

# AN IMPROVED ADAPTIVE MULTI-START ALGORITHM FOR FINDING NEAR-OPTIMAL SOLUTIONS TO THE EUCLIDEAN TSP

DAN BONACHEA, EUGENE INGERMAN, JOSHUA LEVY, AND SCOTT MCPEAK

{bonachea,smcpeak}@cs.berkeley.edu, {jdl,eugening}@math.berkeley.edu

*Computer Science 270, UC Berkeley  
Prof. Sinclair  
Final project  
May 24, 1999*

ABSTRACT. We present an “adaptive multi-start” algorithm for the Euclidean traveling salesman problem that uses a population of good solutions and a multi-parent cross-breeding technique, chosen to exploit the structure of the search space to generate better tours. Our work generalizes and improves upon the approach of Boese et al. [2].

Experiments show the algorithm is a vast improvement over simple “multi-start,” i.e., applying Lin-Kernighan to many random initial tours. Both for random and several standard TSPLIB [5] instances, it is able to find nearly optimal (or optimal) tours for problems of several thousand nodes in a few minutes on a Pentium Pro workstation. We find these results are competitive both in time and tour length with one of the most successful TSP algorithms, iterated Lin-Kernighan.

## 1. BACKGROUND

1.1. **The TSP.** In the traveling salesman problem (TSP) we are given  $n$  “cities”  $c_1, \dots, c_n$  and a positive distance  $d(c_i, c_j)$  for each distinct pair of cities. Our goal is to find an ordering  $\pi$ , or *tour*, of the cities that minimizes the *length* of the tour,  $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$ . We will restrict our attention to the two-dimensional Euclidean TSP, which is the special case where the cities are points in the plane and  $d(c_i, c_j)$  is the Euclidean distance from  $c_i$  to  $c_j$ . Figure 1 shows three TSPs: a random instance, in which points are chosen uniformly at random on a square, and two non-random instances from the TSPLIB [5] archive of problems.

Although TSP and Euclidean TSP are NP-hard, it is usually relatively easy to obtain an approximate TSP solution that is close in length to the optimal solution. Its relative tractability may explain why, for more than three decades, the traveling salesman problem has been a proving ground for techniques of combinatorial optimization. Here we will describe several of the approaches that are related to our algorithm. Details may be found in the excellent survey by Johnson and McGeoch [6].

1.2. **Approximation algorithms for TSP.** A “tour construction” heuristic builds a solution tour from scratch. The random construction heuristic randomly adds legal edges (i.e., edges between points attached to less than two edges already, and that do not form a cycle prematurely). The greedy heuristic adds the shortest legal edge until the tour is complete. The nearest-neighbor heuristic mimics the traveler who always goes to the nearest as-yet-unvisited location. In the worst case the greedy and nearest-neighbor heuristics give tours that are  $\Theta(\log n)$  times longer than the optimal. However, the practical performance of the greedy and nearest-neighbor heuristics are much better—only about 10–20% worse than optimal.

Another widely used type of approximation algorithm for TSP is local search. Two of the simplest local search algorithms are 2-opt and 3-opt. These algorithms try to optimize the existing tour by first removing 2 or 3 edges, and then reconnecting the resulting pieces to obtain a better tour. In

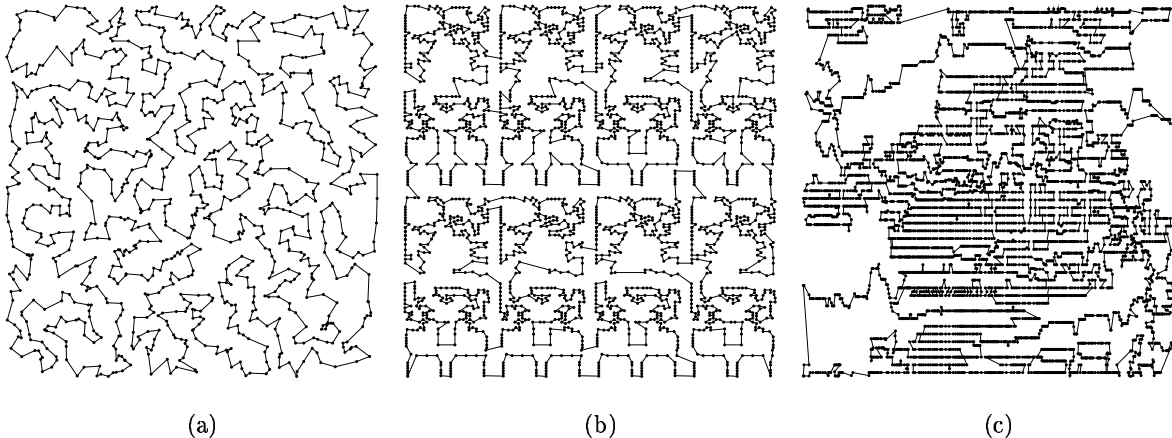


FIGURE 1. Three Euclidean TSPs with sample tours: (a) random 1000-point, (b) TSPLIB’s pr2392, (c) TSPLIB’s rl5934.

practice, 2-opt and 3-opt perform well, creating tours that are only a few percent above optimal. However, there are instances and starting tours for which 2-opt performs an exponential number of moves before halting.

Not all locally optimal solutions found by the local search algorithms are good solutions. Therefore, it may be useful to allow the local search algorithm to make a move “uphill” when it is stuck in a local minimum. However, there is a danger that the local search, after moving uphill, will eventually return to the original locally optimal solution. The tabu search algorithms keep some information about the earlier moves, thus preventing moves that will return to the previous locally optimal solution.

For a long time, the title of “best approximation heuristic” for TSP belonged to the Lin-Kernighan (LK) algorithm. Lin-Kernighan tries to overcome the tendency of local search algorithms to find local minima that are far from the optimal solution by using 3-opt together with a special Lin-Kernighan move. The Lin-Kernighan move, actually a 2-opt move that may increase tour length, allows uphill searching as in tabu search, but at less expense.

Iterated Lin-Kernighan (ILK) is an improvement on the standard Lin-Kernighan algorithm in which certain “double-bridge 4-opt” moves (sometimes called “kicks”) are performed when the LK local search stalls. (These type of kicks are chosen because they are difficult for LK to undo.) It has been found to be one of the most effective algorithms at finding high-quality solutions, even for very large problems [6].

A completely different approach to TSP approximation is used in genetic algorithms. The genetic algorithms start with some “population” of solution tours to the TSP problem, and generate new tours from this population with “mutation” and “crossover” operations that change tours or combine them to form new “child” tours. Then, some selection strategy is used to improve the population. However, the performance of the genetic algorithms is usually not competitive with methods such as LK. Partly, this might be because mutation and crossover operations usually do not take into account the structure of the traveling salesman problem.

## 2. THE ADAPTIVE MULTI-START APPROACH

**2.1. The “big valley.”** In a 1994 paper, Boese et al. [2] proposed a new TSP approximation algorithm. It is based on the observation that, in most cases, the locally optimal solutions—optimized by 2-opt local search in the paper—share many edges with each other and with the optimal solution. The authors called this observation the “big valley” hypothesis and supported it by several experimental results. Their work was restricted to 2-opt local minima and random problem instances, i.e., problems

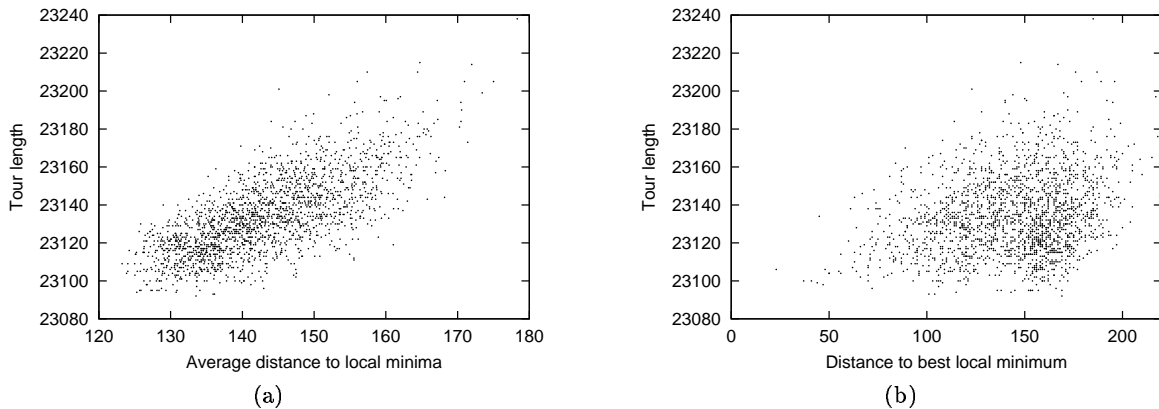


FIGURE 2. For 2000 local minima, (a) the average bond distance to the other local minima, and (b) the bond distance to best local minimum, for a random 1000-point instance.

where points are chosen uniformly at random on a square. The “bond distance” between two tours is defined as the number of edges in one tour not coinciding with an edge in the other. The authors also showed that the bond distance is within a factor of 2 of the 2-opt distance, which is the least number of 2-opt operations needed to get from one tour to another.

**2.2. Adaptive multi-start.** How does one exploit the big valley? Boese et al. present the adaptive multi-start (AMS) algorithm. First, they create a population of tours by running 2-opt local search on random initial tours. Then, repeatedly, they create a “child” tour by adding edges from the “parent” tours with probability proportional to the “fitness” of an edge, where the fitness of an edge is determined by the quality and number of the parent tours containing the edge. The edges that occur more often in the parent population have a better chance of appearing in the new tour. After the legal edges in the parent population are exhausted, the fragments of the partial tour are reconnected at random to form a complete tour, and 2-opt is run on the tour. If the child has a shorter tour length than one of the parents it replaces the worst tour in the population.

In this work, we improve on the method of Boese et al. by (i) choosing different population sizes and numbers of generations to improve quality of results; (ii) using another fitness function; (iii) implementing the child generation more efficiently and effectively; and (iv) using Lin-Kernighan local optimization. While the algorithm of Boese et al. performed only somewhat better than 2-opt, the version of AMS presented here finds tours of much better quality—in fact usually better than those of the iterated Lin-Kernighan algorithm (see §7).

**2.3. The big valley hypothesis for LK tours.** In our algorithm, we use Lin-Kernighan for local optimization on the tours. We now give some evidence that the big valley hypothesis still holds for LK-optimized tours.

For several Euclidean TSPs, we ran the LK algorithm on 2000 random initial tours and compared the 2000 locally optimal tours to each other and to the best locally optimal tours. Figures 2 and 3 show the results for 1000- and 2000-point random problems, and Figure 4 for a non-random 2392-point instance obtained from the TSPLIB [5] archive of problems.

Observe that in each case, the quality of local optima correlates with the distance between the local optima. That is, they cluster around the best local optima. Thus it is plausible that by finding locally optimized tours that are near many of the optima, we will find better tours.

The correlation is not as strong between the quality of local optima and distance to the *best* of the local optima found. Yet this should not be too discouraging: generally, a number of local optima are

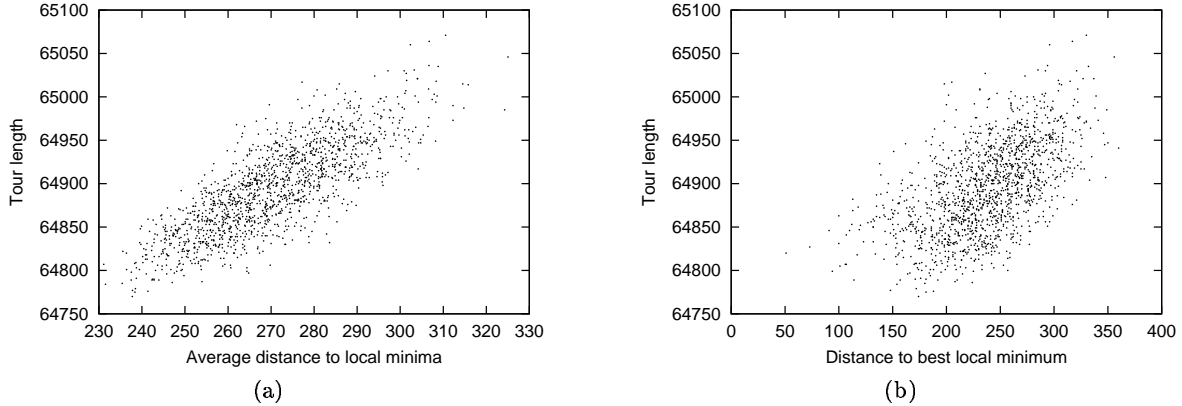


FIGURE 3. For 2000 local minima, (a) the average bond distance to the other local minima, and (b) the bond distance to best local minimum, for a random 2000-point instance.

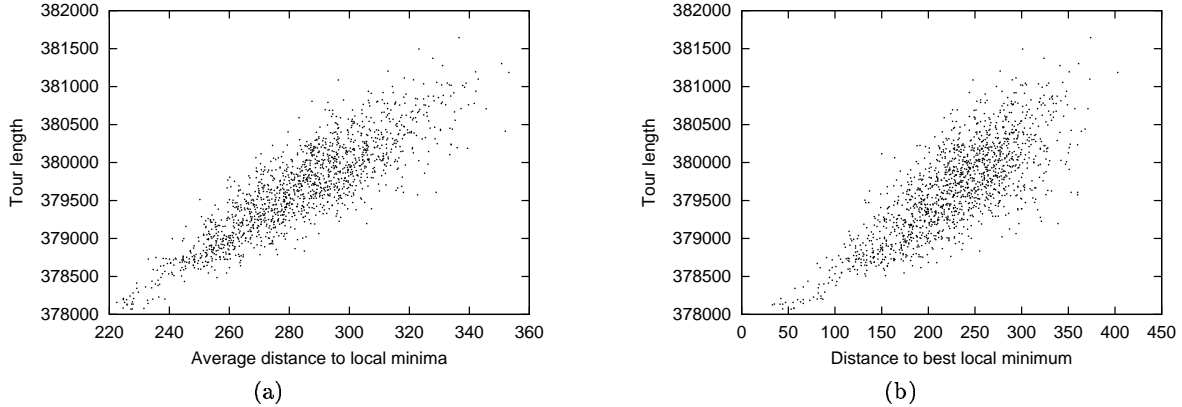


FIGURE 4. For 2000 local minima, (a) the average bond distance to the other local minima, and (b) the bond distance to best local minimum, for TSPLIB's pr2392 instance.

very good, though they may differ in a fair number of edges (indeed, a globally optimal tour might not be unique). This explains the lower bulge of points in the plot, which represent good tours distant from the best local optimum.

The plots give credence to the big valley hypothesis for LK-optimized tours, and suggest that the AMS approach is compatible with LK-optimized tours.

### 3. THE ALGORITHM

**3.1. General structure.** Figure 5 provides a high-level system diagram for our implementation of the AMS algorithm. The algorithm is genetic in nature: at every generation, a randomized cross-breeding operation is applied to the parent tours to create a child tour that contains many of the same edges as the parents. This child tour is subsequently locally optimized and added to the population, provided it is good enough. The fundamental difference between AMS and traditional genetic approaches to the TSP is that AMS repeatedly cross-breeds the *entire* population to create a single child.

The algorithm operates in two phases. In the first phase, it generates a set of initial tours using a tour construction heuristic (such as random, greedy, or nearest-neighbor) and each of these parent tours is locally optimized. The resulting tours are the initial population.

In the second phase, we begin by collecting the union of all the edges which appear anywhere in the population into an “edge pool,” and calculate a fitness value for each edge in the pool based on characteristics of the edge itself and the tours containing that edge. (Note this differs from traditional genetic approaches, which generally assign fitness to individuals in the population, not their components.) Next, we repeatedly choose edges from the pool at random with probability proportional to their fitness values and add them to a child tour we are “growing,” omitting any edges that would create an illegal tour, until no more edges may be added. Because we may exhaust all the edges in the pool and still be left with an incomplete child tour, we may also need to add some new edges to connect the tour fragments. This completion can be accomplished using an edge-length based heuristic such as nearest-neighbor or greedy, or with a simple-minded random selection. Finally, the complete child is passed through the local optimizer, and compared to the population. If the new tour is shorter than the worst tour in the population, it will replace it. If desired, more children can be generated from the same pool. The new population then becomes the basis for the subsequent generation. As we will see, the population eventually reaches a point where no further progress is made, and we stop the algorithm (see §6).

The algorithm described has a large number of parameters that can be adjusted to affect its behavior, such as the choice of tour construction heuristic, the size of the population, the number of children built at each generation, the child completion algorithm, and the choice of fitness function. These options are investigated further in §§5–6.

**3.2. Comparison to Boese et al. algorithm.** At a high level, our algorithm is similar to the algorithm described by Boese et al. However, there are some significant differences. For example, the Boese et al. algorithm spends about half of its time building the initial population and the other

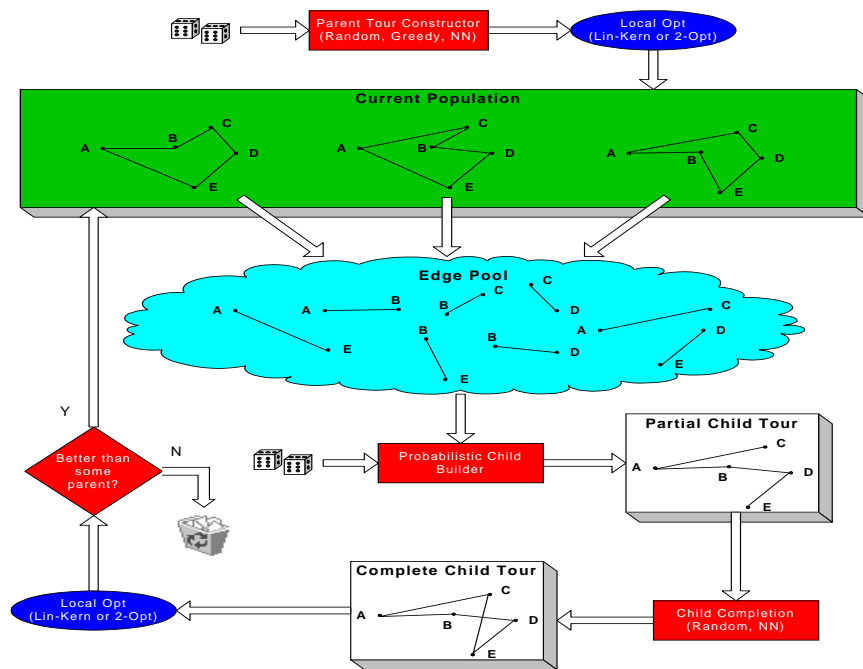


FIGURE 5. Structure of the algorithm.

half generating child paths, while our algorithm, properly adjusted, spends less time on the initial population and focuses on a larger number of generations (a ratio of perhaps 1:1000 instead of 1:1). Our algorithm uses a cost-conscious (and in fact faster) heuristic for child tour completion instead of random search. It uses Lin-Kernighan instead of 2-opt for local optimization, which (not surprisingly) produces much better results. It also runs local search on each child path only once, rather than relying on non-determinism in the local search to provide different optimized children. Finally, we use a more sophisticated fitness function for assigning edge probabilities. The influence of these choices is discussed in §§5–6.

#### 4. IMPLEMENTATION

**4.1. Tools.** We implemented the algorithm described in §3 in about 2500 lines of C++ code. The implementation of the Lin-Kernighan local search algorithm and the parent tour construction heuristics was taken from the CONCORDE [1] library of Applegate et al., a freely available collection of TSP-solving utilities. This library achieves efficient performance by exploiting  $K$ -d tree data structures, which provide efficient ways to query node proximity information about a graph. Unfortunately, the running time complexity for  $K$ -d trees remains unknown, though empirical evidence suggests they are quite fast [3, 4]. To make matters worse, LK itself has questionable running time. However, empirical studies have provided extensive evidence that for most realistic problems, especially Euclidean ones, the running time for LK local search is about  $O(n \log n)$  for implementations using fast data structures such as  $K$ -d trees [6].

In order to study the effect of various parameters on the behavior of the algorithm, our implementation collects a variety of metrics throughout its run that provide insight into its operation. These include some obvious indicators, such as the population tour lengths and the child tour lengths, but also some less obvious ones such as the distribution of edge fitness values in the edge pool and in the generated children. One metric that proved particularly revealing was the “diversity” of the population, defined to be  $(\#(\text{unique edges in pool}) - n)/(n(p - 1))$ , where  $p$  is the size of the population. That is, diversity expresses the amount of similarity between the edge content of the tours in the population, scaled so that the diversity is zero when all the tours in the population are identical, and the diversity is one when all the tours are completely disjoint.

**4.2. Complexity.** The complexity for our implementation is  $O(pS + gc(np \log np + S))$ , where  $g$  is the number of generations,  $c$  is the number of children built per generation, and  $S$  is the running time for the local search.

The  $O(pS)$  term is contributed by the first phase, which is dominated by the local search done for each of the initial parent tours. The running time for the second phase is linear in the number of generations and the number of children created per generation. The time complexity of creating each child tour depends on the running time for the probabilistic child builder (which we will show runs in time  $O(np \log np)$  time) and the time  $O(S)$  required to locally optimize the child.

The time complexity for child completion is  $O(f)$  for random completion or  $O(f^2)$  for the nearest-neighbor completion heuristic, where  $f$  is the number of tour fragments in the child. The number of fragments varies for each child, but must always be less than  $n$  (it is typically less than  $0.1n$ ). We implemented an adaptive completion algorithm which selects random or nearest-neighbor completion “on the fly” based on the value of  $f$ , such that the time complexity for the completion routine never exceeds  $O(np \log np)$ . (In practice it almost always chooses nearest neighbor.)

**4.3. Data structures.** We will now briefly discuss the major data structures utilized to efficiently implement the AMS modules and achieve the stated running time complexity for the child builder. The edge pool was implemented as a hash table of edges, allowing us to collect a union of all the edges in the population in  $O(np)$  time. This union can be performed from scratch at every generation, or as an optimization, can be maintained over the entire run, updating it in  $O(n)$  time each time we add a new tour to the population. Note that at any given generation, the number of edges in the pool will

range anywhere from  $n$  (diversity 0) to  $np$  (diversity 1). We use  $np$  to bound this quantity, although in reality it is much closer to  $n$  for most runs.

Once we’ve collected all the edges, we perform an  $O(np)$  pass over the edges and assign fitness values based on the selected fitness function (see §5). The probabilistic child builder repeatedly selects edges from the pool at random with probability directly proportional to their assigned fitness values, and adds them to the child if they are legal tour edges. Edge selection is implemented by choosing a random number between 1 and the sum of the fitness values for the remaining active edges, finding the edge which corresponds to the chosen value, and marking it as inactive. A naïve implementation might store the edges in an array and perform a worst-case  $O(np)$  search to find the corresponding edge (iterate along the list, decrementing the random value by the fitness of each edge encountered until the value reaches zero). Our implementation maintains a binary search tree with the edges at the leaves and the internal nodes holding information about the total fitness sum of each subtree. This allows us to find the selected edge with a  $O(\log np)$  traversal down the tree, and mark it inactive with another  $O(\log np)$  traversal back up the tree to the root.

After an edge has been selected, we must determine whether it can legally be added to the child tour, and if so, add it. We use a data structure inspired by the traditional linked-list union-find data structure, where the nodes are the graph nodes and the groups are the various tour fragments in the partial tour. Our implementation is streamlined for this particular application so that it performs the check-and-update operation in constant time. The list of nodes is doubly linked, so the check that each endpoint of the edge has degree less than two is a simple matter of counting the number of valid links at each endpoint. Cycle detection is performed by checking whether the newly added edge connects two nodes that are members of the same group and whether the group size is less than  $n$ . The key insight to performing a group merge in constant time is that only the nodes at the ends of each fragment need accurate information about which group they belong to, so we need update just these on a merge, instead of all the nodes in the merged group. Because there are up to  $np$  edges in the edge pool and it takes  $O(\log np) + O(1)$  time to process each edge when it is selected, the running time complexity for the child builder to exhaust the edge pool is  $O(np)(O(\log np) + O(1)) = O(np \log np)$ .

**4.4. Breakdown of CPU usage.** Figure 6 presents a breakdown of the CPU time for a short run of AMS (50 generations of a population of 10 on a 1000-point problem). Clearly, the local search module dominates the execution time, occupying over 76% of the CPU cycles even without including the initialization of Lin-Kernighan and the  $K$ -d tree. While the run analyzed is somewhat shorter than a usual run of AMS (practical limitations prevented us from running more generations in the application profiler), a simple analysis suggests the CPU fraction dedicated to local search grows to over 78% for longer runs, where the effects of executing start-up code are amortized over more generations. It is interesting to note that child tour construction was the second most time-consuming operation, as suggested by the running time analysis.

## 5. FITNESS FUNCTIONS

The AMS algorithm uses a fitness function to decide the probability of picking each edge. The choice of this fitness function greatly influences the algorithm’s performance, both in terms of tour quality and running time. Since there is no single obvious choice, we describe some possibilities and their effectiveness in practice.

**5.1. Design considerations.** We want the fitness of an edge to correlate positively with factors that contribute to good tours. Possible factors are edge length, the lengths of the population tours that contain the edge, and the edge’s “popularity,” or the number of parents containing the edge.

The fitness functions are parameterized so it is possible to strike a favorable balance among the factors. The functions are chosen to be largely independent of input characteristics such as problem size; it would be unsatisfying if the user were forced to tune the parameters for each instance to be solved.

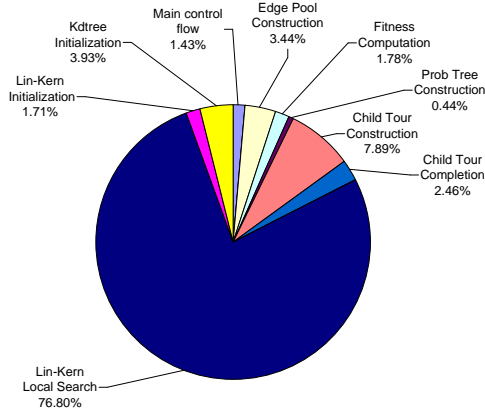


FIGURE 6. Breakdown of run-time for AMS.

**5.2. Alternative fitness functions.** We write  $|e|$  or  $|tour|$  to denote the length of an edge or of a tour in the population. The fitness function used by Boese et al. [2] was simply the sum of the inverse tour lengths of the parents containing the given edge  $e$ :

$$Fit_B(e) = \sum_{tour \ni e} \frac{1}{|tour|}.$$

(Boese et al. used an additional computation to choose probabilities from fitness values.)

This fitness function favors edges that are “popular,” and edges that appear in short parent tours. Both measures contributing to fitness in  $Fit_B$  are “global” in nature. To explore the effect of considering local factors, we added a term dependent on the length of the edge:

$$Fit_1(e) = \sum_{tour \ni e} \left( \frac{\alpha}{n|e|} + \frac{1-\alpha}{|tour|} \right),$$

where the parameter  $\alpha \in [0, 1]$  controls whether global ( $\alpha = 0$ ) or local ( $\alpha = 1$ ) considerations dominate.

By experimenting with inputs and values for  $\alpha$ , we determined that  $\alpha \approx 0.9$  performed best. This is disappointing because when  $\alpha$  is so large, we are mostly ignoring parent tour quality, and thus at least partially defeating the genetic nature of the algorithm. Further, it is not at all clear that inverse proportionality is the best way to favor attributes.

As an alternative, we also considered making the dependence of edge length and parent quality linear:

$$Fit_2(e) = \sum_{tour \ni e} \left[ \alpha \left( 1 - \frac{|e|}{\text{diam } G} \right) + (1-\alpha) \left( 1.5 - \frac{|tour|}{L} \right) \right],$$

where  $\text{diam } G$  is the graph diameter and  $L$  is the average length of the tours in the initial parent pool, after the local search has been applied. (Here we assume that each parent will have lengths between 50% and 150% of  $L$ , a safe assumption when the parents are optimized with Lin-Kernighan.) For this fitness function,  $\alpha \approx 0.5$  was best, though  $Fit_2$  never performed better than  $Fit_1$ .

**5.3. The AMS fitness function.** More experimentation revealed that popularity was dominating *both* edge length and parent quality. This motivated incorporating popularity explicitly,

$$Fit_3(e) = \alpha \frac{|e_{\max}| - |e|}{|e_{\max}| - |e_{\min}|} + (1-\alpha) e^{\beta(\text{pop}(e)/p-1)} \frac{|tour_{\max}| - \min_{tour \ni e} |tour|}{|tour_{\max}| - |tour_{\min}|},$$

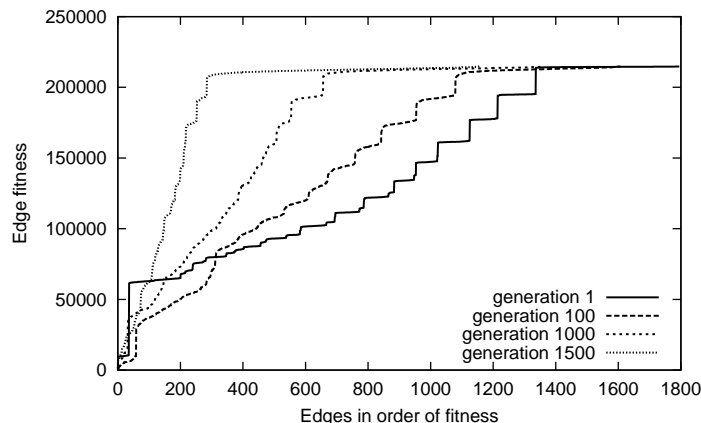


FIGURE 7. Fitnesses of edges in the pool, sorted in increasing order, at several generations.

where  $p$  is the number of parents,  $\text{pop}(e)$  is the popularity of the edge,  $e_{\max}$  and  $e_{\min}$  are the longest and shortest edges in the edge pool, and  $\text{tour}_{\max}$  and  $\text{tour}_{\min}$  are the longest and shortest parent tours. The parameter  $\beta \geq 0$  controls the importance of popularity. This function had more reasonable optimal values  $\alpha \approx 0.05$ ,  $\beta \approx 1$  (see §6). This gives a reasonable but not dominating influence to popularity, and a small but not insignificant influence to edge length.

$\text{Fit}_3$  in general out-performs the other functions described. It is the fitness function we use throughout the rest of the paper. Figure 7 shows, for a typical 1000-point problem and a population of 10, the fitnesses of the edges in the edge pool at several stages of the algorithm, sorted in increasing order by fitness. Thus the upper “plateau” on the generation 1 graph shows that 400 edges have nearly maximal fitness. The “stair steps” below indicate jumps in fitnesses at different popularity levels. One can see that after many generations, a larger and larger portion of the edges have high fitness. (Note the fitnesses are scaled at each generation, so only the shapes, not the absolute values, of the curves are comparable.)

## 6. BEHAVIOR OF THE ALGORITHM

**6.1. A sample run.** Figure 8 shows a typical execution of the algorithm, in this case a 1000-point random instance. The “new child” lengths are very noisy since the result of local optimization is not reliably good. However, the best and worst tours in the population improve steadily as long as the diversity of the population is high. After about 1300 generations, the population “converges”: the best and worst tour lengths coincide, the noise in child generation drops to near zero, diversity drops to nearly zero, and the best tour length stops improving.

The correlation between diversity and convergence indicates interesting similarities between AMS and simulated annealing (SA). The diversity of the population in AMS acts like temperature in SA; when diversity is high, the algorithm is exploring many widely separated regions of the search space. But when diversity is low, the algorithm “homes in on” the best solution in the region it has selected. However, the mechanism for determining the “cooling schedule” differs between AMS and SA. Whereas SA temperature drops according to a user-specified schedule, AMS diversity drops automatically.

**6.2. Selection of parameters.** The parameters, especially population size, influence the diversity schedule. With only a few parents (less than 5, say), AMS can find a solution very quickly, but it is not very near to optimal. With many parents (20 or more), AMS takes quite a bit longer to reach convergence, but the resulting tour is very good. This behavior is illustrated in Figure 9. (Note that

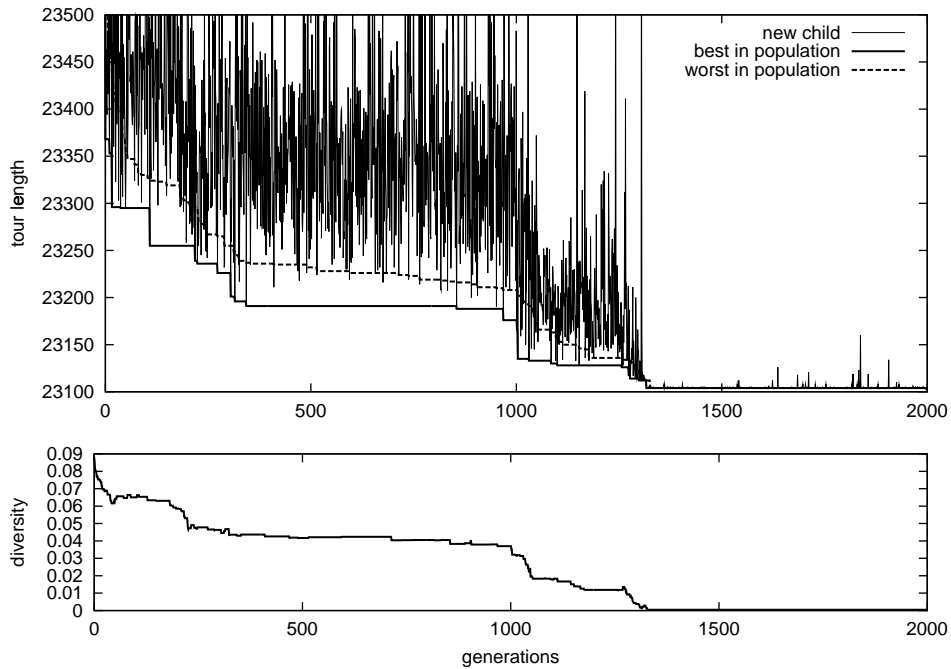


FIGURE 8. A typical run on a 1000-point random problem: best, worst, and new child optimized tour lengths (above) and diversity (below).

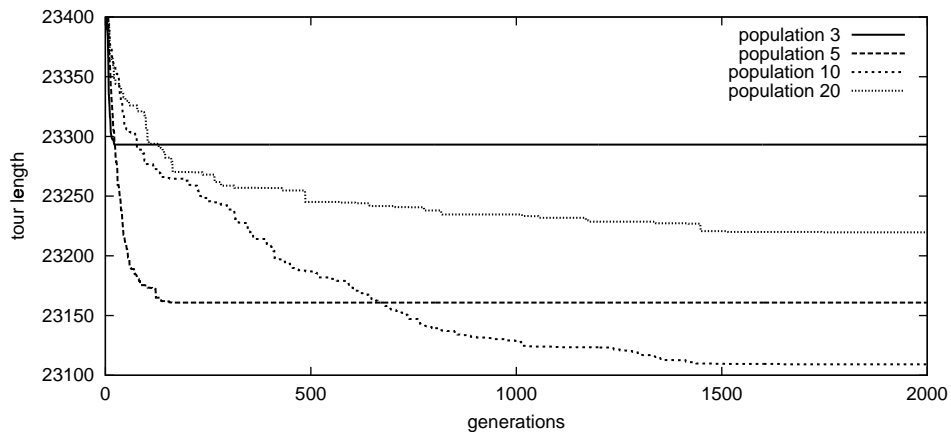


FIGURE 9. Best tour lengths for a 1000-point random problem (average of 10 runs) for various sizes of the population.

in the figure the population of 20 has not converged in the 2000 generations shown. It ultimately reaches tour qualities slightly better than those of the population of 10.)

The value of  $\alpha$  also influences the performance. In general higher  $\alpha$  makes the addition of unpopular or poorly performing edges more likely, raising diversity and slowing convergence. Picking  $\alpha$  slightly larger than zero, say  $\alpha \approx 0.05$  seems to work well—see Figure 10. Performance is not as sensitive to  $\beta$ ; values near 1 work well.

We find the choice of construction heuristic and child completion heuristic have a moderate effect on the algorithm. Generally, random initial tours work better than nearest neighbor or greedy, and

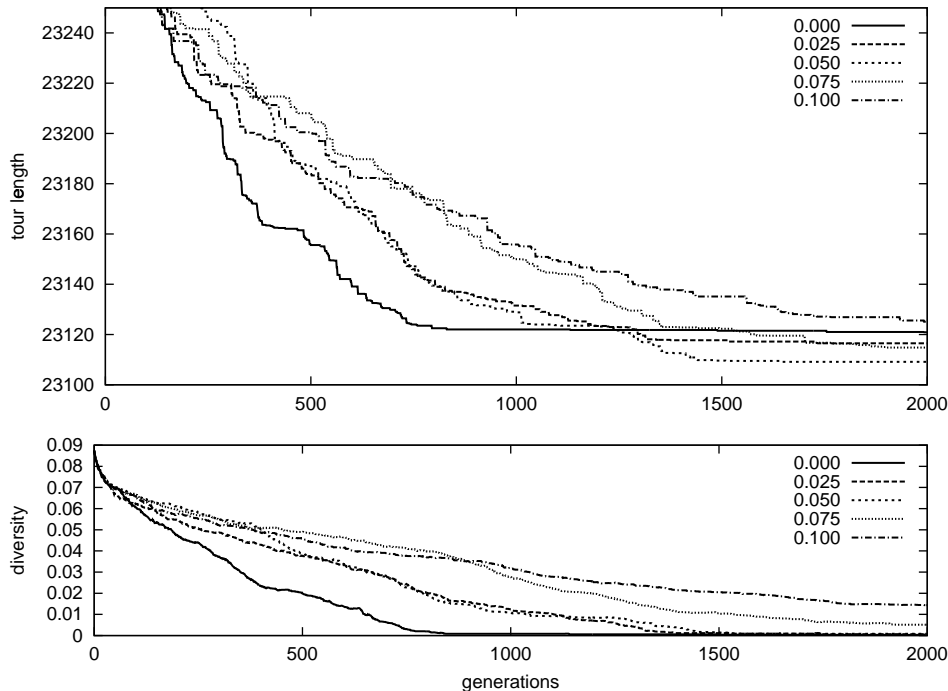


FIGURE 10. Best tour lengths (above) and diversity (below) for 1000-point random problem (average of 10 runs) for various values of  $\alpha$ .

nearest-neighbor tour completion performs better than random tour completion. This is not surprising, since we want as much diversity as possible at the start, and want each child tour to be short. (Boese et al. claim that the choice of child completion heuristic has little effect. This may be a feature of their 2-opt local optimization, or it may be that their experiments involved too few generations to show the difference.)

The number of children per generation does not seem to have a substantial effect on tour quality and only a minor one on speed (after all, the same number of local searches are performed). For simplicity we use one child per generation.

## 7. PERFORMANCE AND COMPARISONS

All running times given in this section were measured on 200 MHz Pentium Pro workstations running the Solaris operating system.

**7.1. Does breeding help?** We can first check if the “adaptive” breeding of good tours is worthwhile by comparing to “unadaptive” multi-start: instead of breeding to get new tours, say we pick them at random, then apply the same local optimization. Figure 11 is a representative comparison showing the best tours found in 5000 local optimizations on a random problem instance. Clearly, the adaptive method is *much* faster at finding good tours.

**7.2. Comparison to iterated Lin-Kernighan.** The CONCORDE library provides an implementation of the iterated Lin-Kernighan (ILK) algorithm. As the algorithm is one of the most successful at finding high-quality tours [6], it is an obvious candidate for comparison with AMS. Moreover, since AMS uses precisely the same LK local search implementation as ILK, runtime comparisons can be performed easily and fairly.

ILK runs quickly on problems of several thousand nodes, applying thousands of “kicks” in a few minutes. However, like LK itself, it eventually gets trapped near a local minimum from which even the kicks find no escape. Thus, for best-quality results, it should be run multiple times with random starting tours. Figure 12 shows both single and multiple runs of ILK on a 1000-point random problem. Both intermediate-quality and final “best” tours are shown for several single runs. Also, to give an indication of the difficulty of finding near-optimal tours, the final results of a few repeated ILK runs—sometimes taking tens or hundreds of times longer—are shown.

For comparison, a sample of runs from AMS is also shown, in this case for a population size of 10. While AMS is clearly much slower than a single run of ILK, it seems to reliably find better tours: many single ILK runs are never able to do as well as a typical single AMS run. If we take into account the time needed to repeat ILK until a high-quality tour is found, we see AMS can be significantly faster. Generally, we see AMS is much more stable in its performance, a property we would expect

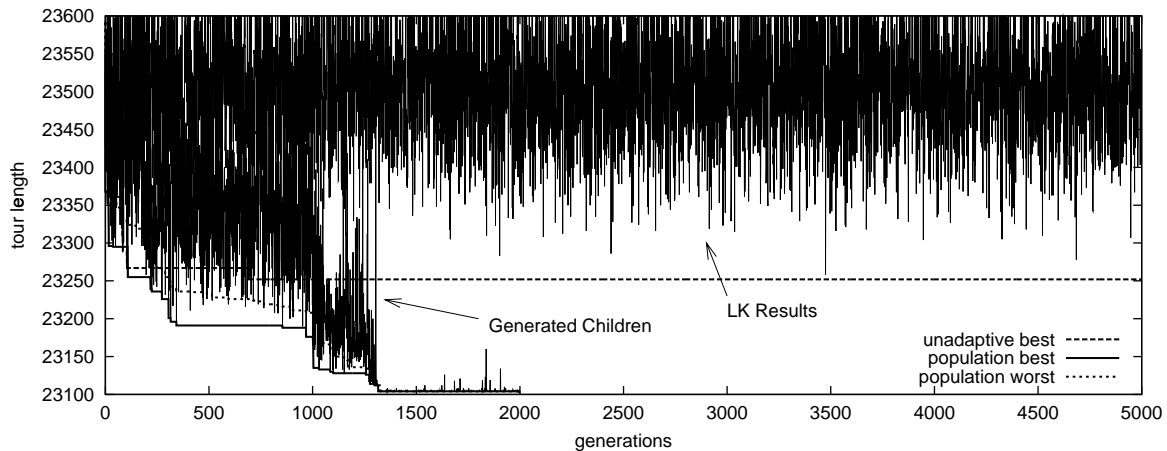


FIGURE 11. A typical run on a 1000-point random problem, and an “unadaptive” repetitive LK run.

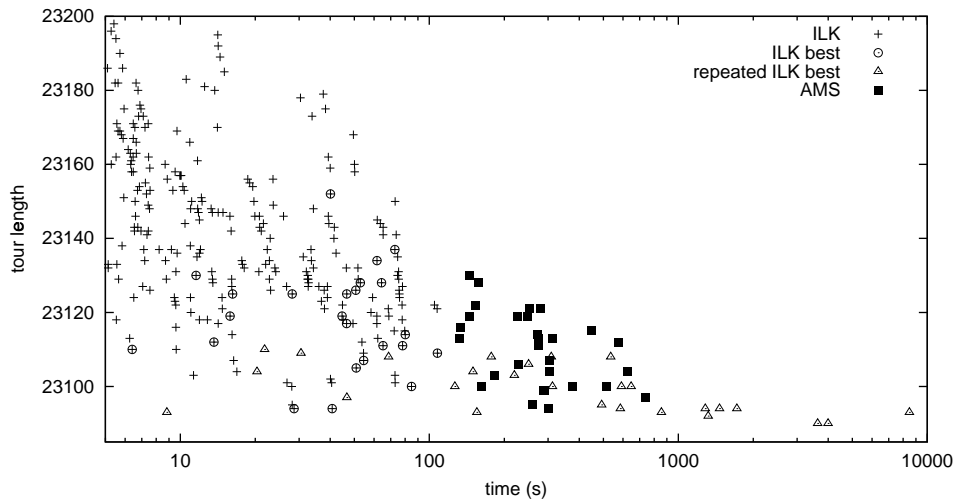


FIGURE 12. Quality of ILK and AMS tours versus CPU time (on logarithmic scale) to find them.

TABLE 1. Performance on TSPLIB problems.

name (size)	pop. size	avg. length	best length	optimal	best % excess	time (min)
gr666	10	294502	294476	294358	0.040%	2
	15	294476	294476	294358	0.040%	5
	20	294478	294358	294358	0.000%	12
u1060	10	224254	224132	224094	0.017%	9
	15	224168	224160	224094	0.029%	20
	20	224131	224115	224094	0.009%	62
pcb1173	10	56910	56894	56892	0.004%	2
	15	56894	56892	56892	0.000%	5
	20	56893	56892	56892	0.000%	9
d2103	10	80649	80512	80450	0.077%	7
	15	80534	80455	80450	0.006%	16
	20	80518	80457	80450	0.009%	61
pr2392	10	378097	378056	378032	0.006%	6
	15	378064	378033	378032	0.000%	14
	20	378055	378032	378032	0.000%	43
rl5934	10	556800	556403	556045	0.064%	146

TABLE 2. Comparison of AMS and ILK on TSPLIB problems.

name (size)	AMS avg.	ILK avg.	AMS best	ILK best	AMS time avg. (min)	ILK time (min)
gr666	294478	294411	294358	294358	12	17
u1060	224131	224144	224115	224121	62	67
pcb1173	56893	56923	56892	56892	5	7
d2103	80534	80643	80455	80563	16	17
pr2392	378064	378446	378033	378107	14	17
rl5934	556800	557345	556403	556451	146	180

from its population-based approach. A larger population, say of size 20, both improves tour quality and reliability, at the expense of a factor of 5–10 in running time.

**7.3. Non-random problems.** The previous results are for randomly chosen points. The TSPLIB archive [5] has many more interesting point configurations, taken from applications ranging from geography to printed-circuit board layouts. For many problems, provably optimal solutions (found by branch and bound methods, for example) are known. Table 1 lists the performance of AMS on six differently structured problems, for various population sizes. (The “optimal” value listed for d2103 is actually the best known upper bound. Also, gr666 uses a non-Euclidean metric.) Results were obtained by averaging ten (for population 10) or five (for populations 15 and 20) runs on each problem. We see that increasing the population size improves the average quality of the tour, and increases the CPU time.

Table 2 compares this performance with that of ILK. In each case, repeated ILK was run ten times, each with the CPU time indicated, which is somewhat more than the average time taken by AMS. The average and best result from the ten (or five in some AMS cases) is listed. We see that for each problem the best AMS result is at least as short as the best ILK result, and often significantly (considering the closeness to optimality) shorter. In all but one case, AMS also performs better on average, particularly for the largest three problems, as summarized in Figure 13.

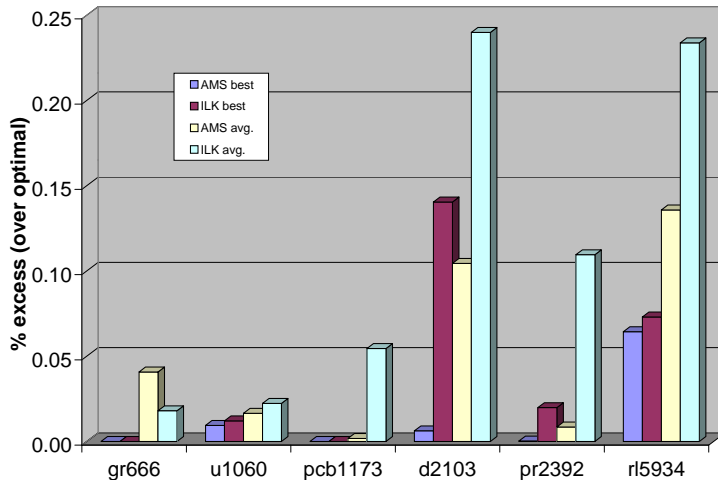


FIGURE 13. ILK and AMS tours from Table 2.

## 8. FUTURE WORK

**8.1. Improving the algorithm.** The number of possible variations on AMS is dizzying. Not only can parameters, such as  $\alpha$  or population size, be varied, but changes could be made in the fitness function, the local search, or the scheme for picking edges and completing children; it would be foolish to think the experiments described more than scratch the surface of what is possible with an algorithm of its type.

A few ideas for improving tour quality seem promising. Looking at a typical run, such as in Figure 8, it is clear that in the final stages of execution, the number of edges in the pool is small. Yet it is also at this stage at which the hard-to-find near-optimal tours are attained. It may be possible, after the edge pool has diminished to nearly its minimum, to implement an exhaustive search that would run in a reasonable amount of time. In this way, we would hope to reap the most reward from the population we have worked so hard to breed.

Another variation would be to increase the size of the population as diversity lowers. Dynamically changing population size may allow more direct control of the algorithm and perhaps yield better results.

Finally, one could obviously use a more powerful local search, such as ILK. This would of course be terribly expensive, but this could be offset in another way, perhaps by parallelizing the algorithm.

We mention two ideas for improving speed. Most of the CPU time is spent in the local search. A few experiments indicate that there is a moderate correlation between the lengths of unoptimized children and their final optimized lengths, which suggests that perhaps not much quality would be lost by discarding longer children before they are optimized.

Another possible speed-up would be to use multiple smaller populations simultaneously. This could improve speed since small populations find better tours faster. The good descendents of these small populations could then be merged into a new population, and breeding could continue.

**8.2. Parallelizing AMS.** It would be possible to parallelize the child generation phase of the algorithm, so that many children are generated on different processors concurrently. Experiments indicate that generating multiple children per generation, especially for large populations, has similar effect to generating one at a time, so the algorithm should behave similarly. Since the local search is expensive relative to other operations in the algorithm, this would have a direct speed benefit, even in the simplest implementation where each processor finishes a child, then waits for the next generation.

**8.3. Extension to other problems.** While this paper has focused on the Euclidean TSP, the AMS algorithm could certainly be applied to the general (symmetric) TSP problem. Since AMS is based on the Lin-Kernighan local search, which is quite robust, it is plausible that the adaptive multi-start approach we have outlined could make headway on many types of traveling salesman problems.

## 9. CONCLUSION

Adaptive multi-start is a promising approach to finding high-quality solutions to the Euclidean TSP and perhaps more general TSPs. It combines good qualities of local search and genetic algorithms, while avoiding many of the pitfalls of each. The algorithm cannot easily get stalled at a poor local minimum, as local searches can, because an entire population of tours is maintained. New tours are constructed through a heuristic that exploits structural properties of the search space. Finally, the richness of options in designing such an algorithm means improvements are likely.

## REFERENCES

- [1] David Applegate, Robert Bixby, Vasek Chvátal, and William Cook, CONCORDE, a computer code for the traveling salesman problem (preliminary version, August 27, 1997), available at:  
<http://www.caam.rice.edu/~keck/concorde.html>
- [2] Kenneth D. Boese, Andrew B. Kahng, and Sudhakar Muddu, "A new adaptive multi-start technique for combinatorial global optimizations," *Operations Research Letters* **16** (1994), pp. 101–113.
- [3] Jon Louis Bentley, "K-d trees for semidynamic point sets," pp. 187–197 in *Proceedings of the Sixth Annual ACM Symposium on Computational Geometry*, ACM, New York, 1990.
- [4] Jon Louis Bentley, "Fast algorithms for geometric traveling salesman problems," *ORSA Journal on Computing* **4** (4) (1992), pp. 387–411.
- [5] Robert Bixby and Gerd Reinelt, TSPLIB, a library of traveling salesman and related problem instances (February 17, 1995 version), available at:  
<http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB>
- [6] David S. Johnson and Lyle A. McGeoch, "The traveling salesman problem: a case study," pp. 215–310 in *Local Search in Combinatorial Optimization*, Emile Aarts and Jan Karel Lenstra, eds., John Wiley, New York, 1997.