Machine Learning, To Appear, 1–24 ()

Genetic Algorithms, Operators, and DNA Fragment Assembly

Machine Learning – To Appear

REBECCA J. PARSONS rebecca@cs.ucf.edu
Los Alamos National Laboratory
Current Address: University of Central Florida, Department of Computer Science,
Orlando, FL 32816-0362

STEPHANIE FORREST forrest@cs.unm.edu
University of New Mexico, Department of Computer Science,
Albuquerque, NM 87131-1386

CHRISTIAN BURKS cb@t10.lanl.gov
Los Alamos National Laboratory, Theoretical Biology and Biophysics Group,
MS K710, Los Alamos, NM 87545

Received October 30, 1993; Revised October, 1994: To Appear

Editor: David Searls

Abstract. We study different genetic algorithm operators for one permutation problem associated with the Human Genome Project—the assembly of DNA sequence fragments from a parent clone whose sequence is unknown into a consensus sequence corresponding to the parent sequence. The sorted-order representation, which does not require specialized operators, is compared with a more traditional permutation representation, which does require specialized operators. The two representations and their associated operators are compared on problems ranging from 2K to 34K base pairs (KB). Edge-recombination crossover used in conjunction with several specialized operators is found to perform best in these experiments; these operators solved a 10KB sequence, consisting of 177 fragments, with no manual intervention. Natural building blocks in the problem are exploited at progressively higher levels through "macro-operators." This significantly improves performance.

Keywords: genetic algorithms, DNA fragment assembly, human genome project, ordering problems, edge-recombination crossover, building blocks

1. Introduction

The computational problems posed by the Human Genome Project are challenging both because they are complex and because they involve large quantities of data. The Human Genome Project plans to identify the exact sequence of base pairs, called a *map*, for the entire human genome which consists of approximately 3 billion base pairs. There are many different components to this project; our problem involves combining partial information about the sequences of DNA fragments into a consistent map that accounts for the known pieces.

We explore the application of a genetic algorithm to the problem of DNA fragment assembly. We draw parallels to a more familiar permutation problem, the Traveling Salesman Problem [(Lawler, Rinnooy Kan, & Shmoys 1985)], both to explicate interesting features of our problem and as a source for possibly useful heuristics. Specifically, we find that the use of specialized operators provides good performance on data sets up to about 10KB in size. Two of these specialized operators, transposition and inversion, are macro-operators in that they transform the individual based on groups of fragments as opposed to single fragments. These groups of fragments, called *contigs*, are the natural building blocks for the fragment assembly problem. We found adding these macro-operators, which operate directly on the building blocks, significantly increased the performance of the genetic algorithm. Throughout the course of a run, the genetic algorithm assembles larger and larger building blocks (contigs), and the macro-operators thus operate at a higher and higher level. This progression is an explicit example of the implicit behavior described by the building-blocks hypothesis.

The accuracy of the various sequencing processes constrain laboratory approaches to DNA sequencing [(Howe & Ward 1989); (Hunkapiller, Kaiser, & Hood 1991); (Hunkapiller *et al.* 1991); (Churchill *et al.* 1993)]. Currently, strands of DNA longer than approximately 500 base pairs cannot routinely be sequenced accurately. Consequently, large strands of DNA are broken into smaller pieces for sequencing. In the shotgun sequencing method, to which this work applies, DNA is first replicated many times, and then individual strands of the double helix are broken randomly into smaller fragments. This produces a set of fragments short enough to sequence. However, this process does not retain either the ordering of the fragments on the parent strand of DNA or the strand of the double helix from which a particular fragment came. This paper addresses the first of these problems, hereafter referred to as the *fragment assembly* problem, relying on previously developed methods for addressing the alignment and strand assignment problems [(Staden 1980); (Kececioglu 1991); (Huang 1992); (Churchill *et al.* 1993)].

Large-scale shotgun sequencing projects require automated solutions that do more than re-create the manual processes, because the complexity of the assembly process grows exponentially with the size of the project. There are several complicating factors to be considered in designing computational solutions. First, there is a large amount of experimental error. Frequently quoted rates are between 0.1% and 10% [(Chen & Hunkapiller 1992)]. Further, repeated DNA sequences can be much longer than individual sequence fragments. Finally, the reagents and the experimentalists' time are valuable resources, so an important objective of any computational system is to monitor the progress of the sequencing to determine if other strategies need to be applied. The target parent size of many upcoming sequencing projects is cosmid size (about 40,000 base pairs, denoted as 40KB). Most experimentalists use coverage (sequencing redundancy at a particular point along the parent DNA) of at least 5 or 7 to compensate for some of the effects of sequencing errors. Using these figures (and an assumed average fragment length of 500 base pairs) leaves an ordering problem of approximately 600 fragments. As the ordering problem is NP-hard [1], an approximate method is required to determine a reasonable layout.

Most fragment assembly packages use a greedy algorithm to form the candidate. Typically in a greedy algorithm, a candidate solution is presented to the researcher who must then massage it to obtain a biologically plausible final result. Simulated annealing has been applied to the ordering step of the fragment assembly problem [(Churchill *et al.* 1993); (Burks *et al.* 1994)], and genetic algorithms have been applied to this problem by the authors [(Parsons, Forrest, & Burks 1993)] and to a related ordering problem, map assembly, by others [(Fickett & Cinkosky 1993); (Cedeno & Vemuri 1993)].

The next section of this paper contains a detailed explanation of the flow of information in the fragment assembly problem and the general computational approach we follow. Section 3 details the genetic algorithm explored in this paper, with the results appearing in Section 4. We explore the answers to some of our questions, and pose additional questions in the final section.

2. The Fragment Assembly Problem

Fragment assembly is only one step in the overall process of building a base-pair map for an unknown segment of DNA. The other steps in the process influence fragment assembly in several ways: they affect the overall quality of the information used by the assembly; they influence the quality of the final solution; and they introduce conflicting information and errors into the process. An overview of the process is shown in Figure 1 (see also [(Churchill *et al.* 1993); (Burks *et al.* 1994)] for a more detailed explanation of the sequencing process).

The laboratory sequencing process provides a set of fragments and, for each fragment, the base pair sequence for that fragment. [2] Because the fragments can come from either of the anti-parallel strands of the parent DNA, the orientation of the fragment relative to the parent is not known. At the assembly stage, the only information available is the sequence of bases, and thus the ordering of the fragments must rely primarily on the similarity of fragments and how they overlap. A particularly important aspect of the general sequencing problem is the precise determination of the relationship and orientation of the fragments. A complicating factor in the overlap computation is the frequent occurrence of repeated sequences, ranging in length from several bases to several thousand bases. [3] Any DNA fragment assembly method based on sequence similarity is bound to be misled by DNA repeats, confusing fragments which are similar because they originate from the same location in the parent sequence with fragments that are similar because they share a repeat pattern.

Once the fragments have been ordered, the final consensus sequence is generated from the ordering. This process includes a detailed alignment step that must account for the insertion and deletion errors potentially present in the data. As shown in Figure 2, the steps from raw sequence data from a random sequencing project to a consensus sequence are as follows [(Churchill *et al.* 1993); (Burks *et al.* 1994)]:

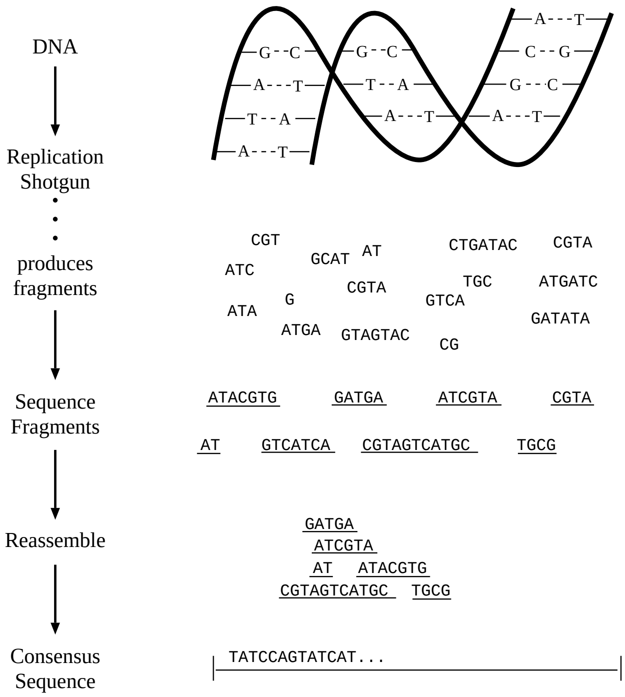


Figure 1. Overview of DNA Mapping Process

1. Compute pairwise relationships. Compare each pair of fragments and determine their similarity, resulting in an overlap strength, alignment and relative orientation of the two fragments. Each possible orientation is tried for the two fragments, and the overlap, orientation, and alignment are chosen to maximize the similarity of the fragments.

2. Totally order the fragments. The ordering algorithm computes the fitness, or figure of merit, for a candidate ordering by examining the overlap information. In addition to the fragment ordering, a particular layout results in *contigs* [(Staden 1980)]. A contig is a layout with no gaps; gaps occur when neighboring fragments do not overlap.

3. Determine initial alignment. Use the alignment, offset and orientation information from the first step and the ordering from the second step to determine an initial alignment of the fragments.

4. Determine the detailed alignment of the fragments, also known as *multiple sequence alignment*. Starting with the initial alignment, the bases within the fragments are examined to determine places where insertion or deletion errors likely occurred. To account for these errors, gaps are inserted into the fragments to bring corresponding bases back into alignment. See [(Waterman 1989)] for a discussion of multiple sequence alignment.

5. Generate the consensus sequence. Each column in the detailed alignment is examined to determine the "consensus" base for this position, yielding the consensus sequence for the contig.

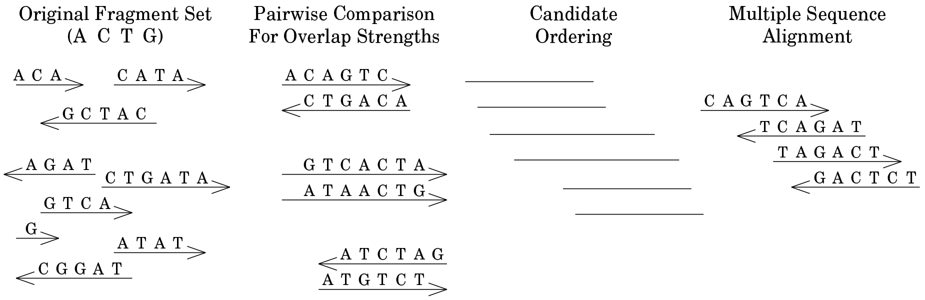


Figure 2. Overview of Sequence Assembly Process

The ordering step must find a total ordering of the given fragments that results in a consensus sequence accurately reflecting the parent sequence. Each fragment must be accounted for in the ordering, and each fragment can only appear in one place in the ordering. With previously solved sequences, the parent sequence is known, and we can judge the quality of an ordering by how closely the final consensus sequence corresponds to the known parent. In practice, however, the fragment sets are being generated to find the parent sequence. Thus, some other criteria must be used to evaluate an ordering. Although a small number of contigs is one goal of the orderings, this metric is not usable as an objective function. Many individuals with vastly different orderings have the same fitness value using this metric, preventing the genetic algorithm from distinguishing them and exploiting the building blocks. We examine two other objective functions, described in Section 3.1, which both use the pairwise-overlap information as the basis for evaluating the fitness of the layout.

2.1. Fragment Assembly and the Traveling Salesman Problem

The relationship between the ordering step and the general class of permutation ordering problems is clear. Probably the best known problem in this class is the Traveling Salesman Problem (TSP) [(Lawler, Rinnooy Kan, & Shmoys 1985)], but there are many others. The fragment assembly problem is quite similar to TSP, with notable differences. First, the solution to TSP is a circular tour of the cities; the endpoints of the tour are therefore irrelevant. In the fragment assembly problem, however, the endpoints represent fragments on opposite ends of the parent sequence. Many solutions which are equivalent for TSP are thus inequivalent in our context. Second, the cities in the TSP are not assumed to have any relationship other than the distances, and the ordering is the final solution to the problem. In the fragment assembly problem, the ordering, referred to as "beads on a string", is only an intermediate step in the solution process; the layout process uses the overlap data to position the bases within the fragments relative to each other. Because there are frequently more than two fragments overlapping each other in the layout, several different orderings of those fragments produce equivalent results after the layout phase, as shown in Figure 3. Additionally, many algorithms for TSP rely on the triangle inequality holding for the distance relation; no such assumption can be made about the overlap strengths. The errors in the overlap strength computation due to sequencing errors, chimeric fragments and DNA repeat sequences tend to invalidate any simplifying assumptions made about the relationships between fragments. Another distinguishing feature of the fragment assembly problem is that the fragments are drawn from both strands of the DNA, and the orientation of the fragment relative to the parent is lost during the sequencing process.

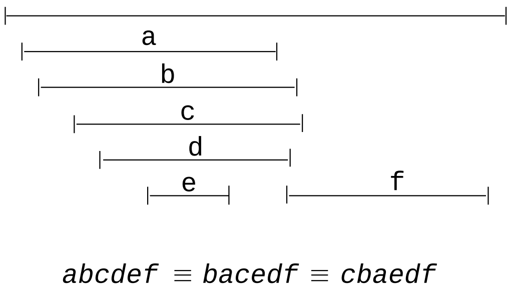


Figure 3. Different Fragment Orders Can Produce Equivalent Consensus Sequence

Genetic algorithms for permutation problems have not been universally successful, but there are successful examples. The primary problem faced by genetic algorithms in this context is representing the solutions in some way that allows the genetic operators to produce legal solutions. The simple representation for a solution is the permutation, represented as a list of the fragments (labeled with unique numbers) in the order in which they should appear. However, the standard genetic operators are then not closed over the space of legal solutions. As the space of illegal solutions is quite large, the probability of the operator forming a legal solution is relatively small.

There are three obvious approaches to this representation problem: (1) choose a representation such that the standard operators are closed over legal solutions; (2) choose specialized operators that guarantee legal solutions; (3) penalize illegal solutions in the fitness function. We chose not to explore the third approach. The search space is so sparsely populated with legal solutions that too much time would be wasted generating and recognizing illegal solutions. In addition, the fitness functions are already costly to compute, and it is not clear what kind of penalty function to apply or whether legal blocks within these individuals would be useful building blocks. We studied the remaining two approaches, first trying the sorted-order representation, in which all solutions map to a legal permutation order [(Syswerda 1989); (Bean 1992)]. We then studied the performance of the simple permutation representation in combination with two special-purpose recombination operators, edge-recombination [(Starkweather *et al.* 1991)] and order crossover [(Davis 1985)]. In addition to recombination, we explore both bit and position mutation for the sorted-order representation, and for the permutation representation, we study position swaps, block moves (transpositions) and block inversions. These operators and representations are described in the next section, along with the data sets used for testing.

3. Genetic Algorithms Applied to Fragment Assembly

Genetic algorithms operate on a population of candidate solutions, called individuals [(Holland 1975); (Goldberg 1989); (Forrest 1993)]. Typically, the population is initialized with random individuals. After that, individuals are deleted from or reproduced in the population on the basis of their relative fitness. New individuals are formed by applying various operators to the existing population of individuals (see below). Each successive population of individuals is called a generation. Processing within the genetic algorithm typically follows these steps:

Selection: Each individual is evaluated to determine its fitness. Individuals are reproduced (copied) differentially based on this fitness. Different genetic algorithms use different methods to implement the idea of differential reproduction. In the "generational GA," which we used, a new population is created at each generation, completely replacing the previous population. Individuals with below average fitness for the population have a low probability of being copied into the next generation, while individuals with high fitness have a high probability of having multiple copies in the next generation. [4]

Crossover: Two individuals are selected from the population and substrings from corresponding positions within the individuals are exchanged. One or both of the new individuals are inserted into the population at the next generation. The purpose of this operator is to allow partial solutions to evolve on different individuals, and then to combine them to produce, sometimes, a better solution. For all of the crossover operators described in this report, the crossover rate specifies, on average, what percentage of new individuals are formed in a generation through crossover.

Mutation: Mutation alters one individual by changing a primitive element of that individual (e.g., flipping one bit). The mutation rate controls the probability with which each component in an individual is changed. The resulting individual replaces the parent of the mutation. Mutation is believed to be effective for two reasons: it explores the search space near existing individuals (local search) and it prevents solution components which have been completely eliminated from the population by selection (fixation) from being lost for all successive generations.

The genetic algorithm begins with a random population, and the cycle of selection, crossover, and mutation is repeated for many generations. The genetic algorithm used to produce the results reported here was implemented by modifying the Genesis [(Grefenstette 1984)] software package.

Although there is some controversy over how well genetic algorithms actually perform and about why they perform as well as they do, the most common explanation is that the average fitness of the population is likely to increase in successive generations, as good partial solutions combine to form even better composite solutions. The process of creating good solutions is described in terms of the combination of *building blocks*, which are portions of solutions that have high fitness. In the context of the fragment assembly problem, a building block is a portion of the ordering representing several fragments that form a contig. When considering different representations for genetic algorithms, it is important to consider what the likely building blocks are and how they might be combined to form a complete solution.

3.1. Fitness Functions

The choice of fitness function is crucial to the success of a genetic algorithm on a particular problem. In the fragment assembly problem the choice of an efficient and reliable measure of fitness is complicated by several factors, including errors in the sequence information (insertions, deletions, etc.), repeated sequences, and the "beads on a string" model. We studied two closely related fitness functions. The first is a natural analog to the fitness function for TSP. Let $I = f[0], f[1], ..., f[n-1]$ be an ordering of fragments, where $f[i] = j$ denotes that fragment j appears in position i of the ordering. The fitness of the individual under the first fitness function `F1` is

$$\texttt{F1}(I) = \sum_{i=0}^{n-2} w_{f[i],f[i+1]}$$

where $w_{i,j}$ is the pairwise overlap strength of fragments i and j. This fitness function only examines the overlap strengths of directly adjacent fragments in the ordering and takes $O(n)$ time to compute for each individual. The optimization process attempts to find a layout that maximizes this function.

The second fitness function accounts for the additional information that is exploited in the subsequent passes of the assembly processing [(Churchill *et al.* 1993)]. In addition to examining the overlap strengths for adjacent fragments, it considers the overlap strengths among all pairs of fragments, penalizing layouts in which fragments with strong overlap are far apart. The specific fitness function `F2`

$$\texttt{F2}(I) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} |i-j| * w_{f[i],f[j]}$$

uses the absolute value of the distance of the fragments in the layout as a weight on the overlap strength of the pair of fragments in those layout positions. The optimization process searches for layouts that minimize this function. This function has complexity $O(n^2)$ because all pairs of fragments must be considered.

3.2. Representations

Genetic algorithms are appealing because they are largely domain independent—the problem specifics are isolated in the fitness function and in the mapping from the individual to the problem-level solution. From this perspective, the first solution to the problem of representing permutation problems—a representation guaranteeing legal solutions—is more appealing than the second. Using this type of representation provides the isolation of problem-specific information from which the genetic algorithm derives its generality. For this reason, we chose to try the sorted-order representation [(Schaffer *et al.* 1989); (Syswerda 1989)], also referred to as the random-key representation [(Bean 1992)].

The sorted-order representation provides a rather complex mapping from the individual to the permutation ordering. The two requisite properties for a legal ordering are that all fragments be present in the ordering and that there be no duplication in the ordering. These properties are ensured through the use of a sort (hence the name "sorted-order"). Specifically, consider a fragment set $f_1, \ldots, f_n$ and an individual $B = b_1, \ldots, b_m$ where each b_i is a bit, and $m = 2^k \geq n$. To find the permutation specified by B, first convert the bit string into n key values, $k_1, \ldots, k_n$, each of k bits, and then sort the key values. If the key value in position j of the individual appears in position i of the sorted list, then fragment f_j is in the *ith* position in the permutation specified by the individual B. [5] Because a fragment (f_j) is identified by a position in the individual (j), and a fragment is placed in the

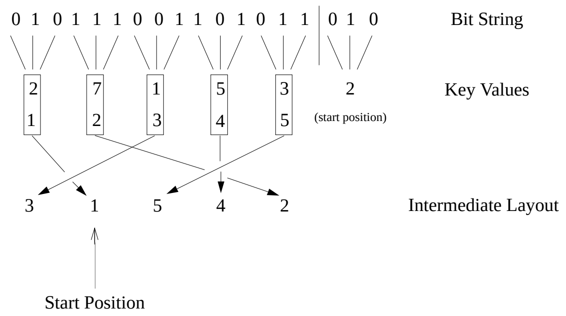


Figure 4. Sorted-order Representation for the Fragment Assembly Problem. Consider a bit string which produces the following integers ($k = 3$): 2 7 1 5 3 2. The fragment layout represented by this individual is 1 5 4 2 3 with an intermediate (before shifting) ordering of 3 1 5 4 2. Because the lowest key value in the individual, 1, appears in the third position of the individual, the first fragment in the intermediate layout is 3. The next lowest value, 2, is in the first position of the individual, and therefore, the second fragment in this layout is 1. The last key, which represents the starting position, is 2, so the first key in the permutation ordering is 1 (because 1 appears in the second position of the intermediate layout). The final layout continues from this position and then wraps to the beginning.

permutation based on the position in the sort order (i) of its key (j), any bit string represents a legal permutation.

To represent the fragment assembly problem more closely (as opposed to TSP, with its circular solution), we introduced a modification to the sorted-order representation. In each individual, add k bits to the end, which designates the starting point of the permutation order. This value does not participate in the sort or in the mapping described above; instead, it allows the shifting of the ordering to allow alternate starting positions. Figure 4 illustrates the mapping.

We also studied the straightforward permutation representation, together with a suite of specialized operators. In this representation, the bit string is again $m = 2^k \geq n$ long. Each k bits represents one fragment (labeled 0 to $2^k - 1$); the position of a fragment in the individual designates the position that fragment occupies in the layout. Although this representation vastly simplifies the mapping from bit strings to permutations, the operators must be more complex to ensure that only legal individuals enter the population.

3.3. Recombination operators

We used both uniform and two-point crossover methods with the sorted-order representation. We found that uniform crossover did not perform well on our problem, so we report only results based on the more standard two-point crossover, in which two points are randomly selected, and the bits between those positions are exchanged [(Goldberg 1989)]. Because the sorted-order representation is closed under the standard genetic operators, no additional processing is required to find a legal permutation.

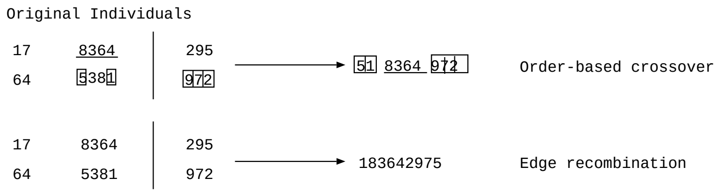
Two special-purpose crossover operators that have been successful in permutation problems are edge recombination [(Starkweather *et al.* 1991)] and order crossover [(Davis 1985)]. Different crossover operators emphasize the preservation of different kinds of information from the parents. Thus, the success of different operators for different permutation problems is likely tied to the ability of the crossover operator to preserve the valued information from the parents under crossover. There are at least three kinds of information that might be important in a permutation ordering: absolute position, relative ordering (e.g., precedence relations as in scheduling applications), and adjacency. In the TSP, particularly due to its circular nature, the only relevant information is probably adjacency information. For the fragment assembly problem, the issue is less clear. Adjacency information in the total ordering is important. However, given the linear nature of the layout, with the definite end points, and the overlapping nature of the fragments in the ordering, relative position may also be important. We chose to experiment with two crossover operators, order crossover and edge recombination. Order crossover preserves relative ordering (and in some cases absolute position and ordering), while edge recombination emphasizes adjacency information. For simplicity of explanation, these crossover operators will be described in terms of integer, not bit values.

In order crossover, as in two-point crossover, two random points are selected. However, instead of exchanging the information between these points, the information from the first parent is copied into those same positions in the offspring, as shown in Figure 5. Then, starting from the beginning of the second parent and the beginning of the offspring, the fragments from the second parent are placed into the offspring, with any fragment already placed in the offspring being skipped in the second parent. Thus, the fragments from the first parent retain their position and relative ordering while the fragments from the second parent retain their relative ordering.

Edge recombination is a more complicated operator. A detailed explanation of the operator implemented here appears in [(Starkweather *et al.* 1991)]. In general, this crossover attempts to preserve adjacencies in the parents, and in particular, those adjacencies that are common to both parents. When neither of those options is possible, a random selection made. An example of how the edge-recombination operator works is shown in Figure 5. Whitley reported additional modifications to the edge-recombination operator, which give even better results on TSP [(Whitley 1993)], but we have not yet tested these modifications.

3.4. Other Operators

The sorted-order representation allows the use of the simple bit-mutation operator [(Goldberg 1989)]. With some (small) probability, each bit in each individual is altered. However, in the permutation representation, point mutation is guaranteed to produce an illegal solution. Consequently, we did not use point mutation with



Adjacency list
(for edge-recombination)

Key	Adjacent Keys
1	7,8,9
2	4,9,7
3	8,6,5,8
4	6,2,6,5
5	9,4,3
6	3,4,4
7	1,8,9,2
8	7,3,3,1
9	2,5,1,7

Figure 5. Order Crossover and Edge Recombination. Illustrates order crossover on the individuals 1 7 8 3 6 4 2 9 5 and 6 4 5 3 8 1 9 7 2. For edge recombination, The offspring begins with the first fragment in the first parent, 1. In examining 1's adjacencies, we select 8 because it has a shared adjacency in its list and the others do not. This shared adjacency is 3, and so it is placed next. Because it has a shared adjacency, 6 is chosen next, followed by 4. At 4, the next fragment 2 is placed in the individual because it has more remaining adjacencies. This process continues until the end, where 5 is placed because it has not yet been placed, yielding the individual 1 8 3 6 4 2 9 7.

the permutation representation. However, we tried a suite of other mutation-like operators.

The simplest of these is the swap. Two positions in the ordering are selected at random, and the fragments in those positions are swapped to create the new individual [(Churchill *et al.* 1993)]. This operator is a restricted form of a 4-opt transformation [(Lin & Kernighan 1973)], using TSP terminology.

We used two other operators, each of which rely on some domain-specific information. These two operators, inversion and transposition, move blocks of fragments, specifically contigs, in the ordering. Although random locations are selected in the individual for these operators, a location simply indicates to which contig(s) the operation will be applied. Inversion reverses the order of the fragments in the selected contig [(Goldberg 1989)]. This operator is useful since fragments come from both strands of DNA, but the total ordering forces an orientation to the data. By inverting a contig, it may be extended, since the contig may be oriented in the opposite direction from that of the adjacent fragments in the ordering. We restrict inversion to operate only on contigs within the layout, instead of the more general inversion operator which randomly selects an area to invert.

Transposition moves a contig to a position between two adjacent contigs which may allow the extension of a contig. This operator allows smaller contigs to form anywhere along the individual; transpositions can correct the relative positions of these clusters, yielding an improved solution. Transposition is also a restricted form of 4-opt, with the restriction focusing on the selection of the edges to break and with what edges to replace them.

Contigs are the natural building blocks of our problem. Transposition and inversion, by design, do not disrupt these building blocks. Instead, they allow for the evolution of larger contigs by treating the smaller contigs as primitive elements.

The meaning of the rates for the mutation and specialized operators are different. For the mutation used with the sorted-order representation, the rate specifies the probability of a bit in an individual being selected for mutation in a given generation. Thus, to find the probability that an individual is altered, this rate must be multiplied by the length of the individual. For the specialized operator, the rate quoted is the probability that an individual will be affected. Thus, a rate of 0.2 implies that there is a 20% probability for each individual that it will be selected for alteration by that operator.

3.5. Data Sets and Implementation Environment

Most of the fragment data sets for which we report results were artificially generated from actual DNA sequences. The parent DNA sequence is replicated by the program and then fragmented, at random locations, to derive a fragment set that is representative of the data sets produced in sequencing laboratories. The generator, GenFrag [(Engle & Burks 1993)], allows fragment sets of different sizes, error content, and coverage to be generated. This approach allowed us to study how these different factors affect the performance of the genetic algorithm. This baseline testing proved useful in identifying particularly successful and unsuccessful configurations of representations, operators, population sizes, etc. Having completed these control experiments, we are beginning to tackle experimentally derived DNA fragment sets.

Three DNA sequences served as the basis for most of the experiments: a human brain DNA sequence, HUMATPK01 [(Sverdlov *et al.* 1987)], accession number M55090 which is 2026 base pairs long; a human MHC class III region DNA with fibronectin type-II repeats HUMMHCFIB [(Matsumoto *et al.* 1991)], with accession number X60189, which is 3835 bases long; and a human apolopoprotein HUMAPOBF [(Carlsson *et al.* 1986)], with accession number M15421 which is 10089 bases long. We are also working with two longer sequences. The 20KB sequence, AMCG, is the initial 40% of the bases from LAMCG, the complete genome of bacteriophage lambda, accession numbers J02459 and M17233 [(Sanger *et al.* 1982)]. The data set designated SETO is the experimental data set made available for the testing of sequencing algorithms [(Seto, Koop, & Hood 1993)].

Most of our results were obtained using fragment sets from the first three parent sequences. We experimented with coverages ranging from 3 to 7 and mean fragment length between 300 and 500 bases. In addition, fragment sets with experimental errors at a rate of 10% mismatch errors and 5% insertion and deletion errors have been used to determine how robust the algorithm is. Table 1 presents some information about the specific fragment sets on which we tested our algorithm.

Table 1. Information on Data Sets: Names: ATPK - HUMATPK01, CFIB - HUMMHCFIB, POBF - HUMAPOBF data set, AMCG - LAMCG and and SETO - experimental data set. Bases: number of base pairs in the known consensus sequence. Coverage: the average number of fragments covering any base of the parent. Gaps: areas of the parent with no coverage. CFIB-5% and CFIB-10% have errors introduced into the fragment sets at the specified rate.

							5%	10%				
	ATPK		CFIB				CFIB	CFIB	POBF		AMCG	SETO
Bases	2026		3835				3835	3835	10089		20100	34475
Coverage	5	7	3	4	5	7	7	5	5	7	7	11
Fragments	26	36	24	39	48	68	68	48	127	177	352	829
Gaps	0	0	1	0	0	0	1	0	1	0	0	0

The overlap computation uses a technique that allows the setting of a cutoff value, such that overlap scores below this value are considered to be zero [(Churchill *et al.* 1993)] The cutoff, which we have set to twenty, provides one filter for spurious overlaps introduced by experimental error.

4. Results

As described earlier, it is difficult to identify a single best measure of performance for this problem. In the case where the consensus sequence for the data is known, the correctness of an ordering can be determined by performing the final steps of the assembly process and comparing the resulting sequence with the known consensus sequence; the goal is a complete match. It is also important to consider the time to reach a solution. Table 2 summarizes the results using the match with the parents, the number of contigs in the solution and the number of function evaluations and generations used by the genetic algorithm. The remainder of this section explores the overall performance achieved by the genetic algorithm and the experiments we performed to analyze the different components of the genetic algorithm: the fitness function, the representation, and the operators. The final section draws conclusions from these data and discusses future directions for research.

The genetic algorithm performs quite well with the appropriate representation and operator set, both in terms of speed and solution quality. Figure 6 shows how the mean and best fitness improves for the CFIB data set. The genetic algorithm solves the smaller two data sets, finding a single-contig solution representing a consensus sequence that completely matches the parent sequence. This result, however, is not all that startling as the fragment sets are relatively small. Most fragment assembly systems can, for data sets of this size with no errors, produce results that are close enough to correct that a small amount of manual intervention produces the correct

Table 2. Genetic Algorithm Performance on Fragment Assembly Problem. Data set name includes coverage in parentheses. Num Gens: Number of Generations, Num Trials: Number of Fitness Function Evaluations (Gen*Pop > Trials since not all individuals change in a generation), Num Contigs: Number of contigs in GA solution, Num Greedy: Number of Contigs in the Greedy Solution, Num Sorted: previous results using sorted order representation (* indicates that this experiment was not performed on sorted order). Other parameter settings: Crossover Rate: 0.3, Specialized Operator Rate: 0.7, Pointswap rate: 0.2, Inversion Rate: 0.4, Transposition Rate: 0.4, Scaling Factor: 2.0. Parent match computed for single contig solutions and computed over entire region of parent.

Data Set Name	Pop Size	Num Gens	Num Trials	Num Contigs	Parent Match	Num Greedy	Num Sorted
ATPK(5)	100	135	4K	1	1.0	3	*
ATPK(7)	500	64	10K	1	1.0	3	*
CFIB(3)	200	167	10K	2		3	4
CFIB(4)	600	193	35K	1	1.0	4	*
CFIB(5)	600	304	55K	1	1.0	3	5
CFIB(5)-10%	600	668	120K	1	.97	6	*
CFIB(7)	500	1200	180K	1	1.0	3	7
CFIB(7)-5%	500	5635	855K	2		5	7
POBF(5)	1000	19K	5.7M	6		10	23
POBF(7)	1500	13K	5.9M	1	1.0	7	43
AMCG(7)	2500	5.6K	2.3M	13		15	*
SETO(11)	2500	137	302K	89		65	*

solution. Even for the smaller data sets, however, the correct representation is crucial to the performance of the genetic algorithm. By using a representation and a suite of operators that exploit the conceptual building blocks of the problem, we were able to construct a genetic algorithm with good performance on our problem.

The results for the POBF data set, with the 10KB parent, are more interesting than those for the smaller data sets. There are 177 fragments in the seven-fold coverage data set. With no manual intervention, the genetic algorithm produces a single-contig solution whose consensus sequence completely matches the parent. We have yet to solve fully the remaining two large data sets, the AMCG data set with a 20KB parent, and the Seto data set, with the consensus sequence of 34KB. Table 2 presents the results on those data sets to date.

The most exhaustive testing involved the CFIB data set, as seen in Table 2. Our fragment sets included some with significant amounts of error; some of the data sets contained a gap. The GA found the correct single-contig solution for all the data sets without gaps and errors. In the case of the data set with 10% mismatch errors, the GA finds a single-contig solution with a 97% match to the parent. Although there were mis-match errors in 10% of the base pairs in the fragment set, the five-fold coverage and the post-processing of the multiple sequence alignment and consensus generation algorithms correctly reproduced 97% of the bases. For the

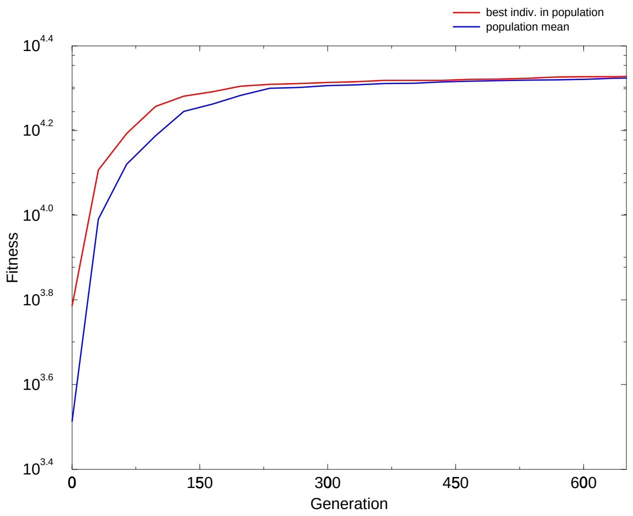


Figure 6. Population mean and best fitness plots. Typical run for CFIB(7) data set.

CFIB data set with the gap but with no errors, the solution found, while existing of two contigs, matched on all the base pairs covered in the solution.

4.1. Comparison to a Greedy Approach

Many of the standard assembly packages use some form of greedy algorithm to find the appropriate ordering [(Staden 1980); (Huang 1992)]. To evaluate the performance of the genetic algorithm, we compare our results to those found by a greedy algorithm. The greedy algorithm we use, which is similar to that used in [(Huang 1992)], examines overlap strengths and picks the strongest overlap that connects some as yet unplaced fragment to the contig being constructed. Table 2 includes the solutions found by our greedy algorithm. The greedy algorithm never uncovers the best solution, except for the small CFIB data set with the gap, where it does find the optimal solution.

For the POBF data set, we looked at the solution found by the greedy algorithm. The fitness values for this data set are shown in Table 3. The results from the F2 function, which penalizes solutions that ignore significant overlap, is particularly informative. The large difference between the F2 scores for the greedy solution and the genetic algorithm solution provides an indication of how much manual editing would be required to convert the greedy solution into a workable solution. The work is even greater than the difference between 1 and 7 contigs initially indicates.

Table 3. Comparison of Greedy Algorithm and Genetic Algorithm on POBF Seven-Fold Coverage Data Set.

	F1 Score	F2 Score	Contigs
Greedy	54,049	6,940,730	7
GA	55,683	1,705,272	1

4.2. Fitness Functions

A critical part of the design of any genetic algorithm is the selection of the appropriate fitness function. For fragment assembly, this selection is complicated by the processing required after the fragment-ordering step. In previous work [(Parsons, Forrest, & Burks 1993)], we compare in detail the two fitness functions described in Section 3.1. These results are reproduced in Table 4. The experiments with the sorted-order representation showed the quadratic fitness function, F2, to be marginally superior in performance to the linear function, F1. We ran a small set of experiments on these two functions using the permutation representation and the edge-recombination operator. In this case the linear function performs better. The results appear in Table 4 for the ATPK data set. Although it can be argued that this objective function is not ideal for the fragment assembly problem, the computational results show that the function is adequate, when compared with F2 and based on the results we have achieved so far. Nevertheless, we consider the design of a more appropriate fitness function to be an important area for continued research.

4.3. Comparison of the Representations

The results of our experiments show that, for our application, the edge-recombination operator in conjunction with the permutation representation and the full suite of other operators is quite successful. The final column in Table 2, reproduced from [(Parsons, Forrest, & Burks 1993)], gives the results for the sorted-order representation. Summarizing our prior results, the sorted-order representation is too disruptive of the building blocks of the fragment assembly problem to be useful in this context. The permutation representation with the edge-recombination operators and the other specialized operators are able to exploit the building blocks in a powerful way.

4.4. Comparison of Operators Using the Permutation Representation

This section describes several experiments we performed to analyze the effectiveness of the various operators and to examine the way in which they interact with each

Table 4. GA Test Results for Two Fitness Functions. Best: score for the best individual, followed by the score for this individual under the other function. Contigs: the number of contigs in the layout specified by the best individual. For data set ATPK with coverage 7. Parameters: 10 runs, different random seeds, Crossover rate of 0.85, 0.7 point swap, 0.03 transposition, 0.27 inversion.

	F1 (O(N), Maximize) Function			F2 (O(N^2), Minimize) Function		
Parent	Best F1	F2 Score	Contigs	Best F2	F1 Score	Contigs
CFIB(5)	13,900	570,627	5	422,049	13,304	4
CFIB(3)	7,534	488,198	4	73,755	7539	3
CFIB(6)	15,809	1,936,597	11	852,087	13,274	8
CFIB(7)	19,513	2,679,743	7	1,243,338	20,163	3
CFIB(5)-10%	3,873	162,581	7	100,179	3,333	8
POBF(5)	31,372	4,773,694	23	1,331,745	24,040	23
POBF(3)	17,989	551,698	13	179,950	18,092	7
POBF(6)	27,029	19,224,270	48	5,776,555	8,305	36
POBF(7)	33,823	22,285,703	43	7,128,031	13,582	28
ATPK(7)	11,284	286,118	1	355,478	9,671	2

other. First, we compared the two crossover operators: order crossover and edge recombination, described in Section 3.3. Table 5 summarizes the results obtained using order crossover. Comparing the results from Table 5 with those in Table 2 clearly demonstrates that, even attempting to tune the genetic algorithm parameters for this representation, edge-recombination is superior to order crossover.

Table 5. Genetic Algorithm Performance Using the Order Crossover Operator. Trials: Number of `F1` Fitness Function Evaluations. Num Contigs: Number of Contigs found in the Best Solution Over 5 Runs.

Population Size	Num Trials	Xover Rate	Swap Rate	Inversion Rate	Transposition Rate	Num Contigs
600	100K	0.3	0.14	0.28	0.28	6
600	200K	0.3	0.14	0.28	0.28	6
600	100K	0.6	0.14	0.28	0.28	9
600	100K	0.8	0.06	0.12	0.12	12
1000	200K	0.8	0.02	0.04	0.04	15
1000	400K	0.8	0.02	0.04	0.04	13
300	400K	0.8	0.02	0.04	0.04	7
300	400K	0.5	0.14	0.28	0.28	8

Next, we ran several experiments applying the operators at various rates. Some of the particular results are worth noting. The genetic algorithm was relatively successful in solving the smaller data sets with traditional parameter settings: (crossover rate of 0.7-0.85 and rates for the specialized operators of 0.05-0.1). However, significantly better performance occurs at much lower crossover rates and much higher rates for the specialized operators. These rates increased the efficiency of the genetic algorithm on the small data set and allowed the genetic algorithm to solve the larger data set. Most of the runs reported here used a crossover rate of 0.2-0.3 and a special operator rate of 0.7-0.85.

Eliminating any one of the operators (either crossover or one of the specialized operators) tended to give poorer results. There is a synergistic effect among the various operators in finding appropriate solutions. The results of these experiments are summarized in Table 6. We tracked the change in the fitness values for each application of each operator over the course of several runs. Figure 7 shows the efficacy of the different operators over the life of twenty different runs. A low-pass filter was applied to smooth out the variations in the data. The graph shows that the utility of pointwise swap decreases as the run progresses, and inversion and transposition remain useful throughout.

Table 6. GA performace for CFIB data set with five-fold coverage, population of 300 and 250,000 trials. * This run completely converged at 50,000 iterations.

Xover	Swap	Trans	Inv	Contigs
0.1	0.14	0.28	0.28	1
0.2	0.14	0.28	0.28	2
0.3	0.14	0.28	0.28	1
0.4	0.14	0.28	0.28	2
0.5	0.14	0.28	0.28	3
0.0	0.3	0.28	0.28	4
0.0	0.3	0.56	0.14	1
0.85	0.0	0.0	0.0	4
0.0	0.0	0.45	0.45	6
0.0	0.1	0.75	0.0	5
0.0	0.1	0.0	0.75	9*

We computed the genealogy of the best individuals in the population to determine which operators contribute to the creation of good individuals and if the mix changes as the population changes. Specifically, every 100 generations, we examined the history of the 5 best individuals and recorded how often each operator occurred in its history during the last 100 generations. Crossover is consistently contributing to the creation of the best individuals, over the course of the run.

5. Discussion and Conclusions

One of the interesting questions raised by this work relates to the way in which the specialized operators exploit the building blocks in our problem and the way

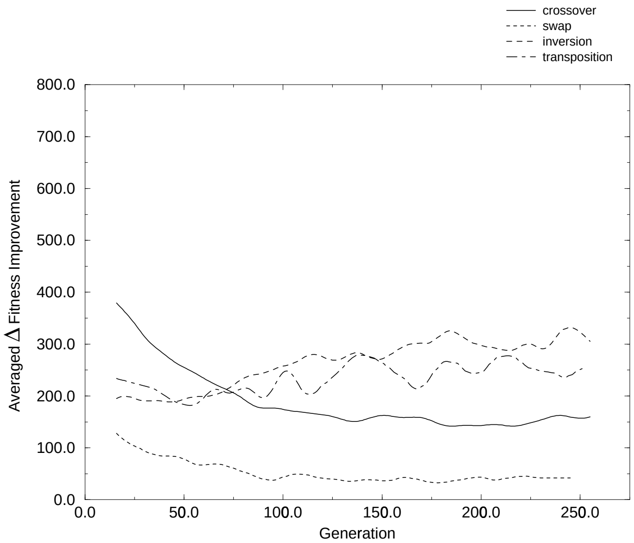


Figure 7. Average change in fitness by operator, smoothed using a low pass filter. Only operations which improved fitness considered.

the full operator suite interacts to improve the genetic algorithm's performance. Performance suffers when any operator is removed from the suite, although a high enough rate of transposition reduces the impact of the loss of the crossover operator. However, if one examines the mechanics of the crossover operator between two individuals that are similar, this result is not that surprising as transposition mimics this kind of crossover operation.

The genealogy for the best individuals, shown in Figure 8, highlights the importance of the crossover operator in the creation of good solutions, even at the low rate at which it is applied. As described earlier, this low rate of crossover was the best we found while tuning the genetic algorithm parameters. One should not conclude, however, that this low rate indicates that crossover is unimportant to the solution power; the genealogy information tells a very different story. Although the average fitness improvement of the operator declines during the run, it remains an active participant in the creation of improved individuals.

The behavior of crossover, transposition and inversion change during the run, while that of the swap operator remains the same. Transposition and inversion affect contigs. Early in the run, contigs are likely to be quite small and frequently contain only one fragment. Thus, in these early generations, the specialized operators act similarly. The decrease of effectiveness of pointwise swap in later generations is consistent with the building block hypothesis. Late in the run, the changes required to improve an individual tend to be on a larger scale than can be obtained by a single swap, i.e. improvements tend to require the movement of contigs, the conceptual building blocks in this problem, rather than movement of individual fragments.

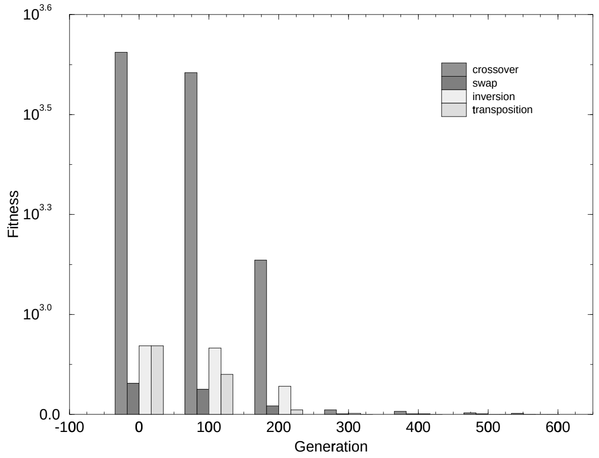


Figure 8. Operator frequency in 100 generation intervals for five best individuals in population for the CFIB seven-fold data set. The operators are displayed in the order in which they appear in the legend.

For many genetic algorithms, a significant problem is premature convergence. Our operator suite tends to prevent this convergence. After the 668 generations of the CFIB seven-fold coverage run, the population was not close to being converged in the traditional sense. Although almost all of the positions in this data set had 50% or more of the individuals with the same fragment in that position, none of those positions had more than 60% of the individuals with the same fragment value. Thus, the GA is still searching for improved solutions. Two factors contribute significantly to this effect. First, in the permutation representation, the alphabet at each position is much larger than the binary alphabet traditionally used in genetic algorithms. Second, and more importantly, the operators used in this genetic algorithm tend to be quite disruptive in terms of the values in a particular position towards the end of the run. Consider an inversion operation, which inverts a contig. An entire block of positions within the individual is changed by this operation, and the operator, at worst, creates an individual with equal fitness to itself. Towards the end of the run when convergence is an issue, the contigs are quite large, meaning many fragment positions are altered. The effects of the operators, coupled with the high rates at which they are applied, counterbalances the strong convergence seen in most genetic algorithm applications.

Some features of the fragment assembly process aid the performance of the genetic algorithm. The results shown in Table 2 for the POBF data set are from runs with the same number of iterations, which represents one-third fewer generations for the larger data set when the size of the population is taken into account. One would expect that increasing the problem by size 35% from the perspective of the optimizer — it must now order 177 fragments instead of 127 — would make the problem more difficult. However, for the 177 fragment problem, the genetic algorithm in that time finds a one-contig solution while the solution for the 127 fragment problem appears stuck at a six-contig solution. With the increase in coverage from five-fold to seven-fold, there is a corresponding increase in the number of essentially equivalent solutions, and the precise ordering becomes less critical. This characteristic of the search space for the higher-coverage problem is another indication that a genetic algorithm is an appropriate approach, because genetic algorithms tend to find near-optimal solutions relatively easily, but have difficulty refining those solutions.

A high degree of redundancy in the search space is also present in the sorted-order representation, as many different combinations of numbers would map to the same permutation ordering. However, the redundancy in the search space is not sufficient to overcome the disruption of the solution building blocks caused by the operators (see [(Parsons, Forrest, & Burks 1993)] for further details).

The work so far demonstrates the feasibility of using genetic algorithms in sequencing problems when the parent sequence is on the order of 10KB and when coverage is sufficient (seven-fold). Since most sequencing labs use coverages at this level or higher, the coverage range is not a restriction. We are still working with parents in the range of 20KB - 35KB. In this range, however, problems in the other phases of the analysis, on which the ordering depends, become more pronounced, in addition to the dramatic increase in the size of the search space. The most obvious problem is that of DNA repeats. Longer parent sequences tend to have more repeat sequences. The regions of these repeats will have high overlap, since the repeat sequences are generally quite similar (> 90% homology is not uncommon) and can be lengthy. Thus, fragments from different sections of the DNA that have repeats will show overlaps that are, from examination of the sequence and overlap information, indistinguishable from overlaps that result from fragments being drawn from the same section of DNA using only overlap information. Additional information must be provided to any program attempting to sequence DNA with these repeated segments to allow repeat-induced overlaps to be distinguished from true overlaps. Some auxiliary data, such as mapping information, is available, but the objective function will have to be redesigned to take this new information into account.

The success of the specialized operators and representations which exploit the conceptual building blocks may influence the solution of permutation problems using techniques other than genetic algorithms. As an example, Burks *et al.* [(Burks *et al.* 1993)] have successfully incorporated the inversion and transposition operators into a simulated annealer with impressive performance improvements. More generally, there are many other interesting questions raised by these experiments, particularly relating to the role of the solution space redundancy and to the synergistic effects among the various operators.

Acknowledgments

The authors wish to thank L. Davis, M. Engle, R. Hightower, D. Mathews, J. Sims, C. Soderlund, P. Stolorz, and D. Whitley for their assistance on this project. This project was initiated during a map assembly workshop at the Santa Fe Institute sponsored by the Santa Fe Institute and the Theoretical Division of Los Alamos National Laboratory. This work was performed in part under the auspices of the United States Department of Energy under contract #W-7405-ENG-36. C.B. was supported by the DOE/OHER Genome Project (ERW-F137; R. Moyzis, P.I.); R.P. was supported in part by a Los Alamos Director's Fellowship; and S.F. was supported by National Science Foundation (grant IRI-9157644), and Sandia University Research Program (grant AE-1679).

Notes

1. NP-hardness follows from a straightforward reduction from Hamiltonian Path.
2. DNA is a double helix comprised of two complementary strands of polynucleotides. Each nucleotide consists of a purine or pyrimidine base attached to a sugar-phosphate moiety. The sugar-phosphate is constant throughout the entire strand, but the bases vary. There are four bases found in DNA: adenine (A), guanine (G), cytosine (C), and thymine (T). From a computational viewpoint, each strand of DNA can be viewed as a character string over an alphabet of four letters. The two strands are complementary in the sense that at corresponding positions A's are always paired with T's and C's with G's, although any of the letters can appear in either strand. These pairs of complementary bases are referred to as "base pairs."
3. There are different families of repeat sequences, each with different characteristic lengths and degree of conservation among the family members. Some repeat sequences arise due to duplicated genes, as an example.
4. We used an elitist policy and sigma scaling with a cutoff value of 2.
5. Ties in the sort are broken arbitrarily. We use a left to right ordering.

References

Bean, J. C. 1992. Genetics and random keys for sequencing and optimization. Technical Report 92-43, The University of Michigan.

Burks, C.; Engle, M.; Lowenstein, M.; Parsons, R.; and Soderlund, C. 1993. Stochastic optimization tools for DNA assembly: integration of physical map and sequence data. Poster presented at Genome Sequencing and Analysis Conference V.

Burks, C.; Engle, M.; Forrest, S.; Parsons, R.; Soderlund, C.; and Stolorz, P. 1994. Stochastic optimization tools for genomic sequence assembly. In Venter, J. C., ed., *Automated DNA Sequencing and Analysis Techniques*. Academic Press.

Carlsson, P.; Darnfors, C.; Olofsson, S.-O.; and Bjursell, G. 1986. Analysis of the human apolipoprotein B gene; complete structure of the B-74 region. *Gene* 49:29–51.

Cedeno, W., and Vemuri, V. 1993. An investigation of DNA mapping with genetic algorithms: preliminary results. In *Proc. of the Fifth Workshop on Neural Networks*, volume 2204 of *SPIE*.

Chen, W. Q., and Hunkapiller, T. 1992. Sequence accuracy of large DNA sequencing projects. *J. DNA Seq. Map* 2:335–342.

Churchill, G.; Burks, C.; Eggert, M.; Engle, M.; and Waterman, M. 1993. Assembling DNA sequence fragments by shuffling and simulated annealing. Technical Report LAUR 93-2287, Los Alamos National Lab., Los Alamos, NM.

Davis, L. 1985. Applying adaptive algorithms to epistatic domains. In *Proc. of the 1985 Joint Conference on Artificial Intelligence*.

Engle, M., and Burks, C. 1993. Artificially generated data sets for testing DNA fragment assembly algorithms. *Genomics* 286–288.

Fickett, J., and Cinkosky, M. 1993. A genetic algorithm for assembling chromosome physical maps. In Cantor, C. R., and Robbins, R., eds., *Proc. of the Second International Conference on Bioinformatics, Supercomputing, and Complex Genome Analysis.* World Scientific. 272–285.

Forrest, S. 1993. Genetic algorithms: Principles of natural selection applied to computation. *Science* 261:872–878.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley Publishing Company.

Grefenstette, J. J. 1984. Genesis: A system for using genetic search procedures. In *Proc. of a Conference on Intelligent Systems and Machines*, 161–165.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: The University of Michigan Press.

Howe, C., and Ward, E., eds. 1989. *Nucleic Acids Sequencing: A Practical Approach.* IRL Press.

Huang, X. 1992. A contig assembly program based on sensitive detection of fragment overlaps. *Genomics* 14:18–25.

Hunkapiller, T.; Kaiser, R.; Koop, B.; and Hood, L. 1991. Large-scale and automated DNA sequence determination. *Science* 254:59–67.

Hunkapiller, T.; Kaiser, R.; and Hood, L. 1991. Large-scale DNA sequencing. *Curr. Opin. Biotech.* 2:92–101.

Kececioglu, J. 1991. *Exact and approximation algorithms for DNA sequence reconstruction.* Ph.D. Dissertation, University of Arizona, Tucson, AZ. TR 91-26, Department of Computer Science.

Lawler, E.; Rinnooy Kan, A.; and Shmoys, D., eds. 1985. *The Traveling Salesman Problem.* New York: John Wiley and Sons.

Lin, S., and Kernighan, H. W. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* 21:498–516.

Matsumoto, K.; Arai, M.; Ishihara, N.; Ando, A.; Inoko, H.; and Ikemura, T. 1991. Cluster of fibronectin type-III repeats found in the human major histocompatibility complex class III region shows highest homology with repeats in an extracellular matrix protein, tenascin. *Genomics* 12:485–491.

Parsons, R.; Forrest, S.; and Burks, C. 1993. Genetic algorithms for DNA sequence assembly. In *Proc. of the 1st International Conference on Intelligent Systems in Molecular Biology*, 310–318. AAAI Press.

Sanger, F.; Coulson, A.; Hill, D.; and Petersen, G. 1982. Nucleotide sequence of bacteriophage lambda DNA. *J. Mol. Biol.* 162:729–773.

Schaffer, J. D.; Caruana, R.; L.J.Eshelman; and R.Das. 1989. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proc. of the Third International Conference on Genetic Algorithms*, 51–60. San Mateo, CA: Morgan Kaufmann.

Seto, D.; Koop, B.; and Hood, L. 1993. An experimentally-derived data set constructed for testing large-scale DNA sequence assembly algorithms. *Genomics* 15:673–676.

Staden, R. 1980. A new computer method for the storage and manipulation of DNA gel reading data. *Nucl. Acids Res.* 8:3673–3694.

Starkweather, T.; McDaniel, S.; Mathias, K.; Whitley, D.; and Whitley, C. 1991. A comparison of genetic sequencing operators. In *4th International Conference on Genetic Algorithms*, 69–76.

Sverdlov, E.; Monastyrskaya, G.; Broude, N.; Ushkarev, Y.; Melkov, A.; Smirnov, Y.; Malyshev, I.; Allikmets, R.; Kostina, M.; Dulubova, I.; Kiyatkin, N.; Grishin, A.; Modyanov, N.; and Ovchinnikov, Y. 1987. Family of human Na+, K+-ATPase genes. Structure of the gene of isoform alpha-III. *Cokl. Biochem.* 297:426–431.

Syswerda, G. 1989. Uniform crossover in genetic algorithms. In *Proc. of the Third International Conference on Genetic Algorithms*, 2–9. San Mateo, CA: Morgan Kaufmann.

Waterman, M. S., ed. 1989. *Mathematical Methods for DNA Sequences.* CRC Press.

Whitley, D. 1993. Personal Communication, August 30.