DNA Computation to solve the Hitting String Problem

George T. Heineman, Richard Resnick, Gabor Sarkozy
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609, USA

WPI-CS-TR-98-15

1 Hitting String Problem

1.1 The Problem

Given a set of strings \( A \), \( |A| = m \), of strings of length \( n \) over an alphabet \( \{0, 1, \$\} \), let \( a_{ij} \) be the \( j^{th} \) character of the \( i^{th} \) string in \( A \). Is there a hitting string \( x \in \{0, 1\} \) with \( |x| = n \) such that for each \( a_i \in A \), there is some \( j, 0 \leq j < n \) for which \( a_{ij} \) and \( x_j \) (the \( j^{th} \) symbol of \( x \)) are identical? Without loss of generality, we assume that the strings in \( A \) are unique, since multiple copies of the same string do not affect the selection of \( x \). We also throw out all columns that consist entirely of \$ character; this can be done in \( O(mn) \) time. If a row consists entirely of \$ characters then no solution is possible. These pre-processing steps can be performed in \( O(mn) \) time.

A hitting string is sparse if \( \exists j, 0 \leq j < n \) such that \( \forall i, 0 \leq i < m, a_{ij} \neq x_j \); a dense hitting string does not have this property. A sparse hitting string can be converted to a dense hitting string in \( O(mn) \) using the algorithm described in appendix A.

1.2 The Approach

We construct a directed acyclic graph \( G = (V, E) \) consisting of vertices \( v_{ij} \) representing the digit positions of each \( a_i \in A \); this results in a graph of \( mn \) vertices. First some definitions: \( row(v_{ij}) = i \), \( column(v_{ij}) = j \), and \( c_j \) is the sequence of \( m \) vertices \( \{v_{0j}, v_{1j}, \ldots, v_{m-1,j}\} \). We define a partial function \( \text{next}(v_{ij}) = v_{kj} \) if \( \exists k \) such that \( m > i > l, a_{lj} = a_{ij} \), and \( \forall c, i < c < l, a_{cj} \neq a_{ij} \); \( \text{next}(v_{ij}) \) is undefined otherwise. We also define the partial function \( \text{prev}(v_{ij}) = v_{kj} \) if and only if \( \text{next}(v_{kj}) = v_{ij} \).

We intend to create a set of edges such that a hitting string \( x \) maps to a path containing at least one vertex from every row \( r_i \), \( 0 \leq i < m \). We populate \( G \) with two types of directed edges, \text{vertical} and \text{cross}. A vertical edge \( (u, \text{next}(u)) \) exists for all nodes \( u \in V \) if \( \text{next}(u) \) is defined. In the worst case, this will create \((m - 1)n\) vertical edges in \( O(mn) \) steps using the algorithm in Figure 1. Vertical edges ensure that paths through \( G \) traverse vertices in \( c_j \) with the same digit value.

Cross edges connect vertices between two adjacent columns, creating opportunities for paths to visit each column in \( G \). Let \( \text{Bottom}[\text{digit}]_j \) be the vertex \( v_{kj} \) where \( \text{digit} \in \{0, 1\}, a_{kj} = \text{digit} \), and \( \text{next}(v_{kj}) \) is undefined. This is the vertex \( v_{kj} \) with largest \( k \) such that \( \forall i, k < i < m, a_{ij} \neq \text{digit} \). If such a vertex does not exist, then \( \text{Bottom}[\text{digit}]_j \) is undefined. Let \( \text{Top}[\text{digit}]_j \) be the vertex \( v_{kj} \) where \( \text{digit} \in \{0, 1\}, a_{kj} = \text{digit} \), and \( \text{prev}(v_{kj}) \) is undefined. This is the vertex \( v_{kj} \) with the smallest \( k \) such that \( \forall c, 0 \leq c < k, a_{cj} \neq \text{digit} \); \( \text{Top}[\text{digit}]_j \) is undefined if no such vertex exists.

Briefly, if each \( c_j \) is connected by a cross edge to \( c_{j+1} \), a path can thread itself through \( G \) starting from \( c_0 \) and terminating in \( c_{n-1} \). A search for a hitting string is analogous to a search for a path that contains vertices from every row. We now show how to construct \( G \) to determine at least one possible dense hitting string. There are at most four cross edges from each \( c_j \) \( \langle \text{Bottom}[\text{digit}]_j, \text{Top}[\text{digit}]_{j+1} \rangle \), with \( \text{digit} \) and \( \text{digit}_{j+1} \) in \( \{0, 1\} \). Naturally, no cross edge is possible if either \( \text{Bottom}[\text{digit}]_j \) or \( \text{Top}[\text{digit}]_{j+1} \) is undefined; note that there are no cross edges leading away from \( c_{n-1} \).
exists a path $P$ and where some

No w that we h av e constructed an acyclic graph $G$, w e sho w that a dense hitting string exists if and only if there

$1/./3$ The Computation

to construct/, therefore/, a DNA sequence for eac h

Observ e that vertical edges gather together all the ro ws with the same digit v alue for a particular column/; w e

The DNA computation in the next section sho ws ho wt oo v ercome this barrier/.

generates all dense hitting strings but is ine/#0Ecien t/, since it m ust generate all p ossible paths to lo cate the solution/. We sho w ho wt oc o n v ert the v ertices

$G$ W e sho w ho wt oc o n v ert the v ertices

edges/. Let

construct/, therefore/, a DNA sequence for eac h

bits

be the binary represen tation of $i$ using the alphabet \{C, G\}. Representing row $r_j$ by the substring $A \cdot bits_r \cdot T$, we define $vertical_{ij}$ to be the concatenation (in any order) of the substrings for row $r_k$ such that $a_{kj} = d$.

head and tail are unique poly-nucleotide sequences generated for each $c_j$ over the alphabet \{A, T\}. Columns $c_1$

now 1/./3:\n

$endprocedure$

Figure 1: Algorithm for calculating values for constructing graph

1.3 The Computation

Now that we have constructed an acyclic graph $G$, we show that a dense hitting string exists if and only if there

exists a path $P (v_0,v_1)(v_1,v_2) \ldots (v_{p-1},v_p)$, with the following properties: (P1) that \forall $i, 0 \leq i < m, \exists j, 0 \leq j < n, some $v_{i,j}$ is a vertex for an edge in $P$ and; (P2) for every vertex $v_{kj}$ in $P$ that belongs to $c_j$, $a_{kj}$ is the same digit.

First, given a dense hitting string $x$, we know that each column $c_j$ contains at least one correct digit position, $v_{kj}, 0 \leq k < m$. In each column $c_j$ either $Top[x_j] = Bottom[x_j]$ (in which case $P_j$ contains the single vertex $Top[x_j]$) or there is a path $P_j$ from $Top[x_j]$ to $Bottom[x_j]$ composed of vertical edges. $v_{kj}$ must be a vertex on $P_j$ by definition of the next partial function. We construct a path on $G$ of the form $P_0 \cdot c e_0 \cdot P_1 \cdot c e_1 \cdot \ldots \cdot c e_{n-1} \cdot P_{n-1}$, where $ce_j$ is a cross edge from $c_j$ to $c_{j+1}$. This path satisfies properties P1 and P2.

In the reverse direction, we are given a directed acyclic graph $G = (V, E)$ composed of a matrix of $n$ columns and $m$ rows, for a total of $mn$ vertices. Edges in $E$ are either down within the same column or start in $c_j$ and end in $c_{j+1}$. From this graph, we generate paths for $dig_1, dig_2 \in \{0, 1\}$ from $Top[dig_1]_{0}$ to $Bottom[dig_2]_{n-1}$; these paths

that contain at least one vertex from each of the $m$ rows map to dense hitting strings. The algorithm in Figure 2

generates all dense hitting strings but is inefficient, since it must generate all possible paths to locate the solution.

The DNA computation in the next section shows how to overcome this barrier.

1.4 DNA Solution

We show how to convert the vertices $V$ and edges $E$ into DNA sequences that will combine to form DNA strings that map to paths through $G$. From these possible paths we show how to extract solutions to the hitting string problem.

The goal is to generate DNA sequences for each column that will combine to form strands containing $n$ sequences. Observe that vertical edges gather together all the rows with the same digit value for a particular column; we construct, therefore, a DNA sequence for each $c_j$ and $digit \in \{0, 1\}$ that contains all the information from the vertical edges. Let $bits_j$ be the binary representation of $i$ using the alphabet \{C, G\}. Representing row $r_j$ by the substring $A \cdot bits_r \cdot T$, we define $vertical_{ij}$ to be the concatenation (in any order) of the substrings for row $r_k$ such that $a_{kj} = d$.

head and tail are unique poly-nucleotide sequences generated for each $c_j$ over the alphabet \{A, T\}. Columns $c_1$

through $c_{n-2}$ each have associated head and tail polyomers while $c_0$ only has head polymers and $c_{n-1}$ only has tail polyomers. There will be at most $2(n-1)$ sequences needed to be generated, thus the length of these poly-nucleotide
function fullPath : boolean
begin
  for $i := 0$ to $m - 1$
    if $\text{count}(\text{row}(i)) = 0$ then return false;
  end for
  return true;
end function

function search (in $u$, inout $P$) : boolean
begin
  $\text{prev} := u$;
  do
    $P$.addVertex($u$);
    $\text{count}[$row($u$)] += 1;
    if (fullPath()) then return true;
  $u := \text{next}(u)$;
  while ($u$ is defined);
  $u := \text{prev}$;
  if (exists cross edge $(u, v)$ for 0) then
    if (search $(v, P) = \text{true}$) then return true;
  end if
  if (exists cross edge $(u, v)$ for 1) then
    if (search $(v, P) = \text{true}$) then return true;
  end if
  $\text{count}[$row($u$)] -= 1;
  $P$.removeVertex($u$);
  return false;
end procedure

function constructHittingString (in $\text{startDigit}$) : boolean
begin
  $\text{top} := \text{Top}[$startDigit$]$;
  if (top is undefined) then return ;
  $P := \text{empty path}$
  return (search (top, $P$));
end procedure

procedure generateDenseHittingStrings
begin
  if (constructHittingString (0) = true) then
    outputPath($P$);
  else if (constructHittingString (1) = true) then
    outputPath($P$);
  end if
end procedure

Figure 2: Creating hitting string from path
<table>
<thead>
<tr>
<th>Element</th>
<th>Column</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>column $c_{jd}$</td>
<td>$0 &lt; j &lt; n - 1$</td>
<td>$\text{tail}<em>j$ : $\text{T} \cdot \text{vertical}</em>{jd} \cdot \text{T} \cdot d \cdot \text{bits}_j \cdot \text{A} \cdot \text{head}_j$</td>
</tr>
<tr>
<td></td>
<td>$j = 0$</td>
<td>$A \cdot \text{vertical}_{jd} \cdot \text{T} \cdot d \cdot \text{bits}_j \cdot \text{A} \cdot \text{head}_j$</td>
</tr>
<tr>
<td></td>
<td>$j = n - 1$</td>
<td>$\text{tail}<em>j$ : $\text{vertical}</em>{jd}$ : $d \cdot \text{bits}_j \cdot \text{A}$</td>
</tr>
<tr>
<td>cross edge $(c_j, c_{j+1})$</td>
<td></td>
<td>$\text{head}<em>j$ : $\text{C} \cdot \text{tail}</em>{j+1}$</td>
</tr>
</tbody>
</table>

Figure 3: DNA sequences for elements of graph $G$

![Diagram of DNA sequences for elements of graph $G$.](image)

Figure 4: Graph for sample problem

![Graph for sample problem.](image)

sequences is bounded by $b = \lceil \log_2(n - 1) \rceil$.

Cross edges are represented by polymere chains of length $2b$ composed of the complement of the head for $v_{ij}$ and the tail for $v_{k,j+1}$. Since head and tail are composed only of A and T nucleotides, cross edges are also composed only of A and T nucleotides.

This mapping to nucleotide strings will construct DNA strands representing all paths $P$ in $G$. Since each head$_{ij}$ and tail$_{ij}$ is a unique $b$-length nucleotide for each column, we can determine the extensions of path DNA. The 5'-strand of a two-strand DNA represents the path $(c_0, c_1) (c_1, c_2) \ldots (c_{p-1}, c_p)$. The path DNA at the 3' end can be extended by a cross edge $(c_j, c_{j+1})$ because the leftmost part of the cross edge is the complement of the rightmost part of a column sequence; also note that there can be no cross edges from $c_{n-1}$.

The DNA strand is constructed from left to right, and since the graph is acyclic, the DNA strings will reach a maximum finite length in finite time. Once the DNA sequences have been formed, they need to be separated to locate hitting string solutions. Using the magnetic technique, we separate out from the mixture those DNA strands that have embedded sequences for each of the $m$ layers. This is accomplished in $m$ steps and the resultant mixture is separated into single-strand DNA.

The first post-processing phase eliminates all DNA strands that are not hitting strings. We search for sequences of the form $\text{A} \cdot \text{bits}_j \cdot \text{T}$ for $0 \leq i < m$. These sequences are the primary markers embedded within each column encoding. A hitting string must contain all the embedded levels. The second post-processing phase reads the results from the remaining strings to determine the actual hitting string $x$. At this point, each strand is a solution and we sequence the entire strand. Within this sequence, we search for secondary markers that embed the column position with each vertex. These sequences are of the form $\text{T} \cdot x_j \cdot \text{bits}_j \cdot \text{A}$, using C to encode $x_j = 0$ and G for $x_j = 1$. We perform $n$ searches for $\text{T} \cdot \text{C} \cdot \text{bits}_j \cdot \text{A}$; for each $j$ that this string is found, $x_j = 0$; otherwise $x_j = 1$. □

1.5 Sample Problem

Consider the simple example of $A = \{ \text{0$\text{\$01}$, 11$\text{\$0}$, 88$\text{\$11}$, 100$\text{\$0}$, 88$\text{\$00}$ \}$, with $m = 5$, and $n = 4$. This creates a graph of twenty vertices and seventeen edges, as shown in Figure 4. Vertical edges are created for each column between
The length of the hitting string will be equivalent to \( zn + \sum_{j=0}^{n-1} 2^j \), and different hitting strings will have different
lengths, and appear at different bands in the resulting gel. Our example of padded strings appears in Figure 7. This approach is clearly limited since the size of the DNA is exponential. If one could discover another additive feature, this counting approach would be feasible.

2 3 Satisfiability

This section shows how to create an instance of the hitting string problem from an instance of 3SAT [Fagin, 1974]. A 3SAT instance is in conjunctive normal form whereby each clause is composed of three literals, either a Boolean variable $x$ or a negated Boolean variable, $\overline{x}$. Let $\phi$ be a formula with $m$ clauses such that $\phi = (a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land \cdots \land (a_k \lor b_k \lor c_k)$. Let $n$ be the total number of unique Boolean variables in $\phi$. Construct $m$ strings such that $a_{kj} = 1$ if the $j^{th}$ Boolean variable is negated in clause $k$, $a_{kj} = 0$ if the $j^{th}$ Boolean variable appears as normal in clause $k$, and $a_{kj} = \$ otherwise. This process can be performed in polynomial time. Hitting string solutions for this set of strings determine the variable assignment of the different Boolean variables. □

Algorithms The following is an algorithm for converting a sparse hitting string into a dense hitting string in $O(mn)$.