Reconstructing Strings from Substrings

Steven S. Skiena*
Gopalakrishnan Sundaram

Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400
e-mail: {skiena,sundaram}@sbcs.sunysb.edu

May 16, 1993

1 Introduction

Suppose that there was an unknown string S, over a known alphabet Σ, containing the secret to life. We are permitted to ask questions of the form "is s a substring of S?", where s is a specific query string over Σ. We are not told where s occurs in S, nor how many times it occurs, just whether or not s a substring of S. Our goal is to determine the exact contents of S using as few queries as possible.

Although this tale is perhaps over-dramatic, it is not completely inaccurate. The problem arises in sequencing by hybridization (SBH) [2, 6, 8, 19, 24], a new and promising approach to DNA sequencing which offers the potential of reduced cost and higher throughput over traditional gel-based approaches.

The basic sequencing by hybridization procedure attaches a set of single-stranded fragments to a substrate, forming a *sequencing chip*. A solution of radiolabeled single-stranded target DNA fragments are exposed to the chip. These fragments hybridize with complementary fragments on the chip, and the hybridized fragments can be identified using a nuclear detector. The target DNA can now be sequenced based on the constraints of which strings are and are not substrings of the target. Pevzner and Lipshutz [24] give an excellent survey of the current state of the art in sequencing by hybridization, both technologically and algorithmically.

In this paper, we consider a variety of problems with application to sequencing by hybridization. First, we develop a theory of interactive sequencing by hybridization, based on

*This work was partially supported by NSF Research Initiation Award CCR-9109289, New York State Science and Technology Foundation Grant RDG-90171, and an ONR Young Investigator Award.

reconstructing strings from substrings. On receiving a sample to sequence, the system will decide which substring it would like to query, and synthesize or extract the corresponding primer. Based on whether or not this primer hybridizes, it will propose a new query, and the process will repeat until the sample is completely sequenced. This approach is applicable to Crkvenjakov and Drmanac's target down approach to sequencing by hybridization [6, 24], although the problem is also of independent interest.

In this paper:

- We provide tight bounds on the complexity of reconstructing unknown strings from substring queries. Specifically, we show that $(\alpha - 1)n + \Theta(\alpha\sqrt{n})$ queries are sufficent to reconstruct an unknown string, where α is the alphabet size and n the length of the string. This matches the information-theoretic lower bound for binary strings. Further, we show that $\sim \alpha n/4$ queries are necessary, which is within a factor of 4 of our upper bound for larger alphabets. This lower bound holds even for a stronger model which returns the number of occurrences of s as a substring of S, instead of simply whether it occurs.

- We demonstrate that subsequence queries are significantly more powerful than substring queries. Specifically, $O(n \lg \alpha + \alpha \lg n)$ subsequence queries suffice for reconstructing, matching the information theoretic lower bound.

- In certain applications, it may already be known that the unknown string is one of a small set of possibilities, and hence can be determined faster than an arbitrary string. We show that building an optimal decision tree is NP-complete, then give an approximation algorithm which gives trees within a constant multiplicative factor of optimal, with the constant depending upon α.

The conventional approach to sequencing by hybridization uses prefabricated chips instead of interactive queries. In the classical sequencing chip $C(m)$, all 4^m single-stranded oligonucleotides of length m are attached to the surface of a substrate. For example, in $C(8)$ all $4^8 = 65,536$ octamers are used.

Pevzner's algorithm [25] for reconstruction using classical sequencing chips interprets the results of a sequencing experiment as a subgraph of the de Bruijn graph, such that any Eulerian path corresponds to a possible sequence. Thus the reconstruction is not unique unless the subgraph consists entirely of a directed induced path. The strength of this requirement means that large sequencing chips are needed to reconstruct relatively short strands of DNA. For example, the classical chip $C(8)$ suffices to reconstruct 200 nucleotide long sequences in only 94 of 100 cases [23], even in error-free experiments.

However, additional information about the sequence is often available, in particular its length. We show that length can be used to help disambiguate the sequence. For example, observe that the digraph in Figure 1 has a unique postman walk of length $w+x+y+z+6a+b$ even though it contains a three cycles. The postman walk of length $w+x+y+z+9a+b$ is not unique, however. Although the problem of constructing a postman walk of length l in a weighted graph is NP-complete, we present an algorithm which tests the uniqueness of

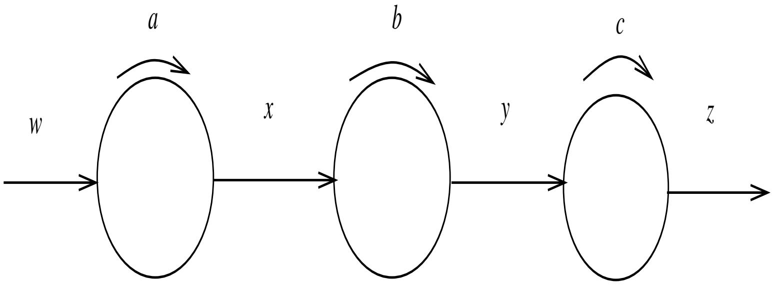

Figure 1: A non-trivial graph with a unique postman tour of length $w + x + y + z + 6a + b$.

a postman walk of given length l in polynomial-time for any unweighted digraph. This new algorithm, based on a restricted-size variant of the knapsack problem, significantly increases the resolving power of classical sequencing chips.

In Section 2, we survey previous work in sequencing by hybridization in more detail. In Section 3, we present our results on reconstructing strings using string and subsequence queries. Our results on testing uniqueness of postman walks appears in Section 4. We conclude with several open problems.

2 Sequencing by Hybridization

Sequencing by hybridization (SBH) is a new and promising approach to DNA sequencing which offers the potential of reduced cost and higher throughput over traditional gel-based approaches. In 1991, Strezoska, et.al. [29] accurately sequenced 100 base pairs of a known sequence using hybridization techniques, although the approach was proposed independently by several groups, including Bains and Smith [2], Drmanac and Crkvenjakov [6], Lysov, et.al [19], Macevicz [20], and Southern [27]. More recently, Crkvenjakov's and Drmanac's laboratories report sequencing a 340 base-pair fragment in a blind experiment [24].

In the classical sequencing chip $C(m)$, all 4^m single-stranded oligonucleotides of length m are attached to the surface of a substrate. For example, in $C(8)$ all $4^8 = 65,536$ octamers are used. Other, non-classical sequencing chip designs are possible. The economies of scale and parallelism implicit in performing thousands of hybridization experiments simultaneously are major advantages of prefabricated sequencing chips. The mass production of a prefabricated array can be used to amortize the high start-up costs of such a procedure, as is the case in the manufacture of VLSI chips. Indeed, Fodor, et. al. [8] used photolithography techniques typical of the semiconductor industry to fabricate the array $C(5)$ of 1024 peptides in only ten steps. Larger chips are currently being developed, and by the analogy with the semiconductor industry chip capacity can be expected to continue to grow exponentially for several years.

Interpreting the data is where the algorithmic aspect of sequencing by hybridization arises. The outcome of an experiment with a classical sequencing chip $C(m)$ is a probability that each of the 4^m strings is a substring of the underlying sequence S. In an experiment without error, the probabilities will all be 0 or 1, so each m-nucleotide fragment of S is unambiguously identified.

The problem of finding the shortest superstring containing each of a given set of strings is known to be NP-complete [9]. Thus finding an optimal solution is computationally intractable, even though approximation algorithms are known [3, 14, 18]. However, efficient algorithms do exist for finding the shortest string consistent with the results of a classical sequencing chip experiment. In particular, Pevzner's algorithm for sequencing chip reconstruction [25] is based on finding Eulerian paths in a subgraph of the de Bruijn digraph [5]. For a given alphabet Σ and length k, the *de Bruijn digraph* $G_k(\Sigma)$ will contain $|\Sigma|^{k-1}$ vertices, each corresponding to a $(k-1)$-length string on Σ. As shown in Figure 2, there will be an edge from vertex u to v labeled $\sigma \in \Sigma$ if the string associated with v consists of

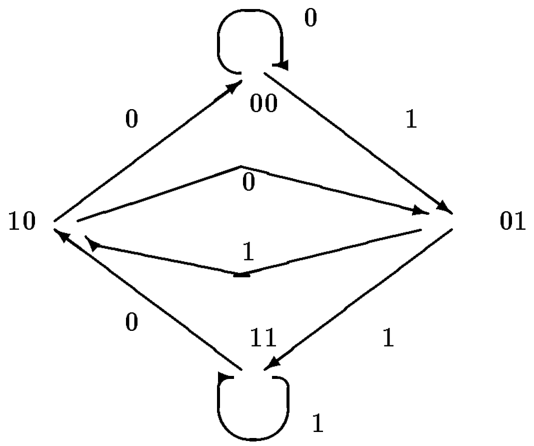

Figure 2: The de Bruijn digraph $G_3(\{0,1\})$.

the last $k-2$ characters of u followed by σ. In any walk along the edges of this graph, the label of each vertex will represent the labels of the last $k-1$ edges traversed. Accordingly, each directed edge (u,v) of this graph represents a unique string of length k, defined by the label of u followed by the label of (u,v).

Since the digraph is strongly connected, and the in-degree of each vertex equals its out-degree, there is an Eulerian cycle in the digraph, ie. a tour which visits each edge exactly once. Further, Eulerian cycles are algorithmically easy to find if they exist. The string defined by the labels of the edges traversed will be shortest string which contains every k-string on Σ as a substring.

A hybridization experiment with the sequencing chip $C(k)$ identifies which k-strings are and are not in S. For each string which is not, we will delete the appropriate edge from $G_k(\Sigma)$. Any postman walk (a walk visiting each edge at least once) on the remaining graph is a possible sequence. If the remaining graph consists of a single Eulerian path, as in Figure

3(a), then the sequence S is completely determined by $C(k)$. However, whenever there is a

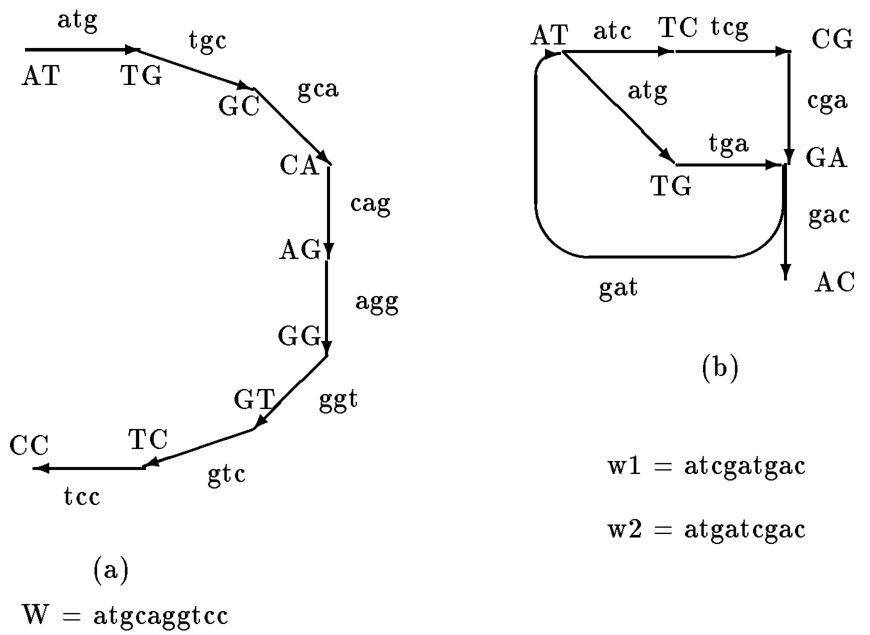

Figure 3: Sequence reconstruction as Eulerian paths in a de Bruijn digraph.

single node of in- or out-degree two, S is not uniquely defined by the data, as is the case in Figure 3(b). This is not an uncommon situation – as mentioned above $C(8)$ suffices to reconstruct 200 nucleotide long sequences in only 94 of 100 cases [23], even in error-free experiments.

We believe that interactive sequencing by hybridization can become a viable technique, perhaps for disambiguating between a relatively small number of possible sequences. Our vision is not far removed from Crkvenjakov and Drmanac's target down approach to sequencing by hybridization [6, 24], which fastens multiple *target* fragments to the substrate, and then uses a single probe to hybridize with each target simultaneously. They use a static sequence of probes, instead of choosing probes interactively based on the outcomes of the probes to date. The efficiencies gained by such interactive reconstruction may make the technique practical for substantially smaller number of targets.

The main barrier to practical interactive sequencing by hybridization is the cost of obtaining the query oligonucleotides. The combinatorial explosion prohibits storing all 4^k primers for even modest-sized k, and synthesizing primers is expensive and difficult. However, Kieleczawa, Dunn, and Studier's recent primer walking technique [15] suggests that strings of three to four hexamers can be used to construct large probe strings cheaply.

3 Reconstructing Unknown Strings

In this section, we consider the problem of reconstructing strings from substring queries. It should be clear that any string can always be so reconstructed. Suppose the length of the string $n = |S|$ is known in advance. All $|\Sigma|^n$ strings can be tested as substrings of S, and the only 'yes' answer defines S. If the length of S is unknown, we can try all strings of length i starting at $i = 1$. We terminate when there is only one matched substring of length n and none of length $n+1$. However, we seek to determine S using as few questions as possible.

Any interactive strategy for determining strings can be specified by a *decision tree* [21]. A decision tree is a rooted binary tree, where each internal node is labeled by a substring query and each leaf by a candidate string. For each node, all leaf nodes of the left subtree contain the given query substring, while none of the leaf nodes of the right subtree do. Figure

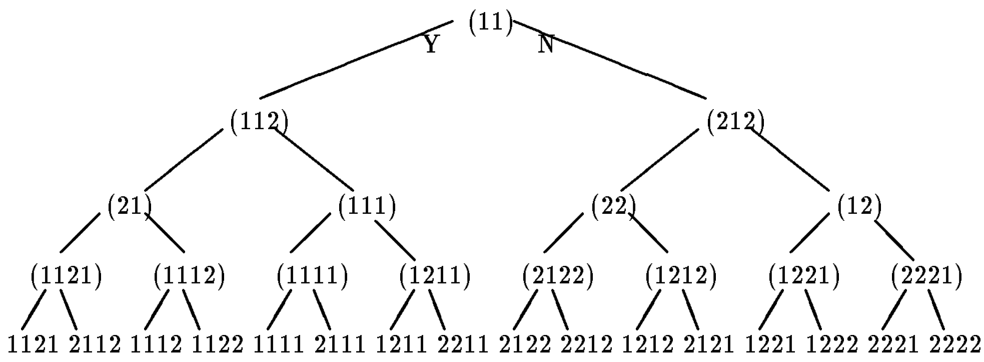

Figure 4: An optimal decision tree for four-character binary strings.

4 gives an optimal decision tree for four-character binary strings. This tree is of height 4, which is clearly optimal for any binary tree with 16 leaves. Such perfect binary trees are not always possible - consider the set of all two-character binary strings, $\{11, 12, 21, 22\}$, or all binary strings of length six. For each node in the tree, we seek a substring query which partitions the set of candidate strings as evenly as possible. Guibas and Odlyzko [12] and Wilf [30] consider the problem of counting the number of strings with a given set of substrings and forbidden substrings. However, the resulting formulae are far too cumbersome apply to constructing large decision trees.

There are two different problems we will consider. In Section 3.1, the unknown string may be any string on the alphabet Σ of prescribed or unprescribed length. A lower bound for reconstruction via substrings is given in Section 3.2. In certain applications either context or previous experiments may have already reduced the set of candidates down to a small set of strings. Therefore, in Section 3.3, we consider the algorithmic problem of constructing good decision trees for arbitrary restricted sets of strings. An algorithm for subsequence queries is given in Section 3.4.

We use the following notation for strings. The *length* of a string S, $|S|$, is the number of characters it contains. We use α to denote the length of the alphabet. The string representing the concatenation of strings a and b will be denoted ab. If $S = ab$, then a is a prefix of S and b a suffix of S – note that a string of length n has n distinct non-empty prefixes and n non-empty suffixes. If $S = ab$, then $S - b = a$. The string formed by concatenating i copies of s will be denoted s^i.

3.1 Unrestricted Candidate Strings

In this section, we consider interactive strategies for determining unknown, unrestricted strings. Suppose the string is known to be of length n. Any such strategy can be represented by a decision tree with α^n leaves. This exponential growth prohibits the explicit construction of decision trees for even modest-length strings, and so our strategies are presented as algorithms which generates the next query in response to the results of previous queries. We can use lower bounds on the height of decision trees to assess how good the worst-case complexity of our strategies are.

Another subtlety of the problem is whether the length of the unknown string is presented in advance, or must be determined using the results of queries. We shall see that a somewhat simpler and more efficient strategy is possible if n is known, although the worst case complexity of both strategies are identical except for lower order terms.

Lemma 1 *$n \lg \alpha$ substring queries are necessary to determine an unknown string S on alphabet Σ, where $|S| = n$ and $|\Sigma| = \alpha$.*

Proof: By an information theoretic argument, $\lg(\alpha^n)$ bits are requires to specify S. The result follows since the result of each substring query provides at most one bit of information about S. ∎

Theorem 2 *An unknown string S of known length n can be reconstructed in $(\alpha - 1)n + 2\lg n + O(\alpha)$ substring queries. If n is unknown, $(\alpha - 1)n + \Theta(\alpha\sqrt{n})$ queries suffice.*

Proof: First we show a slightly weaker result, that any unknown string of known length can be reconstructed in $\alpha(n+1)$ queries. Begin by making substring queries of single character substrings, so after at most α queries we know a character of S. Let s be a known substring of S and $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_\alpha\}$. In general, we can increase the length of this known substring by one character by querying on the strings $s\sigma_i$, for $1 \leq i \leq \alpha$. At least one of these query strings must be a substring of S, unless s is a suffix of S. When s can no longer be extended, s is a suffix of S and we can continue the process by prepending each character to the known substring, until it is of length n and S is determined.

Now we show how to reduce the multiplicative constant by one. By performing queries emulating a binary search, in $\lg(n+1)$ queries we can determine largest l such that ${\sigma_\alpha}^l$ is a substring of S. If $l > 0$, initialize s to ${\sigma_\alpha}^l$, otherwise use another $\alpha - 1$ queries to determine a single character of S.

Now suppose that s is a known substring of S, but $s\sigma_i$ is not, for all $1 \leq i < \alpha$. Therefore, either s is the suffix of S or $S\sigma_\alpha$ is a substring of S. We will assume the latter and continue with the procedure. If this suffix of σ_α's reaches length $i+1$, we must have extended past the right end of S, so $s = s'{\sigma_\alpha}^j$ where $0 \le j \le i$ and s' is a string whose last character is not σ_α. If i is sufficiently small, we can find j and thus the right end of S using a linear search with $j+1$ queries. If i is sufficiently large, it is more efficient to performing a binary search, determining the longest such substring s of S in $\lg(i+1)$ queries. We continue this phase until s is a suffix of S. With the suffix and n known, we repeatedly extend s by one prepended character (using $\alpha - 1$ queries) until S is completely determined.

To analyze the complexity of this strategy, observe that each character of s is determined using at most $\alpha - 1$ queries. An additional $(l - j + 1) \cdot (\alpha - 1)$ queries may have been "wasted" determining the non-existant suffix of S, but if so we may charge this to our savings in determining the first l characters of S in the first $\log(n+1)$ queries, leaving a excess of only $O(\alpha)$ queries.

If n is unknown, we will use a different strategy to identify the ends of S. After finding a single character of S in α queries, we will extend the known substring s by one character to the right in $\alpha - 1$ queries as before, provided s does not end with an unverified suffix of ${\sigma_\alpha}^{\sqrt{m}}$, where $m = |s|$. If it does, we will issue a query to verify s. If $s \in S$ the procedure continues, but if not it means that the string of unverified σ_α's is terminated by the right end of S. The correct suffix can be determined using $\sqrt{m}$ queries of the form $s - {\sigma_\alpha}^i$ for $1 \le i \le \sqrt{m}$. We can then repeat the previous procedure with suffix replaced by prefix to determine the rest of S.

In general, $\alpha - 1$ queries are used to determine each character of S. At most $2\alpha\sqrt{n}$ queries are wasted building the non-existant prefix and suffix, independent of the T_n verification queries, where $T_n = T(n - \lceil\sqrt{n}\rceil) + 1$ and $T_1 = 1$. We observe that each query for $|s| > n/2$ verifies at least $\sqrt{n/2}$ characters, and so at most $(n/2)/\sqrt{n/2} = \sqrt{n/2}$ queries are necessary to verify the final $n/2$ characters of S. Therefore,

$$T_n \le \sum_{i=1}^{\lg n} \sqrt{n/2^i} = \sqrt{n} \sum_{i=1}^{\lg n} 1/2^{i/2} \le 2\sqrt{n} \sum_{i=1}^{(\lg n)/2} 1/2^i \le 4\sqrt{n}$$

so $(\alpha - 1)n + \Theta(\alpha\sqrt{n})$ total queries suffice to determine S. ∎

We note that the bounds in Lemma 1 and Theorem 2 are tight for binary strings and when $|S| = 1$.

3.2 A Lower Bound for Reconstruction

Lemma 1 gave an information theoretic argument that $n \lg \alpha$ substring queries are necessary to determine an unknown string S of length n. We also showed that $n(\alpha - 1) + \Theta(\alpha\sqrt{n})$ queries suffice. In this section, we give an adversarial argument to show that $n(\alpha - 3)/4$ substring queries are necessary, approaching our upper bound within a factor of 4. Further, $(n\alpha/4)(1 - o(1))$ queries remain necessary even under a stronger model, which returns the number of times a query string occurs in S.

For ease of presentation, let us initially assume that S is a *circular* string of length $n = \alpha^m$, for some m. To prove the lower bound, we formulate the problem of determining S

as a two-person query game. Holmes and Moriarty play a game in which Holmes (the great detective) seeks to determine the unknown string S, where Moriarty chooses S and acts as an oracle, replying either 'yes' or 'no' to each of Holmes' substring queries. The game is over once Holmes finds out S. Of course, Moriarty is Holmes' adversary in Conan Doyle's famous mystery stories.

The game proceeds in discrete steps. Holmes tries to minimize the length of the game, Moriarty to maximize it. Many search problems can be formulated as such two-person query games. For example, sorting can be considered as a two person query game in which we seek to find an unknown permutation f over $\{1,\ldots,n\}$ by asking queries of the form 'Is $f^{-1}(i) < f^{-1}(j)$?'. Ko and Teng [16] consider a generalization of the game Mastermind, specifically the problem of identifying an unknown permutation f by asking permutation queries, in which the adversary replies in how many places f and query permutation q coincide.

We assume that both Holmes and Moriarty have infinite computing power. Moriarty does not have to choose S at the start of the game, but may instead maintain the set Q of strings consistent with the results of all previous queries, i.e., he plays an adversary strategy. Provided Q is non-empty, he can always choose any sequence from Q for S.

Suppose Moriarty tells Holmes at the beginning of the game that S is a de Bruijn sequence $d(\alpha,m)$. A de Bruijn sequence $d(\alpha,m)$ of span m is a circular sequence of length α^m, which contains all the strings of length m exactly once. For example, 0000100110101110 and 0000101001101110 are two distinct de Bruijn sequences of span 4 on the binary alphabet $\{0,1\}$.

Let $D(\alpha,m)$ denote the set of distinct de Bruijn sequences on α. It is well known [5, 26] that

$$|D(\alpha,m)| = ((\alpha-1)!)^{\alpha^{m-1}}\alpha^{\alpha^{m-1}-m} = \frac{1}{n}(\alpha!)^{n/\alpha}$$

Good [11] demonstrated how to construct the sequences of $D(\alpha,m)$, by building a directed graph $G(\alpha,m)$ where the vertex set of $G(\alpha,m)$ represents each string of length $m-1$ on α. Create a directed edge between $x = x_1 \ldots x_{m-1}$ and $y = y_1 \ldots y_{m-1}$ iff $x_i = y_{i-1}$ for $i = 2,\ldots,m-1$. Therefore, each edge is labeled by a string of length m, namely xy_{m-1}. In summary, G contains $\alpha^{m-1}(= n/\alpha)$ vertices and $\alpha^m(= n)$ edges, with each vertex of indegree and outdegree α. There exists a 1-1 correspondence between $D(\alpha,m)$ and the set of all Eulerian circuits of G. Hence, finding an unknown de Bruijn sequence in $D(\alpha,m)$ is equivalent to finding an unknown Eulerian circuit in $G(\alpha,m)$.

Let f_v be a 1-1 function, $f_v : \Sigma \to \Sigma$. An Eulerian circuit c of $G(\alpha,m)$ and a vertex v defines such a 1-1 function where $f_v(x)$ is mapped to y when we enter the vertex $v = x_1 \ldots x_{m-1}$ in c by the edge xv and leave v by the edge vy. Thus c defines a set of $|V|$ 1-1 functions $F = \{f'_1,\ldots,f'_{n/\alpha}\}$. Note that from the equation for $|D(\alpha,m)|$, the converse is not true, i.e., an arbitrary set of functions may not give an Eulerian circuit because the implied circuit may not be connected.

We can represent f_v as an $\alpha \times \alpha$ permutation matrix. To keep track of the information Moriarty has revealed to Holmes, he maintains the state of F using a set of n/α $\alpha \times \alpha$ matrices $f_1,\ldots,f_{n/\alpha}$. At any moment in time, each bit of each f_v in F is either 'yes', 'no', or 'unknown'. If $f_v[x,y] = \text{'}unknown'$, then Holmes does not know whether xvy is a substring of S. If $f_v[x,y] = \text{'}no'$, then S does not contain xvy as a substring. If the $f_v[x,y] = \text{'}yes'$, then xvy is a substring of S. There can be exactly one 'yes' in each row and column. Thus each matrix element determines whether a specific $m+1$ length string is contained in s.

Initially, Q corresponds to the set of all Eulerian circuits of $G(\alpha, m)$, so F corresponds to n/α matrices with all bits set to 'unknown'. At the end of the game Q contains exactly one Eulerian circuit, corresponding to S, and all bits in the matrices of F are either 'yes' or 'no', with the property that there is exactly one 'yes' bit in every row and column.

We now we propose a simple strategy for Moriarty to answer queries. Let q be Holmes' query string. For a query q with $|q| \leq m$, Moriarty will always say 'YES', because any de Bruijn sequence in $D(\alpha, m)$ contains all substrings of length m. Therefore, from Holmes's perspective it never pays to ask queries of length $\leq m$. Suppose $|q| \geq m+1$. Let $q_1, \ldots, q_{|q|-m}$ be the $(m+1)$-length substrings of q such that q_i appears before q_j in q, for $i < j$. Note that these q_i's must be distinct, since S is a de Bruijn sequence. Let z be the smallest index such $f_v[x,y] = \text{'}unknown'$, where $q_z = xvy$ and $v = x_1 \ldots x_{m-1}$, i.e., z is smallest index such that Holmes does not know whether S contains q_z or not. Moriarty will reply 'YES' if all the matrix elements corresponding to all substrings in q are 'yes' and respond 'NO' if there is one entry corresponding to a substring q is 'no'.(Again, from Holmes's perspective it does not pay to ask queries when everything is known about the query).

When Moriarty replies 'NO' and z exists, he changes the state of F by:

1. Removing all strings from Q which contains q_z as a substring.

2. Setting $f_v[x,y] = \text{'}no'$.

3. Maintaining a row-column constraint. If there is a partition of Σ into B_1, B_2 such that the all entries of the induced submatrix B_1B_2 of f_v are 'no', then all entries in the induced submatrix B_2B_1 of f_v are set 'no'.

4. A mapping $y = f_v(x)$ is said to be *forbidden* if all circuits in Q do not contain xvy as a substring, where $v = x_1 \ldots x_{m-1}$. Change all forbidden mappings from 'unknown' to 'no'.

5. Recur on the changes until the state of the matrix reaches a fixed point.

Figure 3.2 reflects the state F at a given instant in time.

Therefore, Moriarty interprets Holmes's question as a single query of length $m+1$, and answers accordingly. (Therefore, from Holmes's point of view it always suffices to ask queries of length $m+1$.)

Lemma 3 *Moriarty's strategy always maintains a non-empty set Q of consistent Eulerian cycles.*

Proof: It is sufficient to prove that Q is non-empty when there are any 'unknown' entries in any matrix of F. The state of the strategy is certainly feasible at the start of the game.

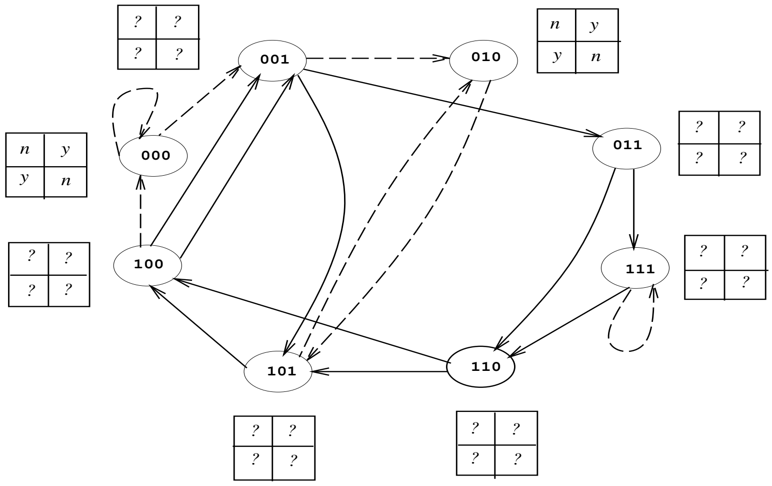

Figure 5: The graph $G(2,3)$ and representative matrices.

Now assume the state is feasible immediately before a query q. If Moriarty replies 'YES' to query, he does not update the set Q, so the state remains feasible after the query. Otherwise, there must exist a f_v such that at least one row x of f_v contains at least two 'unknown's, say (x, y_1), (x, y_2). Therefore, Q must contain at least two Eulerian circuits with mappings $y_1 = f_v(x)$, $y_2 = f_v(x)$. Since Moriarty marks all forbidden mappings 'no' after each query, the state remains feasible after the query. ∎

How many queries are asked by Holmes before all entries in the matrices of F are filled with 'yes' and 'no's? By the end of the game, all bits in all the n/α matricies f_v have been set to 'no' and 'yes'. We consider three classes of 'no' bits:

1. The set of *green* bits G_v filled by Moriarty to maintain the the row-column constraint that each row and column in f_v contains exactly one 'yes' bit.

2. The set of *red* bits R_v, corresponding to all the forbidden mappings.

3. The set of *blue* bits B_v, corresponding to actual queries by Holmes for which Moriarty answered 'no'. Obviously, the number of blue bits represents a lower bound on Holmes' queries.

Finally, let Y_v denote the bits which are set 'yes'. We know that

$$|B_v| + |G_v| + |R_v| + |Y_v| = \alpha^2,$$

as there are exactly α^2 bits in f_v. Since each row and column contain exactly one 'yes' entry, $|Y_v| = \alpha$.

First, we give a bound on the number of green bits in a matrix f_v.

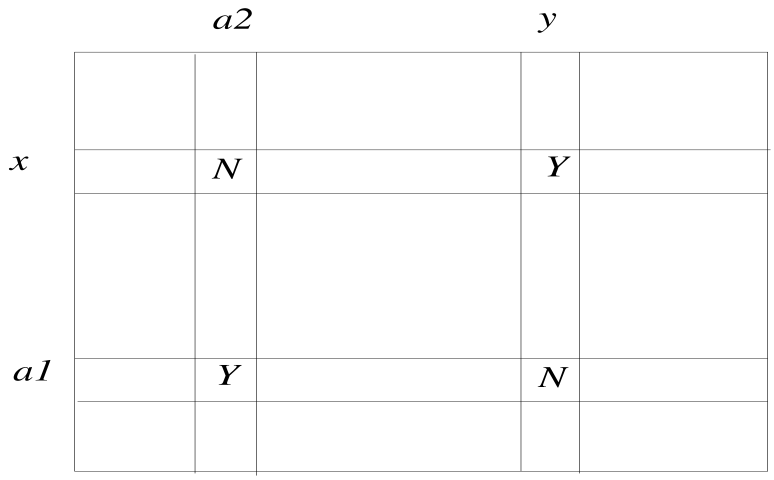

Figure 6: At most $\alpha(\alpha-1)/2$ bits are green in f_v.

Lemma 4 *In any matrix f_v, $|G_v| \leq \binom{\alpha}{2}$.*

Proof: Consider the state of f_v after the game. Let $f_v[x,y]$ and $f_v[a_1,a_2]$ be 'yes' bits, where $x \neq a_1$ and $y \neq a_2$. Suppose $f_v[x,a_2]$ is a green bit. We claim that $f_v[a_1,y]$ is either blue or red. First observe that $f_v[a_1,y] \neq$ 'yes', for that would create two 'yes' bits in one column. Suppose, $f_v[a_1,y]$ were green. Since green bits correspond to unasked queries, we can change $f_v[x,a_2]$ and $f_v[a_1,y]$ to 'yes' bits, and $f_v[a_1,a_2]$ and $f_v[x,y]$ to 'no' bits, resulting in a different consistent solution and contradicting the fact that Moriarty set both $f_v[x,a_2]$ and $f_v[a_1,y]$ to green because of a row-column constraint. Refer Figure 3.2. Therefore, $f_v[a_1,y]$ must be either a blue bit or a red bit, and the total number of green bits in the row x and column y is at most $\alpha-1$, because any green bits have a corresponding non-green 'no' bit. Removing the xth row and the yth column from f_v and applying the argument recursively gives at most $\sum_{i=1}^{\alpha-1} i = \binom{\alpha}{2}$ green bits. ∎

Lemma 4 is interesting in its own right. Suppose, we want to identify an 1-1 function, $f : \Sigma \rightarrow \Sigma$, by asking queries of the form 'Is $y = f(x)$?'. Then, from Lemma 4 $\binom{|\Sigma|}{2}$ queries are necessary and sufficient.

Therefore, there are at least $\binom{\alpha}{2}$ red and blue bits, correspond to forbidden mappings and actual 'NO' queries, respectively. Now, we characterize the forbidden mappings and thus give an upper bound on the number of forbidden mappings created during the game.

To characterize the forbidden mappings, let us consider the graph G which evolves dynamically as the game proceeds. Initially, $G = G(\alpha, m)$. Whenever there is a mapping $y = f_v(x)$ implying xv is mapped to vy, then xv and vy are removed from G and replaced by one edge from x to y, labeled xvy. Whenever Moriarty replies 'NO' to a query 'Is xv

mapped to vy?', the edges xv and vy cannot be replaced by an edge xvy in G. Isolated vertices are discarded from G as they are created, and by the end of the game G consists a single vertex with a self loop labeled S. With the exception of isolated vertices, we do not permit G to become disconnected, because that would leave S as two disconnected Eulerian circuits and hence not a de Bruijn sequence.

Under what condition can G possibly become disconnected? After a series of queries, articulation vertices may appear in G. Suppose there exists an articulation vertex v such that the components of $G - v$ can be partitioned into two sets, S_1 and S_2, such that all edges in $S_1 \cup \{v\}$ which are incident on v can only be mapped to edges within $S_1 \cup \{v\}$. In this case it is clearly impossible to traverse all the edges in G exactly once. Therefore, if an articulation vertex is created in G during the game, Moriarty must ensure that this situation does not arise. For example, suppose only one remaining edge xv in $S_1 \cup \{v\}$ can be mapped into an edge in $S_2 \cup \{v\}$. Then, Moriarty must forbid all assignments of the form $y = f_v(x)$, for all edges yv in $S_1 \cup \{v\}$. Therefore, all such bits in f_v are set to 'no' and colored red. We claim that all forbidden mappings are mappings of this form.

Lemma 5 *Forbidden mappings are created only when there exists an articulation vertex v in G such that the components of $G - v$ can be partitioned into two sets, S_1 and S_2, such that there exists a vertex $x \in S_1$ such that for all $v_1 \in S_1 - x$ and all $v_2 \in S_2$, $f_v(v_1) \neq v_2$. Further, these forbidden mappings are exactly $f_v(x) \neq v_1$ for all $v_1 \in S_1$.*

Proof: We use induction on the number of vertices of G. Suppose G contains an articulation vertex v, then let $C_1, \ldots, C_r$ denote the connected components resulting from the removal of v. Remove v from G, and in each component C_i, pair off the vertices incident and adjacent to v so that each component remains balanced. There always exists such an assignment as the indegree and outdegree of v in each component must be equal. By the induction hypothesis, the only forbidden mappings in each component are as stated in the lemma, since a mapping in component C_i is independent of any mapping in any other component. As there is a valid mapping associated with v so that all the components can be connected, a set of Eulerian circuits in the components connected through v will give an Eulerian circuit in G.

Otherwise, G does not contain an articulation vertex. We claim no mapping is a forbidden mapping. Choose an arbitrary vertex v of G. Any mapping $f_v = \Sigma \to \Sigma$ provides a consistant mapping for G, and leaves v as an isolated vertex. Applying the induction hypothesis to the resulting graph, we show that none of the mappings are forbidden with v. Since, this property holds true for all vertices, none of the mappings in this case are forbidden. Thus, all forbidden mappings are the mappings are as stated in the lemma. ∎

For a vertex v, we give a bound on the number of such forbidden mappings created in f_v, during the game. Consider the first set of forbidden mappings created in f_v. See Figure 3.2. Suppose xv is the only edge in $S_1 \cup \{v\}$ that can be mapped to an edge in $S_2 \cup \{v\}$. Then the forbidden mappings created are $y_i = f_v(x)$, $y_i \in S_1$, such that $f_v[x, y_i] =$ '*unknown*'. Clearly, there can be at most $|S_1|$ such mappings. Then, we claim that at least $(|S_1| - 1) \cdot |S_2|$ blue bits are set in f_v, corresponding to the bits $a_i = f_v(b_j)$, for all edges a_i, v in $S_1 \cup \{v\}$ and v, b_j in $S_2 \cup \{v\}$, with $a_i \neq x$, since these are the first forbidden mappings created with

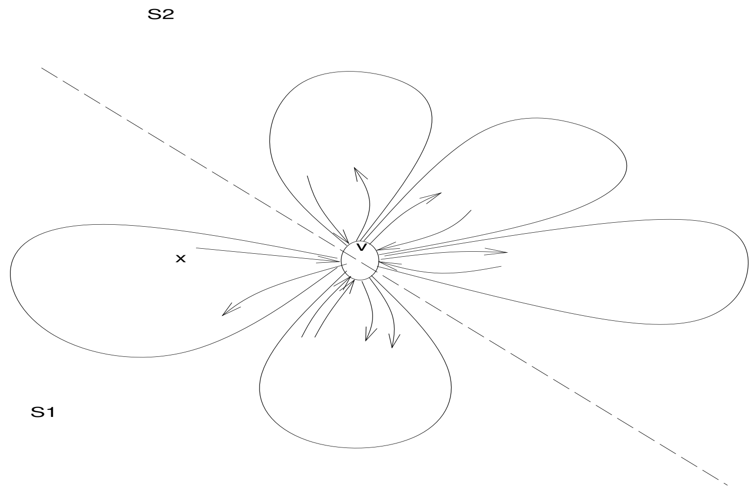

Figure 7: Forbidden mappings associated with vertex v.

the vertex v (therefore they cannot be red) and an edge from $S_1 \cup \{v\}$ can be mapped to an edge in $S_2 \cup \{v\}$ without violating the row-column constraint (therefore they cannot be green). Since $|S_2| \cdot (|S_1| - 1) \geq |S_1| + 1$, the number of red bits created for the first time is at most one more than the number of blue bits.

Observe that, from the row-column constraint, the edge (x, v) in $S_1 \cup \{v\}$ must eventually be mapped to an edge in $S_2 \cup \{v\}$. Also, observe that only one edge from S_1 can be mapped into S_2 and vice versa. Once, an assignment from S_1 into S_2 and S_2 into S_2, $b_i = f_v(a_j)$ such that, a_j in S_1 and b_i in S_2 will be charged to green bits.

Suppose a set of forbidden mappings associated with the vertex v is created at a later time. Then

Lemma 6 *Let S_1, S_2 and T_1, T_2 represent the partitions of $G - v$ creating forbidden mappings in f_v, where (1) the forbidden mappings associated with S were created prior to those of T and (2) the vertex x defining the forbidden mapping belongs to T_1, not T_2. Then either $T_1 \subset S_1$ or $T_1 \subset S_2$.*

Proof: The lemma follows from the necessary and sufficient condition that S_1 and S_2 must be connected components. ∎

Hence the only forbidden mappings created next time are within the components S_1 and S_2, since all edges except two between S_1 and S_2 were already charged as 'blue' bits or will be charged as 'green' bits, and the two edges which connect S_1 and S_2 can be connected anywhere within component. Hence, we can apply the argument to both S_1 and S_2 separately.

Hence, each time a set of forbidden mappings are created, the number of forbidden mappings is at most the number of blue bits gained plus one. There can be at most α sets of forbidden mappings created as each vertex has degree of α. Therefore,

Lemma 7 $|R_v| \leq |B_v| + \alpha$.

and from Lemma 4 $|B_v| \geq \alpha(\alpha - 3)/4$ so

Corollary 1 *Reconstructing a circular string of length n on Σ requires at least $n(\alpha - 3)/4$ substring queries.*

Theorem 8 *$\alpha n/4 - O(n)$ substring queries are necessary to reconstruct a string of length n, even if the frequency of the query substring is also returned.*

Proof: In the previous discussion we assumed that the unknown string S was a circular string. We may relax this assumption by defining, for any Eulerian circuit c, an associated linear string S_c of length $|c| + m$. Let σ be a character in Σ. S_c begins and ends with σ^m, corresponding to breaking c at a specific self-loop in G. The $m + 1$ substrings of S_c are identical to that of S.

Now suppose that each substring query returns a frequency count, instead of 'yes' or 'no'. Observe that a query string q occurs at most once in $G(\alpha, m)$ if $|q| > m$, and exactly $\alpha^{m-|q|}$ times if $|q| \leq m$. In S_c, a string of length q occurs at most once if $|q| > m$, and occurs $\alpha^{m-|q|}$ unless it is of the form σ_1^l, which occurs exactly $\alpha^{m-|q|} + (m - l + 1)$ times. Thus returning the frequency count returns no additional information about S_c, and the lower bound result holds for the stronger model. ∎

3.3 Restricted Candidate Strings

In certain applications, we may have much more knowledge about an unknown string S than just its alphabet and length. For example, there may be a previously defined set of forbidden substrings, none of which can be in S. More generally, we can consider the problem where we are given as input a set of candidates C, and limit the problem to identifying which member of C is S.

Such *model-based* recognition problems occur frequently in testing and classification procedures. In general, we are given a finite set of candidates $C = \{C_1, \ldots, C_n\}$ and a finite set of tests $T = \{T_1, \ldots, T_m\}$, where $T_i \in C$. Each test distinguishes the candidates in T_i from those in $C - T_i$. There always exists a decision strategy for identifying the right candidate from C, provided that there exists at least one test T_k for each pair of candidates C_i and C_j such that $C_i \in T_k$ and $C_j \in C - T_k$. Such a decision strategy is given by a decision tree of height at least $\lceil \lg n \rceil$.

Hyafil and Rivest [13] proved that the problem of constructing a minimum height or minimum path-length decision tree is NP-complete. Despite this result, there is some hope for being able to construct optimal decision trees for special types of models and queries. For example, optimal decision trees for non-degenerate polygonal models all sharing a common point can be efficiently constructed [1], although the problem becomes hard if either the common point or degeneracy assumption is removed.

In this section, we show that the minimum height decision tree problem remains NP-complete for strings and substring queries. Fortunately, we give an efficient approximation algorithm which gives provably good trees, within a multiplicative constant of the height of the optimal tree for finite-sized alphabets.

Theorem 9 *Building a minimum height decision tree for substring queries is NP-complete.*

Proof: Our proof is based on the reduction of Hyafil and Rivest [13], who transform exact-cover-by-three-sets to minimum height decision tree. In exact-cover-by-three-sets, we are given as input a universal set U and a set V of three element subsets of U. We seek a subset of $|U|/3$ elements of V whose union is U.

In the reduction of [13], the set of candidates $C = U \cup \{a, b, c\}$, where a, b, c are three elements not in U. The test set T consists of union of V and the set of singleton sets for C. They show that this problem has a decision tree of height $|U|/3 + 2$ iff there exists an exact cover for the original instance. We will construct a set of $n = |C|$ strings on an alphabet of $|V| + 3$, such that the only non-trivial substring queries emulate members of T.

The characters of the ith string, for $1 \leq i \leq |U|$ correspond to the three-element subsets of V containing U_i, so exactly 3 strings contain any such character. The last three strings are each only one character long, consisting of the only occurence of each of the remaining three characters, representing a, b, and c.

Observe that the only substring queries contained in at least three strings are the single character strings cooresponding to tests. Further, these characters are in exactly three strings. By the analysis of [13], there is a decision tree of height $|U|/3 + 2$ iff there is an exact-cover-by-three-sets. ∎

Since the problem is NP-complete, we seek heuristics which can deliver provably good although non-optimal height trees. The most natural strategy is the greedy heuristic, which for each internal node selects the substring which partitions the candidate set as evenly as possible. It has been shown [1] that the greedy tree has height at most $\lg m$ times that of the optimal tree, where m is the number of objects to be distinguished, and that in general this bound is tight. However, for the special case of substring queries, we give a constant factor approximation algorithm, with the constant a function of α.

Lemma 10 *Let M be a set of strings on alphabet Σ. There exists a string s which is contained in at least $|M|/\alpha$ strings from M.*

Proof: Since the substring relation is transitive, the most ubiquitous substring must consist of a single character. Each string is comprised of at least one character, so the average character in Σ appears in at least $|M|/\alpha$ strings. ∎

Lemma 11 *Let M be a set of strings with a common substring s. Then there exists a string longer than s common to at least $|M|/(2\alpha)$ of the strings.*

Proof: If s is a proper substring of S, then it may be a prefix or suffix of S or else contained in the interior of S. With multiple occurrences of s in S, all three conditions are possible. Provided s is not a suffix of S, then $s\sigma_i$ is contained in S for some character $\sigma_i \in \Sigma$. If s is not a prefix of S, then $\sigma_i S$ is contained in S for some character $\sigma_i \in \Sigma$. At least half of the strings of M are either not prefixes or suffixes, and by the pigeonhole principle at least $1/\alpha$ of these must be extended by the same character. ∎

Lemma 12 *For any set of strings M on alphabet Σ, there exists a substring s which is contained in at least $|M|/(2\alpha+1)$ and at most $2\alpha|M|/(2\alpha+1)$ strings.*

Proof: Suppose we have a set M' of strings containing a common substring s. By Lemma 11, one of the 2α single character extensions of s is contained in at least $|M'|/(2\alpha)$ and at most $|M'|$ strings.

We will continue extending s until $|M|/(2\alpha+1) \leq |M'| \leq 2\alpha|M|/(2\alpha+1)$. Initially, choose s to be the character occurring in the largest number of strings – by Lemma 10 $|M'| \geq |M|/\alpha \geq |M|/(2\alpha+1)$. At each iteration, $|M'|$ is reduced by a factor of at most $1/(2\alpha)$ and eventually goes to 1, so at some point M' must be of the prescribed size. ∎

Theorem 13 *For any set of strings M on an alphabet Σ, there exists a decision tree of height at most $\log_{(2\alpha+1)/(2\alpha)}|M|$. Further, such a tree can be constructed in $O(N \lg |M|)$ time, where N is the total number of characters in M.*

Proof: By constructing s according to Lemma 12, we obtain a substring which partitions at least $2\alpha/(2\alpha+1)$th of the largest subset of strings at each level of the tree. A tree with such a partition at each node has the specified height.

For the time analysis, we can scan through each string and establish pointers to one instance of each of at most α distinct characters in $O(N)$ time. After picking the most common character, we can check the one character extensions of each string in constant time. A total of at most N extensions are performed before we identify the desired subset and partition the strings.

At each subsequent level of the tree, we work with smaller subsets totaling at most $|M|$ strings with at most N characters, so $O(N)$ time suffices. The result follows since the resulting tree has height $O(\lg |M|)$. ∎

We note that the strategy of Theorem 13 can be applied to the set of all α^n strings, providing an alternate solution to the problem of Section 3.1. However, the resulting constants are inferior to those of Theorem 2.

3.4 Reconstruction via Subsequence Queries

Given two sequences $a = a_1 \ldots a_n$ and $b = b_1 \ldots b_m$, with $m \leq n$, we say b is a *subsequence* of a if for some $1 \leq i_1 < \ldots < i_m \leq n$, we have $a_{i_h} = b_h$, for all h, $1 \leq h \leq m$. In this section, we consider the problem of reconstructing an unknown string S over Σ using *subsequence* queries, ie. asking whether q is a subsequence of S. Our main result is that $O(n \lg \alpha + \alpha \lg n)$

subsequence queries suffice to reconstruct an unknown string of length n. The matches the information-theoretic lower bound of Lemma 1 and proves that subsequence queries are more powerful than substring queries for reconstruction.

Our basic strategy is first to determine the character composition of S, and then interleave the resulting subsequences of S into longer subsequences. By repeatedly interleaving the resulting subsequences, we reconstruct S.

For each character x in Σ, we can use subsequence queries to perform a binary search to count how many times x occurs in S, for the problem is equivalent to asking for the largest i such that x^i is a subsequence of S. Even if n is unknown, we can perform one-sided binary searches, so $2\alpha \lg n$ subsequence queries suffice to determine the character composition of S.

Lemma 14 *Let A and B be character-disjoint subsequences of S, where $a = |A|$, $b = |B|$, and $a \leq b$. We can interleave A and B to obtain a length $a + b$ subsequence of S using at most* $2.18a + b + 5$ *subsequence queries.*

Proof: For any subsequence A of S, the characters of A partition S into $a + 1$ (possibly empty) substrings, say $S_0, \ldots, S_a$. If B is a character-disjoint subsequence of S, the problem of interleaving A and B reduces to determining how many characters of B lie in each of $S_0, \ldots, S_a$.

Let $B_{(x,y)}$ denote the substring consisting of the xth through yth characters of B. Determining the number of characters of B in S_0 is equivallent to finding the largest i_0 such that $B_{(1,i_0)}A$ is a subsequence of S, which can be found in $2\lceil \lg i_0 \rceil$ subsequence queries through a one-sided binary search. With i_0 known, the number of characters of B in S_1, i_1 follows from a one-sided binary search with queries of the form $B_{(1,i_0)} a_1 B_{i_0+1, i_0+i_1} a_2 \ldots a_a$.

In general, let $i_j = |S_j \cap B|$. By proceeding in this manner, we can interleave subsequences A and B using

$$T(a, b) = \sum_{j=1}^{a+1} 2\lceil \lg i_j \rceil \leq 2(a + 1) + 2 \sum_{j=1}^{a+1} \lg i_j$$

queries. Since $b = \sum_{j=1}^{a+1} i_j$, we know that

$$2 \sum_{j=1}^{a+1} \lg i_j = 2 \lg(\prod_{j=1}^{a+1} i_j) \leq 2 \lg((b/a)^{a+1}) = 2(a + 1) \lg(b/a)$$

as $\prod_{j=1}^{a+1} i_j$ attains the maximum when $i_j = i_k$ for all $0 \leq j \leq k \leq a$ and $\lg x$ is a monotone increasing function for all $x \geq 1$. Thus $T(a, b) \leq 2(a + 1)(1 + \lg(b/a))$. Let $k = \lg(b/a)$, so $b = a2^k$. Now

$$T(a, b) = 2(a + 1)(1 + k) \leq 2(a + 1) + 0.18a + 3 + b$$

by the following claim, giving the result.

Claim 1 *$2(a+1)k \leq b + 0.18a + 3$.*

Proof: We must show that $2(a+1)k \leq a2^k + 0.18a$, as $b = a2^k$. Consider the function $f(k) = 2(a+1)k - a2^k$. f does not have have a minimum and attains a maximum when $k = \lg_2(2(a+1)/a\ln 2)$. The corresponding maximum value is

$$2(a+1)\lg_2(\frac{2}{\ln 2}\cdot(1+1/a)) - \frac{2a}{\ln 2}\cdot(1+1/a).$$

Let $x = 2(1+1/a)/\ln 2$. The above expression can be rewritten as $a[2\lg_2 x - x] + 2\lg_2 x$. The maximum value of $2\lg_2 x - x$ is $2\lg_2(2\lg_2 e) - 2\lg_2 e \leq 0.18$. The maximum value of $2\lg_2 x$ is 3 as $a \geq 1$, which proves the claim. ▮

Theorem 15 *$2\alpha\lg n + 1.59n\lg\alpha + 5\alpha$ subsequence queries suffice to reconstruct a unknown string S of unknown length n.*

Proof: We have shown how to decompose S into α single character subsequences using at most $2\alpha\lg n$ queries. Repeatedly applying the merge procedure of Lemma 14 reconstructs S using a merge tree of height $\lg\alpha$. For $n = a+b$ and $a \leq b$, $2.18a + b + 5 \leq 1.59n + 5$, so

$$2\alpha\lg n + 1.59n\lg\alpha + 5\sum_{i=1}^{\lg\alpha}\alpha/2^i \leq 2\alpha\lg n + 1.59n\lg\alpha + 5\alpha$$

queries suffice. ▮

4 Verifying Strings with Substrings

Strictly speaking, a sequence S is uniquely determined by a classical sequencing chip $C(m)$ iff the implied subgraph of the de Bruijn graph $D(\alpha, m)$ is exactly a path. However, often we have additional information about the sequence which we can exploit to show that $C(m)$ implies S. In this section, we show how specifying the length of S can be used to increase the resolving power of classical sequencing chips.

A special application of these results is in using sequencing by hybridization to verify the validity of a DNA sequence derived from a different experimental procedure. Let S' be an experimentally determined sequence of S. We need to verify whether $S' = S$ using substring queries. In general, our objective is to minimize the cost in deciding whether $S = S'$, where the cost of a query is somehow a function of its length. Certain strings are expensive to verify under such a model, for example, verifying $a^l b a^l$ requires asking at least one query of length $l+1$. Minimizing the length of the longest query is equivalent to asking what is the smallest classical sequencing chip $C(k)$ such that $C(k)$ verifies S, given that we know $|S| = n$.

This problem can be stated as testing whether an unweighted digraph has a unique *postman walk* of length n. In a digraph $G = (V, E)$, a walk W from x to y is a sequence of vertices $x, v_1, \ldots, v_k, y$ such that $(v_i, v_{i+1}) \in E$. A walk W is called a *postman walk* if all edges in E appear in W at least once. A postman walk W is said to be a *minimum postman walk* if there does not exist a postman walk W' such that $|W'| < |W|$.

Let A be the set of all k-substrings contained in S. As in the de Bruijn graph construction of Section 3.2, for each k-string $x_1, \ldots, x_k$ in A, we create two vertices $x = x_1, \ldots, x_{k-1}$ and $y = x_2, \ldots, x_k$ and directed edge from x to y. It can be easily seen that there exists a unique string of length l, containing all k-strings in A if and only if there exists a unique postman walk of length $l = n - k + 1$ in G.

Finding the minimum postman walk is known as the *Chinese postman problem* [17]. Polynomial algorithms based on bipartite matching exist for directed and undirected graphs, [7], although the problem is NP-complete for mixed graphs [22].

For a digraph G, there may exist positive integers l such that a postman walk of length l either (1) does not exist, (2) exists and is unique, or (3) exists and non-unique. For example, the graph in Figure 1, (1) does not have a postman walk of length $w+x+y+z+6a+b+1$, (2) has a unique postman walk of length $w+x+y+z+6a+b$, and (3) does not have unique postman walk of length $w+x+y+z+9a+b$ Note that uniqueness is not a monotone property with respect to the length of the walk, as a walk of length $w+x+y+z+9a+b$ can be made non-unique whereas a walk of length $w+x+y+z+6a+2b$ is unique.

We note that our algorithm for testing the uniqueness of postman walks depends heavily on the graph being unweighted. Indeed, for weighted graphs the problem of testing the existance of a postman walk of given length is hard.

Theorem 16 *Given a weighted directed graph G and a target integer t, the problem of deciding whether G has a postman walk of length exactly t is NP-hard.*

Proof: We reduce the integer partition problem to fixed length postman walk. In integer partition, we are given a set S of integers and seek a subset $S' \in S$ such that the sum of the weights in S' equals the sum of the weights in $S - S'$.

For each element of S, we create a pair of directed edges between two vertices, one of weight s_i, the other of weight 0. Intermediate vertices of indegree and outdegree 1 can be used to remove multiedges. These gadgets are strung together to form a directed path, with a feedback edge of weight X defining a directed cycle. See Figure 4.

The vertices of indegree and outdegree 1 define the starting and ending points of any postman walk. X is chosen to be substantially larger than any of the edge weights, to ensure that any postman walk through G of weight t makes the same number of trips through X.

We claim that G contains a postman walk of length $t = 3X + \frac{3}{2}\sum_{s \in S} s$ iff S can be partitioned. Because X is so large, any postman walk of length t traverses X exactly three times. Each of the edges labeled with S and zeros must be traversed at least once, and exactly one of each pair can be traversed twice. The set of edges traversed twice define the partition. ∎

In Section 4.1, we provide characterization of the uniqueness of given length postman walks, in a digraph. In Section 4.2 we give an $O(n^3)$ algorithm to test the existence based on the lemmas proved in 4.1.

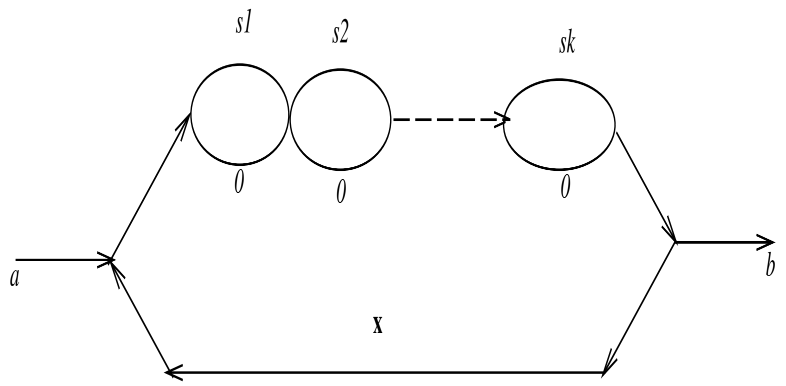

Figure 8: Reducing Partition to Finding a Postman walk in G.

4.1 Existence of Unique Postman Walks

In this section, we characterize unique postman walks of length l, for all l in a given directed graph G. For ease of presentation, assume that the start and end vertices of all postman walks in G are two fixed vertices a and b. We will relax this assumption later. Here W denotes a postman walk and $|W|$ denotes the length of the walk. Also, P denotes a minimum postman walk in G. A subwalk of a postman walk W is a subsequence of W. A self walk $\{x, v_1, \ldots, v_i, x\}$ of W is said to be *redundant* if $W' = W \setminus \{x, v_1, \ldots, v_i, x\}$ remains a postman walk. We let x^i denote the ith appearance of a vertex x in a given walk. The following lemmas characterize postman walks in G.

Lemma 17 *Let G be a directed graph containing a vertex of indegree or outdegree greater than 2. Then for any positive integer k, a postman walk W of length k in G cannot be unique.*

Proof: Let x be a vertex of indegree or outdegree greater than two. Then there exists subwalks $w_1 = \{x^i, \ldots, x^{i+1}\}$ and $w_2 = \{x^{i+1}, \ldots, x^{i+2}\}$ such that $w_1 \neq w_2$. Interchanging w_1 and w_2 yields a different postman walk. ∎

Lemma 18 *In any postman walk W, where $|W| > |P|$, there is a vertex of G which appears at least three times in W.*

Proof: Since $|W| > |P|$, there exists a non-empty set of edges which are traversed in W more often than in P. Further, unless G is a directed cycle, there is at least one such edge (u, v) where either u has indegree 2 or v has outdegree 2, so either u or v must be traversed at least three times.

If G is a directed cycle, and if the start vertex a and the end vertex b are the same, then a appears at least three times in W. If $a \neq b$, again a has to appear three times, as in the minimum postman walk a has to appear two times. ∎

Lemma 19 *If a postman walk W of length k is unique, then (1) the minimum postman walk P is unique and (2) W can be constructed by adding redundant self walks to P.*

Proof: Suppose W is a unique postman walk of length k. First, we prove that P is unique. If $|P| = k$, then clearly the minimum walk must be unique. Otherwise, $|W| > |P|$, and by Lemma 18 there exists a vertex x which appears at least three times in W. Consider the subwalks $w_1 = \{x^1, \ldots, x^2\}$ and $w_2 = \{x^2, \ldots, x^3\}$. If $w_1 \neq w_2$ then W cannot be unique, violating our assumption. Thus $w_1 = w_2$, so w_1 is a redundant self-walk.

Let $W' = W - w_1$. Observe that the uniqueness of W implies that W' is a unique postman walk of length $|W'|$. Since $|W'| < |W|$, we may repeat this procedure until a unique postman walk of length $|P|$ remains. By reversing this procedure, W can be constructed by adding redundant self walks to P. ∎

Lemma 19 gives a necessary condition for a unique postman walk W, with $|W| \geq |P|$, i.e., W is constructed by adding redundant self walks to P. For a walk W, we define the operation *add self walk*, *ASW(w,i)*, as inserting the self walk $w = (x, z_1, \ldots, x)$ at x^i.

Also, two walks W and W' are equivalent if W' can be obtained from W by moving the self walk $x^i, \ldots, x^{i+1}$ to $x^j, \ldots, x^{j+1}$ in W, for some $j \neq i$. We say W' is obtained from W by applying the *MOVE(x,i,j)* operation to W.

Lemma 20 *Consider a walk W' created from W by an arbitrary sequence of ADD and and MOVE operations. There exists a sequence of ADD operations followed by a sequence of MOVE operations which can be used to construct W.*

Proof: Let W' be obtained from W by an arbitrary sequence S of ADD and MOVE operations. Consider the first ADD operation in S which follows a sequence of MOVE operations. There is a clearly defined vertex after which we are inserting the new walk. We expand any subsequent move operation involving this vertex to include the new walk. Applying this argument repeatedly, we can promote all ADD operations before any MOVE operation, and still obtain W'. ∎

Lemma 21 *In G, if the minimum postman walk P is unique, then any postman walk W can be obtained from applying ASW and MOVE operations to P.*

Proof: We prove this lemma by contradiction. Suppose there exists a postman walk W, which cannot be obtained from P by adding only self walks to P and applying MOVE operations. Remove a set of redundant self walks from W until none remain, and denote the resulting sequence by W'. Let $x(\neq b)$ denote the first vertex at which P and W' differ. Clearly, x must have outdegree 2. Let $e_1 = (x, v_1)$ and $e_2 = (x, v_2)$ be the two directed

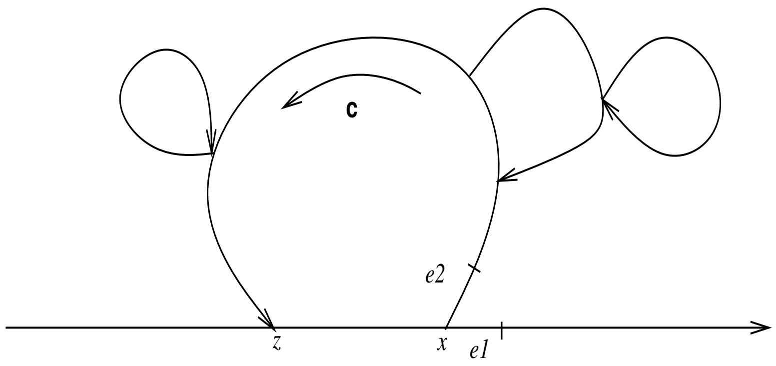

Figure 9: Structure of P in G.

edges emanating from x. In P, x must have been visited exactly twice, and without loss of generality let us assume that in P, e_1 is visited first. See Figure 4.1.

Now consider the subwalk $w = x^1, \ldots, x^2$ in P. Let Y denote the multiset of vertices in the subwalk w excluding x, and $Y' \subset Y$, possibly empty, consisting of all vertices in Y which appeared before x^1 in P. If $Y' \neq \emptyset$, let $z \in Y'$ denote the vertex which appeared first in P. The Lemmas 22, 23 and 24 characterize both P and G.

Lemma 22 *A vertex $y \in Y$ cannot appear after x^2 in P.*

Proof: Suppose not. Then we have two distinct subwalks $x^1 \ldots y^1$ and $x^2 \ldots y^2$ which can be interchanged, thus contradicting the uniqueness of P. ∎

Lemma 23 *If $Y' \neq \emptyset$, then the induced graph formed by $Y' \cup \{x\}$ is homeomorphic to the directed edge (z, x). If $Y' = \emptyset$, then x is a cut node of G.*

Proof: If $Y' = \emptyset$, then from the definition of Y', Lemma 22 and the fact that P is a postman walk it follows that x is a cut node.

When $Y' \neq \emptyset$, first observe that y must have outdegree 1, for all $y \in Y'$. Otherwise we have two different subwalks $y, y_1, \ldots, x$ and $y, y_2, \ldots, x$, contradicting the uniqueness of P. Also, for $y \in Y' - z$, y must have indegree 1. Otherwise there must exist a vertex $y' \in Y'$ such that y' has outdegree greater than 1, since in P each vertex is seen exactly twice. But we have just seen that all vertices in Y' has outdegree 1. Hence, all vertices in $Y' - z$ has to have both indegree and outdegree 1, and the lemmas follows from the fact that they are connected. ∎

Lemma 24 *Removal of x and z from G results in three or more connected components viz., $Y' \setminus \{z\}$, $Y \setminus Y'$, and components consists of $(G \setminus Y) \setminus \{x, z\}$.*

Proof: Follows from Lemmas 22 and 23. ∎

It follows that after leaving x through edge e_1, the return path to x is forced. Applying the arguments recursively to $(Y \setminus Y') \setminus \{x\}$ there must be a cycle c contained in G, such that the self-walk w from vertex x through the edge e_1 visits all edges in c.

Now consider W'. By using the MOVE operation, we can move the subwalks $x, v_1, \ldots, x$ before the subwalks $x, v_2, \ldots$ in W'. Note that the last vertex b of W' is not in w. We consider two cases, for either W' visits the edge e_1 exactly once or e_1 is visited more than once. In the first case, the subwalk $x^i, v_1, \ldots, x^{i+1}$ is the same as that of w and therefore W differs from P at a later vertex than x^2, in P, contradicting the assumption.

In the second case, we can use repeated MOVE operations, to make the edges of c form a self-walk. Since we traverse the edges of c more than once, this self-walk is redundant. Deleting c leaves a smaller walk W'' which must also have the property that it cannot be obtained from P by using the operations ASW and MOVE. Recursively applying this argument to W'' we obtain a walk W^∞ such that W^∞ is not obtained from P via ASW/MOVE and either (1) $|W^\infty| = |P|$ or (2) W^∞ differs from P the last vertex b in P. In the first case, $W^\infty = P$ since P is unique. In the second case, the difference must be a redundant self-walks since P is a postman walk. In either case, we obtain a contradiction. ∎

Therefore, verifying the uniqueness of a walk W is equivalent to checking whether there is only one way of adding redundant self walks of appropriate length to P, and further that MOVE operations cannot be applied to this walk to obtain different walks. In the next section, we present an algorithm to test these conditions.

4.2 An Algorithm to Test Uniqueness

We now give an algorithm to test whether there exists a unique postman walk W of length l from a to b in G. G must be a sparse graph, with $|V|$ vertices and at most $2|V|$ edges for W to be unique. Note that the block graph of the strongly connected components of G must be isomorphic to a path, or G contains no postman walk. In the following discussion, let W denote a postman walk of length l and $l' = l - |P|$.

We can use the algorithm of [7] to find a minimum postman walk in G, by finding a minimum weight matching in G', where G' contains the unbalanced (ie. indegree $\neq$ outdegree) vertices with directed edges of weight corresponding to shortest path between them. The minimum postman walk is unique if and only if there does not exist more than one minimum weight matching in G'. Finding the K best matchings can be done in $O(Kn^3)$ time [4]. Hence, in $O(|V|^3)$ time we can find a unique minimum postman walk if there exists one. If the minimum postman walk is not unique, then by Lemma 19 there does not exist a unique postman walk W in G for any length l.

Next, we must check whether there exists a unique way of adding self-walks of length l' to P. We claim that the following algorithm suffices:

1. Compute the set U, where $i \in U$ iff there exists a self-walk of length i in G, for all $1 \leq i \leq l'$.

2. Compute the set U', where $i \in U'$ iff there exists a *unique* self-walk of length i in G, for all $1 \leq i \leq l'$.

3. For each $u_i \in U \setminus U'$, solve the knapsack problem with elements U and target $l' - u_i$. If there exists a solution, the postman walk is not unique.

4. A self-walk of length i in G is *prime* iff there does not exist a self-walk of length i' such that i' divides i. From U', construct the maximal set of prime self-walk lengths U''. Test whether there is a unique way to express l' as an integer linear combination of elements of U''. If so, there is a unique way to add self-walks of length l' to P.

To prove the correctness of this procedure, suppose that two different sets of self-walks W_1 and W_2 of total length l' could be added to P. Decompose W_1 and W_2 into simple self-walks, which define multisets of prime elements of U. Either these multisets are identical or they are different. If they are identical, then there exists some prime element which is the length of two distinct self-walks, which would have been discovered in step 3. If they are not identical, then there is not a unique linear combinatation of U'' adding up to l', which would have been discovered in step 4.

Now, we consider the time complexity of this procedure. To compute the set U, for each vertex x, and for each i, $1 \leq i \leq l'$, we will compute $N[x,i]$, the number of self walks of length i starting and ending at x. Let C_j be a $|V| \times |V|$ matrix denoting the number of length j walks between each pair of vertices. Since any particular vertex v in G has indegree at most two, the number of $j+1$ length walks from u to v equals the sum of length-j walks from u to each of these two ancestors of v. Thus each element of C_{j+1} can be computed in constant time and $O(|V|^2 \cdot l')$ time suffices to construct the l' matrices. $N[x,i]$ is given by $C_i[x,x]$.

From these matricies, we will compute U and U'. From the values of $N[x,i]$ for all vertices x and $1 \leq i \leq l'$, we know the set of self-walk lengths U which can be achieved. In constructing U', observe that it is not sufficient to test whether there exists an x such that $N[x,i] = 1$, for there might be two disjoint cycles of length i. Therefore, starting from $i = 1$ and proceeding until $i = l'$, we count the number of vertices such that $N[x,i] = 1$. If the count is at most i, and i cannot be expressed as the multiple of two distinct smaller elements of U', then the self-walk of length i is unique and is added to U'. With this procedure, U' can be constructed in $O(l'^2)$ time.

Although the knapsack problem in step 3 is NP-complete, all our weights are bounded by l', which at most the length n of the walk. Hence, we can use the standard dynamic programming algorithm to solve this knapsack in $O(l'^2)$ [10]. With at most l' instances of knapsack, step 3 can be performed in $O(l'^3)$ time.

Testing whether there is a unique way to express l' as a integer linear combination of elements of U'' is the bottleneck of step 4. Note that there does not exist an integer $x \leq l'$ such that $x = c_1 \cdot u_1 = c_2 \cdot u_2$ where $u_1 \neq u_2$ and $u_1, u_2 \in U''$, for such an x would have been eliminated in step 3. For each prime element, we can group all its multiples less than l' into an equivalence class. Therefore, the total number of elements in all the $|U''|$ equivalence classes is at most l'. If there is a unique weighted knapsack solution, it contains at most one element in each equivalence class. For each element in the first equivalence class, i.e., integers of the form $c \cdot u_1''$, we can solve a knapsack problem on the elements of all other equivalence classes with a target size of $l' - c \cdot u_1''$. If the solution is unique, at most one such problem will have a solution. By repeating this argument for each equivalence class, we determine whether the solution is unique by solving at most l' knapsack problems, each in $O(l'^2)$ time.

Now, if we have established that there is a unique way to add the required self walks, we must decide whether these walks can be placed uniquely in P. For each self-walk $w = x, \ldots, x$, insert it into P by replacing an instance of x with w, creating W. For any vertex x, the subwalk between x^i and x^{i+1} in W defines a self-walk for x. If any two self-walks for x are not identical, they can be interchanged giving a different walk. All self-walks for each vertex are identical, then W is a unique walk of length l. For each of the vertices, the self-walks can be analyzed in linear time, for a total of $O(l^2)$.

In conclusion, we have shown

Lemma 25 *In an unweighted directed graph G we can determine whether there exists a unique postman walk of length l from fixed vertices a to b in $O(l^3)$ time.*

The algorithm of Lemma 25 can be invoked for each pair of vertices to test whether there is a unique postman walk of length l in G. However, this is unnecessary. Consider the block graph of G, where each block corresponds to a strongly connected component. For a postman walk to exist, the block graph of G must be isomorphic to a directed path. Let $C_1, \ldots, C_r$ denote the strongly connected components of G.

There are several possible cases. Suppose there is only one vertex in the block graph of G, ie. $r = 1$.

Lemma 26 *If G is strongly connected, then there cannot exist a unique postman walk of length greater than $|P|$.*

Proof: Consider a unique postman walk $W = x, x_1, \ldots, y$ of length greater than $|P|$ in G. Since W is unique, all postman walks of length less than $|W|$ must start at x and end at y, for otherwise we can augment this shorter postman walk with a path into x, since G is strongly connected. Since $|W| > |P|$, by Lemma 18 there exists a vertex z which occurs at least 3 times in W. This defines two self-walks for z. If they are not identical, W is not unique, if they are identical, then deleting one from W leaves a smaller postman which must be unique. However, this can be extended to a walk of length $|W|$ since G is strongly connected. In either case, we obtain a contradiction. ∎

Since the block graph of G must contain more than one component, $r > 1$, and for W to be unique, the first component C_1 and last component C_r cannot both contain more than one vertex. Otherwise W can be extended by one edge in C_1 and shortened by one edge in C_r and still remain a postman walk, by the argument above. If C_1 and C_r consist of exactly one vertex each, we are left with the a-b path problem which was solved in $O(l^3)$ time in Lemma 25. The only remaining case is when exactly one of C_1 and C_r contains more than one vertex.

Suppose C_1 contains more than one vertex – the case in which C_r has more than one vertex is similar. If W is unique, then the minimum postman walk $P = x^1, \ldots, x^2, \ldots, y$ must be unique. There cannot exist a self-walk w of length less than or equal to l' in $G \setminus C_1$, for deleting w and replacing it with an equal-length walk in C_1 yields another walk of length $|W|$. Therefore, all self-walks of length $\leq l'$ in W must occur in C_1, and further must pass through x. Suppose the self-loop went through a vertex in C_1 other than x. Then the redundant instance can be deleted and replaced with an equal length walk to x to yield another walk of length $|W|$.

Therefore, the unique walk of length $|W|$ starts with a uniquely defined walk w' through x containing all the redundant self-walks which will appear in W. By the previous analysis, C_1 must consist of a directed cycle c containing x and self-walks only of length $> l'$. w' consists of l' steps backwards around c, and hence can be easily constructed within $O(l^3)$ time.

In conclusion, we have shown that

Theorem 27 *In a directed graph G, given a positive integer l, in $O(l^3)$ time we can determine whether there exists a unique postman walk of length l.*

5 Conclusions

We have developed a theory of interactive sequencing by hybridization, providing tight bounds on the complexity of reconstructing strings from substring queries. Several interesting problems remain open:

- Further tighten the bounds for interactive determination of unrestricted strings. We conjecture that $\approx (\alpha - 1)n$ queries are necessary and sufficient, meaning that our upper bound is essentially tight. It would be interesting to know the exact value of the lower order terms for reconstructing binary strings, even for small n.

- The approximation algorithm for minimum height decision tree presented in Theorem 13 uses a prescribed query at each internal node, which is not necessarily the most even possible split. Therefore, the greedy tree can do no worse and possibly better – is it provably better than the specified tree?

- Generalize these problems for the case of positive and/or negative errors. Under such a model, the result of a given substring query may be reported incorrectly, as is the case with real-life sequencing by hybridization. The related problem of searching a sorted list with "lies" has been extensively studied [28].

- Sequencing chips perform all queries in parallel. How many rounds does it take to determine an unknown string when we can make $f(n, \alpha)$ queries per round? Parallelizing the strategy of Theorem 2 in an obvious way gives $n + O(1)$ rounds of α queries per round, but can we do better?

References

[1] E. Arkin, J. Mitchell, H. Meijer, D. Rappaport, and S. Skiena. Decision trees for geometric objects. In *Ninth ACM Symp. Computational Geometry*, 1993.

[2] W. Bains and G. Smith. A novel method for nucleic acid sequence determination. *J. Theor. Biol.*, 135:303–307, 1988.

[3] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. In *Proc. 23th ACM Symposium on Theory of Computing*, pages 328–336, 1991.

[4] C.K. Chegireddy and H.W. Hamacher. Algorithms for finding k-best perfect matchings. *Discrete Applied Mathematics*, 18:155–165, 1987.

[5] N.G. de Bruijn. A combinatorial problem. *Proc. Kon. Ned. Akad. Wetensch*, 49:758–764, 1946.

[6] R. Dramanac and R. Crkvenjakov. DNA sequencing by hybridization. Yugoslav Patent Application 570, 1987.

[7] J. Edmonds and E. Johnson. Matching, Euler tours, and the Chinese postman problem. *Math. Prog.*, 5:88–124, 1973.

[8] S. Fodor, J. Read, M. Pirrung, L. Stryer, A. Lu, and D. Solas. Light-directed, spatially addressable parallel chemical synthesis. *Science*, 251:767–773, 1991.

[9] J. Gallant, D. Maier, and J. Storer. On finding minimal length superstrings. *J. Computer System Sci.*, 20:50–58, 1980.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco CA, 1979.

[11] I. J. Good. Normal recurring decimals. *J. London Math. Soc.*, 21:167–172, 1946.

[12] L. Guibas and A. Odlyzko. String overlaps, pattern matching, and non-transitive games. *J. Combinatorial Theory (Series A)*, 23:183–208, 1981.

[13] L. Hyafil and R. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5:15–17, 1976.

[14] T. Jiang and M. Li. Approximating shortest superstrings with constraints. In *Workshop on Algorithms and Data Structures*, 1993.

[15] J. Kieleczawa, J. Dunn, and F. Studier. DNA sequencing by primer walking with strings of congiuous hexamers. *Science*, 258:1787–1991, 1992.

[16] K-I. Ko and S-C. Teng. On the number of queries necessary to identify a permutation. *Journal of Algorithms*, 7:449–462, 1986.

[17] M.-K. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1:273–277, 1962.

[18] M. Li. Towards a DNA sequencing theory. In *Proc. 31st IEEE Foundations of Computer Science Conf.*, pages 125–134, 1990.

[19] Y. Lysov, V. Florent'ev, A. Khorlin, K. Khrapko, V. Shik, and A. Mirzabekov. *Dokl. Acad. Sci. USSR*, 303:1508–, 1988.

[20] S. Macevicz. International Patent Application PS US89 04741, 1989.

[21] B. Moret. Decision trees and diagrams. *Computing Surveys*, pages 593–623, 1982.

[22] C. Papadimitriou. The complexity of edge traversing. *J. ACM*, 23:544–554, 1976.

[23] P. Pevzner, Y. Lysov, K. Khrapko, A. Belyavski, V. Florentiev, and A. Mizabelkov. Improved chips for sequencing by hybridization. *J. Biomolecular Structure and Dynamics*, 9:399–410, 1991.

[24] P. A. Pevzner and R. J. Lipshutz. Towards DNA sequencing by hybridization. Submitted for publication, 1992.

[25] P.A. Pevzner. l-tuple DNA sequencing: Computer analysis. *J. Biomolecular Structure and Dynamics*, 7:63–73, 1989.

[26] C. Flye Saint-Marie. Solution to question nr. 48. *l'Intermediaire des Mathematiciens*, pages 107–110, 1894.

[27] E. Southern. United Kingdom Patent Application GB8810400, 1988.

[28] J. Spencer. Ulam's searching game with a fixed number of lies. *Theoretical Computer Science*, 95:307–321, 1992.

[29] Z. Strezoska, T. Paunesku, D. Radosavljevic, I. Labat, R. Drmanac, and R. Crkvenjakov. DNA sequencing by hybridization: 100 bases read by a non-gel-based method. *Proc. Nat. Acad. Science USA*, 88:10089–10093, 1991.

[30] H. Wilf. Strings, substrings, and the nearest integer function. *Amer. Math. Monthly*, 94:855–860, 1987.