Algorithmic Topics in Bioinformatics: The Partial Digest Problem (Paper for CSE 202: Prof. Russell Impagliazzo)

Dustin Boswell (dboswell [at] cs ucsd edu)

January 6, 2004

1 Introduction

One of the interesting tasks in Computational Biology is *Restriction Site Mapping*. A DNA strand can be thought of as a string on the letters {A,T,G,C}. When a particular *restriction enzyme* is added to a DNA solution, the DNA is cut at particular *restriction sites*. For example, the enzyme *EcoRI* cuts at every location of the pattern GAATTC. The goal of Restriction Site Mapping is to determine the locations of every site for a given enzyme.

Unfortunately, the DNA string cannot be explicitly observed. Instead, there are various biochemical techniques that allow us to obtain indirect information about where the restriction sites could be. For example, gel electrophoresis is a technique by which the DNA fragments are moved through an electric field. Since the smaller a fragment is, the faster it will move, its length can be inferred from how far it traveled in the field. And the fragment length tells us the distance between some pair of restriction sites. Given the fragment lengths, it is then an algorithmic task to compute the locations of all restriction sites. While this typically doesn't need to be done in real-time, we would like an algorithm that scales well with the DNA length, and with the number of fragments.

For the *Partial Digest Approach*, a batch of DNA is exposed to an enzyme in limited quantity, so not every site is actually cut. The resultant batch contains fragments of all possible lengths between sites. Let $X = \{x_1..x_n\}$ be the set of restriction site locations on a DNA strand. Then $\Delta X = \{x_i - x_j : 1 \leq j < i \leq n\}$ is the (multi)set of all $\binom{n}{2}$ distances between sites produced by this approach.

Partial Digest Problem (PDP) *Given the (multi)set ΔX, compute a set X which could have produced ΔX.*

For example, given $\Delta X = \{1, 5, 6, 7, 11, 12\}$, $X = \{0, 1, 7, 12\}$ is a correct answer. So is $Y = \{0, 5, 11, 12\}$. (Without loss of generality, we require that the first element be 0.) In this example, there are only 2 possible answers, and they happen to be mirror images of each other. In general, however, there can be many possible X for a given ΔX. If X and Y are such that $\Delta X = \Delta Y$, X and Y are *homeometric sets*. Let $H(n)$ denote the maximum number of mutually homeometric sets (of size n) for a given ΔX. [skie90] show that

i) for infinitely many n, $H(n) > \frac{1}{2}n^{0.810}$, for some ΔX

ii) for all n, $H(n) < \frac{1}{2}n^{1.233}$

iii) $H(n) = 2^k$ for some k, or else $H(n) = 0$.

As it turns out, typical problem instances have only 1 solution.

2 Solving PDP via polynomial factoring

[lemk88] showed how to convert an instance of PDP to one of factoring a polynomial constructed from ΔX. They use the method of given by [rose82] and show that the whole process can be done in pseudo-polynomial time. (It is polynomial in $max(\Delta X)$. We would prefer that the algorithm be polynomial only in n, and independent of the value of the elements in the set.)

The method constructs a polynomial whose exponents are the elements of ΔX. One drawback, is that this method only works on inputs with integer values. Interestingly, [lemk88] showed that an arbitrary instance of PDP can be converted to an equivalent integer instance of PDP in polynomial time. Unfortunately, the resulting instance has values that are exponentially large.

3 A Backtracking Algorithm

Skiena et al. created a backtracking algorithm to solve this problem that does not depend on the values of ΔX, but only on n. We present it here.

The partial digest problem can be viewed as the task of selecting $n-1$ of the $\binom{n}{2}$ inputs to constitute the "base lengths" (distances between consecutive x_i). Let us consider the more general sub-problem: given a set L of "unaccounted lengths" and a partial set of locations X', complete the rest of X' such that all elements in L are "accounted for". (An element $l_i \in L$ is accounted for if there is an assigned pair (x_i, x_j) such that $|x_i - x_j| = l_i$.) Our original problem is just the special case where $L = \Delta X - \{max(\Delta X)\}$, and $X' = \{0, max(\Delta X)\}$ (where we start the algorithm off by assigning the maximum distance $width = max(\Delta X)$ to the locations $(0, width)$.

We can break down this problem into a series of decisions "Where do we *place* the largest element l_{max} of L?". When we "place" a piece l_i, we are assigning two locations (x_i, x_j) such that $|x_i - x_j| = l_i$. (x_j is chosen from X', x_i is not yet in X'.) As the algorithm recurses on subproblems, every element in L will have been assigned a unique pair (x_i, x_j). The set X' contains all the locations $\{x_i\}$ as they are assigned. Notice that when a new location x_i is added to X', the distances from each location $x_j \in X'$

to x_i must account for some element in L. That is, if X' currently has m elements, adding a $m+1$ element must account for m elements in L (and those elements are then removed from L). If adding some element x_i to X' would create a (multi)set of lengths that is *not* a subset of the (multi)set L, then x_i is a an invalid choice. To reiterate, we require that adding an element x_i must be "consistent" in that it only creates new "not yet accounted for" lengths that are in L.

We would like that the number of choices for our decision of where to place l_{max} be small. In fact there are only two choices:

Lemma 1 *The only possible assignments for* l_{max} *are* $(x_j = \mathbf{0}\ ,\ x_i = l_{max})$ *or* $(x_i = width - l_{max}\ ,\ x_j = \mathbf{width})$.

Proof. If $(0 < x_a\ ,\ x_b < width)$ were an assignment such that $x_b - x_a = l_{max}$, this would imply that either $(0\ , x_b)$ or $(x_a\ , width)$ was a new length $l' > l_{max}$, which contradicts l_{max} being maximal. Thus, for each subproblem we have to make the choice of whether to add l_{max} or $width - l_{max}$ to X'.

Using this, we give the following algorithm (pseudo-code adapted from [pevz00]):

```
set X = ∅;

set PartialDigest( list L )
  width = DeleteMax(L);
  X = {0, width};
  if Place(L) return X;
  else return ``invalid input'';

bool Place(list L)
  if L = ∅ return TRUE;

  l_max=DeleteMax(L);

  if Δ(l_max, X) ⊆ L                    //if placing on the left is consistent
    X = {l_max} ∪ X;                    //then try this choice
    if Place(L - Δ(l_max, X))           //if subprob is feasible, so is this
      return TRUE;
    else X = X - {l_max};               //otherwise backtrack

  if Δ(width - l_max, X) ⊆ L            //if placing on the right is consistent
    X = {width - l_max} ∪ X;            //then try that choice
    if Place(L - Δ(width - l_max, X))   //check if feasible
      return TRUE;
    else X = X - {width - l_max};       //otherwise backtrack

  return FALSE;                         //if neither choice worked, we're infeasible
```

In this code, Place() returns a boolean indicating whether the subproblem was feasible, and if so adds the necessary elements to X. $\Delta(l_{max}, X)$ returns the (multi)set $\{|l_{max} - x_i| : x_i \in X\}$.

Lemma 2 *(Correctness) If L is valid, PartialDigest() returns a X such that $\Delta X = L$.*

$\Delta X \subseteq L$: Suppose $\Delta X \supset L$ so that $d \in \Delta X$ and $d \notin L$, where $d = |x_i - x_j|$. That would mean at some point in the procedure, x_i was added to X. But the algorithm specifically checks that when an element x_i is added, that all new distances (specifically $|x_i - x_j|$) are in L, which contradicts d not being in L.

$L \subseteq \Delta X$: Suppose $L \supset \Delta X$ so that $l \in L$ and $l \notin \Delta X$. The algorithm only removes elements from L when they are accounted for. And Place() only terminates if $L = \emptyset$, hence l would have been accounted for, or else Place() would not have terminated.

Since Place() only recurses on smaller sub-problems, we know Place() eventually halts. And since Place() (potentially) tries both choices for where to place l_{max}, we know it (potentially) considers all possible subproblems.

Lemma 3 *(Recursion depth) Place() recurses at most n levels.*

At the first call $|X| = 2$. Since each recursion adds one element to X, when making the i^{th} call, $|X| = i + 1$. By definition the final X contains n elements. Thus when making call $n - 1$ we already have the correct size of X. Moreover, the depth is bounded by n at all times. Consider the size of L, which is initially $\binom{n}{2}$-1. At the i^{th} call (starting at $i = 1$), X is of size $i + 1$. And adding the $i + 2^{th}$ element (recall from before) in turn removes $i + 1$ elements from L (those that become accounted for). The following relation describes this bound on the depth based on the cumutalive number of elements removed from L:

$$\begin{aligned} \binom{n}{2} - 1 \; - \sum_{i=1}^{depth-1} (i+1) = \binom{n}{2} - \sum_{i=0}^{depth} (i) \geq 0 \\ \frac{n \cdot (n-1)}{2} - \frac{(depth-1) \cdot depth}{2} \geq 0 \\ depth \leq n \end{aligned}$$

Lemma 4 *(Worst-case runtime) The algorithm is $O(2^n n lg(n))$.*

The recursion is a tree of depth $\leq n$, and of degree ≤ 2, so there are at most $O(2^n)$ calls to Place(). Each call requires computing the new distances implied by l_{max} (performing $\Delta(l_{max}, X)$) which takes $O(n)$, and then for each new distance, performing a binary search on the sorted list L to see if that element is there (L is of size $\leq \binom{n}{2}$, and $lg\binom{n}{2} = O(lg(n))$). From this we get $O(2^n n lg(n))$. The fact that the algorithm ever achieves this bound was shown by [zhan94], who constructed exotic and complicated inputs for which this algorithm takes exponential time.

[skie90], however also claims the following:

Lemma 5 *(Expected runtime) The algorithm runs in $O(n^2 lg(n))$ on "average".*

If the n points of X were chosen as random points on the real interval $[0, width]$ (in so-called "general position"), the elements in ΔX will be unique. Furthermore, on a given decision about where to place l_{max}, the probability that the "wrong" choice for l_{max} is consistent with L will be 0 (since random real intervals will typically not line up). This means that the recursion only proceeds when it is correct, and hence the recursion tree is only a line of length n, which gives $O(n \cdot nlg(n))$.

I personally think this analysis isn't very useful. [skie90] in fact shows that when X is constructed randomly as a set of integers that are "near each other", the number of calls to Place() greatly exceeds n.

Also, I claim that their implementation of performing binary searches to check for consistency is not optimal. They assume L is sorted, and that checking if $O(n)$ elements are in L takes $O(n \cdot lg(n))$. I claim that it can be done in $O(n)$. Notice that if we maintained X in sorted order, then the (multi)set $\Delta(l_{max}, X)$ can be generated in sorted order as well. (To do this we maintain two pointers x_l and x_r initially set to $x_l = 1$, $x_r = n$, and work inwards, always outputting the larger of $(l_{max} - X[x_l])$ and $(X[x_r] - l_{max})$, until x_l and x_r meet.) Then, checking if $\Delta(l_{max}, X) \subseteq L$ can be done in linear time using a "merge check" method. And maintaining X in sorted order can be done in linear time since we only add one element on a given call to Place(). Thus, any appearance of "$nlg(n)$" in their runtime analysis can be replaced with just "n".

4 Poly-time solvable instances

[dani00] shows that PDP can be viewed as a quadratic program. In this context they show that certain classes of inputs are poly-time solvable. For example, the cases given by [zhan94] which take exponential time for the backtracking algorithm, are poly-solvable with their method. Also the class of inputs for which the following hold:

- the solution X is unique
- the multiset ΔX has no repeats
- the backtracking algorithm on this input would never backtrack more than a constant number of times in a row.

is shown to be poly-solvable by their method as well.

5 Noisy versions of the Problem are NP-Hard

In the biology lab, the partial digest approach introduces three sources of noise to the problem:

i) measurement error - in practice, the elements in ΔX can be observed with 0.1% error at best.

ii) additions - as restriction enzymes are not perfect, and stray DNA can contaminate the specimen, extraneous elements in ΔX may be present.

iii) deletions - it is difficult to expose the DNA for the precise amount of time that all distance pairs arise in the *partial* digest ΔX, so elements of ΔX may be missing.

To handle the first and third problem, [skie94] extend their backtracking algorithm to handle distance "intervals" which they show can handle measurement errors up to $O(1/n^2)$. While their algorithm works well in practice, [ciel02] has shown that variations of the PDP problem reflecting the second and third scenarios are NP-hard. For example, consider:

Min Partial Digest Superset (MPDS) *Given a (multi)set L ($|L| = m$), find the minimum n^* such that $|X| = n^*$ and $L \subseteq \Delta X$.*

This reflects the third scenario where L was missing elements to begin with, and we are looking for the smallest answer X that accounts for all of the input L. In the case of no omissions, $|L| = m = \binom{n}{2}$ and $n^* = n$. Otherwise, $m = \binom{n^*}{2} - \#ommisions$. Notice that a trivial solution that satisfies $L \subseteq \Delta X_{triv}$ is

$$X_{triv} = \{x_0 = 0\ ,\ x_i = x_{i-1} + l_i\},$$

which just constructs the consecutive distances between elements in X_{triv} to be equal to the elements in L. Thus, a trivial bound is $n^* \leq m + 1$.

Theorem 1 *MPDS is NP-hard.*

We follow the proof of [ciel02] which reduces *Equal Sum Subsets* (which is NP-Complete; see [woeg92]) to MPDS.

Equal Sum Subsets (ESS) *Given a set*[1] *of m numbers L, are there two*

[1] multi-sets are disallowed since if an element appears twice, the problem is trivially solved.

disjoint, non-empty sets A, B such that $sum(A) = sum(B)$?

ESS is just a variant of *Partition* where elements can be removed.

Lemma 6 *There is a solution (A, B) to ESS instance L if and only if the minimum n^* to MPDS instance L is $n^* \leq m$.*

$(A, B) \Rightarrow n^* \leq m$: W.l.o.g., we can reorder L so that $A = (l_1, \ldots, l_r)$ and $B = (l_{r+1}, \ldots, l_s)$, with $1 \leq r < s \leq m$. We construct an X to solve MPDS as follows. In the same way we created X_{triv} we "lay down" the elements of A starting at 0:

$$X^A = \{x_0^A = 0\ ,\ x_i^A = x_{i-1}^A + l_i\}\ for\ 1 \leq i \leq r$$

then we also lay down the elements of B *on top of* A, also starting at 0:

$$X^B = \{x_0^B = 0\ ,\ x_i^B = x_{i-1}^B + l_{i+r}\}\ for\ 1 \leq i \leq s - r$$

Lastly, we lay down the "leftovers" of $L - (A \cup B)$.

$$X^C = \{x_0^C = 0\ ,\ x_i^C = x_{i-1}^C + l_{i+s}\}\ for\ 1 \leq i \leq m - s$$

We then let $X = X^A \cup X^B \cup X^C$. It is clear that $\Delta X \supseteq L$ since each distance is explicitly constructed in X. And by construction, we also know that $|X| \leq |X^A| + |X^B| + |X^C| - 2 = (r + 1) + (s - r + 1) + (m - s + 1) - 2 = m + 3 - 2 = m + 1$ (the "-2" is so we don't count the 2 duplicate 0's in X. But now we observe that $|X|$ is in fact $\leq m$: recall that since $sum(A) = sum(B)$, X^A and X^B will "line up" at the end. That is, $max(X^A) = max(X^B)$, which means there is at least one other redundant element in constructing X, and hence the actual number of elements in the set X is at least 1 smaller than $m + 1$.

$(A, B) \Leftarrow n^* \leq m$: Let $X = (x_1, \ldots, x_{n^*})$ be an optimal solution for MPDS such that $n^* \leq m$. Since $L \subseteq \Delta X$ we know for each $l_i \in L$ there is an associated pair (x_i, x_j) such that $l_i = |x_i - x_j|$. We now define the graph $G = (V, E)$ with $V = X$ and $E = \{(x_i, x_j) : (x_i, x_j)$ is associated with some $l_i\ \}$. The cardinalities are simply $|V| = n^*$ and $|E| = |L| = m$. Since we are given that $m \geq n^*$ it must be the case that G has a cycle. Call the cycle $C = c_1 \ldots c_s$. where c_i is the value of a corresponding vertex x_j. We can partition C into $I^+ \cup I^-$ such that

$$I^+ = \{i \in [1, s] \mid c_{i+1} > c_i\ \}$$
$$I^- = \{i \in [1, s] \mid c_{i+1} < c_i\ \}$$

Using the convention $c_{s+1} = c_1$, we can describe the closed cycle by:

$$\begin{aligned} 0 &= \sum_i (c_{i+1} - c_i) \\ &= \sum_{i \in I^+} (c_{i+1} - c_i) + \sum_{i \in I^-} (c_{i+1} - c_i) \\ &= \sum_{i \in I^+} |c_{i+1} - c_i| - \sum_{i \in I^-} |c_{i+1} - c_i| \\ &= \sum_{i \in I^+} a_i - \sum_{i \in I^-} b_i \end{aligned}$$

where the sets $A = \{a_i = |c_{i+1} - c_i| : i \in I^+\}$, and $B = \{b_i = |c_{i+1} - c_i| : i \in I^-\}$ are a solution to ESS since they have equal sums.

And we claim without proof that these reductions can be done in polynomial time, which completes the reduction.

6 Conclusion

The complexity of the original Partial Digest Problem is still unknown. While there is a pseudo-polynomial time algorithm, and a backtracking algorithm that runs in expected poly-time on "random" instances, no-one has found a true poly-time algorithm. At the same time, [skie90] considers it highly unlikely that the problem is NP-hard.

At this point, finding a poly-time algorithm is more of theoretical interest than a practical one. There are 3 main reasons why. First, a biological application will always present a noisy version of the input. Second, the extended version of the backtracking algorithm presented here (one that can handle noise) seems to work well on real-life instances of the problem despite lacking a tight worst-case bound. And the third reason is that the partial digest approach is becoming less popular as other cheaper biological techniques arise for sequencing DNA.

7 References (main papers in bold)

[ciel02] M. Cieliebak, S. Eidenbenz, P. Penna: *Noisy Data Make the Partial Digest Problem NP-hard*, Technical Report no. 381, ETH Zurich, Department of Computer Science, 2002.

[daki00] T. Dakic: *On the Turnpike problem*, PhD thesis, Simon Fraser University, 2000.

[lemk88] P. Lemke, M. Werman: *On the complexity of inverting the autocorrelation function of a finite integer sequence, and the problem of locating n points on a line, given the* $\binom{n}{2}$ *unlabelled distances between them*, Preprint 453, Institute for Mathematics and its Application IMA, 1988.

[pevz00] P. Pevzner: *Computational Molecular Biology: An Algorithmic Approach*, MIT Press, 2000.

[rose82] J. Rosenblatt, P. Seymour: *The Structure of Homometric Sets*, SIAM Journal on Algebra and Discrete Methods 3 (pages 343-350), 1982.

[skie90] S. Skiena, W. Smith, P. Lemke: *Reconstructing sets from interpoint distances*, Sixth AM Symposium on Computational Geometry, pages 332-339, 1990.

[skie94] S. Skiena, G. Sundaram: *A Partial Digest Approach to Restriction Site Mapping*, Bulletin of Mathematical Biology, 56:275-294, 1994.

[woeg92] G. Woeginger, Z. Yu: *On the equal-subset-sum problem*, Information Processing Letters, 42:299-302, 1992.

[zhan94] Z. Zhang: *An Exponential Example for a Partial Digest Mapping Algorithm*, Journal of Computational Biology, 1(3):235-239, 1994.