CONF-930745-1

LA-UR 93-1617

TITLE: # GENETIC ALGORITHMS FOR DNA SEQUENCE ASSEMBLY

AUTHOR(S): Rebecca Parsons, Stephanie Forrest, and Christian Burks

SUBMITTED TO. Intelligent Systems in Molecular Biology Meeting
July 6-9, 1993
Washington, DC

Los Alamos National Laboratory
Los Alamos New Mexico 87545

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Genetic Algorithms for DNA Sequence Assembly

Rebecca Parsons*
Stephanie Forrest†
Christian Burks‡

April 13, 1993

Abstract

This paper describes a genetic algorithm application to the DNA fragment assembly problems. The genetic algorithm uses a random key representation for representing the orderings of fragments. Two different fitness functions, both based on pairwise overlap strengths between fragments, were 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.

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

1 Introduction

Computers are playing an increasing role in the information flow associated with large-scale sequencing projects [3, 7]. 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 [11] has shown that the fragment assembly problem, recast as the determination of the shortest common superstring (SCS), is NP complete (i.e., computationally infeasible for large data sets). Even for the over simplified,

*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@unmvax.cs.unm.edu, phone: (505) 277 3112

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

reduced approach of modeling fragment assembly as determining the order of beads (fragments) on a string, $O(N!)$ operations are required for N fragments. The determination of overlap strengths [6] is also daunting, requiring $O(N^2)$ operations. In particular, these bottlenecks are significant for the proposed sequence throughput levels of the human genome project (10^5 - 10^7 bp/day)[17].

Randomized search methods have recently been applied to the fragment assembly problem with good success [6]. In this paper, we extend this work to include genetic algorithms. The assembly of overlapping sets of fragments into large segments of contiguous sequence, described as *contigs* [32], 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. We report our initial results using the genetic algorithm to generate orderings of DNA sequence fragments that maximize the overlap strengths of adjacent pairs in the resulting layout.

2 Introduction to Genetic Algorithms

Genetic algorithms (GAs) are a biased sampling algorithm based on the Darwinian notions of variation, differential reproduction, and inheritance [15, 12]. 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 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. The 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 generally 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 re-introduction 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 [12, 8]

A primary consideration in the application of GAs to a particular problem is the selection of a representation for the problem solutions as bit strings. In many cases, simply considering the bit string to be a binary number, or a series of binary numbers, is sufficient. However, as discussed in the next section, this representation is not necessarily appropriate for the fragment assembly problem. Another consideration in the design of a GA for a particular problem is the

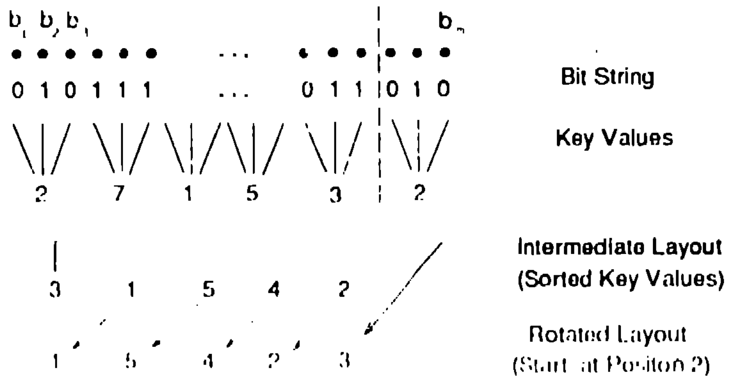

Figure 1: Random Key Representation for the Fragment Assembly Problem

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.

2.1 Representation Issues

The fragment assembly problem is closely related to the familiar combinatorial optimization problem the traveling salesperson (TSP) [20]. 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 [6], which essentially represents the number of overlapping contiguous base pairs between the two fragments. One significant difference between the two problems is that the tour of the TSP is a circle, thus, all starting points on the circle represent equivalent solutions. In contrast, a layout has a definite starting and ending point. This issue is discussed at the end of this section.

Unfortunately, TSP and the fragment assembly problem are not easily amendable 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 ($2^{k-1} < n \leq 2^k$) 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. Grefenstette *et al* [13], Oliver *et al* [24], Whitley [36], and Muhlenbein [23] explore different representations but concentrate on different crossover operators for TSP. 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 suggested by Bean [1], insures 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 is possible and may slightly improve performance [25]. 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 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 layout 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.

2.2 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

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

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 the value, the resulting layout should have fragments adjacent to each other in the layout that likely overlap in the parent sequence.

The second fitness function is a elaboration of the previous function and is expressed by the following equation [6]:

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

where i and j range over the positions in the layout, and $w[i, j]$ 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 a pair of fragments with a large overlap strength moves 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. In contrast, 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. In Section 3.3 we compare the layouts resulting from the two fitness functions.

3 Genetic Algorithms and DNA Sequence Assembly

3.1 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 [9]. 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 HUMMHCFIB [22] with accession number X60189, is 5835 bases in length; the second parent, a human apolopeprotein HUMAPOBF [5] 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 [6], and any overlap strengths of less than twenty was 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 system[11]

3.2 GA Performance

As described in Section 2.2, we chose two different 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 sequence assembly, we initially attempt to find a total ordering of the fragments; this total ordering translates to a consensus sequence, which is a series of contigs. Several factors influence the desirability 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. We also intuitively believe that fewer contigs represent a better sequence. However, when dealing with a real problem where the parent sequence is unknown and the degree of coverage of the actual parent is also unknown, these measures do not apply, 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 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.

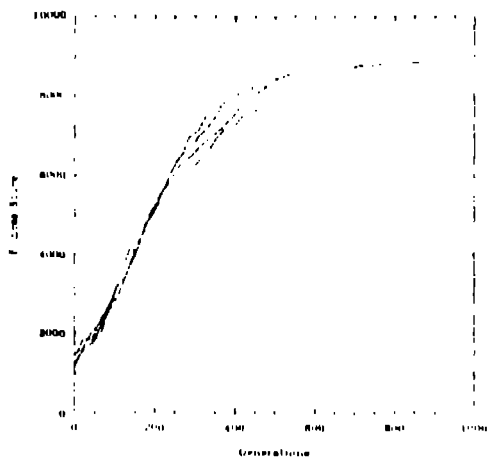

Figure 2: GA Performance on One Data Set

Figure 3 shows the performance of the GA using the two different fitness functions. For each of the data sets, the table includes the best value found across all runs for the data set in terms of the number of contigs found for the corresponding layout, the score under the other fitness function and the number of contigs for this run of that data set. The parent sequence reference includes the last four letters of the identifier of the sequence, and the mean coverage and mean fragment length used to generate the fragment 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 100K. 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 [25].

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 first line in both tables give similar layouts. While the scores under the F1 function are

	F1 (Linear, Maximize) Function			
Parent	Best	F2 Score	Frags	Contigs
CFIB(5,400)	13,900	570,627	48	5
CFIB(3,500)	7531	488,198	24	4
CFIB(6,300)	15,809	1,936,597	77	11
CFIB(7,400)	19,513	2,679,743	68	7
CFIB(errors)	3873	162581	48	7
POBF(5,400)	31,372	4,773,694	127	23
POBF(3,500)	17,989	551,698	61	13
POBF(6,300)	27,029	19,224,270	202	18
POBF(7,400)	33,823	22,285,703	177	43

	F2 (Quadratic, Minimize) Function			
Parent	Best	F1 Score	Frags	Contigs
CFIB(5,400)	422,049	13,304	48	4
CFIB(3,500)	73,755	7539	24	3
CFIB(6,300)	852,087	13,274	77	8
CFIB(7,400)	1,243,328	20163	68	3
CFIB(errors)			48	
POBF(5,400)	1,331,745	24,040	127	23
POBF(3,500)	179,950		61	22
POBF(6,300)	5,776,555	8305	202	36
POBF(7,400)			177	

Figure 3. GA Test Results for Two Fitness Functions

similar, the scores under the F2 function differ by 20%. Notice also that the second line of the two tables 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.

3.3 Comparison of Fitness Functions

The layouts produced using the F2 fitness function are never worse than those produced using the F1 function. In one case it is significantly better, although neither got very close to a reasonable layout. 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.

3.4 Comparisons to Other Approaches

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 fitness function. Table 1 summarizes the results. Improvements in the fitness of the individual were not significant compared to that required for an acceptable layout.

Bits	Fitness Score				
	Start Value	After 0K	After 100K	After 500K	Optimal (F1) Value
294	447	849	936	101	2704
1024	353	939	969	1131	8000+

Figure 1: Hill Climbing Results

4 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 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 the random keys representation using two related fitness functions. We found that the performance of the two functions is comparable in most cases, with the function F2 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 random keys representation for this problem [25]. 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 [1], we found that the rate at which the GA improves is faster than that 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 sequence assembly. However, we did find that the early solutions found for the larger problems were not as of high a 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 [25]. Alternate representations are possible for this problem. We further address this issue in Section 4.2.

4.1 Related Work

Rawlings' [29] directory lists fifteen software packages with at least some sequence assembly capabilities; unfortunately, the algorithmic basis of these and more recently developed tools has not been extensively reviewed; however see Kececioglu [19] for a brief review. Probably the best known sequence assembly package, and the inspiration for many of the others that have been developed, is that of Staden [32]. 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 [19]and Huang [16]) 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 [33, 30] . In addition, most (if not all) of these approaches are based on "greedy" procedures that extend a contig a fragment from a given starting point. 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. [26, 27], who used interval graph theory (briefly sketched by Sedgewick [31]) 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 [16]. Over the past few years, several groups [?1, 35, 21, 2, 18, 19] explored extending this approach to solving the SCS problem, and, conversely, casting the sequence 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 sequence 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 [10, 28].

4.2 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.

In Section 3.3, we saw 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.

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 it's best solution or solutions to another optimization program for refinement. This technique is consistent with the role of the GA as a course grain optimizer with it's 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 Mike Engle, Paul Stolorz, and Cari 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

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

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

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

[4] C. Burks, M.L. Engle, S. Forrest, R. Parsons, C.A. Soderlund, and P.E. Stolorz. Optimization tools for DNA fragment assembly. Algorithm comparison. Poster at the 1993 DOE Human Genome Workshop, manuscript in preperation, 1993.

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

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

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

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

[9] M.L. Engle and C. Burks. Artificially generated data sets for testing DNA fragment assembly algorithms. *Genomics*, 1993. In Press.

[10] J.W. Fickett and M.J. Cinkosky. A genetic algorithm for assembling chromosome physical maps. Submitted manuscript, 1992.

[11] J.K. Gallant. The complexity of the overlap method for sequencing biopolymers. *Journal of Theoretical Biology*, 101:1 17, 1983.

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

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

[14] John J. Grefenstette. Genesis: A system for using genetic search procedures. In *Proceedings of a Conference on Intelligent Systems and Machines*, pages 161–5, Rochester, MI, 1984.

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

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

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

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

[19] J.D. Kececioglu. *Exact and approximation algorithms for DNA sequence reconstruction.* PhD thesis, University of Arizona, Tucson, AZ., 1991. TR 91-26, Department of Computer Science.

[20] E.L. Lawler, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors. *The Traveling Salesman Problem.* John Wiley and Sons, New York, 1985.

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

[22] K.I. Matsumoto, M. Arai, N. Ishihara, A. Ando, H. Inoko, and T. Ikemura. 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*, pages 485–491, 1991.

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

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

[25] Rebecca Parsons and Christian Burks. An analysis of the random keys representation for DNA sequence assembly. Submitted for Publication, 1993.

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

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

[28] D.M. Platt and T.I. Dix. Construction of restriction maps using a genetic algorithm. In T.N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences, Vol. I: Systems Architecture and Biotechnology*, pages 620–629. IEEE Computer Society Press, Los Alamitos, CA, 1993.

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

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

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

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

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

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

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

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