TECHNICAL REPORT NO. 314

Parallel Genetic Algorithms Applied to the Traveling Salesman Problem

by

Prasanna Jog, Jung Y. Suh and Dirk Van Gucht

August 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

Parallel Genetic Algorithms Applied to the Traveling Salesman Problem *

Prasanna Jog   Jung Y. Suh   Dirk Van Gucht

Computer Science Department, Indiana University
Bloomington, IN 47405-4101, USA

August 14, 1990

Abstract

Genetic algorithms are adaptive search algorithms that have been shown to be robust optimization algorithms for multimodal real-valued functions and a variety of combinatorial optimization problems. In contrast to more standard search algorithms, genetic algorithms base their progress on the performance of a population of candidate solutions, rather than on a single candidate solution.

We will concentrate on the application of genetic algorithms to the *traveling salesman problem*. For this problem, there exist several such algorithms, ranging from *pure* genetic algorithms to genetic algorithms which incorporate heuristic information. We will review these algorithms and contrast their performance.

A serious drawback of genetic algorithms is their inefficiency when implemented on a sequential machine. However, due to their inherent parallel properties, they can be successfully implemented on parallel machines, resulting in considerable speed-up. We will review *parallel genetic algorithms* and indicate how they have been used in the traveling salesman problem.

1 Introduction

Suppose we have an *object space* X and a function $f : X \rightarrow R^+$ (R^+ denotes the positive real numbers) and our task is to find a global optimum for that function. *Genetic algorithms* are a class of adaptive search algorithms invented by John Holland [16] to solve (or approximately solve) such problems. Genetic algorithms differ from more standard search algorithms (e.g., gradient descent, controlled random search, hill-climbing, simulated annealing etc.) in that the search is conducted using the information of a *population* of *structures* of the object space X instead of that of a single structure. The motivation for this approach is that by considering many structures as potential candidate solutions, the risk of getting trapped in a local optimum is greatly reduced. In Figure 1 we show the layout of a typical genetic algorithm (GA). The initial population P(0) consists of structures of X, usually chosen at random. Alternatively, P(0) may contain heuristically chosen structures. In either case, the initial population should contain a wide variety of structures. Each structure x in P(0) is then evaluated by applying to it the function f. The genetic algorithm then enters a loop. Each iteration of that loop is called a *generation*. The new population P(t+1) is constructed in two steps 1) the *selection* step and 2) the *recombination step*. In the selection step, a temporary population (say $\mathtt{P'(t+1)}$) is constructed by choosing structures in P(t) according to their relative performance. For example, if we are maximizing f, the structures with greater than average performance will be selected with higher probability than the structures with below average performance. This resembles the *survival of the fittest principle* of natural evolution. After

*Presented at Sym. on Parallel Optimization2, Univ. of Wisc., Madison, July 1990.

```
P(t) denotes the population at time t.

t ← 0;
initialize P(t);
evaluate P(t);
while (termination condition is not satisfied)
{
   t ← t+1;
   select P(t);
   recombine P(t);
   evaluate P(t);
}
```

Figure 1: Layout of a Standard Genetic Algorithm

the selection step, the temporary population $P'(t+1)$ is *recombined*. (The resulting population is the new population P(t+1).) Typically, recombination is accomplished by applying several *recombination operators*, such as *crossover*, *mutation*, *inversion* [9,16], or *local improvement* [40], to the structures in $P'(t+1)$. After the recombination step is completed, the new population is reevaluated and a termination condition is checked for validity.

The difference between genetic algorithms and earlier introduced evolutionary algorithms is the usage of crossover in the former. A *crossover* operation produces an offspring from two structures. It is typically defined such that a sufficient amount of information embedded in the parents appears in the offspring. Therefore, crossover can be viewed as a *history preserving* operation which at the same time introduces a new structure to be tested in the competition. This is clearly different from a random mutation which was the central operator in evolutionary algorithms.

Genetic algorithms have been applied to global function optimization, combinatorial optimization, machine learning etc.[10]. In this paper, we will concentrate on the application of genetic algorithms to the *traveling salesman problem*. For this problem, there exist several different genetic algorithms, ranging from *pure* genetic algorithms to genetic algorithms which incorporate heuristic information. We will review these algorithms and contrast their performance (Section 2).

A serious drawback of genetic algorithms is their inefficiency when implemented on a sequential machine. However, due to their inherent parallel properties, they can be successfully implemented on parallel machines, resulting in considerable speed-up. We will review *parallel genetic algorithms* in general, and subsequently indicate how they have been used in the traveling salesman problem (Section 3). In Section 4 we will show some additional results of applying parallel genetic algorithms to large TSP's (the largest problem is a 1000 cities problem). This section also contains data that indicates the relative importance of certain recombination operations.

In Section 5 we discuss how the research on parallel genetic algorithms can be used in the parallelization of probabilistic sequential search algorithms.

2 Genetic algorithms for the traveling salesman problem.

The traveling salesman problem (TSP) is easily stated: Given a complete graph with N nodes, find the shortest Hamiltonian tour through the graph (in this paper, we will assume Euclidean distances between nodes). For an excellent discussion of the TSP, we refer to [22].

For the TSP, the object space X consists of all Hamiltonian tours (tours, for short) and f,

$$A = 9\ 8\ 4 || 5\ 6\ 7 || 1\ 3\ 2\ 10$$
$$B = 8\ 7\ 1 || 2\ 3\ 10 || 9\ 5\ 4\ 6$$

Two aligned tours

$$A' = 9\ 8\ 4 || 2\ 3\ 10 || 1\ 6\ 5\ 7$$
$$B' = 8\ 10\ 1 || 5\ 6\ 7 || 9\ 2\ 4\ 3$$

Result of the PMX crossover on the two above tours

Figure 2: Illustration of a PMX crossover

the function to be optimized, returns the length of tours. In recent years, a variety of GA's for the TSP have been proposed, [4,11,10,12,15,14,18,23,27,28,29,40,39,37]. These algorithms can be separated into two groups: 1) the *pure genetic algorithms*, i.e., algorithms that do *not* use domain specific information about the TSP [11,10,14,29], and 2) *heuristic genetic algorithms* (HGA), i.e., algorithms that do use domain specific information about the TSP [4,12,15,14,18,23,27,28,40,39,37].

2.1 Pure genetic algorithms for the TSP

Pure genetic algorithms use recombination operators that apply to arbitrary permutations, hence they can be used in any problem domain involving objects represented as permutations[1].

- An example of a domain independent mutation operator is the *inversion* operator [16,14]. This operator, for a given tour, simply reverses a randomly chosen subtour.
- We now consider various pure crossover operators. All these crossover operators share the property of preserving, in the offspring, subtours of the parents. The intuition is that in doing so, potentially good subtours from the parents are combined to obtain a possible better tour as an offspring.

  1. The *partially matched crossover* (PMX) [11,10] starts by aligning two tours, subsequently two crossing sites are picked uniformly at random along the tours (see top part of Figure 2). These two points define a matching section that is used to effect a cross through position-by-position exchange operations. Notice that it might be necessary to change the original tours outside the matching section to guarantee that the new objects are again tours (the result of a PMX crossover is shown in the bottom part of Figure 2). The *order crossover* operator [8,7,10,29,38] is a variant of the PMX crossover.
  2. The *cycle crossover* (CX) operator [8,7,10,29,38] performs recombination under the constraint that each city name come from one parent or the other. Consider the tours at the top of Figure 3. The result of applying the cycle crossover is shown at the bottom part of Figure 3.
  3. The Grefenstette crossover (Gref-X) [15,14,34] operator takes two tours and constructs an offspring as follows. Randomly choose a city as the current city for the offspring tour. Consider the four edges incident to the current city in the parents. Randomly select one of these four edges and include it in the offspring. (If none of the parental edges leads to an unvisited city, create an edge to a randomly chosen unvisited city.) Repeat this

[1]From now on, we will use the term *tour* instead of permutation since we are only considering the TSP.

$A = 9\ 8\ 2\ 1\ 7\ 4\ 5\ 10\ 6\ 3$
$B = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$

Two tours before applying the CX crossover

$A' = 9\ 2\ 3\ 1\ 5\ 4\ 7\ 8\ 6\ 10$
$B' = 1\ 8\ 2\ 4\ 7\ 6\ 5\ 10\ 9\ 3$

Two tours resulting from the CX crossover

Figure 3: Illustration of a CX crossover

$A = 9\ 8\ 2\ 1\ 7\ 4\ 5\ 10\ 6\ 3$
$B = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$

Two tours before applying the Gref-X

$B = 9\ 8\ 2\ 3\ 4\ 5\ 6\ 7\ 1\ 10$
A possible offspring after a Gref-X

Figure 4: Illustration of a Gref-X crossover

process until all cities have been visited. Consider the tours at the top of Figure 4. A possible offspring of these tours is shown at the bottom of this figure.

4. The Mulhenbeim-crossover [27] uses a donor and receiver tour. From the donor a random substring is chosen. All nodes which are included in the string are deleted from the receiver. Then the substring of the donor is copied over and the remaining receiver nodes in the order they appear. Consider the donor and receiver at the top of Figure 5. Assuming that the section between the two || is the portion donated by the donor, the resulting offspring is shown at the bottom of the figure.

2.2 Heuristic genetic algorithms for the TSP

Heuristic genetic algorithms use recombination operators that incorporate heuristic information about the TSP. The essential heuristic in all these operators is the observation that solutions to the TSP contain short edges. Hence, when the opportunity arises to select between edges, the operators choose the shorter ones.

- Heuristic mutation is usually implemented in the form of so called *local improvement operators*. Examples include the 2-opt and 3-opt operators of Lin and Kernighan [24] and the Or-opt operator of [22,30].

$donor \;\;\; = 9\ 8\ \|2\ 1\ 7\ 4\|5\ 10\ 6\ 3$
$receiver = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$

$offspring = 2\ 1\ 7\ 4\ 3\ 5\ 6\ 9\ 10\ 3$

Figure 5: The Mulhenbeim-crossover operator

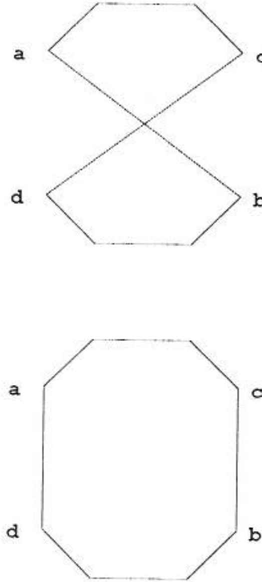


Figure 6: Illustration of a 2-opt operation

1. The 2-opt operator randomly selects two edges (a, b), (c, d) from a tour (see Figure 6) and checks if $ED(a, b) + ED(c, d) > ED(a, d) + ED(c, b)$ (ED stands for Euclidean distance). If this is the case, the tour is replaced by removing the edges (a, b), (c, d) and replacing them with the edges (a, d), (c, b).
2. The 3-opt operator is similar to the 2-opt operator, except that it applies to three rather than two edges.
3. The Or-opt was introduced by Or [22,30] and is a variant of the 3-opt operator. The advantage of the Or-opt operator is that it only considers a small percentage of the exchanges that would be considered by a regular 3-opt operator. To understand how the Or-opt procedure works, we refer to Figure 3. For each connected string of s cities in the current tour (s can be 3, 2, or 1), we test to see if that string can be relocated between two other cities at reduced cost. If it can, we make the appropriate changes. For example, for $s = 3$ (see Figure 7), we test to see if the string of the three adjacent cities m, n, p in the current tour is considered for insertion between a pair of connected cities i and j outside of the string. The insertion is performed if the total cost of the edges to be erased, $\{a, m\}, \{p, b\}$, and $\{i, j\}$, exceeds the cost of the new edges to be added, $\{i, m\}, \{p, j\}$ and $\{a, b\}$.

- Heuristic information has also been incorporated in various crossover operators for the TSP:
  1. The *Brady-crossover* [4,3] takes two tours and searches for subroutes where some common subset of cities are visited in a different order (see Figure 8). The shorter subroute is then copied over to replace the longer one. In the figure, we assume that the sequence 2 1 7 4 5 is shorter than the sequence 2 7 5 1 4.
  2. The *Heuristic-crossover* (Heur-X)[2] [15] operator constructs an offspring from two parent tours as follows: Pick a random city as the starting point for the offspring's tour. Compare the two edges leaving the starting city in the parents and choose the shorter

[2] In fact, the crossover we describe here is a variant of the crossover introduced in [15].

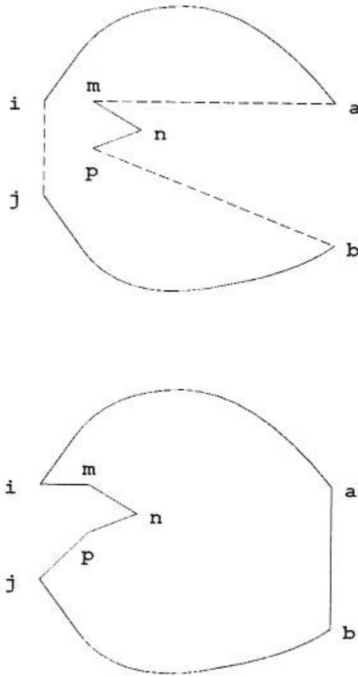


Figure 7: Illustration of a Or-opt local improvement operation.

$$A = 9\ 8\ ||2\ 1\ 7\ 4\ 5||10\ 6\ 3$$
$$B = 8\ 9\ ||2\ 7\ 5\ 1\ 4||3\ 6\ 10$$

Two tours before applying the Brady-crossover

$$C = 8\ 9\ 2\ 1\ 7\ 4\ 5\ 3\ 6\ 10$$

A possible offspring after a Brady-crossover

Figure 8: Illustration of a Brady crossover operation

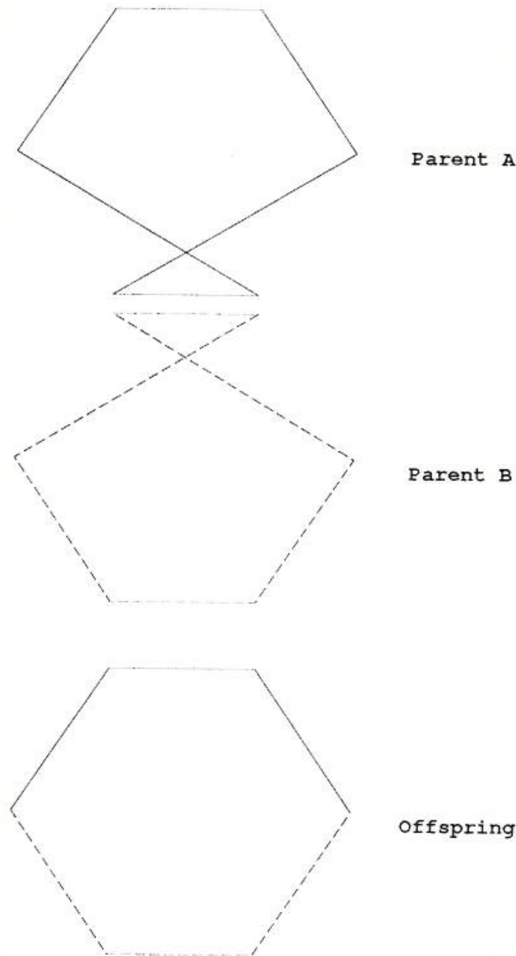


Figure 9: The Grefenstette-crossover operator

edge. Continue to extend the partial tour by choosing the shorter of the two edges in the parents which extend the tour. If the shorter parental edge would introduce a cycle into the partial tour, check if the other parental edge introduces a cycle. In case the second edge does not introduce a cycle, extend the tour with this edge, otherwise extend the tour by a random edge. Continue until a complete tour is generated (see Figure 9). Variations of the Grefenstette-crossover including more heuristic information where introduced by [18,23,40,37].

2.3 Performance results of genetic algorithms for the TSP

We now describe the performance results of the pure and heuristic genetic algorithms. We will indicate the size of the TSP problems attempted and the quality of the obtained solutions. Since most papers do not report the time required to run the algorithms, we can sometimes not report on this performance parameter. Instead, we will give the population size and the number of generations for the GA's to converge.

We first consider the performance of pure genetic algorithms. These results were obtained from [10,14] and shown in Figure 10[3]. In this table, the column *algorithm* indicates the chosen

[3]We do not report the results of Lie87, because we consider TSP problems with 15 cities too small.

Reference	Algorithm	Problem	Population	Best	Average	Time
[10]	PMX	33-cities	2000	10.0	NA	300
[10]	Inversion	33-cities	2000	70.0	NA	300
[14]	Random	100-cities	100	480.0	NA	200
[14]	PMX	100-cities	100	210.0	NA	200

Figure 10: Results of pure genetic algorithms applied to the TSP

recombination operators, *population* denotes the population size, *best* gives the percentage above the known optimum for the TSP under consideration, of the best solution found by the GA, *average* gives the percentage above the optimum for a sample of solutions produced by the GA, *time* gives the number of generations. The 33-cities problem (100-cities problem) is a TSP with 33 (100) randomly distributed cities in a square.

As can be seen from this table, pure genetic algorithms perform poorly, especially when larger TSP problems are considered. We therefore don't consider such algorithms in the rest of the paper. Argument for this poor performance can be found in [14].

We next consider the performance of heuristic genetic algorithms. These results were obtained from [4,12,15,14,18,27,28,40,39,37] and are shown in Figure 11. We have organized the results chronologically since this reflects 1) how the problem sizes of attempted TSP problems increased, and 2) how the fine tuning of these algorithms led to better performance. The 50-cities (100-cities, 200-cities) problem is a TSP of 50 (100, 200) randomly distributed cities in a square. The optimum tour length for such problems can be estimated as described in [1]. The *Krolak* problem is the often cited 100 cities problem described in [21]. This problem has been used in numerous studies as a benchmark ([22], Chapter 7). The *lattice* problem is a TSP of 100 cities arranged in a ten by ten grid. This problem has been used as a benchmark in the performance of simulated annealing [5,35]. The *Grotschel* problem is a TSP with 442 cities non-uniformly distributed in a square [35]. The *Padberg* problem is a TSP problem of 532 cities in the United States [31].

As a general comment, it can be see that genetic algorithms are very good heuristic algorithms for the TSP. In several cases the best solution was the optimum solution, in other cases, the performance was to within 1% of the optimum. Also notice how the robustness of the genetic algorithms is reflected in their average performance. To even better appreciate the quality of these algorithms, we refer to Chapter 7 in [22] where a survey is given of the performance of many other heuristics for the TSP. Compared to this study, heuristic GA's compare well or outperform the best results obtained with the best heuristics reported.

The table in Figure 11 does not indicate all the features that where considered in GA's for the TSP. For example, some authors have studied the effects of the population size [4,18,27,28], the initial diversity in the population [14], the quality of local improvement [18], the robustness of the GA (i.e., its ability to repeatedly find similar results) [12,18,17] and the usage of dynamic recombination operators [37,40,18].

3 Parallel Genetic Algorithms

Genetic algorithms, when implemented on a sequential machine, are notoriously slow. Luckily, genetic algorithms lend themselves naturally to speed-up through parallelization. This led to the introduction of *parallel genetic algorithms* (PGA) [6,12,13,17,25,27,28,26,32,33,36,40,41,42].

In addition to seeking speed-up, this research has also pointed out some weaknesses in the design choices for genetic algorithms as originally proposed by Holland [16]:

Ref.	Algorithm	Problem	Population	Best	Avg	Time	Comments
[4]	2-opt Brady-X	64-city	24(max)	NA	1.3	1.0s IBM 3081D	Avg. 100
[15]	Heur-X	100-cities	100	16.0	27.0	400	
[37]	Inversion Heur-X	50-cities	10	5.5	NA	1000	Dynamic-X
[37]	Inversion Heur-X	100-cities	12	7.2	NA	3700	Dynamic-X
[37]	Inversion Heur-X	200-cities	15	3.4	NA	5800	Dynamix-X
[40]	2-opt Heur-X	krolak	100	1.8	3.7	500	Low % 2-opt Avg 5
[40]	2-opt Heur-X	lattice	100	0.0	0.4	200	Low % 2-opt Avg 5
[40]	2-opt Heur-X	200-cities	100	2.4	4.5	850	Low % 2-opt Avg 5
[39]	2-opt Heur-X	krolak	100	1.2	2.3	400	Parallel Low % 2-opt Avg 10
[39]	2-opt Heur-X	lattice	100	0.0	1.1	200	Parallel Low % 2-opt Avg 10
[39]	2-opt Heur-X	200-cities	100	1.9	3.6	800	Parallel Low % 2-opt Avg 10
[14]	Heur-X	100-cities	100	NA	< 7.4	200	Post 2-opt
[27]	2-opt Muhl-CX	krolak	100max	NA	1.9	NA	Parallel GA Avg 5 High % 2-opt
[28]	2-opt Muhl-CX	krolak	24	0.0	0.0	60	Parallel GA High % 2-opt Avg 25
[28]	2-opt Muhl-CX	Grotschel 442 cities	24	1.0	NA	NA	Parallel GA High % 2-opt Best
[18]	2-opt 0r-opt Heur-X	krolak	100	0.0	1.4	590	Low % 2-opt Low % Or-opt
[18]	2-opt 0r-opt Heur-X	lattice	100	0.0	0.4	420	Low % 2-opt Low % Or-opt
[12]	2-opt Muhl-CX	Grotschel 442 cities	64	0.3	0.4	NA	Moderate % 2-opt Avg 10
[12]	2-opt Muhl-CX	Padberg 532 cities	64	0.2	0.4	1200	Moderate % 2-opt Avg 10

Figure 11: Results of heuristic genetic algorithms applied to the TSP

- In accordance with nature, it is more natural to view a population as consisting of a set of independent structures, each with it own local behavior, i.e., each has the opportunity to initiate or undergo recombination operations, without the control of a global agent.
- Selection in standard GA's is a global process, i.e., selection of an individual depends on its performance relative to the average performance of the *entire* population. This is quite unlike nature and inefficient to parallelize [17]. Parallel GA's therefore introduce *local selection* without affecting the performance of the algorithm.
- PGA's are very reliable, especially when implemented is as distributed a fashion as possible, i.e., the failure of a processor usually does not affect the rest of the computation.
- PGA's allow for asynchronous behavior. This is not possible in standard GA's. This allows different structures to evolve at different speeds which may result in the global speed-up of the algorithm as well as the maintenance of diversity, a critical component for the success of a GA.

Parallel genetic algorithms can be categorized according to their level of distributedness of the population (coarse-grained versus fine-grained), and the manner in which the recombination and selection strategies are supported.

In a typical *coarse-grained* PGA, the population is divided into sub-populations. Each processor of the parallel machine gets allocated a sub-population and independently runs a standard GA. In particular, the recombination and selection operations are performed within a sub-population. To support global competition between the sub-population, on an occasional basis, communication is established between the processors to facilitate selection between the sub-populations. As a side-effect of this global selection, individuals can migrate to other populations.

A typical *fine-grained* PGA is obtained by allocating a single individual[4] to each processor. Each processor is powerful enough to perform evaluations of individuals, and to perform recombination operations such as mutation, local improvement and crossover. Occasionaly communication is established between the processors to facilitate recombination and selection.

3.1 Coarse-grained parallel genetic algorithms

Pettey et.al. [33] describe a coarse-grained PGA (implemented on an Intel iPSC, with a n-cube interconnection network, upto 16 processors) applied to De Jong's [9] test bed of five global function optimization problems. To support global selection, once every generation each processor sends its best individual to its neigboring processors and incorporates the best individuals, sent to it by its neigbors, into its local population. Best results were obtained with the maximum of 16 processors, each with a sub-population of size 50, and were comparable with results obtained with standard GA's. In [32] a theoretical analysis of this PGA is given.

Tanese [41] describes a coarse-grained PGA (implemented on a 64 NCUBE/six hypercube made by NCUBE corporation) applied to a GA-hard global function optimization problem (see [41] for a definition of GA-hardness). To support global optimization, once every five generations, each processors sends a portion of its best individual to *one* of its neighbors. A fixed population of 400 individuals was distributed over 1 through 32 processors. The quality of the solution obtained with the PGA was similar to that of a standard GA. Furthermore, a nearly linear speed-up was recorded. ([42] is a continuation of this study.) An interesting addition is the introduction of a *partitioned* PGA, i.e., a PGA without migration. The author reports that this PGA is a better optimizer

[4]Sometimes, instead of a single individual, a small number of individuals is allocated to each processor.

than a standard GA on the same problems, however, significant variations exists in the average performance of the sub-populations. To improve this, a reasonably small degree of migration was introduced to yield both the quality of the partitioned PGA and to obtain comparable performance average amongst the sub-population.

Cohoon et.al. [6] describe a PGA which is a hybrid of the Pettey and Tanese algorithms After a fixed amount of generations, each processors sends a set of its best individuals to all its neighbors. After receiving these individuals, each processor selects a new population on the basis of the old population and these new individuals. The chosen application problem is the quadratic assignment problem. In a simulation of the parallel algorithm on a sequential machine, the authors report better quality of solutions for PGA than for a standard GA.

3.2 Fine-grained parallel genetic algorithms

Muhlenbein, Gorges-Schleuter and Kramer [27,28] describe a fine-grained PGA and apply it to the traveling salesman problem. Their algorithm (see [27,28]) is as follows:

1. Take a sequential heuristic algorithm for the TSP (such as 2-opt).

2. Give the problem to N processors with different starting configurations.

3. Each processor computes a local optimum according to the sequential heuristic algorithm.

4. Each process performs a local selection by assigning ranking weights to individuals in neighboring processors (better individuals receive higher weights and each processor has four neighbors). A mating partner is chosen probabilistically according to this weight distribution.

5. Perform crossover and mutation (see Section 2).

6. Reduce the problem by collapsing common subtours, solve the reduced problem and expand these solutions.

7. If not converged go back to step 3.

The fine-grained nature of this algorithm is transparent in a) steps 2, 3, and 6 where individuals processors independently work towards a local minima, b) step 4 where local selection is performed instead of global selection as done in standard GA's, and c) step 5 where crossover and mutation are done independently as local processes of the processors. This algorithm, and a refined version reported in [12] was successfully applied to various TSP problems (see Figure 11). The work reported in [26] applies a similar fine-grained PGA to the quadratic assignment problem.

Suh and Van Gucht [39] describe a fine-grained PGA and apply it to the traveling salesman problem. Their approach considers a framework consisting of a pool of processors which execute identical or nearly identical tasks in parallel. Each processor has a local memory large enough to store a small number of structures, one of which is called the *local structure*. The collection of all these local structures in the processors constitutes the population of the genetic algorithm. Each processor is capable of performing local tasks and communicating with the other processors. The local tasks and communication serve to 1) perform evaluation of individuals 2) perform recombination operations and 3) perform local selections (as described in [39]), there exists a variety of reasonable local selection strategies). Their algorithm is very similar to the one of Muhlenbeim et.al., except that 1) instead of letting processors completely converge to local optima (as in step 3 of the Muhlenbein et.al. algorithm) only a certain amount of local improvements is performed before processors can interact, and 2) communication can be established directly between each

pair of processors, regardless of their geometric relationship within the connection network of the parallel machine. This algorithm was successfully applied to various TSP problems (see Figure 11). and yields the same results as standard GA's. In Section 4 we will apply this algorithm to large TSP problems with different rates of local improvement operations.

Manderick and Spiessens [25] describe a fine-grained PGA very similar to the algorithms of [27,28,26,12] and [39]. Each individual of the population is allocated to a single processor. The processors are interconnected in a grid. Each processor performs the evaluation of its individual. Mutation occurs locally within the processors. Local selection is performed by each processor and is implemented by the calculation the fitness distribution in the neighborhood of that processor. Each processor selects a new individual on the basis of this distribution. Crossover is done between neighboring processors. A simulation of this algorithm was applied to global function optimization problems [9,41] and a comparison was made with a standard GA. The quality of various performance measures showed only minor differences between the PGA and the standard GA.

Sannier and Goodman [36] describe a fine-grained PGA allocating a single or small number of individual to each processor. The processors are interconnected in a grid. Communication can be established between each pair of processors, but the likehood of this depends inversely on the distance in the grid between the processors. This communication is used to perform inter-processor selection and crossover. This algorithm was applied to a survival game in which individuals compete for food.

4 Parallel algorithms applied to large TSP's

A closer look at Figure 11 reveals that an important calibration factor in determining the quality of solutions obtained for various TSPs is the amount of local improvement operations performed on a tour during each generation.

The intuition is a follows: if too little local improvement is performed, the selection pressure of the GA will make it prematurely converge into a sub-quality local optimum. If too much local improvement is performed, however, there will be few chances for tours to crossover and selection will have few opportunities to weed out poor tours. This will unnecessarily increase the convergence time.

It therefore is reasonable to state that a balance has to be stricken between the desired quality of the solution and the time in which it is obtained. For example, the work of [40,39,18] emphasizes speed, whereas the work of [27,28] emphasizes quality of solution. The best balance can be found in the work of [12], in which high quality solutions are obtained, in the presence of plentiful use of the recombination operations and local selection.

The natural question arises to what degree one can decrease the amount of local improvement operations per generation and still obtain high quality solutions. In this section, we address this question by applying the parallel genetic algorithm of [39] to a variety TSP's of growing size[5]. Besides the krolak, and padberg problems (see Section 2), we have considered:

- The hamiltonian *tour* 318-cities problem of non-uniformly distributed cities of [24]. The optimum solution for this problem is reported in [31].
- The lattice-400 problem, consisting of 400 cities arranged in a 20 by 20 grid. This problem has been studied in the context of simulated annealing [19].

[5] The algorithms were run on the BBN Butterfly at the Iowa State University. We were able to use up to 90 processors.

- The 1000-cities problem of 1000 randomly distributed cities in a square. Problems of this size have been studied in the context of simulated annealing. To our knowledge, this is the largest problem yet attempted with a genetic algorithm.

In the top part of Figure 12, we show results of experiments run with a parallel version, in the style of [39], of the genetic algorithm with 2-opt and Or-opt of [18]. The *low* column contains results of this algorithm with a low amount of local improvement operation per generation. The *moderate* column contains results of this algorithm with a moderate amount of local improvement operation per generation. The x column indicates the amount of local improvements in the following sense. If $x = 0.1$ and we are working on a TSP of size s, then $0.1 * s$ are performed on each tour per generation. The *best* (*average*) column contains the percentage away from the optimum of the best (average) solution found by the PGA. The *time* columns give the time in seconds of the PGA run with the maximum number of available processors (typically 70) [6]. (In Appendix A, we show the best tours obtained with the PGA's for a selection of TSP's.) This figure clearly indicates the thesis that more local improvement per generation improves the quality of the solutions, but clearly at the expense of extra time.

It could be asked what the limit behavior is of a PGA with an "unlimited" amount of local improvements between generations. Such algorithms are PGA's without selection and crossover. We call such algorithms *independent strategies*, because they consist of running, in parallel, independent sequential local searches. In the bottom part of Figure 12, we show the results of the independent strategy on a selection of TSP's. Whereas on the smaller TSP's (i.e., size 100), the results are not significantly different, on the larger problems (above 500 cities), it is clear that crossover and selection do play a critical role in the performance of GA's for the TSP. It is also interesting to observe that on larger TSP's, PGA's with a low amount of local improvement yield better solutions than the independent strategies and this with better efficiency.

Finally, a word about speed-up. Our PGA is coded so that it can be simulated on a varying number of processors. This allows us to measure the speed-up of the algorithm as a function of the number of processors. Clearly, the more available processors, the closer our implementation comes to a true fine-grained distributed algorithm. Figure 13 shows the speed-up curve of the PGA applied to the krolak problem. (Similar curves can be obtained for all the other reported TSP's.) As can be seen, the speed-up curve is nearly linear up to 90 processors.

5 Parallelizing probabilistic sequential search algorithms

So far, we have concentrated on genetic algorithms and discussed how heuristic operations, such as 2-opt and Or-opt, can be successfully integrated. Originally these heuristics were used in *probabilistic sequential search algorithms*. These algorithms typically take the following format: [7]

1. Generate a random candidate solution.

2. Attempt to find an improved candidate solution by some heuristic transformation.

3. If an improved solution is found, then replace the previous candidate solution by this better one.

[6] In this respect, we would also like to point out that the majority of the time is spent in calculating double float square roots since for the large TSP's we were not able to distribute the distance matrix to each processor. If this could be done, all the algorithms would run a factor of 4 faster (this is a conservative estimate).

[7] More recent algorithms, such as simulated annealing [2,5,20,35] take essentially the same format.

Problem			Low				Moderate		
	x	Best	Avg	Time(in sec)		x	Best	Avg	Time(in sec)
lattice(100)	0.1	0.8	2.3	17		1.0	0.8	1.2	42
krolak(100)	0.2	0.5	2.1	17		3.0	0.0	0.5	70
318-cities	0.2	2.9	4.6	275		1.6	2.0	2.9	730
400-lattice	0.15	1.9	2.2	310		1.2	0.4	1.0	1060
padberg(532)	0.2	2.9	3.8	470		0.5	1.6	2.4	1070
1000-cities	0.2	2.9	3.1	2700		1.2	1.0	1.6	7000

Parallel genetic algorithms with low and moderate local improvement for various TSP's

Problem	Best	Avg	Time
lattice(100)	1.7	1.9	33
krolak(100)	0.0	0.2	73
padberg(532)	7.4	8.1	490
1000-cities	4.0	4.2	5500

The independent strategy for various TSP's

Figure 12: Performance of parallel genetic algorithms

4. Repeat from step 1 if the solution is not satisfactory and time permits [8].

An alternative way of looking at the results of parallel genetic algorithms would be to shift the attention to the sequential algorithm built around the local improvement operator and view the parallelism in the GA's as a technique to parallelize these sequential search algorithms. This is the viewpoint we will take in this section. We will focus on the 2-opt operator for medium-size TSP's (100 cities) (For a more detailed discussion see [17].)

The simplest way to parallelize probabilistic sequential search algorithms is to let the processors work independently. That is, each processor randomly generates an initial candidate solution and then repeatedly applies the 2-opt operator. At the end, that is after a prealloted time has expired, all processors stop and the best available solution is taken as the solution of the parallel algorithm. This is called the *independent strategy*[9]. A potential problem with this strategy is that some processors may get caught in local optima or may simply search the wrong areas of the search space. Intuitively, it seems likely that we may do better if we let the processors work independently for a while, then exchange information about "good" candidate solutions, then again work independently for a while, then again exchange information and so on. Clearly there are many ways to exchange information and thus there are many such strategies. Such a strategy will be called an *interdependent strategy* and the processing time between information exchanges is called a *generation*. One way to exchange information is to redistribute a certain number of best candidate solutions after each generation. More sophisticated strategies would involve exchanging structural parts of candidate solutions as with crossover operators of genetic algorithms.

[8]Step 2 will be called an *attempt at local improvement* or a *trial*. If an improved solution is found in a trial we shall say that the trial has been *successful*.

[9]This kind of strategy can also be found in the work of [4,28,35] where it is investigated as a technique to see how genetic algorithms differ in quality and time of performance compared to simply running a certain number of sequential algorithms and taking as solution the best obtained local optima. It turns out that GA's perform better than such algorithms.

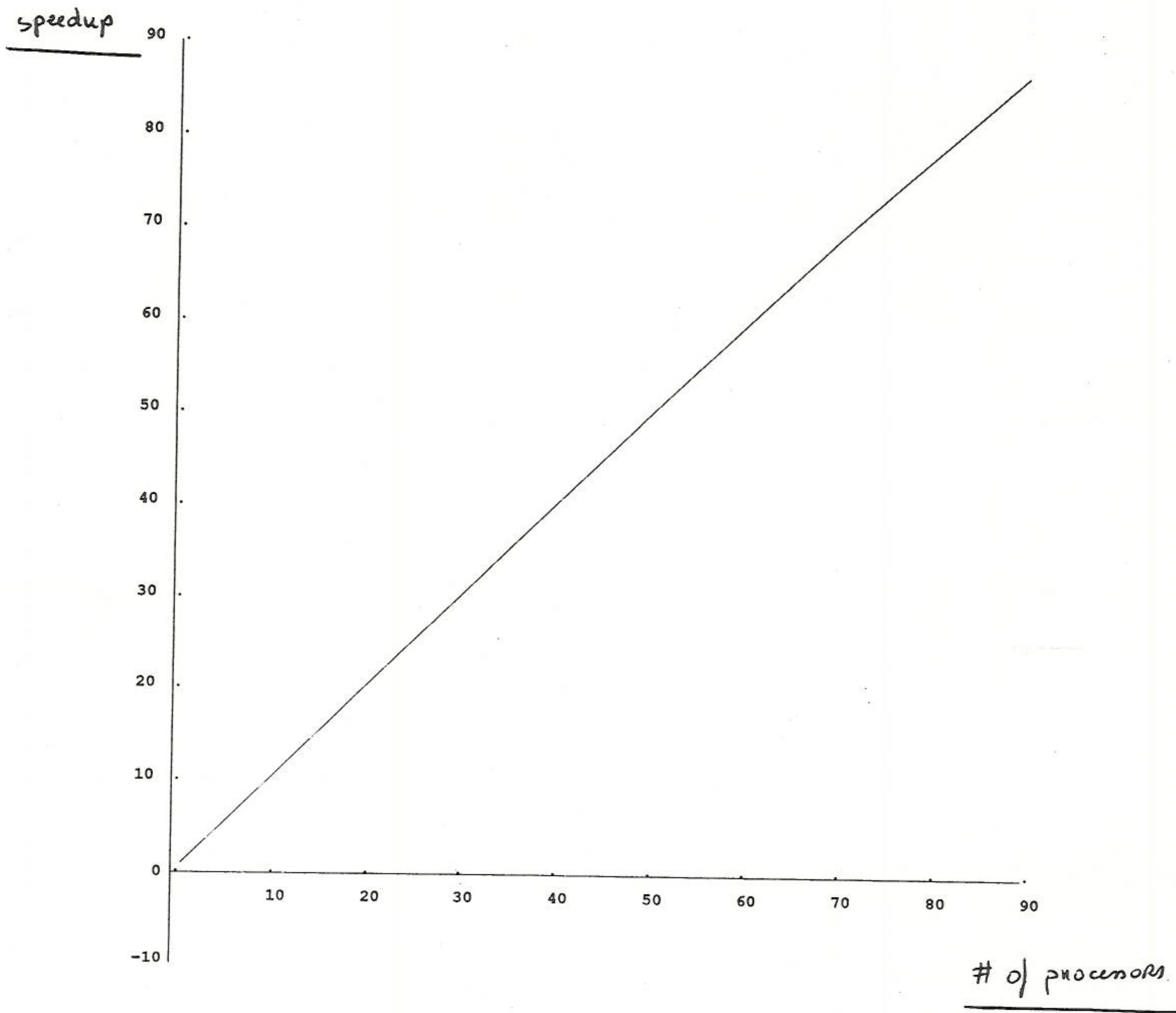


Figure 13: Speed-up curve for the lattice problem

The independent strategy has the advantage that each processor is able to devote its entire time to attempting local improvements. In the interdependent strategies the processors spend some time exchanging information and thus in the same total time, they attempt a smaller number of local improvements than the independent strategy. But information exchange may help the parallel algorithm as a whole to search the space more efficiently. However if too much time is spent in communication, then the performance will obviously degrade, since any randomly chosen candidate solution will require a certain number of successful local improvements to achieve a solution close to the optimum. We will indicate the tradeoffs involved in communication versus working in isolation and try to come up with some preferred strategies for the parallelization of probabilistic sequential search algorithms.

To compare the various approaches, it is necessary to introduce several performance measures: (1) *accuracy*, in terms of how good are the optimums obtained, (2) *efficiency*, that is how much time is taken for the execution, (3) *robustness*, that is how confident are we of obtaining good solutions repeatedly and (4) *speedup*, which is the ratio of efficiency of the sequential algorithm and the efficiency of the parallel algorithm. Probabilistic sequential search algorithms usually follow a principle of diminishing returns in the sense that getting values very close to the optimal may take an extremely long time, and so much accuracy may not be necessary. Hence for the following study, we will define efficiency as the time required to come up with a tour within 10% of the global optimum[10].

5.1 The Independent Strategy

As stated above in the independent strategy each processor randomly generates an initial tour and repeatedly applies the 2-opt operator until the given time expires.

The independent strategy shows a speedup of only about 2 as the number of processors is increased from 1 to 80. It therefore seems that the additional processors are not being utilized in a meaningful way. However, this is not true. With one processor about 40% of the experiments fail, in the sense that by the end of given time they are unable to come up with a tour which is within 10% of the global optimum. (Our definition of efficiency in the previous section implies that in such cases efficiency can not be measured.) Therefore the parallel algorithm is not very robust if it has only a few processors. After the number of processors increases to above 20, no experiments fail and the algorithm is now robust. The accuracy of the algorithm improves as the number of processors increases and with about 80 processors we can get an accuracy of within 5% of the optimum.

Even with 80 processors up to 30% of the processors fail to find good optimums and the good accuracy is only a result of the using a large number of processors. So some processors are not doing any useful work and we may be wasting processor power. We will next attempt to make all processors search useful regions of the search space by exchanging information at regular intervals.

5.2 Redistributing the Best

A simple way of exchanging information is to redistribute the one best tour from among all processors at regular intervals. We shall call this the *One Best Strategy*. The important variable to adjust here is the *time per generation*, or *tpg*. As *tpg* increases information exchange is done less

[10] The results we give are from experiments done on a BBN Butterfly at the University of Maryland. We chose the Butterfly because it is a nice example of a shared memory MIMD machine and because on the Butterfly the processors can easily be made to work, when required in isolation from each other. (For example, in the independent strategy when each processor accesses its tour to improve it, that access does not interfere with another processor trying to access its tour.)

often. (If *tpg* is equal to the total time then we have the independent strategy.) We found that low values of *tpg* make the algorithm more efficient. (About four times faster than the independent strategy.) However, the accuracy worsens significantly. Not only that but the robustness fails to improve as the number of processors is increased. The speedup curve is now close to logarithmic until about 16 processors and then it flattens out.

These results can be explained as follows: By redistributing the one best tour very often, all processors concentrate on the tours that currently looks to be "good". When all processors try to improve this tour at least one processor is often able to find a big improvement. Hence the efficiency improves significantly. However, a tour that looks to be the best in a particular generation may not be a good one in the long run. It is possible that it may just lead to a local optimum, and so the one best strategy could get trapped in a local optimum. This phenomenon of getting trapped in a local optimum is called *premature convergence*. It also explains why the accuracy is significantly worse than the independent strategy.

We can avoid premature convergence by redistributing the k best instead of the one best strategy. We shall call this the *k Best Strategy*. We find that as k increases the efficiency worsens, but the robustness improves. With $k = 8$ and 16 or more processors the algorithm is very robust. However, the accuracy remains significantly worse than the independent strategy.

So while we have been able to improve the efficiency and maintain robustness by the k best strategy, we have not been able to match the accuracy of the independent strategy. Clearly, to improve the accuracy and efficiency, as well as to get good speedup we need a more sophisticated strategy of information exchange.

5.3 The Selection Strategy

Holland has shown [16] that in the absence of any disruptive operators the strategy of allocating exponentially more trials to the observed best is optimal in searching good regions of the search space. This strategy is called the *selection strategy*. The selection strategy combines the idea of creating multiple copies of good candidate solutions so that a push is given towards what seems to be the global optimum, as well as the idea of avoiding premature convergence by keeping around a few currently bad looking candidate solutions.

We tested this *global* selection strategy as a strategy of exchanging information after each generation. In terms of speedup results, the selection strategy performs very poorly beyond about 16 processors. The reason for this is that it requires a global synchronization of all processors and the selection strategy spends a considerable amount of time in redistributing the tours. Luckily, this situation can be resolved by instead of considering global selection, it is possible to implement selection as local processes. In this way it is possible to both get quality solutions with speedups as seen in the k best strategy.

Finally, to improve the robustness and quality of the interdependent strategies, one can introduce structural exchange operators, yielding of course the parallel genetic algorithms studied in Section 3.

6 Conclusion

In this paper, we have given a review of parallel genetic algorithms applied to the traveling salesman problem. Our main conclusion is that such algorithms make excellent approximation algorithms for this problem when compared with more standard sequential heuristic algorithms. This is especially the case on large TSP's. Since the PGA's lend themselves naturally to parallelism, such algorithms can also be efficiently run on parallel machine.

This paper however leaves open some research issues:

- As indicated, the crossover plays a role in the quality of tours obtained with PGA's. It would be interesting to more thoroughly study its effect. As mentioned before, we expects the role of crossover to gain importance with growing size TSP's.
- Even PGA's which use low amounts of local improvement can yield good results. An interesting question would be to determine how small we can make this amount (thereby improving the efficiency of the algorithm) and still get reasonable tours.
- Although we have purposely selected TSP's with different structural properties (for example, compare the random cities problems and the lattice problems) we did not thoroughly determine if these properties play a critical role in the quality of the PGA's. We conjecture that indeed such structural properties can be important in the outcome of the algorithms.

We plan to address these issues in a forthcoming paper.

References

[1] J. Beardwood, J.H. Halton, and J.M. Hammersley. The shortest path through many points. *Proc. of Cambridge Philos. Soc.*, 55:299–327, 1959.

[2] E. Bonomi and Jean-luc Lutton. The n-city traveling salesman problem: Statistical mechanics and the metropolis algorithm. *SIAM Review*, 26(4):551–568, October 1984.

[3] D.G. Bounds. New optmization methods from physics and biology. *Nature*, 329(17):215–219, September 1987.

[4] R.M. Brady. Optimization strategies gleaned from biological evolution. *Nature*, 317(31):804–807, October 1985.

[5] V. Cerny. Thermodynamical approach to the traveling salesman problem. *Journal of Optimization Theory and Application*, 45(1):41–52, January 1985.

[6] J.P. Cohoon, S.U. Hedge, W.N. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 148–154, July 1987.

[7] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of 9th IJCAI*, pages 162–164, August 1985.

[8] L. Davis. Job shop scheduling with genetic algorithms. *Proc of an International Conference on Genetic Algorithms*, pages 136–140, July 1985.

[9] K.A. Dejong. Adaptive system design: A genetic approach. *I.E.E.E. Trans. on Systems Man and Cybernetics*, SMC-10(9):556–574, September 1980.

[10] D. Goldberg. *Genetic Algorithms in Serahc, Optimization and Machine Learning.* Addison-Wesley, New York, 1988.

[11] D. E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 154–159, July 1985.

[12] M. Gorges-Schleuter. Asparagos an asynchronous parallel genetic optimization strategy. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 422–427. Morgan Kaufmann, July 1989.

[13] J. J. Grefenstette. Parallel adaptive algorithm for function optimization. Technical Report CS-81-9, Vanderbilt University, November 1981.

[14] J. J. Grefenstette. Incorporating probelm specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[15] J. J. Grefenstette, R. Gopal, B. J. Rosmaita, and D. Van Gucht. Genetic algorithms for the traveling salesman problem. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms*, pages 160–168, July 1985.

[16] J. H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[17] P. Jog. *Parallelization of probabilistic sequential search algorithms*. PhD thesis, Indiana University, 1989.

[18] P. Jog, J.Y. Suh, and D. Van Gucht. The effect of local improvement, selection and crossover on a genetic algorithm for the traveling salesman problem. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms*. Morgan Kauffman, 1989.

[19] S. Kirkpatrick. Optimization by simulated annealing:quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1983.

[20] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[21] P. D. Krolak, W. Felts, and G. Marble. A man-machine approach toward solving the traveling salesman problem. *Communications of the ACM*, 14:327–334, 1971.

[22] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem*. Wiley Interscience, 1985.

[23] G.E. Liepins and Hilliard. Greedy genetics. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and their applications*, pages 90–99, July 1987.

[24] S. Lin and B.W. Kernighan. An efficient heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[25] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 428–433. Morgan Kaufmann, July 1989.

[26] H. Muhlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 416–421. Morgan Kaufmann, July 1989.

[27] H. Muhlenbein, M. Gorges-Scheulter, and O. Kramer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 4:269–279, 1987.

[28] H. Muhlenbein, M. Gorges-Scheulter, and O. Kramer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7:70–85, 1988.

[29] I.M. Oliver, D.J. Smith, and J. Holland. A study of permutation crossover operators on the traveling salesman problem. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 224–230, July 1987.

[30] I Or. *Traveling Salesman-Type Combinatorial Problems and their relation to the Logistics of Regional Blood Banking.* PhD thesis, Northwestern University, 1976.

[31] W. Padberg and G. Rnaldi. Optimization of a 532-city symmetric tsp. *Operation Research Letters*, 6:1–7, 1987.

[32] C.P. Pettey and M.R. Leuze. A theoretical investigation of a parallel genetic algorithm. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 398–405. Morgan Kaufmann, July 1989.

[33] C.P. Pettey, M.R. Leuze, and J.J. Grefenstette. A parallel genetic algorithm. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 155–161, July 1987.

[34] B. Rosmaita. Exodus, an extension of the genetic algorithm to deal with permuations. Master's thesis, Vanderbilt University, 1985.

[35] Y. Rossier, M. Troyon, and T.M. Liebling. Probabilistic exchange algorithms and the euclidean traveling salesman problem. *OR Spektrum*, 8, 1986.

[36] A.V. Sannier and E.D. Goodman. Genetic learning procedures in distributed environments. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 162–169, July 1987.

[37] D.J. Sirag and P.T. Weisser. Toward a unified thermodynamic genetic operator. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 116–122, July 1987.

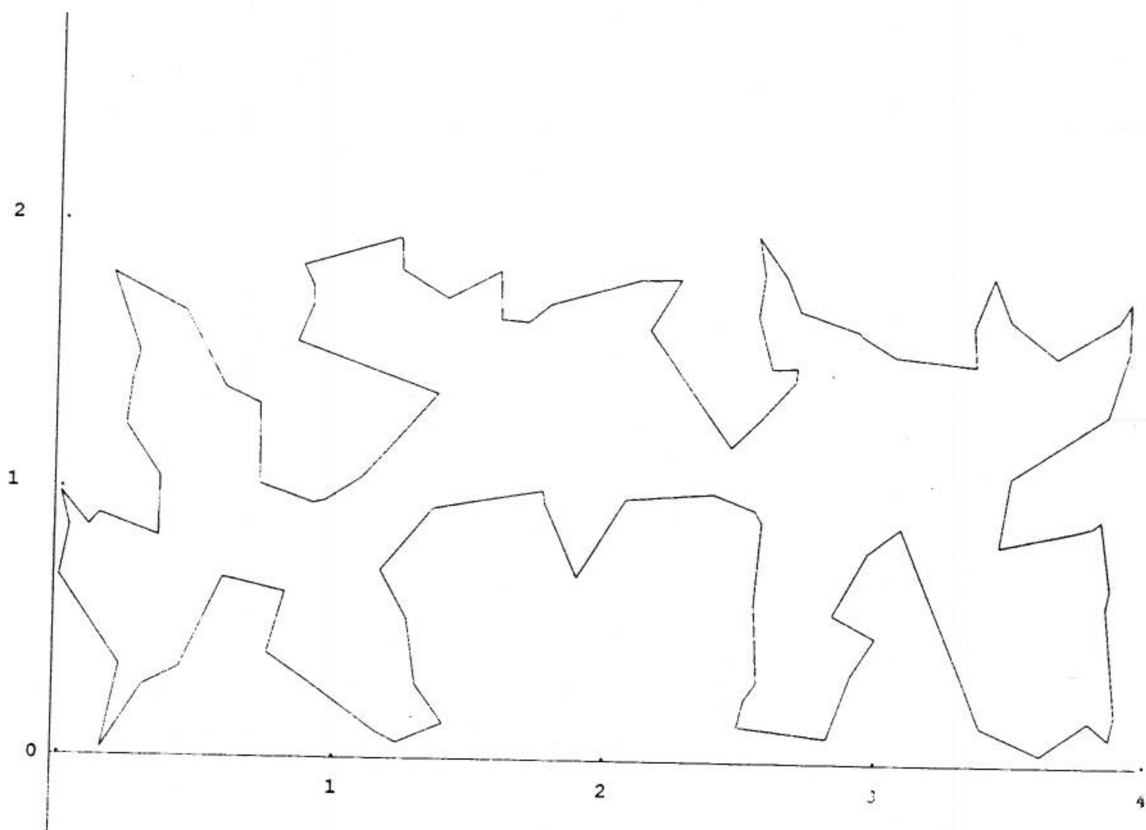
[38] D Smith. Bin packing with adaptive search. In Grefenstette J.J., editor, *Pproceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 202–206, July 1985.

[39] J.Y. Suh and D. Van Gucht. Distributed genetic algorithms. Technical Report 225, Indiana University, July 1987.

[40] J.Y. Suh and D. Van Gucht. Incorporating heuristic information into genetic search. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 100–107, July 1987.

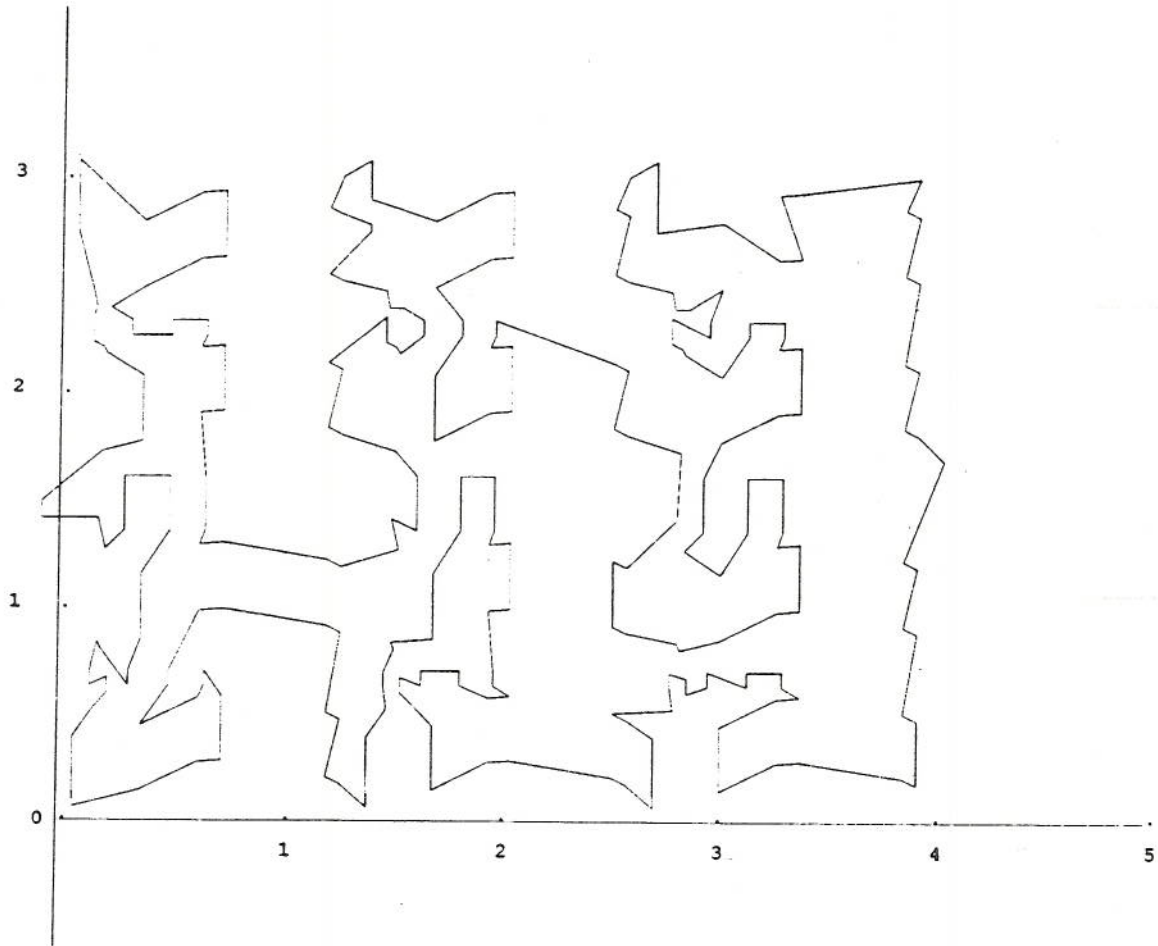
[41] R. Tanese. Parallel genetic algorithms for a hypercube. In Grefenstette J.J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 177–183, July 1987.

[42] R. Tanese. Distributed genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 434–440. Morgan Kaufmann, July 1989.



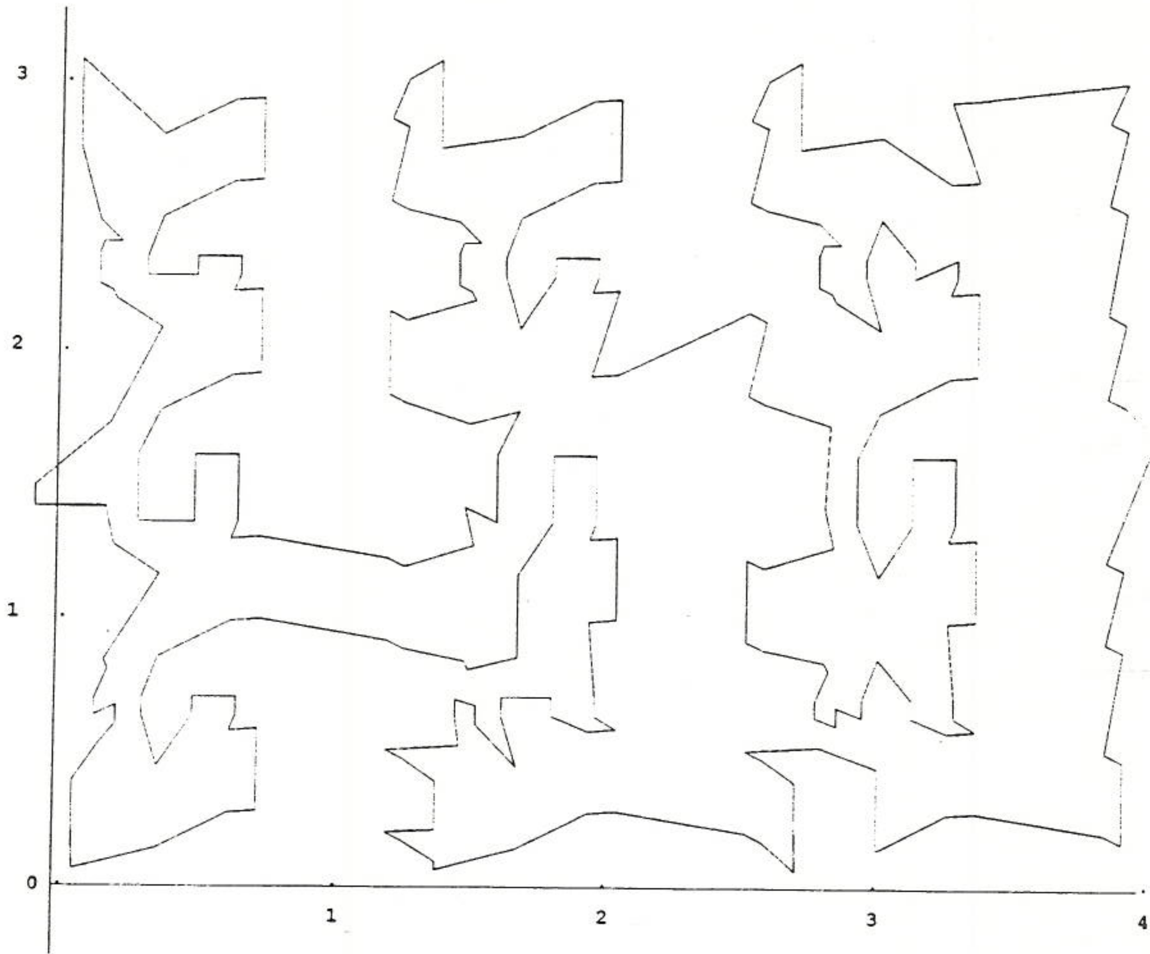
Length = 21285 (0.0%)

Figure 14: Best tour with moderate local improvement for the krolak problem



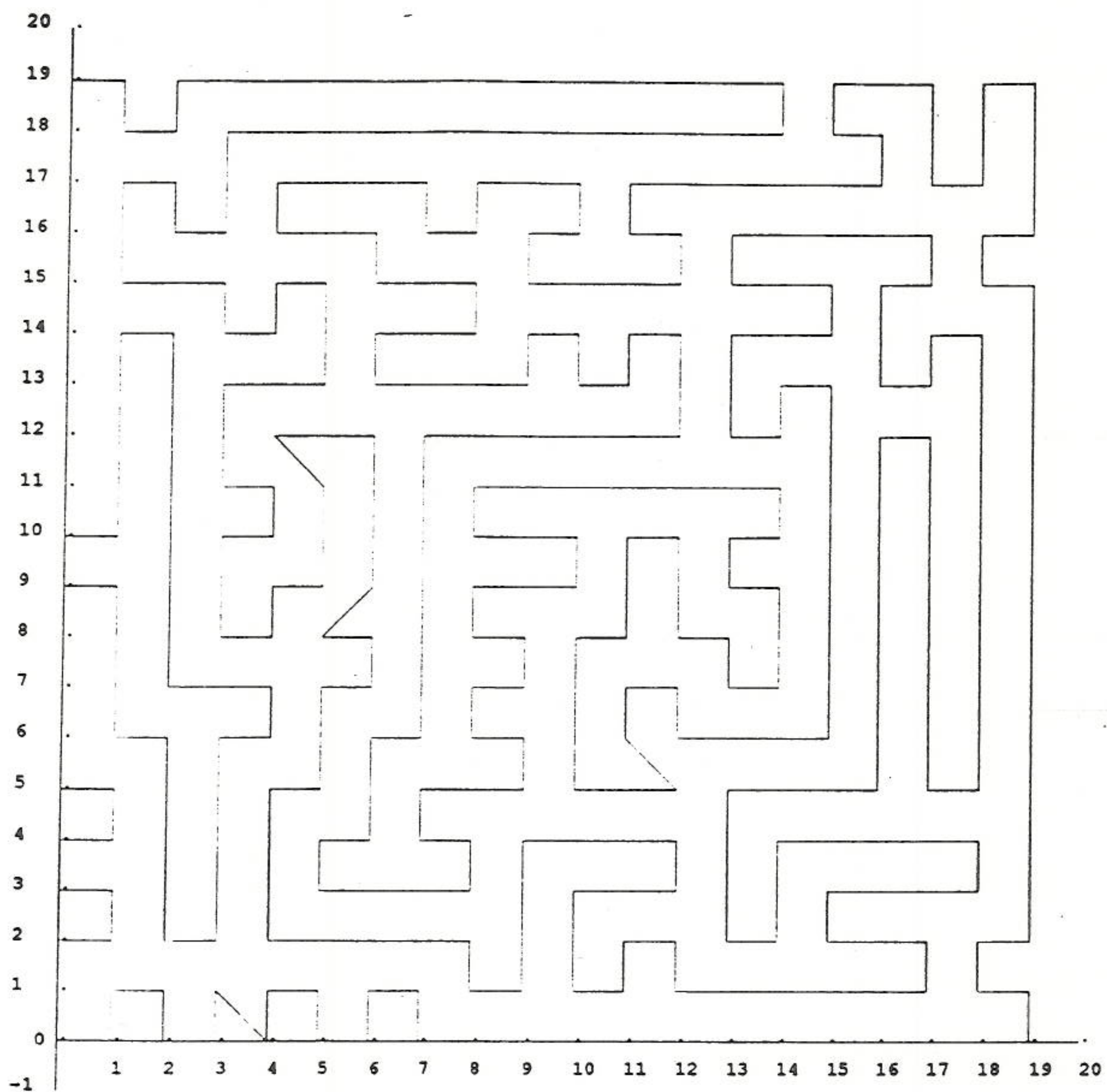
Length = 43600 (2.9%)

Figure 15: Best tour with low local improvement for the 318-cities problem



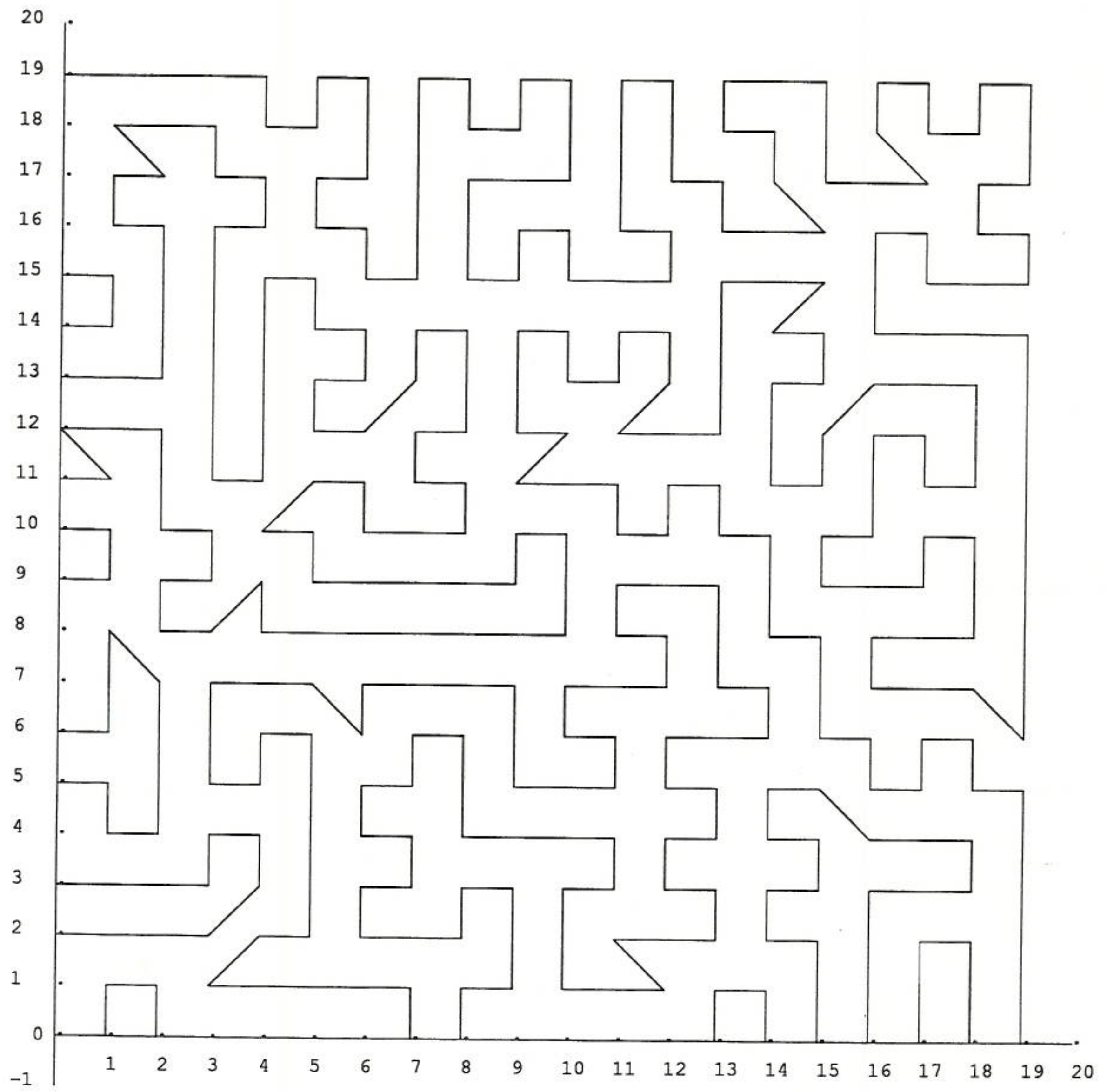
Length = 42856 (2.0%)

Figure 16: Best tour with moderate local improvement for the 318-cities problem



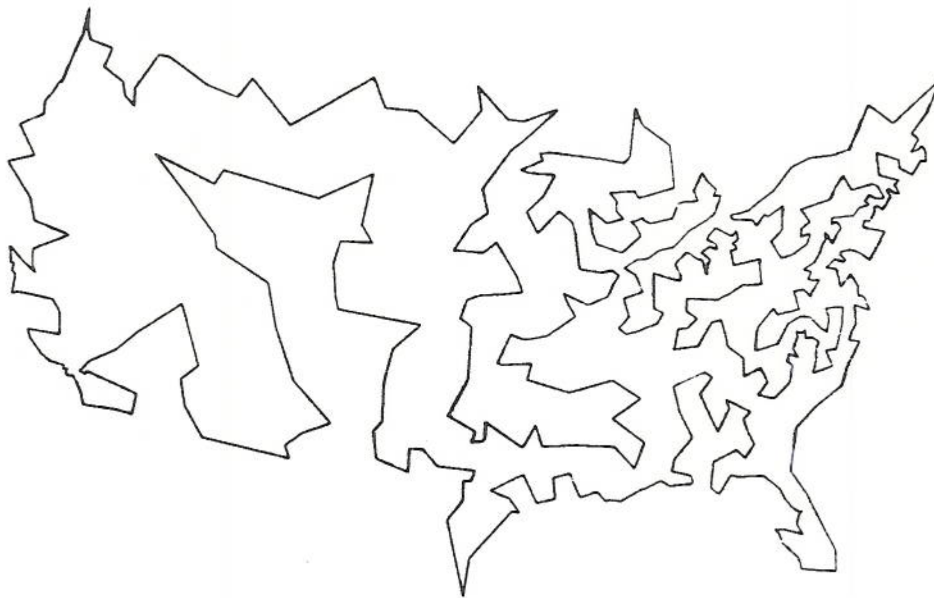
Length = 401.66 (0.4%)

Figure 17: Best tour with low local improvement for the 400-lattice problem



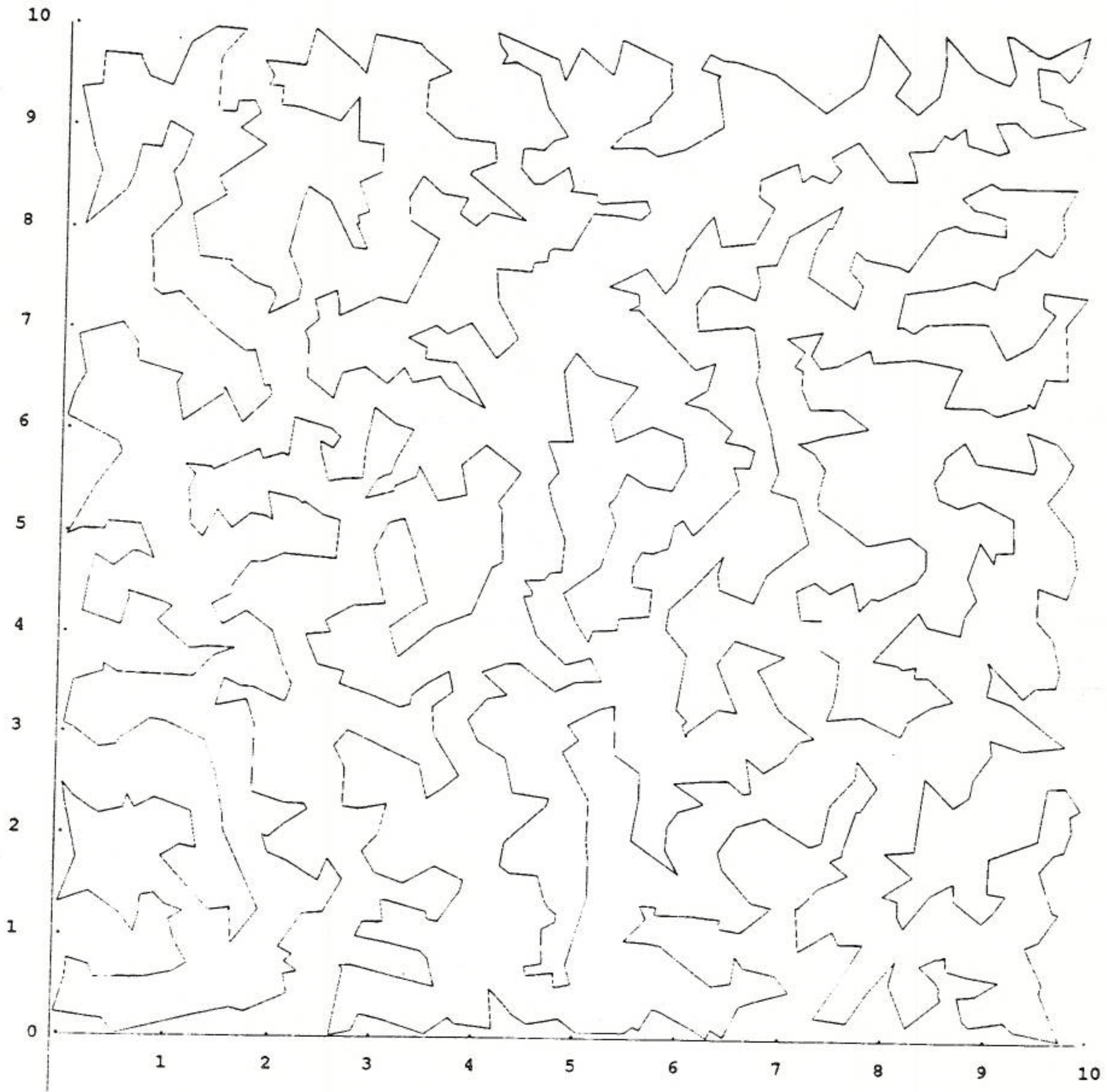
Length = 407.56 (1.9%)

Figure 18: Best tour with moderate local improvement for the 400-lattice problem



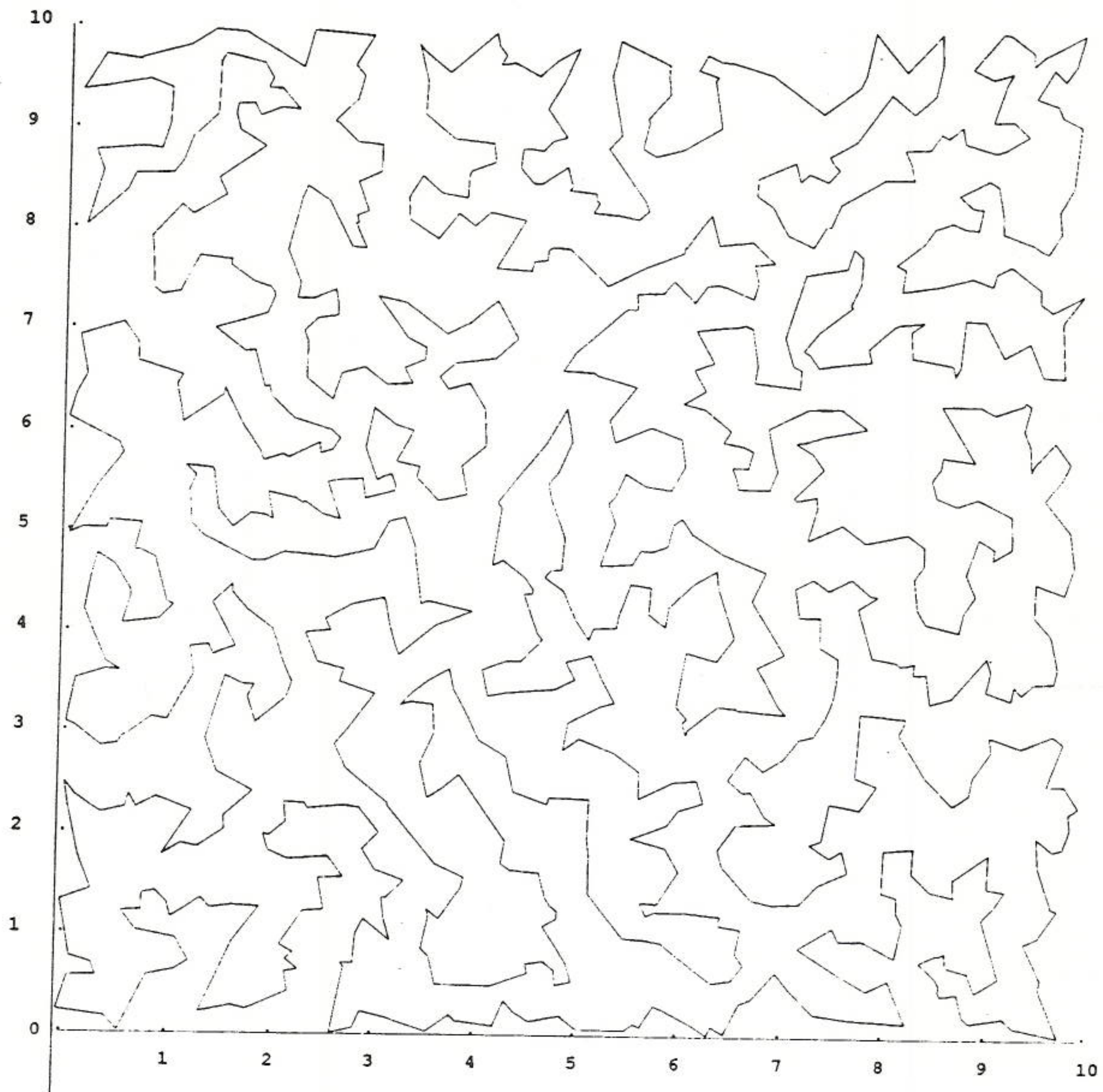
Length = 28118 (1.6%)

Figure 19: Best tour with moderate local improvement for the padberg problem



Length = 243.79 (2.9%)

Figure 20: Best tour with low local improvement for the 1000-cities problem



Length = 239.55 (1.2%)

Figure 21: Best tour with moderate local improvement for the 1000-cities problem