Genetic Algorithm for the Double Digest Problem in Genetics

A dissertation submitted in partial fulfilment
of the requirements of M.Tech.(Computer Science)
degree of Indian Statistical Institute, Kolkata
by

Satyajit Banerjee

under the supervision of

Dr. C. A. Murthy & Dr. S. Sur-Kolay
Indian Statistical Institute
203, Barrackpore Trunk Road
Kolkata-700 108.

July 9, 2002

Indian Statistical Institute

203, Barrackpore Trunk Road,

Kolkata-700 108.

Certificate of Approval

This is to certify that this thesis titled "**Genetic algorithm for the Double Digest Problem in Genetics**" submitted by **Satyajit Banerjee** towards partial fulfillment of requirements for the degree of M. Tech in Computer Science at Indian Statistical Institute, Kolkata embodies the work done under our supervision.

CAM 9.7.2002.

Sur-Kolay 9.7.'02

Dr. C. A. Murthy,
Dr. S. Sur-Kolay,
Indian Statistical Institute,
Kolkata-700 108.

Acknowledgements

I take pleasure in thanking Dr.C.A. Murthy and S. Sur-Koley for their friendly guidance throughout the dissertation period. Their pleasant and encouraging words have always kept my spirits up.

I take the opportunity to thank my classmates, friends and my family for their encouragement to finish this work.

Satyajit Banerjee

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last decade, the progress in the filed of genetics has been phenomenal. In particular, the gene mapping and sequencing problem have drawn the attention of mathematicians and computer scientists with the consequence of the emerging field of computational molecular biology [1]. Many problems related to identification and analysis of physical sequencing of DNA molecules have been formulated very elegantly as combinatorial problems and even been proven to be NP-hard. Herein lies the necessity for designing efficient heuristics. Genetic algorithms have been found to be very successful in solving computationally complex problems. Thus the motivation for applying genetic algorithms to combinatorially hard problems in genetics arose.

1.2 What are Genetic Algorithms?

GA is one of the most promising tools in the Soft Computing domain. Due to its versatile ability to tackle optimization problems arising from almost every branch of science and technology it has become an important tool to solve 'hard' problems nowadays [2, 3]. With GAs having such versatile abilities, one might think that the inner workings of a GA would be very complex.In fact, the opposite is true. Simple GAs are mainly based on simple string alteration and substring concatenation,nothing more, nothing less. Even more complex versions of GAs still use these basic ideas as the core of their search engines.

```
BEGIN AGA
    Create initial population at random.
    WHILE NOT stop DO
        BEGIN
            Natural selection: Select parents from the population.
            Cross over: Produce children from the selected parents.
            Mutation: Mutate individual in the population.
            Restore the children for the next generation population.
        END
    Output the best individual found.
END AGA.
```

Figure 1.1: Structure of Genetic Algorithm.

1.2.1 Structure of a Genetic Algorithm

For applying GA on some particular optimization problem the Representation of strings to encode the search space and an appropriate Cost Function are to be judiciously designed before hand. Once the string representation and cost function are fixed, a set of initially chosen strings are subjected to a set of genetic operations,namely *natural selection, cross over* and *mutation* to obtain the set of next generation strings. Each operation is associated with a non-zero probability of being applied. Further details of these operators appear in section 1.6. The same process is repeated for some pre-defined number of times. Over the generations the process converges and the genetic operations give birth to the optimum strings. The pseudo-code of the abstract genetic algorithm is furnished in Figure 1.1.

1.2.2 Stochastic nature and power of Genetic Algorithm

Many of the combinatorial optimization problems having substantial practical importance, are known to be computationally hard. The inherent stochastic nature of Genetic Algorithm is found suitable in solving such hard problems. As the operations of the genetic algorithm are very simple and can explore the entire search space very efficiently, GAs are used extensively on computationally hard optimization problems to get the near optimal result in considerably small time.

1.2.3 Simple and Elitist models

GAs are broadly categorized into two models as follows:

Simple model of GA

These are GAs consisting of only the above mentioned Genetic Operators namely *natural selection*, *cross over* and *mutation*. In this category there is no explicit preservation of the better(and even the best) strings. So it may very well happen that better(or the best) strings can get generated and lost susequently over the generations.

Elitist model of GA

In this category the GAs are equipped with one more operation called *elitism* apart from the above mentioned genetic operators. Elitism is basically a method of preserving the best(sometimes better also) strings evolved out of the genetic operations over the generations.

1.2.4 Basic Genetic Operators

One minor operator namely *natural selection* and two main operators namely *cross over*, *mutation* form the body of the GA.They are briefly described below with examples of binary strings of length 10.

Natural Selection

It is the process of choosing strings in the working Population with probability proportional to their cost(in case of maximization problem). This method is an implementation of the law 'Survival of the fittest'. It is evident from the choosing methods that multiple copies of the same string can exist in the working population. This is even desirable, since the stronger strings will begin to dominate, eradicating the weaker ones from the population. There are also difficulties with this, as it can lead to premature convergence on a local optimum.

Cross Over

Crossover in biological terms refers to the blending of chromosomes from the parents to produce new chromosomes for the offspring. The analogy carries over to Crossover operator in GAs. It is a method of substring concatenation.

Given two parent strings from the working domain the GA first calculates whether crossover should take place using a parameter called the Crossover probability. If so then a Cross Over cut-point(ie. an index in the string) is randomly fixed and both the parent strings are split at this cut point. Now two offsprings are generated by cross-concatenating the four substrings.

Example: Let the parent strings be $p1 = 1111010101$, $p2 = 0101001101$ and let 4 be the Cross Over cut-point. Then the splitted substrings are $p1(L) = 1111$, $p1(R) = 010101$, $p2(L) = 0101$, $p2(R) = 001101$. Therefore, after cross-concatenation the generated offsprings are $os1 = p1(L)||p2(R) = 1111001101$, $os2 = p2(L)||p1(R) = 0101010101$.

Mutation

Natural selection and crossover alone can obviously generate a staggering amount of differing strings. However, depending on the initial population chosen, there may not be enough variety of strings to ensure that the GA explores the entire problem space. Or the GA may find itself converging on strings that are not quite close to the optimum it seeks due to a bad initial population.

Some of these problems are overcome by introducing a mutation operator into the GA. The GA has a mutation probability which dictates the frequency at which mutation occurs.The GA checks to see if it should perform a mutation by randomly generating a number between 0 & 1 and then checking against the Mutation probability. If it should, it randomly changes a position of the string to a new one. In our binary strings, 1s

Example: Let the GA decide to mutate the bit position 4 of the string s_1 = 1101000001. Then the resulting string after the Mututation will be s_2 = 1100000001.

1.2.5 Convergence property

The basic philosophy of genetic algorithm lies in the fact that if the genetic operators are appropriately designed for an elitist model of GA to solve a specific problem in hand then the process is guaranteed to converge along with the optimal strings in the final population after infinite iterations. If alone the Mutation operation is properly designed so as to esure that each string in the working domain can be mutated to any string in one step then it can be proved to converge to the optimum solution [4].

1.3 Scope of this project

In the area of gene mapping, when a restriction enzymes is applied on a DNA , the DNA molecule is cut up into fragments at specific locations which depend on the DNA sequence and the enzyme. By gel electrophoresis method, the lengths of these digested fragments can be measured and then from the values of these lengths, a valid map sequence has to be derived. It is known that data from the application of a single enzyme does not suffice nor does that from application of two different enzymes. However, it is possible to construct a valid map by application of two enzymes separately as well as in conjunction on the same DNA. Multiple valid solutions may be possible. This is the essence of the Double Digest Problem (DDP), which is a well-known problem in genetics.

As DDP is an NP-complete problem, our major goal of this project is to design an elitist GA to obtain *all* valid solutions to the Double Digest Problem (DDP). Incidentally, not only is DDP NP-complete but also structurally it has certain strong similarities with the famous NP-complete problem of Traveling Salesman Problem (TSP). Several heuristic approaches to TSP, including GA with a variety of genetic operators, exist in the literature [5]. So, during the process of designing a GA for DDP, GA solution approaches to TSP were revisited and a set of new genetic operators have been proposed. Finally, the GA for DDP was designed with a totally new set of genetic operators and algorithmic structure.

It needs to be emphasized that almost all the existing genetic algorithms produce *an* optimal solution to the given problem and terminate, but not all the optimal solutions if there are more than one. Intuitively, producing all optimal solutions seems to be more difficult especially for problems where the number of optimal solutions can be very high. DDP happens to be one such problem and hence the design of a GA for it is all the more challenging.

1.4 Organization of this report

In Chapter 2, the Travelling Salesman Problem(TSP) is revisited with a set of new genetic operators. Chapter 3 consists of the definition of Double Digest Problem(DDP), the compexities of the structure of its solution, existence of multiple solutions, partioning its solution space and different approaches to solve DDP. In Chapter 4, the DDP is addressed by genetic algorithm along with detailed description of the string representation cost function, genetic operators, results etc.

Chapter 2

TSP revisited with a new set of genetic operators

2.1 Formal definition of TSP and its relevance

The Travelling Salesman Problem is, given a collection of cities,to determine the shortest tour which visits each city precisely once and then returns to its starting point.More formally we may define the TSP as follows:

Let for n number of cities an $n \times n$ Distance matrix $C = [c_{i,j}]$,is given. We want find the non-cyclic permutation π of the cities $(1, ..., n)$ so as to minimize the total distance,

$$\zeta = \sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} + c_{\pi(n),\pi(1)} \tag{2.1}$$

The Travelling Salesman Problem is a very well known NP-complete problem and therefore any problem belonging to the NP-Class can be reduced to TSP in polynomial time. Therefore solving TSP has many fold implications as there are several NP-complete problems in real life.

2.2 A new approach with modified Genetic operations & Greedy amalgamation

In this section a few modifications of the genetic operations are suggested over the existing genetic operations as discussed in the preceding section. The structure of the algorithm is given in Figure 2.1. Greedy heuristic method is partially injected into the classical structure of genetic algorithm as will be discussed shortly.

```
BEGIN ALGM_TSP
    Make initial population of tours random.
    WHILE NOT stop DO
        BEGIN
            Natural selection: Select parents from the population.
            cross over: Produce children from the selected parents.
            Mutation: Mutate the individual.
            Elitism: Restore the best tour obtained among the present
            and previous generation to the next generation.
        END
    Output the best individual found.
END ALGM_TSP
```

Figure 2.1: The abstract structure of Genetic Algorithm

2.2.1 String representation and Cost function

As we are interested in finding the shortest tour for a given collection of n cities, we designate the cities by 0 through $n-1$ and without loss of generality assume that the staring point is city #0. So there are as many as $(n-1)!$ many different tours possible out of which we have to find the shortest one.

So the natural choice of string representation is an array T of $n-1$ number of cities which basically a permutation of $(1, ..., n-1)$. The starting point is always assumed to be city #0 and thus dropped from the string.

The Cost function

$$\zeta : \{\text{Set of all valid strings}\} \rightarrow \{\text{Set of non-negative real numbers}\}$$

is basically the total distance of the tour and defined in exactly in the similar way as that of in eqn. 2.1

$$\zeta = c_{0,T(1)} + \sum_{i=1}^{n-2} c_{T(i),T(i+1)} + c_{T(n-1),0} \tag{2.2}$$

Where, $C = [c_{i,j}]$ is an $n \times n$ cost matrix such that $c_{i,j}$ gives the cost of visiting city $\#j$ from city $\#i$, for all i, j. Therefore our goal in this case is to find the string with minimum cost along with its cost-value.

2.3 Different Genetic Operators for TSP: Literature survey

Out of the different types of cross over and mutation operators used to solve TSP by GA [5] two cross over and two mutation operators which are somewhat related to our designed genetic operators are described below. with examples.

2.3.1 Two different Cross Over operations

Partially-Mapped Crossover (PMX)

The partially-mapped crossover operator was suggested by Goldberg and Lingle (1985). It passes on ordering and value information from the parent tours to the offspring tours. A portion of one parent's string is mapped onto a portion of the other parent's string and the remaining information is exchanged. Consider, for example the following two parent tours: (12345678) & (37516824)

The PMX operator creates an offspring in the following way. First, it selects uniformly at random two cut points along the strings, which represent the parent tours. Suppose that the first cut point is selected between the third and the fourth string element, and the second one between the sixth and the seventh string element. For example, $(123 * 456 * 78)$ and $(375 * 168 * 24)$. The substrings between the cut points are called the mapping sections. In our example they define the mappings $4 \leftrightarrow 1$, $5 \leftrightarrow 6$ and $6 \leftrightarrow 8$. Now the mapping section of the first parent is copied into the second offspring, and the mapping section of the second parent is copied into the first offspring, growing offspring 1: $(xxx * 168 * xx)$ and offspring 2: $(xxx * 456 * xx)$. Then offspring $i(i - 1, 2)$ is filled up by copying the elements of the i^{th} parent. In case a city is already

present in the offspring it is replaced according to the mappings. For example, the first element of offspring1 would be a '1' like the first element of the first parent. However, there is already a '1' present in offspring1. Hence, because of the mapping 1 ⇔ 4 we choose the first element of offspring 1 to be a 4. The second, third and seventh elements of offspring 1 can be taken from the first parent. However, the last element of offspring 1 would be an 8, which is already present. Because of the mappings 8 ⇔ 6, and 6 ⇔ 5, it is chosen to be a 5. Hence, offspring-1 is (4 2 3 * 1 6 8 * 7 5). And analogously, we find offspring-2 to be (3 7 8 * 4 5 6 * 2 1). It is worth noting that the absolute positions of some elements of both parents are preserved.

A variation of the PMX operator is described in Grefenstette (1987): given two parents the offspring is created as follows. First, the second parent string is copied onto the offspring. Next, an arbitrary subtour is chosen from the first parent. Lastly, minimal changes are made in the offspring necessary to achieve the chosen subtour. For example, consider parent tours (1 2 3 4 5 6 7 8) and (1 5 3 7 2 4 6 8) and suppose that subtour (3 4 5) is chosen. This gives offspring (1 3 4 5 7 2 6 8).

Order Based Crossover (OBC)

The order based crossover operator (Syswerda 1991) selects at random several positions in a parent tour, and the order of the cities in the selected positions of this parent is imposed on the other parent. For example, consider the parents (1 2 3 4 5 6 7 8) and (2 4 6 8 7 5 3 1) and suppose that in the second parent the second, third, and sixth positions are selected. The cities in these positions are city 4, city 6 and city 5 respectively. In the first parent these cities are present at the fourth, fifth and sixth positions. Now the offspring is equal to parent 1 except in the fourth, fifth and sixth positions (1 2 3 * * * 7 8). We add the missing cities to the offspring in the same order in which they appear in the second parent tour. This results in (1 2 3 4 6 5 7 8). Exchanging the role of the first parent and the second parent gives, using the same selected positions, (2 4 3 8 7 5 6 1):

2.3.2 Two different Mutation operations

Displacement Mutation (DM)

The displacement mutation operator (Michalewicz 1992) first selects a subtour at random. This subtour is removed from the tour and inserted in a random

place. For example, consider the tour represented by (1 2 3 4 5 6 7 8); and suppose that the subtour (3 4 5) is selected. Hence, after the removal of the subtour we have (1 2 6 7 8): Suppose that we randomly select city 7 to be the city after which the subtour is inserted. This results in (1 2 6 7 3 4 5 8). Displacement mutation is also called cut mutation (Banzhaf 1990).

Insertion Mutation (ISM)

The insertion mutation operator ((Fogel 1988); (Michalewicz 1992)) randomly chooses a city in the tour, removes it from this tour, and inserts it in a randomly selected place. For example, consider the tour (1 2 3 4 5 6 7 8); and suppose that the insertion mutation operator selects city 4, removes it, and randomly inserts it after city 7. Hence, the resulting offspring is The insertion mutation operator is also called the position based mutation operator (Syswerda 1991).

2.3.3 Brief description of the new Genetic operators and Greedy Heuristic method

In the following sections the additions, alterations and modifications of the basic genetic operators are described.

Natural Selection

This operation is designed by a common method of doing natural selecton in GA called the Roulette Wheel method. The Roulette Wheel method simply chooses the strings in a statistical fashion based solely upon their relative (ie. percentage) cost/fitness values.

Here the Natural Selection is done by randomly choosing strings from the working population with probability inversaly proportional to their cost.

Cross Over

The primitive cross over operation as described in the previous chapter with binary strings, might generate invalid strings in this case since here all the strings are permutations of $(1, .., n-1)$. e.g., Consider for $n = 10$, $s_1 = 1, 2, 3, 4, 6, 9, 8, 5, 7$, $s_2 = 2, 1, 9, 8, 5, 6, 3, 7, 4$ are two the two strings eligible for cross over and let the cross over cut-point is fixed to be 4. Then the

pair of offsprings generated by cross-concatenation of the four substrings are respectively, $a_1 = 1, 2, 3, 4, 5, 6, 3, 7, 4$, $a_2 = 2, 1, 9, 8, 6, 9, 8, 5, 7$. Clearly neither of the offsprings are valid strings 3 & 4 in the first offspring and 8 & 9 in the second offspring are repeated.

So to get rid of this difficulty the cross over is done in the following way: A random number is generated at first in [0, 1]. If the number is less than cross-over probability, then the cross-over operation takes place. A randomly chosen cross over cut-point divides the parent strings in left and right substrings. The left substrings of the parents s_1 and s_2 are copied to the left substrings of the offsprings c_1 and c_2 respectively. Then the elements of the right substring of s_1 are inserted in the right substring of c_1 in the order in which they occur in s_2. Similarly, the right part of c_2 is obtained by inserting the elements of the right substring of s_2 in the order it occurs in s_1. For the previous example now the generated pair of offsprings will be, $b_1 = 1, 2, 3, 4, 9, 8, 5, 6, 7$, $b_2 = 2, 1, 9, 8, 3, 4, 6, 5, 7$. Clearly this method allows only the generation of valid strings. Apart from that it is a kind of order preserving cross over which helps to attain the optima efficiently. Regarding Cross Over probability it is kept fixed across the generations and its value is fixed at 0.85 which is a standard value recomended in the literature.

Mutation

For mutation operation also a slight modification is done over the primitive version as discussed in the previous chapter with binary strings.

Here in this case each element of the string is made liable to be mutated as follows, For each i, while mutating the ith element of the string, First a random number between 0 & 1 is generated and compared with the Mutation probability. If the generated random number happens to be less than or equal to the Mutation probability then, a random index betwen 1 & n - 1 is generated(say j) and the i^{th} & j^{th} elements of the string are swapped. For example, let $s = 1, 3, 2, 5, 4, 9, 8, 7, 6$ be the string and 3^{rd} element is through with the Mutation probability check. Let 7 is the randomly generated index for swapping. Therefore, after swapping the mutated string will look like $s1 = 1, 3, 8, 5, 4, 9, 2, 7, 6$. Clearly this mutation operation ensures the generation of valid strings only.

It is also clear from the description of of the mutation operartion that each valid string can be mutated to any string in one step with non-zero probability. As we discussed in the previous chapter this criterion is essential for faster

convergence.

Regarding mutation prbability it is not kept constant over the generations. Rather it is varied in cycles of appropriate intervals linearly from $1/(n-1)$ to 0.45. The argument for applying this variation is that it helps in exploring the search space efficiently and disallow the GA to get stuck in the local optima.

Elitism

In this operation the string with least Cost is preserved over the generations. But as our GA method is amalgamated with greedy heuristic an appropriate no of best strings are preserved over the generations to apply the greedy heuristic on them. In the next section the greedy heuristic is discussed.

Amalgamation of Greedy heuristic

This basically a foreign operation injected in the classical structure of GA. This works on the set of best strings preserved over the generations by the Elitism operation. This works as follows,
For a chosen string and for all i, it finds the minimum cost destination for going from i^{th} element city to the set of cities lying in the right side of the ith index from the cost matrix. It swaps the $(i+1)$th element with the city-element corresponding to min-cost destination. After executing the operation for all i it checks whether the resulting string has smaller cost than the original string. If so, the original string is replaced by the resulting string.

As because it is a deterministic heuristic method which works with the cost matrix in hand, it helps the stochastic environment of the working of GA to derive some positive direction.

2.4 Results and Comparisons

One standard GA and our suggested amalgamated GA are run on three different types of data sets namely, Practical, Artificial & Minimal Deceptive data sets with #cities 50, 80 and 100 and then compared susequently. These three types of data sets and the standard GA for comparison are briefly described below.

2.4.1 Practical Data Set

These are practical instances of the Travelling Salesman Problem. One can easily collect/generate the instances for testing the performance of algorithms. There are a plenty of the Practical TSP instances which are considered to be Benchmark instances in TSPLIB. The Benchmark instances are tried by several algorithms from time to time and the best known solutions are recorded against each instance. The Benchmark instances are recomended to work on to judge the performance of one's algorithm.

2.4.2 Artificial Data Set

These are artificially generated data set such that the minimum cost tour can be easily calculated by some nice mathematical formula. So the performance of the algorithm on such instances can be easily evaluated since the very solution is known, e.g., let the distance between the city #i & city #j is $|i - j|$ for all i, j. If we assume w.l.g that the staring point is alwyas city #1 then it can be shown that the optimum strings should be having a configuration of an increasing sequence followed by a decreasing sequence with the relaxation that any one sequnces can be also of zero length. Therefore, the optimum cost will be $2n - 2$.

2.4.3 Minimal Deceptive Data Set

These instances are artificially constructed to judge the roboustness of GAs. These class has a peculiar property. In these instances, the minimum strings are surrounded by strings having very high costs. These are intentionally designed to befool the GA search engines. Naturally it is the most difficult hardle for a new algorithm to stand.

2.4.4 A standard GA developed by Michel Lalena

In this GA, the tours are encoded as a 2-dimensional array (an $N \times N$ matrix) of bits that store city adjacencies in both directions.A set bit indicates a city connection. If element $[X, Y]$ is set, then city X connects to city Y. Only 2 bits will be set in every row and column. Now in every iteration, a number of tours are chosen from the list of tours. This is the tournament set. The best two of these tours from the tournament will be combined to form two new

Table 2.1: Performance of the algorithm on different data sets

Data Types	Data Size	Our algorithm	Other algorithm
Artificial	50	98	408
Data Set	80	158	648
	100	198	844
Practical	48	38,071	40,053
Data Set	70	776	895
	100	25,330	27,800
Minimal	50	50	48,002
Deceptive	80	80	72,008
Data Set	100	100	80,010

tours using crossover. These two new solutions will replace the worst two tours from the tournament.

A greedy crossover operation combines the two tours to hopefully form two better tours. All adjacencies that are shared by the parent are placed in both children. This is done by performing a binary AND on the two parent matrices. When the parents disagree, the children alternate which parent they will get an adjacency from. If an adjacency produces a conflict (city used twice or incomplete tour), then a random city is used instead. As the cross over itself is very powerful no mutation is needed in this case.

The results obtained are shown in the following Table 2.1 with five different initial populations per instance and for each initial population, the number of trials is 10; and for each trial the number of iterations is 10,000.

2.5 Discussion and Conclusion

It is clear from the results obtained that the newly designed genetic operators in our algorithm is strong enough to find its room in the literature. The most attractive features are that even on artificial data set and minimal deceptive data set its performance is superb.

Chapter 3

Double Digest Problem(DDP)

3.1 Formal definition of Double Digest Problem and its relevence

The Double Digest Problem is a very wellknown problem in biological computing domain. The problem is basically related to the recognition of physical sequencing of a DNA. Formally the problem is described as follows, Let, A & B are two restriction enzymes and D is a DNA molecule of length L. We are given three multisets $a = ||A|| = \{a_i : 1 \leq i \leq m\}, b = ||B|| = \{b_i : 1 \leq i \leq n\}$ and $c = ||A\&B|| = \{c_i : 1 \leq i \leq l\}$. Those are the collection of fragments obtained by the the application of enzyme A, enzyme B and enzymes A & B together respectively on the DNA molecules. Clearly, in absence of any measurement error,

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{n} b_i = \sum_{i=1}^{l} c_i = L \tag{3.1}$$

For σ a permutation of $(1, 2, ..., m)$ and μ a permutation of $(1, 2, ..., n)$, call (σ, μ) a configuration. By ordering A & B acording to $\sigma\&\mu$,respectively we obtain the set of location of cut sites.

$$S = \{s : s = \sum_{j=1}^{r} a_{\sigma(j)} \text{ or } s = \sum_{j=1}^{t} a_{\mu(j)}; 0 \leq r \leq n, 0 \leq t \leq m\} \tag{3.2}$$

Because we want to record only the location of cut sites, the set is not allowed repetitions, that is, S is not a multiset. Now label the elements of S such that,

$$S = \{s_j : 0 \leq j \leq l\} \text{ with } s_i \leq s_j \text{ for } i \leq j \tag{3.3}$$

The double digest implied by the configuration (σ, μ) can now be defined by

$$C(\sigma, \mu) = \{c_j(\sigma, \mu) : c_j(\sigma, \mu) = s_j - s_{j-1} \text{ for some } 1 \leq j \leq l\} \qquad (3.4)$$

where we assume the set is ordered by size in the the index j. Then the problem is to find a configuration (σ, μ) such that $C = C(\sigma, \mu)$,where $C = A\&B$ is determined by experiment. The problem in general is NP-complete.

3.2 Multiple solutions in the Double Digest Problem

In many instances, the solution of DDP is not unique. For example, with $a = ||A|| = \{1, 3, 3, 12\}$, $b = ||B|| = \{1, 2, 3, 3, 4, 6\}$ and $c = ||C|| = ||A\&B|| = \{1, 1, 1, 1, 2, 2, 2, 3, 6\}$ two distinct solutions are shown in Figure 3.1. Our goal

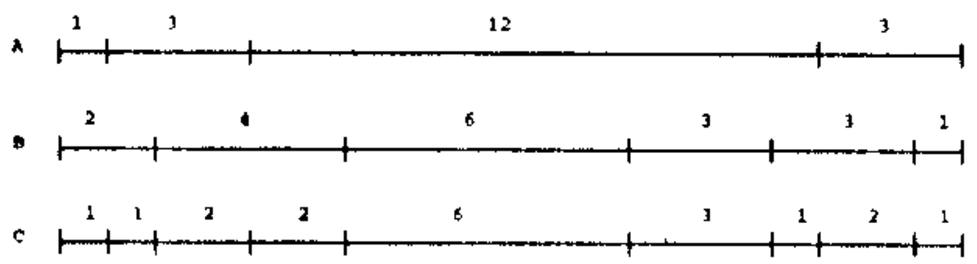

Figure 3.1: Existence of multiple solution of DDP

is to find all posible solutions of a given DDP. The criterion of getting all possible solutions makes the task more difficult.

3.3 Classifying Multiple Solutions

Though the structure of multiple solutions of DDP is very complex, a few types of equivlence relatios can be defined over the set of all solutions. A few types of equivalence relations are defined below with examples:

3.3.1 Reflection Equivalence

Whenever $\sigma = (\sigma_1, ...\sigma_n)$ and $\mu = (\mu_1, ..., \mu_m)$ is a solution to DDP then clearly $\sigma' = (\sigma'_1, ...\sigma'_n)$ and $\mu' = (\mu'_1, ..., \mu_{m'})$ is also a solution to that DDP. In a very real sense, they present the same solution to the problem, as they differ only by an arbitrary choice of orientation. Therefore it is quite reasonable to consider the set of solutions modulo the reflection relation.

3.3.2 Overlap Equivalence

Let a solution configuration M has $t-1$ coincident cut sites, then there are t(connected) components in the map. If he components are permuted and/or any subset of the components are reflected and/or the set of pefectly contained enzyme fragments within any other enzyme fragment are permuted the resulting configurations still remain valid solutions. According to this observation the overlap equivalence is defined as follows, If a solution configuration C can be transformed to another solution configuration C' by a sequence of above mentioned operations, then , C and C' are said to belong in the same overlap equivalence class.

3.3.3 Overlap Size Equivalence

Let us define the overlap size data of a map to be

$$\{(|A_{i_s}|, |B_{j_s}|, |C_s| : C_s = A_{i_s} \bigcap B_{j_s}\} \tag{3.5}$$

Two solutions to DDP with data $\{a_1, ..., a_n\}, \{b_1, ...b_m\}$ and $\{c_1...\}$ are said to be overlap size equivalent if they have the same set of overlap size data.

3.3.4 Cassette Equivalence

This is more or less a generalization of the above three equivalence classes. This equivalence class is based on cassette transformations of restriction maps. To start let us define what a cassette is. For each pair i, j with $1 \leq i \leq j \leq l$ define,

$$I_C = \{C_k : C_i \leq C_k \leq C_j\}$$

which is the set of intervals from C_i to C_j. The cassette defined by I_C is the pair of sets of intervals (I_A, I_B), the sets of all blocks of A and B, respectively, that contain a block of I_C. Define m_A & m_B to be the minimal elements of

the the leftmost blocks of I_A & I_B respectively. The left overlap is defined to be $m_A - m_B$. The right overlap is defined similarly, by substituting maximal for minimal, and rightmost for leftmost.

Now if two disjoint cassettes of solution $[A, B]$ to DDP(a,b,c) have identical left and right overlaps and if when the overlaps are non zero, the DNA comprising the overlap is a single double digest problem, then they can be exchaged to form a new solution.$[A', B']$ to DDP(a,b,c). Also if the left and right overlaps of a cassette have same absolute value but different sign,then the cassette can be reflected. These are two types of cassette transfomations.See Figure 3.2 and 3.3

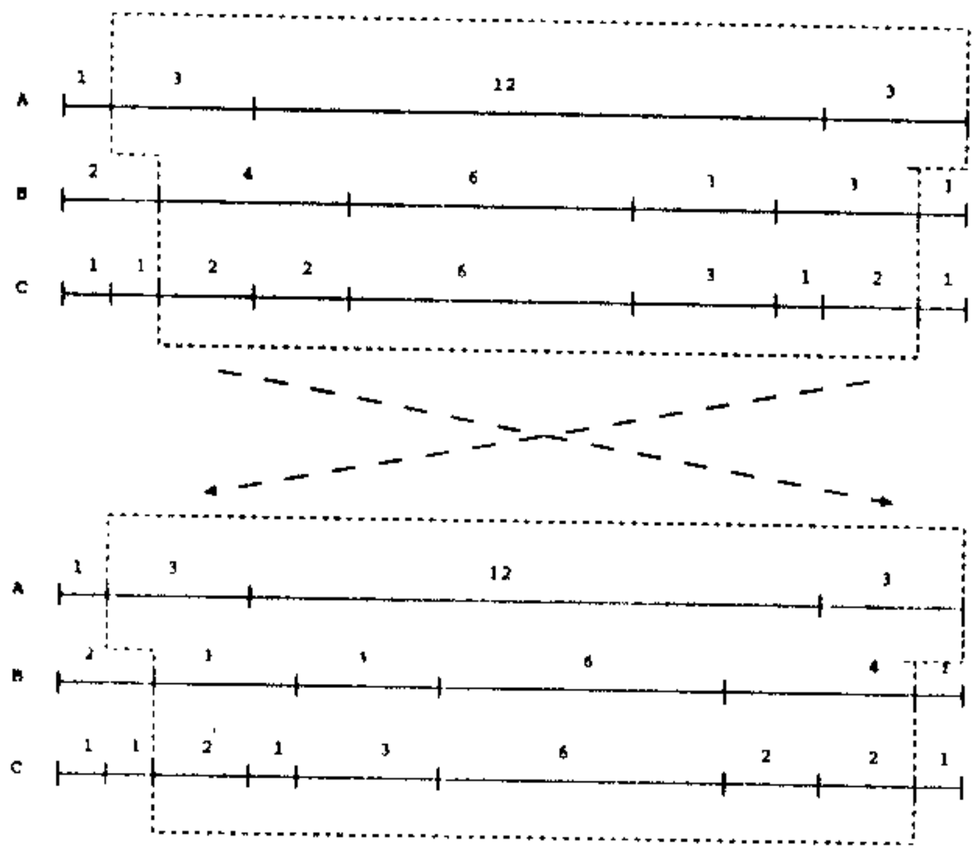

Figure 3.2: Cassette reflection

Now the cassette equivalence on the set of all solutions of to DDP is defined as follows, $[A, B]$ and $[A', B']$ belong to the same cassette equivalence class if and only if there is a sequence of cassette transformations and permutation of non border block uncut fragments tarnsforming $[A, B]$ into $[A', b']$. The idea of this cassette equivalence is simple enough yet a strong one. What basically intended is to put the solution configurations, having same set containments and boundary consisting of the superimposed fragments for all enzyme fragments,in one equivalence class. More formally, we define the idea below, Let, $I_s(X) = \{|C_i| : C_i \subseteq X\}$. It is a multiset where C_is are superimposed fragments and X is any enzyme fragment.

For $|I(X)| > 1$ define the following,
$I_s^*(X) = \{|C_i| : C_i \subseteq X \& C_i$ is either the leftmost or rightmost containment of $X\}$
For $|I(X)| = 1$,we set $I_s^*(X) = \{0, 0\}$

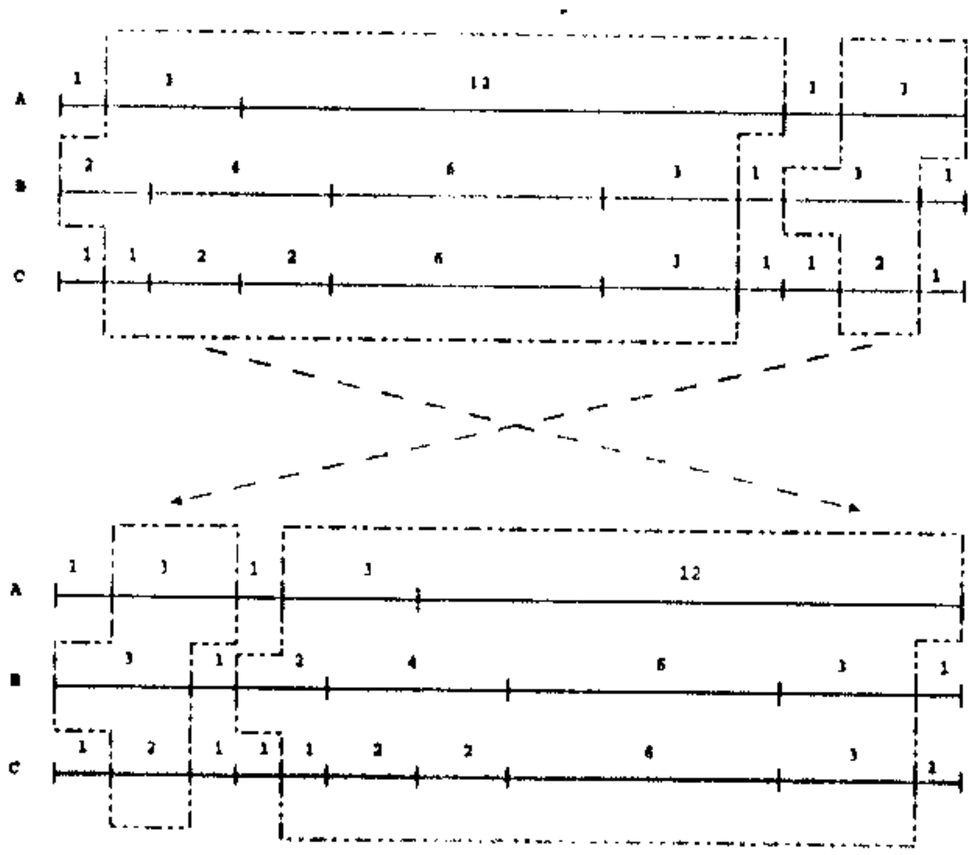

Figure 3.3: Cassette exchange

Now we can define the $I * D$ which is the characteristic of an equivalence class as follows.

$$I * D = (\{(I_s^*(A_i), I_s(A_i)) : A_i \in A\}, \{(I_s^*(B_j), I_s(B_j)) : B_j \in B\})$$

It is worth observing that cassette transformations do not change $I * D$ of a restriction map. Infact if the restriction map does not have coincident cut sites then, $[A, B]$ and $[A', B']$ belong to the same equivalence class if and only if $I * D[A, B] = I * D[A', B']$. For more general restriction maps with coincident cutsites the defintion of valid cassette has to be altered a bit to retain the validity of the statement in general. The required modifications will be described later.

3.4 Different techniques of solving DDP

The double digest problem is a typically well known problem in the Biocomputing domain and is attemted by different techniques from time to time. Due to the basic difficulty of DDP, we are not likely to obtain a polynomial time algorithm for it.Still, other NP-complete problems are approached by useful heuristic methods. In some cases the methods have great practical value. Three different approaches are described below.

3.4.1 Integer Programming

At the begining a few notations are needed. For a solution we assign double digest fragments $c_1, ..c_{|A\&B|}$ by $(c_1, .., c_{|A\&B|})E = (a_1, .., a_n)$, where E is a $|A\&B| \times n$ matrix of zeroes and ones.We have

$$a_i = \sum_{k=1}^{|A\&B|} C_k e_{ki}$$

. Similarly $(c_1, ..., c_{|A\&B|})F = (b_1, .., b_m)$, where F is a $|A\&B| \times m$ matrix of zeroes and ones. Obviously we wish to assign each C fragment once and only once to each single digest. Therefore we have the additional identities,

$$\sum_{j=1}^{n} e_{ij} = \sum_{k=1}^{m} f_{ik} = 1$$

Now DDP is solved by finding the consistent solutions of the two systems:

$$minimize\{\alpha + \beta\} \text{ where } \alpha, \beta \in I^+$$

$$-\alpha \leq a_i - (cE)_i \leq \alpha \text{ for all } i = 1, \ldots, n$$

,

$$e_{ij} \in \{0, 1\} \text{ for all } i, j$$

,

$$\sum_{k=1}^{n} e_{ik} = 1 \text{ for all } i = 1, \ldots, |A\&B|$$

,

$$-\beta \leq b_j - (cF)_j \leq \beta \text{ for all } j = 1, \ldots, m$$

,

$$f_{ij} \in \{0, 1\} \text{ for all } i, j$$

,

$$\sum_{k=1}^{m} f_{ik} = 1 \text{ for all } i = 1, \ldots, |A\&B|$$

This is a problem in integer linear programming and the packages available have not proved too useful for DDP.

3.4.2 Partition Problem

Another approach to DDP is to consider it as a complex, interrelated partition problem. Each a_i is a sum of disjoint c_k:

$a_1 = \sum_{k \in R_1} c_k$

.

.

$a_n = \sum_{k \in R_n} c_k$

where, $\bigcup R_i = \{1, 2, ..., |A\&B|\} and R_k \bigcap R_j = \phi, i \neq j.$

Likewise,

$b_1 = \sum_{k \in S_1} c_k$

.

.

$a_m = \sum_{k \in S_n} c_k$

where, $\bigcup S_i = \{1, 2, ..., |A\&B|\}$ and $S_i \bigcap S_j = \phi, i \neq j.$

Thus it is quite natural to think DDP as partition problem. In fact this approach is just a restatement of the integer linear programming formulation.

3.4.3 Travelling Salesman Problem

The TSP minimizes the cost of permutation of $(1, 2, \ldots, n)$. Problem DDP has two permutations, one of $(1, \ldots, n)$ and one of $(1, \ldots, m)$, but any computational scheme to minimize the salesman's tour might be adapted to DDP.The idea is to have two salesmen (named A and B) who work together to minimize the route cost. They both tour disjoint sets of cities, n cities (for A) and m cities (for B) and their reward is the goodness of fit in the double digest data C. Different useful heuristics are designed for these class of NP-complete problems. DDP is also attempted by Soft-Computing tools like Simulated Annealing etc. Here we attempt the problem with Genetic Algorithm for the first time. It is presented in the next chapter.

Chapter 4

Solving DDP by Genetic Algorithm

4.1 Basic Objective

First of all the structure of the solution of DDP is very complex and it is an NP-complete problem in general. Apart from that we have multiple objectives e.g.,

1. Partitioning the solution space with respect to cassette equivalence relation and get hold of one representative solution from each of the partitions.

2. Generate all equivalent solutions corresponding to each representative solution. Thus basically generating the entire solution space.

So, the accmulative effect of all these make it a very complicated affair. Naturally, the classical Genetic operators are to be properly modified to tackle such a peoblem. Though the structure of classical GA is also slightly modified to create provisions for multiple solutions, in essence it remains the same. In fact that is where basically the beauty of Genetic Algorithm lies. Its inherent simpicity can be restored even in dealing with very complicated optimization problem.

4.2 Structure of the algorithm

For DDP the Elitist model of Genetic algoriyhm is used here.Out of the four operations of genetic algorithm namely Natural Selection, Cross over,Mutation and Elitismm, other than Natural selection all the other three operations are modified to fit into the problem.

In the algorithm a set of randomly selected configurations is treated as the initial population. Then the set of configurations are subjected to *Natural selection, Order preseving weighted cross over, Partially deterministic compound mutation* and *Equivalence checker elitism* respectively to form the set of evoluted configurations of the next generation. Then again the new generation is treated as as the initial population and subjected to the sequence of genetic operations to form the next generation. In this way the whole process is repeated until some predetermined terminting criterion is attained. After termination of the process it is expected that one representative solution from each of the cassette equivalence classes are recorded. The structure of the algorithm is shown in Figure 4.1. The detailed description of the string representation, cost function and modified genetic operators are described in the subsequent sections.

4.3 String representation and Cost function

The string/configuration is represented as a doublet structure. Each doublet is consisting of one permutation of $(a_1, ..a_m)$ and one permutation of $(b_1, .., b_n)$. This is basically implemented by a structure of two arrays containing the two permutations. Let $C_s{}' = (c_1{}', ...c_q{}')$ is the superimposed fragment set in increasing order and $C_s = (c_1, ..., c_p)$ is the given set of fragments corresponding two both the enzymes in increasing order.Now the cost of configuration is defined as follows:

$$\zeta = \sum_{i=1}^{max(p,q)} (c_i - c_i')^2 \tag{4.1}$$

If index runs out of bound for some sequence the corresponding element will be assumed zero. It is very clear from the definition of the cost function that its value is always non negative and the cost function value is zero if and only if the configuration is a solution. This cost function enables us to identify the minimal configurations easily since its cost value is known to be zero unlike other optimization problem where the characterization of optimal strings are seldom known.

BEGIN ALGM_DDP
Make initial population at random.
WHILE NOT stop **DO**
BEGIN
Natural selection: Select parents from the population.
Order preserving weighted cross over: Produce children from the selected parents.
Partially deterministic compound mutation: Mutate the individual.
Equivalence checker elitism: Reserve the solution strings subject to cassette equivalence check.
Randomly choose the next generation population from the non-solution strings of the present and previous population.
END
Generate the partitions from the non-equivalen representative strings.
END ALGM_DDP.

Figure 4.1: Structure of the DDP algorithm

4.4 Description of the modified genetic operators

4.4.1 Natural Selection

This operation is kept unchanged with the traditional natural selection operation. Here strings are randomly chosen from the population with probability inversely proportional (as it is a minimization problem) to their costs as we discussed previously.

4.4.2 Order preserving weigted cross over

Modification over the traditional cross over operation is necessary due to the following reasons:

1. Here strings are doublet structure.

2. The constituent of the doublet are individually permutations of fragments. Traditional cross over might generate invalid strings.

3. There can be repetition of fragments in the constituent permutation depending on the fragment set corresponding to an enzyme.

So here the cross over is done in the following way:
Given two parent strings first a random number betwee zero and one is generated, if the number happens to be more than the cross over probability then the perent strings are considered eligible for cross over operation. Once the two strings are through with the cross over probaility check, a coin toss is done to decide which of the permutations of the doublet will participate in the cross over. Let p_1 and p_2 are thepermutations eligible for cross over. A random cut point is fixed leading to the generation of the four substrings $p_{1,1}, p_{1,2}, p_{2,1}, p_{2,2}$ then the children permutations are generated as follows, the left parts of the two children $c_{1,1}$ and $c_{2,1}$ are simply identical with p11 and p21 respectively.Now the right part of the first child $c_{1,2}$ is prepared by reordering the elements of $p_{1,2}$ in the order in which they occur in p_2 along with matching the repetitive weight. Similarly $c_{2,2}$ is prepared from $p_{2,2}$ with the weighted ordering of p_1.
For example, let the two eligible permutations are $p_1 = (1,3,2,1,3,4,2,2)$ and $p2 = (1,2,2,2,4,3,3,1)$ and let the randomly chosen cut point is 3. Then the four substrings generated $p_{1,1} = (1,3,2), p_{1,2} = (1,3,4,2,2), p_{2,1} = (1,2,2) and p_{2,2} = (2,4,3,3,1)$ then $c_{1,1} = p_{1,1}$ and $c_{2,1} = p_{2,1}$. Now we assign a two-tuple (e_i, w_i) for the ith element of $p_{1,2}$ for all i, where e_i is the fragment length and w_i is the count ofrepetition of the ith element in p_i in its position.So the tuples assigned for the elements of $p_{1,2}$ are $(1,2), (3,2), (4,1), (2,2), (2,3)$ respectively.In the same way if we assign tuple for elements of p_2, they will be $(1,1), (2,1), (2,2), (2,3), (4,1), (3,1), (3,2), (1,2)$. Acording to p_2 the ordering of the tuples of $p_{1,2}$ will be $(2,2), (2,3)(4,1), (3,2)$ and $(1,2)$. So the resulting $c_{1,2} = (2,2,4,3,1)$ and similrly $c_{2,2} = (3,1,3,4,2)$. Therefore, the two children generated are $c_1 = (1,3,2,2,2,4,3,1)$ and $c_2 = (1,2,2,3,1,3,4,2)$.

It is very clear from the description of the cross over that it would not generate any invalid permutation and therefore, no invalid doublet. In addition the cross over operation is carefully designed so as to explore a large amount of the search space efficiently.

4.4.3 Partially deterministic compound mutation

Given a string, here also a coin toss is needed to determine which permutation of the doublet is to be mutated. Once the eligible permutation is decided,the i^{th} element of the permutation is done in the following way for all i. A random number between zero and one is chosen. If this number is less than the permutation probability than the i^{th} position is mutated. For that a random index within the bound of the permutation j is generated and the i^{th} and j^{th} elements are swapped.

Apart from this a some what deterministic swap is also done. The minimum and the maximum costs of the generation are maintained. And depending upon the cost of the processing string it is ranked within the population. As it is obvious from the definion of the cost function that more is the distance of the unequal swaping fragments more will be the change in cost. So according to the rank of the working string in the generation a swaping distance is fixed in the unitary fashion. That if the rank is good then less distance swaping and if the rank is bad then more distence swaping. Then two random fragments having such position difference is swaped with the anticipation that it would generate low cost string.

The mutation operation so designed is having the property that from every string any string can be ganerated in one step with non zero probability. This condition is needed to ensure the uniform convergence of the elitist model of GA to optimal solution after infinite number of iterations [4].

4.4.4 Equivalence checker elitism

This is basically a process of preseving good strings across the generations. Though there exists characterization of the solution strings with respect to our designed cost function and thus a very simple way to identify them(by checking whether its cost is zero), we do not have any notion of good strings due to the structural complexity of the solutions of DDP. A string having a slight structural difference than some solution may have a very high cost. So, what is done instead is that whenever any solution string is generated within the population it is taken away. Then it is stored in some reserve space subject to cassette equivalence checking, with all other existing non-equivalent solution strings. Since we want to get hold of one representative string from each partition. The equivalence checking is performed by checking the I*D of each string as we explained in the previous chapter. After taking away the solution strings the next population is formed from the present population and

previous population by randomly choosing the non solution strings.

With this set of four genetic operations, it is expected that all non equivalent strings will be stored in the reserve space after sufficient number of iterations.

4.5 Generation of the whole partition from a representative string

We stated in the previous chapter that the very definition of a cassette needs little modification to generalize the characterization of cassette equivalence for configurations with coincident cut sites. As it is shown in Figure 4.2 the dotted portion can be reflected and in Figure 4.3 the dotted portions can be swapped with out changing the I*D. That is these operations allso give rise to valid strings in that partition. But with our usual definition of cassette these operations are not possible since the dotted portions are not atall a cassette. So we include these in our definition of cassette as follows.

For each pair i, j with $1 \leq i \leq j \leq l$ define,

$$I_C = \{C_k : C_i \leq C_k \leq C_j\}$$

which is the set of intervals from C_i to C_j. The cassette defined by I_C is the pair of sets of intervals (I_A, I_B), the sets of all blocks of A and B, respectively, that contain a block of I_c. In addition to this, if the left or right boundary of I_C happens to be a coincident cut site then smaller of the enzyme A fragment and enzyme fragment touching the coincident cut site externally will also be included in (I_A, I_B) to form possibility. In case of a tie both the frgments will be separately be included to (I_A, I_B) to form two more possibilities.

Clearly for a given $I_c = \{c_k : c_i \leq c_k \leq c_j\}$ the cassette is not unique now rather there can be as many as nine cassettes possible in the worst case for a given I_C.

Here we simply find the closure of the given representative string with respect to the cassette transformations (both cassette reflection and cassette swaping) and permutations of non border blocks uncut fragments. The algorithmic structure is shown in Figure 4.4.

4.6 Results

The designed algorithm is run on some artifical and practical data and the result obtained is shown in Table 4.1, with five different initial populations per instance and for each initial population, the number of trials is 10; and for each trial the number of iterations is 10,000.

Table 4.1: Results obtained by running the algorithm on different DDP instances

Problem Description	No of Equivalence Classes	Total no. of solutions
A = {1,2,2,3,3,4} B = {1,1,2,2,4,5} A& B = {1,1,1,1,1,2,2,3 3}	18	3210
A= {1,2,3,4,5,6,7,8,9 } B = {15,15,15 } A&B = {1 1 1 1 2 2 2 3 6 }	172	15840
A= {1,2,3,3,4,4,5,5} B = {1,2,3,3,3,7,8} A&B = {1,1,1,1,1,1,2,2,2,2,2,3,4,4}	393	36660
A= {5509,5626,6527,6766,7233,16841} B = {3526,4878.5643,5804,7421,21230 } A&B = {1120,1868,2564,2752,3240, 3526,3758,3775,4669,5509,15721}	1	2
A= {1,3,3,12 } B = {1,2,3,3,4,6} A&B = {1,1,1,1,2,2,2,3,6 }	18	208
A= {18,8,7} B = {12,7,5,3,2,1,1,1,1} A&B = {8,7,5,4,2,2,1,1,1,1,1}	65	930
A= {18,8,7} B = {9,5,5,5,4,2,2,1} A&B = {7,5,5,4,3,2,2,2,2,1}	52	686
A= {12,7,5,3,2,1,1,1,1} B = {9,5,5,5,4,2,2,1} A&B = {9,5,3,3,2,2,1,1,1,1,1,1,1,1,1}	376	92920

4.7 Conclusion

As the number of iteration of the genetic algorithm strictly depends on instance size and its structural complexity, it may happen for very large instances with very complicated structure that a few reprsentative strings are missed out. The only remidy is to increase the number of iterations. But once we get hold of all the representative strings the next part of generating the entire partitions and thereby the entire solution space is a perfectly deterministic algorithm(though a little bit costly) and therefore will certainly generate all the strings of the parcions. To conclude we add that we have run our algorithm on the examples with substantially large number of iterations and the observation in almost all the cases is that the process converges within 5,000 iterations. This fast convergence evaluates the power and efficiency of the designed genetic operators and the algorithm.

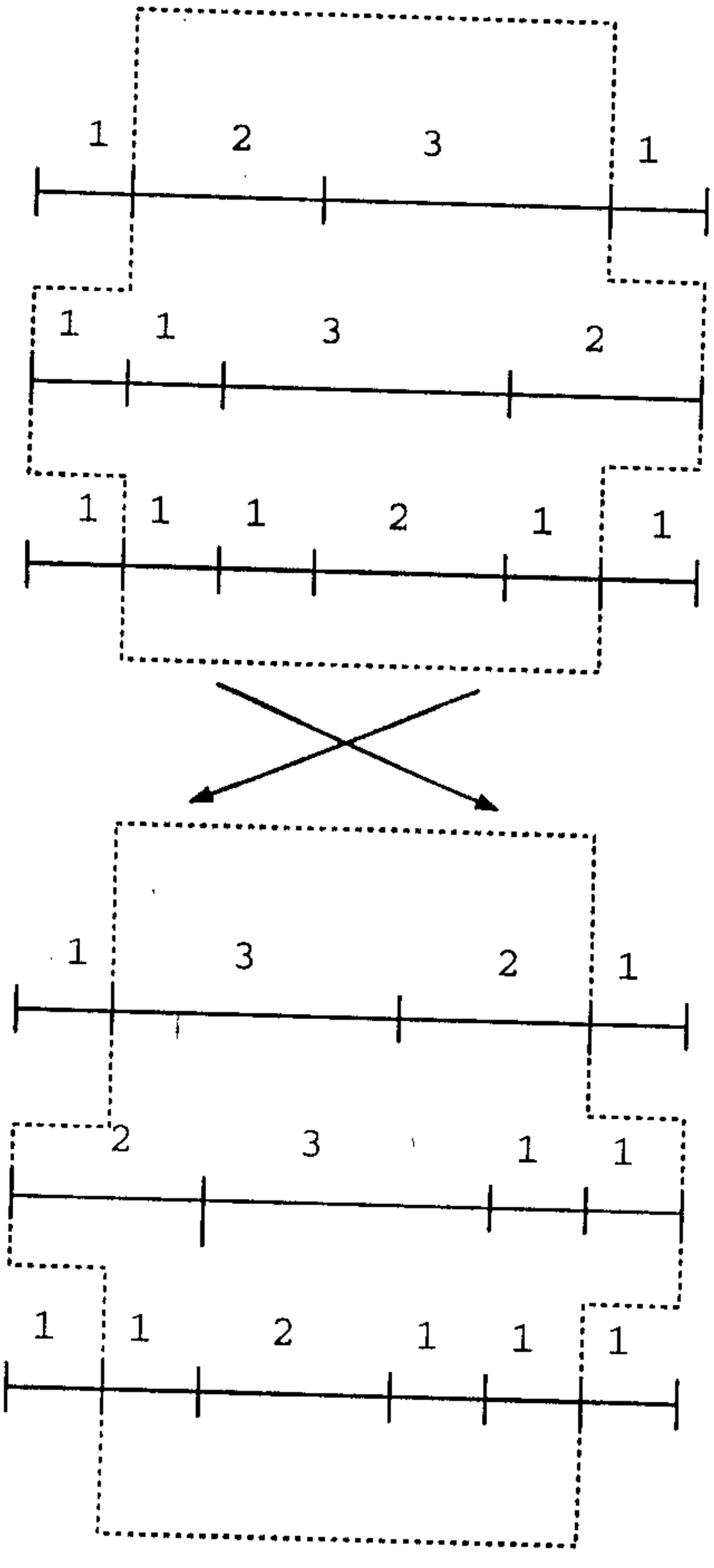

Figure 4.2: Non-minimal cassette reflection

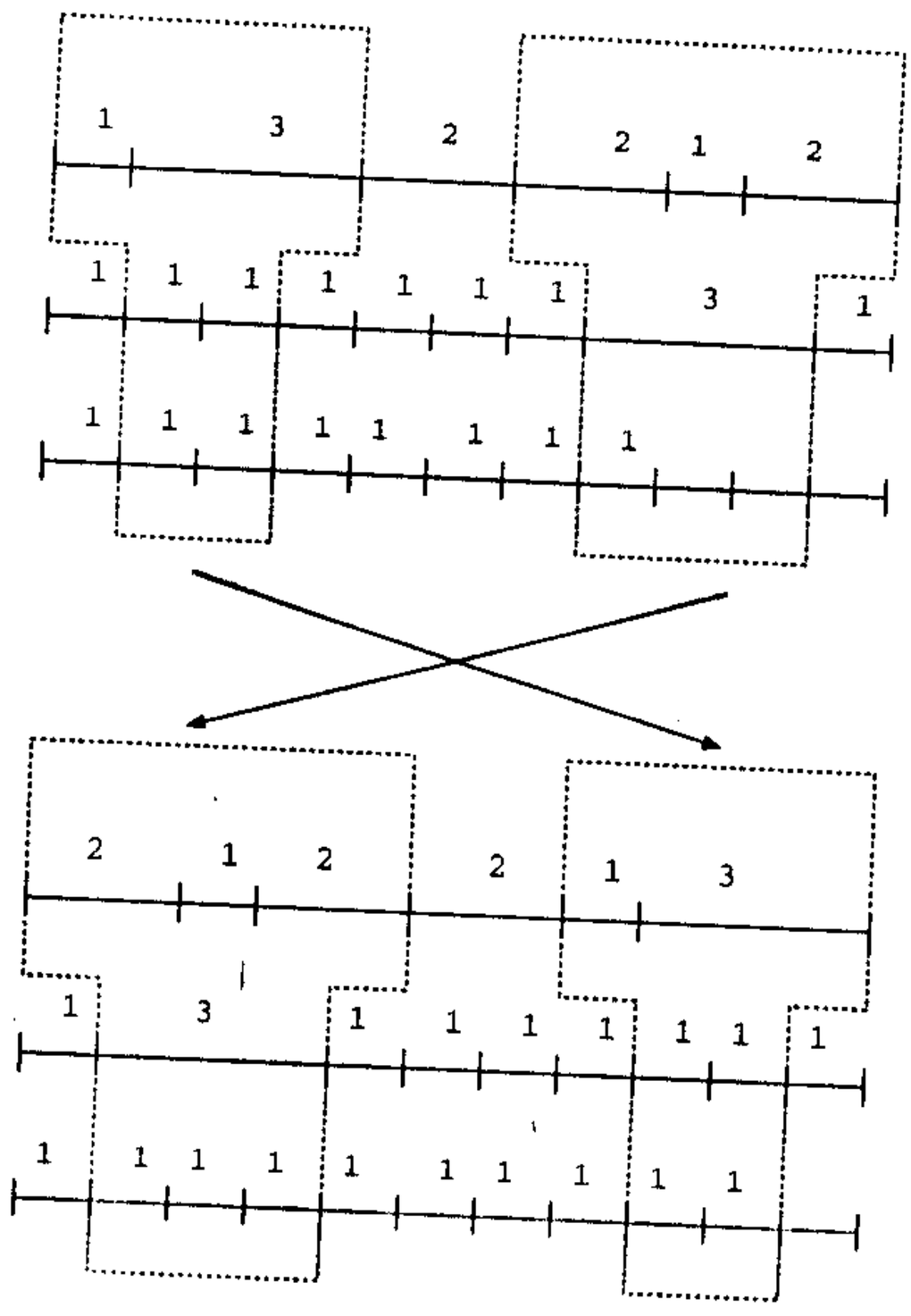

Figure 4.3: Non-minimal cassette exchange

```
proc GEN_PARTITION (string s)
begin
    closure(s) = s;
    while(there is a string in closure(s) yet to be operated on)
    do
        Let, a is such a string in closure(s) to be operated.
        while(there is scope of cassette transformations or permutation
        of uncut border block in a)
        do
            Let, b is a generated string from a by any of the operations.
            if(b does not belong to closure(s))
                closure(s) = closure(s)∪{b};
        od
    od
end
```

Figure 4.4: Algorithm for generating the partition from the representative string.

Bibliography

[1] Michael S. Waterman, *Introduction to Computational Biology: Maps, sequences and genomes*, Chapman & Hall, UK, 1995.

[2] D. E . Goldberg, *Genetic Algorithm in Search ,Optimization and Machine Learning*, Addison-Wesley Press, New York, 1989.

[3] Z Michalewich, "Genetic Algorithm + Data Structure = Evolution Programs", Springer-Verlag, New York, 1992.

[4] D. Bhandari, C. A. murthy, S. K. Pal, "Genetic Algorithm with Elitist Model and its Convergence", International Journal of Pattern Recognition and and Artificial Intelligence, vol. 10, no. 6, pp. 1147-1151, 1999.

[5] P. Larranaga, C.M.H. Kuijpers, R.H. Murga I. Inza and S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators", Department of Computer Science and Artificial Intelligence, University of the Basque Country, E-20080 Donostia - San Sebasti´an, Spain

[6] Bernd Freisleben and Peter Merz, "A Genetic Local search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problem", Department of Electrical Engineering and Computer Science (FB 12) University of Saigen.

[7] E. Lawler, J.K. Lenstra and D.B. Shmoys, "The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization", Wiley, 1985.