, then the third city in the tour is c_6. The operator works by randomly choosing a position i. The offspring tour is derived from the values $\pi_1, \pi_2, \ldots \pi_i, \tau_{i+1}, \tau_{i+2}, \ldots \tau_n$. With this operator, the offspring significantly resembles parent p_1, but its resemblance to parent p_2 is usually minimal. Because of this situation, most GAs using this operator will produce two offspring for each pair of parents, where the parents alternately take on the role of p_1 and p_2.
- *Partially-mapped crossover* [6]: This operator tries to maintain the positions of the cities in the parents in the offspring. A substring s of parent p_1 chosen at random replaces a substring t in a copy γ of p_2. A cleanup procedure is then applied to γ to removes duplicates. (If possible, a duplicate d is replaced with the city in t that was overwritten by the occurrence of d in s).
- *Edge recombination crossover* [21]. From the parents's TSTs, an adjacency list is constructed for each city. The offspring tour is constructed by iteratively extending the tour to the city c with a minimal number of entries in its adjacency list. If there are multiple cities with a minimal number of entries, then one of those cities is selected at random. After extending the tour, the adjacency lists are updated to remove occurrences of c. The process repeats until all cities have been reached. The advantage of this method is that every edge in the offspring tour comes from one of the parents. Thus in GA parlance, the offspring is completely constructed from building blocks associated with its parents.

We believe it is the difficulty in recognizing and combining building blocks that makes TSP a hard problem for GAs. In the section that follows we propose a new representation for a TST string and an accompanying crossover operator. Experiments indicate that the representation and operator can effectively blend building blocks of both parents. We call our method the SPAN method. The representation and string combination operation is our primary contribution.

4. The SPAN method

In developing a GA for the TSP we had two conflicting goals. The first and primary goal is to generate the best possible solution. The second goal is to have the method run as fast as possible. As mentioned in the previous section, to generate much better solutions we use a different method of representing and combining solutions. To help achieve the second goal we abandoned the notion of absolute fitness for approximate fitness. An approximation scheme allows us to skip the steps of normalizing the scores to produce fitness values.

4.1 Perturbed city locations

Our string representation maintains two pieces of information for each city—its actual location and a perturbed location. For example, in Figure 2(a), we show the set of city locations from the instance depicted in Figure 1 along with a set of perturbed locations. In Figure 2(a), the line segments connect the true location and perturbed location pairs, where black points are original vertices and grey points are their perturbed locations. In Figure 2(b), we show the MST for the perturbed locations. Observe that the MST for the perturbed locations is different from (but similar to) the MST in Figure 1(b) for the true locations. In Figure 2(c), we show a repetition-free TST for the MST of the perturbed locations. Finally, in Figure 2(d), we show the TST for the unperturbed city locations using the vertex ordering from the repetition-free TST for the perturbed locations in Figure 2(c).

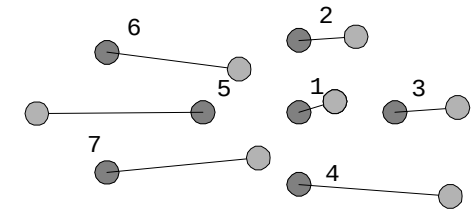
It is clear, if we were to have a different set of perturbations, we could have a different MST and different derived tours for the perturbed and unperturbed city locations. This observation is the intuition behind our approach. We begin by constructing a population of strings, where each string embodies a distinct set of perturbed locations, and thus a distinct TST.

With such a population, there are a variety of meaningful ways of combining two strings to produce an offspring string. For example, some of the possible methods are

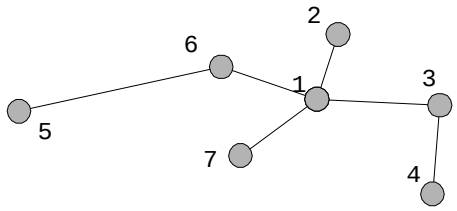
- The perturbed location of a city in the offspring is the geocenter of the perturbed locations of that city in its parents.
- The perturbed location of a city in the offspring is selected randomly to be one of the perturbed locations of that city in its parents.

The method we chose for our SPAN method is a variant on the first of the preceding methods.

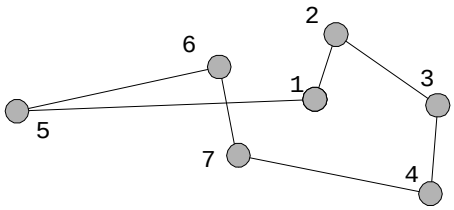
Figure 2. Perturbed vertices with MST and derived TSTs.



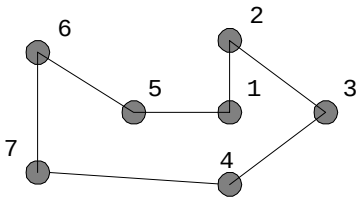
(a) Vertex perturbation correspondence



(b) MST of perturbed vertices



(c) TST derived using perturbed vertices



(d) TST derived using true vertices

- To compute the perturbed location of a city in the offspring first compute the geocenter of the perturbed locations of that city in its parents. Then use that geocenter as the mean value of a two-dimensional normal distribution where the variance is a function of the true intercity distances. Next draw a random variate from that distribution to use as the perturbed location. In practice, the variance for the normal distribution was a scaling of one standard deviation of the true intercity distances.

There are some properties to this method that are worth noting.

- The MSTs computed from the strings often have similar features because the perturbed locations are not that dissimilar from the true locations. As such, the MSTs using the perturbed locations are similar to the MST for the true locations. Therefore, in practice the TSTs derived from the MSTs have an acceptable cost.
- By using perturbed locations, the MSTs for the perturbed locations generally have some differences with the MST of the true locations. Therefore, the TSTs derived from the MSTs for the perturbed locations are generally different from each other and different from the TST derived from the MST of the true locations.

Thus, in practice we have a method that produces an offspring that is similar to, but not the same as, its parents. Also, because of the probabilistic aspects of the perturbation, different matings of the same strings produce different offspring. As a result, the operator allows for a good search of a given solution-space neighborhood. When coupled with the mechanics of the GA, we have an effective method that performs a directed but diverse search.

In summary, the crossover operator works in the following manner:

Produce a new set of perturbed locations using the two parents' sets of perturbed locations as a basis.
Compute, using heuristic H, the repetition-free TST γ derived from the MST of the offspring's perturbed locations.
Compute the TST using the true vertex locations with the vertex order from γ.

Our mutation operator also uses a perturbation method. The operator chooses a city from the string to be mutated and reperturbs the perturbed location of that city. The new value for the perturbed location is drawn from a two-dimensional normal distribution where the mean is the previous value of the perturbed location and the variance is a function of the true intercity distances. The variance for the distribution was a scaling of one standard deviation of the intercity distances. In our experiments, the operator was particularly effective in improving tours by fine-tuning small substrings. The GA for TSP by Valenzuela and Williams [20] also uses a perturbation-like method for the mutation operator. Their mutation operator uses a greedy nearest-neighbor method where the perturbed locations allow a different permutation to be generated.

4.2 Approximate fitness

In the genetic algorithm paradigm, determining which solutions are to be involved in crossover is done probabilistically with respect to fitness—better scoring solutions are more likely to be selected to be a parent than worse scoring solutions. Selecting which parents and offspring are to survive from the current population to form the starting population for the next generation is also done probabilistically with respect to fitness—better scoring solutions are more likely to be selected for survival that worse scoring solutions. Because fitness is calculated with respect to the current population, the fitnesses of the strings should be calculated twice per generation—once when the offspring are added to the population and then again after the population is culled to produce the starting population for the next generation. To avoid these linear-time calculations, when a mating is performed, instead of choosing two parents probabilistically with respect to fitness, we choose three strings in a random uniform manner. The two strings with better costing tours are mated, their offspring replaces the third string [5]. Preliminary experiments using this method for SPAN versus using the traditional selection process indicated that the approximate fitness scheme did not materially affect the solution quality, but did decrease the running time. Therefore, we use approximate fitness in SPAN.

5. Results

We test our method using nine standard TSP instances, where the number of cities varied from 30 to 225. The instances are listed in Table 1. Instance Osh comes from Oliver, Smith, and Holland [13], the other instances come from the TSP library of Reinelt [17].

Table 1: TSP Instances

Instance	Number of Cities
Osh	30
Eil	51
Berlin	52
St	70
KroA	100
Bier	127
Pr	136
Ch	150
Tsp	225

We chose to compare our method to the EC method of Whitley, T. Starkweather, and D. Shaner [21] (also see reference [19]), a method producing state-of-the-art GA results for the TSP problem. The EC method uses the edge recombination operator.

The results that we report in subsequent tables are all averages. Each entry represents the average of 25 runs of the associated method. For the SPAN method, the number of generations considered is a function of problem size— instances composed of less then 100 cities ran for 1,000 generations and instances composed of 100 or more cities ran for 2,000 generations. The EC method was allowed to run until the population was in equilibrium, which for our purposes was no improvement in the best-seen solution for 500 generations. Only for the smallest problem instance did the EC method require less than 2,000 generations (terminating after 1938 generations). On average, the EC method required approximately five times the number of generations used for SPAN. We allowed the EC method more generations to ensure that its solution quality was maximal. The population size per generation was a function of instance size. For instances composed of less then 100 cities, the population size was 100; for instances composed of 100 or more cities, the population size was 150. The perturbed location for a city in the initial population is drawn from a normal distribution whose mean is the true location of the city and whose variance is $1.5s^2$, where s is the standard deviation of the intercity distances. The crossover rate was set to 90% per generation. The perturbed location of a city in an offspring is drawn from a normal distribution whose mean was the average of the perturbed locations of the city in the parents and whose variance is $0.05s^2$. The attempted mutation rate was 90% per generation. For each

mutation, a perturbed city location is selected at random and mutated, the new perturbed location is drawn from a normal distribution whose mean is the unmutated perturbed city location and whose variance is s^2. The mutation was introduced into the population only if the new string had lower cost than the original string. (We also ran an experiment where a mutation was always accepted. This variant of SPAN produces results that were on average within 1% of the quality of the basic SPAN method.)

Table 2 shows the percentage improvement of the SPAN method over the EC method. The table indicates that the SPAN consistently outperforms the EC method. The minimum average improvement is 10.4% and the maximum average improvement is 50.8% average improvement. The overall improvement is 32.1%. Based on these results, we believe significant consideration must be made for using the SPAN method in TSP GAs.

Table 2: Comparison of EC and SPAN Solution Quality

Instance	EC Cost	SPAN Cost	SPAN Percentage Improvement
Osh	491	440	10.4%
Eil	579	481	16.9%
Berlin	10416	8890	14.6%
St	1112	742	33.2%
KroA	41272	23740	42.5%
Bier	200306	137801	31.2%
Pr	198380	112314	43.4%
Ch	14056	7666	45.5%
Tsp	10163	5003	50.8%

To show that the quality of a SPAN solution is not strictly based on its use of heuristic H, we present Table 3. This table compares the average solution quality of H and SPAN. The table indicates that a significant percentage of the improvement is due to the GA aspects of SPAN. At a minimum, SPAN improved the average solution quality by 7.1%. Its best average improvement was 21.1%. Overall, SPAN improved the solution quality by 15.1%. Such results are significant and demonstrate that SPAN is an effective GA for the TSP.

To show that the quality of a SPAN solution is based in part on its use of heuristic H, we ran an experiment where random spanning trees were used rather than MSTs. Although the traditional GA features of SPAN caused the population to improve over time, the final population was not substantially different in solution quality from the original population. We do not recommend considering random trees as avenue of future investigation.

We also ran an experiment using a variant V of SPAN, where a perturbed location of a city in the offspring was drawn from a normal distribution whose mean was randomly selected from one of the parents's perturbed locations for that city and where the variance is a function of the true intercity distances. Table 4 presents the results of the experiment with V. The results show that using SPAN's perturbing method contributes on average a minimum of 3.3% improvement; the maximum average improvement is 16.7%; and the overall average improvement is 10.5%. Thus we recommend the use of SPAN over the use of V.

A table of average running times is presented in Table 5, with all tests being run on an Alpha workstation. The running times are all in the acceptable range. We note that the running times of the SPAN method are in fact greater than the running times of the EC method. The principle reason being that both the SPAN crossover and mutation operators are $O(n^2)$ operations while the EC method has an $O(n)$ crossover and an $O(1)$ mutation operator. For example, the EC solution to the KroA instance of 100 cities takes on average approximately 10 minutes and the EC solution to the Tsp instance of 225 cities takes on average approximately 30 minutes. Note that the greater run times for the last 5 instances is primarily because SPAN ran for twice the number of generations (2,000) with a 50% population increase (150).

Table 3: Evaluation of H and SPAN Solution Quality

Instance	H Cost	SPAN Percentage Improvement
Osh	558	21.1%
Eil	570	15.7%
Berlin	10160	12.5%
St	887	16.3%
KroA	28602	17.0%
Bier	153968	10.5%
Pr	137136	18.1%
Ch	9349	18.0%
Tsp	5385	7.1%

Table 4: Evaluation of V and SPAN Solution Quality

Instance	V Cost	SPAN Percentage Improvement
Osh	455	3.3%
Eil	522	7.9%
Berlin	9501	6.4%
St	822	9.9%
KroA	28487	16.7%
Bier	158428	10.0%
Pr	134825	16.7%
Ch	9135	16.0%
Tsp	5417	7.6%

6. Summary and future research

We have demonstrated a new GA method for the TSP. This method uses a string representation that associates with each city a perceived location that is a perturbation of its true location. A perturbed location is generated using a normal distribution whose mean is the true city location. Our crossover operator takes two strings and produces an offspring, where the perturbed location of a city in the offspring is generated from a normal distribution whose mean is the average of the parents's perturbed location for that city. The score of a string is the cost of the TST derived from the minimum spanning tree of the perturbed locations. (The true locations are used in measuring the derived TST.)

To speedup our method, rather than using absolute fitness, we use a fitness approximation. To perform a crossover, three strings are selected with the better two strings used as parents. Our method works quite well in practice. Its solutions as measured using a broad standard test suite are approximately 32% better than the best previously published GA. The running time of the SPAN method is very acceptable given its solution quality. For example, the 100-city KroA instance takes on average 27 minutes to solve and the 225-city Tsp instance takes on average 134 minutes to solve.

Table 5: Average SPAN running times.

Instance	SPAN Running Time (Minutes)
Osh	3.4
Eil	9.6
Berlin	10.0
St	14.0
KroA	107.8
Bier	173.5
Pr	197.2
Ch	241.1
Tsp	539.1

We are currently completing a parallel implementation of the SPAN method. We are also testing the efficacy of our perturbed location approach to other problems (e.g. Minimum Steiner Tree Problem). The preliminary results for these problems are quite encouraging.

7. Acknowledgments

This research was supported in part by the Fulbright Program, National Science Foundation, and the Virginia Aerospace Consortium. Their support is greatly appreciated.

8. References

1. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *"Introduction to algorithms,"* McGraw-Hill: New York, 1990.
2. L. Davis, "Applying adaptive algorithms to epistatic domains," *International Conference on Genetic Algorithms and Their Applications*, pp. 162-164, Los Angeles, CA, 1985.
3. M. R. Garey and D. S. Johnson, "Computers And Intractability— A guide to the theory of NP-completeness," W. H. Freeman and Co: San Francisco, CA, 979.
4. P. C. Gilmore, E. L. Lawler, and D. B. Shmoys, "Well-solved special cases," in *"The traveling salesman problem,"* E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (editors), Wiley: New York, 1985.
5. D. E. Goldberg, *"Genetic Algorithms in Search, Optimization, and Machine Learning,"* Addison-Wesley: Reading, MA, 1988.
6. D. E. Goldberg, A R. Lingle, Jr., "Alleles, loci, and the traveling salesperson problem," *International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, pp. 154-159, 1985.
7. J. J. Grefenstette, R. Gopal, B. J. Rosmaita, and D. Van Gucht, "Genetic algorithms for the traveling salesperson problem," *International Conference on Genetic Algorithms and Their Applications*, pp. 160-168, Pittsburgh, PA, 1985.
8. J. H. Holland, *"Adaptation in Natural and Artificial Systems,"* University of Michigan Press: Ann Arbor, MI, 1975.
9. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, "Optimization by simulated annealing," *Science*, pp. 671-680, May 13, 1983.
10. J. B. Kruskal Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, pp. 48-50, 1956.
11. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (editors), *"The traveling salesman problem,"* Wiley: New York, 1985.
12. S. Lin and B. W. Kernighan, "An Effective Heuristic for the Travelling Salesman Problem," *Operations Research*, pp. 498-516, 1973.
13. I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the traveling salesman problem," *International Conference on Genetic Algorithms*, Cambridge, MA, pp. 224-230, 1987.
14. J.-Y. Potvin, "Genetic algorithms for the traveling salesman problem," *Annals of Operations Research*, pp. 339-370, 1996.

15. R. C. Prim, “Shortest connection networks and some generalizations,” *Bell Systems Technical Journal*, pp. 1389-1401, 1957.
16. R. M. Karp, “Probabilistic analysis of partitioning algorithms for the traveling-salesman problem,” Mathematics for Operations Research, pp. 209-225, 1977.
17. G. Reinelt, “TSPLIB — A traveling salesman problem library,” *ORSA Journal on Computing*, pp. 376-384, 1991. Data instances are available at University of Heidelberg, http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.
18. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, “An analysis of several heuristics for the traveling salesman problem,” *SIAM Journal of Computing*, pp. 563-581, 1977.
19. T. Starkweather, S. McDaniel, K. Mathias, D. Whitely, C. Whitely, “A comparison of genetic sequencing operators, *International Conference on Genetic Algorithms and Their Applications*, pp. 69-76, 1991.
20. C. L. Valenzuela and L. P. Williams, “Improving Simple Heuristic Algorithms for the Travelling Salesman Problem using a Genetic Algorithm,” *International Conference on Genetic Algorithms*, 1997, pp. 458-464.
21. D. Whitley, T. Starkweather, and D. Shaner, “Traveling salesman and sequence scheduling: quality solutions using genetic edge recombination,” in “Handbook of genetic algorithms,” L. Davis (editor), Van Nostrand Reinhold: New York, pp. 350-372, 1990.