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 $next(v_{ij}) = v_{lj}$ if $\exists l$ such that $m > l > i, a_{lj} = a_{ij}$, and $\forall c, i < c < l, a_{cj} \neq a_{ij}$; $next(v_{ij})$ is undefined otherwise. We also define the partial function $prev\ (v_{lj}) = v_{kj}$ if and only if $next\ (v_{kj}) = v_{lj}$.

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, *vertical* and *cross*. A vertical edge (u, $next(u)$) exists for all nodes $u \in V$ if $next(u)$ 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 $Bottom[digit]_j$ be the vertex v_{kj} where $digit \in \{0, 1\}, a_{kj} = digit$, and $next(v_{kj})$ is undefined. This is the vertex v_{kj} with largest k such that $\forall i$, $k < i < m$, $a_{ij} \neq digit$. If such a vertex does not exist, then $Bottom[digit]_j$ is undefined. Let $Top[digit]_j$ be the vertex v_{kj} where $digit \in \{0, 1\}, a_{kj} = digit$, and $prev(v_{kj})$ is undefined. This is the vertex v_{kj} with the smallest k such that $\forall c$, $0 \leq c < k$, $a_{cj} \neq digit$; $Top[digit]_j$ 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 ($Bottom[dig_j]_j$, $Top[dig_{j+1}]_{j+1}$), with dig_j and dig_{j+1} in $\{0, 1\}$. Naturally, no cross edge is possible if either $Bottom[dig_j]_j$ or $Top[dig_{j+1}]_{j+1}$ is undefined; note that there are no cross edges leading away from c_{n-1}.

```
procedure createVerticalEdges
begin
    maxVertical := 0;
    for j := 0 to n − 1
        lastPos[0] := lastPos[1] := −1;
        numVertical := 0;
        for i := m − 1 downto 0
            prevArray[i][j] := nextArray[i][j] := −1;
            digit := a_ij;
            if (digit = 0) or (digit = 1) then
                if (lastPos[digit] ≥ 0) then
                    nextArray[i][j] := lastPos[digit];
                    prevArray[lastPos[digit]][j] := i;
                    numVertical++;
                end if
                lastPos[digit] := i;
            end if
        end for
    maxVertical := max (maxVertical, numVertical);
    end for
end procedure
```

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_{ij} 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, $v_{k_j,j}$, $0 \leq k_j < m$. In each column c_j either $Top[x_j]_j = Bottom[x_j]_j$ (in which case P_j contains the single vertex $Top[x_j]_j$) or there is a path P_j from $Top[x_j]_j$ to $Bottom[x_j]_j$ composed of vertical edges. $v_{k_j,j}$ must be a vertex on P_j by definition of the *next* partial function. We construct a path on G of the form $P_0 \cdot ce_0 \cdot P_1 \cdot ce_1 \cdots ce_{n-1} \cdot P_{n-1}$, 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 $Bottom[dig_2]_{n-1}$; those 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 $bits_i$ be the binary representation of i using the alphabet $\{\mathbf{C}, \mathbf{G}\}$. Representing row r_i by the substring $\mathbf{A} \cdot bits_i \cdot \mathbf{T}$, we define $vertical_{jd}$ to be the concatenation (in any order) of the substrings for row r_k such that $a_{kj} = d$.

$head_j$ and $tail_j$ are unique poly-nucleotide sequences generated for each c_j over the alphabet $\{\mathbf{A}, \mathbf{T}\}$. Columns c_1 through c_{n-2} each have associated *head* and *tail* polymeres while c_0 only has *head* polymeres and c_{n-1} only has *tail* polymeres. 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 (count[i] = 0) then return false;
    end for
    return true;
end function

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

function constructHittingString (in startDigit) : boolean
begin
    top := Top[startDigit]_0;
    if (top is undefined) then return ;
    P := 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

Element	Column	Composition
column c_{jd}	$0 < j < n-1$	$tail_j \cdot$ $\mathbf{T} \cdot \mathbf{A} \cdot vertical_{jd} \cdot \mathbf{T} \cdot$ $d \cdot bits_j \cdot \mathbf{A} \cdot head_j$
	j = 0	$\mathbf{A} \cdot vertical_{jd} \cdot \mathbf{T} \cdot$ $d \cdot bits_j \cdot \mathbf{A} \cdot head_j$
	$j = n-1$	$tail_j \cdot$ $vertical_{jd} \cdot$ $d \cdot bits_j \cdot \mathbf{A}$
cross edge (c_j, c_{j+1})		$\overline{head_j} \cdot$ $\mathbf{C} \cdot \overline{tail_{j+1}}$

Figure 3: DNA sequences for elements of graph G

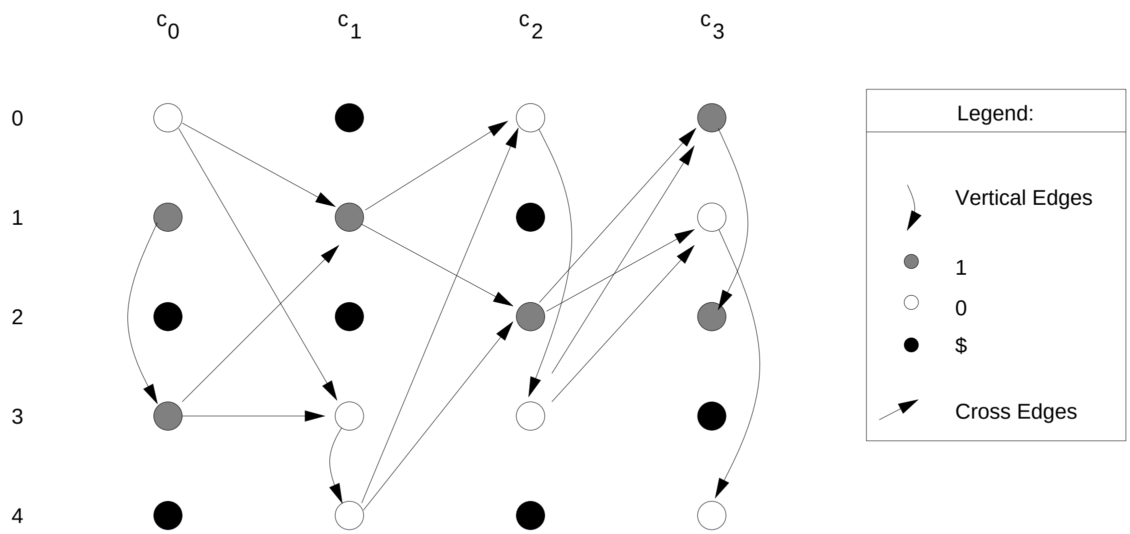

Figure 4: Graph for sample problem

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 $\mathbf{A} \cdot bits_i \cdot \mathbf{T}$ 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 $\mathbf{T} \cdot x_j \cdot bits_j \cdot \mathbf{A}$, using **C** to encode $x_j = 0$ and **G** for $x_j = 1$. We perform n searches for $\mathbf{T} \cdot \mathbf{C} \cdot bits_j \cdot \mathbf{A}$; for each j that this string is found, $x_j = 0$; otherwise $x_j = 1$. □

1.5 Sample Problem

Consider the simple example of $A = \{$ 0\$01, 11\$0, \$\$11, 100\$, \$0\$0 $\}$, 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

Columns	c_{00}	**A**· CCC·**T**· C · CC·**A**· ATA
	c_{01}	**A**· CCG· **T**·**A**· CGG·**T**· G · CC·**A**· ATA
	c_{10}	TTA · **A**· CGG· T·**A**· GCC·**T**· C · CG·**A**· TAT
	c_{11}	TTA · **A**· CCG·**T**· G · CG·**A**· TAT
	c_{20}	TTT · **A**· CCC· **T**·**A**· CGG·**T**· C · GC·**A**· AAA
	c_{21}	TTT · **A**· CGC·**T**· G · GC·**A**· AAA
	c_{30}	AAT · **A**· CCG· **T**·**A**· GCC·**T**· C · GG·**A**
	c_{31}	AAT · **A**· CCC· **T**·**A**· CGC·**T**· G · GG·**A**
Cross Edges	(c_0,c_1)	TAT · **C**· AAT
	(c_1,c_2)	ATA · **C**· AAA
	(c_2,c_3)	TTT · **C**· TTA

Figure 5: DNA sequences for sample problem

Solution for: 0010

```
    C00                         C10                             C21                   C30

(A CCC T C CC A ATA) (TTA A CGG T A GCC T C CG A TAT) (TTT A CGC T G GC A AAA) (AAT A CCG T A GCC T C GG A)
               (TAT C AAT)                       (ATA C AAA)               (TTT C TTA)
```

Solution for: 1011

```
     C01                        C10                                 C21                      C31

(A CCG T A CGG T G CC A ATA) (TTA A CGG T A GCC T C CG A TAT) (TTT A CGC T G GC A AAA) (AAT A CCC T A CGC T G GG A)
                       (TAT C AAT)                       (ATA C AAA)               (TTT C TTA)
```

Solution for: 1001

```
     C01                        C10                                 C20                                C31

(A CCG T A CGG T G CC A ATA) (TTA A CGG T A GCC T C CG A TAT) (TTT A CCC T A CGG T C GC A AAA) (AAT A CCC T A CGC T G GG A)
                       (TAT C AAT)                       (ATA C AAA)                       (TTT C TTA)
```

Figure 6: DNA solution for sample problem

vertices representing the same digit. There are five vertical edges and twelve cross edges. Because there are $m = 5$ strings in A, we need three digit nucleotide sequences to represent the bit position for each row $0 \leq i < m$.

Figure 5 contains the DNA solution for this example graph, and Figure 6 contains the three hitting string solutions. Recall that only $2(n-1)$ unique *head*/*tail* strings are necessary; in this case $\lceil \log(2(n-1)) \rceil = 3$.

1.6 Counting hitting strings

A more difficult problem with current technology is counting the total number of unique hitting strings for a given A. If one can accurately produce exact sequences of very long DNA, then for "reasonable" n, one possible solution would be to use the device of measuring the length of the DNA strands, and count the number of unique bands that appear in the gel. **Richard: add something about this technology here**. For this to work properly, we must ensure that x_1 and x_2 are the same if and only if $|x_1| = |x_2|$. To do this, we first construct the nucleotide sequences for the columns to be of the same length z; currently it fluctuates based upon the number of vertical edges within a column. Given a particular column encoding, such as Figure 5, include "padding" bases to extend the shorter fragments to be of size z, where $z = 13 + 5 * \mathit{MaxVertical}$ and $\mathit{MaxVertical}$ is the largest number of vertical edges in any single column. The padding must be carefully added to ensure that no additional bindings are possible between the DNA sequences. For c_0, prepend a string of length b of alternating bases **C** and **A**. This ensures no false match to some head_j. Then, for those columns with less than *MaxVertical*, insert alternating bases of **C** and **A**. Finally, for c_{n-1}, append a string of length b of alternating bases **C** and **A**. Under this situation, all hitting strings have the same length zn. Now, for those sequences representing c_j, insert alternating sequences of **C** and **A** of length 2^j. 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

Columns	c_{00}	CAC · **C**· ACA· C·**A**· CCC·**T**· C · CC·**A**· ATA
	c_{01}	CAC · **CA**· CCG· T·**A**· CGG·**T**· G · CC·**A**· ATA
	c_{10}	TTA · **A**· CGG· T·**A**· GCC·**T**· C · CG·**A**· TAT
	c_{11}	TTA · **CAC**· ACA· C·**A**· CCG·**T**· G · CG·**A**· TAT
	c_{20}	TTT · **A**· CCC· T·**A**· CGG·**T**· C · GC·**A**· AAA
	c_{21}	TTT · **CACAC**· ACA· C·**A**· CGC·**T**· G · GC·**A**· AAA
	c_{30}	AAT · **A**· CCG· T·**A**· GCC·**T**· C · GG·**A**· CAC
	c_{31}	AAT · **CACACACAA**· CCC· T·**A**· CGC·**T**· G · GG·**A**· CAC
Cross Edges	(c_0,c_1)	TAT · **C**· AAT
	(c_1,c_2)	ATA · **C**· AAA
	(c_2,c_3)	TTT · **C**· TTA

Figure 7: Padded DNA sequences for counting Hitting Strings

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 ϕ be a formula with m clauses such that $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$. Let n be the total number of unique Boolean variables in ϕ. 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 $O(mn)$.