Agathoklis Georgiou

CS 531: Introduction to Bioinformatics

Project Title: Double Digest Problem

June 8, 2011

**Table of Contents:**

1. Introduction
2. More about the Problem
3. Motivation for a solution
4. Problem Fundamentals
5. Problem Formulation
6. More Details
7. Results
8. Conclusion
9. Sources
10. Introduction

In this project I will thoroughly discuss the Double Digest problem and provide a computer program which solves by a Brute Force and Branch and Bound approach. But before getting into the details of the solution I will briefly give a general definition of what the Double digest problem really is and why I chose to do this project.

Double Digest is the set of DNA fragments generated by the digestion of a DNA molecule by two special proteins called restriction enzymes. Mapping this approach requires the digestion of DNA by each of the two enzymes singly, and then by the two enzymes applied together in such a way so that fragments between successive sites are formed, thus the sequence of these fragments will usually be determined more easily. The fragment lengths can then be measured by other processes such as Gel Electrophoresis, Autoradiography and Fluorescence. Then, these fragments can be placed in the original order to obtain the whole sequence.

1. More about the Problem

In 1987 Goldstein and Waterman showed that the Double Digest problem is *NP* complete for DNA restriction mapping and is extremely hard or impossible to calculate in cases where the number of the DNA fragments generated by the two restriction enzymes is very large. Even though some approaches have been used, they have proved to be effective usually for small problems. Some of these approaches have been the Exhaustive Search, Simulated Annealing and Fragment Matching. The reason that this problem is so hard is due to its search space which grows as a factorial function (*x*!)(*y*!), where x and y are the lengths of DNA fragments generated by the two restriction enzymes.

This project will present an Exhaustive and Branch and bound search approach but will not go over simulated annealing. In both approaches, the initial steps are the same:

Let the two restriction enzymes be x and y. Also, let X1 = {a1, a2,…..,an} and X2= {b1, b2,…,bn} be the two sets of restriction cuts obtained when applying enzymes x and y on those sets respectively. We can confidently say that a set X’ = {c1, c2,….,cn} will contain the cuts which are obtained by applying enzymes x and y on sequences X1 and X2. Even though the order of the two sets X1 and X2 is arbitrary, by combining their restriction sites as if they were digested by a single enzyme we will obtain all the restriction cuts in X’. Figure 1 below shows an example of this process. Once we obtain the set X’ we will provide it as input to the two

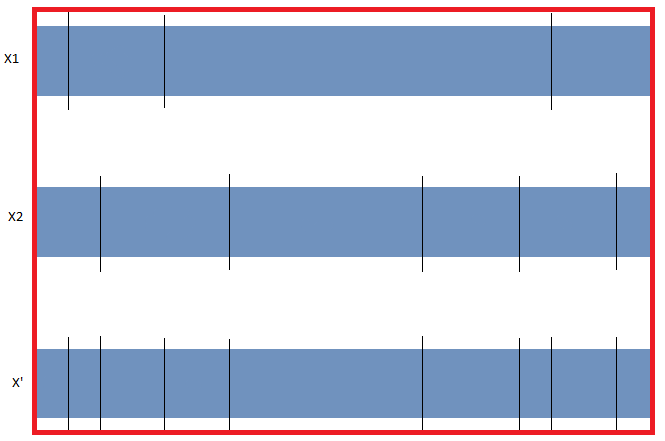


Figure 1: An example of combining the restriction sites of x and y into z

algorithms which will go through a process of various steps to generate the set of fragments from the combination of the two original restriction sites X1 and X2. Those steps will be discussed later in the paper.

The Double Digest problem is solved very often in molecular biology labs for purposes of restriction map construction for newly cloned DNA sequences and for applications in protein determination in proteomics using mass spectrometry, Recombinant DNA technology, cDNA/genomic library construction and especially in DNA mapping. But as mentioned earlier, the problem can be difficult to solve if the numbers of the generated fragments are large. As mentioned earlier, this problem has been intractable for very large numbers of fragments.

1. Motivation for a solution

Since I was very young I had a great interest on how life works at a very low level and at the same time, had a passion for technology. During my undergraduate degree I had enjoyed all my biology and chemistry courses but as my programming experience kept increasing rapidly, my knowledge of the biological and chemical world remained generally the same. I still maintained a personal hobby of learning more about biology, specifically at the cellular and molecular level but in most cases during an undergraduate degree students do not get to see the application of programming to solve intriguing problems in other sciences such as biology, chemistry and physics. This is probably due to the complexity of these problems.

Getting into the Masters program changed a lot of things including my opportunity to apply my programming experience in the fields that I always wanted to learn more about. And this Project is the best opportunity to do so.

I chose to do this project initially because I was very motivated by the lectures by Dr. Iyad Kanj and Jonathan Gemmel in the “Introduction to Bio Informatics” class. I was especially intrigued by the Partial and double digest problems we went over and how their application offers so much in the field of molecular biology and genetics. I had the chance to solve a Partial digest problem in class and in the midterm and was really interested in understanding the process deeper and being able to solve a more complicated problem. Additionally, I really wanted to test the brute force and branch and bound solutions and see how different these approaches are in terms of implementation and performance.

This project was the perfect opportunity to get a much better insight on the Double Digest problem and to have the chance to implement the two algorithms to solve it. After spending a reasonable amount of time and practicing many examples I became confident that I can solve this problem by the specifications provided. Yet, before proceeding in showing specifics of the programming nature of the two algorithmic approaches I will lay down some fundamentals of the problem, such as how the input is represented in terms of restriction enzymes, fragments and multi-sets. These inputs will be treated as sets, some unique, some not, and they will be passed to the algorithms which will proceed in providing with the answers.

1. Problem Fundamentals

The problem begins with two sets which contain the location of all cuts in the restriction map, including the start and end. We will name those sets X1 and X2. From these sets we can calculate the fragments but that will be done later on. Joining these sets will result into a new set X’ which contains all the unique cuts between them. X’ is the new restriction map which will be used later on to produce the multi-set DX’ and compare against the final solution X’’ to check if the fragments are the same between them.

n will be the number of cuts for X1, X2, X’ and for every set X’’ which is generated by the algorithms. This number is extremely important as it defines the number of cuts which exist in the solution. Both X’ and X’’ must have the same value for n or else the solution will be erroneous.

DX’ will be the multi-set of integers representing all possible lengths of each of the restriction cuts in every X we compute from a double digest. Each new DX’ computed will be compared against the original DX. If they are not equal then we will attempt to compute a different X and another DX from it until a solution is found. Each of the algorithms handles this procedure in an entirely different approach. The brute force approach will attempt to find all possible combinations of X’ sequences and generate a new DX’ from them, while the Branch and bound approach will backtrack and try a different combination early on. The two algorithms vary extremely in terms of the time and complexity it takes to find the solution.

1. Problem Formulation

The problem can be separated into a sequence of imperative steps:

1. Two inputs X1, X2 are received.
2. X1, X2 are combined to generate X’.
3. X’ is used to generate DX’.
4. DX’ is used as input to both algorithms to produce a new X’’.
5. The fragments of X’’ are generated by each algorithm and compared against X’.
6. Program decides if a solution is found and moves to step 7 or generates a new X’ and repeats steps 3-6.
7. Program Terminates.

Figure 2 below shows how methods of this class interact to achieve these steps:

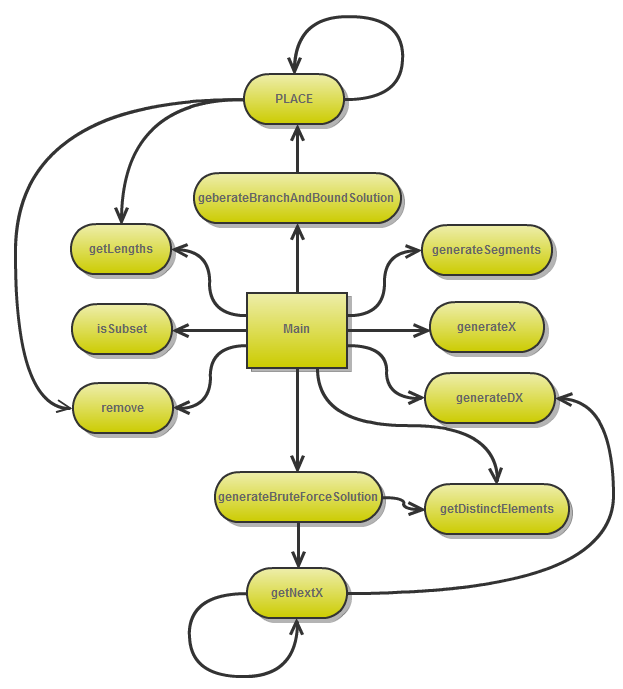
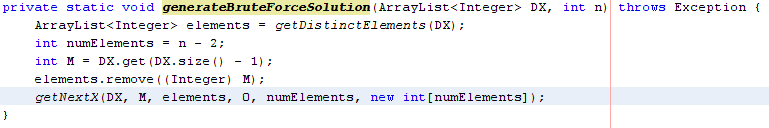


Figure 2: interaction of methods involved in the project.

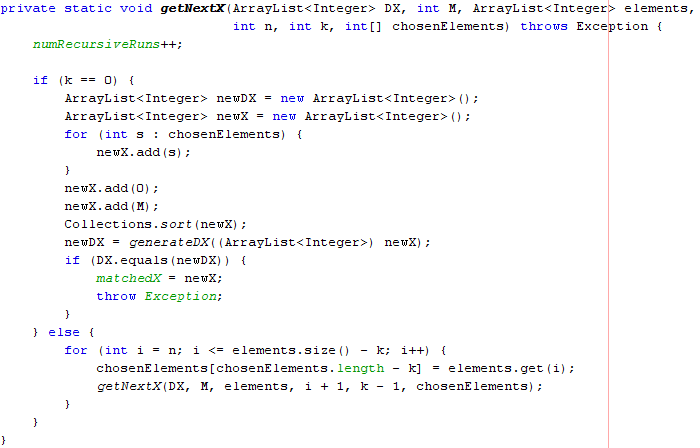
Figure 2 shows that the main method calls the majority of methods which are meant to provide utilities such as computing the X and DX, performing various operations such as removing elements from a list, computing the lengths of fragments, checking if a list is a subset of a multi-set, generating distinct elements from a list and generating fragments. The two problem solving algorithms are included in generateBrandchAndBoundSolution() and generateBruteforceSolution. Each of these two methods calls an additional recursive method which ultimately solves the problem. Of course, the recursive methods have a completely different structure and complexity since they solve the problem in a different approach. The next section gives a thorough demonstration of how these two methods are structured and how they compare to each other in terms of running time and efficiency.

1. More Details

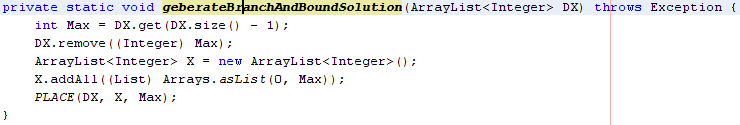
The generateBruteForceSolution() method below prepares the program for the first call of getNext(). From then on, getNext() will run recursively until a solution is found. Notice that the DX, the Max number, another integer starting from 0, the number of elements and a new array which will hold the solution, are passed.



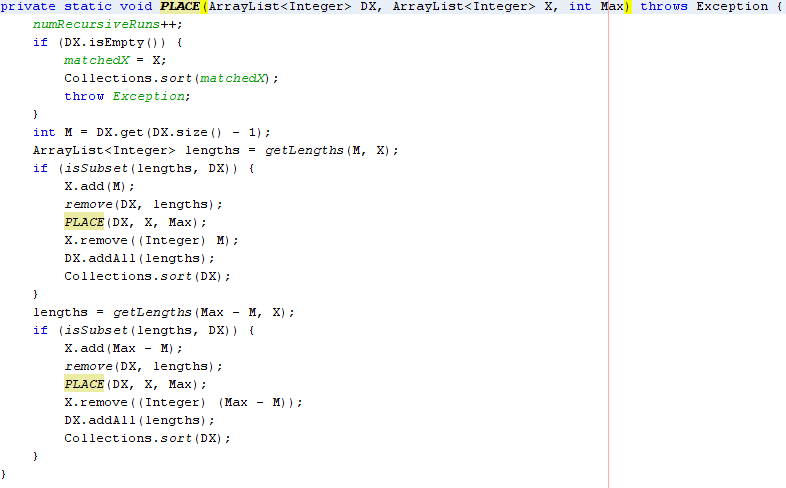
Once k equals 0 it means that the program has acquired a sequence which may be a solution. 0 and the current Max number will be added, along with all the elements of the list chosenElements. From that sequence we compute the DX and compare it to our original DX. If they are the same then a solution is found, an Exception is thrown and all recursive calls end. If not, the recursion continues in computing combinations exhaustively, which renders this approach, extremely ineffective and time consuming.



Just like the Brute Force method, generateBranchAndBound() will prepare the program for the first recursive call. A noticeable difference is that this method requires fewer arguments to be passed on. At this point there aren’t many other differences but the next method PLACE() is where the Branch and Bound approach really shines.



If the DX is empty then we have found a solution. No more calls are required and an exception is thrown to terminate all recursive calls. No generation of a new DX is required. Already, even when a solution is found we the program can stop without performing any more computations. If a solution is not found it attempts to find a new Maximum value from DX and compute the digest of that number in the set X. Then it undoes these operations and recurse. The same happens in the following code with the difference that the program generates the digest of the Max number minus the current max. If an invalid digest is computed the program will fall back to the last valid solution and continue.



1. Results

After running both algorithms multiple times with a variety of List sizes I recorded some data shown in Figure 3 below:

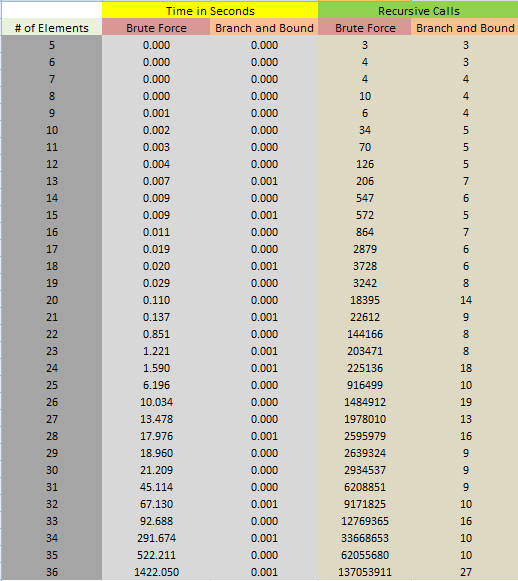


Figure 3: Collection of data from 36 runs of Distinct List sizes

Figures 4 and 5 show graphs which correspond to the data from Figure 3:

Figure 4: Time elapsed over List size.

Figure 5: Number of recursive calls over List size

Figure 4 which shows time elapsed and Figure 5 which shows the number of recursive calls for each method definitely justify the data recorded in Figure 3. After a certain number of List sizes it would take a great amount of time for the brute force algorithm to complete. Thus, the List sizes in the sample data go from 5 to 36. For set sizes less than 5, the time to completion and number of recursive calls is the same for both algorithms.

The complexity of both algorithms is clearly shown. The brute force approach solves the problem in exponential time and the branch and bound in logarithmic, both in terms of elapsed time and number of recursive calls. Clearly, there is absolutely no reason why one would use the Brute force approach for solving such a problem, especially when algorithms such as Branch and Bound solve it in milliseconds…

1. Conclusion

Throughout this paper I have demonstrated two solutions to the Double Digest Problem. After giving a general definition of the problem and expressing why I would like to work on it I have presented two algorithms which successfully provide a solution. One is by a Brute force approach and the other is by Branch and Bound. The data presented in the table and graphs describes the nature of these two approaches precisely. The overall experience of implementing these algorithms has been very fulfilling and I definitely learned a lot more on this problem. The doubleDigest.java file that I provide along with this paper will serve as a tool which can be tested against any number of cases.

1. Sources
2. Yin Zhang and Zhijun Wu. (2007). Solving Large Double Digestion Problems for DNA Restriction Mapping by Using Branch-and-Bound Integer Linear Programming, 1-13.
3. <http://en.wikipedia.org/wiki/Restriction_map>
4. <http://de.wikipedia.org/wiki/Double_Digest_Problem>