The Traveling Salesman Problem: A Case Study in Local Optimization

David S. Johnson[1]
Lyle A. McGeoch[2]

Abstract

This paper will appear as a chapter in the book *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra (eds.), John Wiley and Sons, New York. The traveling salesman problem (TSP) has been an early proving ground for many approaches to combinatorial optimization, including classical local optimization techniques as well as many of the more recent variants on local optimization, such as simulated annealing, tabu search, neural networks, and genetic algorithms. This chapter discusses how these various approaches have been adapted to the TSP and evaluates their relative success in this perhaps atypical domain from both a theoretical and an experimental point of view.

November 20, 1995

[1] Room 2D-150, AT&T Bell Laboratories, Murray Hill, NJ 07974, dsj@research.att.com
[2] Department of Mathematics and Computer Science, Amherst College, Amherst, MA 01002, lam@cs.amherst.edu

3.3. How to Make 2-Opt and 3-Opt Run Quickly

In order to obtain running times for 2- and 3-Opt that are subquadratic in practice, we must first figure out a way to avoid looking at all possible moves in our search for one that shortens the tour. Fortunately, there is a simple and effective way to do this, apparently first noticed by Lin and Kernighan [1973].

To illustrate this, let us consider the case of 2-Opt, and let us assume a fixed orientation of the tour, with each tour edge having a unique representation as a pair (x,y), where x is the immediate predecessor of y in tour order. Then each possible 2-Opt move can be viewed as corresponding to a 4-tuple of cities $< t_1, t_2, t_3, t_4 >$, where (t_1, t_2) and (t_4, t_3) are the oriented tour edges deleted and $\{t_2, t_3\}$ and $\{t_1, t_4\}$ are the edges that replace them. Under this convention, each move actually corresponds to two distinct 4-tuples. For instance, the 2-Opt move illustrated in Figure 1 corresponds, under the counter-clockwise tour orientation, to the 4-tuples $< a,b,c,d >$ and $< d,c,b,a >$. As first observed by Steiglitz and Weiner [1968], we can exploit this double representation as follows. Observe that for an improving move it must be the case that either (a) $d(t_1,t_2) > d(t_2,t_3)$ or (b) $d(t_3,t_4) > d(t_4,t_1)$ or both. But this means that (a) must hold for at least one of the two representations of the move. Consequently, we cannot miss an improving move if we restrict attention to 4-tuples satisfying (a). This means that in our search, having fixed t_1 and t_2, the possibilities for t_3 can be limited to those cities that are closer to t_2 than is t_1. (Note that given the choice of t_3, there is only one possible choice of t_4 such that the move will yield a tour rather than two disjoint cycles.) As the algorithm progresses, the set of candidates for t_3 is more and more likely to be a small set. Analogously for 3-Opt (as observed by Lin and Kernighan [1973]), we may restrict attention to 6-tuples $< t_1, t_2, t_3, t_4, t_5, t_6 >$ in which $d(t_1,t_2) > d(t_2,t_3)$ and $d(t_1,t_2) + d(t_3,t_4) > d(t_2,t_3) + d(t_4,t_5)$. This limits the search for t_3 as in 2-Opt but also imposes a substantial constraint on the possibilities for t_5. (Degenerate 6-tuples that repeat some cities and hence correspond to 2-Opt moves are allowed.)

In order to exploit the above observations, we need a data structure that allows us to quickly identify the allowable candidates for t_3 (t_5). Steiglitz and Weiner proposed storing for each city c a list of the remaining cities in order of increasing distance from c. To consider candidates for t_3, we then need only start at the beginning of t_2's list and proceed down it until a city x with $d(t_2,x) \geq d(t_1,t_2)$ is encountered. (The search for t_5 candidates in 3-Opt works similarly, using t_4's list.) The drawback to this approach is that it would take $\Theta(N^2 \log N)$ time to set up the lists and $\Theta(N^2)$ space to store them. Our *neighbor-list* implementations make the reasonable compromise of storing sorted lists containing only the k nearest neighbors for each city for a suitable fixed k, typically $k = 20$. We then halt our searches for t_3 (t_5) if either we reach a city that is too far away or we get to the end of a list. This data structure takes only $\Theta(Nk)$ space and can be constructed in time $\Theta(N^2 \log k)$, a bound that can be improved to roughly $\Theta(N \log N + Nk)$ for geometric instances using *k-d* trees. As we saw in the last section, even with these speedups, the construction time for neighbor lists dominates the overall running time for the neighbor-list implementations of 2- and 3-Opt. Note for future reference, however, that the neighbor lists are static structures and could be re-used by subsequent runs of the algorithms on the same instance, thus amortizing the cost of their construction. Thus the time needed to perform *m* runs of 3-Opt from different randomized Greedy starts would be substantially less than *m* times the time to perform a single run.

The loss of tour quality involved in using truncated neighbor lists is not a major issue in practice, although it is possible that the final solution may not be truly locally optimal. For random Euclidean instances, going from $k = 20$ to $k = 80$ only tends to improve the final tour by 0.1 or 0.2% on average, at significant running time cost. For some `TSPLIB` instances the difference can be greater, but even here the point of diminishing returns sets in around $k = 40$, especially if one uses a variant in which the 40 neighbors of *c* are chosen so as to include the 10 closest cities in each of the four quadrants of the plane surrounding *c*. For random distance matrices and random starts, increasing *k* may actually make the final tour get worse; the optimal choice seems to be about $k = 15$.

The above ideas can for fixed *k* reduce the time for finding an improving 2-Opt (3-Opt) move from $\Theta(N^2)$ ($\Theta(N^3)$) to $O(N)$, but given that there are roughly $\Theta(N)$ moves made for random Euclidean instances, a further idea is needed. This is provided by the concept of the *don't-look* bit, introduced by Bentley [1992], another way of trading a slight increase in the possibility of a non-locally optimal final tour for a major reduction in running time, again based on the idea that we need only consider improving moves. Here the observation is that if for a given choice of t_1 we previously failed to find an improving move, and if t_1's tour neighbors have not changed since that time, then it is unlikely that we will find an improving move if we look again starting at t_1. We exploit this observation by means of special flags for each of the cities, which we call *don't-look* bits. Initially, these are all turned off. The bit for city *c* is turned on whenever a search for an improving move with $t_1 = c$ fails and is turned off whenever a move is performed in which *c* is an endpoint of one of the deleted edges. In considering candidates for t_1, we ignore all cities whose don't-look bits are on. This is done by maintaining a first-in, first-out queue of cities whose bits are off.

As a result of all this, the total number of t_1's considered throughout the course of the algorithm on random Euclidean instances is reduced to roughly $\Theta(N)$, although the constant is fairly large. The total search time is also $\Theta(N)$. The running time bottleneck thus becomes the time to perform the moves that are found, which depends intimately on the method used for representing the current tour. For the results reported in the previous section, we use the *two-level tree* data structure of Fredman, Johnson, McGeoch, and Ostheimer [1995], which theoretically yields a cost of $O(N^{1/2})$ per move. Although this is asymptotically worse than the splay tree representation mentioned in Section 3.1, it is faster for the range of *N* we consider. See Fredman, Johnson, McGeoch, and Ostheimer [1995] for more details. An alternative *segment tree* data structure due to Applegate, Chvátal, and Cook [1990] is also competitive for $N \le 100{,}000$; see the above reference, Reinelt [1992], and Reinelt [1994].

Beyond the above algorithmic ideas, our implementation includes several tricks that merely yield constant factor improvements, such as caching of distances. See Johnson, Bentley, McGeoch, and Rothberg [1996] for more complete details. That paper also examines in more detail the effects of various choices one must make in an implementation, such as what one does when there is more than one improving move to choose from. Choosing the best move seems a good policy, but the best move can be expensive to find, so there are tradeoffs involved. The neighbor-list implementations typically choose the first improving move found, but the search is biased so as to find better moves first. The paper also includes an extensive evaluation of the alternative *on-the-fly* implementation of Bentley [1992] for geometric instances, which dispenses with the neighbor lists entirely and uses a *k-d* tree directly to find candidates for t_3 and t_5. This enables us to consider all the legal candidates in order, without truncation. It also reduces preprocessing time and the storage space needed by the algorithm, which for the neighbor-list implementation can be substantial when N is large. For $N = 10^6$ the reduction is roughly from 275 to 72 megabytes. The drawback is that each candidate takes longer to identify, so the local optimization phase takes more time. For 2-Opt the overall result is slightly better running times than for the neighbor-list implementation and slightly better tours; for 3-Opt it is slightly slower running times and slightly worse tours (this last may be due to implementation details other than the data structure chosen, however). Another approach, advocated by Reinelt [1991,1994] and Jünger, Reinelt, and Rinaldi [1994] is to stick to neighbor lists, but use Delaunay triangulations instead of *k-d* trees in constructing the lists. As implemented by the above authors, this approach yields comparable running times for 2-Opt but, because of other choices made in the implementation, worse tours. For 3-Opt both running times and tours appear to be worse than with the neighbor-list implementation.

3.4. Parallel 2- and 3-Opt

One oft-mentioned method for speeding up TSP algorithms is the use of parallelism. Although the times reported above for 2- and 3-Opt on a single processor seem reasonable for most applications, one can conceive of situations where much greater speed would be needed because of real-time constraints, or where the number of cities grows so large that many hours would be required if one uses only one processor. Perhaps more significantly, the memory requirement of roughly 275 bytes per city for our neighbor-list implementations means that the largest instance a 30-megabyte workstation can handle is only on the order of 100,000 cities, so if one wants to solve larger instances on such machines, one must resort to parallelism of some sort, for instance by distributing the problem across a network of workstations. Various schemes have been proposed for this, all exploiting a coarse-grained parallelism where each processor has associated with it enough memory to run TSP algorithms on instances up to some reasonable size. We shall discuss three basic approaches that can be applied not only to 2- and 3-Opt but also to many of the alternative approaches to be covered in later sections.

Geometric Partitioning

For 2-dimensional geometric instances, a partitioning scheme proposed by Karp [1977] can be used. This scheme partitions the cities in a manner similar to that used in the construction of the *k-d* tree data structure. It is based on a recursive subdivision of the overall region containing the cities into rectangles, with the set of cities corresponding to a given rectangle being comprised of all the cities in the rectangle's interior together with some of the cities on its boundary. Suppose we wish to construct a partition in which no set contains more than K cities. The recursive step of the partitioning scheme works as follows. Let C_R be the set of cities assigned to rectangle R, and suppose $|C_R| > K$. One subdivides R into two subrectangles as follows. Suppose without loss of generality that the x-coordinates of the cities in C_R have a larger range than the y-coordinates. Find a city c in C_R whose x-coordinate has the median value for cities in C_R. Divide R into two subrectangles R_1 and R_2 by drawing a vertical line through c, letting R_1 be the rectangle to the left of the line. Each city in C_R to the left of the line is assigned to R_1, each city to the right is assigned to R_2, and cities on the line are divided as equally as possible between the two rectangles, except that city c itself is assigned to *both* R_1 and R_2.

Once the cities have been partitioned in this way into subrectangles, none of which has more than K cities assigned to it, one can send each subrectangle to a processor, and have that processor run 2- or 3-Opt (or any other TSP algorithm) on the corresponding set of cities. The union of the tours thus found will be a connected Eulerian subgraph, and so it can be converted to a tour for the entire instance by using shortcuts as in the Christofides algorithm.

Although we have not implemented this scheme as a parallel algorithm, its value as a way to reduce memory requirements can be exploited even on a single processor: we simply solve the subproblems one after another, reusing the space required for each one. To the extent that the running time of our basic TSP algorithm is superlinear, this sequential scheme can even provide speed-ups over performing the algorithm on the complete instance. For $N = 10^6$ and 3-Opt, the savings in run-time obtained by partitioning into 1024 subproblems was over a factor of 2. Unfortunately, there was also a corresponding loss in tour quality, with the original 3.0% average excess over the Held-Karp bound increasing to 5.5%. Partitioning into just 16 subproblems, which just might be enough to allow us to handle 10^6 cities on our hypothetical 30 megabyte workstation, greatly reduces the tour-quality penalty; now the increase is only to 3.2%. This general scheme can also be applied to instances with fewer cities, although for such instances the deterioration in tour quality sets in earlier, i.e., for smaller numbers of subproblems. A rule of thumb seems to be that as soon as the number of cites in a subproblem drops below 1000, one can expect significant deterioration.

Reinelt [1994] has proposed two variants on the above sheme, both based on generating a sparse graph whose vertices are the cities and whose connected components can serve as the subproblems to be solved by local optimization. In the first scheme, the sparse graph simply contains for each city the edges to that city's nearest 2 (or 3) neighbors. This approach tends to yield some very small connected components, and Rohe

[1995] has suggested a process whereby one successively merges nearby small components until each merged component contains between $M/2$ and M cities, where M is the maximum size subproblem one wishes to consider. In Reinelt's second scheme, the sparse graph is generated using edges from the Delaunay triangulation of the set of cities. The edges of the triangulation are considered in order of increasing length, and an edge is added to the graph unless it would cause the creation of a subproblem with more than M cities or would reduce the total number of subproblems below some prescribed threshold.

Given such a partition into connected components, a tour can be constructed as follows. The global ordering of the components within the tour is determined first, along with the identities of the edges that will link each component to its successor. Each subproblem then has two distinguished cities, i.e., those that are involved in inter-component links. Before applying local optimization to a subproblem, we add an unbreakable edge between these two. This insures that the resulting tour for the subproblem will contain a Hamiltonian path linking the entry and exit cities. Once all subproblems have been solved in this way, it will be straightforward to assemble the overall tour. Based on experiments with 24 selected instances from `TSPLIB`, Reinelt concludes that neither of these two approaches is preferable to a rectangle-based scheme. It should be noted, however, that his first variant is more general than the others: It does not explicitly rely on the geometric nature of the instance (although having the triangle inequality hold would probably be helpful).

Tour-Based Partitioning

One reason for the loss in tour quality engendered by geometric partitioning schemes is the fact that each subproblem is handled in isolation, with only limited opportunities for intelligent patching together of the subtours. Another drawback of the scheme is that, although it can be generalized to higher dimensions, it is restricted to geometric instances. The alternative *tour-based* partitioning scheme avoids these pitfalls by basing its partition of the cities on the structure of the current tour and by performing more than one partitioning phase.

One begins by using a simple heuristic to generate an initial tour and then breaks that tour up into k segments of length N/k, where k is greater than or equal to the number of processors available. Each segment is then handed to a processor, which converts the segment into a tour by adding an edge between its endpoints and attempts to improve the tour by local optimization (2-Opt, 3-Opt, etc.), subject to the constraint that the added edge cannot be deleted. The resulting tour can thus be turned back into a segment with the same two endpoints, and the improved segments can then be put back together into a new overall tour. We can then construct a revised partition where each new segment takes half its cities from each of two adjacent old segments and repeat the parallel local optimization phase. Additional phases can be performed until a time limit is exceeded or no significant further improvement is obtained. This scheme has been tried by Applegate and Cook [1994] on a 10,907,064-city instance using a network of 10 workstations and a k of about 1000. They assigned 100 subproblems to each processor, and each processor ran its assigned subproblems in sequence. A neighbor-list implementation of the Lin-

Kernighan algorithm of Section 4.2 was used as the local-optimization engine. The resulting tour was within 4.3% of the Held-Karp bound for the instance.

Note that there still appears to be a tour-quality penalty for partitioning in this way. Despite the shifting of the segment boundaries, it remains difficult to move a city very far away from its original position in the tour. An alternative tour-based partitioning scheme suggested by Allwright and Carpenter [1989] helps alleviate this drawback. As in the above scheme, they partition the current tour into segments, but now they create $2k$ segments and hand out *pairs* of segments to processors. If the segments are $S_1, S_2, ..., S_{2k}$ in order around the tour, we pair S_1 with S_{2k}, S_2 with S_{2k-1}, and in general S_j with S_{2k-j+1}, $1 \le j \le k$. Note that the first and last pairs are degenerate, as the segments share an endpoint, and so each can be viewed as a single segment and treated as in the previous scheme. A non-degenerate pair of segments is converted into a tour by adding two edges, each connecting one of the endpoints of the first segment to one of the endpoints of the second. The added edges are chosen so that each shortcuts a path made up of tour edges not in either segment. The processor is then free to perform any local optimization step on its subtour that does not break an added edge. After a phase of local optimization based on such a partition, an overall tour can then be reassembled from its parts, and we can repartition the tour and try again.

Allwright and Carpenter proposed this scheme in the context of 2-Opt, but did not attempt a serious test of the algorithm. Verhoeven, Aarts, and Swinkels [1995] modified the approach and tested it on `TSPLIB` instances with as many as 11,849 cities, using a network of 512 INMOS T805 transputers. Their modification includes a clever repartitioning scheme that guarantees that their algorithm will not halt until it has found a solution that is locally optimal with respect to the 2-Opt neighborhood for the complete problem. Consequently, they experience little if any degradation in tour quality as the number of processors increases. Unfortunately, this scheme requires $\Omega(N^2/p + pN)$ time, where p is the number of processors. This is $\Omega(N^{1.5})$ regardless of p and so can provide significant speedups only when compared to the naive $\Omega(N^2)$ implementation of 2-Opt, which we already know how to speed up even more while using only a single processor. Thus for the 11,849-city instance and 512 processors, the running time is almost a factor 200 greater than the time for neighbor-list 2-Opt on a single SGI Challenge processor, a factor that appears to be growing with instance size.

Some of this factor can be blamed on the fact that the transputers are slower processors, but most is due to the nature of the approach itself, which is driven by the desire to obtain local optimality with respect to the full 2-Opt neighborhood. Nor is this necessarily that desirable a goal. The tours found by Verhoeven et al., although as good as the tours they obtained with their single-processor implementation of full 2-Opt, are still not as good as those obtained using neighbor-list 2-Opt. For the 11,849-city instance they are some 3% worse. This appears to be typical and is probably a consequence of the different orders in which moves are considered and made in the two algorithms. For an analogous parallel implementation of a 3-Opt variant, Verhoeven, Aarts, van de Sluis, and Vaessens [1992] report much better tour quality results (ones comparable to the neighbor-list implementation), although their running times are still far in excess of those for the single-processor neighbor-list implementation. Thus this does not appear to be a competitive approach. For a partition-based parallel scheme to succeed, it probably must be organized so that it can use the same speed-up tricks our sequential implementations exploited, even if this means that local optimality cannot be guaranteed.

One might also wish to consider the following hybrid between the geometric and tour-based partitioning schemes proposed by Rohe [1995]. Given one's current tour, one generates subproblems consisting of cities closely related in a geometric sense, for example all cities in a given rectangle or the M nearest neighbors of a given city. For a given subproblem S, the current tour induces a collection of paths through the cities of S. We add additional edges to link these paths up in the same order they occur in the overall tour, yielding a tour for S, and then perform local optimization on this tour, with the added edges being fixed and undeletable. This results in a new set of paths through S which can replace the old set in the tour for the entire instance. Assuming that local optimization improved the tour for S, the new overall tour will be shorter as well. When multiple subproblems of this sort are handled in parallel, there is some danger that the overall result will be a collection of disjoint cycles rather than a tour, even if no two subproblems share a city. In Rohe's scheme, one checks for such subtours and patches them together using additional 2-Opt moves. Rohe applied this scheme in a multi-phase approach to the same 10,907,064-city instance for which we reported Applegate and Cook's results in the previous section. Using the same Lin-Kernighan local search engine and two weeks on a network of four IBM 550 workstations, he reduced the excess over the Held-Karp bound from Applegate and Cook's 4.3% to 1.57%. This is roughly the tour quality that Lin-Kernighan might be expected to produce were it to be run on the whole (unpartitioned) instance. Rohe has also obtained a tour of this quality for an even bigger instance, one whose cities correspond to the locations in the sky of some 18 million stars.

Using Parallelism in Preprocessing and the Search for Improving Moves

When memory is not a constraint, there are much simpler ways to obtain significant speedups over our fast sequential implementations of local optimization algorithms without paying a penalty in tour quality. First recall that typically over 2/3 of the time for the neighbor-list implementations of 2- and 3-Opt is spent in building the neighbor lists themselves, which is readily parallelizable assuming each processor has enough memory to store the full instance.

Similarly, the search for improving moves typically dominates the remaining time and offers ample opportunities for parallelism. For example, when neighbor-list 3-Opt is applied to a random Euclidean instance, we typically evaluate 50 or more moves for every move actually made. There is no reason why these searches cannot be performed in parallel. Each processor may need access to the entire instance and all the neighbor lists, but assuming there is enough memory so that all this information can be replicated at each processor, significant reductions in running time should be possible.

We know of no serious attempt to exploit these two avenues of parallelism in the context of 2- and 3-Opt, but search parallelization becomes even more attractive for some of the algorithms we shall discuss in later sections, and an application of it to the Lin-Kernighan algorithm will be mentioned in Section 4.

3.5. Other Simple Local Optimization Algorithms

Simpler local optimization algorithms than 2- and 3-Opt exist. For example, one could restrict exchanges to those in which two adjacent cities in the current tour are interchanged, thus reducing the size of a neighborhood from the $N(N-2)/2$ of 2-Opt to N. However, such simpler neighborhoods are not powerful enough to yield results at all close to those of 2-Opt, and as observed above, versions of 2-Opt can be implemented to run far more quickly than its neighborhood size would suggest. Thus the real game is to devise local optimization algorithms that are a bit more complicated than 2-Opt (or 3-Opt) but produce better tours.

Between 2- and 3-Opt

A first, simple extension is what Bentley [1992] calls *2.5-Opt*. In this algorithm we expand the 2-Opt neighborhood to include a simple form of 3-Opt move that can be found with little extra effort. In a 2.5-Opt move one relocates a single city from its current location to a position between two current tour neighbors elsewhere in the tour. In Figure 2, this corresponds to the situation where *b* and *c* are the same city. The search for such moves can be incorporated into our basic 2-Opt search as follows: For each t_3 candidate (i.e., each city that is closer to t_2 than is t_1) we evaluate the corresponding 2-Opt move *plus* the 2.5-Opt move that places t_2 between t_3 and t_4 and the 2.5-Opt move that places t_3 between t_1 and t_2. 2.5-Opt has been implemented for geometric instances using the on-the-fly approach by Bentley [1992]. For random Euclidean instances it yields a 0.5% improvement in average tour length over 2-Opt at the cost of a 30-40% increase in running time. For our testbed of `TSPLIB` instances, the improvement in tour length over 2-Opt was more like 1%. This was still 1% worse than what our *neighbor-list* implementation of full 3-Opt obtains, but it took only half the time.

A second algorithm that is intermediate between 2- and 3-Opt is the *Or-Opt* algorithm originally proposed by Or [1976] and popularized by Golden and Stewart [1985]. Here we restrict attention to 3-Opt moves in which a segment consisting of three or fewer consecutive cities is excised from the tour and placed between two tour neighbors elsewhere in the tour. In Figure 2, this corresponds to the situation where cities *b* and *c* are 0, 1, or 2 cities apart in the tour, a natural generalization of the 2.5-Opt move. A version of Or-Opt could be implemented by slightly modifying an implementation of 2.5-Opt. Such an implementation should be a bit faster than full 3-Opt, although a slight loss in tour quality is to be expected. To date, however, Or-Opt has only been studied by researchers who assumed that 3-Opt requires $\Theta(N^3)$ time and so settled for Or-Opt implementations that take quadratic time. Such implementations are not competitive with our *neighbor-list* implementation of full 3-Opt in either running time or tour quality.

154. W. T. RHEE AND M. TALAGRAND, ‘‘A sharp deviation inequality for the stochastic traveling salesman problem,’’ *Annals of Probability* **17** (1988), 1-8.

155. N. ROBERTSON AND P. D. SEYMOUR, ‘‘Graph Minors X. Obstructions to tree-decomposition,’’ *J. Combinatorial Theory Ser. B* **52** (1991), 153-190.

156. A. ROHE, private communication (1995).

157. D. J. ROSENKRANTZ, R. E. STEARNS, AND P. M. LEWIS, II, ‘‘An analysis of several heuristics for the traveling salesman problem,’’ *SIAM J. Comput.* **6** (1977), 563-581.

158. Y. ROSSIER, M. TROYON, AND T. M. LIEBLING, ‘‘Probabilistic exchange algorithms and Euclidean traveling salesman problems,’’ *OR Spektrum* **8** (1986), 151-164.

159. S. SAHNI AND T. GONZALEZ, ‘‘P-complete approximation problems,’’ *J. Assoc. Comput. Mach.* **23** (1976), 555-565.

160. B. SCHNETZLER, ‘‘Des opérateurs d’échange et une méthode de relaxation pour le problème du voyageur de commerce,’’ *Recherche Operationelle/Operations Research* **26** (1992), 57-81.

161. D. B. SHMOYS AND D. P. WILLIAMSON, ‘‘Analyzing the Held-Karp TSP bound: A monotonicity property with applications,’’ *Inform. Process. Lett.* **35** (1990), 281-285.

162. M. W. SIMMEN, ‘‘Parameter sensitivity on the elastic net approach to the traveling salesman problem,’’ *Neural Computation* **3** (1991), 363-374.

163. D. D. SLEATOR AND R. E. TARJAN, ‘‘Self-adjusting binary search trees,’’ *J. Assoc. Comput. Mach.* **32** (1985), 652-686.

164. T. STARKWEATHER, S. MCDANIEL, K. MATHIAS, D. WHITLEY, AND C. WHITLEY, ‘‘A comparison of genetic sequencing operators,’’ in *Proc. 4th Int. Conf. on Genetic Algorithms*, R. K. Belew and L. B. Booker (Eds.), Morgan Kaufman, San Francisco, CA, 1991, 69-76.

165. J. M. STEELE, ‘‘Complete convergence of short paths and Karp’s algorithm for the TSP,’’ *Math. Oper. Res.* **6** (1989), 374-378.

166. K. STEIGLITZ AND P. WEINER, ‘‘Some improved algorithms for computer solution of the traveling salesman problem,’’ *in* ‘‘Proceedings 6th Ann. Allerton Conf. on Communication, Control, and Computing,’’ Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois, Urbana, Ill., 1968, 814-821.

167. D. STEIN, *Scheduling Dial-a-Ride Transportation Systems: An Asymptotic Approach*, Ph.D. Dissertation, Harvard University, Cambridge, MA, 1977.

168. J. Y. SUH AND D. VAN GUCHT, ‘‘Distributed genetic algorithms,’’ Report No. 225, Computer Science Department, Indiana University, Bloomington, IN, 1987a.

169. J. Y. SUH AND D. VAN GUCHT, ‘‘Incorporating heuristic information into genetic search,’’ in *Proc. 2nd Int. Conf. on Genetic Algorithms*, J. J. Grefenstette (Ed.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1987b, 100-107.

170. M. TROYON, *Quelques Heuristiques et Résultats Asymptotiques pour trois Problèmes d’Optimisation Combinatoire*, Thèse No. 754, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1988.

171. N. L. J. ULDER, E. H. L. AARTS, H.-J. BANDELT, P. J. M. VAN LAARHOVEN, AND E. PESCH, ‘‘Genetic local search algorithms for the traveling salesman problem,’’ in *Proc. 1st Int. Workshop on Parallel Problem Solving from Nature*, H. P. Schwefel and R. Männer (Eds.), Lecture Notes in Computer Science, Vol. 496, Springer, Berlin, 1991, 109-116.

172. A. I. VAKHUTINSKY AND B. L. GOLDEN, ‘‘A hierarchical strategy for solving traveling salesman problems using elastic nets,’’ *J. Heuristics*, to appear.

173. C. L. VALENZUELA AND A. J. JONES, ''Evolutionary divide and conquer (I): A novel genetic approach to the TSP,'' *Evolutionary Computation* **1** (1994), 313-333.

174. D. E. VAN DEN BOUT AND T. K. MILLER, III, ''Improving the performance of the Hopfield-Tank neural network through normalization and annealing,'' *Biol. Cybern.* **62** (1989), 129-139.

175. J. VAN LEEUWEN AND A. A. SCHOONE, ''Untangling a traveling salesman tour in the plane,'' Report No. RUU-CS-80-11, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1980.

176. M. G. A. VERHOEVEN, E. H. L. AARTS, E. VAN DE SLUIS, AND R. J. M. VAESSENS, ''Parallel local search and the travelling salesman problem,'' in *Parallel Problem Solving from Nature 2*, North-Holland, Amsterdam, 1992, 543-552.

177. M. G. A. VERHOEVEN, E. H. L. AARTS, AND P. C. J. SWINKELS, ''A parallel 2-opt algorithm for the traveling salesman problem,'' *Future Generation Computing Systems* **11** (1995), 175-182.

178. M. G. A. VERHOEVEN, P. C. J. SWINKELS, AND E. H. L. AARTS, ''Parallel local search for the traveling salesman,'' manuscript (1995).

179. P. WEINER, S. L. SAVAGE, AND A. BAGCHI, ''Neighborhood search algorithms for finding optimal traveling salesman tours must be inefficient,'' *in* ''Proceedings 5th Ann. ACM Symp. on Theory of Computing,'' Association for Computing Machinery, New York, 1973, 207-213.

180. D. WHITLEY, T. STARKWEATHER, AND D. FUQUAY, ''Scheduling problems and traveling salesman: The genetic edge recombination operator,'' in *Proc. 3rd Int. Conf. on Genetic Algorithms*, J. D. Schaffer (Ed.), Morgan Kaufman, San Francisco, CA, 1989, 133-140.

181. G. V. WILSON AND G. S. PAWLEY, ''On the stability of the traveling salesman problem algorithm of Hopfield and Tank,'' *Biol. Cybern.* **58** (1988), 63-70.

182. L. WOLSEY, ''Heuristic analysis, linear programming, and branch and bound,'' *Math. Prog. Stud.* **13** (1980), 121-134.

183. X. XU AND W. T. TSAI, ''Effective neural algorithms for the traveling salesman problem,'' *Neural Networks* **4** (1991), 193-205.