A Comparison of Genetic Sequencing Operators

T. Starkweather, S. McDaniel,
K. Mathias, D. Whitley
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523

C. Whitley
Mechanical Engineering Dept.
Colorado State University
Fort Collins, CO 80523

Abstract

This work compares six sequencing operators that have been developed for use with genetic algorithms. An improved version of the edge recombination operator is presented, the concepts of adjacency, order, and position are reviewed in the context of these operators, and results are compared for a 30 city "Blind" Traveling Salesman Problem and a real world warehouse/shipping scheduling application. Results indicate that the effectiveness of different operators is dependent on the problem domain; operators which work well in problems where adjacency is important (e.g., the Traveling Salesman) may not be effective for other types of sequencing problems. Operators which perform poorly on the Blind Traveling Salesman Problem work extremely well for the warehouse scheduling task.

1 INTRODUCTION

Gil Syswerda [5] conducted a study in which "edge recombination" (a genetic operator specifically designed for the Traveling Salesman Problem) performed poorly relative to other operators on a job sequence scheduling task. While the population size used by Syswerda was small (30 strings) and good results were obtained on this problem using mutations alone (no recombination), Syswerda's discussion of the relative importance of position, order and adjacency for different sequencing tasks raises an issue that has not been adequately addressed. Researchers, including ourselves [8] [10], seem to tacitly assume that all sequencing tasks are similar and that one genetic operator should suffice for all types of sequencing problems.

This paper compares six different operators on two different sequencing tasks. The comparisons include an improved version of "edge recombination." The problems are a 30 city "Blind" Traveling Salesman Problem and a 195 element sequencing task for a real world warehouse/shipping scheduling application. Our experiments show that different operators are better suited to different kinds of sequencing tasks. Edge recombination is only roughly competitive with operators such as PMX on the warehouse scheduling problem and the resulting search is an order of magnitude slower than operators which stress relative order as opposed to adjacency.

The genetic algorithm used in these experiments is *GENITOR,* which was developed at Colorado State University [7]; our results also suggest that *GENITOR* is a key part of our improved performance on the Traveling Salesman Problem. It uses a one-at-a-time replacement paradigm in which only one pair of strings reproduces during any given generation and only one offspring is generated. The new offspring replaces the worst string in the population rather than one of its parents. This ensures that the best string found so far will never be replaced in the population.

We do not offer comparative results in this paper to other approaches for the Traveling Salesman Problem. We do note, however, that *GENITOR* is really solving a more difficult version of the Traveling Salesman Problem than that solved by most other algorithms. Most algorithms use local edge information to do local improvements; *GENITOR* uses only the overall value of the total sequence. David Goldberg [2] points out that genetic algorithms are actually solving a "Blind" Traveling Salesman Problem. The Blind Traveling Salesman Problem is interesting because for certain types of sequence optimization tasks local information is not readily available. In a fair comparison on the "classic" Traveling Salesman Problem, the genetic algorithm would be allowed to also use local information about edge costs. In other words, parent and offspring tours could be improved by using local information. A group of European researchers [6] have used this type of strategy and have achieved impressive results on classic Traveling Salesman Problems with up to 666 cities. They found that by combining genetic operators and local search methods developed by Lin/Kernigan they obtained superior results to 5 other approaches on 8 different problems ranging in size from 48 to 666 cities (median size 318). Thus, it appears likely that genetic methods could become a basic part of the tool kit for solving Traveling Salesman Problems and that the genetic algorithm could be viewed as a structure in which to organize and apply many of the tools that already exist. The reader interested in comparison tests on the classic Traveling Salesman Problem should consult the results of Ulder et al. [6].

2 EMPHASIS OF DIFFERENT SEQUENCING OPERATORS

Six genetic sequencing operators are compared in this study: improved edge recombination, order crossover, variants proposed by Syswerda [5] which we shall refer to as order crossover #2 and position crossover, PMX crossover, and cycle crossover.

2.1 ENHANCED EDGE RECOMBINATION

The edge recombination operator is different from other genetic sequencing operators in that it emphasizes adjacency information instead of the order or position of the items in the sequence. The "edge table" used by the operator is really an adjacency table listing the connections into and out of a city found in the two parent sequences. The edges are then used to construct offspring in such a way that we avoid isolating cities or elements in the sequence.

For example, the tour [b a d f g e c j h i] contains the links [ba, ad, df, fg, ge, ec, cj, jh, hi, ib], when one considers the tour as a hamiltonian cycle. In order to preserve links present in the two parent sequences a table is built which contains all the links present in each parent tour. Building the offspring then proceeds as follows: (1) Select a starting element. This can be one of the starting elements from a parent, or can be chosen from the set of elements which have the fewest entries in the edge table. (2) Of the elements that have links to this previous element, choose the element which has the fewest number of links remaining in its edge table entry, breaking ties randomly. (3) Repeat step 2 until the new offspring sequence in complete.

An example is given of edge recombination is given in figure 1. Suppose element a is selected randomly to start the offspring tour. Since a has been used, all occurrences of a are removed from the right-hand side of the edge table. Element a has links to elements b, f, and j. Elements b and f both have 3 links remaining in their table entries, but element j has only 2 links remaining. Therefore, j is selected as the next element in the offspring, and all occurrences of j are removed from the right-hand side of the edge table. Element j has links to i and h, both of which have 3 links remaining. Therefore, one of these elements are selected at random

```
Parent 1:     a b c d e f g h i j
Parent 2:     c f a j h d i g b e

Offspring:    a j i h d c f e b g

Edge table:   city   links
              a      b, f, j
              b      a, c, g, e
              c      b, d, e, f
              d      d, f, b, c
              e      d, f, b, c
              f      e, g, c, a
              g      f, h, i, b
              h      g, i, j, d
              i      h, j, d, g
              j      i, a, h
```

Figure 1: Edge Recombination

(element i in figure 1) and the process continues until the child tour is complete.

When the edge recombination operator was first implemented, we realized that it had no active mechanism to preserve "common subsequences" between the 2 parents. We have developed a simple solution to this problem. When the "edge table" is constructed, if an item is already in the edge table and we are trying to insert it again, that element of the sequence must be a common edge. The elements of a sequence are stored in the edge table as integers, so if an element is already present, the value is inverted: if A is already in the table, change the integer to -A. The sign acts as a flag. Consider the following sequences and edge table: [a b c d e f] and [c d e b f a].

```
a: b, -f, c             d: -c, -e
b: a, c, e, f           e: -d, f, b
c: b, -d, a             f: e, -a, b
```

The new edge table is the same as the old edge table, except for the flagged elements. One of three cases holds for an edge table entry. 1) If four elements are entered in the table as connections to a given table entry, that entry is not part of a common subsequence. 2) If three elements are entered as connections to a given table entry, then one of the first two elements will be negative and represents the *beginning* of a common subtour. 3) If only two elements are entered for a given table entry, both must be negative and that entry is an internal element in a common subsequence. Giving priority to negative entries when constructing offspring affects edge recombination for case 2 only. In case 1, no connecting elements have negative values, and in case 3 both connecting elements are negative, so edge recombination behaves just as before. In case 2, the negative element which represents the start of a common subtour is given

```
Parent 1:     a b c d e f g h i j
Parent 2:     c f a j h d i g b e
Cross Pts:      *         *
Offspring:    f g a j h d i b c e
```

Figure 2: Order Crossover #1

```
Parent 1:     a b c d e f g h i j
Parent 2:     c f a j h d i g b e
Cross Pts:        * *     *   *
Offspring:    a j c d e f g h i b
```

Figure 3: Order Crossover #2

first priority for being chosen. Once this common subsequence is started, each internal element (case 3) of the sequence has only one edge in and one edge out, so it is guaranteed that the common sections of the sequence will be preserved. The implementation of this idea (along with better mechanisms to ensure random choices when random choices are indicated) improved our performance on the Blind Traveling Salesman Problem. Using a single population of 1000 and a total of 30,000 recombinations *GENITOR* with the enhanced edge recombination operator finds the optimal solution on the 30 city problem described in Whitley et al. [8] on 30 out of 30 runs. On a 105 city problem the new operator finds the "best known" solution on 14/30 runs with no parameter tuning.

2.2 ORDER CROSSOVER

The original order crossover operator (which we refer to as order crossover) was developed by Davis [1] (also see [4]). The offspring inherits the elements between the two crossover points, inclusive, from the selected parent in the same order and position as they appeared in that parent. The remaining elements are inherited from the alternate parent in the order in which they appear in that parent, beginning with the first position following the second crossover point and skipping over all elements already present in the offspring. Thus, although the purported goal is to preserve the relative order of elements in the sequences to be combined, part of the offspring inherits the order, adjacency *and* absolute position of part of one parent string, and the other part of the offspring inherits the relative order of the remaining elements from the other parent, with disruption occurring whenever an element is present that has already been chosen.

An example is given in figure 2. The elements a, j, h, d, and i are inherited from P2 in the order and position in which they occur in P2. Then, starting from the first position after the second crossover point, the child tour inherits from P1. In this example, position 8 is this next position. P1[8] = h, which is already present in the offspring, so P1 is search until an element is found which is not already present in the child tour. Since h, i, and j are already present in the child, the search continues from the beginning of the string and Off[8] = P1[2] = b, Off[9] = P1[3] = c, Off[10] = P1[4] = e, and so on until the new tour is complete.

2.3 ORDER CROSSOVER #2

The operator which was developed by Syswerda [5] differs from the above order operator in that several key positions are chosen randomly and the order in which these elements appear in one parent is imposed on the other parent to produce two offspring; in our experiments we produce only one offspring.

In the example of figure 3, positions 3, 4, 7, and 9 have been selected as the key positions. The ordering of the elements in these positions from Parent 2 will be imposed on Parent 1. The elements (in order) from Parent 2 are a, j, i and b. In Parent 1 these same elements are found in positions 1, 2, 9 and 10. In Parent 1 P1[1] = a, P1[2] = b, P1[9] = i and P1[10] = j, where P1 is Parent 1 and the position is used as an index. In the offspring the elements in these positions (i.e., 1, 2, 9, 10) are reordered to match the order of the same elements found in Parent 2 (i.e., a, j, i, b). Therefore Off[1] = a, Off[2] = j, Off[9] = i and Off[10] = b, where Off is the offspring under construction. All other elements in the offspring are copied directly from Parent 1.

2.4 PARTIALLY MAPPED CROSSOVER (PMX)

This operator is described in detail by Goldberg and Lingle [3]. A parent and two crossover sites are selected randomly and the elements between the two starting positions in one of the parents are directly inherited by the offspring. Each element between the two crossover points in the alternate parent are mapped to the position held by this element in the first parent. Then the remaining elements are inherited from the alternate parent. Just as in the order crossover operator #1, the section of the first parent which is copied directly to the offspring preserves order, adjacency and position for that section. However, it seems that more disruption occurs when mapping the other elements from the unselected parent. In the first example (figure 4), the elements in positions 3, 4, 5 and 6 are inherited by the child from Parent 1. Then beginning with position 3, the element in P1 (c) is located in P2 (position 7) and this position in the offspring is filled with the element in Parent 2 at position 3: Off[7] = P2[3]. Moving to position 4 in Parent 1, we find a d and see that it occurs at position 1 in Parent 2, so Off[8] = P2[5] = a and f (P1[6]) is at P2[10] so Off[10] = P2[6] = g. The remaining elements are inherited from P2: Off[2] = P2[2] = i,

```
Parent 1:    a b c d e f g h i j
Cross Pts:     *       *
Parent 2:    d i j h a g c e b f
Offspring:   h i c d e f j a b g
```

Figure 4: PMX Crossover Example 1

```
Parent 1:    a b c d e f g h i j
Cross Pts:     *       *
Parent 2:    c f a j h d i g b e
Offspring:   a j c d e f i g b h
```

Figure 5: PMX Crossover Example 2

and Off[9] = P2[9] = b.

Since the segment of elements from the alternate parent does not contain any elements in the key segment of the first parent, both adjacency and relative order are preserved.

In the second example (figure 5), the mapping proceeds as above with Off[3 to 6] = P1[3 to 6]. Next Off[1] = P2[3] = a, since P1[3] = c and P2[1] = c. Next, we note that P1[4] = d and P2[4] = j. Since P2[6] = d, this is the preferred position for j in the offspring, but it has already been filled. City j is skipped over temporarily. Element h maps to element e which occupies position 10 in parent 2, so Off[10] = h. City d maps to element f which occupies position 2 in parent 2, so Off[2] = d; even though this is a duplicate it is left in the offspring temporarily. Elements i, g and b are then inherited from P2 leaving a sequence with no j element and two d elements. The element d which is outside the originally selected positions 3 through 6 is replaced with a j resulting in a complete and legal sequence. Note that when this substitution occurs, it results in a mutation where neither adjacency, position, or relative order is preserved by the substitution. Also note that PMX is influenced by position, especially in Example 2.

2.5 CYCLE CROSSOVER

Originally developed by Oliver et al. [4], this operator preserves the absolute position of elements in the parent sequence. A parent sequence and a cycle starting point are randomly selected. The element at the cycle starting point of the selected parent is inherited by the child. The element which is in the same position in the other parent cannot then be placed in this position so its position is found in the selected parent and is inherited from that position by the child. This continues until the cycle is completed by encountering the initial item in the unselected parent. Any elements which are not yet present in the offspring are inherited from the unselected parent. Note that cycle crossover always preserves the position of elements from one parent or the other without any disruption.

```
Parent 1:    a b c d e f g h i j
Cross Pts:         *
Parent 2:    c f a j h d i g b e
Offspring:   c b a d e f g h i j
```

Figure 6: Cycle Crossover

```
Parent 1:    a b c d e f g h i j
Cross Pts:     * *     *     *
Parent 2:    c f a j h d i g b e
Offspring:   a b c j h f d g i e
```

Figure 7: Position Based Crossover

In figure 6, position four in Parent 1 is the selected starting position for the cycle and Off[4] = P1[4] = d. Parent 2 is then searched until the position of element d is found (P2[6]) and the offspring tour at this position inherits the element in this position from Parent 1, Off[6] = P1[6] = f. f occurs in P2 at position 2, so Off[2] = P1[2] = b followed by Off[9] = p1[9] = i, Off[7] = P1[7] = h, Off[5] = P1[5] = e, and Off[10] = P1[10] = j. This completes a cycle, since P2[5] = j and P1[5] = d, which was the starting element in the cycle. Now any remaining elements are inherited from Parent 2: Off[1] = P2[1] = c and Off[3] = P2[3] = a.

2.6 POSITION BASED CROSSOVER

This operator, also proposed by Syswerda [5], is intended to preserve position information during the recombination process. Several random locations in the sequence are selected along with one parent; the elements in those positions are inherited from that parent. The remaining elements are inherited in the order in which they appear in the alternate parent, skipping over all elements which have already been included in the offspring. Thus, the operator appears to be similar to Davis' Order Crossover #1 operator except that the elements copied from the selected parent come from random locations in the sequence and not from adjacent locations; although designed as a "position" operator, it certainly is less effective at preserving position than cycle crossover and probably less effective at preserving position than PMX. We argue this is really another order operator.

In figure 7 the elements b, c, f and i are inherited from Parent 1 in positions 2, 3, 6 and 9 respectively. The remaining elements are inherited from Parent 2 as follows: Off[1] = P2[3] since P2[1] and P2[2] have already been included in the offspring. Then going in order, Off[4] = P2[4], Off[5] = P2[5], Off[7] = P2[6], Off[8] = P2[8] and Off[10] = P2[10].

Op	Bias	Trials	Pop	Best	Avg
Edge	1.5	50000	500	16/30	421.6
Order #1	1.5	50000	500	8/30	429.5
Order #2	1.5	50000	500	9/30	440.5
Position	1.5	50000	500	11/30	431.3
PMX	1.5	50000	500	437	514.6
Cycle	1.5	50000	500	459	519.9

Table 1: 30 City Results (untuned)

Op	Bias	Trials	Pop	Best	Avg
Edge	1.4	30000	1000	30/30	420.0
Order #1	1.1	100000	1000	25/30	420.7
Order #2	1.2	100000	1000	18/30	421.4
Position	1.2	120000	1000	18/30	423.2
PMX	1.2	120000	1400	1/30	452.8
Cycle	1.1	150000	1500	440	490.3

Table 2: 30 City Results (tuned)

3 THE 30 CITY BLIND TRAVELING SALESMAN

Each of the above operators was used to solve the 30 city Traveling Salesman Problem. In order to compare the performance on two levels they were each run using the same parameters for 30 experiments and then each was tuned for best results. The parameters for the first comparison were: selection bias of 1.5, population size of 500, no explicit mutation and 50,000 trials. The only exception to this is that the cycle crossover operator always mutates whenever the offspring and the selected parent are identical. Results appear in Table 1.

We attempted to optimize the performance of each operator by tuning the following parameters: bias, population size and number of trials. The results in able 2 are similar to those in Table 1, although PMX and cycle showed very little improvement despite the parameter tuning. The three order crossover operators (Order #1, Order #2 and Position) have similar performance. Using higher selection bias values in general gave poorer results for all operators except edge recombination. PMX and cycle crossover in particular converged too quickly in most cases to find the optimal solution. The improved edge recombination operator found the optimal solution 28 out of 30 times using a population of 650, bias of 1.7 and 30,000 recombinations. As shown, a larger population found the optimal solution on every attempt. Our results differ somewhat from the results cited by Oliver et. al.[4]. The ranking of the operators in terms of performance is the same (Order #1 is better than PMX which is better than Cycle). The main difference is that all of these operators produced much better results in the current study. Order crossover #1 and PMX both failed to find the optimal solution to this problem in the Oliver et al. study. The main difference in the two studies is the use of the *GENITOR* algorithm instead of the standard generational genetic algorithm. This strongly suggests that the use of *GENITOR* is partially responsible for the positive results we have obtained on this problem.

In a previous study we found that the old edge recombination operator coupled with a distributed genetic algorithm found the best known solution on 30 out of 30 attempts using 10 subpopulations of 200 individuals each, using up to a total of 70,000 evaluations/recombinations (7,000 per subpopulation). On the 105 city problem the old edge recombination operator operator coupled with a distributed genetic algorithm matched the best known solution on 15 out of 30 attempts using up to 2 million recombinations; all results were within 1% of the best known solution and 29/30 were within 0.5% of the best known. Using the enhanced edge recombination operator we found the best known solution on 14/30 runs using a single population algorithm (popsize: 5000) and only 1 million recombinations. These are first run results with no parameter tuning. While these results are not directly comparable, they do support the notion that the enhanced edge recombination operator is more effective than the original implementation.

3.1 DISCUSSION

The key difference between the operators is the information which each attempts to preserve during recombination. For the Traveling Salesman Problem the important information would seem to be the adjacency information. The edge recombination operator explicitly preserves adjacency information and clearly has the best performance on this problem. Information about absolute position appears to be relatively unimportant. None of the operators use mutation (except cycle crossover when the offspring is identical to one of the parents). We have done some experiments which suggest that the performance of some of the operators can be improved if mutation is used; resolving this issue requires further tests. Perhaps most surprising is the difference in performance between the order operators and among the position preserving operators. These differences can be explained by looking at how the operator preserves adjacency information, relative order and position. Adjacency information is clearly important, but the results obtained with the order operators (order crossover #1, order crossover #2, and the so-called position based crossover) suggests that order information is useful for solving this problem. PMX may produce a greater emphasis on absolute position than the other order operators; the cycle operator clearly stresses absolute position.

It is important to note that the performance of these operators on a given problem is directly related to the nature of that problem. In other problems such as scheduling, the important information may not be adjacency, but may have a higher correlation to the position in the string or the relative order among the encoded elements in the string.

4 A WAREHOUSE/SHIPPING SCHEDULER

A prototype scheduling system has been developed for the Coors brewery in Golden, Co., which uses a genetic algorithm to optimize the allocation of beer production and inventory to the orders at the plant. A simulator was constructed consisting of a representation for beer production, the contents of inventory, arrangement of truck and rail loading docks, and orders for a 24 hour period. Preliminary tests indicated that the system is viable and subsequent tests of the system used real data from the plant.

The objective of the Coors scheduling package is the efficient allocation of orders to loading docks in the plant based on a fixed production schedule. Beer production occurs on multiple production lines which operate 24 hours a day. Each line produces different product types. There are numerous product types which can be produced, based on type of beer as well as various packages and labels. The data which is available for each line includes flow-rate, start and stop times, and product-type. The scheduling simulator analyzes the production schedule for each line and creates a time-sorted list composed of the product-type, amount, and time available. This time-sorted production list is then examined during an event driven simulation. An input file to the scheduling simulator contains the contents of inventory at the start of the time-period which is to be scheduled. New orders enter a loading dock upon completion of a previous order; the inventory is initially checked for product needed by the new order. Minimizing the contents of inventory is an important aspect of this problem. Inventory impacts the physical work of moving product more than once on the plant floor, the physical space occupied by product in storage, as well as refrigeration costs, etc. The schedule simulation places orders in rail and truck loading docks and attempts to fill the orders with the product that comes out of production and inventory. An order consists of one or more product type and an associated amount. In the actual data for the test scheduling period 195 customer orders are present and waiting to be filled. The schedule simulator attempts an efficient mapping between the product that is available and these orders. A "good" schedule is one which minimizes the average inventory in the system, and fills as many orders as possible. Each individual in the population maintained by the genetic algorithm is a sequence of customer order numbers. This sequence is mapped to the loading docks by the schedule simulator and orders are filled and placed in the docks based strictly on this sequence. Initially these sequences are randomly created, and as genetic search progresses new sequences are created by the process of genetic recombination. For the genetic algorithm to work, an evaluation for the sequence is needed. The evaluation of the sequence of orders is obtained using a scheduling simulator, which models operation of the plant and creates a shipping schedule based on the sequence.

Our results indicate that for this sequencing problem, relative order of the items which make up the sequence is more important than adjacency. This is not surprising given the nature of the problem: the relative order in which product is used will clearly affect inventory more than adjacency. Adjacency would appear to be almost irrelevant in this domain. This means that genetic recombination operators which perform well on the Traveling Salesman because they stress adjacency will be poor for sequencing tasks where relative order is critical. Experiments with the same 6 recombination operators tested on the 30 city Blind Traveling Salesman Problem were conducted on this warehouse/shipping sequencing task.

Figure 8 gives two graphs comparing the six operators on both the Blind Traveling Salesman Problem and the warehouse/shipping scheduler. As the graphs show, the results of schedule optimization with the six operators are almost the opposite of the results for the Traveling Salesman Problem.

The graph of the warehouse/shipping scheduler shows the comparative results for the 6 operators with up to 30,000 recombinations. (The population size is 200, the selective bias is 1.7, the number of runs is 15; no parameter tuning was used.) Both of Syswerda's operators did extremely well on this problem; they also did relatively well on the Traveling Salesman Problem. When we use edge recombination with up to 200,000 recombinations it finds solutions comparable to those found by PMX in 20,000 recombinations; both results are inferior to order crossover #2 and the position based operator.

The difference in search speed displayed by the operators coupled with increased performance in workstations has allowed us to achieve 2 orders of magnitude improvement in execution speed on the scheduling application. We are currently using a 15 MIP workstation. This means that our scheduler now executes in minutes rather than hours. (e.g., 6 hours becomes 3.6 minutes given 100 times faster execution). Scheduling 195 jobs in under 5 minutes is close to real time in the context of this warehouse/shipping problem. This allows quick rescheduling in the event of line breakages, shortfalls in production, and other unforeseen circumstances. Our execution times are, of course, dependent on the complexity of the evaluation function.

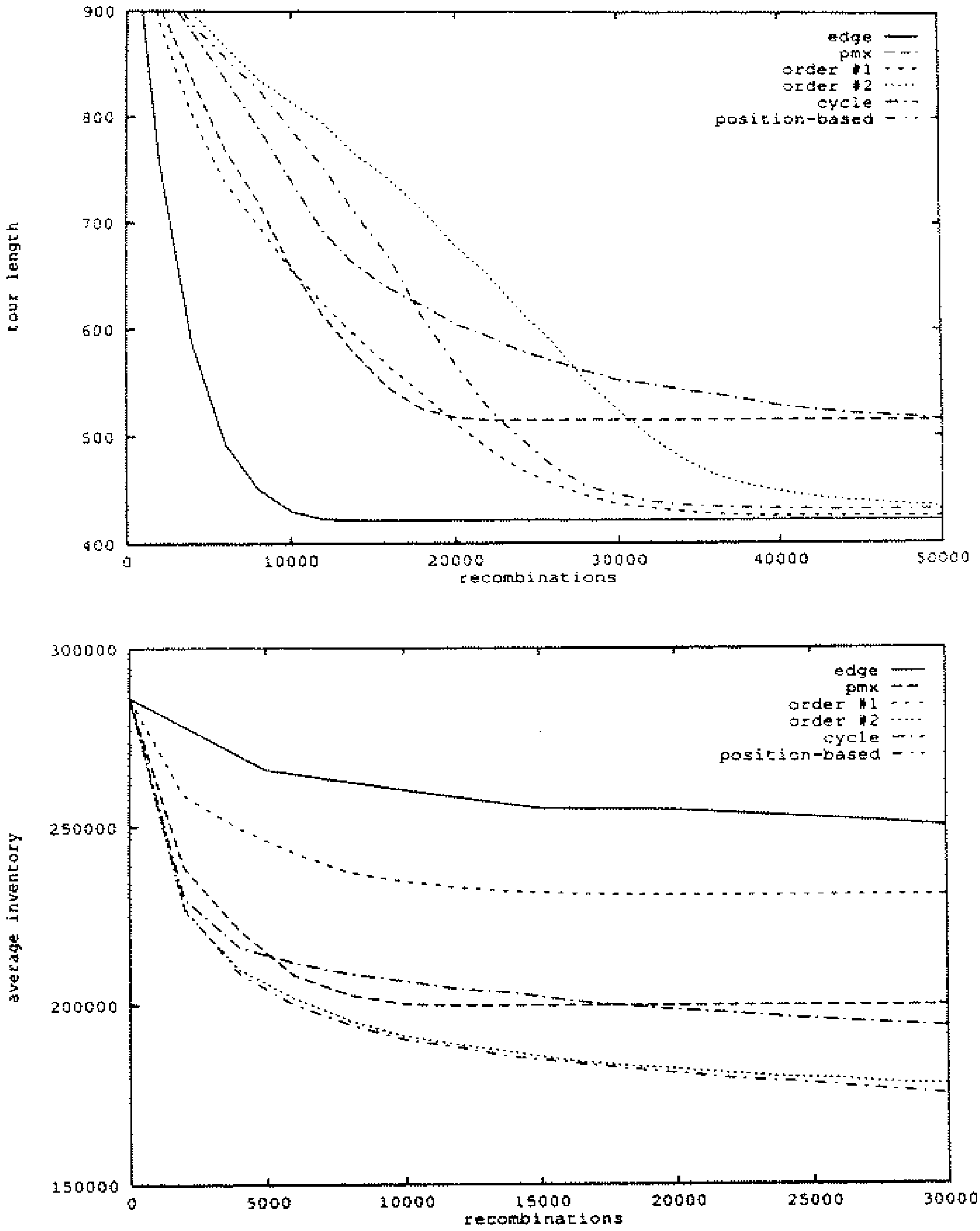

Figure 8: Graphs of 6 Operators for the Blind 30 City Traveling Salesman Problem (top) and the Warehouse/Shipping Scheduler (bottom).

ACKNOWLEDGEMENTS

This research was supported in part by a grant from the Colorado Institute of Artificial Intelligence (CIAI). CIAI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the State of Colorado.

References

[1] L. Davis. (1985) "Applying Adaptive Algorithms to Epistatic Domains." In *Proc. International Joint Conference on Artificial Intelligence.*

[2] D. Goldberg. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, MA.

[3] D. Goldberg and R. Lingle. (1985) "Alleles, loci, and the Traveling Salesman Problem." In *Proc. International Conference on Genetic Algorithms and their Applications.*

[4] I. Oliver, D. Smith, and J. Holland. (1987) "A Study of Permutation Crossover Operators on the Traveling Salesman Problem." In *Proc. Second International Conference on Genetic Algorithms and their Applications.*

[5] G. Syswerda. (1990) "Schedule Optimization Using Genetic Algorithms." In *Handbook of Genetic Algorithms.* l. Davis, ed. Van Nostrand Reinhold, New York.

[6] N. Ulder, E. Pesch, P. van Laarhoven, H. Bandelt, E. Aarts. (1990) "Improving TSP Exchange Heuristics by Population Genetics." In *Parallel Problem Solving In Nature.* Springer/Verlag.

[7] D. Whitley and J. Kauth (1988) "*GENITOR*: A Different Genetic Algorithm" In *Proc. Rocky Mountain Conf. on Artificial Intelligence.*

[8] D. Whitley, T. Starkweather, and D. Fuquay. (1989) "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator." In *Proc. Third Int'l. Conference on Genetic Algorithms and their Applications.* J. D. Shaeffer, ed. Morgan Kaufmann.

[9] D. Whitley and T. Starkweather. (1990) "*GENITOR II*: A Distributed Genetic Algorithm." *Journal of Experimental and Theoretical Artificial Intelligence.* 2:189-214.

[10] D. Whitley, T. Starkweather, and D. Shaner. (1990) "Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination." In *Handbook of Genetic Algorithms.* L. Davis, ed. Van Nostrand Reinhold, New York.