Genetic Algorithms for DNA Sequence Assembly

Rebecca Parsons*
Stephanie Forrest†
Christian Burks‡

Abstract

This paper describes a genetic algorithm application to the DNA sequence assembly problem. The genetic algorithm uses a sorted order representation for representing the orderings of fragments. Two different fitness functions, both based on pairwise overlap strengths between fragments, are tested. The paper concludes that the genetic algorithm is a promising method for fragment assembly problems, achieving usable solutions quickly, but that the current fitness functions are flawed and that other representations might be more appropriate.[1]

Keywords: genetic algorithm, DNA sequence assembly, human genome project, ordering problems, random key representation.

Introduction

Computers are playing an increasing role in large-scale sequencing projects [Burks, 1989; Cinkosky *et al.*, 1991]. The scope of these projects makes it impractical to assemble by hand the resulting sequence fragments into a coherent consensus sequence. Furthermore, the computational complexity of the task makes software tools based on emulation of the manual approach increasingly impractical. For example, Gallant *et al.* have shown that the determination of the shortest common superstring (SCS) for a four-letter alphabet, in which the fragment assembly problem can be recast, is NP-complete (i.e., computationally infeasible for large data sets) [Gallant *et al.*, 1980]. Even the over-simplified, reduced approach of modeling fragment assembly as determining the order of beads (fragments) on a string is NP-complete.[2] The determination of overlap strengths is non-trivial, requiring $O(N^2)$ operations for N fragments [Churchill *et al.*, 1993]. In particular, these bottlenecks are significant for the proposed sequence throughput levels of the human genome project (10^5 - 10^7 bp/day) [Hunkapiller *et al.*, 1991].

The assembly of overlapping sets of fragments into large segments of contiguous sequence, described as *contigs* [Staden, 1980], requires the pairwise alignment of the fragments among themselves, followed by the synthesis of these pairwise results into an optimal global alignment, expressed as a consensus sequence. Randomized search methods have recently been successfully applied to the fragment assembly problem [Churchill *et al.*, 1993]. In this paper, we extend this work to include genetic algorithms. We report our initial results using the genetic algorithm to generate orderings of DNA sequence fragments. The goal is to maximize the overlap strengths of adjacent pairs in the resulting layout.

Introduction to Genetic Algorithms

Genetic algorithms (GAs) are a biased sampling algorithm based on the Darwinian notions of variation, differential reproduction, and inheritance [Holland, 1975; Goldberg, 1989]. Computation in a GA proceeds by creating successive populations of individuals, generally bit strings, with each individual representing a potential solution to the problem. To generate a new population from an existing one, individuals are selected in proportion to their fitness; as a consequence, less fit individuals have few if any representatives (copies) in the new population and more fit individuals have

*Los Alamos National Laboratory, Computer Research Applications Group, MS B265, Los Alamos, NM 87545. Email: rebecca@lanl.gov, phone: (505) 667-2655, Fax: (505) 665-5220 (Corresponding Author)

†Department of Computer Science, University of New Mexico, Albuquerque, NM. 87131-1386 Email: forrest@cs.unm.edu, phone: (505) 277-3112

‡Los Alamos National Laboratory, Theoretical Biology and Biophysics Group, MSK710. Email: cb@intron.lanl.gov, phone: (505) 667-6683

[1]This work was performed under the auspices of the DOE. C.B. was supported in part by the DOE 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).

[2]The proof of this involves a reduction to the Hamiltonian Path problem.

many representatives. Variation is introduced through two operators: crossover and mutation. Crossover simply exchanges substrings between two individuals, creating hybrids which replace their parents, while mutation flips the value of individual bits. Each individual's fitness is evaluated, and the procedure is iterated. A cycle of evaluation, reproduction, crossover and mutation is called a generation. Populations are typically initialized with randomly generated strings, and over time, the GA procedure usually evolves populations of highly fit individuals.

A common crossover operator is two-point crossover in which two crossover points are randomly selected, and the bits between the two crossover points are exchanged, creating two new children. The purpose of crossover is to take two good partial solutions and combine their parts to produce, sometimes, a better solution. The purpose of mutation is both to explore random parts of the search space and to allow for the reintroduction of "genetic" material that may have died off in an earlier generation. Without mutation and crossover, no new points in the search space would be explored. For details on the implementation of genetic algorithms, the reader is referred to [Goldberg, 1989; Davis, 1991].

A primary consideration in the application of GAs to a particular problem is the selection of a representation for candidate solutions. The obvious representation of a layout, an ordered sequence of numbers representing the fragments, is not necessarily appropriate for the fragment assembly problem; we address this issue in the next section. Another crucial consideration in the design of a GA for a particular problem is the creation of a function to evaluate the fitness of an individual. In this paper, we report on two different fitness measures for the fragment assembly problem.

Representation Issues

The fragment assembly problem is closely related to the familiar combinatorial optimization problem, the traveling salesperson (TSP) [Lawler *et al.*, 1985]. Like TSP, the fragment assembly problem restricts legal solutions to those in which each fragment (city) appears exactly once. In addition, both problems have a measure of how desirable it is to have two fragments (cities) adjacent in the final solution. In the case of fragment assembly, this measure is the overlap strength [Churchill *et al.*, 1993], which essentially represents the number of overlapping contiguous base pairs between the two fragments. One significant difference between the two problems is that a legal tour in the TSP is a circle; thus, all starting points on the circle represent equivalent solutions. In contrast, a layout has definite starting and ending points. This issue is discussed at the end of this section.

Unfortunately, TSP and the fragment assembly problem are not easily amenable to a GA approach. The problem is the mapping of a bit string into a layout such that the crossover operation produces a legal layout. The obvious mapping from bit strings to layouts, which for n fragments, uses k bits ($2^{k-1} \leq n \leq 2^k$) for each fragment in the layout and n fragment positions, does not maintain legal layouts using standard crossover. Nothing in the crossover operation prevents the same fragment designator from appearing more than once in the new individual, even if it only appeared once in each of the parents.

There are two ways to approach this problem: a specialized crossover function which only produces legal layouts, or a different mapping from bit strings to layouts. A number of authors explore different representations but concentrate on different crossover operators for TSP [Grefenstette *et al.*, 1985; Oliver *et al.*, 1987; Whitley, 1989; Muhlenbein, 1990]. Muhlenbein also proposes a change in the overall search strategy of the GA by clustering members of the population and performing a smaller number of more directed local searches.

We choose the second approach—selecting a different mapping from bit strings to layouts. This mapping, a modification of one being studied by several authors [Bean, 1992; Syswerda, 1989; Schaffer *et al.*, 1989], ensures that all bit strings represent a legal layout. Thus, the standard two-point crossover operator, or any crossover operator, always produces a legal layout. Our representation is illustrated in Figure 1. For a layout problem consisting of n fragments, select a value k such that $2^k \geq n$, a number of bits sufficient to represent a number up to n. Increasing the value of k beyond the minimum required may slightly improve performance [Parsons and Burks, 1993]. The bit string required is then $m = k * (n + 1)$ bits in length. Mapping a bitstring $b_1, b_2, ..., b_m$ to a layout of n fragments $f_1, f_2, ..., f_n$, proceeds by considering the string as a sequence of $n + 1$ key values, each of k bits. The first n keys are then sorted. If the key value in position j of the individual appears in position i of the sorted list, then fragment j is in position i of the intermediate layout. The last key in the individual specifies the starting position in the intermediate layout; the final layout is simply a rotation of the intermediate layout with the fragment in the specified starting position as the first fragment in the final layout. This starting position is required to allow crossover to swap fragments from one end of the layout to the other. Since the layout is linear, standard crossover would not perform this operation without the starting position.

As an example, consider a bit string (see Figure 1) which produces the following integers, using 3 bits per fragment: 2 7 1 5 3 2. Under this mapping, the fragment layout represented by this bit string is 1 5 4 2 3 with an intermediate layout of 3 1 5 4 2. Since the lowest value, 1, appears in the third position of the string, the first fragment in the intermediate layout is 3. The next lowest value, 2, is in the first position of the bit string, and therefore, the second fragment in this lay-

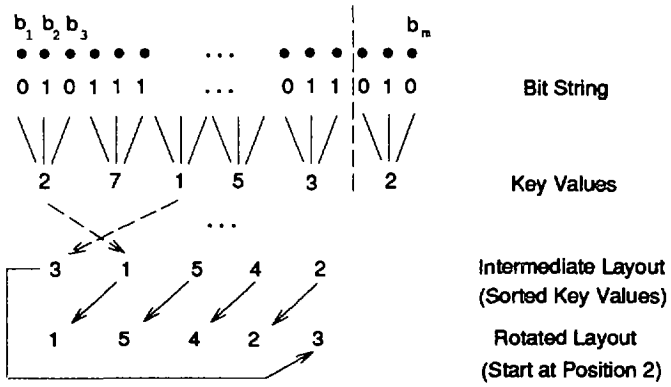

Figure 1: Sorted Order Representation for the Fragment Assembly Problem

out is 1. The last key is 2, so the final layout begins at the second position of the intermediate layout, and wraps to the beginning.

Fitness Function

A GA fitness function maps an individual to its fitness value; this value is what the GA uses to select individuals for the new population. To be effective, the fitness function should be relatively easy to compute and should also accurately reflect the desirability of the solution. We report here on the results using two related fitness functions. The relationship of fragment assembly to TSP suggests a simple fitness function—the sum of the overlap strengths for each of the adjacent pairs in the layout:

$$\mathrm{F1} = \sum_{i=1}^{n-1} w[i, i+1]$$

where $w[i, i+1]$ is the overlap strength of fragments in positions i and fragment $i+1$. The GA attempts to find an ordering and a starting point for the fragments such that the sum of the overlap strengths for adjacent fragments is maximized. By maximizing this value, the resulting layout should have fragments adjacent to each other in the layout that likely have strong overlap in the parent sequence.

The second fitness function, F2, is an elaboration of the previous function and is expressed by the following equation [Churchill *et al.*, 1993]:

$$\mathrm{F2} = \sum_{i=1}^{n} \sum_{j=1}^{n} |i-j| * w[i, j]$$

where i and j range over the positions in the layout. The GA attempts to minimize this value. This fitness function, in addition to examining the strengths of overlap for adjacent fragments, also examines the overlap strengths for fragments farther apart in the layout. The fitness of a layout declines as fragments with a large overlap strength move away from each other. Minimizing this value brings fragments with high overlap strength closer together.

The first function is much easier to compute, requiring looking only at a linear number of overlap strengths. The second function more heavily penalizes a string with fragments clustered improperly than does the first function, although it requires $O(N^2)$ operations to compute. Below, we compare the layouts resulting from the two fitness functions.

Genetic Algorithms and DNA Fragment Assembly

Test Environment

The test sets for this paper were generated using a program to create fragments from a parent sequence in a manner that models the experimental process [Engle and Burks, 1993]. Two parent sequences were used, with several fragment sets generated for each parent.

The first parent sequence, a human MHC class III region DNA with fibronectin type-II repeats HUMMHC-FIB [Matsumoto *et al.*, 1991] with accession number X60189, is 3835 bases in length; the second parent, a human apolopoprotein HUMAPOBF [Carlsson *et al.*, 1986] with accession number M15421, is 10089 bases in length.

For the two test sets, we created five fragment sets by varying the mean coverage between three and seven and the mean fragment length between 200 and 500. We generated one fragment set introducing errors at a rate of 10% into the data. Overlap strengths were computed for each of these fragment sets [Churchill *et al.*, 1993], and any overlap strengths of less than twenty were discarded as noise. For each fragment set, we made five runs of the GA, varying the random seed for the run. The implementation of this GA is a modification of GENESIS, a public domain GA software package [Grefenstette, 1984].

GA Performance

We chose two different, but related, fitness functions with which to test the performance of this GA representation. There are, however, two separate questions to be answered in this context: i) How well does the GA perform in optimizing the particular function? and ii) How good are these solutions as layouts? In our overall approach to fragment assembly, we initially attempt to find a total ordering of the fragments; this total ordering translates to a consensus sequence corresponding to each contig. Several factors influence the quality of a particular consensus sequence. In our test cases, we can measure the percentage of the parent sequence covered by the layout and the percentage of the covered area which is correctly matched by the consensus sequence. Also, at least intuitively, fewer contigs represent a better sequence. However, when dealing with a real problem where the parent sequence is unknown and the distribution of coverage over the actual parent is also unknown, these measures can not be applied (although presumably a smaller number of contigs is still preferable to a larger number).

One desirable characteristic of GAs which applies to our application is that they tend to improve rapidly in the early generations. The graph in Figure 2 shows the change in the F1 fitness scores for 4 of the test runs. The fitness score improves significantly in the first 25% of the run, and then eventually levels off. For this run, the value at which it stabilizes is not an optimal value. This graph is representative of the graphs for the other runs as well.

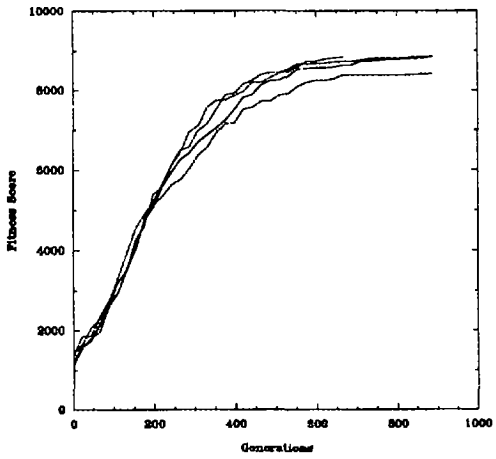

Figure 2: GA Performance on One Data Set

We used relatively standard parameter settings and operators for these runs: a crossover rate of 0.6 with the two-point crossover operator, a mutation rate of .005 per bit per generation, and an iteration count of 400K. The population sizes were set considerable smaller than is generally the case for GAs with strings of this length. We chose these settings based on our preliminary analysis of this representation [Parsons and Burks, 1993]. Table 1 shows, for each of the data sets, the number of fragments to be sequenced, the size, in bits, of the individual with which the GA is operating, and the population size used for the runs on this data set.

Parent	Fragments	Bits	Population
CFIB(5,400)	48	294	200
CFIB(3,500)	24	125	200
CFIB(6,300)	77	446	500
CFIB(7,400)	68	483	500
CFIB(errors)	48	294	200
POBF(5,400)	127	1024	600
POBF(3,500)	61	372	600
POBF(6,300)	202	1624	1200
POBF(7,400)	177	1424	1500
POBF(errors)	127	1024	600

Table 1: Test Data Set Information

Table 2 shows the performance of the GA using the two different fitness functions. For each of the data sets, five runs were made using the same parameter settings but varying the random seed. These runs were repeated for each of the different fitness functions. The first table provides the results from the simpler, linear fitness function (F1); the second table lists the results for the fitness function (F2) which incorporates the weighted distance measures. Recall that a larger F1 score is desirable, and a smaller F2 score is desirable. The first column in the table identifies the parent sequence, using the last four letters of the identifier of the sequence. The qualifiers following this identifier give the average coverage and average fragment length used to generate the data set. For example, the entry for CFIB(7,400) gives the data found for the HUMMHC-FIB sequence with a target mean fragment length of 400 and target mean coverage of the sequence of 7 fragments. The entry labeled CFIB(errors) is the data set with the errors introduced into the fragments. The second column, labeled *Best*, gives the fitness score for the best individual found amongst all five runs. The next column evaluates that same individual using the other fitness function (F2 in the case of the first table, F1 for the second table). The final column lists the number of contigs present in the consensus sequence found, based on the linear ordering given in the best individual.

An interesting fact demonstrated by this table is that equally good layouts from the standpoint of contigs have vastly different fitness scores under both fitness functions. For example, the individuals found for the

	F1 (O(N), Maximize) Function		
Parent	Best	F2 Score	Contigs
CFIB(5,400)	13,900	570,627	5
CFIB(3,500)	7534	488,198	4
CFIB(6,300)	15,809	1,936,597	11
CFIB(7,400)	19,513	2,679,743	7
CFIB(errors)	3873	162,581	7
POBF(5,400)	31,372	4,773,694	23
POBF(3,500)	17,989	551,698	13
POBF(6,300)	27,029	19,224,270	48
POBF(7,400)	33,823	22,285,703	43
POBF(errors)	9234	1,193,219	27

	F2 ($O(N^2)$, Minimize) Function		
Parent	Best	F1 Score	Contigs
CFIB(5,400)	422,049	13,304	4
CFIB(3,500)	73,755	7539	3
CFIB(6,300)	852,087	13,274	8
CFIB(7,400)	1,243,338	20163	3
CFIB(errors)	100,179	3,333	8
POBF(5,400)	1,331,745	24,040	23
POBF(3,500)	179,950	18,092	7
POBF(6,300)	5,776,555	8305	36
POBF(7,400)	7,128,031	13582	28

Table 2: 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.

CFIB(5,400) data set give similar layouts. While the scores under the F1 function are similar, the scores under the F2 function differ by 20%. Notice also that the individuals found for the CFIB(3,500) data set differ under the F1 function by less than 1% while the F2 scores differ by almost 70%. In this case, as well, the layouts are similar, using the number of contigs as the measure.

Comparison of Fitness Functions

The layouts produced using the F2 fitness function are consistently better than those produced using the F1 function. In one case the F2 layout is significantly better, although neither got very close to a reasonable layout. In only one case is the solution found using the F1 function better, but the two layouts are still comparable. However, given the difference in the time required to compute the functions, additional iterations could be applied in the case of function F1 which should provide additional opportunities for improvement. Neither objective function, however, appears to do a good job of characterizing good layouts, since the fitness scores for roughly comparable layouts differ so dramatically. We are beginning to compare the types of solutions found by these functions, using metrics other than the number of contigs, in order to get some insight into what an appropriate fitness function is for this problem.

	Greedy Algorithm		
Parent	F1 Score	F2 Score	Contigs
CFIB(5,400)	13,089	675,782	3
CFIB(3,500)	7352	99,519	2
CFIB(6,300)	16,934	1,649,322	4
CFIB(7,400)	19,909	3,641,818	2
CFIB(errors)	3926	218,414	2
POBF(5,400)	36,297	1,957,601	10
POBF(3,500)	18,509	625,075	7
POBF(6,300)	45,054	2,884,182	6
POBF(7,400)	53,255	9,166,942	5
POBF(errors)	10,639	408,010	15

Table 3: Results of the Greedy Algorithm

Analysis of the Representation

To better understand the performance of the GA and its operators using the sorted order representation, we used a small test problem (five fragments) and simulated the GA operation by hand, tracing the way the operators formed the final (optimum) solution. This tracing showed that the representation does not incrementally construct improved solutions as do traditional genetic algorithms. This behavior stems from the way that the representation separates the problem-level information (ie: fragments) from the individual (ie: the bits themselves). While the bit level building blocks are maintained and augmented, this does not in general translate into the retention and construction of problem-level building blocks (see [Parsons and Burks, 1993] for more information on this issue).

Comparisons to Other Approaches

To test the performance of this approach, we first compare the genetic algorithm to the algorithm implemented by Huang [Huang, 1992]. There are two primary differences between this approach and ours. First, Huang uses a much more precise measure of overlap strength than the one used here. Second, his layout algorithm is basically a greedy algorithm which builds up the layout by extending outward using the most likely pair of overlapping fragments. We implemented a greedy algorithm using the same methodology but our existing computation of overlap strengths. Our greedy algorithm randomly selects a starting fragment, call it n. It then extends the layout by selecting the fragment m that has the strongest overlap with n but has not already been placed in the layout. If no fragment m has a non-zero overlap, then a fragment is selected at random from those remaining to be placed in the layout.

Table 3 summarizes the results of the greedy algorithm on the same data sets tested with the genetic

Bits	Fitness Score				
	Start Value	After 20K	After 100K	After 500K	Optimal Value
294	2432	4774	5312	5482	13,900+
1424	1898	5673	7159	7230	53,000+

Table 4: Hill Climbing Results

algorithm. In general, greedy algorithms perform very well on some data sets, while producing poor results on other, slightly different, data sets. The results for this greedy algorithm varied widely on this problem, as expected. In terms of the number of contigs in the sequence, the greedy algorithm consistently did quite well. However, the greedy sequences left several strong overlap values unaccounted for (far apart in the layout). This behavior explains the large values of the F2 fitness score for these layouts. Although the number of contigs is low, even in those instances, the contigs contain areas of very low overlap. This reliance on low overlaps to maintain the contigs can make the algorithm more prone to errors resulting from repeats or noisy data than other approaches. In addition, since there are areas where strong overlaps are essentially ignored, the solution does not accurately reflect the data.

To test the applicability of more traditional optimization techniques, we implemented a standard hill climbing algorithm for this problem. The hill climber randomly selects a starting bit string and evaluates its fitness. It then randomly alters the string at one bit position and re-evaluates the fitness. If the new string has a higher fitness, then it continues the search from this point in the space. Otherwise, the search continues from the previous best string.

The results from the hill climber were disappointing. Sample fragment sets for each of the sequences were run for 20,000, 100,000 and 500,000 evaluations of the F1 fitness function. Table 4 summarizes the results. Improvements in the fitness of the individual were not significant compared to that required for an acceptable layout.

Conclusions

Sequencing projects using random sequencing strategies can benefit from feedback on the quality of the fragment set created at various stages in the project. This evaluation can help focus sequencing on areas that are not being properly covered by the random approach, or show that insufficient data is available overall to derive the parent sequence. The feedback required for this evaluation is a proposed consensus sequence, based on the available data. The proposed sequence does not need to be perfect, but should have a small number of contigs, should accurately reflect the available data, and should be derivable quickly. These properties are consistent with the use of GAs, which generally provide near-optimal solutions relatively quickly. GAs get their performance from the ability to process many different areas of the search space in parallel.

In this paper, we have examined one particular implementation of a GA for the problem of DNA sequence assembly. There are two critical components in the design of a GA for a problem—the representation and the fitness function. We reported on the performance of this sorted order representation using two related fitness functions. We found that the performance of the two functions is comparable in most cases, with the function F2 almost always performing at least as well as the simpler, linear fitness function, F1. Neither fitness functions appears to represent the desirability of a layout appropriately, however.

For the sequences we examined, the solutions found for the small sequence are quite reasonable. For the larger sequence, only one of the data sets gave a usable layout after 400K iterations. However, this is not a large number of iterations in comparison to the size of the search space. In a related report, we examine the sorted order representation for this problem [Parsons and Burks, 1993]. There we draw the preliminary conclusion that this representation may be adversely affecting the ability of the GA to evolve solutions. The issue of population size and number of iterations, closely related parameters in a GA, must be re-examined in light of the behavior of the representation.

In examining the output at various stages of the GA, we also find that this particular representation does have the characteristic that significant improvements occur quite early in the run. In comparing the performance of the GA to that of the other stochastic approaches, primarily different forms of simulated annealing [Burks *et al.*, 1993], we found that the GA improves faster than the annealer. The annealer ultimately finds a better solution from the standpoint of the objective function but we have yet to study the difference in the usability of the final solutions versus the time to compute them.

Given the initial rapid improvement in the solutions found, the GA proves itself to be a useful tool for directing the course of random sequencing strategies for fragment assembly. However, we did find that the early solutions found for the larger problems were not as high quality as those for the smaller problems. We conjecture that the representation we are using for the GA does not maintain the stability of building blocks and does not allow for incrementally increasing solutions as is typically the case with GAs [Parsons and Burks, 1993]. Alternate representations are possible for this problem. We further address this issue in the final section.

Related Work

Rawlings' [Rawlings, 1986] directory lists fifteen software packages with at least some fragment assembly capabilities; unfortunately, the algorithmic basis of these and more recently-developed tools has not been extensively reviewed; however see Kececioglu [Kececioglu, 1991] for a brief review. Probably the best known fragment assembly package, and the inspiration for many of the others that have been developed, is that of Staden [Staden, 1980]. In the following review, we focus primarily on the issues surrounding layout generation, which is more computationally intensive than calculation of overlap strengths; however, several of the papers cited (e.g. [Kececioglu, 1991]and [Huang, 1992]) also focus on improved algorithms for overlap strength determination. This approach was originally created for application (i) to parent sequences (and corresponding sets of sequence fragments requiring assembling) relatively short compared to those being determined now, and (ii) in a highly-interactive, user-directed mode that is adequate for occasional use when sequence data are accumulating relatively slowly, but which quickly becomes impossible in a steady stream of sequence data [Staden, 1987; Rice *et al.*, 1991] . In addition, most (if not all) of these approaches are based on "greedy" procedures that extend a contig from a given starting point by adding a fragment based only on the local overlap strengths. These approaches had the advantage of efficiency gained by selectively sampling from the set of all possible layouts, but the distinct drawbacks of (i) inexactness due to the high degree of intervention required to achieve the end result, and (ii) potentially ending up at a less-than-optimal final solution.

The first drawback was addressed by Peltola *et al.* [Peltola *et al.*, 1983; Peltola *et al.*, 1984], who used interval graph theory (briefly sketched by Sedgewick [Sedgewick, 1990]) for developing a formal representation of the greedy approach, extending a given contig by the strongest overlapping fragment among the remaining fragments not already in a contig. This allowed one to automate the assembly process; this approach was used most recently by Huang [Huang, 1992]. Over the past few years, several groups [Tarhio and Ukkonen, 1988; Turner, 1989; Li, 1990; Blum *et al.*, 1991; Kececioglu and Myers, 1989; Kececioglu, 1991] explored extending this approach to solving the SCS problem, and, conversely, casting the fragment assembly problem in terms of SCS. They found that greedy solutions, in the worst case, were not guaranteed to have length less than within a factor of two (or more) of the global solution. Kececioglu and Myers were the first to allow for errors in the input fragment sequences, and to provide theoretical analysis of the effect of this constraint on finding SCS solutions. They present several alternative strategies for fragment assembly taking this into account.

We are not aware of other work in applying GAs to the problem of DNA sequence assembly, although they have been applied to the related problem of DNA mapping [Fickett and Cinkosky, 1993; Platt and Dix, 1993].

Future Work

The solutions that the GA finds quickly are, for small problems, quite usable for the screening tasks we are targeting. For the larger problems, however, these early solutions are not always good enough. We plan to explore other representations for fragment assembly in search of one that maintains building blocks in the usual fashion but retains the early improvement behavior of this representation. A desirable feature of this new representation would be the ability to incrementally add fragments to a layout without having to completely re-run the optimization process. One possibility that we have begun to explore is to retain the use of the sort ordering, which implies the closure of the operators over the space of bit strings, but use some other mapping from this sort ordering to the layout. Another possibility is to explore different crossover operators that ensure the legality of the resulting individual. One such operator, the edge-recombination operator [Starkweather *et al.*, 1991] seems particularly applicable since it encourages retention of adjacencies found in the parents.

We have seen that the objective functions do not necessarily accurately reflect the desirability of the solution. Thus, we plan to experiment with other objective functions, specifically looking for a function that incorporates the desire for contiguity of the solution in addition to the quality of the overlaps. To facilitate the design of the new function, we plan to begin by characterizing the types of solutions found using the different fitness functions. One potential function to try is to modify the F1 function to incorporate a sliding window within which the overlap strengths are examined, since the layout problem itself relates more closely to the problem of finding a partial order rather than a total order. It is also possible to attempt to combine with the existing fitness functions some measure of the contiguity of the layout. It is an open question how to balance these different objectives to obtain workable solutions.

Preliminary analysis of the fitness of individuals has also pointed at the desirability in this context of using genetic operators in addition to the crossover operator — specifically various inversion operators. Inversion operators re-order blocks within an individual. While this operator is useless for the traveling salesperson problem, due to its circularity, the fragment assembly problem does have definite start and end points, and thus can benefit from this type of operator.

We would also like to explore the possibility of combining the GA with other techniques. Given the significant improvements found early on, one reasonable scenario is to run the GA and then pass its best solu-

tion or solutions to another optimization program for refinement. This technique is consistent with the role of the GA as a course-grain optimizer with its ability to explore large areas of the search space in parallel. GAs have the potential to significantly increase the amount of data that can be realistically processed and thus contribute to the success of large-scale sequencing projects.

Acknowledgements: The authors wish to thank M. Engle, P. Stolorz, and C. Soderlund for their assistance on this project. This work was initiated during a map assembly workshop at the Santa Fe Institute sponsored by SFI and the Theoretical Division at Los Alamos National Laboratory.

References

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

Blum, A.; Jiang, T.; Li, M.; Tromp, J.; and Yannakakis, M. 1991. Linear approximation of shortest superstrings. In *Proceedings of the 23rd ACM Symposium on Theory of Computing.* 328–336.

Burks, C.; Engle, M.L.; Forrest, S.; Parsons, R.; Soderlund, C.A.; and Stolorz, P.E. 1993. Stochastic optimization tools for genomic sequence assembly. In *Automated DNA Sequencing and Analysis Techniques*, Venter, J. C., editor. Academic Press. In Press.

Burks, C. 1989. The flow of nucleotide sequence data into data banks: role and impact of large-scale sequencing projects. In *Computers and DNA*, Bell, G.I. and Marr, T., editors. Addison-Wesley, Reading, MA. 35–45.

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* 29–51.

Churchill, G.; Burks, C.; Eggert, M.; Engle, M.L.; and Waterman, M.S. 1993. Assembling DNA sequence fragments by shuffling and simulated annealing. Manuscript in preparation.

Cinkosky, M.J.; Fickett, J.W.; Gilna, P.; and Burks, C. 1991. Electronic data publishing and genbank. *Science* 252:1273–1277.

Davis, Lawrence, editor 1991. *The Genetic Algorithms Handbook.* Van Nostrand Reinhold.

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

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

Gallant, J.K.; Maier, David; and Storer, James 1980. On finding minimal length superstrings. *Journal of Computer and Systems Sciences* 20(1):50–58.

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

Grefenstette, J.; Gopal, R.; Rosmaita, B.; and Van Gucht, D. 1985. Genetic algorithms for the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms and Applications.* 160–168.

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

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

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

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

Kececioglu, J. and Myers, E. 1989. A procedural interface for a fragment assembly tool. Technical Report TR-89-5, Depart of Computer Science, University of Arizona, Tucson, AZ.

Kececioglu, J.D. 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.L.; Rinnooy Kan, A.H.G.; and Shmoys, D.B., editors 1985. *The Traveling Salesman Problem.* John Wiley and Sons, New York.

Li, M. 1990. Towards a DNA sequencing theory. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science.* 125–134.

Matsumoto, K.I.; 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* 485–491.

Muhlenbein, H. 1990. Evolution in time and space - the parallel genetic algorithm. In *Proceedings of the Foundations of Genetic Algorithms Workshop.* 316–337.

Oliver, I.M.; Smith, D.J.; and Holland, J.R.C. 1987. A study of the permutation crossover operators on the traveling salesman problem. In *2nd International Conference on Genetic Algorithms.* 224–230.

Parsons, Rebecca and Burks, Christian 1993. An analysis of the random keys representation for DNA sequence assembly. Manuscript in Preparation.

Peltola, H.; Soderlund, H.; Tarhio, J.; and Ukkonen, E. 1983. Algorithms for some string matching problems arising in molecular genetics. In *Information Processing 83*, Mason, R.E.A., editor. Elsevier Science Publishers B. V., (North-Holland). 59–64.

Peltola, H.; Soderlund, H.; and Ukkonen, E. 1984. SEQAID: a DNA sequence assembling program based on a mathematical model. *Nucl. Acids Res.* 12:307–321.

Platt, D.M. and Dix, T.I. 1993. Construction of restriction maps using a genetic algorithm. In *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences, Vol. I: Systems Architecture and Biotechnology*, Mudge, T.N.; Milutinovic, V.; and Hunter, L., editors. IEEE Computer Society Press, Los Alamitos, CA. 756–762.

Rawlings, C.J. 1986. *Software Directory for Molecular Biologists.* Macmillan Publishers, England.

Rice, P.M.; Elliston, K.; and Gribskov, M. 1991. DNA. In *Sequence Analysis Primer*, Gribskov, M. and Devereux, J., editors. Stockton Press, New York. 1–59.

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

Sedgewick, R. 1990. *Algorithms in C.* Addison-Wesley, Reading, MA.

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

Staden, R. 1987. Computer handling of DNA sequencing projects. In *Nucleic Acid and Protein Sequence Analysis: A Practical Approach*, Bishop, M.J. and Rawlings, C.J., editors. IRL Press, Oxford. 173–217.

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.

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

Tarhio, J. and Ukkonen, E. 1988. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science* 57:131–145.

Turner, J. 1989. Approximation algorithms for the shortest common superstring problem. *Inform. Comput.* 83:1–20.

Whitley, D. 1989. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *3rd International Conference on Genetic Algorithms.* 116–121.