String Barcoding: Uncovering Optimal Virus Signatures

Sam Rash
University of California at Davis
1 Shields Avenue
Davis, CA 95616
1-530-752-7004
rash3@llnl.gov

Dan Gusfield
University of California at Davis
1 Shields Avenue
Davis, CA 95616
1-530-752-7004
gusfield@cs.ucdavis.edu

ABSTRACT

There are many critical situations when one needs to rapidly identify an unidentified pathogen from among a given set of previously sequenced pathogens. DNA or RNA hybridization chips can be designed for such identifications. Each cell in the chip can report the presence or absence of a specific substring of DNA in the unidentified pathogen. Properly designed, the collection of reports obtained from the cells can uniquely identify any pathogen in the set, or determine that the unidentified pathogen is not in the set. There is a limit to the number of cells on a chip, and a range of substring lengths that a cell can handle. So, given the full sequences of a set of pathogens, the problem is to design the chip by selecting the smallest set of substrings of the appropriate lengths, so that each pathogen in the set has a unique set of cells that report a substring. For any given pathogen, the set of reporting cells is its signature, and hence the entire system is a "barcode" system for the pathogens.

Previous work addressed this problem [1], but focused on pathogens of bacterial size, and hence had to make many compromises for the sake of efficiency. The substrings lengths were severely restricted, and no optimality or near-optimality was guaranteed. In this paper, we focus on viral-size pathogens. We show that for genomes of this size, it is practical to solve the barcode design problem optimally, or near-optimally, without artificially constraining the problem. We also efficiently find barcodes that provide a level of redundancy, tolerating a number of errors or mutations. The key technical ideas are the use of suffix trees to identify the critical substrings, integer-linear programming (ILP) to express the minimization problem, and a simple idea that dramatically reduces the size of the ILP, allowing it to be solved efficiently by the commercial ILP solver CPLEX. We report extensive tests of our approach on various collections of virus DNA and RNA sequences.

Categories and Subject Descriptors

J.3 [**Life and Medical Sciences**] - Biology and genetics

General Terms

Algorithms and Performance.

Keywords

String barcoding, barcoding, virus signatures, testing set, suffix trees.

1. INTRODUCTION

The string barcoding problem is a problem useful in identifying an unknown string as one of a set of known strings. In particular, consider the case where we have a database of strings. We are then given an unknown string and wish to know which of the strings in our database the unknown one is most similar to. In the biological realm this may be the case where we have a database of the DNA for known viruses and wish to identify an unknown virus. If we have access to the entire DNA sequence for the unknown virus, we can simply use known similarity searching programs such as BLAST[2] to identify the unknown virus. In reality, it is not this simple, however. We do not have access to the entire sequence for the unknown virus (string). Instead, we can only *test* for the presence of some particular substring in the unknown string and get a "yes" or "no". These can simply be called *substring tests*. In this case, it is still possible to identify the unknown virus. What we need is a set of substring tests such that on every virus in the known set, the set of answers (yes or no) that we receive is unique with respect to any other virus in the known set. Then, if we have an unknown virus, we perform the entire set of tests and compare the answers we receive with the answers for every known virus.

In the realm of biology, however, each substring test carries a cost. Due to this cost, we would like to minimize the number of tests. This is the purpose of the string barcoding problem presented in this thesis. We present an approach that starts with a set of known strings (viruses) and builds a minimum cardinality set of substrings that allow us to identify an unknown string using substring tests.

We present a method that uses suffix trees [3] to reduce all forms of the string barcoding problem to an Integer Linear Program (ILP). The implementation presented includes many features that make it a practical and realistic package. This includes features such as length constraints and error tolerance measures to accommodate mutations.

2. BACKGROUND

Just prior to completion of this paper, another paper was published on a problem that is essentially the same as the string barcoding problem. Borneman et al [1] discuss what they call probe selection algorithms. The key difference in their problem formulation is that they can precisely control the set of candidate substrings (probes) that they wish to use for tests in order to build a barcode. They typically do this by limiting the length of all substrings to exactly one length (8, for example). In the approach here, we do not use any such prior limitations on the data. We *allow* length constraints as the biological model requires, but they are not necessary in order to find a solution.

In short, where they control the computational complexity by restricting the set of candidate probes/substrings, we keep the complexity practical by using suffix trees and other methods to reduce the size of the ILP. The key difference between our method and theirs is that our reductions in the problem size do not change the optimal solution obtained whereas their approach may exclude the optimal solution. Another key point is that while their paper only mentions that using a redundancy parameter for robustness would be a good idea, our implementation already uses this parameter (discussed later).

One comment from the paper also stands out. Borneman et al [1] write, "Furthermore, even if we did have complete sequences of these clones, computing optimal probe sets for large data sets is computationally infeasible." Since we have actually solved the problem on fairly large datasets, the validity of their statement is called into question. However, the statement is quite ambiguous. While the number of strings in some of their datasets is higher than ours, they do not mention the lengths of the strings within those datasets. Since the complexity of our approach is based on both parameters, it is difficult to assess the validity of their statement with respect to our results. Nevertheless, what can be ascertained is that their statement helps make the point that our reductions using suffix trees and other techniques are sufficient to transform what appears to be an unsolvable problem into one that is solvable in practice. Lastly, their NP-hard result does not have an implication for the problem presented here.

The Chemical and Biological National Security Program (CBNSP) headed by Tom Slezak at Lawerence Livermore National Laboratory (LLNL) has also been doing work similar to DNA barcoding. They key difference between the string barcoding problem and CBNSP's goal is that they do not aim to find a barcode for every string in a set. Rather, they may input 500 viruses and wish to have signatures for only 10 of those viruses. This change alone typically simplifies the complexity of the problem in our approach.

Thus far, the approach at CBNSP has been to use suffix trees to find the minimum number of relevant substrings and then test each of those as a signature using BLAST and a BLAST database of all the viruses they are concerned with. When they have multiple genomes of a single virus that they are interested in, they have used MSA (Multiple Sequence Alignment) to expedite the process. The MSA technique allows them to quickly find conserved regions in the genomes of interest which can be used as candidate primers. It should be noted that while this technique is a sufficient method, it may miss many good signatures due to the way the global alignment of the sequences may turn out using MSA.

3. PROBLEM DEFINITION AND COMPLEXITY

We will be given a set of m strings (virus genomes), $S = \{s_1, s_2, ..., s_m\}$. The goal is to find a set of substrings S' such that for any pair of strings $s_i, s_j \in S$, there is at least one substring $s' \in S'$ such that s' is a substring of s_i or s_j, but not both. We say S' *distinguishes* S if it has this property. We then wish to find the minimum cardinality set S'. We call the set S' the *testing set*. A substring that is in the testing set may also be referred to as a *signature substring*. Also, a finite alphabet is assumed.

Once we have found an S', we can associate a binary vector of size $|S'|$ with each string in S. In the vector for $s \in S$, there is a 1 in position i if and only if $s'_i \in S'$ is a substring of s. Otherwise, position i has a 0. We call this vector the *barcode* for string s. We also consider all the positions in the barcode for s that have 1's. Take the substrings corresponding to the 1's. This set of strings defines the *signature* for s. Note that no other string will have exactly that set of substrings present since the barcode for s is unique. Obvious upper and lower bounds of $\lceil log_2 m \rceil$ and m exist on the size of S' as well.

The problem of general barcoding, a close cousin to string barcoding, is NP-hard [4]. Garey and Johnson refer to this problem as the *minimal testing set* problem and state that this is an NP-hard problem. It is unknown whether or not the unconstrained, basic string barcoding problem is NP-complete or not. However, if we add one more parameter k to the problem and instead ask what the minimum cardinality set S' is such that that each $s' \in S'$ is length k or shorter, the problem is NP-complete. We refer to this variant of the problem as the *max-length string barcoding problem*.

• ***Theorem 1.1***: The max-length string barcoding problem with an alphabet of size at least 3 is NP-complete.

Additionally, the ILP generated has properties such that it cannot be approximated to a within a factor of $n^{1-\varepsilon}$ for any $\varepsilon > 0$ [5].

4. ILP IMPLEMENTATION

4.1 Basic Implementation

The method in which we solve this problem is by reducing it to an integer linear program. Naively, we could enumerate all possible distinct substrings from the original set of strings. We next create a variable for each substring in the integer linear program. We have an equation for each pair of strings in the original set. We put the variable for a substring s in an equation for a pair of strings S_1, S_2 if s appears in exactly one of those strings. For example, let S_1 and S_2 be strings and $s_1, ..., s_k$ be the set of all substrings present in exactly one of S_1 or S_2. Then each v_i is a variable for each s_i then the equation for that pair would look like:

$$v_1 + ... + v_k \geq 1 \qquad (1)$$

We have an equation for every pair. The objective function for the constructed ILP will be to minimize the sum of the v_i's. It should be clear that if we find a solution to this system of equations, we can take the substrings for variables that are set to

1. If there exists a substring for a pair of strings that is in exactly one of the strings, then it will have a variable in the equation for that pair. If there is a variable in the equation for a pair, then there must be a substring that exists in exactly one of the pair. This creates a one-to-one correspondence between substrings that can differentiate a pair of strings and variables that can satisfy an equation for a pair. This creates the necessary mapping. Furthermore, since we minimize the sum of the v_i's, this in turn minimizes the number of substrings in S' and hence the optimal solution to the ILP is indeed the optimal solution to the string barcoding instance used to construct the ILP.

The complexity of this problem primarily depends on the number of variables and the number of constraints in the ILP. The major approach taken in this project is to reduce the number of variables necessary in the integer linear program by using suffix trees on the original set of strings. The reader should consult [3] for the details of suffix trees.

As discussed in [3], the suffix tree encodes the information about each substring that we need. In a suffix tree for a set of strings, each leaf is labeled to indicate which string it occurs in. For example, if a leaf is labeled j, this means that the suffix spelled out on the walk from the root to this leaf is a suffix of string j. Furthermore, we also know that the string created from a walk on the root ending at any point on this path is a substring of string j. If we choose substrings based only on what pairs they differentiate, then we need only choose one string per root to node walk. The next point is a key point, if not *the* key point with respect to the advantages of using suffix trees: The string used for a given root to node walk can be any string that starts at the root and ends on the edge into the node (including at least 1 character from the edge). This is so because all of the strings that can be generated for a given root to node walk will occur in exactly the same set of original strings and hence distinguish the same pairs of original strings. For simplicity's sake, this is assumed to be the substring including the entire label on the edge unless otherwise stated.

Now, if we consider each node, it has a set of leaves below it. This means the substring spelled out by the walk from the root to this node is present in the full strings that label the leaves below it. Then, for each node v, we can find all strings that have leaves below it and then we know the substring on the walk to v is present in all those strings. We can immediately extrapolate this data to say that the string on the walk from the root to v is present in all strings with nodes below it and not in any other and hence can distinguish all the relevant pairs. Then, the string barcoding problem can be phrased in terms of suffix trees. That is, find a minimum cardinality set of nodes such that in the suffix tree described above, for every pair of strings i,j at least one node v in S' has a leaf below it labeled i (or j) but not j (or i). We must discuss one more detail about the use of suffix trees and then an example will be presented in order to clarify the whole process.

Since it is important for efficiency to reduce the number of variables in the ILP, we present one more immediate optimization to reduce the number of number of variables. We consider that each node in the suffix tree has a binary vector associated with it. The vector is of size n where n is the number of strings in S. Position i in this vector has a 1 if a leaf below has s_i in its label. Then, if two nodes have the same binary vector, we only need one variable for those two nodes. To decide which string to associate with the variable, we can either use some secondary criteria or arbitrarily choose one. Since both distinguish the same set of pairs of strings, we need only use one. In general, we only need one variable for each unique binary vector that is present. We say that two nodes are *distinct* in this context if they have different binary vectors associated with them. In the current implementation, the criteria is arbitrary since all other filters are done prior to this duplicate removal. In our data, we saw reductions in the number of variables of around 75% or even up to 90% in some data.

4.2 Example: from strings to ILP

In order to clarify the process of using suffix trees, we present a small example. As input we will use the three strings *cagtgc*, *cagttc*, and *catgga*. We label these strings 1, 2, and 3 respectively. The reader should refer to figures 1.1-1.5 as details are mentioned.

If we use the most naive approach and enumerate all distinct substrings, we have 41 distinct substrings. However, using the suffix tree alone reduces the number of substrings to 24. In figure 1.2, nodes are labeled with subsets instead of binary vectors in order make the example easier to follow. Note that it is easy to go from the subset to a binary vector. If we examine figure 1.2, we will find that only 6 distinct subsets label nodes. Furthermore, any substring that occurs in all strings cannot be used to differentiate any pair. This means that any node labeled with all strings is not useful, so we end up with 5 candidate signature substrings. This is a further saving over the 24 nodes in the suffix tree. Compared to the most naive approach which contained 41 substrings, we now only have 5. This is just over 12% of the original number of substrings.

Figure 1.3 shows the actual input to CPLEX. In order, the input is the objective function, a list of the ILP constraints, and the objective function. The variable numbers correspond to the node number. For example. node *v18* appears in the ILP as *X18*. Also note that the theoretical minimum is included to accelerate the search (CPLEX can do more pruning).

If we solve the ILP in figure 1.3, we find that an optimal solution of value 2 by setting X18 and X22 to 1 and all other variables to 0. This corresponds to using nodes *v18* and *v22*. Hence, we examine the suffix tree in figure 1.1 and find that this means using the strings *tg* and *atgga*. The following two figures illustrate the process.

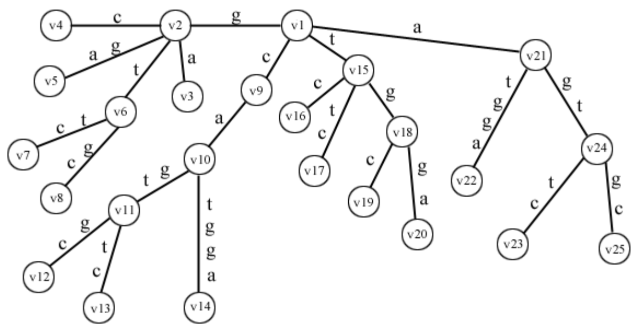

Figure 1.1 - suffix tree for set of strings *cagtgc*, *cagttc*, and *catgga*

v1 - {1,2,3}	*v2* - {1,2,3}	*v3* - {3}	*v4* - {1}	*v5* - {3}
v6 - {1,2}	*v7* - {2}	*v8* - {1}	*v9* - {1,2,3}	*v10* - {1,2,3}
v11 - {1,2}	*v12* - {1}	*v13* - {2}	*v14* - {3}	*v15* - {1,2,3}
v16 - {2}	*v17* - {2}	*v18* - {1,3}	*v19* - {1}	*v20* - {3}
v21 - {1,2,3}	*v22* - {3}	*v23* - {2}	*v24* - {1,2}	*v25* - {1}

Figure 1.2 - table of string labels for each node in suffix tree from figure 1.1

```
minimize
X18 + X22 + X11 + X17 + X8          #objective function
st
X18 + X22 + X11 + X17 + X8 >= 2  #this is the theoretical  minimum
X18 + X17 + X8 >= 1               #constraint to cover pair 1,2
X22 + X11 + X8 >= 1               #constraint to cover pair 1,3
X18 + X22 + X11 + X17 >= 1        #constraint to cover pair 2,3
binaries                          #all variables are 0/1
X18  X22  X11  X17  X8
end
```

Figure 1.3 - ILP constructed for suffix tree in figure 1.1 using no additional constraints (length, etc)

cagtgc	1	0
cagttc	0	0
catgga	1	1

Figure 1.4 - barcodes

cagtgc	{"tg"}
cagttc	∅
catgga	{"tg", "atgga"}

Figure 1.5 - signatures

4.3 Extensions to Basic Implementation

We implement length constraints by filtering out strings that fall outside the valid length range. This further reduces the number of variables in the ILP.

The next set of features deal with the problem of errors that occur when testing an unknown sequence. Errors in this problem typically refer to the case where a signature is first developed for a virus. The virus then mutates in nature and we wish to still be able to identify the mutated version. We implement two methods for dealing with this problem.

In order to deal with (point) mutations and sequencing errors that occur, we have implemented a method of redundancy. Instead of requiring a single substring for every pair of strings, we require *r* substrings for each. This results in the equation of the form

$$v_1 + \ldots + v_k \geq r \qquad (2)$$

Solving this modified ILP will result in having *r* substrings to differentiate each pair of strings that gives a larger barcode than if a smaller redundancy is used.

The analysis of this approach is omitted due to space limitations. However, the conclusion from it is that using low values such as *r=5* result in a high confidence that a signature will remain valid given reasonably high mutation rates, short generation times such as those observed in viruses, and a reasonable period of time (1 year).

The second method for error tolerance is to impose a minimum edit distance between substrings that are present in the testing set, *S'*. An edit distance between a pair of strings is simply a way to measure how many unit operations on one string are necessary to change it into the other (it is a symmetric value). In the world of DNA, we can essentially consider these as mutations (point mutations, insertions, deletions). Enforcing a minimum edit distance lowers the likelihood that a small number of mutations will cause a chosen substring to appear in a virus that it did not previously appear in. For a discussion of edit distances and how to compute them, we refer the reader to [3]. The change to the ILP is that for each pair of candidate substrings s_i, s_j (with v_i,v_j being the respective variables) that are not at least the minimum edit apart, we add a constraint to the ILP of the form

$$v_i + v_j \leq 1 \qquad (3)$$

This enforces that at most one of the substrings will be in *S'*. Again, the analysis is omitted for brevity. The result of the analysis, however, again shows that using small edit distances such as two or four results in added confidence that mutations in nature will not invalidate a signature.

As in the case of CBNSP, it is sometimes the situation that we may have a set of organisms and wish to find signatures for only a subset of the organisms. Our implementation includes this feature. We can control what pairs are distinguished by adding or removing the appropriate equation of the form (1) or (2) in the ILP.

5. RESULTS

We used CPLEX version 7.100 with its default parameters. CPLEX was run on a Compaq AlphaServer 6/525 DEC21000. It had 8 cpus each running at 523mhz with a 4 mb cache.. However, only one cpu was used per execution of CPLEX (no parallelism was exploited). The machine had 4 gigabytes of physical memory.

5.1 Input Parameters and Output Measures

The input parameters measured include the properties of the data and the parameters fed to the program. The data properties include the size of the problem (number of strings and lengths of strings) and the average edit distance between pairs of string. The edit distance is normalized to be edit operations per character. The program parameters include the set of viruses within the input set that we actually want signatures for, the minimum edit distance required between signatures, and redundancy of coverage. This is how many substrings we want for each pair of original strings that can be used to determine which one of the original pair an unknown string potentially is.

CPLEX bounds how close a solution is to the optimal is by a gap measurement. Since CPLEX is a branch and bound algorithm, we note that *best node* refers to the best solution value to the LP relaxation at any queued (unexplored) node. The value of *best integer* is the best solution value obtained thus far. Then, CPLEX defines *gap = abs(best integer - best node)/(1e-10 + abs(best integer))*. Note that the gap is a bound, of course, and that a solution that has a non-zero gap may indeed be optimal, but just has not been proven by completely searching the solution tree. The primary output measurements we took were the time to reach a 25% gap and the gap measurement after 4 hours of execution. A 25% gap was chosen as this would guarantee a solution that was at most 33% larger than the optimal. An execution time of 4 hours was chosen as it seemed to be a reasonable amount of time to devote to building signatures for a set of viruses.

5.2 Data Collection Methods

Timing data was collected on a variety of datasets that were generated by randomly sampling a set of viruses obtained from Genbank. For each set of parameters, a set of 15-25 files with similar input data properties were run through the pipeline and averages were computed. The input files to the r-runs, s-runs, and len-run were sampled from the same master file. The hiv-runs used another set that used strains of the HIV virus. Tables 1.1 and 1.2 summarize the results on the various datasets (discussed shortly).

5.3 Data Used

All data were viral sequences taken from Genbank. We essentially had two master datasets. The first was a random selection of 10,462 viruses from Genbank. The second was a selection of 4,548 HIV distinct sequences that were different HIV strains. The first dataset was the master data file that the input for the r-runs, s-runs, and len-runs. The HIV master file was used to generate input files for the hiv-runs. The difference between these two master datasets is the average edit distance per character is lower in the HIV dataset. This was the goal as the HIV set was used in order to test if the similarity of the viruses in an input set would have a noticeable effect on the running time or solution size.

The final data set was one from Lawrence Livermore National Laboratory. The data here was a file containing 536 viruses. Included in these files were 19 distinct sequences for Venezuelan equine encephalitis (VEE) and 4 versions of Variola (smallpox). The goal on this dataset was to obtain a signature for each of the two sets of interest (one for VEE and one for smallpox).

5.4 Results by Dataset

Our baseline was a single run on a dataset from the Genbank master file in which no constraints were applied to data. This is the hardest case to solve and 4 hours was not enough for CPLEX to even find a single integer feasible solution using its default behavior. It is omitted from the table as no real data could be collected. However, in this case it is possible to use set cover approximations to achieve much faster run times and an answer that is within a logarithmic factor of the optimal. While the guaranteed factor here may not be acceptable in large datasets, the empirical results show that the theoretical minimum is often obtained.

In all subsequent cases of data sizes and program parameters, we obtained a practical solution in a reasonable amount of time. The rest of the results section is spent examining what parameters have the largest effect on execution time. The complete results are in tables 1.1 and 1.2.

The first set of data we will examine is the r-runs. We discuss the most notable results here. These datasets show that increasing redundancy decreases the running time. This is a big plus as increased redundancy also makes a signature more robust and error tolerant (sec 4.7). The other interesting item with respect to increasing redundancy is that on our datasets, using a redundancy of 5 resulted in slower times than using 2, but using 10 outperformed both cases. This means that execution time is not monotonically decreasing as we increase redundancy. Figures 3.1 and 3.2 illustrate this graphically.

At this time, we have one plausible explanation for why a redundancy level of 5 was worse than other levels tested, both higher and lower. Consider a single equation of the form in equation (2) and assume around 10 variables in it. Since 10 choose 10 and 10 choose 2 are smaller numbers than 10 choose 5, there are fewer feasible ways to satisfy equation (2). This results in fewer branches to explore in the solution tree for CPLEX. If this is the case, then the redundancy pattern that resulted in the minimal computation time would depend on the number of variables in the ILP and how many variables exist in the constraint for each pair. Still, the main conclusion is that increasing the redundancy of coverage decreases computation time drastically.

The len1 run tested whether restricting the length of candidate substrings to a very narrow range, 17 to 21 bp, would have a major effect on the time to solve the ILP. This run used the same dataset as the r-runs. While there was an apparent drop in the time to reach a 25% gap and average 4 hour gap versus the comparable r-run (r2), it was not dramatic. Such a drop should be expected as the tight length restriction reduces the number of variables in the ILP and therefore limits the possible branches in the solution tree to be explored. We conclude that restricting the length beyond the standard restriction (15-40bp) is not necessary to improve the computability of the string barcoding problem.

The final set of runs done on the Genbank (non-HIV) data were the s-runs. This was to see how the growth of the problem size would affect the running times and gaps. This is illustrated in figures 2.1 and 2.2. In general, increasing the total input size and the number of strings increases the time to reach 25% gap and the gap at 4 hours. We do notice that as the size increases, the increase in time appears to slow. The reader should note that the sizes of problems solved are practical.

The last set of results based on the data from Genbank are the hiv-runs. Recall that these runs were to test the effect of increased similarity on execution time. The results we obtained suggest that increased similarity may lower computation time slightly. The reader can see this in table 1.2 and examines r7 vs hiv0. This drop may be due to fewer variables or because increased similarity results in fewer substrings that may be used to differentiate each pair of strings and hence there are fewer paths to explore in the solution tree. Note that the drop in time to reach 25% gap is larger than the drop in gap at 4 hours.

The last dataset of interest is that of the CBNSP effort at LLNL. We only wish to briefly mention that our method was successful in finding signatures for the LLNL dataset. The only problem is that their assay requires that a signature substring actually be *three* substrings that are all within 300 bp of each other. We believe our implementation could be adapted to solve this special case, but time did not permit us to do so. The significance of this result is that we were able to run on an input file containing almost 550 viruses. A dataset this large is typically not possible as the ILP becomes too large for CPLEX to run with only 2 gigabytes of memory available for data (CPLEX constraint). Since the goal at CBNSP is to obtain a signature for a small subset of the 550 viruses, the resulting ILP was much smaller.

6. Conclusion

It should be clear there is an urgent need for rapid virus detection. The string barcoding problem is an ideal expression of this problem and the solution presented here is quite practical.

The CBNSP is an effort already in place to begin to satisfy this need. Given the usefulness of the problem, it is unfortunate that the most basic constraints ensure that the problem is NP-complete. Despite this fact, however, there is still hope of solving this problem in practice. The implementation presented here finds the provable optimum in many cases of data and parameter combinations. In even more cases, it can get provably close to the optimum (10% gap or less) in just 4 hours or reach a 25% gap rather quickly (in minutes). Additionally, since signature development would likely be done once, and then those signatures used many times, one can afford to spend more time on the initial signature computation in order to minimize the cost of producing field technology. This means that in practice, the solution obtained may be bounded even tighter and provably closer to the optimum since a longer time than 4 hours may be used.

Furthermore, the size of datasets that can be processed are reasonable. Depending on the redundancy and other parameters that affect the size of the ILP, we can process anywhere from 150-300 strings in a dataset. Even larger dataset sizes are possible (1000 or even larger) if the subset of the viruses for which signatures are desired is small (as in CBNSP). Additionally, as we see advances in memory and processor technology, the size of datasets that can be processed will increase even more, which will increase the effectiveness of this method.

Lastly, the reader should recall that the implementation here provides more flexibility and robustness than any prior approach. The CBNSP approach is for a special case of the string barcoding problem and hence is not as well suited for the general case as our method. While the approach presented by Borneman et al [1] does address a more general version of the string barcoding problem than the CBNSP, it still is not as complete and flexible as our method. In short, the method presented here solves the broadest version of the string barcoding problem of any other known method at the time of this writing.

7. Future Directions

The first item is to determine if the unconstrained problem is NP-complete or not. The proof obtained for the max-length variant breaks down when you try to apply it to the unconstrained case. Another approach may be necessary. The other theoretical question left open is to explain the large reduction in unique nodes in the suffix tree. Recall that each node has a binary vector associated with it. We say a node is unique in some set of nodes if no other node exists with the same binary vector. We observed a much greater reduction than a simple analysis could account for.

On the implementation side of things, it would be helpful to expand on the implementation's robustness as far as the size of problem it can handle. For example, we would like to use 1000 viruses and find signatures for the entire set simultaneously. This would not be feasible in the current implementation. The suffix tree, other auxiliary data structures, and the ILP itself require too much memory in order to execute on a dataset of such a size. A divide and conquer approach where the whole problem is split into sub problems and solved and then combined may be appropriate. Along the same lines, it would be helpful to adapt our approach to work for the special case with the CBNSP.

8. ACKNOLWEDGEMENTS

We would like to thank the National Science Foundation. The research here was partially supported by grant DBI-9723346 from the National Science Foundation.

9. TABLES AND GRAPHS OF RESULTS

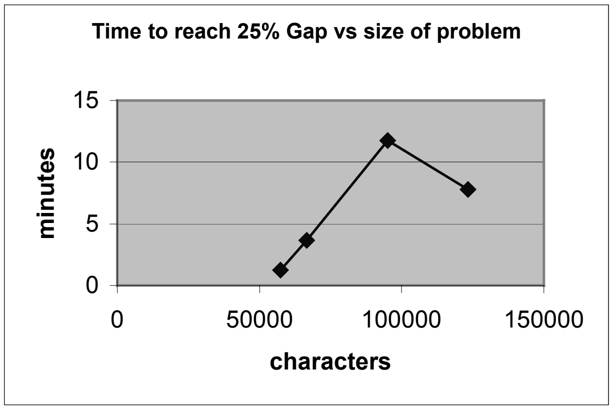

Figure 2.1 - effect of problem size on time to reach 25% gap

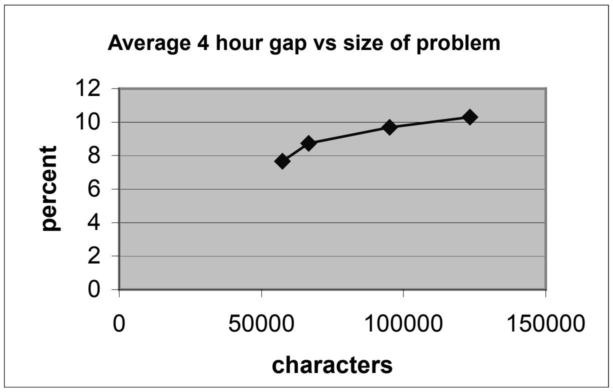

Figure 2.2 - effect of problem size on average 4 hour gap

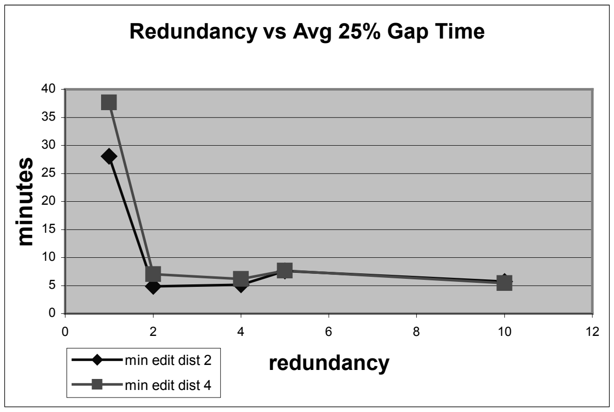

Figure 3.1 - effect of redundancy on avg 25% gap

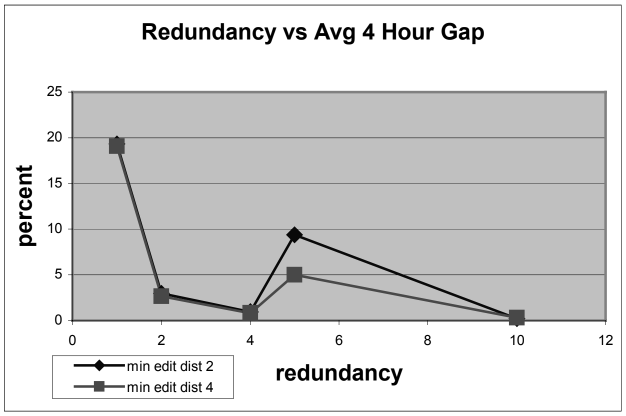

Figure 3.2 - effect of redundancy on avg gap at 4 hours

Table 1.1 - Input data properties for each run

run	avg num of str	avg string size (chars)	avg total input size (chars)	avg edit distance	avg num vars	% vars left
s0	51.12	1123.17	57416.30	0.603	2842.48	6.17
s1	70.64	942.19	66556.20	0.597	5154.80	8.33
s2	105.96	897.63	95112.60	0.596	7273.24	9.11
s3	129.92	948.56	123237.00	0.595	11107.28	9.64
hiv0	91.44	967.50	88468.70	0.589	3712.04	5.01
hiv1	89.28	684.91	61149.00	0.584	2176.28	3.34
hiv4	90.80	723.47	65691.00	0.583	2353.00	3.45
hiv5	90.40	1085.01	98084.90	0.589	4635.84	5.82
hiv6	90.92	849.47	77233.70	0.585	4884.92	6.57
len0	105.40	1086.28	114494.00	0.600	5047.44	6.57
len1	83.60	1044.12	87288.50	0.606	3358.60	5.87
r0	83.60	1044.12	87288.50	0.606	1679.80	2.21
r1	83.60	1044.12	87288.50	0.606	3103.67	4.10
r2	83.60	1044.12	87288.50	0.606	5316.67	7.05
r3	83.60	1044.12	87288.50	0.606	6180.40	8.21
r4	83.60	1044.12	87288.50	0.606	9140.73	12.23
r5	83.60	1044.12	87288.50	0.606	1679.80	2.21
r6	83.60	1044.12	87288.50	0.606	3103.67	4.10
r7	83.60	1044.12	87288.50	0.606	5316.67	7.05
r8	83.60	1044.12	87288.50	0.606	6734.47	8.84
r9	83.60	1044.12	87288.50	0.606	9140.73	12.23

Table 1.2 - program parameters and output measures

run	min len	max len	min edit dist	red	avg sol size	% feasible	% opt 4 hr	% 25% gap 4 hr	avg opt time (num sample)	avg 25% gap time in min (num sample)	avg % gap 4 hr (num samples)
s0	15	40	4	5	99.92	100.00	12.00	100.00	50.92(3)	1.30(25)	7.67(25)
s1	15	40	4	5	117.20	100.00	0.00	100.00	n/a	3.84(25)	8.73(25)
s2	15	40	4	5	115.70	92.00	0.00	100.00	n/a	12.28(23)	9.68(23)
s3	15	40	4	5	200.91	92.00	0.00	100.00	n/a	8.14(23)	10.29(23)
hiv0	15	40	4	4	89.44	100.00	52.00	100.00	58.42(15)	2.64(25)	0.79(25)
hiv1	15	40	4	2	45.12	100.00	40.00	100.00	63.84(10)	2.17(25)	2.47(25)
hiv4	15	40	2	2	43.88	100.00	32.00	100.00	77.99(8)	2.77(25)	2.44(25)
hiv5	15	40	2	5	132.76	100.00	0.00	100.00	n/a	3.79(25)	7.49(25)
hiv6	15	40	4	5	126.61	92.00	0.00	100.00	n/a	4.19(23)	7.06(23)
len0	17	21	4	5	160.29	96.00	0.00	100.00	n/a	8.90(24)	9.90(24)
len1	17	21	2	4	106.67	100.00	73.33	100.00	42.70(11)	3.33(15)	0.51(15)
r0	15	40	2	1	43.20	100.00	0.00	93.33	n/a	28.03(14)	19.34(15)
r1	15	40	2	2	50.13	100.00	33.33	100.00	20.54(5)	4.84(15)	2.94(15)
r2	15	40	2	4	95.67	100.00	53.33	100.00	28.15(8)	5.16(15)	0.95(15)
r3	15	40	2	5	140.93	93.33	0.00	100.00	n/a	7.58(14)	9.37(14)
r4	15	40	2	10	255.64	93.33	71.43	100.00	31.26(10)	5.71(14)	0.20(14)
r5	15	40	4	1	45.73	100.00	0.00	93.33	n/a	37.63(14)	19.09(15)
r6	15	40	4	2	53.60	100.00	40.00	100.00	56.58(6)	7.02(15)	2.62(15)
r7	15	40	4	4	102.64	93.33	50.00	100.00	53.06(7)	6.16(14)	0.81(14)
r8	15	40	4	5	127.43	93.33	14.29	100.00	155.77(2)	7.63(14)	5.00(14)
r9	15	40	4	10	269.42	80.00	58.33	100.00	10.21(7)	5.41(12)	0.30(12)

REFERENCES

[1] James Borneman, Marek Chrobak, Gianluca Della Vedova, Andres Figueroa, Tao Jiang, *Probe Selection Algorithms with Applications in the Analysis of Microbial Communities*, Bioinformatics, Vol No. 1, pages 1-9.

[2] http://www.ncbi.nlm.nih.gov/BLAST/

[3] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York 1999.

[4] Michael R. Garey, David S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W H Freeman & Co, 1979

[5] Pierluigi Crescenzi, Viggo Kann. *A compendium of NP optimization problems,* available online at

[6] http://www.nada.kth.se/~viggo/wwwcompendium/node199.html

[7] Thomas Kämpke, Marksu Kieninger, Michael Mecklenburg. *Efficient primer design algorithm.* Bioinformatics Vol. 17, 2001, pp. 214-225.

[8] Ralf Herwig, Armin O. Schmitt, Matthias Steinfath, John O'Brien, Henrik Seidel, Sebastian Meier-Ewert, Hans Lehrach, Uwe Radelof. *Information Theoretical probe selection for hybridisation experiments.* Bioinformatics, Vol. 16, 2000, pp. 890-898

[9] Pearson, W. R., Robins, G., Wrege, D. E., and Zhang, T. *On the Primer Selection Problem for Polymerase Chain Reaction Experiments*, Discrete and Applied Mathematics, Vol. 71, 1996, pp. 231-246.

[10] Koichiro Doi, Hiroshi Imai. *A Greedy Algorithm for Minimizng the number of Primers in Multiple PCR Experiments,* Japanese Society for Bioinformatics, http://www.jsbi.org/journal/GIW99/GIW99F08.html

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, New York 1999, pgs 974-978.

[12] http://www.niaid.nih.gov/dait/cross-species/page5.htm

[13] http://www.ilog.com/products/cplex/