1991

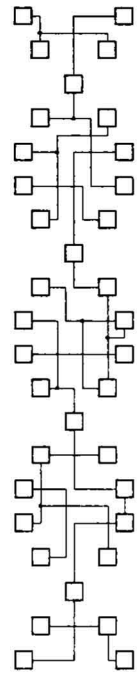
CPGA: a two-dimensional, order-based genetic algorithm for cell placement

John T. Cooklis

CPGA:
A Two-Dimensional, Order-Based Genetic Algorithm for Cell Placement

by

John T. Cooklis



Thesis Committee:

Al Biles, Associate Professor

George A. Brown, Professor

Dr. Tony Chang, Professor

9/10/91

CPGA: A Two-Dimensional, Order-Based Genetic Algorithm
for Cell Placement

I, John T. Cooklis, hereby deny permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Table of Contents

Table of Figures

Chapter 1:
Introduction

Genetic algorithms, like simulated annealing and neural networks, are an attempt to model a naturally-occurring process and apply it to artificial problem domains. The direct result of John Holland's pioneering study of adaptation [HOLL75], genetic algorithms represent an optimization technique that is particularly well-suited to large, noisy search spaces with many local extrema. Efficient search of such difficult search spaces is accomplished through a balance between exploitation of existing solutions and exploration of the search space to yield new solutions.

Traditional genetic algorithms are based on *chromosomes*, which (in the context of genetic algorithms) are strings of symbols. The goal of this project is to investigate the extensibility of the chromosome and its corresponding genetic operators to two dimensions. This extension is made in an effort to adapt the algorithm to a two-dimensional optimization problem. While this extension further decreases the similarity between natural and artificial genetics, the underlying principles of evolution still apply.

1.1 Cell Placement

A familiar problem that arises in VLSI design automation is that of *cell placement*. Given a set of interconnected functional blocks (cells), the placement problem consists of finding an arrangement of the cells that minimizes the total length of the interconnections (nets). Achieving this

goal offers several advantages, including lower interconnect impedance, improved signal quality, and smaller signal delays. Additionally, a good placement simplifies the task of *routing*, the next stage of design. Routing involves defining the paths of the nets through the interconnect space between the cells. A good placement minimizes wasted interconnect space, which is a precious resource to VLSI designers.

Consider the cell placement shown in Figure 1-1. The importance of effective utilization of interconnect space is apparent even in this simple 3 x 3 array. Problems arise near more heavily-connected cells, such as the cells numbered 3, 4 and 5.

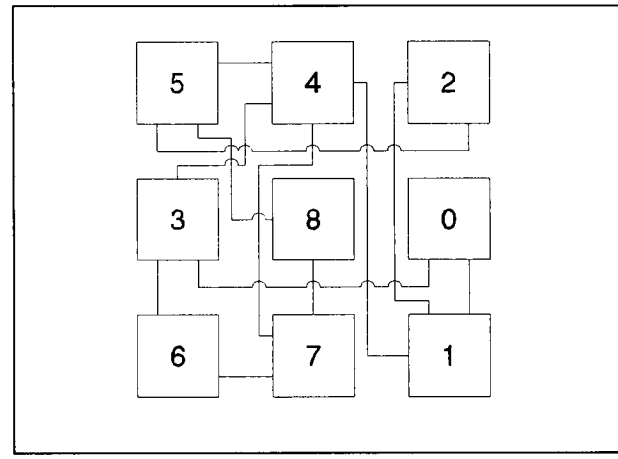


Figure 1-1 Poor cell placement

Figure 1-2 shows an optimal placement of the 9 cells. The net connectivity of the design has been preserved; only the relative positions of the cells has changed.

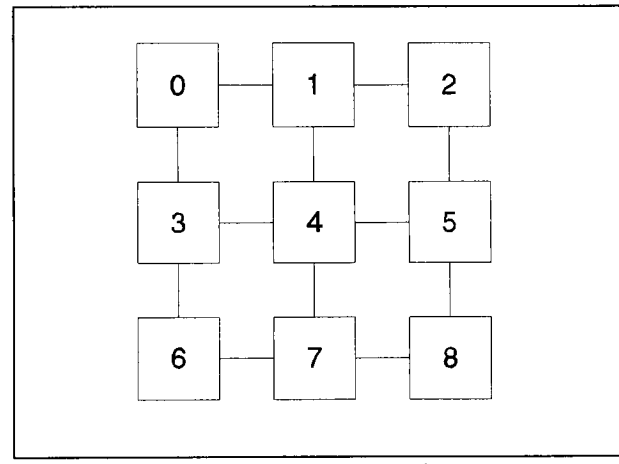


Figure 1-2 Optimal cell placement

In a rectangular array of regular cells, placement may be defined as the search for an optimal permutation of the cell positions. This definition is allowed by the regularity of the cell array, which permits a given cell to occupy any position. The placement problem is clearly much more complex for designs in which the cells are not uniform in size and shape.

1.2 Problem Statement

In this project, an order-based genetic algorithm with two-dimensional chromosomes and genetic operators is developed for the cell placement problem. It is tested extensively with a set of problems for which optimal solutions are known in order to "tune" various parameter values and provide information on the performance of the algorithm.

The specific problem that will be investigated is placement of an 6 x 6 array of cells. Given a description of the interconnections, the Cell Placement Genetic Algorithm (*CPGA*) will attempt to find a permutation of the cell positions that minimizes the total length of the nets.

The size of the search space confronting the genetic algorithm in this case is the number of permutations of the 6 x 6 array:

$$36! = 3.72 \times 10^{41}$$

It will be shown that there is a placement problem for which only 8 optimal solutions exist in this search space.

The magnitude of this problem becomes apparent when considering an iterative search of the domain. If it were possible to generate and test 1 trillion cell placements per second (clearly impossible with current technology), an exhaustive search would take approximately 10^{22} years to generate all possible solutions. The Big Bang is estimated to have occurred on the order of 10^{10} years ago.

It is usually not necessary to find an optimal solution to a given placement problem; it is sufficient to find a solution which satisfies a set of requirements related to routability and signal quality. Genetic algorithms are appropriate for this type of problem since they tend to find adequate solutions quickly. In contrast, more traditional calculus-based optimization techniques focus on finding optimal solutions without regard for intermediate solutions which are "good enough."

1.3 Modelling the Problem

The cell placement problem will be modelled with a rectangular array of regular cells. The distance between the center of a cell and the center of its horizontal or vertical neighbor will be 1 "unit."

The length of a net between two cells will be computed as the *Manhattan distance* between the centers of the cells; that is, the sum of the horizontal and vertical distances. This is a fairly accurate approximation of the actual net length, since routed nets are typically orthogonal.

Nets may be weighted to reflect critical or multiple nets connecting pairs of cells. The connections and their corresponding weights will be specified by the user.

1.4 Overview

Chapter 2 introduces relevant background information on genetic algorithms, particularly order-based genetic algorithms such as those used to solve the traveling salesman problem. Its focus is on the extension of genetic algorithms to two dimensions.

Chapter 3 provides implementation details of *CPGA*. This chapter gives a high-level description of the C++ classes developed.

Chapter 4 presents the output of *CPGA* for a variety of test cases. The optimization of parameter values is discussed, as well as the behavior of the system and its performance.

Chapter 5 examines the suitability of genetic algorithms to the cell placement problem and suggests extensions and improvements to the system.

Chapter 2:
Background

In a genetic search, a population of potential solutions (*individuals*) evolves over time subject to some *adaptive plan*. As defined by Holland [HOLL75], the task of this adaptive plan is to produce individuals with high *fitness*; that is, individuals that are well-suited to the environment facing the adaptive plan at the time. Although this environment changes with time in natural systems, it is typically stationary in artificial evolution.

The adaptive plan determines the application of genetic operators that modify existing structures to yield new, more highly fit ones. Adaptation, then, involves the repeated application of these operators on a population of structures to yield "better" structures.

This chapter presents the fundamental concepts of adaptation in *CPGA*. A high-level description of the algorithm is presented, followed by a description of the representation and genetic operators used. In addition, an analysis of *schemata* processing by the algorithm is discussed.

2.1 Genetic Algorithm Structure

Pseudocode for the algorithm used by *CPGA* is shown below. It is based primarily on the work presented in [GOLD89a].

initialize the population P with randomly-generated individuals

repeat

evaluate all individuals in P using an objective function

apply a transformation to the objective function value of each individual to yield a fitness value for the individual (scaling)

copy individuals from P into mating pool M based on their fitness values (selection/reproduction)

select pairs of individuals from M and perform crossover with probability P_x to yield pairs of new offspring

perform mutation on the new offspring with probability P_m

insert the new offspring into population P, replacing the old contents of P

until some termination criterion is satisfied

One iteration of the repeat loop is termed a *generation*. Common criteria for termination of the algorithm are: 1) n generations have elapsed, where n is specified by the user; 2) an individual has been found whose objective function value satisfies some threshold value; and 3) the population has *converged* on a particular structure. Some combination of these conditions may be used to specify termination.

Part of the efficiency of the genetic algorithm lies in the fact that by maintaining a population of solutions, it explores many areas of the search space simultaneously. Additionally, good individuals are exploited by selection, crossover, and mutation to produce better ones. Although these operations are randomized, genetic search is not a random search.

2.2 Representation

The structures that evolve in a genetic algorithm are based on codings of the parameter space for the particular problem being solved rather than on the parameters themselves [GOLD85b]. These codings are typically strings of symbols over some small alphabet; strings of binary digits are the most frequently-used representation. The genetic operators are then designed to manipulate individuals coded with the chosen representation.

The string representation is analogous to *chromosomes* in nature. Chromosomes are strings of *genes* defining the characteristics of the individual containing them. The *locus* of a gene is its position within the chromosome. Each gene has several possible values, or *alleles*. A *phenotype* is a particular combination of these allele values.

A diverse population of individuals (phenotypes) may be thought of as a *gene pool*. During reproduction, *crossover* causes substrings of the parents' chromosomes to be swapped, yielding a new individual with some of the characteristics of both parents. *Mutation* causes random changes to occur to genes' allele values, and it typically occurs with a very small probability.

By processing chromosomes, which are codings of the parameter space, genetic algorithms search the "gene space" of a problem rather than the parameter space [REND85]. This characteristic makes genetic

algorithms insensitive to properties of the search space, such as continuity and the existence of derivatives, on which other optimization techniques rely.

2.1.1 Order-Based Representations

One of the most popular combinatorial optimization problems of the last several decades has been the Traveling Salesman Problem (TSP). The goal of the TSP is to find the shortest tour through a set of N cities that visits each city exactly once (i.e. the shortest Hamiltonian path). The TSP belongs to a class of problems considered to be NP-complete; no polynomial-time solution is believed to exist [GOLD85a]. Attempts to solve the TSP with genetic algorithms have generally been successful.

If the cities are labelled with integers from 1 to N, the most straightforward chromosomal representation for the problem is a list of these labels in the order in which the corresponding cities are visited. This may be thought of as a *permutation* or *order-based representation.*

Chromosomes in this representation must be valid permutations of the list of cities. Therefore, the crossover and mutation operators must ensure that the chromosomes resulting from their application are still valid permutations. This restriction does not exist for genetic algorithms that are not order-based, since all possible strings are valid.

2.1.2 Placement Representation for CPGA

The representation chosen for *CPGA* is the most natural one for the problem: a two-dimensional, order-based coding corresponding to the relative positions of cells in the placement. With the cells numbered from 0 to *N*-1, a chromosome is a rectangular array of the cell numbers. A particular phenotype must be a valid permutation of cell numbers. Every individual in the population represents a possible cell placement.

Although the specific cell placement problem investigated in this project is square (6 x 6), this is not a restriction of the algorithm. The operators are capable of processing rectangular chromosomes as well.

2.3 The Objective Function

The objective function in a genetic algorithm is used to determine the performance of individuals in the population. This is the only feedback that the adaptive plan receives from the environment. The goal of genetic algorithms is to find individuals that maximize (or minimize) the objective function value.

The objective function in *CPGA* computes the total length of the nets for the individual (i) being evaluated:

(2.1)

$$o_i = \sum_{j=0}^{n} w_j \ell_j$$

where:

o_i = objective function value of individual i
n = total number of nets in the chromosome
w_j = weight of net j
ℓ_j = length of net j

The goal of *CPGA* is to minimize this objective function.

The weight of a net is an integer value indicating the criticalness of the net. It may also be used to specify multiple nets connecting the same pair of cells. The weights of the nets are specified by the user.

The length of a net is computed as the Manhattan distance between the centers of the cells connected by the net, where the distance between a cell and its horizontal or vertical neighbor is defined to be 1. As noted earlier, this is a good approximation of the length of routed nets.

2.4 Fitness

As in nature, reproduction of individuals in a genetic algorithm is based on their *fitness*. The more highly-fit individuals in the population survive to produce more offspring than the less fit individuals. Although it may be possible to use the objective function value itself as a fitness value, it is most often the case that a transformation must be applied to this value first. Davis refers to these transformations as *fitness techniques*

[DAVI91]. Fitness techniques are required when the objective function is to be minimized (as in *CPGA*) or the objective function may take on negative values [GREF89].

2.4.1 Raw Fitness

Because the more highly-fit individuals in *CPGA* have lower objective function values, the following function is applied to the objective function value of each individual to produce a raw fitness value:

(2.2)

$$r_i = \frac{1}{o_i}$$

where:

r_i = raw fitness value of individual i
o_i = objective function value of individual i

The least-fit individual in the population will have the lowest unscaled fitness value, while the most fit individual will have the highest value.

Since the initial population in the genetic algorithm is generated randomly, it is likely that the entire population will have mediocre fitness values early in the genetic search. The best individuals will have fitness values that are only slightly better than the worst individuals. In order to properly exploit the more fit individuals, the fitness values need to be scaled such that the better performers "stand out" more.

The opposite extreme occurs late in the genetic search, when a few "super individuals" may dominate the population. In this case, the difference in fitness between the best and worst individuals needs to be reduced to prevent the less fit individuals from being lost completely, taking any potentially useful genetic material with them. This undesirable phenomenon is referred to as *premature convergence* [GOLD89a].

Both of these problems may be avoided with proper scaling of the fitness values. Fitness scaling serves to regulate the competition among members of the population. The particular scaling method used in this project is *linear scaling*.

2.4.2 Linear Fitness Scaling

The goal of linear scaling is to control the ratio of the best fitness to the average fitness. This ratio is referred to as the *scale factor*. Scaling with a constant scale factor governs selection based on fitness, stabilizing the convergence properties of the genetic algorithm.

The effect of linear scaling is illustrated in Figure 2-1, where a line is drawn between the raw fitness values of the worst (least fit) and best (most fit) individuals. All individuals in the population fall somewhere along this line, with the average individual typically near the middle. Linear scaling attempts to pivot this line about the average fitness value until the ratio of the maximum fitness value to the average fitness value is equal to the scale

factor. As indicated by the arrows in the figure, the line in the case shown is pivoted clockwise so that the fitness of the above average individuals is decreased and the fitness of the below average individuals is increased. This operation is likely to occur late in the genetic search when a few individuals have fitness values that are much higher than average.

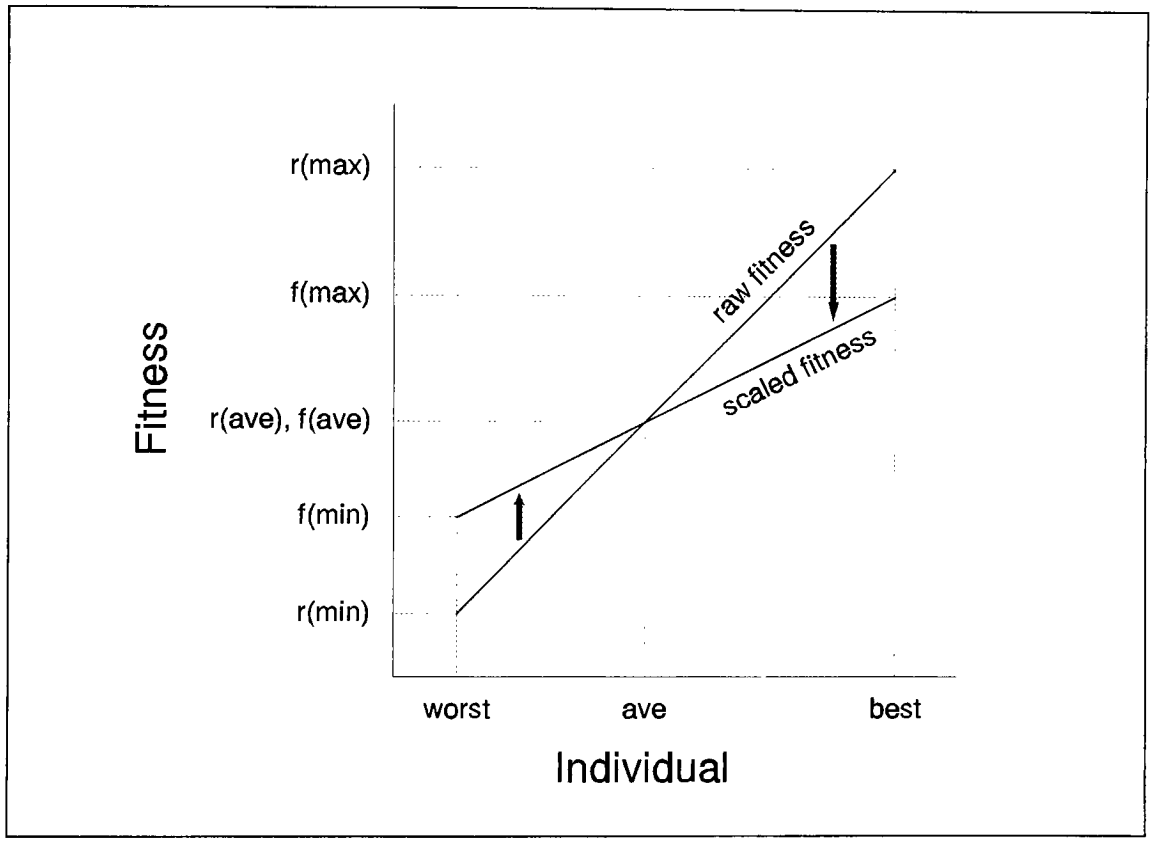


Figure 2-1 Linear fitness scaling

The linear scaling operation is subject to the constraint that the fitness of the worst individual may not be negative. If pivoting to achieve the desired scale factor would violate this constraint, the line is pivoted as much as possible. In this case, the actual scale factor used is less than the desired scale factor.

The net effect of pivoting the fitness line is to augment each individual's fitness by an amount proportional to the difference between the individual's fitness and the average fitness. This is shown in (2.3):

(2.3)

$$f_i = r_i + a(r_i - r_{ave})$$

where:

f_i = scaled fitness value of individual i
r_i = raw fitness value of individual i
r_{ave} = average raw fitness of population

The value of a is calculated based on the scale factor and the non-negativity requirement. This value may be positive or negative, depending on which direction the fitness line is pivoted.

2.5 Selection

Selection is the process of determining the number of offspring each individual will receive based on the individual's fitness. The selection process is divided into two parts: 1) determining the *expected* number of offspring of each individual; and 2) converting the expected number into a discrete number (sampling) [BAKE87].

During selection, the selected individuals are copied into an intermediate *mating pool* [GOLD85b]. With the proper sampling algorithm, individuals with high expected values receive more representatives in the mating pool than individuals with low expected values. Pairs of individuals

are later chosen from the mating pool to undergo crossover and mutation, yielding new phenotypes.

2.5.1 *Expected Offspring*

If the population size is kept constant in the genetic algorithm, the expected number of offspring of an individual may be computed from the individual's scaled fitness as follows:

(2.4)

$$e_i = N\frac{f_i}{\sum f} = \frac{f_i}{f_{ave}}$$

where:

e_i = expected number of offspring of individual i
f_i = scaled fitness value of individual i
f_{ave} = average scaled fitness of population
N = population size

This reflects Holland's concept of reproduction in proportion to measured performance [HOLL75]. The sum of the expected number of offspring for all individuals is equal to *N*. Therefore, the population size remains constant from one generation to the next.

Note that an individual's expected number of offspring is a real number. Since it is impossible to realize a non-discrete number of offspring, a type of sampling error results. This sampling error induces *genetic drift*, a suspected cause of premature convergence. Genetic drift will be discussed in greater detail later in this paper.

2.5.2 *Stochastic Universal Sampling*

The goal of the sampling algorithm is to minimize sampling error in order to avoid the problems associated with genetic drift. A thorough analysis of existing sampling algorithms led to Baker's development of Stochastic Universal Sampling (SUS). For more detail on SUS and a comparison of various sampling algorithms, the reader is referred to [BAKE87].

One of the parameters Baker used to study selection algorithms is *spread*. This is the range of possible values for the number of offspring an individual receives. If an individual's expected value is 1.6, then *minimum spread* is achieved if the selection algorithm guarantees that the individual will receive either 1 or 2 representatives in the mating pool.

SUS may be thought of as a type of "roulette wheel" selection. A roulette wheel is constructed such that every individual in the population is given a slot. The size of an individual's slot is proportional to the individual's expected number of offspring. The circumference of the wheel is equal to the sum of the expected values, which is N, the population size.

Typical selection algorithms use a single "pointer" to the wheel. The wheel is spun N times, and each time the chosen individual is placed in the mating pool. This clearly does not guarantee minimum spread; in fact, it is possible (although improbable) for the same individual to be selected N times.

In SUS, N evenly-spaced pointers are placed around the wheel, and a single spin of the wheel selects all of the individuals for the mating pool. An individual with a high expected value has a large slot in the wheel that may span several pointers. The number of copies of the individual that are placed in the mating pool is equal to the number of pointers spanned. SUS guarantees that minimum spread is achieved.

2.6 Crossover

Crossover is a means of forming new phenotypes from existing ones. As part of the mating process, segments of the parents' chromosomes are exchanged to yield a new individual with some of the characteristics of both parents. Crossover techniques in genetic algorithms attempt to model this natural process.

After SUS, pairs of individuals are selected from the mating pool to participate in crossover. Every individual in the mating pool is involved in crossover exactly once. Note that if SUS is used, the mating pool must be shuffled before crossover since all representatives of a given individual will appear in consecutive array locations. Shuffling the mating pool avoids copies of the same individual from undergoing crossover with one another, producing offspring identical to the parents.

In genetic algorithms where the representation is not order-based (e.g. bit strings), crossover is the exchange of substrings between the

parents. This always yields valid phenotypes, since every possible combination of alleles is valid. With order-based representations, however, care must be taken to prevent the formation of invalid phenotypes. One possible solution is to perform the usual exchange of substrings and "repair" the result to yield a valid chromosome [DAVI91].

Several crossover operators have been developed for the TSP that always generate valid offspring. A few of these have been extended to work with the two-dimensional representation chosen for this project. For a detailed description of the operators used for the TSP, the reader is referred to [GOLD85a], [OLIV87], and [GOLD89a].

2.6.1 Region Selection

The first step of typical crossover operators is to determine the portions of the chromosomes that will be exchanged between the two parents. For one-dimensional representations, this is usually done by randomly selecting start and end indices for the crossover region. In this case, the crossover region is simply a random substring.

For two-dimensional chromosomes, the problem is slightly more complex. It is desirable to select a contiguous subregion of cells as the crossover region. In this way, highly-fit *building blocks* composed of small groups of cells tend to be preserved. The concept of building blocks will be discussed in greater detail later in this paper.

A simple way of defining a contiguous crossover region is to randomly choose a rectangular subregion of cells. This method is illustrated Figure 2-2, where the shaded area indicates the crossover region.

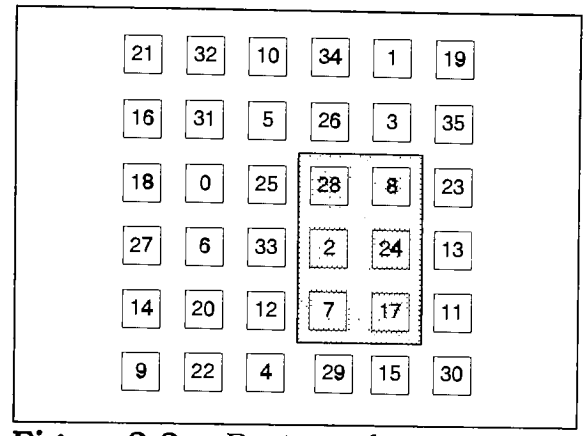


Figure 2-2 Rectangular crossover region

Another method for choosing a contiguous subregion is based on the flood fill algorithm used in computer graphics. Given the total number of cells (n) to be selected, the *flood select* procedure operates as follows:

```
randomly choose a locus and insert it into a queue
while (n > 0)
{
        remove a locus from the queue
        mark the locus as selected and decrement n
        for each neighbor of the locus
        {
                if the neighbor is not already selected or enqueued
                        enqueue the neighbor
        }
}
```

Since every selected cell is guaranteed to have a neighbor that is also selected (assuming $n > 1$), the subregion is contiguous. The regions will tend to be diamond-shaped if only horizontal and vertical neighbors (4-neighbors) are considered or square if diagonal neighbors are considered

as well (8-neighbors). The flood select starts with a single cell and spirals outward until the proper number of cells has been selected.

If the enqueuing procedure is altered so that loci are inserted into random positions in the queue, the shape of the selected region will tend to be less regular. This modification essentially alters the "outward spiraling" growth pattern of the region, causing the region to grow from random positions along its perimeter.

The flood select procedure is clearly more computationally complex than rectangular region selection. However, the shape of the selected regions is irregular, adding diversity to the crossover operation. One of the parameters of *CPGA* determines whether rectangular region selection or flood selection is used for crossover region selection.

2.6.2 Partially Matched Crossover

Partially matched crossover (PMX), also referred to as partially mapped crossover, was developed in [GOLD85a] for use with the Traveling Salesman Problem. This operator is easily extended to the two-dimensional representation used in *CPGA*.

In applying PMX to string representations (e.g. the TSP), two crossover sites are first chosen at random. The region between these sites is termed the *matching section*. Given two parent chromosomes A and B, the goal is to perform the necessary swaps within parent A so that its

matching section matches that of parent B. After the same procedure is applied to parent B, the result is two new offspring A' and B' containing the matching sections of the parents.

The results of PMX applied to two parent strings is shown in Figure 2-3. Within string A, it is necessary to swap the positions of the 5 and the 4, the 6 and the 9, and the 8 and the 3 to yield string A'. Within string B, the positions of the 4 and the 5, the 9 and the 6, and the 3 and the 8 are swapped to produce B'. String A' contains the matching section of string B, and B' contains the matching section of A.

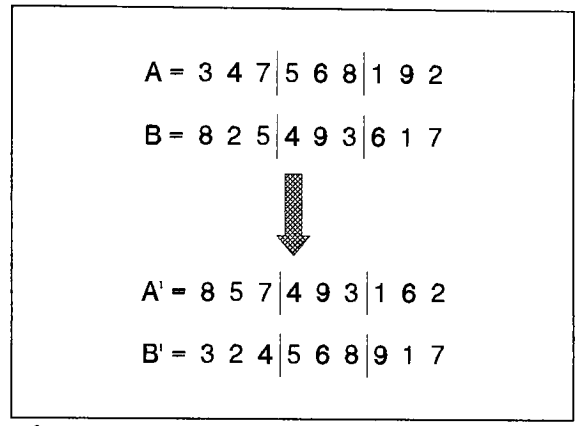


Figure 2-3 PMX applied to strings

Extending PMX to two-dimensional chromosomes is straightforward, as shown in Figure 2-4. The shaded areas in the figure denote the *matching region*. Cells within the matching region of A are swapped with cells outside the matching region to produce A', which contains the matching region of B. The appropriate swaps are made within B to yield B', which contains the matching region of A. The shaded cells outside of the matching region in A' and B' were swapped with cells inside of the matching region during PMX, corrupting the area of the chromosomes outside of the matching region.

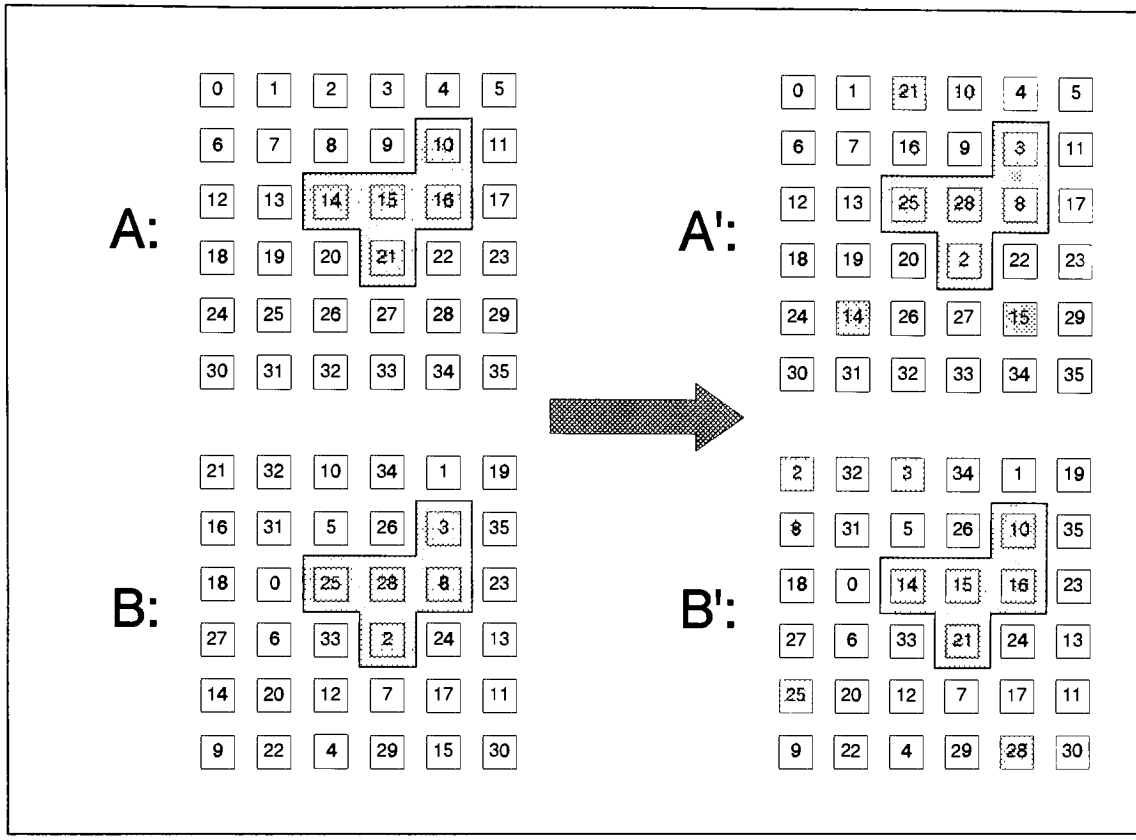


Figure 2-4 Partially matched crossover in *CPGA*

Note that in both the one- and two-dimensional cases, it was necessary to corrupt portions of each parent chromosome outside of the matching section in order to produce valid offspring. This is an undesirable side effect, but it is unavoidable because of the nature of the permutation representation.

2.6.3 *Order Crossover*

Order crossover (OX) was developed in an attempt to preserve the relative positions of elements affected by the crossover operation. Its operation is shown in Figure 2-5 and proceeds as follows.

To create offspring A', all of the elements in A outside of the matching section that appear within the matching section of B are replaced with *holes.* This is shown in Step 1, where "H" is used to denote holes. In step 2, the holes are shifted to the matching section using a "sliding" motion; that is, holes are moved to the matching section one position at a time by exchanging them with their immediate neighbors. In step 3, the matching section of B is copied into the holes in the matching section of A'.

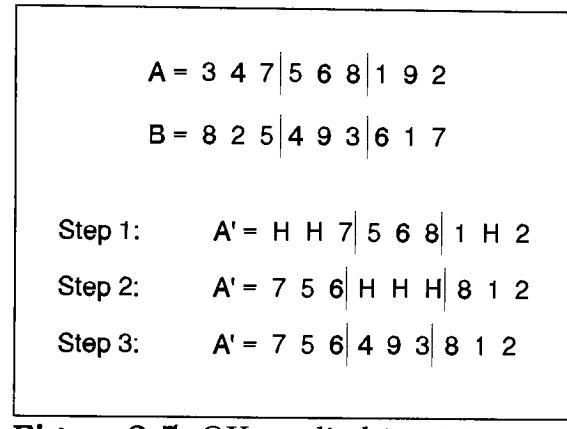


Figure 2-5 OX applied to strings

By using a sliding motion to shift the holes, the relative positions of elements outside of the matching section are maintained. This is important for preserving small building blocks in the chromosomes.

The extension of order crossover to two dimensions is somewhat more complex than for PMX. The first step of order crossover is shown for offspring A' in Figure 2-6. Elements of parent A outside of the matching

region that appear within the matching region of B have been replaced with holes.

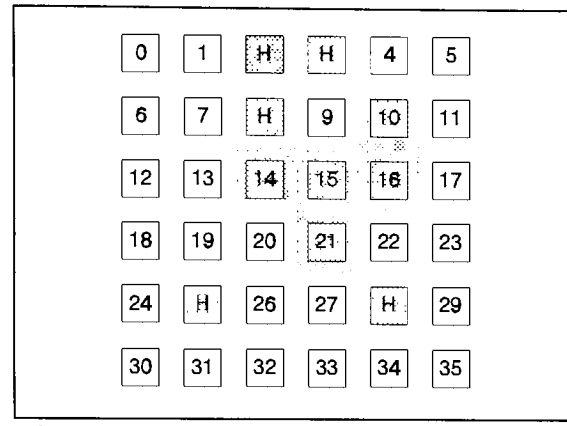


Figure 2-6 Step 1 of order crossover in *CPGA*

The second step of order crossover involves shifting the holes to the matching region. One way to do this is to map every hole outside of the matching region to a non-hole destination inside of the matching region. The holes are then shifted along a path leading to their destinations. This problem is analogous to the "sliding block" puzzle.

One way to minimize the disruption to the chromosome outside of the matching section is to attempt to map each hole to the nearest non-hole in the matching region. An example of a relatively poor mapping is shown in Figure 2-7, while a better mapping is shown in Figure 2-8.

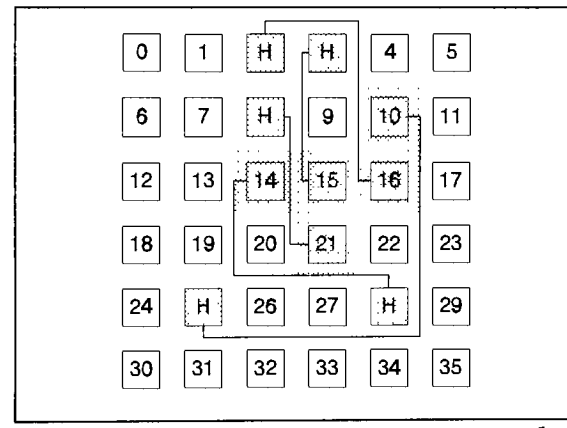


Figure 2-7 Poor mapping in order crossover

The problem of finding an optimal mapping is itself worthy of a genetic algorithm. However, "good" mappings may be found by using a greedy

heuristic as follows. First, two lists I and O are created containing the non-holes inside the matching region and the holes outside of the matching region, respectively. Next, an element o is removed from list O and the element i in list I that is closest to o is found. (Once again, Manhattan distances are used). Element i is mapped to o, and i is removed from list I. This procedure is repeated until list O is empty.

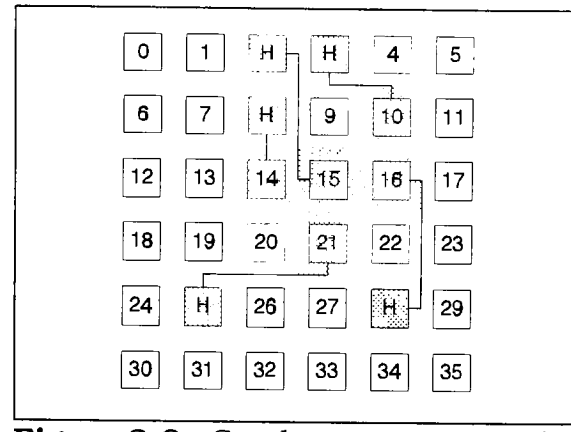


Figure 2-8 Good mapping in order crossover

This heuristic is fairly expensive given the frequency of crossover in a genetic algorithm. Furthermore, it is not clear how much better the results of order crossover with a good mapping are than the results with a poor mapping. One of the parameters of *CPGA* specifies whether to use this heuristic or simply use the random mapping resulting from the construction of lists I and O.

Once a mapping has been established, the holes must be shifted from their positions outside of the matching region to their destinations inside of the matching region. This is accomplished by swapping holes with their horizontal and vertical neighbors along a path leading to the destination. A minimal path of this type results from performing all necessary horizontal movement first, then all necessary vertical movement (or vice versa). One

such path is illustrated in Figure 2-9. The result of shifting the hole along this path is shown in Figure 2-10. Once all of the holes have been moved to their destinations in this manner, the matching region is copied from the mate.

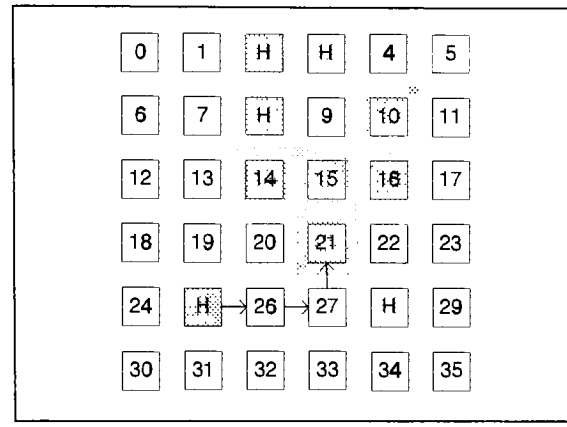


Figure 2-9 Minimal path for hole movement in order crossover

A problem arises during hole movement when a hole must pass through other holes already in the matching region in order to reach its destination. If the simple neighbor swapping scheme is used, then a hole in the matching region may end up outside of the matching region after the operation is complete. This undesirable side effect may be prevented by using two pointers (*c* and *n*) into the chromosome as follows:

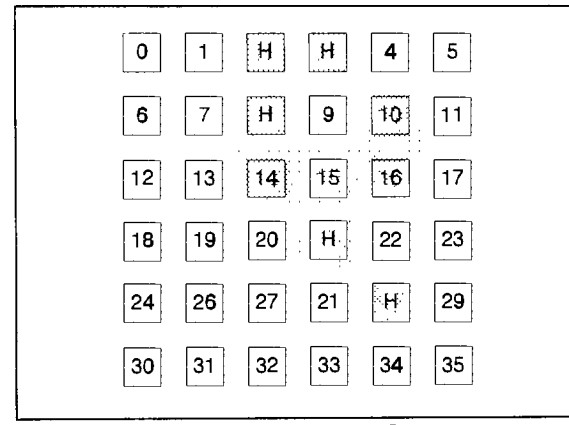


Figure 2-10 Results of hole movement in order crossover

```
c = starting locus of hole H
while c ≠ destination locus of H
{
        n = next locus along the path to the destination locus of H
        if the cell occupying n is not a hole
        {
                swap the cells at c and n
                c = n
        }
}
```

The condition placed on swapping causes a hole to skip over other holes that are on the path to its destination locus. Therefore, once a hole is in the matching region, it will not be affected by the movement of other holes.

2.7 Mutation

Mutation occurs after crossover in a genetic algorithm and causes allele values to change randomly with a very low probability. In genetic algorithms using bit string representations, this operation is performed by simply complementing bits in the new offspring with a probability on the order of 0.1% per bit. Mutation prevents reproduction and crossover from losing potentially useful genetic material [GOLD89a]. Without mutation, a particular locus could *converge*; that is, every member of the population could have the same allele value at that locus. This is an irrecoverable loss, since subsequent applications of reproduction and crossover cannot alter the value at the converged locus.

2.7.1 Inversion in the TSP

In order-based representations such as that used for the TSP, randomly altering allele values is not guaranteed to produce valid chromosomes. A commonly-used mutation operator for the TSP that always yields valid chromosomes is *inversion*. Inversion is performed by

randomly selecting two points in the string and reversing the enclosed substring. This operation is illustrated in Figure 2-11.

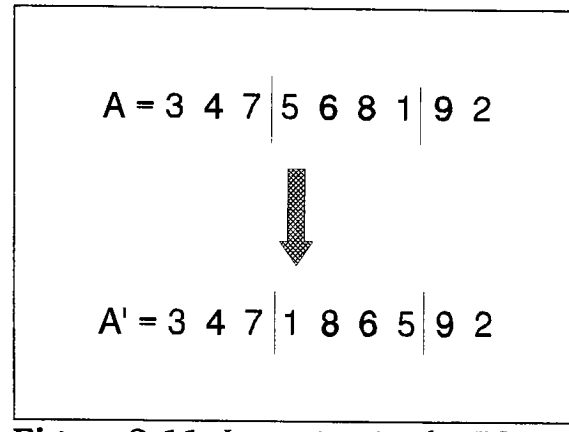


Figure 2-11 Inversion in the TSP

The mutation resulting as a side effect of the crossover operation can prevent convergence at a particular locus. However, it is possible for the population to prematurely converge on a particular structure representing a local maxima in the search space. In this case, the alleles at all loci have converged simultaneously. If this occurs, reproduction and crossover alone cannot escape the local maxima. Inversion provides a means for this escape by reintroducing new structures into the population.

Another role of inversion is to "stir up" the genetic material available for crossover [WHIT87]. This allows coadapted alleles to cluster within the chromosome, resulting in the formation of new building blocks.

2.7.2 Inversion in CPGA

The extension of inversion to two-dimensional chromosomes is straightforward. Inversion in two dimensions may be accomplished by randomly selecting a rectangular subregion and reflecting it about either the x or y axis. An example of this operation is shown in Figure 2-12. Note

that inversion does not affect genes outside of the mutation region.

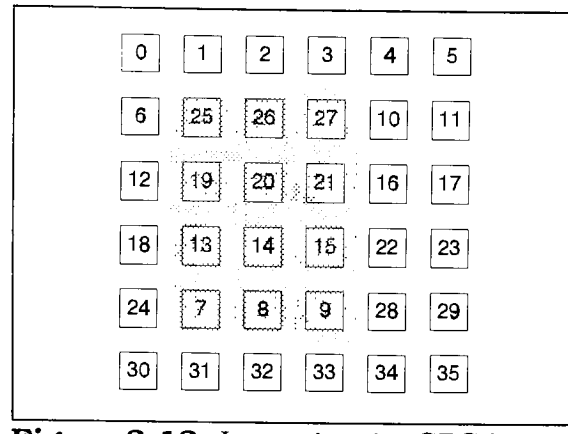


Figure 2-12 Inversion in *CPGA*

2.8 Performance

Evaluating the performance of a genetic algorithm involves measuring the quality of the solutions it produces during its search. This measurement is useful for comparing the effectiveness of different operators (as with PMX and OX) or finding optimal values for various parameters. The latter operation is important for "tuning" the algorithm to achieve the best possible performance. Methods for determining genetic search performance are presented in this section, as well as a discussion of premature convergence.

2.8.1 Performance Measurement

Because genetic algorithms are randomized searches, it is necessary to perform a sufficiently large number of iterations (e.g. 30) of the algorithm under the same conditions and observe the average performance. Three measures are commonly used to evaluate genetic algorithm performance: *online performance*, *offline performance*, and *best individual* [BAKE85].

Online performance is simply the average evaluation of all individuals generated during the genetic search. This measure is appropriate when

emphasis is placed on the expense of generating and evaluating solutions [SCHA89]. Genetic algorithms that explore many poor structures during their search will have poor online performance.

Offline performance at a particular generation is determined by averaging the best individuals occurring in that generation over all iterations of the algorithm. This is typically plotted as a function of the total number of individuals generated, as in [DAVI91]. Offline performance is useful when considering the convergence properties of the genetic algorithm. The exploration of many poor structures during genetic search will not negatively impact the offline performance of the genetic algorithm.

Best individual is simply the best individual generated in all iterations of the algorithm. Baker suggests this measure for comparing genetic algorithms used for function optimization [BAKE85]. In this project, offline performance and best individual together are used to study the properties of the genetic search.

2.8.2 Premature Convergence and Genetic Drift

Near the end of a genetic search, the population begins to *converge* on a particular structure; that is, the members of the population obtain a high degree of uniformity at all loci. In genetic algorithms employing *niche* and *speciation* techniques, the population converges on several stable sub-

populations. Such techniques are useful for obtaining information on more than one peak in a multimodal function optimization problem [GOLD87b].

Premature convergence occurs when the population converges too rapidly on a suboptimal solution. Goldberg cites two reasons for this phenomenon: a problem is *GA-hard*, or the genetic algorithm suffers from stochastic sampling errors [GOLD87a]. It has been shown that it is difficult to construct intentionally misleading (GA-hard) problems [GOLD89a]. Furthermore, the success of genetic algorithms across a wide range of problem domains reinforces this notion.

Sampling errors are the unavoidable consequence of finite population sizes. As noted earlier, it is impossible with a finite population to realize an individual's expected number of offspring with arbitrary precision. This leads to a phenomenon known as *genetic drift*, where sampling errors accumulate, causing the population to converge on a structure that has no particular selective advantage. Genetic drift may be reduced by increasing the mutation rate, which restores lost allele values and slows convergence. However, excessive mutation also has the undesirable effect of disrupting high-performance building blocks [BOOK87].

Increasing the population size reduces undersampling and genetic drift, but at the expense of a lower convergence rate and greater execution time. The population size chosen for *CPGA* is 250, which is somewhat larger than that of "typical" genetic algorithms (e.g. 50).

Sampling errors are also reduced by using an accurate, consistent sampling algorithm. *CPGA* employs stochastic universal sampling (SUS), which has been shown to have superior sampling characteristics. It is suggested that SUS is an optimal sampling algorithm.

More advanced schemes for reducing genetic drift are presented in [BAKE85] and [BOOK87]. A thorough investigation of genetic drift is presented in [GOLD87a].

2.9 Schemata Theory

When considering the effectiveness of genetic search, the question arises: What information present in the population is being exploited to guide the search toward better solutions? Genetic algorithms do not attempt to improve the quality of the solutions by searching for particular allele values independently of one another. In *CPGA*, for example, the effect an allele has on the overall fitness of the phenotype is very much dependent on the positions of other alleles. This phenomenon is analogous to *epistasis* in biological systems. Because of epistatic effects, adaptation becomes a search for coadapted sets of alleles - clusters of alleles ("building blocks") that together enhance the overall fitness of the phenotype [HOLL75]. It is this observation that led Holland to formalize the notion of coadapted sets of alleles, or *schemata*.

A complete discussion of schemata theory in conventional genetic algorithms is provided in [GOLD89a]. The goal of this section is to extend this theory to two-dimensional chromosomes.

2.9.1 Absolute Order Schemata

Schemata may be thought of as matching templates for phenotypes. Using the metasymbol "!" for "don't care" positions, a phenotype and three absolute order schemata (o_a-schemata) that match it are shown in Figure 2-13. The positions in the schema with specified allele values are referred to as the *defined* positions. The *order* of a schema is the number of defined positions.

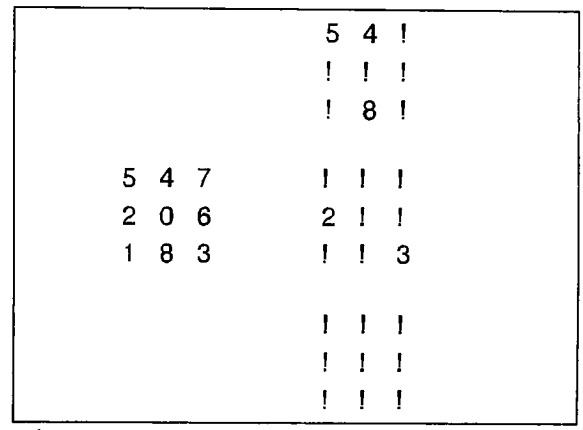


Figure 2-13 Absolute order schemata

The total number of o_a-schemata for a chromosome size of n genes is given by:

(2.5)

$$N_{aos} = \sum_{d=0}^{n} \binom{n}{d} \frac{n!}{(n - d)!}$$

where d is the number of defined positions.

Every phenotype is a representative of 2^n o_a-schemata, since every locus in a matching schema may contain either a "!" or the allele value at that locus in the chromosome. The number of schemata N_p represented in a population of size *P* is given by:

(2.6)

$$2^n \leq N_p < P2^n$$

Therefore, in searching with a population of *N* structures, *CPGA* is in effect searching a much larger schema space. This property of genetic algorithms is referred to as *implicit parallelism.*

The *defining perimeter* of a schema is the smallest rectangular box (*defining box*) that encloses all of the defined positions. The *defining area* is the number of loci enclosed by the defining perimeter. Figure 2-14 shows an o_a-schema of order 3 with a defining area of 6. The defining perimeter is shown as a dashed line.

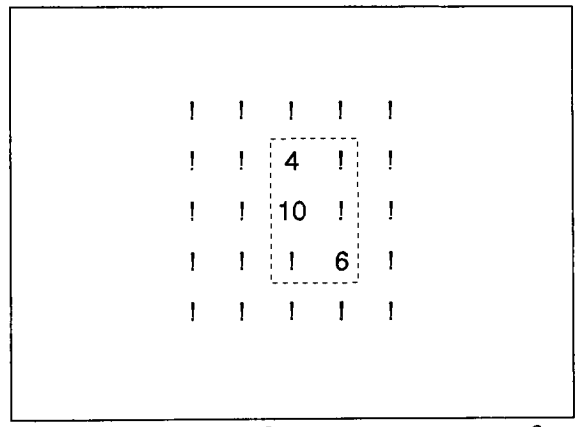


Figure 2-14 Defining perimeter of o_a schema

2.9.2 *Relative Order Schemata*

We now remove the restriction that the defined positions of a schema must occupy certain absolute positions within the chromosome. For a particular relative o-schema (o_r-schema), the position of the defining box is irrelevant; only the relative positions of the defined positions is important. Furthermore, the defining box is allowed to "wrap" past the boundaries of the chromosome. The top and bottom edges of the chromosome are considered to be adjacent, as are the left and right edges. Therefore, there are n possible positions for the defining box of an o_r-schema. Figure 2-15 shows 3 possible positions for the defining box of a particular o_r-schema.

The n possible positions of the defining box for a given o_r-schema represent n distinct o_a-schemata. Therefore, the total number of o_r-schemata is given by (2.7).

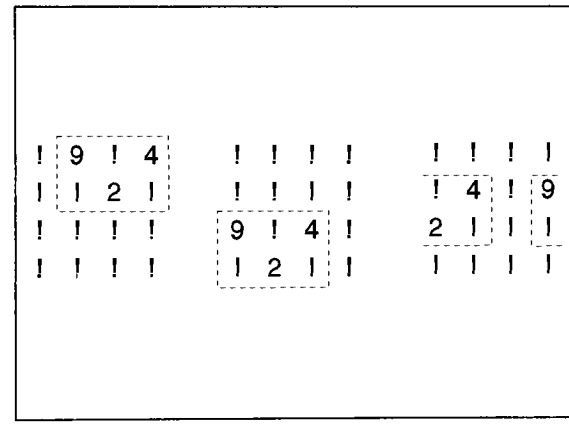


Figure 2-15 An o_r-schema

(2.7)

$$N_{ros} = \frac{1}{n} N_{aos} = \frac{1}{n} \sum_{d=0}^{n} \binom{n}{d} \frac{n!}{(n - d)!}$$

As with o_a-schemata, every phenotype is a representative of 2^n o_r-schemata. The total number of o_r-schemata represented in a population of

size P is given in (2.6). Because there are fewer o_r-schemata than o_a-schemata, a population represents a greater proportion of the possible o_r-schemata [OLIV87].

2.9.3 Crossover and Relative O-Schemata

We now consider the effects of crossover on o_r-schemata. The following analysis is based on work presented in [GOLD89a] and [OLIV87] pertaining to the TSP.

The probability that an o_r-schema will survive crossover intact is:

(2.8)

$$P(S_x) = P(S_x \mid W)P(W) + P(S_x \mid O)P(O) + P(S_x \mid C)P(C)$$

where the events are defined as follows:

S_x occurs when the schema survives crossover; that is, when its defining box remains intact

W occurs when the defining box of the schema is entirely within the crossover region

O occurs when the defining box is entirely outside of the crossover region

C occurs when the defining box is cut by the crossover region

Note that events W, O, and C are mutually exclusive, and:

(2.9)

$$P(W) + P(O) + P(C) = 1$$

Equation (2.8) may be simplified by observing that $P(S_x/C)$ is negligible. A schema could survive being cut by crossover, for example, if the matching regions of the two parents are identical, which is highly improbable. Also, note that $P(S_x/W) = 1$, since a schema survives if its

defining area is transferred intact from one chromosome to another. These two simplifications reduce (2.8) to the following:

(2.10)

$$P(S_x) \approx P(W) + P(S_x \mid O)P(O)$$

The probability that the defining box of the schema is completely within the crossover region, *P(W)*, is equal to the number of ways the defining box could fit completely within the crossover region divided by the number of possible positions for the defining box within the chromosome. The crossover region is assumed to be rectangular, with height x_h and width x_w. For a defining box with height d_h and width d_w, *P(W)* is given in (2.11).

(2.11)

$$P(W) = \begin{cases} \dfrac{(x_h - d_h + 1)(x_w - d_w + 1)}{n} & \text{if } d_h \le x_h \text{ and } d_w \le x_w \\ 0 & \text{otherwise} \end{cases}$$

For PMX, we define n_c as the number of cells outside of the crossover region that are "corrupted" by the swaps performed during crossover. This value varies between 0 (for parents with the same subset of alleles in their crossover regions) and $x_h x_w$ (for parents with no common allele values in the crossover region). Therefore, the probability that a schema survives PMX given that it is completely outside of the crossover region is:

(2.12)

$$P(S_x \mid O) = \begin{cases} 1 - \left(\dfrac{n_c d_h d_w}{n - x_h x_w} \right) & \text{if } (n_c d_h d_w) < (n - x_h x_w) \\ 0 & \text{otherwise} \end{cases}$$

To compute *P(O)*, the probability that the defining box is completely outside of the crossover region, we partition the region of the chromosome outside of the crossover region into 2 overlapping regions I and II, as shown in Figure 2-16. Region III is the intersection of regions I and II. For the purpose of illustration, the crossover region is shown in the upper left-hand corner of the chromosome. This in no way restricts the following analysis; the position of the crossover region is irrelevant since schemata are permitted to wrap past the boundaries of the chromosome.

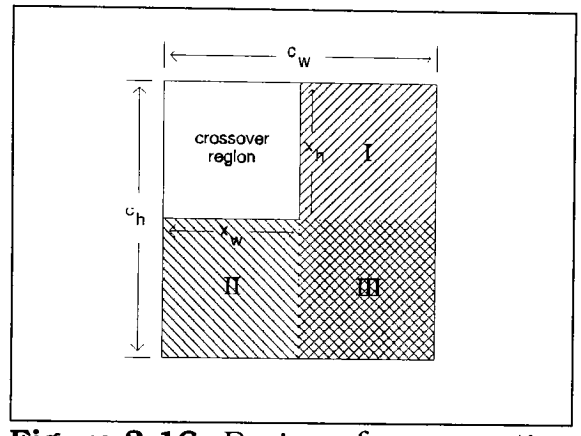


Figure 2-16 Regions for computing P(O)

To find *P(O)*, we simply add the probabilities of the defining box being completely within each of regions I and II and subtract from that result the probability that it is completely within region III. These probabilities are found in a manner analogous to (2.11). Also, as in (2.11), each of the terms goes to 0 when one of the dimensions of the defining box exceeds the corresponding dimension of the region. *P(O)*, then, is given by:

(2.13)

$$P(O) = \frac{1}{n}[(c_w - x_w - d_w + 1)(c_h - d_h + 1) + (c_w - d_w + 1)(c_h - x_h - d_h + 1) - (c_w - x_w - d_w + 1)(c_h - x_h - d_h + 1)]$$

which reduces to:

(2.14)

$$P(O) = \frac{(c_h - d_h + 1)(c_w - d_w + 1) - x_h x_w}{n}$$

Substituting the results from (2.11), (2.12), and (2.14) into (2.10), we obtain:

(2.15)

$$P(S_x) \approx \frac{(x_h - d_h + 1)(x_w - d_w + 1)}{n} + \left(1 - \frac{n_c d_h d_w}{n - x_h x_w}\right)\left[\frac{(c_h - d_h + 1)(c_w - d_w + 1) - x_h x_w}{n}\right]$$

Equation (2.15) shows that schemata with smaller defining areas have a higher probability of surviving crossover. Note that (2.12) was derived for the PMX operator. Qualitatively, we might speculate that order crossover is more destructive of schemata outside of the crossover region since it involves shifting cells not directly involved in the crossover operation. However, OX tends to preserve the relative positions of elements (i.e. A is "to the left of" B), giving rise to schemata that are "similar" to the destroyed schemata. In *CPGA*, these *similar* schemata will probably be associated with fitness values that are close to those of the original schemata. Shifting cells by a relatively small amount within the chromo-

some is not likely to seriously impact the total net length of the placement represented by the chromosome.

2.9.4 Mutation and Relative O-Schemata

A schema is destroyed by inversion if its defining box is partially or completely within the rectangular mutation region, and it survives if its defining box is completely outside of the mutation region. Therefore, the probability that a schema survives inversion is simply the probability that its defining box is completely outside of the mutation region. The derivation follows from that of (2.14):

(2.16)

$$P(S_m) = \frac{(c_h - d_h + 1)(c_w - d_w + 1) - m_h m_w}{n}$$

where m_h and m_w are the dimensions of the mutation region. As with crossover, schemata with smaller defining areas are more likely to survive mutation.

Note that if the defining box is completely within the mutation region, then inversion will create a mirror image of the original schema. This is comparable to the creation of similar schemata by order crossover.

2.9.5 The Fundamental Theorem and CPGA

As noted earlier, the processing of phenotypes by genetic operators causes implicit processing of the schemata represented. An analysis of the

growth rates of schemata is presented in [GOLD89a] and is shown here for *CPGA*.

Goldberg defines *m(H,t)* as the number of representatives of a schema *H* at time (generation) *t*; that is, the number of individuals in the population whose chromosomes match schema *H*. Furthermore, *f(H)* is defined to be the average fitness of all individuals representing (matched by) schema *H*. Applying (2.4) to this fitness value, we obtain the expected number of offspring e_r of each representative of *H*:

(2.17)

$$e_r = \frac{Nf(H)}{\sum f} = \frac{f(H)}{f_{ave}}$$

where f_{ave} is the average fitness of the population and *N* is the population size. Therefore,

(2.18)

$$m(H,t+1) = m(H,T)\left(\frac{f(H)}{f_{ave}}\right)$$

Equation (2.18) shows that a schema grows at a rate equal to the ratio of the average fitness of its representatives to the average fitness of the population.

If a schema *H* remains above average by an amount cf_{ave}, then:

(2.19)

$$m(H,t+1) = m(H,t)\left(\frac{f_{ave} + cf_{ave}}{f_{ave}}\right) = (1 + c)m(H,t)$$

When c remains constant, we obtain the following equation starting from t=0:

(2.20)

$$m(H,t) = m(H,0)(1 + c)^t$$

As Goldberg points out, this is the equation for compound interest. Reproduction, then, results in an exponentially increasing number of representatives for above-average schemata, and an exponentially decreasing number of representatives for below-average ($c < 0$) schemata.

If we take into account the probability that the schema survives crossover and mutation, then (2.18) becomes:

(2.21)

$$m(H,t+1) = m(H,T)\left(\frac{f(H)}{f_{ave}}\right)P(S_x)_H P(S_m)_H$$

where $P(S_x)_H$ is the probability that H survives crossover and $P(S_m)_H$ is the probability that H survives mutation. As noted earlier, schemata with smaller defining areas have a higher probability of surviving both crossover and mutation. Therefore, above-average schemata with small defining areas will receive exponentially increasing numbers of representatives over time (generations). This conclusion is known as the Fundamental Theorem of genetic algorithms.

Chapter 3:
Implementation

The Cell Placement Genetic Algorithm is written in C++, an object-oriented extension of the C programming language. The software was developed on an MS-DOS PC using Borland C++ 2.0. This integrated package includes a compiler, assembler, debugger, and profiler.

CPGA has been run on several PC's with processors ranging from a 20 MHz 80286 to a 33 MHz 80486. The code is written in a highly portable manner in order to facilitate compilation and execution on Unix workstations. There are only a few minor MS-DOS dependencies, and they are documented in the source code.

This chapter describes the C++ implementation of *CPGA*. It also provides details concerning the execution of the program, such as command-line arguments and file formats.

3.1 C++ Class Description

The C++ class hierarchy for *CPGA* is shown in Figure 3-1. (Note that a few supplementary classes are not shown.) The following sections describe each of the classes in detail. For more information, the reader is referred to the header (.H) files in the source listing.

3.1.1 The Object Class

The Object class is an abstract superclass from which all other classes are derived. It is illegal to create an instance of this class; it serves

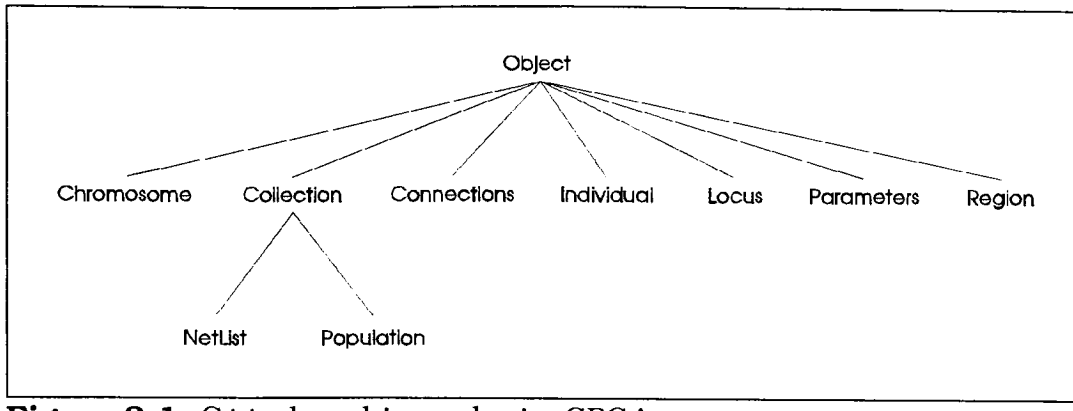


Figure 3-1 C++ class hierarchy in *CPGA*

only to define properties that are common to all other classes.

Dynamic allocation of objects in C++ is accomplished with the *new* operator. This operator returns a pointer to the storage allocated for an object upon success or a null pointer if there is insufficient storage. The Object class overloads this operator by defining a *new* operator that tests the pointer returned by the global *new* operator. If a null pointer is returned, an error message is displayed and the program terminates. This redefined operator provides a central place to check for memory allocation failures, since it is called every time an instance of a class derived from Object is created using the *new* operator.

The Object class also defines pure virtual functions that are to be implemented by derived classes. The function *className( )* returns a pointer to a string containing the receiver's class. Every class defines a static data member *name*, which holds one copy of the class name that is shared by all instances of the class.

For debugging purposes, the function *isValid( )* is defined. This function tests an object to ensure that it is a valid object of its particular class. The functions *input( )* and *output( )* are used to perform stream I/O with objects.

3.1.2 The Collection Class

The Collection class is an abstract class derived from Object. It serves as a base class for classes that are collections of objects, such as the NetList class.

The Collection class defines the data member *count*, containing the number of objects in a Collection. It also provides the function *numItems( )*, which returns the value of *count*, and the function *isEmpty( )*, which returns TRUE if *count* is equal to zero.

3.1.3 The Chromosome Class

The two-dimensional chromosomal representation used in *CPGA* is defined in the Chromosome class. A Chromosome contains a two-dimensional array of allele values (cell numbers) and the associated objective function value. Operators are defined for the assignment of Chromosomes and for accessing alleles within the chromosome given an integer index or a Locus.

The member function *xover( )* performs partially matched or order crossover on two Chromosomes. Depending on the crossover type specified in the arguments to the program, it calls either *pmx( )* or *ox( )* to perform this operation. The *mutate( )* function performs mutation on a Chromosome, and *objFn( )* computes the objective function of a Chromosome given a NetList.

3.1.4 The Locus Class

Instances of the Locus class are used to specify positions within a Chromosome. A Locus contains two data members, *rowNum* and *colNum*, which are used as indexes into the two-dimensional array of a Chromosome.

Several member functions are defined for assignment, addition, and comparison of loci. Also, functions are provided for determining the loci of neighboring positions within the Chromosome. These neighbor functions are used primarily by the flood selection operation.

3.1.5 The ChromosomeMap Class

The ChromosomeMap class is used to quickly find the Locus of a particular allele within a Chromosome. The constructor for this class takes a Chromosome as an argument and builds an array Locus values using

alleles as indexes. Given an allele value, the [ ] operator returns the locus of the allele within the corresponding Chromosome.

This class also defines an array of flags indicating which alleles have been marked as holes by the order crossover operator. As a hole is shifted through a Chromosome by this operator, the ChromosomeMap is updated to reflect changes in the locus of the hole, as well as the loci of non-holes along the path to the hole's destination.

3.1.6 The Region Class

The crossover operation causes the exchange of a randomly selected region between two parent chromosomes. Instances of the Region class are used to specify the portion to be exchanged. A Region contains a two-dimensional array of flags the same size as a Chromosome. The array entry at a particular location is TRUE if the locus is part of the region or FALSE if it is not. The Region class provides access functions that return the value of a flag at a particular location in the array given a Locus or an integer index. The function *size( )* returns the number of loci that are part of the region.

The constructor for the Region class uses the region type specified in the parameters to the program. It calls either *rectRegion( )* or *floodRegion( )* to generate a random rectangular or flood region, respectively.

For the purpose of comparing flood and rectangular regions, the distribution of the number of cells selected as part of the region is the same for both region types. The Region constructor generates two random values, *rw* and *rh*. If rectangular regions are to be used, these two values are passed to *rectRegion( )* as the region width and height, respectively. If flood regions are used, the product of these two values is passed to *floodRegion( )* as the number of loci to select as part of the region.

3.1.7 The ListElement and LinkedList Classes

The ListElement class is used in conjunction with the LinkedList class to implement a doubly linked list of Objects. A ListElement contains two pointers, *prev* and *next*, which point to the previous and next elements in the list, respectively. The data member *item* is a pointer to an Object.

The LinkedList class contains data members *first* and *last*, which point to the first and last ListElements in the list, respectively. The *cur* data member is used by an iterator function to point to the current ListElement while traversing the list.

Member functions are provided for accessing, deleting, and inserting ListElements. These operations may occur relative to the *first*, *last*, or *cur* pointer. The function *insertRandomly( )* inserts an element into a random position in the list.

3.1.8 The LocusQueue Class

The LocusQueue class uses a LinkedList to implement a queue of loci. It also maintains an array of flags indicating which loci are enqueued. This class is used by the flood select routine as described in Chapter 2.

The member function *insert( )* inserts a locus into the rear of the queue, and *remove( )* removes a locus from the front of the queue. The *insertRandomly( )* function inserts a locus into a random position in the queue. The function *inQueue( )* is provided to test whether or not a particular locus is in the queue.

3.1.9 The Connections Class

The Connections class is used to specify the connections between cells in the array. A two-dimensional array of connection weights (*conArray*) is defined that uses allele values (cell numbers) as indexes. The entry *conArray[i][j]* contains the weight of the connection between cell *i* and cell *j*. The access function is written such that *conArray[i][j]* returns the same value as *conArray[j][i]*.

The member function *optLen( )* returns the sum of the weights in the array. A chromosome whose objective function value is equal to *optLen( )* represents an optimal placement, since every net must have a length equal to 1 (the smallest possible net length.) It is possible that for a particular

placement problem, there may not exist a solution whose total net length is equal to *optLen( )*.

The Connections array is read from a file specified on the command line of the program. Note that there is only one instance of this class during program execution.

3.1.10 The Net and NetList Classes

The first implementation of the objective function in *CPGA* tested for connections between every possible pair of cells. Upon finding a connection, it would compute the Manhattan distance between the cells, multiply this distance by the weight of the connection, and add the result to the objective function value for the chromosome. The use of a profiler revealed the inefficiency of this $O(n^2)$ search for connections: over 50% of the overall execution time of the program was spent evaluating chromosomes in this manner.

A much more efficient method was developed that searches the Connections array once, building a linked list of the connections as they are encountered. This linked list is then passed to the objective function every time a chromosome is evaluated. Using the linked list rather than the Connections array significantly reduced the time spent computing the objective function, and the net result was a doubling of the execution speed of the program.

The Net and NetList classes are used to implement the linked list of connections. A Net contains the cell numbers (allele values) for the pair of cells connected by the net, as well as the weight of the net. A NetList uses a LinkedList to maintain a list of Nets. As with the Connections class, there is only one instance of a NetList during the execution of the program.

3.1.11 The Individual Class

The Individual class in *CPGA* encapsulates all the information related to an individual in the population. An Individual contains a chromosome, its fitness value *(fitness)*, and its expected number of offspring *(expOff)*. The static data member *created* is used to track the total number of individuals created. It is incremented every time a new individual is created randomly or through crossover or mutation.

Member functions are provided for returning the objective function value of the chromosome and the value of the variable *created*. The function *resetCounter( )* sets the value of *created* to 0. Crossover is performed with the *xover( )* member function, which takes as arguments a mate (another Individual) and the parameters to the algorithm (specifying the crossover type and probability of crossover.) The function *mutate( )* performs inversion on an Individual.

3.1.12 The Population Class

The Population class defines an array of Individuals, as well as statistics such as the average objective function value *(aveObjFnVal)*, the average fitness *(aveFitness)*, and the number of individuals in the population with the best objective function value *(numConverged)*. The data member *best* is a pointer to an individual with the best objective value.

The member function *evaluate( )* calculates raw fitness values for all individuals by calling *computeRawFitness( )*. It then scales the fitness values using the linear scaling algorithm described in Chapter 2.

The member function *generation( )* performs one generation of the genetic algorithm. It takes as parameters a pointer to the population where the new individuals will be placed *(newPop)* and the parameters of the genetic algorithm. The function first calls *sus( )* to perform stochastic universal sampling based on the scaled fitness values of the individuals. The sampled individuals are placed in *newPop*, and the *permute( )* function is called to randomly shuffle the new population. Next, crossover and mutation are performed on members of *newPop* according to the genetic algorithm parameters. Finally, *evaluate( )* is called to compute scaled fitness values for the new population.

3.1.13 The Parameters Class

The Parameters class is used to specify various parameters of the genetic algorithm. The constructor for this class reads the parameter values from the command line, and only one instance of the Parameters class exists during program execution.

Parameters on the command line take the form:

param=value

where *param* is one of the following:

r	run number
s	seed for the random number generator
i	number of iterations to perform
g	number of generations per iteration
rt	region type for crossover
nt	neighbor type for flood regions
qt	queue type for flood regions
xt	crossover type
px	probability of crossover
pm	probability of mutation
sf	scale factor
con	name of file containing connections
out	output file name

The *r* parameter allows the user to track the progress of *CPGA* when the program is applied to several problems in a batch mode. The value of this parameter is simply written to standard output during program execution. The default value for *r* is 1.

The *s* parameter specifies the seed value for the random number generator. If a value of 0 is given, a seed is generated using the system clock. The seed value used by a particular run of the program is written to the output file for that run. The user may reproduce the output of a particular run by specifying the seed value in the output file for the run. The default value for this parameter is 0.

The number of iterations of the genetic algorithm is given by the *i* parameter. The *g* parameter specifies the number of generations per iteration. The default value for *i* is 30, and the default value of *g* is 1000.

The *rt* parameter specifies the region type to use during crossover. Legal values for this parameter are "r" and "f", indicating rectangular and flood regions, respectively. The default region type is flood.

If flood regions are used, the *nt* parameter indicates the neighbor type used by the flood select algorithm. Possible values for this parameter are 4 and 8, with a default value of 4. This parameter has no effect if rectangular regions are used.

The *qt* parameter specifies the queue insertion method for the locus queue used by the flood select algorithm A value of "n" indicates normal insertion, and a value of "r" indicates random insertion. The default value for this parameter is "r".

The type of crossover operator to be used is specified with the *xt* parameter. The value "p" indicates PMX, and "o" indicates order crossover. The default is order crossover.

The parameters *px* and *pm* specify the probabilities of crossover and mutation, respectively. Legal values for these parameters are real numbers in the range [0, 1]. The default value for *px* is 0.7, and the default for *pm* is 0.06.

The scale factor used by the linear scaling algorithm is specified with the *sf* parameter. The scale factor is a real number that must be greater than 1. The default value for *sf* is 2.

The *con* parameter specifies the name of the file containing the connections array for the placement problem being solved. The extension ".con" is appended to the specified name before the file is opened.

The *out* parameter specifies the name of the output files that will be created by the program. For example, if "out=test" is specified, then the files "test.out" and "test.sta" will be created. The output file formats are described in detail in the next section.

When *CPGA* is executed, it first creates an instance of the Parameters class. The constructor for this class reads the parameters from the command line, assigning default values to the unspecified parameters. All parameters are public data members of the Parameters class. The

connections array is read from the specified file and stored as a public data member.

Since the instance of the Parameters class is declared as "const", functions may access but not modify the parameter values. This object is passed by reference to functions that require one or more of the parameter values.

3.2 Program Operation

CPGA is invoked with a command of the form:

C:\>**cpga con=p1 out=p1 i=15 xt=p pm=0.05**

In this example, the file "p1.con" must exist in the current directory. The output files "p1.out" and "p1.sta" will be created in the current directory.

The program displays the parameter values that will be used before starting the first iteration of the algorithm, as shown in Figure 3-2.

During execution, a line of text is written to the display every generation with statistics that show the progress of the program. For example, consider the last line in Figure 3-2. The first three numbers on the line that are separated by colons are the run number, the iteration number, and the generation number. The value of "opt" is the value returned by the function *Connections::optLen( )* as described earlier. The value of "run" is the objective function value of the best individual found so far in all iterations of the genetic algorithm, "it" is the best individual found

```
Cell Placement Genetic Algorithm
++++++++++++++++++++++++++++++

Parameters
  run number       = 1
  seed             = 13827
  iterations       = 15
  generations      = 1000
  region type      = flood
    neighbor type  = 4-neighbors
    queue type     = random
  crossover type   = PMX
  P(crossover)     = 0.7
  P(mutation)      = 0.05
  scale factor     = 2
  threshold        = 60
  input file       = p1.con
  output file      = p1.out
  stats file       = p1.sta


Iteration 1 **********************************
 1: 1:  0  opt: 60  run: 196  it: 196  gen: 196  num:  1  ave: 238.47
 1: 1:  1  opt: 60  run: 196  it: 196  gen: 196  num:  1  ave: 237.28
 1: 1:  2  opt: 60  run: 196  it: 196  gen: 196  num:  2  ave: 234.95
 1: 1:  3  opt: 60  run: 193  it: 193  gen: 193  num:  1  ave: 233.33
 1: 1:  4  opt: 60  run: 193  it: 193  gen: 193  num:  1  ave: 232.8
 1: 1:  5  opt: 60  run: 193  it: 193  gen: 199  num:  1  ave: 232.92
 1: 1:  6  opt: 60  run: 193  it: 193  gen: 199  num:  1  ave: 232.86
 1: 1:  7  opt: 60  run: 189  it: 189  gen: 189  num:  1  ave: 230.81
                              .
                              .
                              .
```

Figure 3-2 Sample output of *CPGA* (standard output)

in the current iteration, and "gen" is the best individual in the current generation. The number of individuals with the best objective function value in the current generation is shown as "num", while "ave" is the average objective function value of the population.

In Figure 3-2, note that the average objective function value of the population ("ave") decreases as the algorithm progresses. Near the end of an iteration, the value of "num" may increase, indicating convergence. If the value of "run" becomes equal to the value of "opt", then an optimal

solution has been found. However, as noted previously, it may not be possible to find a solution whose objective function value is equal to "opt".

3.2.1 Input File Format

The connections file is a text file that specifies the connections (nets) between cells in the array. A simple connections file is shown in Figure 3-3.

```
Connections
0 1 1
2 7 1
3 4 2
6 3 1
9 2 4
0 0 0
```

Figure 3-3 Sample connections file

The file begins with the keyword "Connections", followed by a list of the nets. Each net is represented with three numbers: the numbers of the two cells that are connected by the net followed by the weight of the net. For example, in Figure 3-3, cells 3 and 4 are connected with a net with weight 2. The last line, "0 0 0", indicates the end of the file.

3.2.2 Output File Format

When the last iteration of the genetic algorithm is complete, *CPGA* creates an output file (".out" extension) and a statistics file (".sta" extension). A sample output file is shown in Figure 3-4.

```
Average Best Value: 77

Individual
Chromosome
  5   4   3   2   1   0
 11  10   9   8   7   6
 17  16  15  14  13  12
 23  22  21  20  19  18
 29  28  27  26  25  24
 35  34  33  32  31  30
Objective Function Value: 60

Parameters
  run number       = 1
  seed             = 9011
  iterations       = 15
  generations      = 1000
  region type      = flood
    neighbor type  = 4-neighbors
    queue type     = random
  crossover type   = greedy OX
  P(crossover)     = 0.7
  P(mutation)      = 0.06
  scale factor     = 2
  threshold        = 60
  input file       = p1.con
  output file      = p1.out
  stats file       = p1.sta

Connections
1 0 1
2 1 1
3 2 1
  .
  .
  .
34 33 1
35 29 1
35 34 1
0 0 0
```

Figure 3-4 Sample output file

The first line of the output file shows the average objective function value of the best solutions found in all iterations of the genetic algorithm. This is a measure of the offline performance of the algorithm. It is followed by a textual representation of the best placement found in all iterations and its objective function value. Next are the parameter values used by the

algorithm. Finally, at the end of the file is a list of the connections specified in the input file.

3.2.3 Statistics File Format

The statistics file contains statistics from each generation of the genetic algorithm averaged over all iterations. Each line contains four values separated by commas. The first value is the generation number, followed by the number of individuals created, the number of converged individuals, and the best objective function value for the generation. Note that the statistics file is not intended to be viewed; it may be imported by software packages such as spreadsheets for plotting.

```
0,250,1,204.53334
1,437.733337,1.2,199.066666
2,632.533325,1.066667,197.53334
3,824.200012,1.4,197.800003
      .
      .
      .
997,189688.140625,127.199997,78.800003
998,189877.203125,135.133331,78.800003
999,190068,133.933334,78.800003
```

Figure 3-5 Sample statistics file

Chapter 4:
Results

This chapter describes the optimization of the parameter values for *CPGA*. The performance of the optimized genetic search is then compared with that of a simple "generate and test" search. Finally, the results of applying the program to several test problems is presented.

4.1 Parameter Optimization

Before a genetic algorithm may be usefully applied to problems, it must be "tuned" by experimentally determining parameter values that optimize its performance. Since there are usually many parameters, each of which may have a range of possible values, testing every combination of parameter values is clearly impractical. Therefore, educated guesses are made to determine initial values for the parameters. They may then be varied one at a time while the overall performance of the genetic search is observed. Optimizing each parameter independently of the others will not reveal possible interactions between parameters, however.

Since genetic searches are randomized, performance must be measured by averaging the results of many iterations of the algorithm performed under the same conditions. As defined in Chapter 2, *offline performance* is found by averaging the objective function value of the best individual found in each generation over all iterations of the algorithm. This value may then be plotted as a function of the generation number to create a graphical representation of the performance.

The placement problem used for parameter optimization is shown in Figure 4-1. An optimal placement of the 36 cells is shown; every net has a length equal to 1. Note that the cell array as a whole may be rotated and/or reflected about either axis to yield a total of 8 optimal solutions. These are the only placements for which every net has a length of exactly 1. Therefore, of the 3.7×10^{41} possible solutions, only 8 are optimal.

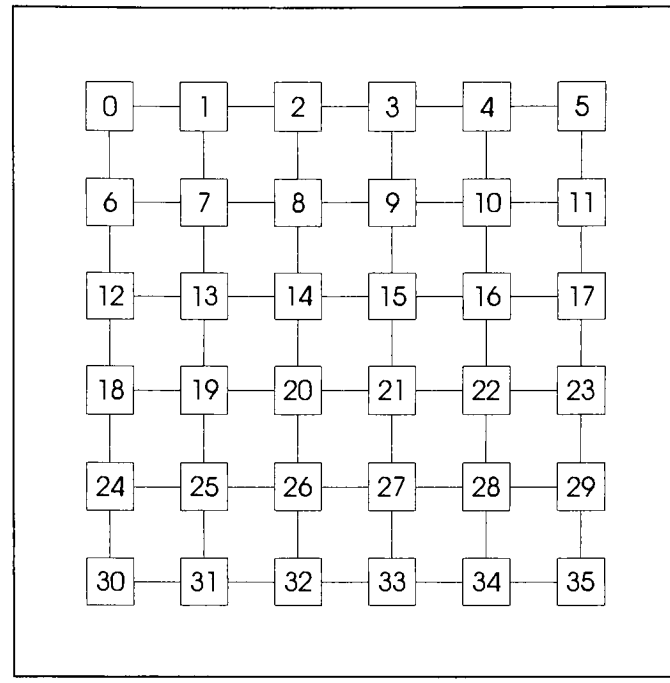


Figure 4-1 Placement problem P1

Based on the results of other experiments in genetic algorithms, the following initial values were chosen for the parameters:

seed value:	0
number of iterations:	30
generations/iteration:	1000
crossover type:	PMX
crossover region type:	rectangular
neighbor type:	n/a
queue type:	n/a
P(crossover):	0.70
P(mutation):	0.05
scale factor:	2.0

The seed value and number of iterations remain constant throughout the experiment. A seed value of 0 specifies that the seed for the random number generator will be taken from the system clock. Note that the neighbor type and queue type parameters only apply to flood regions.

4.1.1 Crossover Type and Probability of Crossover

The first parameters studied were the crossover type and probability of crossover. The possible crossover types are partially matched crossover (PMX), order crossover (OX), and order crossover with the greedy heuristic described in Chapter 2 (Greedy OX). The program was run using each of these operators for values of the probability of crossover ranging from 0 to 1 in 0.1 increments. The result is shown in Figure 4-2.

The figure shows that order crossover is superior to partially matched crossover, illustrating the importance of preserving the relative positions of cells during crossover. Also note that the greedy heuristic, which attempts to minimize the disruption to the chromosome outside of the crossover region, improves the performance of order crossover. Based on the results shown in Figure 4-2, greedy OX was chosen as the crossover operator, and 0.7 was chosen as the probability of crossover.

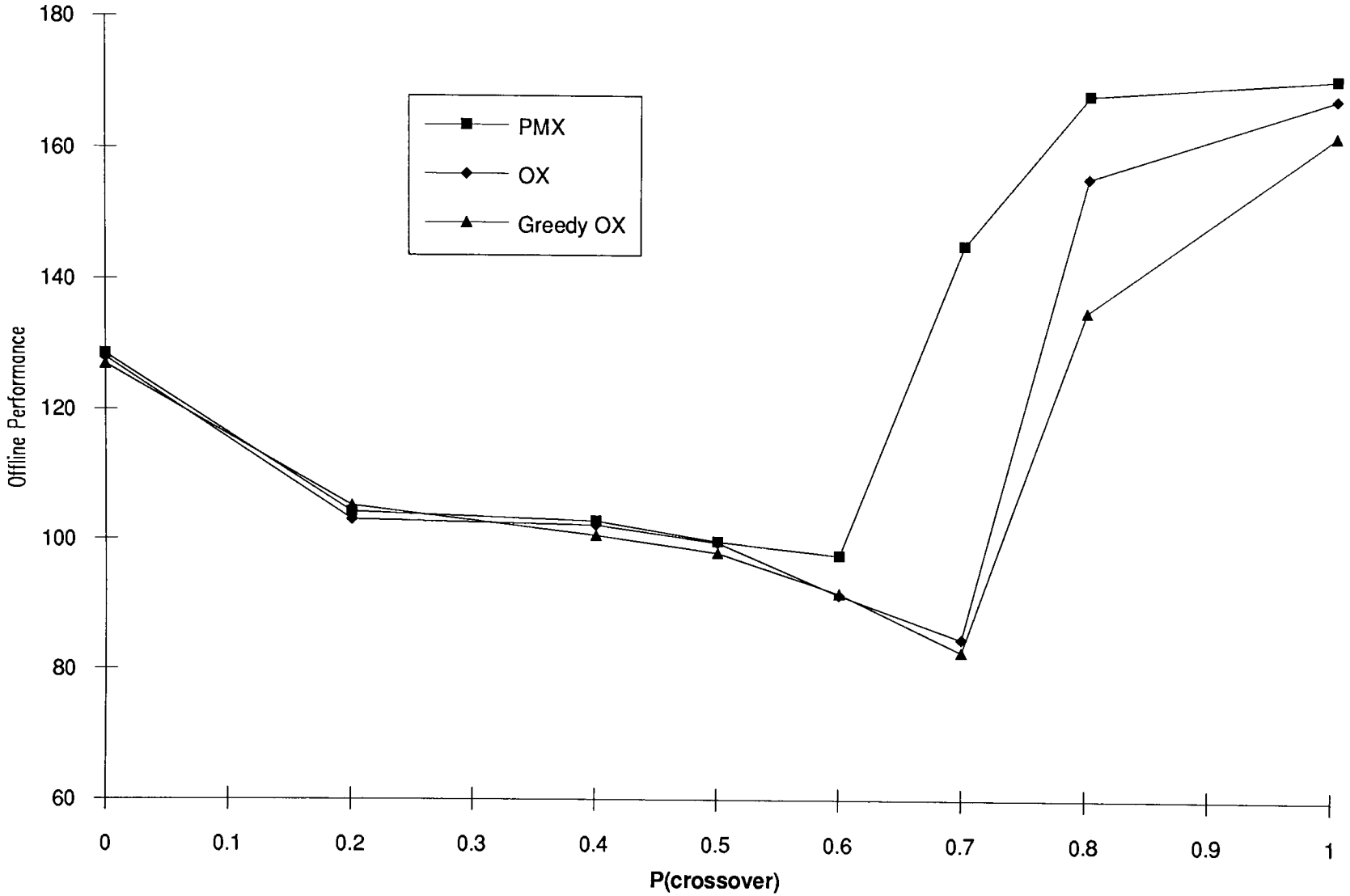


Figure 4-2 Offline performance of crossover operators

4.1.2 Crossover Region Type

The next parameter studied was the crossover region type. Possible region types are rectangular and flood. For flood regions, 4- or 8-neighbors may be used, and insertion of loci into the locus queue may be normal or random. The four possible flood region types are designated *F4N*, *F4R*, *F8N*, and *F8R*, where "F" indicates flood regions, "4" or "8" indicates the neighbor type, and "N" or "R" indicates normal or random queue insertion. The offline performance of these region types as a function of time (generations) is shown in Figure 4-3, where *Rect* indicates rectangular regions.

Figure 4-3 shows that using *F8R* regions produces the worst performance. This combination of 8-neighbors and random queue insertion results in the most irregularly-shaped regions. *F8N* and *F4N*, which tend to produce square and diamond-shaped regions, respectively, have approximately the same performance. Both of these region types are slightly better than rectangular regions.

The best region type appears to be *F4R*, which produces contiguous regions that are more regular than *F8R* but less regular than *F4N*, *F8N*, and rectangular regions. Although the genetic algorithm tends to converge more slowly with *F4R* regions, after approximately 700 generations its offline performance surpasses that of the other 4 region types.

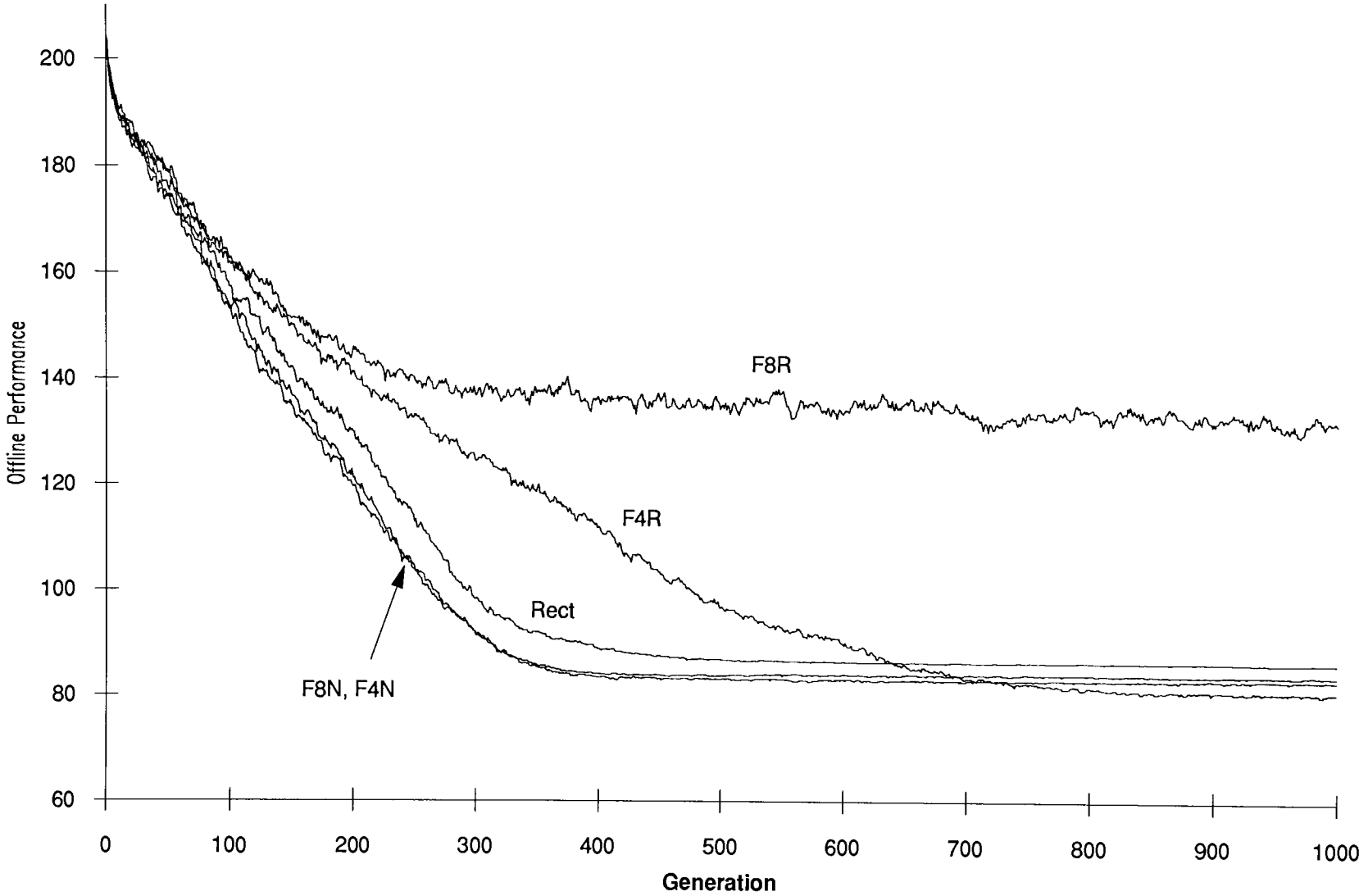


Figure 4-3 Offline performance of crossover region types

The results indicate that highly irregular *(F8R)* and highly regular *(Rect)* regions result in worse performance than "moderately" irregular regions. Based on Figure 4-3, *F4R* was selected as the region type.

4.1.3 Probability of Mutation

The effect of varying the probability of mutation was investigated next. Figure 4-4 shows the effect of this parameter on offline performance. With a low probability of mutation (0% 2%), the population converges quickly (300 400 generations) on relatively good solutions. Increasing the mutation rate to 6% causes the algorithm to find better solutions (at the expense of slower convergence) by reducing premature convergence. Further increasing this parameter results in worse performance. At high mutation rates, mutation prevents good solutions from evolving by quickly destroying highly fit building blocks. Based on Figure 4-4, 6% was chosen for the probability of mutation.

The convergence properties of the algorithm with varying mutation rates are more dramatically illustrated in Figure 4-5. This graph shows the average number of individuals with the best objective function value at each generation. (Recall that the population size is 250.) With no mutation, the population converges in roughly 300 generations. When the probability of mutation is 6%, roughly half of the population has converged after 1000 generations. Values greater than 8% prevent convergence from occurring.

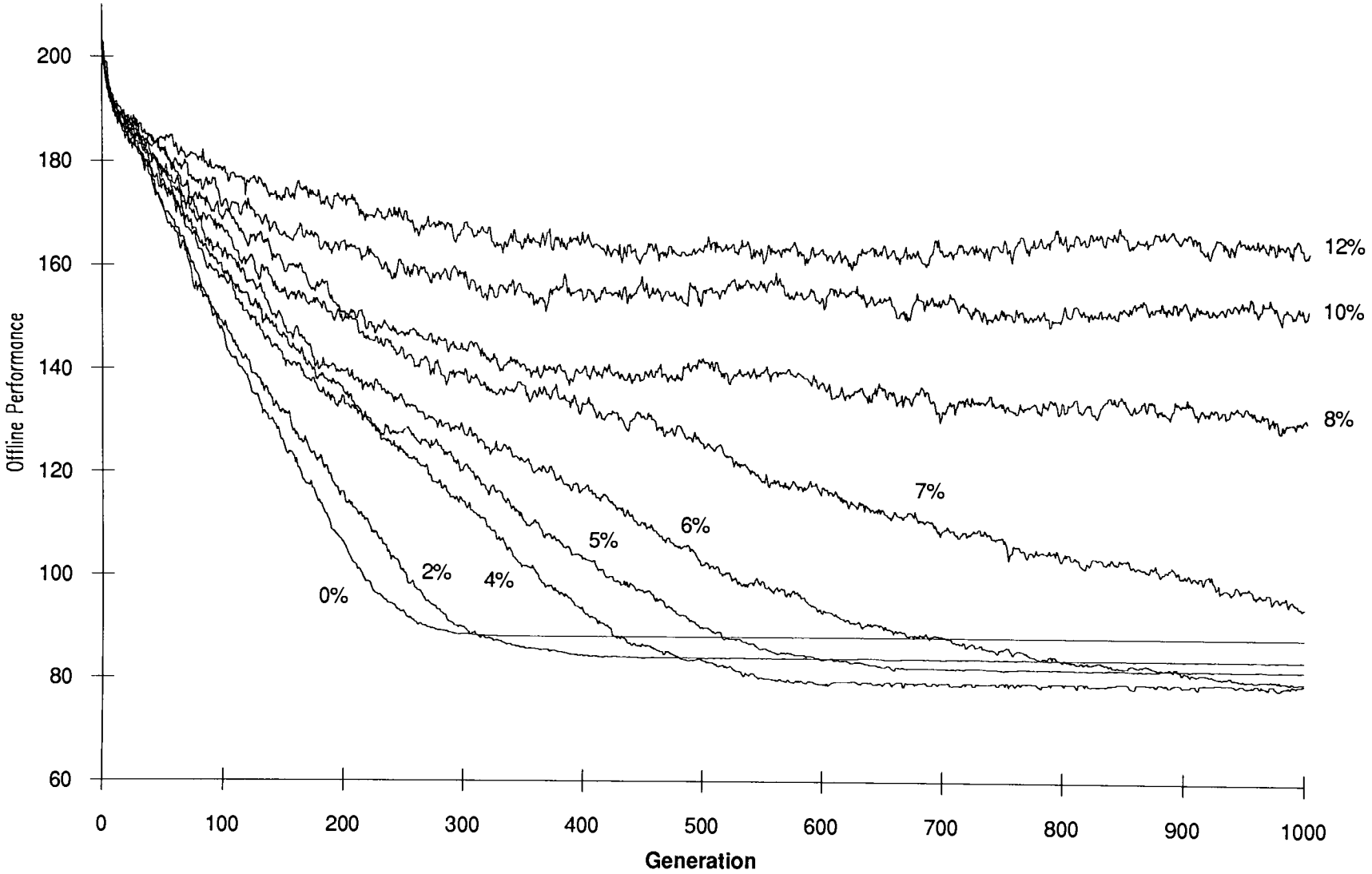


Figure 4-4 Offline performance with varying probability of mutation

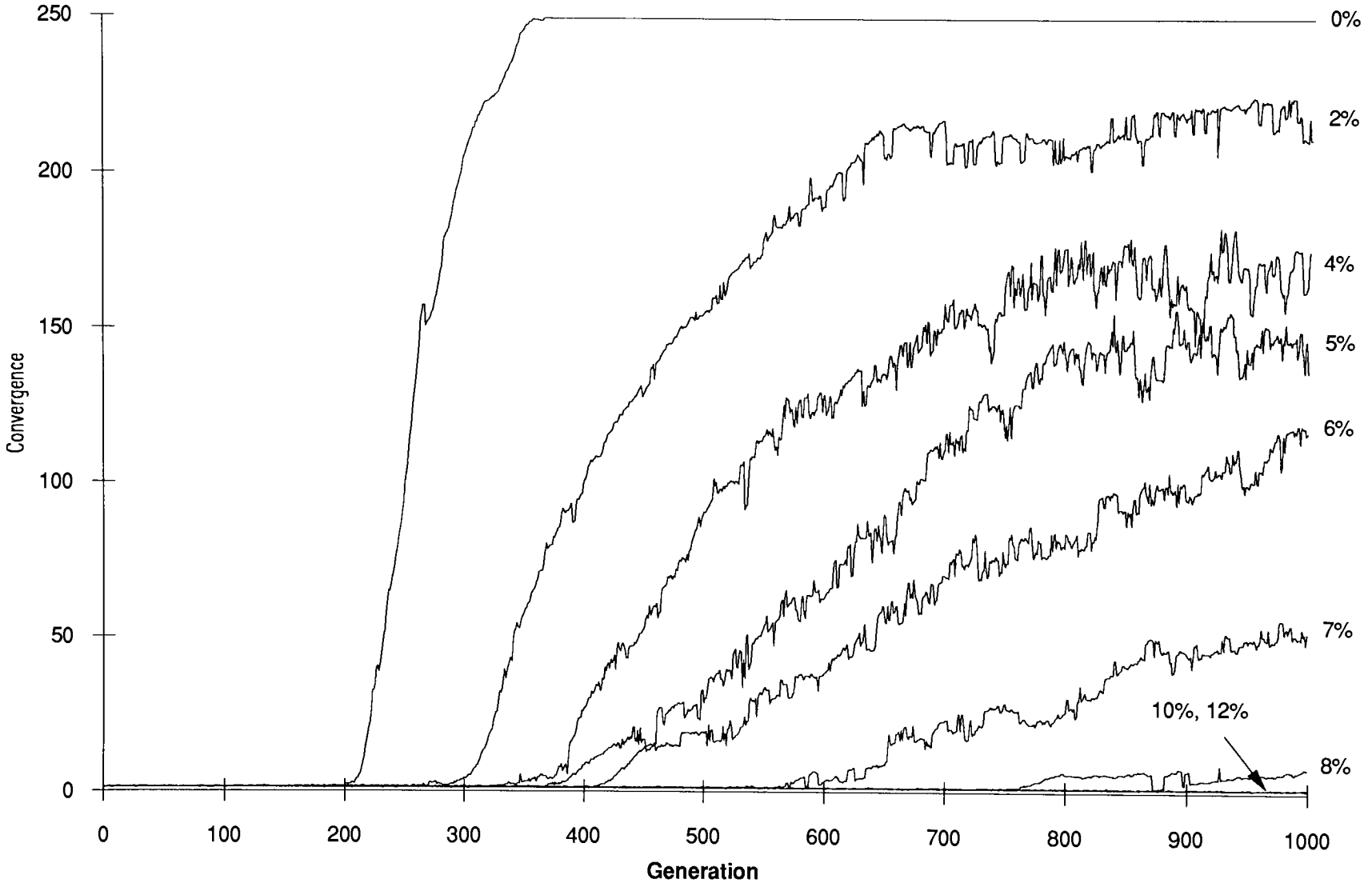


Figure 4-5 Convergence with varying probability of mutation

4.1.4 Scale Factor

The scale factor used by the linear scaling algorithm was the last parameter that was optimized. The offline performance for several values of this parameter is shown in Figure 4-6. For values close to 1, highly fit individuals do not receive enough copies in the mating pool to be properly exploited, and the performance of the algorithm suffers. For values greater than 2.1, these individuals receive too many copies, and this imbalance causes the population to converge prematurely on sub-optimal solutions. From the results shown in Figure 4-6, a value of 2 was chosen for this parameter.

The convergence properties are better illustrated in Figure 4-7. The graph shows that scale factors of 1.9 and less prevent the population from converging within 1000 generations. There is a significant improvement when the parameter is increased to 2, and values greater than 2 cause more rapid convergence.

It was found that increasing the scale factor beyond 2.1 results in little change in the offline performance. This is probably due to the non-negativity requirement of the linear scaling algorithm. If the desired scale factor is relatively high, then pivoting the fitness line using this value would cause the least fit individual to have a negative scaled fitness value. In this case, the line is pivoted as much as possible so that the least fit individual has a scaled fitness of 0. Therefore, the actual scale factor used is less

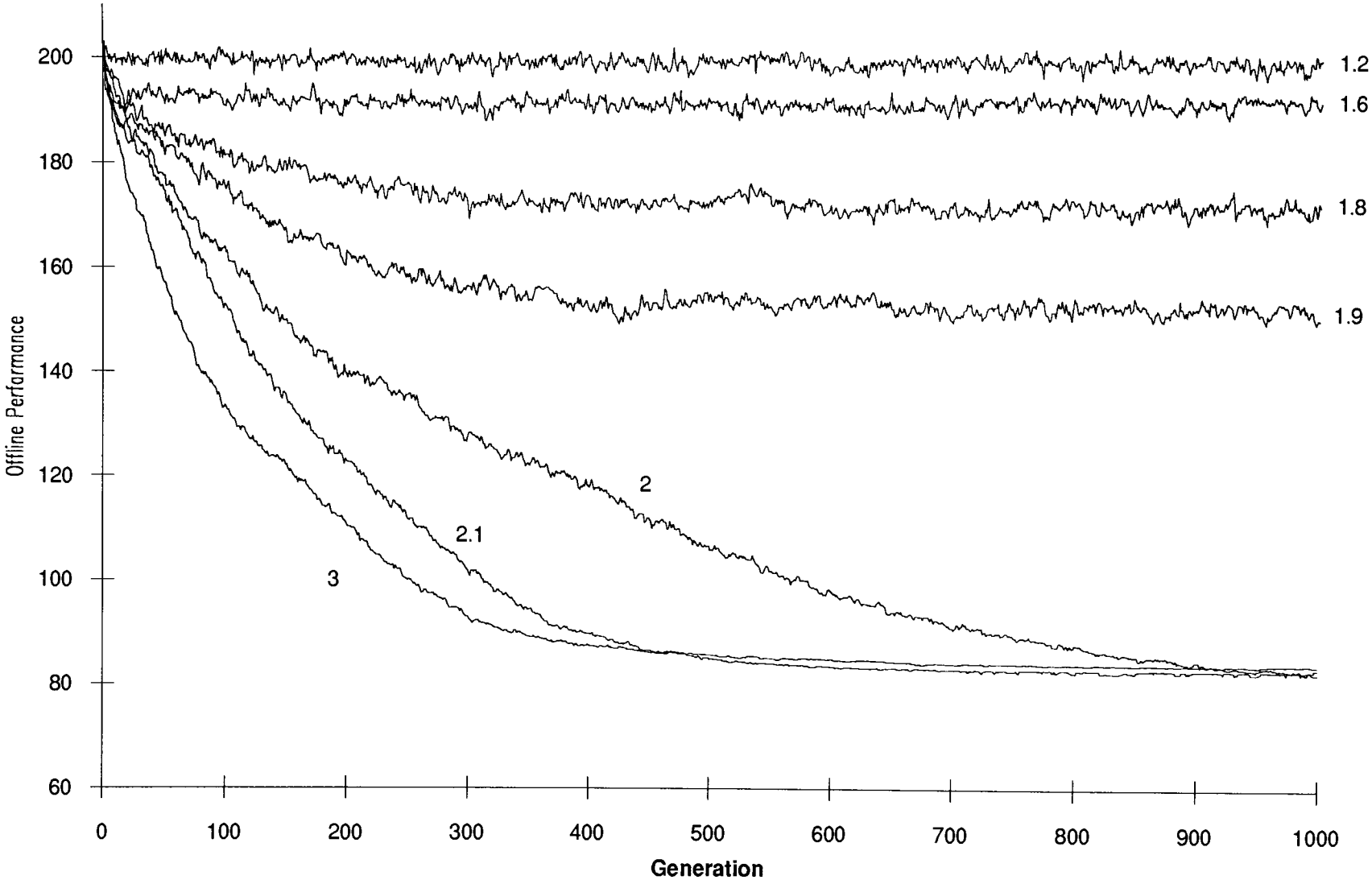


Figure 4-6 Offline performance with varying scale factor

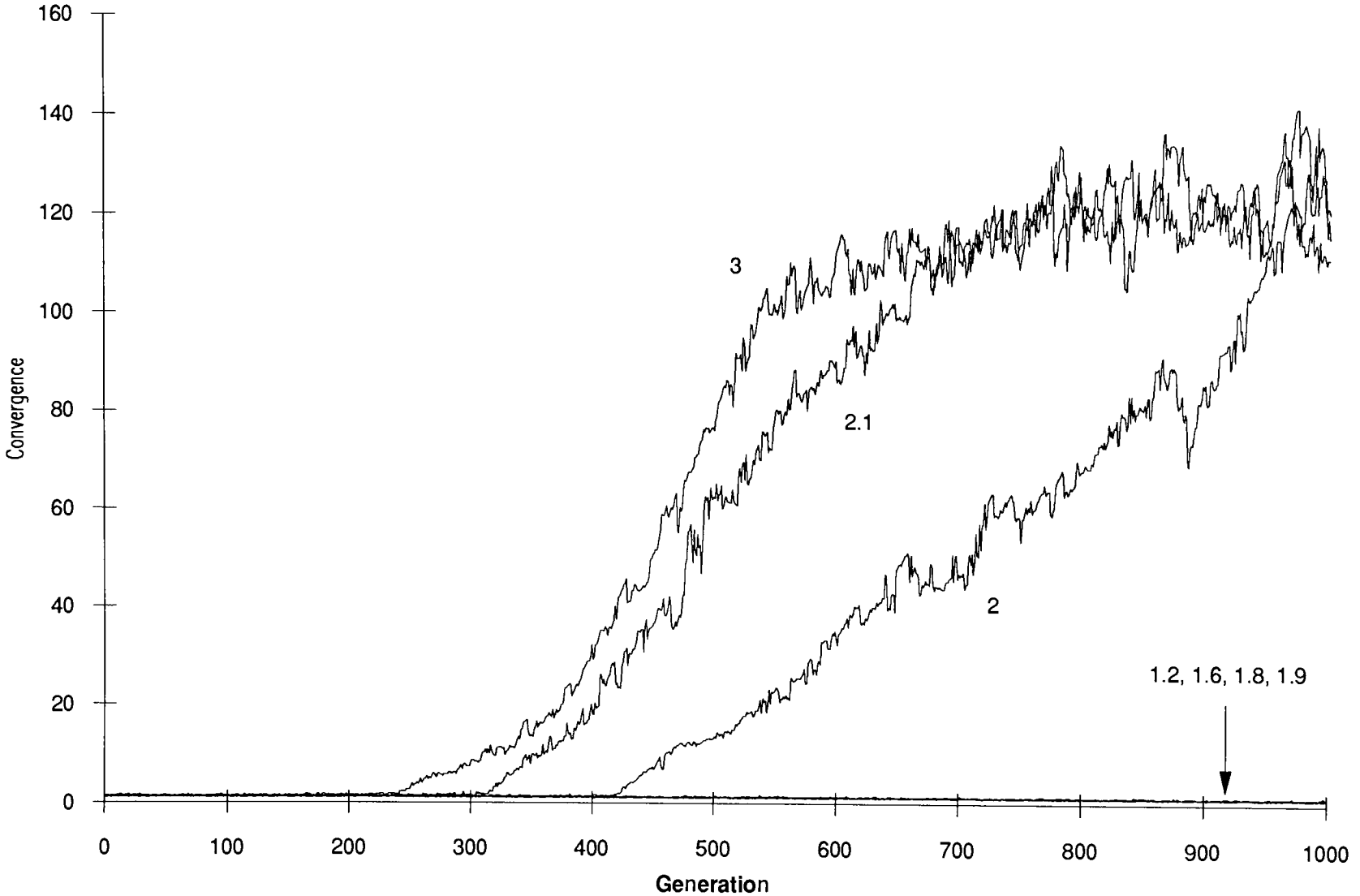


Figure 4-7 Convergence with varying scale factor

than the desired scale factor. Further increasing the desired scale factor in this case has no effect on fitness scaling, and therefore does not affect the performance of the algorithm.

4.1.5 Summary

Based on the results of these experiments, the following parameter values were selected:

seed value:	0
number of iterations:	30
generations/iteration:	1000
crossover type:	Greedy OX
crossover region type:	flood
neighbor type:	4-neighbors
queue type:	random
P(crossover):	0.70
P(mutation):	0.06
scale factor:	2.0

These have been made the default parameter values for *CPGA*.

4.2 Comparison of *CPGA* with a Random Search

It is interesting to compare the performance of the optimized genetic algorithm with that of a generate-and-test search. A simple program was written to generate and evaluate populations of random individuals. The offline performance at a particular generation is determined by averaging the best individual found in all generations up to and including that generation. This value is plotted as a function of the number of individuals generated, and the resulting graph is shown in Figure 4-8.

Genetic algorithms are *randomized* searches: crossover and mutation occur with certain probabilities, mating occurs between randomly selected individuals, crossover and mutation regions are randomly selected, etc. However, the performance of the genetic algorithm is far superior to that of the purely random search, as shown in Figure 4-8. This is because a random search represents exploration of the domain without regard for the quality of the intermediate solutions found. Genetic search, however, is a balance between exploration of the domain and exploitation of these intermediate solutions. The proper use of information in a population yields a new generation of better solutions. This is the basis for the success of genetic algorithms.

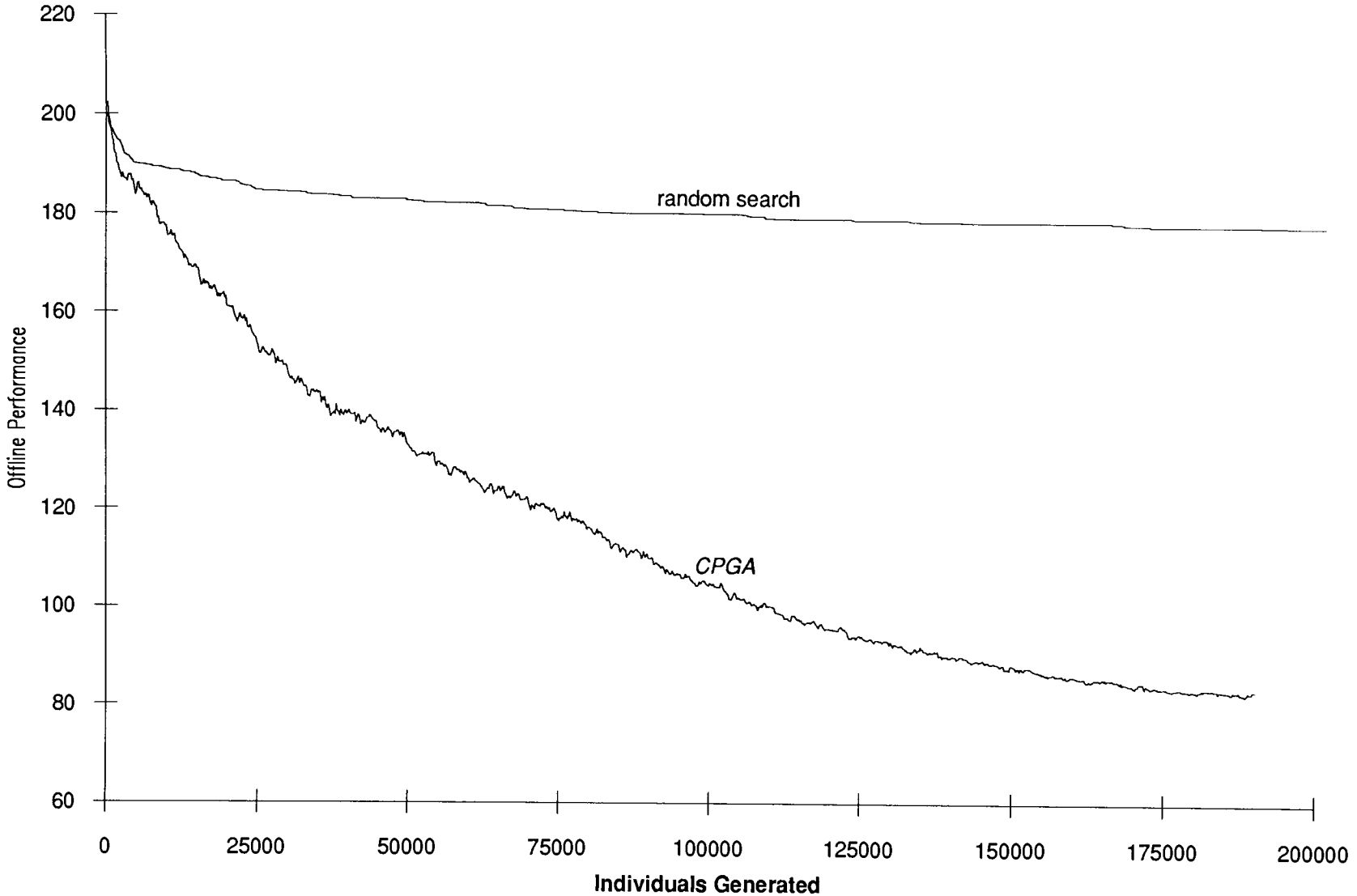


Figure 4-8 Performance comparison of CPGA and a random search

4.3 Test Problems

After the parameters of the genetic algorithm were optimized, *CPGA* was applied to eight test problems. Fifteen iterations of the genetic algorithm were performed for each problem. The problems (designated P1 through P8) and the best solutions found by *CPGA* are shown in Figure 4-9 through Figure 4-24. These figures show *rat's nests*, where nets are simply drawn as straight lines between the cells.

Optimal solutions were found by *CPGA* for problems P1, P3, and P4. Problems P3 and P4 are much easier than P1; the sets of connections for these problems are subsets of the set of connections for P1. As a result, many more optimal solutions exist for P3 and P4 than for P1. For example, the number of optimal solutions for P3 is:

$$8^4 \times 4! = 98{,}304$$

Recall that P1, the problem used for parameter optimization, has only 8 optimal solutions.

Suboptimal solutions were found for problems P2, P5, and P6. The solutions found are very good, however: *CPGA*'s solution for P6 has a total net length that is only 2 greater than that of the optimal solution.

Problems P7 and P8 are randomly-generated problems with 50 and 100 nets, respectively. The solutions found by *CPGA* illustrate how a good placement significantly reduces the net density in the array, making the design easier to route.

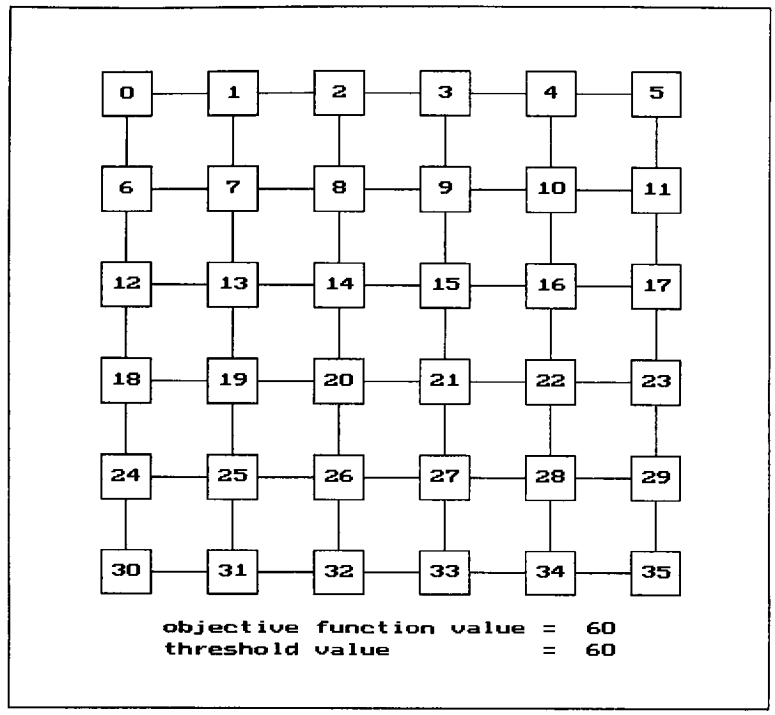


Figure 4-9 Problem P1

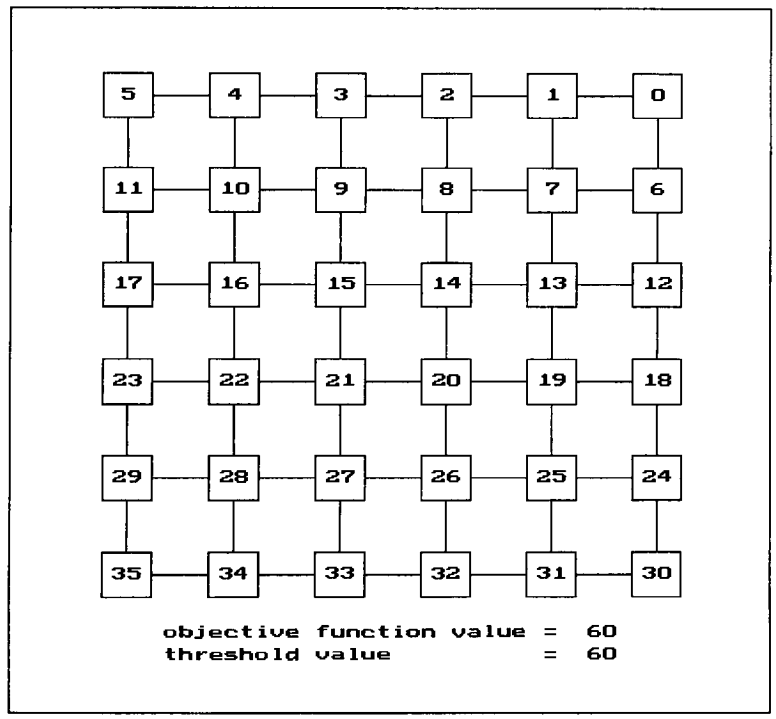


Figure 4-10 *CPGA* solution to P1

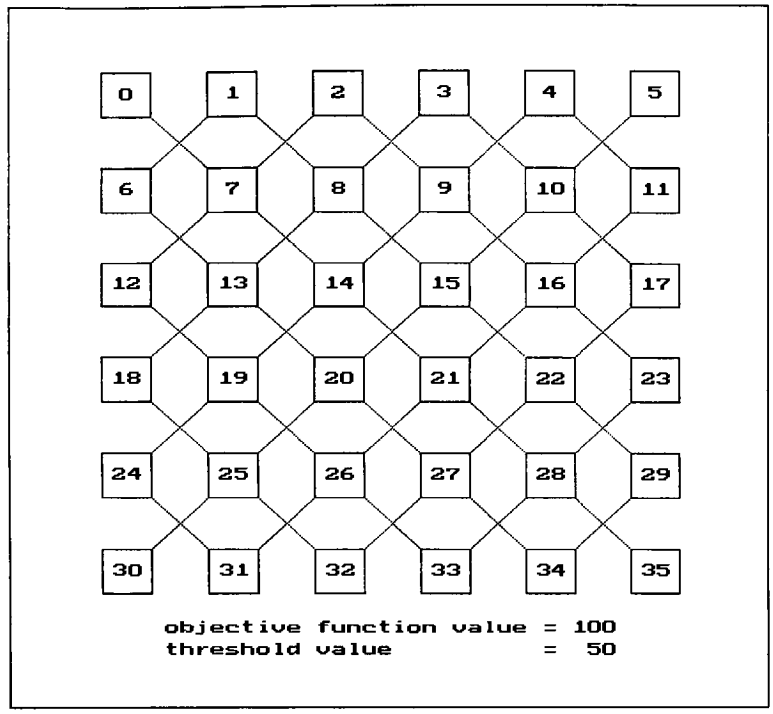


Figure 4-11 Problem P2

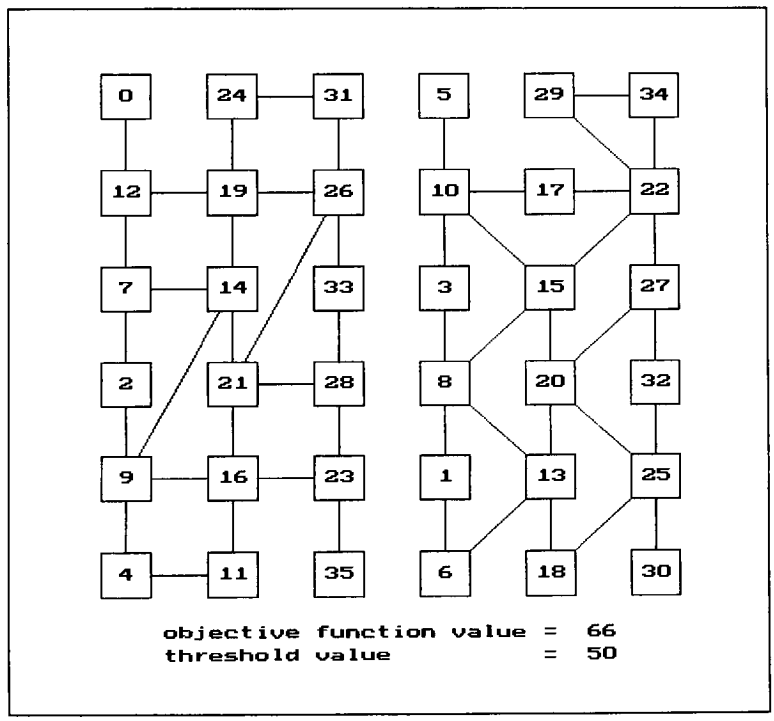


Figure 4-12 *CPGA* solution to P2

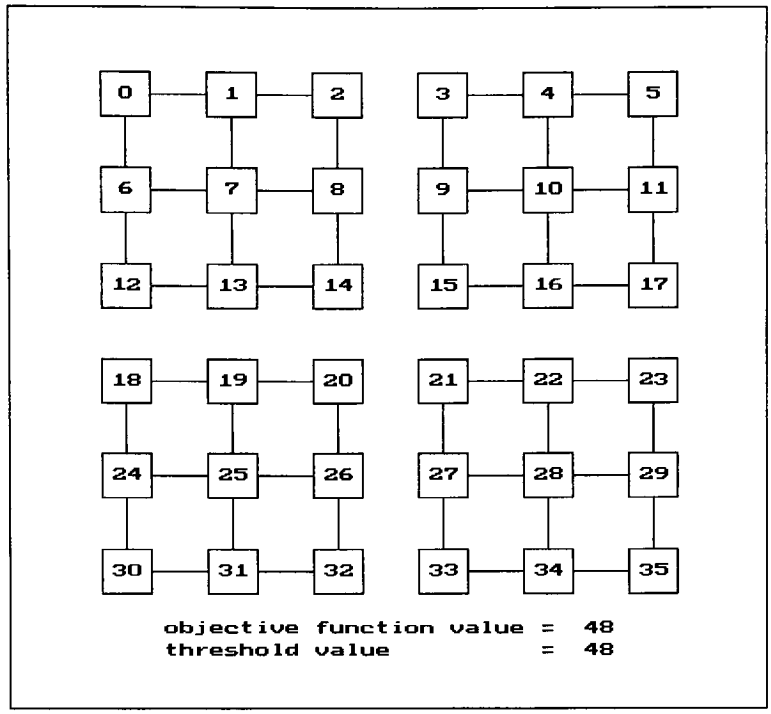


Figure 4-13 Problem P3

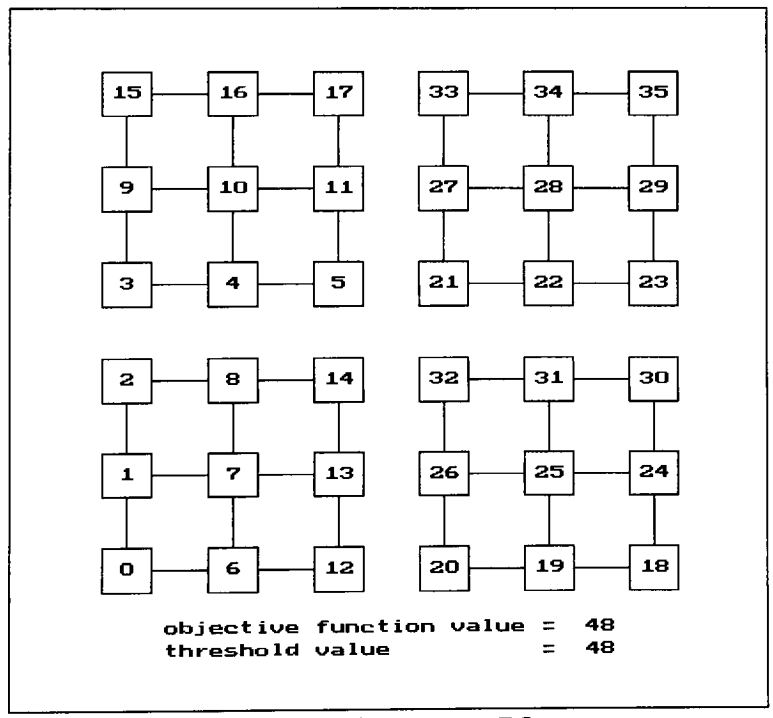


Figure 4-14 *CPGA* solution to P3

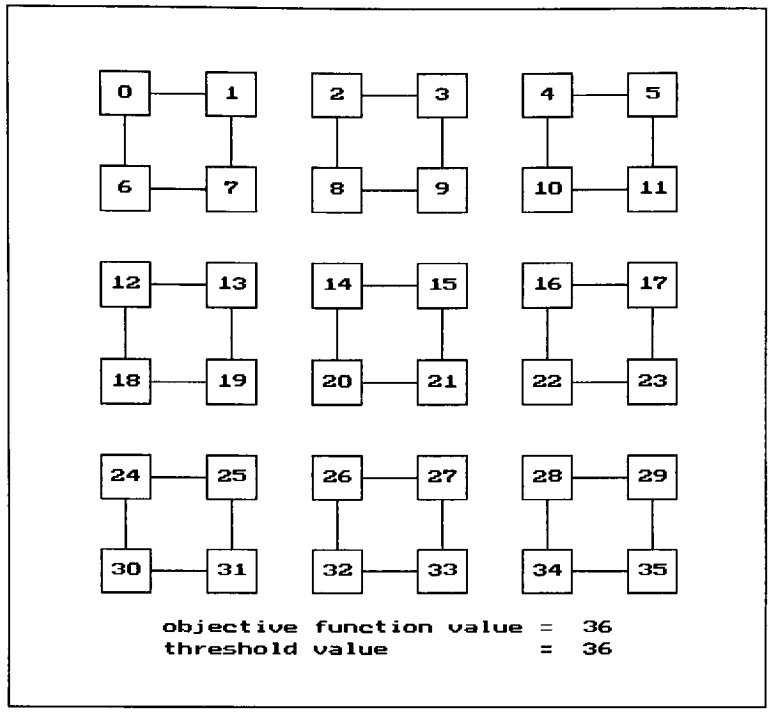


Figure 4-15 Problem P4

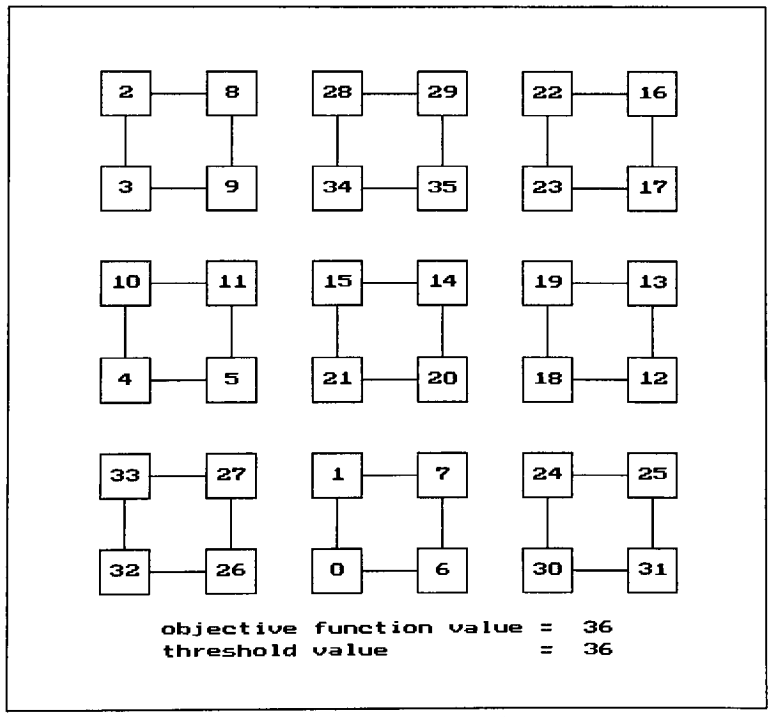


Figure 4-16 *CPGA* solution to P4

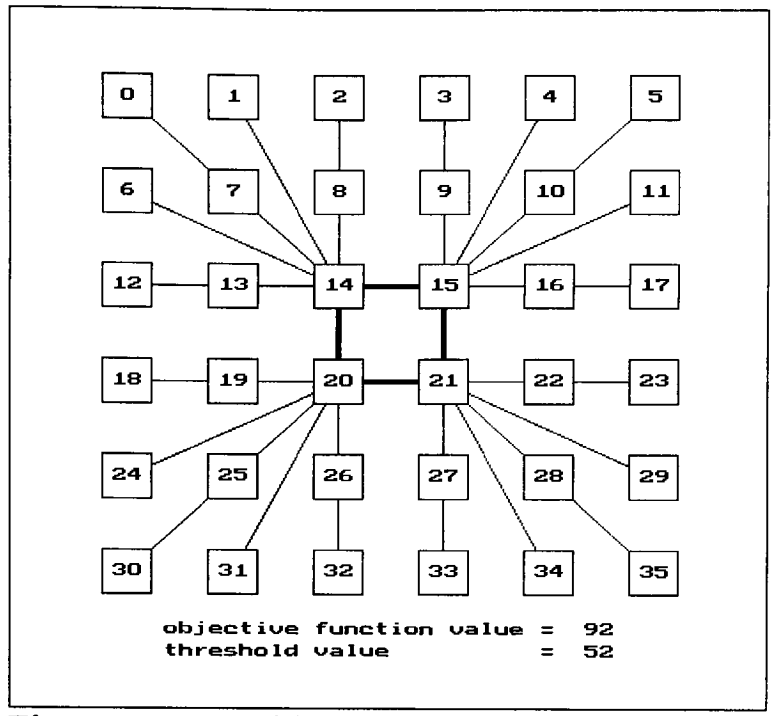


Figure 4-17 Problem P5

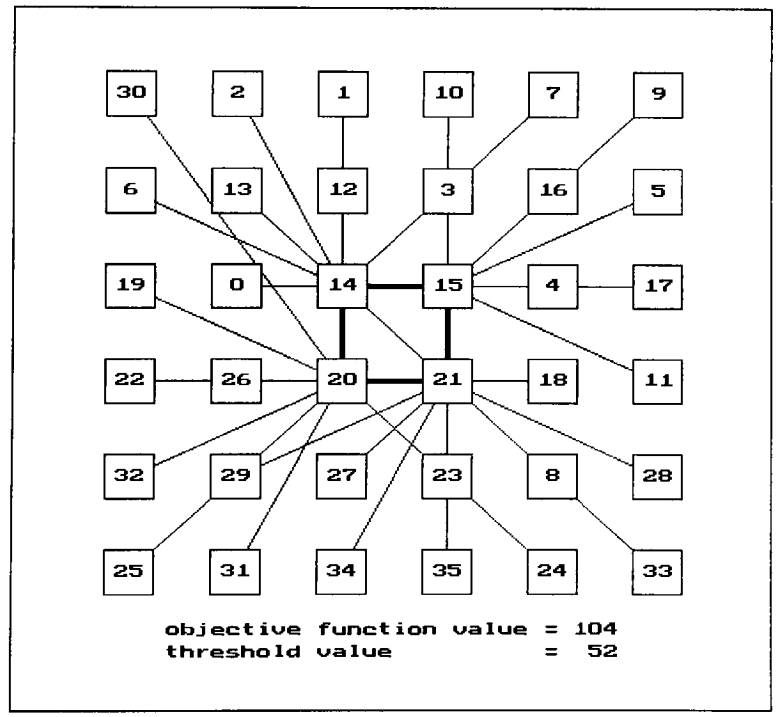


Figure 4-18 *CPGA* solution to P5

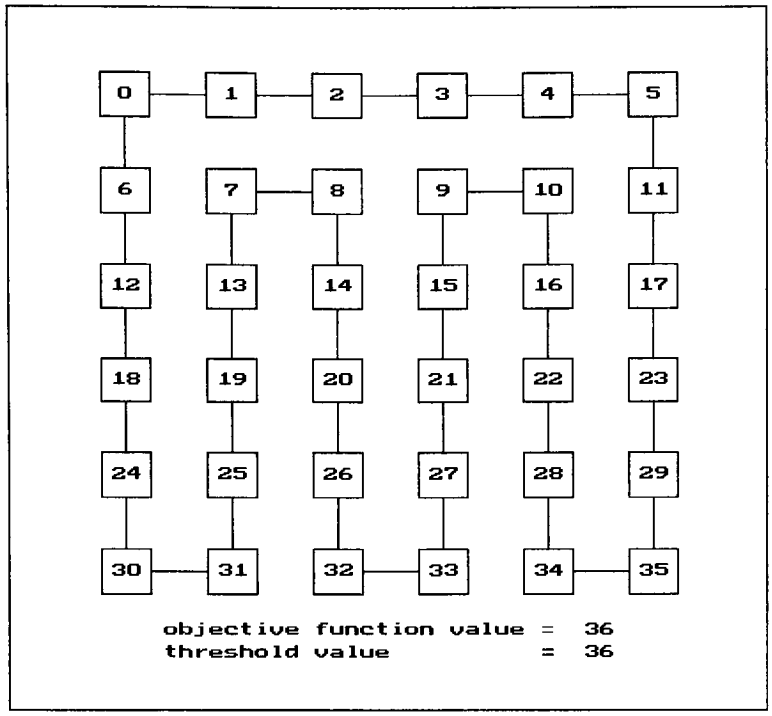


Figure 4-19 Problem P6

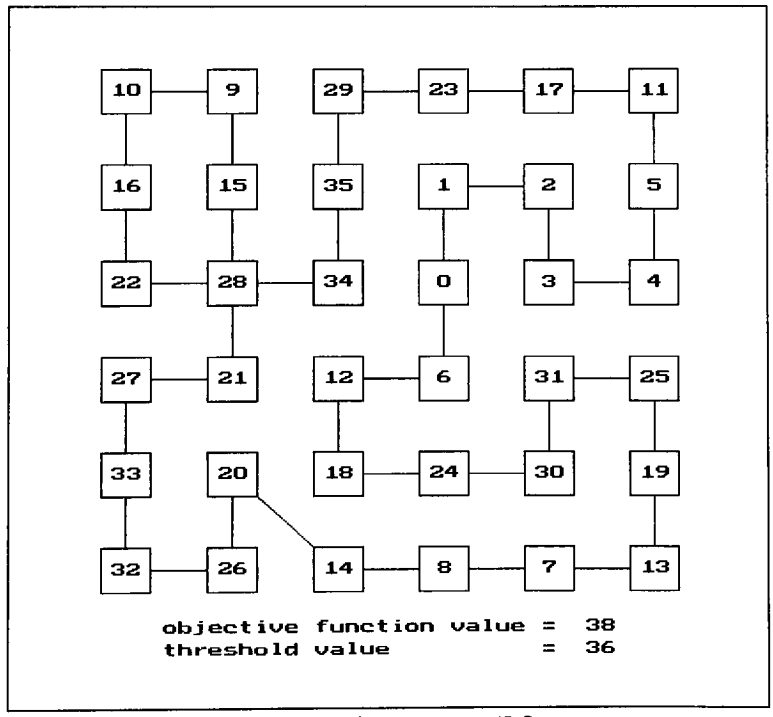


Figure 4-20 *CPGA* solution to P6

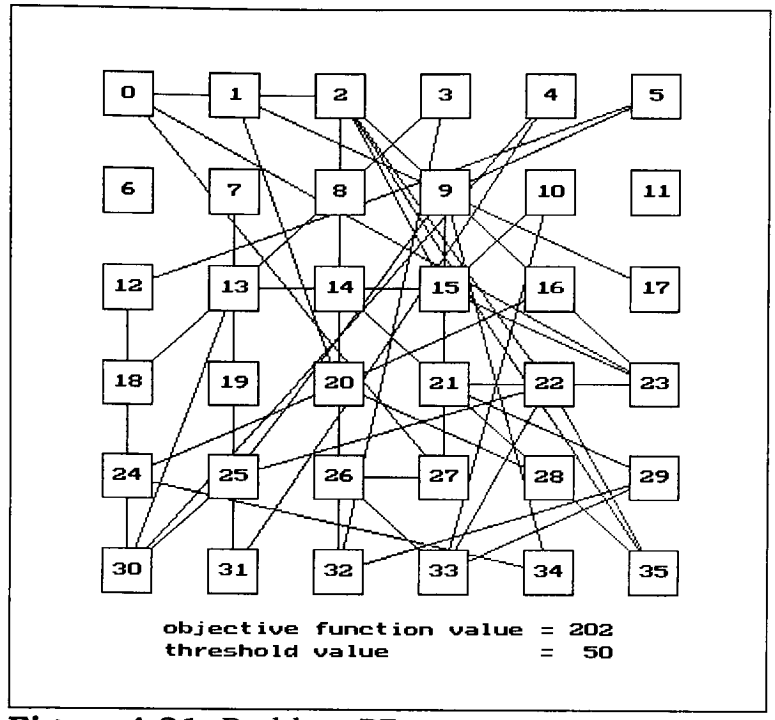


Figure 4-21 Problem P7

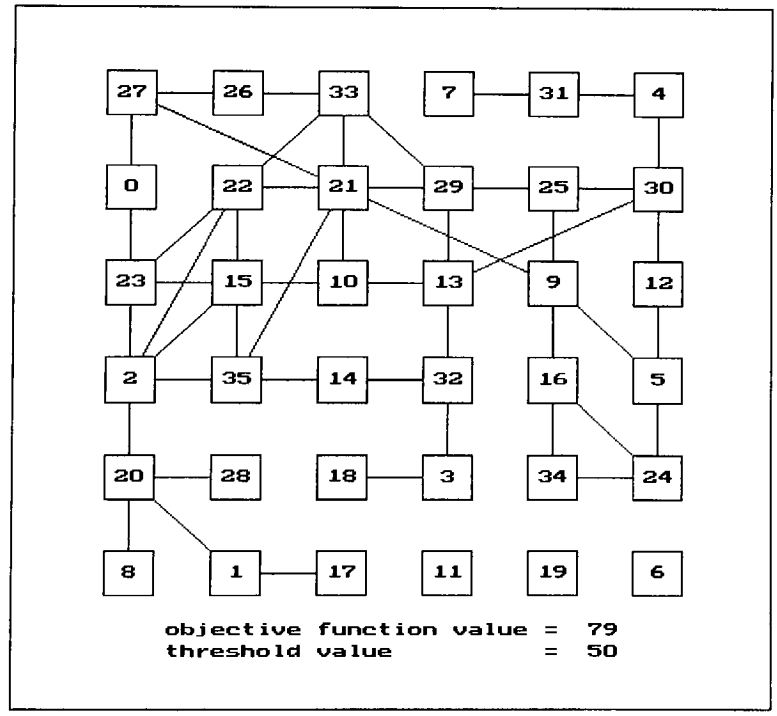


Figure 4-22 *CPGA* solution to P7

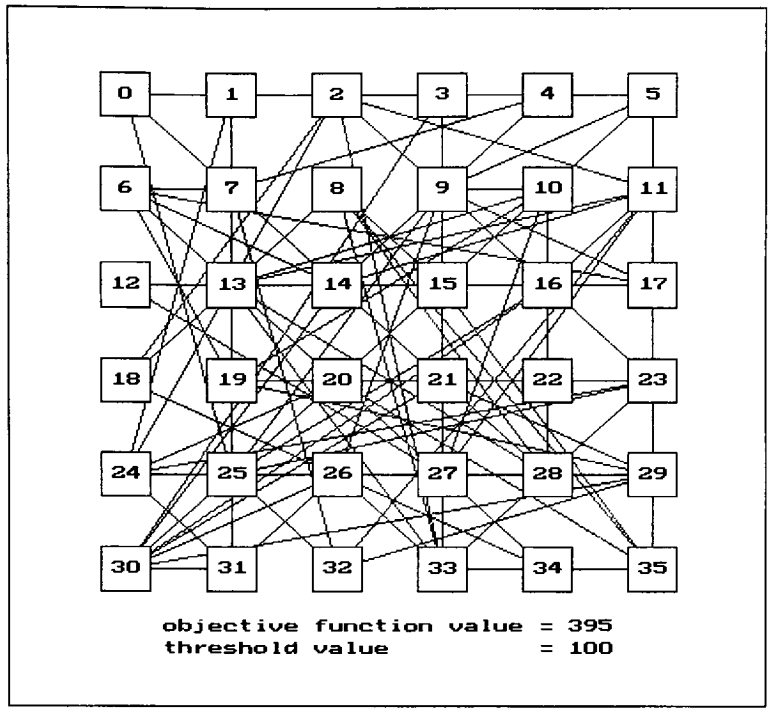


Figure 4-23 Problem P8

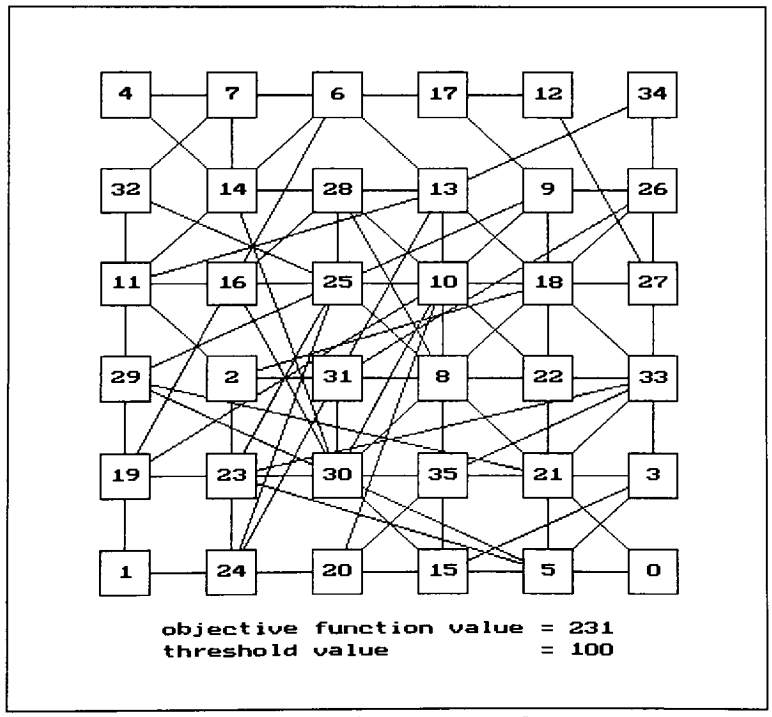


Figure 4-24 *CPGA* solution to P8

Chapter 5:
Conclusion

Cell placement is a relatively new application of genetic algorithms. Because of the encouraging results obtained so far, a great deal of research is being conducted in this area. Efficient cell placement for large circuits remains an elusive goal, however.

This chapter provides a brief overview of other placement algorithms. Improvements and enhancements to *CPGA* are suggested, as well as areas warranting further investigation. Finally, conclusions are drawn based on the observed performance of *CPGA*.

5.1 Overview of Placement Algorithms

This section summarizes an exhaustive study by Shahookar and Mazumder of current cell placement techniques. For more detailed information on the topics presented, the reader is referred to [SHAH91].

5.1.1 VLSI Technologies

Perhaps the most regular (and therefore easiest to place) VLSI geometry is the *gate array*. Gate arrays are prefabricated, rectangular arrays of logic gates with horizontal and vertical routing channels between the rows and columns. Once circuit has been designed, fabrication of a custom gate array simply requires additional masking steps to create the connections between gates.

Slightly less regular is *standard cell* layout. Standard cells are pre-designed functional blocks with equal height but varying widths. The width of a cell depends on its complexity. Cells are laid out in rows with horizontal routing channels between them. The input and output terminals of a cell are located at the top and/or bottom edge of the cell, while power is received through horizontal connections with adjacent cells.

The most difficult type of VLSI layout to place is composed of irregularly-shaped *macro blocks*. Macro blocks do not fit in regular rows and columns; placing them is analogous to a bin packing problem. Space is reserved around each block for routing connections.

The simplest way to model cell placement is with the *Checkerboard model*, which is used by *CPGA*. In this model, cells are assumed to be squares of equal size, and all connections are made at the centers of the cells. The length of a connection from one cell to its horizontal or vertical neighbor is one unit. When placing irregularly-shaped cells with this model, neighboring cells may overlap, creating illegal placements. This overlap may be factored into the objective function as a penalty.

5.1.2 Classification of Placement Algorithms

Two major classes of placement algorithms exist: *constructive* and *iterative improvement*. A constructive algorithm generates a placement from scratch. In iterative improvement, an initial placement is repeatedly

modified such that each modification results in a reduction in cost. *Repeated iterative improvement* simply repeats this process for several different initial placements. Genetic algorithms such as *CPGA* are repeated iterative improvement methods.

Placement algorithms may also be divided into *deterministic* and *probabilistic* algorithms. Deterministic algorithms are based on fixed rules or formulas and arrive at the same solution every time they are applied to a particular problem. Probabilistic algorithms may produce a different solution each time they are run depending on an initial "seed" value. Constructive algorithms tend to be deterministic, while iterative improvement algorithms such as genetic algorithms tend to be probabilistic.

5.1.3 Simulated Annealing

Simulated annealing is an iterative improvement algorithm modelled after the process by which metals are annealed. Starting with an initial placement, random changes to the placement (pairwise interchange, rotation of a block of cells, etc.) are generated. All changes that result in a cost reduction are accepted. Changes that cause an increase in cost are accepted with a probability that decreases with the magnitude of the cost increase. A "temperature" parameter also controls the probability of accepting cost-increasing moves; as the temperature decreases, the probability of accepting such moves decreases. This temperature parame-

ter starts at a high value and gradually decreases until only cost-reducing moves are accepted. The algorithm eventually converges on a low-cost placement.

As with genetic algorithms, the quality of the solutions found by simulated annealing depends on various parameters such as the initial temperature and cooling schedule. Much research has been devoted to finding values for these parameters that cause the algorithm to consistently find good solutions for a wide range of placement problems.

Simulated annealing is one of the most heavily-researched cell placement algorithms. Although it requires a relatively large amount of processor time, it produces very good placements. The algorithm has been applied with a great deal of success to many other combinatorial optimization problems as well.

5.1.4 Force-Directed Placement

The force-directed placement algorithm models nets as "springs." Cells connected by a net exert an attractive force on each other that is proportional to the distance between them (Hooke's law for springs). The goal of the algorithm is to find a placement for which the system is in equilibrium and a minimal energy state. This is achieved when the net force on each cell is zero and the total spring tension is minimized.

Both constructive and iterative variations of this method exist. Constructive algorithms model the cell coordinates as variables and derive a set of equations representing the forces on the cells. These equations are set equal to zero and solved simultaneously to yield the coordinates of the cells. The algorithm must ensure that the trivial solution, where connected cells are placed at the same location (resulting in zero net force), is not generated.

Iterative force-directed approaches start with an initial placement that is generated randomly or by a constructive method. The desired location for each cell is calculated based on the forces acting on it, and the cell is moved to that location. This is may be done by simply exchanging the cell with the one occupying its target location or by using a more sophisticated shifting scheme such as that used by order crossover in *CPGA*.

In the iterative approach, the cell with the strongest net force acting on it is usually selected as the first cell to move. Other possibilities include selecting the cell based on connectivity or simply selecting it randomly. Various methods exist for determining where to move the cell currently occupying the selected cell's target position.

One of the disadvantages of the force-directed placement algorithms is that achieving the minimal energy state does not always achieve optimal net length, and vice-versa. Also, because of the nature of the formulas

used, these algorithms tend to place all of the cells at the center of the array. Overall, force-directed algorithms are faster than simulated annealing, but the solutions found are not as good.

5.1.5 Placement by Partitioning

Placement by partitioning is performed by repeatedly dividing a circuit into subcircuits so that the number of connections between subcircuits is minimized. As the circuit is partitioned, the available chip area is also partitioned. When the partitioning is complete, each subcircuit is assigned to a partition of the chip area. If the circuit and chip area are repeatedly partitioned until each subcircuit contains exactly one cell, the result is an assignment of each cell to a unique location in the chip. This technique is also referred to as the *min-cut algorithm.*

The advantage of this method is the efficiency of the hierarchical decomposition of a large problem into smaller problems. This greatly reduces the size of the search space at each level of partitioning. By minimizing the connections between partitions, the interaction between the portions of the circuit that are placed independently is minimized.

The primary disadvantage lies in the fact that finding optimal partitions is also an NP-complete problem. Furthermore, finding optimal partitions does not guarantee an optimal placement. The advantage of this method is that heuristics for partitioning are much more well-developed

than those for placement. Like force-directed methods, partitioning is faster than simulated annealing but produces inferior solutions.

5.1.6 Numerical Optimization

A wide variety of traditional numerical optimization techniques have also been used for placement. These methods are deterministic and tend to be computationally intense. The primary disadvantage is that placement is nonlinear; it must either be solved with nonlinear programming methods or approximated by a linear problem before applying linear programming methods. Numerical optimization techniques are comparable to force-directed methods in speed and solution quality.

5.1.7 Genetic Algorithms

The last method of cell placement presented in [SHAH91] is the genetic algorithm. Three packages developed in recent years were discussed: Genie (1986), ESP (1987), and GASP (1990). Cell placement is still a relatively new application of genetic algorithms, and the initial results are promising.

The overall performance of the genetic algorithm is roughly equivalent to that of simulated annealing in terms of both execution time and quality of solutions. One disadvantage of genetic algorithms pointed out by the authors is the amount of memory required to store a population of

solutions. Simulated annealing repeatedly makes modifications to a single solution and therefore does not require as much memory.

The convergence properties of the genetic algorithm and simulated annealing are quite different. In a comparison between TimberWolf 3.3, a simulated annealing placement package, and GASP, a genetic algorithm written by Shahookar and Mazumder, it was found that GASP showed rapid improvement early in the search, but the performance levelled off later in the search. (*CPGA* follows this pattern, also.) TimberWolf, on the other hand, showed little improvement in the first half of the run. Therefore, if the quality of the solutions is not as critical, the genetic algorithm can find much better solutions in less time than simulated annealing.

Another difference between GASP and TimberWolf is in the number of solutions that were generated during the search. Although the execution times of the two programs were comparable, the TimberWolf explored 20 to 50 times more solutions than GASP.

5.1.8 Performance of Placement Methods

Of the placement methods discussed in [SHAH91], simulated annealing and genetic algorithms generate the best solutions but require the most computation time. Partitioning algorithms are next in terms of the quality of the solutions found, and they are much faster than genetic algorithms and simulated annealing. Force-directed algorithms and

numerical optimization techniques are both faster than genetic algorithms and simulated annealing, but they are slower than partitioning. The solutions found by these two methods are the poorest of the five methods discussed.

5.2 Improvements and Extensions to *CPGA*

This section suggests several improvements and extensions to *CPGA* based on the experience gained in implementing the genetic algorithm. These enhancements would improve the functionality of the program and possibly improve its overall performance as well.

5.2.1 Net Length Calculations

The model used by *CPGA* to estimate routed net lengths is the *source-to-sink* model. This model is exact for nets that connect pairs of cells. However, with this model, an output of a cell that is connected to inputs of three other cells would be treated as three separate nets. In real circuit layouts, the connection would not be made in this manner since it is a waste of interconnect space. Therefore, the source-to-sink model is not accurate for nets that connect more than two cells.

Several complex models for routing nets that connect more than two cells are presented in [SHAH91]. In the *Steiner tree* model, a net can branch at any point along its length. This method is seldom implemented

by routers because of the difficulty in calculating optimal branch points and the routes from the branch points to the destination cells.

In the *minimal spanning tree* model, nets may only branch from the points where they connect to the cells. Algorithms exist for computing minimal spanning trees from lists of nets and cell coordinates.

In a *chain connection*, there is no branching; the cells that are joined by a net are connected in sequence to form a chain. Although easy to implement, this method uses more interconnect space than spanning trees.

A more accurate estimate of routed net lengths that could easily be incorporated into *CPGA* is the *semiperimeter method*. With this method, the length of a net is approximated by half of the perimeter of a bounding box that encloses all of the cells connected by the net. This estimate is exact for nets that connect two or three cells (assuming that Manhattan wiring is used), and it is slightly lower than the actual net length for nets that connect four or more cells. In typical circuits, nets connecting two or three cells are most common. Therefore, this model would be a more accurate approximation than the source-to-sink model currently used by *CPGA*.

5.2.2 Crossover Regions

Recall that for the purpose of comparing flood and rectangular regions, the distribution of the number of cells selected as part of the crossover region is the same for both region types. (This is further

described in Chapter 3.) Lifting this restriction for flood regions could improve performance, since it would allow more diversity in the size of the regions. Rather than selecting two random numbers in the range [1..6] (for the 6 x 6 array) and using their product as the region size, the Region constructor could select a single random number in the range [1..36] and use this value instead.

It would be interesting to make the average crossover region size a parameter of *CPGA*. The region size for each crossover operation could then be randomly generated using a normal distribution about this point. This would reveal whether or not there is an optimal region size for crossover. Note that similar experiments could be conducted for the mutation region size.

As the genetic search progresses and the average fitness of the population increases, the portions of the chromosomes that are highly fit (building blocks) become larger. Another possible experiment with crossover regions is to increase the average region size with time or perhaps as a function of the fitness of the population.

With either flood or rectangular regions, the probability of a given cell being part of the crossover region is not independent of its locus. A cell in a corner or along an edge of the chromosome has a lower probability of being selected than cells towards the center of the chromosome. This is a result of the fact that the selected cells form a contiguous region. The

probability of a cell being selected is related to the probability that one of its neighbors is selected. Since the corner and edge cells have fewer neighbors, their probability of being selected is lower.

If we remove the constraint that the selected cells must form a contiguous region, then we could simply select cells randomly with equal probability. However, this method does not tend to preserve small building blocks. The performance of the highly irregular F8R regions seems to indicate that randomly selecting cells in this manner would result in even worse performance.

5.2.3 Crossover Operators

The order crossover operator used by *CPGA* shifts a hole from outside to inside of the crossover region in two steps. First, the hole is shifted horizontally until it is in the correct column. It is then shifted vertically to its destination. Another possible method for moving the hole is along an approximated straight line from its location outside of the crossover region to its destination inside of the region. This could be accomplished with Bresenham's line algorithm, an efficient line drawing method used in computer graphics. This algorithm uses only integer calculations and would be straightforward to implement in *CPGA*.

Using Bresenham's algorithm would reduce the average number of swaps needed to shift a hole to its destination. This is because some

diagonal movement would occur (unless the hole is in the same row or column as its destination). Each diagonal step is equivalent to one horizontal and one vertical step in the scheme currently being used. Therefore, diagonal movement results in fewer overall swaps. Reducing the number of swaps reduces the disruption of building blocks within the chromosome, which should improve the overall performance of the order crossover operation.

Only two crossover operators were investigated in this project: order crossover and PMX. Other crossover operators developed for permutation representations could be extended to two dimensions as well. For example, *cycle crossover* [OLIV87] was originally developed for the Traveling Salesman Problem (TSP), but this operator could be easily adapted for use in this project. It would be interesting to develop new crossover operators for the two-dimensional representation used in *CPGA* and study their performance characteristics. It may also be worthwhile to investigate combinations of crossover operators, where the operator to use for a particular crossover operation is chosen with roulette wheel selection. It is possible that a certain combination of operators will work better than any single operator.

5.2.4 *Mutation Operators*

Only one mutation operator was used in this project: inversion. There are many other possibilities for mutation that could be investigated: swapping alleles or groups of alleles within the chromosome, rotating square regions, etc. As with crossover, it may also be the case that some combination of mutation operators will produce better results than any single operator. Also, as noted earlier, the size of the mutation region is a parameter that may warrant further investigation.

5.3 Areas for Future Work

This section describes topics for future genetic algorithm research in the area of cell placement as well as for other applications. Many of these ideas are currently being investigated with great interest.

5.3.1 *Object-Oriented Programming and Genetic Algorithms*

The primary advantage of using C++ for *CPGA* was the data abstraction capabilities of the language. For example, the Chromosome class neatly encapsulates all of the relevant data structures and functions for implementing chromosomes. *CPGA* makes little use of the powerful inheritance capabilities of the C++, however.

For the purpose of genetic algorithm research, a more generic approach could be taken by developing a class library for implementing

genetic algorithms for any application. Classes in such a library would contain only those data structures and functions that are considered to be common to all genetic algorithms. For example, a Chromosome class could contain data members for the objective function and fitness values, as well as functions that manipulate these values such as the linear scaling algorithm. The generic Chromosome class would not contain the actual chromosomal representation, however. Developers of a specific genetic algorithm could then derive from the generic class their own chromosome class (e.g. CPGA_Chromosome) containing the representation and member functions (such as the objective function) that are specific to their application. They could then reuse many generic library functions without having to write them for their particular genetic algorithms. Once a well-designed class library has been implemented, genetic algorithms such as *CPGA* could be developed in significantly less time.

5.3.2 Guiding Genetic Search

A great deal of research is currently being conducted to find ways to "guide" the genetic search so that it converges on good solutions more rapidly. One approach is to seed the initial population with above-average individuals. Rather than randomly generating these individuals, some heuristic could be used to quickly create "good" solutions for the initial population. As noted in [GREF87], seeding must be done carefully, since

a few superior individuals in the population can lead to premature convergence. The initial population must be diverse enough to ensure that the genetic algorithm will explore many areas of the search space.

Another technique used to guide genetic search is *elitism*. In elitist approaches, the best individual from a given generation is copied to the next generation without being subjected to crossover or mutation. This guarantees that the individual will survive intact from one generation to the next. Although this approach has had some success [DAVI91], it may lead to premature convergence by forcing the genetic search in a particular direction.

Attempts have been made to introduce problem-specific knowledge into genetic operators. For example, a mutation operator may perform a local hill-climbing search by modifying a chromosome until an improvement in its objective function value is achieved. Knowledge-based crossover operators have been developed to increase the likelihood that offspring are more fit than their parents. Some experiments with such operators have met with success ([LIEP87], [SUH87]), while others have shown that "probabilistic choices are usually preferable to deterministic ones" [GREF87]. It seems likely that knowledge-based operators negatively affect the genetic algorithm's balance between exploration and exploitation. However, the development of these operators is still a worthwhile area for future research.

5.3.3 *Thermodynamic Genetic Operators*

An interesting hybrid of genetic algorithms and simulated annealing was created by Sirag and Weisser [SIRA87]. They essentially developed a thermodynamic operator which combines crossover, inversion, and mutation with concepts from simulated annealing. The level of "activity" of this unified operator decreases with the global system temperature. As with simulated annealing, the performance of the algorithm depends heavily on the annealing schedule chosen.

The developers have successfully applied this operator to the Traveling Salesman Problem. The quality of the solutions obtained makes this hybrid approach an intriguing area for further study.

5.3.4 *Adaptation in Genetic Search*

As well as adapting a population of individuals to their artificial environment, a genetic algorithm may simultaneously adapt parameters which control its search. In an interesting study by Schaffer and Morishima [SCHA87b], special information was appended to each chromosome indicating at what points in the chromosome crossover was permitted to occur. This information was passed on to offspring during crossover. If crossover at certain loci yielded poor offspring, then the crossover points would in effect die out with the individuals. The result was an adaptation

of the allowable crossover points and an overall improvement in the performance of the genetic algorithm.

Davis suggests adaptation of the probabilities of applying genetic operators [DAVI89]. These probabilities are adapted over the course of a run based on the quality of the offspring produced by the operators. An interesting application of this idea would be in determining the probabilities for several "competing" crossover or mutation operators. This would show whether a single operator or some combination of operators results in the best performance.

The intriguing aspect of adapting parameters during genetic search is that it involves little additional overhead. Finding optimal parameter values is a fairly difficult task, and designing the algorithm to find them as it searches would be a valuable feature.

5.3.5 Placement of Irregular Shapes

CPGA makes the simplifying assumptions that the cells being placed are identical in size and shape, and they are placed on a rectangular grid. While this is true for gate arrays, it is clearly not the case for standard cell or macro design. The problem of placing irregularly-shaped objects is much more complex.

One possible way of modelling irregular shapes in a genetic placement algorithm is to break up the shapes into squares of equal size. A

component is then made up of a cluster of square cells which must be moved together as a unit. Note that the orientation of a given component becomes important to the overall placement.

Another consideration with this approach is that overlapping components represent illegal placements. If overlap occurs, it could be accounted for as a penalty that is factored into the objective function for the placement. However, as noted by Davis, a genetic algorithm which allows the creation of illegal solutions will perform worse than one which does not [DAVI91]. Therefore, the genetic operators should be designed so that they do not produce placements with overlapping components.

Bin packing, which is the problem of putting rectangular boxes of arbitrary dimensions into a rectangular bin, is similar in many ways to placement of irregular shapes. An interesting approach to bin packing was taken by Smith [SMIT85]. Smith used a genetic algorithm to find the order in which to pack the boxes. This order was then given to one of two decoding algorithms that form legal packings. The fitness of a packing is computed as the ratio of the area of the packed boxes to the area of the bin. Therefore, dense packings have high fitness values. Smith's method was found to be 300 times faster than a deterministic algorithm using heuristics and dynamic programming techniques.

Another related problem is that of symbolic layout compaction. A symbolic layout is a representation of a circuit which contains information

on the relative layout of the components. In layout compaction, an attempt is made to minimize the circuit area by sliding elements closer to one another while retaining the topology of the connections. A genetic algorithm for layout compaction is described in [FOUR85].

5.3.6 Hierarchical Placement

It is evident that genetic algorithms can efficiently place small arrays of cells such as the 6 x 6 array used in this project. For large arrays, the size of the problem (and the execution time needed to get good solutions) increases with the factorial of the number of cells. Using *CPGA* "as is" for arrays with tens of thousands of cells would require a great deal of execution time.

A better way to place large arrays would be to use a min-cut algorithm to divide the design into manageable partitions which are fairly independent of one another. Once the cells comprising each partition are placed, the partitions themselves could be treated as cells and the entire design placed with the same algorithm. Performing hierarchical placement in this manner would result in a logarithmic reduction in the size of the problem. However, the quality of the placement would be very much dependent on the efficiency of the partitioning algorithm.

It is likely that a good partitioning of the design would result in partitions of varying sizes. Therefore, the placement algorithm must be

capable of placing irregular shapes. Also note that for very large designs, several levels of this hierarchical decomposition could be performed.

The task of efficiently partitioning the design may be reduced by the fact that large VLSI circuits are designed in a hierarchical manner. The design engineer may have several levels of functional blocks which make up the complete design. The resulting hierarchy would greatly simplify (or possibly eliminate) the partitioning stage of the placement algorithm.

5.3.7 Parallel Genetic Algorithms

Because of the processor time required by genetic algorithms, there is a great deal of research being conducted into ways to implement genetic algorithms on parallel processors. Intuitively, many stages of *CPGA* could easily be parallelized: the objective function and fitness calculations, selection[1], crossover, and mutation. This is possible because most of the operations manipulate individuals (or pairs of individuals, in the case of crossover) independently of one another, allowing different individuals to be operated on simultaneously by separate processors. Note that synchronization must occur at several points in the genetic algorithm.

One approach to parallelizing genetic algorithms being taken by researchers is to divide the population into several subpopulations,

[1]Although SUS can not be implemented with parallel processors, the Remainder Stochastic Independent Sampling (RSIS) selection algorithm developed by Baker produces comparable results and can be partially executed in parallel [BAKE87].

assigning each subpopulation its own processor. Each processor runs the serial genetic algorithm on its subpopulation, periodically passing copies of its most highly-fit individuals to other processors. An example of this technique implemented on a 64-processor hypercube is described in [TANE87]. The results of this experiment show a near-linear speedup for certain classes of problems.

5.4 Conclusion

This project has demonstrated the successful use of two-dimensional chromosomes and operators in a genetic algorithm. The particular problem solved by *CPGA*, cell placement, is a combinatorial optimization problem. However, it should be possible to create two-dimensional genetic algorithms for a wide variety of search and optimization problems where two-dimensional representations are more natural than string-based representations. It seems likely that similar success could be achieved with three-dimensional genetic algorithms as well.

Object-oriented programming languages seem natural for the implementation of genetic algorithms. For example, the concept of individuals as objects with encapsulated information (chromosomes) and functions which are permitted to manipulate this information (genetic operators) is well-suited to the OOP paradigm. To simplify the development of genetic algorithms for new applications, it would be useful to design a

generic class library. C++ seems to be a reasonable choice for the language since it is compiled and therefore much faster than interpreted languages such as Smalltalk. Also, C++ provides low-level operators for bit manipulation, which is a necessity for genetic algorithms that use bit strings for their chromosomal representation.

Genetic algorithms are very successful at solving problems with large, noisy search spaces that defy other optimization techniques. Furthermore, genetic algorithms can be used to find good solutions quickly or find near-optimal solutions after a longer period of time. In the case of cell placement, it is sufficient to find a solution that is "good enough" in the sense that it is routable and the resulting signal delays are acceptable. The rapid convergence of the genetic algorithm early in its search may give it an advantage over simulated annealing for problems of this nature.

The study of genetic algorithms is still relatively new. The increasing number of papers published in recent years indicates the growing popularity of artificial evolution among researchers. However, given the trend toward parallel computing, the future usefulness of genetic algorithms depends on the development of efficient implementations for multiple processor systems. Techniques such as the parallel evolution of subpopulations described earlier are being researched heavily with promising results. The success of this research ensures that genetic algorithms will continue to be studied and applied for many years to come.

References

[BAKE85] Baker, James E. "Adaptive Selection Methods for Genetic Algorithms." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[BAKE87] Baker, James E. "Reducing Bias and Inefficiency in the Selection Algorithm." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[BOOK85] Booker, Lashon B. "Improving the Performance of Genetic Algorithms in Classifier Systems." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[BOOK87] Booker, Lashon B. "Improving Search in Genetic Algorithms." *Genetic Algorithms and Simulated Annealing.* Lawrence Davis, ed. Los Altos: Morgan Kaufmann, 1987.

[DAVI87] Davis, Lawrence, and Martha Steenstrup. "Genetic Algorithms and Simulated Annealing: An Overview." *Genetic Algorithms and Simulated Annealing.* Lawrence Davis, ed. Los Altos: Morgan Kaufmann, 1987.

[DAVI89] Davis, Lawrence. "Adapting Operator Probabilities in Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[DAVI91] Davis, Lawrence, ed. *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold, 1991.

[DEJO89] De Jong, Kenneth, and William M. Spears. "Using Genetic Algorithms to Solve NP-Complete Problems." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[FOUR85] Fourman, Michael P. "Compaction of Symbolic Layout Using Genetic Algorithms." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[GOLD85a] Goldberg, David E. and Robert Lingle, Jr. "Alleles, Loci, and the Traveling Salesman Problem." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[GOLD85b] Goldberg, David E. "Genetic Algorithms and Rule Learning in Dynamic System Control." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[GOLD87a] Goldberg, David E., and Philip Segrest. "Finite Markov Chain Analysis of Genetic Algorithms." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[GOLD87b] Goldberg, David E., and Jon Richardson. "Genetic Algorithms with Sharing for Multimodal Function Optimization." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[GOLD89a] Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading: Addison-Wesley, 1989.

[GOLD89b] Goldberg, David E. "Sizing Populations for Serial and Parallel Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[GOLD89c] Goldberg, David E. "Zen and the Art of Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[GREF85] Grefenstette, John J., et al. "Genetic Algorithms for the Traveling Salesman Problem." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[GREF87] Grefenstette, John J. "Incorporating Problem Specific Knowledge into Genetic Algorithms." *Genetic Algorithms and Simulated Annealing.* Lawrence Davis, ed. Los Altos: Morgan Kaufmann, 1987.

[GREF89] Grefenstette, John J., and James E. Baker. "How Genetic Algorithms Work: A Critical Look at Implicit Parallelism." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[HOLL75] Holland, John H. *Adaptation in Natural and Artificial Systems.* Ann Arbor: The University of Michigan Press, 1975.

[JOG89] Jog, Prasanna, Jung Y. Suh, and Dirk Van Gucht. "The Effects of Population Size, Heuristic Crossover and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[LIEP87] Liepins, G.E., et al. "Greedy Genetics." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[LIPP89] Lippman, Stanley B. *C++ Primer.* Reading: Addison-Wesley, 1989.

[MUKH86] Mukherjee, Amar. *Introduction to nMOS and CMOS VLSI Systems Design.* Englewood Cliffs: Prentice Hall, 1986.

[OLIV87] Oliver, I. M., D. J. Smith, and J. R. C. Holland. "A Study of Permutation Crossover Operators on the Traveling Salesman Problem." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[REND85] Rendell, Larry A. "Genetic Plans and the Probabilistic Learning System: Synthesis and Results." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[SCHA87a] Schaffer, J. David. "Some Effects of Selection Procedures on Hyperplane Sampling by Genetic Algorithms." *Genetic Algorithms and Simulated Annealing.* Lawrence Davis, ed. Los Altos: Morgan Kaufmann, 1987.

[SCHA87b] Schaffer, J. David, and Amy Morishita. "An Adaptive Crossover Distribution Mechanism for Genetic Algorithms." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[SCHA89] Schaffer, J. David, et al. "A Study of Control Parameters Affecting the Online Performance of Genetic Algorithms in Function Optimization." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[SHAH91] Shahookar, K., and P. Mazumder. "VLSI Cell Placement Techniques." *ACM Computing Surveys*, Vol. 23, No. 2, June 1991.

[SIRA87] Sirag, David J., and Paul T. Weisser. "Toward a Unified Thermodynamic Genetic Operator." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[SMIT85] Smith, Derek. "Bin Packing with Adaptive Search." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1985.

[SUDK88] Sudkamp, Thomas A. *Languages and Machines: An Introduction to the Theory of Computer Science.* Reading: Addison-Wesley, 1988.

[SUH87] Suh, Jung Y., and Dirk Van Gucht. "Incorporating Heuristic Information into Genetic Search." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[SYSW89] Syswerda, Gilbert. "Uniform Crossover in Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[SYSW91] Syswerda, Gilbert. "Schedule Optimization Using Genetic Algorithms." *Handbook of Genetic Algorithms.* Lawrence Davis, ed. New York: Van Nostrand Reinhold, 1991.

[TANE87] Tanese, Reiko. "Parallel Genetic Algorithm for a Hypercube." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[WHIT87] Whitley, Darrell. "Using Reproductive Evaluation to Improve Genetic Search and Heuristic Discovery." *Proceedings of the Second International Conference on Genetic Algorithms.* John J. Grefenstette, ed. Hillsdale: Lawrence Erlbaum, 1987.

[WHIT89a] Whitley, Darrel, Timothy Starkweather, and D'Ann Fuquay. "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[WHIT89b] Whitley, Darrell. "The *GENITOR* Algorithm and Selection Pressure: Why Rank-Based Allocation is Best." *Proceedings of the Third International Conference on Genetic Algorithms.* J. David Schaffer, ed. San Mateo: Morgan Kaufmann, 1989.

[WHIT91] Whitley, Darrell, Timothy Starkweather, and Daniel Shaner. "The Traveling Salesman Problem and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination." *Handbook of Genetic Algorithms.* Lawrence Davis, ed. New York: Van Nostrand Reinhold, 1991.