Dynamic Multiple Pattern Matching

by

Ramana Murthy Idury

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Department of Computer Science
Rice University
Houston, Texas

May, 1994

Dynamic Multiple Pattern Matching

Ramana Murthy Idury

Abstract

Pattern matching algorithms are among the most important and practical contributions of theoretical computer science. Pattern matching is used in a wide variety of applications such as text editing, information retrieval, DNA sequencing, and computer vision. In this thesis, we study the problem of pattern matching with multiple patterns. The set of patterns is called a *dictionary*. Furthermore, the dictionary is made dynamic by allowing insertion and deletion of individual patterns. We preprocess the dictionary so as to provide efficient searching as well as efficient updates. We first present a DFA-based solution to the one dimensional version of the problem where the patterns are strings. We then use this solution to solve the two dimensional version of the problem where the patterns are restricted to have square shapes. Finally, we solve the general case, where the patterns can have any rectangular shape, by reducing this problem to a range searching problem in computational geometry.

Acknowledgments

I am grateful to my advisor Alejandro Schäffer for the valuable support he provided throughout my research. I appreciate his generosity and good nature, which always warmed my heart.

My thanks to Mark Krentel and Peter Varman for serving on my thesis committee and providing constructive suggestions in writing this dissertation.

I specially thank Amihood Amir of Georgia Tech, and Martin Farach of DIMACS for inspiring me to work in the area of pattern matching.

I acknowledge the support that I received from the faculty members of the department. I am also thankful to the administrative and support staff in the department, who took care of several problems for me. I also thank my friends and colleagues who made my stay at Rice a memorable one.

I wish to acknowledge the financial support from the W. M. Keck Foundation. I have also enjoyed my meetings with several people from the Keck Center for Computational Biology, and Baylor College of Medicine.

I should also mention my parents, my brothers, and their families for their love and affection.

Finally, I am deeply thankful to my wonderful wife, Radha, for her infinite love, support, and encouragement that she provided throughout my graduate studies. I appreciate the patience and endurance she showed during this period.

To my wife Radha

Contents

Illustrations

Chapter 1

Introduction

Pattern matching algorithms are among the most important and practical applications of theoretical computer science. Whenever one edits a text with `emacs`, searches a file for keywords with `grep`, or compares two files with `diff`, one is using some kind of string matching algorithm. While these applications continue to generate further interest in classical string matching problems [38], new problems are emerging in two dimensional pattern matching with applications to computer vision [4]. With the advent of the Human Genome Project and its efforts to recognize all the human genes, the importance of pattern matching algorithms continues to grow [60].

In a typical pattern matching problem, we are given a *pattern* and a *text*. The aim is to search for all occurrences of the pattern in the text. The standard way to do this is to preprocess either the pattern or the text so that the searching can be done faster. In many pattern matching problems, either the pattern or the text is fixed and the other is varied. Therefore it is efficient to preprocess the fixed part only once, and use it for subsequent searches. The cost of preprocessing becomes critical, both in terms of time and space, as the size of the fixed part gets very large. In this regard, efficient preprocessing is equally important as efficient searching. In this thesis, we work on finding efficient solutions for pattern matching problems where the cost of preprocessing is substantial.

1.1 Pattern Matching with Strings

Many pattern matching problems can be classified into one of the following two paradigms depending on what is fixed (and therefore preprocessed):

- **Fixed Pattern Matching (FPM):** Given a *fixed* pattern P over an alphabet Σ, preprocess it so as to be able to find all its occurrences in a *query* text T.

A classical linear time solution for this problem was given by Knuth, Morris, and Pratt [52], or KMP. Their solution is based on constructing a deterministic finite automaton for the pattern and using it to search any given text. The automaton consists of *forward* transitions which are taken when a text symbol matches a pattern symbol, and *backward* transitions which are taken when the text symbol does not match the pattern symbol. They also define a failure function $fail$ that captures all the backward transitions. There are other solutions for the FPM problem. Boyer and Moore [25] gave a solution with better practical performance [17], Karp and Rabin [51] provided a randomized solution, while Galil and Seiferas [41] used the periodicity of strings to obtain tighter space bounds for the preprocessing. A more practical solution from a space point of view is due to Crochemore and Perrin [31]. The FPM paradigm was also extended to tree patterns and texts [53, 34].

- **Fixed Text Matching (FTM):** Given a *fixed* text T over an alphabet Σ, preprocess it so as to be able to find all occurrences of a *query* pattern P in the text.

Traditionally this problem was solved in linear time and space using *suffix trees* [63, 57, 28]. A suffix tree can be succinctly defined as an ordered search tree on the suffixes of a text in which all chains in the tree are compacted into one edge. There are other similar representations for the suffix trees, *subword automata* [24] and *suffix arrays* [56], that are equally efficient. The suffix tree is a very versatile data structure and can be used to solve many other pattern matching problems such as finding the longest common substring of two or more strings. Interestingly suffix trees can be used to solve some of the FPM problems.

The Fixed Pattern Matching paradigm can be naturally extended to multiple patterns as follows:

- **Multiple Pattern Matching (MPM):** Given a *fixed* set of patterns $D = P_1, \ldots, P_n$ over an alphabet Σ, preprocess it so as to be able to search for all occurrences of all the patterns in a *query* text T. The set of patterns is also called a *dictionary*.

Aho and Corasick, or AC, solved this problem with linear preprocessing and searching times [1]. The salient feature of their algorithm is the extension of the KMP algorithm to multiple patterns. They define two important functions *goto* and $fail$ that capture the forward and backward transitions of the automaton. This is perhaps the first string matching problem where both preprocessing and searching costs can be substantial. The AC algorithm has applications in bibliographic search [1], two dimensional pattern matching [23, 21], and protein sequencing [3]. Another solution for the MPM problem is due to Commentz-Walter [29], which can be viewed as a generalization of the Boyer-Moore approach. An important property shared by all the solutions to the MPM problem is their dependence on the alphabet size. The time bounds have a multiplicative $\log \sigma$ factor, where $\sigma \leq |\Sigma|$ is the number of distinct characters of Σ in D.

Now that we have a set of patterns, it is natural to ask what happens if the set gets changed.

- **Dynamic Dictionary Matching (DDM):** Given a *dictionary* of patterns $D = P_1, \ldots, P_n$ over an alphabet Σ, preprocess it so as to be able to search for all occurrences of all the patterns in a query text T. In addition, make the dictionary *dynamic* by supporting *insertion* and *deletion* of individual patterns from the dictionary.

One motivation for the DDM problem is that the dictionary, which can be viewed as a query set of known patterns such as genes, can be very large compared to the query texts. Moreover, as our understanding of individual patterns improves, we need to update the database by inserting or deleting patterns. In such a case, it is very expensive to preprocess the entire dictionary from scratch (the AC algorithm cannot handle insertions or deletions). Therefore it is important to obtain solutions for efficient updates even at the expense of a slower searching time.

Amir and Farach [9] solved this problem by building a suffix tree for the dictionary of patterns, and using this as a search automaton. They obtained the following time bounds:

Dictionary Preprocessing: $O(d\ (\log \sigma + \log d))$, where d is the sum of the lengths of patterns in D.

Searching a text T: $O(t\ (\log \sigma + \log d))$, where t is the length of the text.

Inserting a pattern P: $O(p\ (\log \sigma + \log d))$, where p is the length of the pattern.

Deleting a pattern P: Amortized time $O(p\ (\log \sigma + \log d))$.

Later, Amir, Farach, Galil, Giancarlo, and Park, or AFGGP, improved the solution to handle deletion in worst case $O(p(\log \sigma + \log d))$ time [12]. Amir, Farach, and Matias [14] provided a randomized solution for the problem.

One may wonder whether the failure function approach of KMP and AC is of any use at all in solving the DDM problem. In this thesis, we show that it is possible to:

1. solve the problem by taking the failure function approach, and obtain the same time bounds as the AFGGP algorithm.

2. achieve linear search time at the expense of slower update times. For any given constant $k \geq 2$, we can search a text in $O(t\ (k + \log \sigma))$ time and insert or delete a pattern in $O(p\ (d^{1/k} + \log \sigma))$ time.

3. preprocess the initial dictionary of patterns in linear time irrespective of the search strategy.

To summarize, we show on one hand that it is possible to achieve the time bounds of AFGGP, and provide alternate search strategies in addition. On the other hand, we show that it is possible to achieve the same preprocessing and searching times of AC, and provide sublinear update times in addition.

In addition to the results above, we also show that the problem of maintaining the failure function dynamically can be reduced to a problem of maintaining a dynamic list of balanced parenthesis under the operations: *insert* a pair of parentheses, *delete* a pair of parentheses, and *find* the nearest enclosing parentheses. The reduction to the parenthesis problem is exploited in joint work with Amir, Farach, La Poutré, and Schäffer [13] to show that the AFGGP time bounds are not theoretically optimal. In particular, we show that the time bounds of DDM can be improved by a factor of $\log \log d$ when the alphabet size is fixed. This result has applications in bottom-up tree pattern matching [26, 27].

1.2 Pattern Matching in Two Dimensions

The FPM paradigm and the MPM paradigm have natural analogs in two dimensions. Here both patterns and texts are rectangular matrices of characters over an alphabet Σ.

- **Two Dimensional Fixed Pattern Matching (2FPM):** Given a *fixed* rectangular pattern P preprocess it so as to be able to search for all its occurrences in a *query* text.

This problem was originally solved by Bird [23] and independently by Baker [21]. An important feature shared by both the algorithms is the concept of *linearization.* Both algorithms treat the pattern as a one dimensional string where each character corresponds to a column in the original pattern. They use the AC algorithm as a subroutine to transform the columns of pattern and text into these special characters. Because of their reliance on the AC algorithm, the running time of both these algorithms is dependent on the alphabet size. Amir, Benson, and Farach [5], and Galil and Park [40] recently provided alphabet-independent solutions for the problem. Amir and Farach [10] also solved a variation of the 2FPM problem for nonrectangular patterns.

At this stage one may think of extending the 2FPM problem to multiple rectangular patterns as we did for the FPM problem. There are, however, problems associated with this extension. In order for the failure function approach to work, we need a uniform way to align the patterns. Unlike strings, rectangular patterns can have different aspect ratios and therefore do not align in a uniform way.

This problem has remained open since it was first posed by Baker in 1978 [21]. As a big step towards solving this general problem, we first solve an important subset of the problem, where the patterns are restricted to be squares. Unlike rectangles with arbitrary shapes, square patterns have a natural alignment along their major diagonal, and hence, it is possible to extend 2FPM to multiple square patterns.

- **Square Dynamic Dictionary Matching (SDDM):** Given a dictionary of *square* patterns, support efficient searching, insertion, and deletion operations on the dictionary.

Amir and Farach [11] solved the static version of the problem, in which insertion and deletion are not supported, with the following time bounds:

Dictionary Preprocessing: $O(d \log n)$, where n is the number of patterns, and d is the sum of areas of all patterns.

Text Searching: $O(t \log n)$.

We solve the SDDM problem by generalizing the approach of Amir and Farach to support insertions and deletions. While the linearization along the major diagonals is common to both the algorithms, our dynamic algorithm handles the maintenance of failure functions in a novel way. In addition, our dynamic algorithm uses our DDM solution as a subroutine (the static version correspondingly uses the AC algorithm). We obtain the following time bounds for the SDDM problem.

Dictionary Preprocessing: $O(d \log d)$.

Inserting or Deleting a Pattern: $O(p \log d)$.

Text Searching: $O(t \log d)$.

Finally, we solve the general problem defined below:

- **Two Dimensional Multiple Pattern Matching (2MPM):** Given a *fixed* set of rectangular patterns $D = P_1, \ldots, P_n$ over an alphabet Σ, preprocess it so as to be able to search for all occurrences of all the patterns in a rectangular *query* text T.

As before, the set of patterns is also called a *dictionary*. The size (i.e. sum of areas of all patterns) of the dictionary is denoted by d. We define the *linear size* of a rectangular pattern to be the *smaller* of its height and width. Let B the biggest linear size of any pattern. We obtain the following time bounds for the above problem:

Dictionary Preprocessing: $O(d \log(n + \sigma))$.

Text Searching: $O(t \log^2 d \log(B + n + \sigma) + tocc)$, where *tocc* is the total number of pattern occurrences reported.

The most interesting feature of our algorithm is a connection between two dimensional pattern matching and some multidimensional range searching problems. Our 2MPM algorithm also includes new adaptations of techniques from algorithms for *other* two dimensional string matching problems.

We give a summary of publication of our results. Our result on one dimensional dynamic dictionary matching appeared in the *Proceedings of the Third Symposium on Combinatorial Pattern Matching* [48]. Our solution to the dynamic dictionary matching problem with square patterns was published in the *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* [13]. Finally, our result on multiple pattern matching with rectangular patterns appeared in the *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing* [50].

1.3 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we provide background material from previous relevant papers, which is necessary to understand the rest of the thesis. We describe the algorithmic details of our automata-based solution to the one dimensional dynamic dictionary matching problem in Section 3. We have implemented a variation of this algorithm to demonstrate its correctness, simplicity, and practicality. Chapter 4 covers the relevant implementation details. We present our solution to the two dimensional dynamic dictionary matching problem with square patterns in Chapter 5. We then present our solution to the two dimensional multiple pattern matching problem with rectangular patterns in Chapter 6. Finally, in Chapter 7, we summarize our work and discuss possible extensions and other open problems.

Chapter 2

Preliminaries

In this chapter, we provide the background material necessary to understand the rest of the thesis. In Section 2.1, we give the key definitions and notations used in the later chapters. We then present the automata-based Knuth-Morris-Pratt algorithm [52] for the fixed pattern matching problem in Section 2.2. In Section 2.3, we briefly describe the Aho-Corasick algorithm [1] for the multiple pattern matching, which can be viewed as an extension of the Knuth-Morris-Pratt algorithm to multiple patterns. We present the data structure of Dietz and Sleator [32], which plays a very important role in all our dynamic dictionary algorithms, in Section 2.4. Finally, we present an algorithm for the dynamic parenthesis maintenance problem by Güting and Wood [46] in Section 2.5.

2.1 Definitions and Notations

An *alphabet* is a set of characters or symbols. For example, the set of lower-case letters $\{a, b, \ldots, z\}$ denotes an alphabet, with individual letters as the characters. We often use Σ to denote an alphabet, and the letters a, b to denote characters.

A *string*, $x = a_1 \ldots a_n$, is a sequence of characters over an alphabet Σ. That is, for each $1 \leq i \leq n$, $a_i \in \Sigma$, and $x \in \Sigma^*$. The letters x, y often denote a string. The length of a string x is defined as the number of characters in it, and is denoted by $|x|$. For example $abracadabra$ is a string of length 11.

A *null* string is a string of length 0. That is, it does not contain any characters. We often use ϵ or λ to denote a null string.

The *concatenation* of two strings x and y, written as xy, is another string which is a sequence of the characters of x, followed by the characters of y. For example, if $x = left$ and $y = most$, then $xy = leftmost$.

Suppose $x = a_1 \ldots a_n$ is a string. Then a substring of x is defined as another string written as $a_i \ldots a_j$, such that $1 \leq i \leq j \leq n$. That is, a substring is a contiguous part of a string. We write a substring as $x[i, j]$. For example, *part* is a substring of *compartment*. There can be many substrings of a string. Every string is a substring of itself. Also, the null string is a substring of every string.

There are two special cases of substrings which are very important in pattern matching problems. A *prefix* is a substring of a string, that begins with the first character of the string. That is, a prefix of a string x can be written as $x[1, j]$. A *suffix* is a substring of a string, that ends with the last character of the string. That is, a suffix of a string x of length n can be written as $x[i, n]$. For example, $left$ is a prefix of $leftmost$, and $most$ is a suffix of $leftmost$. The null string is both a prefix and a suffix of every string.

Two strings $x = a_1 \ldots a_n$ and $y = b_1 \ldots b_m$ are *equal*, denoted $x = y$, if they match character by character, that is, if $n = m$ and $a_i = b_i$ for all $1 \leq i \leq n$. Otherwise they are not equal.

We define a total order on the strings over an alphabet, called a *lexicographic ordering*, which is sometimes called a *dictionary ordering* in the literature. Intuitively, this is the same as the ordering of the words in a dictionary. For strings $x = a_1 \ldots a_n$ and $y = b_1 \ldots b_m$ such that $x \neq y$, we say that x comes before y in the lexicographic ordering, denoted $x < y$, if there exists an $i \leq \min(n, m)$ such that $a_i < b_i$ and $a_j = b_j$ for all $1 \leq j < i$, or $n < m$ and $a_i = b_i$ for all $1 \leq i \leq n$ (that is, x is a prefix of y). For example, $nest < next$ since the first two characters of both the strings are equal but the third character (s) in $nest$ is smaller than the third character (x) in $next$.

A string $x = a_1 \ldots a_n$ *matches* another (longer) string $y = b_1 \ldots b_m$, if x is a substring of y, that is, if $a_1 \ldots a_n = b_{k-n+1} \ldots b_k$ for some $n \leq k \leq m$. We say that x matches y at location k. There can be several locations k where x matches y. For example, aba matches $abababba$ at locations 3 and 5.

2.2 Fixed Pattern Matching Problem

We have defined the fixed pattern matching (FPM) problem earlier in Chapter 1. There are many efficient solutions for the fixed pattern matching problem as noted in Chapter 1. For the purpose of this thesis, we are interested in the automata-based solution of Knuth, Morris, and Pratt [52], which we call the KMP algorithm. The KMP algorithm preprocesses the pattern in $O(p)$ time, and searches the text in $O(t)$ time.

In the preprocessing step, the KMP algorithm builds a search automaton on the pattern, and uses this automaton to search the text in the searching step. For a

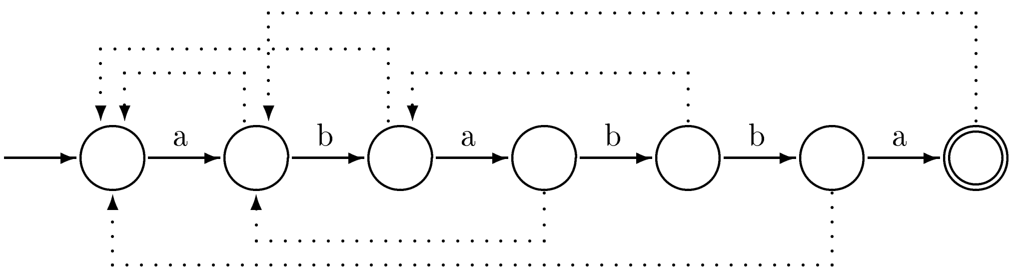

Figure 2.1 Search Automaton for the Pattern *ababba*

pattern P of length p, the automaton consists of exactly $p+1$ states, with a one-to-one correspondence with the $p+1$ prefixes of P. As a result of this, we use prefixes to denoted states whenever there is no confusion. The initial state corresponds to the null prefix ϵ of P, and the only final state corresponds to the pattern P itself. Figure 2.1 shows the automaton for the pattern *ababba*.

There are two types of transitions on the states of the automaton: the *forward* transitions and the *backward* transitions defined formally as follows:

> **Definition 2.1** Let x be a state (prefix) in the automaton, and a be a symbol in Σ. If xa is also a state, then $forward(x, a) = xa$. Otherwise, if $x = \epsilon$ then $forward(x, a) = x$, else $forward(x, a)$ is undefined.

> **Definition 2.2** For a prefix $x \neq \epsilon$, $backward(x) = y$, where y is the longest prefix of the pattern such that $|y| < |x|$ and y is a proper suffix of x.

In Figure 2.1, the forward transitions are shown by solid edges directed from left to right, with the label on each edge indicating the corresponding symbol. The backward transitions are shown by dotted edges directed from right to left. The forward transitions from the initial state to itself are not shown. In Figure 2.1, $forward(aba, b) = abab$ but $forward(ab, b)$ is undefined. Also, $backward(abab) = ab$ since ab is the longest prefix which is also a proper suffix of $abab$.

There are some observations that can be made about the transitions. There can be at most one forward transition, and at most one backward transition out of any state. All the forward transitions form a chain with the starting state as the first node and the final state as the last node. The backward transitions form a directed rooted tree, with the starting state as the root, and with the edges directed towards the root.

We start the searching process in the start state and at the first character of the text. The present state corresponds to the longest prefix of the pattern matched at the current location of the text. Intuitively, the forward transitions correspond to the case when there is a match between the current text and pattern characters. Whenever there is a match, we take the corresponding forward transition and go to the next state with a longer prefix. If there is no such match, we backtrack to a state with a smaller prefix using the backward transition, and continue our search from there.

We will not go into the details of how to build the search automaton here. We end this section showing the basic search loop:

```
state = ε
while not the end of text do
    while forward(state, symbol) is undefined do
        state ← backward(state)
    state ← forward(state, symbol)
    if state = finalstate then
        report a match
    symbol ← nextsymbol
```

2.3 Multiple Pattern Matching Problem

We have defined the multiple pattern matching (MPM) problem earlier in Chapter 1. In this section, we review the solution of Aho and Corasick [1], which can be viewed as a generalization of the KMP algorithm to multiple patterns. The Aho-Corasick algorithm, which we call the AC algorithm, preprocesses the set of patterns in $O(d \log \sigma)$ time, where $d = p_1 + \cdots + p_n$ is the sum of the lengths of all the patterns, and σ is the number of distinct characters of Σ that appear in some pattern. Like the KMP algorithm, the AC algorithm builds an automaton on the patterns and uses it to search the text in time $O(t \log \sigma + tocc)$ time, where $tocc$ is the size of the output, which is the total number of pattern-location pairs reported.

As in the KMP algorithm, there is a one-to-one correspondence between the states of the automaton and the prefixes of the patterns. As a result of this, we use prefixes to denote states whenever there is no confusion. The initial state corresponds to the

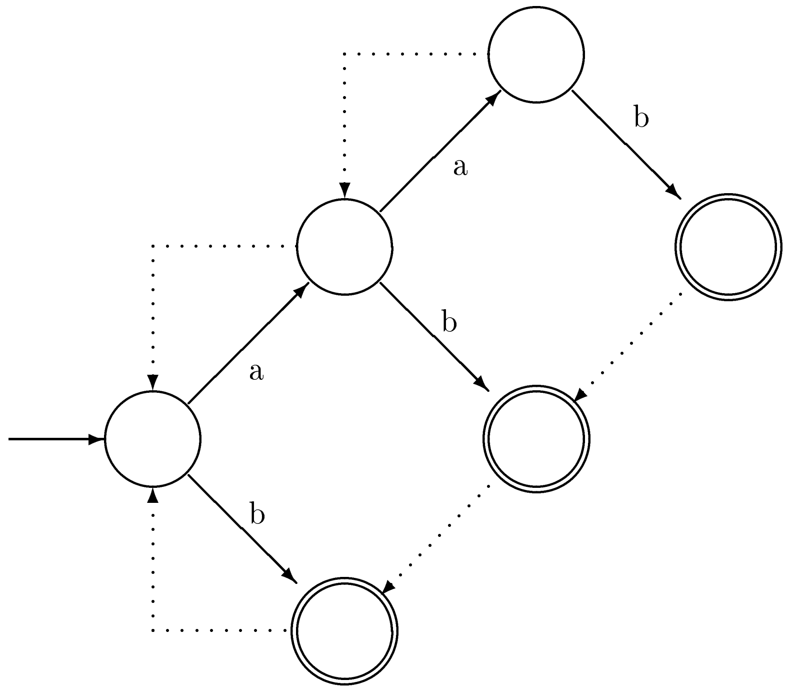

Figure 2.2 Search Automaton for the set of Patterns $\{b, ab, aab\}$

null prefix ϵ of P, and the final states correspond to the full patterns. Note that there can be as many final states as the number of patterns. Figure 2.2 shows the automaton for the set of patterns $\{b, ab, aab\}$.

As in the KMP algorithm, there are two types of transitions on the states of the automaton: the *goto* transitions and the *fail* transitions defined as follows:

Definition 2.3 Let x be a state (prefix) in the automaton, and a be a symbol in Σ. If xa is also a state, then $goto(x, a) = xa$. Otherwise, if $x = \epsilon$ then $goto(x, a) = x$, else $goto(x, a)$ is undefined.

Definition 2.4 For a prefix $x \neq \epsilon$, $fail(x) = y$, where y is the longest prefix of some pattern such that $|y| < |x|$ and y is a proper suffix of x.

In Figure 2.2, the goto transitions are shown by solid edges directed from left to right, with the label on each edge indicating the corresponding symbol. The fail transitions are shown by dotted edges directed from right to left. In Figure 2.2, $goto(aa, b) = aab$ but $goto(ab, b)$ is undefined. Also, $fail(aab) = ab$ since ab is the longest prefix which is also a proper suffix of aab. The goto and fail transitions in the AC automaton are the extensions of forward and backward transitions respectively in the KMP algorithm.

There are some important observations that can be made about the transitions. There can be at most one fail transition out of every state, but there can be more goto transitions out of a state. This explains the multiplicative $\log \sigma$ factor in the time bounds of the AC algorithm which does not exist in the KMP algorithm. We have to organize the goto transitions out of a state in some efficient way. One way, which is space-efficient, is to organize the goto transitions in some sort of a balanced binary search tree. Since there can be as many as σ goto transitions out of a state this will cause a $\log \sigma$ penalty in the time bounds.

Another important property about the transitions of an AC automaton is that the goto (and also fail) transitions form a directed rooted tree, with the starting state as the root, and with the edges directed away from (towards) the root.

We start the searching process in the start state and at the first character of the text. The present state corresponds to the longest prefix of some pattern matched at the current location of the text. The goto transitions correspond to a match between the current text and pattern characters. Whenever there is a match, we take the corresponding goto transition and go to the next state with a longer prefix. If there is no such match, we backtrack to a state with a smaller prefix using the fail transition, and continue our search from there.

We end this section showing the basic search loop:

$state = \epsilon$
while not the end of text **do**
  while $goto(state, symbol)$ is undefined **do**
    $state \leftarrow fail(state)$
  $state \leftarrow goto(state, symbol)$
  if $state \in finalstates$ **then**
    report all the matches
  $symbol \leftarrow nextsymbol$

2.4 Order Maintenance Problem

In the order maintenance problem, we define the following operations on a linearly ordered list L, initially containing one element. We attach an L to the name of every operation to emphasize that they are list operations.

$L\_Insert(x, y)$: Insert a new element y immediately after the element x in L.

$L\_Delete(x)$: Delete the element x from L.

$L\_Order(x, y)$: Return true if x is before y in L, false otherwise.

Dietz and Sleator [32] provide a dynamic data structure for solving the order maintenance problem in which all three operations can be implemented in worst-case $O(1)$ time. We call the data structure a *DS list*. All our dynamic dictionary matching algorithms, which will be presented in later chapters, make use of the dynamic data structure of Dietz and Sleator for the order maintenance problem. This versatile data structure has applications in other dynamic problems [33, 2, 39]. We believe that the DS list could become increasingly useful as the dynamic structures get increasingly complicated. The reason is that for compound structures such as strings, trees, sets, and graphs it is easy to define natural orders for which it is expensive (not a constant time operation) to determine the relative order of two objects. The DS list provides us a way to do this in constant time under suitable conditions.

We now give a brief description of the data structure of Dietz and Sleator. Actually the data structure comes in two varieties: one with amortized bounds and the other with worst-case bounds. The amortized version is very simple to state and implement. The worst-case version is interesting from a theoretical point of view.

2.4.1 Amortized Time Algorithm

We now present the amortized version. Most of the description is taken from [32]. The data structure consists of a circularly linked list of records. Each record r is labeled with an integer $v(r)$. The successor of a record is denoted $s(r)$. The set of integers available for labeling is $\{0, 1, \ldots, M-1\}$. This set is called the *arena*, and M the size of the arena. At any time M is chosen so that

$$M > n^2$$

always holds. The list initially contains an element (which is never deleted), called the *base* and denoted by b. The label of the base is arbitrary. For convenience we use $v_b(r)$ to denote $(v(r) - v(b)) \bmod M$. The labels of the records in the list are always maintained such that $v_b(r) < v_b(s(r))$.

With this scheme, L_Order and L_Delete are implemented easily. $L\_Order(x, y)$ is done by comparing $v_b(x)$ and $v_b(y)$. $L\_Delete(x)$ is done by simply deleting x from the list.

Before describing the insertion algorithm, we define some notation that simplifies the description of the algorithm. Suppose $L\_Insert(x, y)$ is the operation being performed. Let v_0 denote $v(x)$, v_1 denote $v(s(x))$, v_2 denote $v(s^2(x)) = v(s(s(x)))$, and so on. Let $w_i = (v_i - v_0) \bmod M$ for $0 \le i < n$. Define $w_n = M$. The insertion algorithm walks down the list until $w_j > j^2$:

$j \leftarrow 1$
while $w_j \le j^2$ **do**
    $j \leftarrow j + 1$

Relabel the $j-1$ records $s^1(x), \ldots, s^{j-1}(x)$ with the labels

$$v(s^k(x)) = \left(\lfloor w_j \frac{k}{j} \rfloor + v_0\right) \bmod M.$$

After this step, the labels between x and $s(x)$ differ by at least 2. In other words, there will be at least one integer between these two labels which is not the label of any other record. Therefore, we can insert y after x by making this integer as the label of y.

This method gives us worst-case $O(1)$ time for *L_Delete* and *L_Order* operations, but gives an amortized $O(\log n)$ time for *L_Insert* operation. To get an amortized $O(1)$ time for the insertion, we break the list into sublists, with each sublist storing approximately $\Theta(\log n)$ records. We apply the strategy above to the sublists, treating each sublist as an element. Records within each sublist are given monotonically increasing labels. An inserted element is given a label which is an intermediate value between the labels of its neighbors. Whenever necessary, all the elements within the sublist are relabeled. For $L\_Order(x, y)$, if x and y are in the same sublist, their labels are directly compared. If they are in different sublists, then the relative order of their sublists within the list gives the relative order of x and y. When we delete an element, we first delete the element from its sublist. If the sublist becomes empty, we delete it from the list. If a sublist is found to contain more than $\Theta(\log n)$ elements, we break it into smaller sublists, each containing $\Theta(\log n)$ elements. We also relabel all the elements in the newly created sublists in the process.

2.4.2 Worst-Case Algorithm

We assume for now that the list contains at most N elements, where N is a fixed integer. The data structure is a tree of height 4. The root has a limit of $O(N/\log^3 N)$

on the number of children allowed. All other internal nodes have a limit of $O(\log N)$ on the number of children allowed. The algorithm builds the *dense sequential file* data structure of Willard [64, 65] on the children of the root. The algorithm assigns file addresses to children of the root and real numbers to the other nodes. With this scheme an $L\_Order(x, y)$ query is performed as follows: Let w be the least common ancestor of x and y, and x' and y' be the children of w on the paths from w to x and y respectively. The relative order of x and y is the same as that of x' and y'. If w is the root of the tree then the dense file addresses of x' and y' can be compared in constant time to give the relative order. Otherwise the real numbers associated with x' and y' give the order.

Whenever a new insertion causes the number of allowed children of a node x to exceed the limit of x, the algorithm splits x into two new nodes, called *overflow nodes*. Since we need to perform an insertion in constant time, the actual split is distributed over later insertions. To make N variable, we keep N as a power of 2. When the number of elements in the list exceed a fixed fraction of N, we start copying the elements into a larger data structure with a bigger value for N. This is done in an incremental fashion over subsequent insertions to keep the time for a single insertion $O(1)$. Deletions are handled similarly; when an item is deleted it is marked as deleted (but not actually deleted). When making a fresh copy of the data structure, the marked leaves are discarded. This process is reminiscent of incremental garbage collection.

2.5 Dynamic Parentheses Maintenance Problem

We define the *dynamic parenthesis maintenance problem* to be the problem of maintaining a sequence B, of well-balanced parentheses under the following three operations:

Insert parentheses: A *valid insertion* is a pair of locations (l_i, r_i) in B such that adding a left parenthesis immediately left of l_i and a right parenthesis immediately right of r_i leaves B well-balanced and the new parentheses match. Given such a valid insertion pair (l_i, r_i), insert the above mentioned matching parenthesis into B.

Delete parentheses: Given a pair (l_i, r_i) of matching parentheses, remove l_i and r_i from B.

Find enclosing parentheses: Given a parenthesis l_i, or equivalently its matching r_i, find the nearest parenthesis pair (l_j, r_j) such that (l_j, r_j) enclose (l_i, r_i). That is, (l_j, r_j) are a matching parentheses pair that enclose (l_i, r_i), but there is no matching pair (l_k, r_k) such that (l_j, r_j) enclose (l_k, r_k), and (l_k, r_k) enclose (l_i, r_i).

The dynamic parenthesis maintenance problem was first considered by Güting and Wood [46] who gave a solution where each operation takes $O(\log n)$ time on a parenthesis string of size n. Güting and Wood applied their solution to problems in computational geometry. Recently, Amir et al. [13] improved the time of each operation to $O(\log n/\log\log n)$. The dynamic parenthesis maintenance problem plays a major role in dynamic dictionary matching as we will see later.

In this section we briefly describe the solution of Güting and Wood. The parentheses are represented as leaves in a balanced binary tree, called a *parenthesis tree*. In addition, each pair of matching parentheses are connected to each other by a separate link. The parenthesis tree is augmented with a BALANCE information in the following way: For every node p in the tree, let the pair $\text{BALANCE}(p) = \langle close(p), open(p)\rangle$ denote the number of unmatched closing and unmatched opening parentheses in its subtree.

For each node p of the tree, the $\text{BALANCE}(p)$ can be computed recursively as follows:

(a) *p is a leaf.* Then

$$\text{BALANCE}(p) = \begin{cases} \langle 0, 1\rangle & \text{if } p \text{ is `('}, \\ \langle 1, 0\rangle & \text{if } p \text{ is `)'}. \end{cases}$$

(b) *p is an internal node with children q and r.* We compute

$$\text{BALANCE}(p) = \text{BALANCE}(q) \oplus \text{BALANCE}(r),$$

where

$$\langle i, j\rangle \oplus \langle k, l\rangle = \begin{cases} \langle i, j-k+l\rangle & \text{if } j \geq k, \\ \langle i-j+k, l\rangle & \text{otherwise.} \end{cases}$$

Figure 2.3 shows a parenthesis tree with along with the BALANCE information. It is straightforward to update the balance information during the insertion and deletion of individual parentheses as long as they are valid. Finding the nearest enclosing

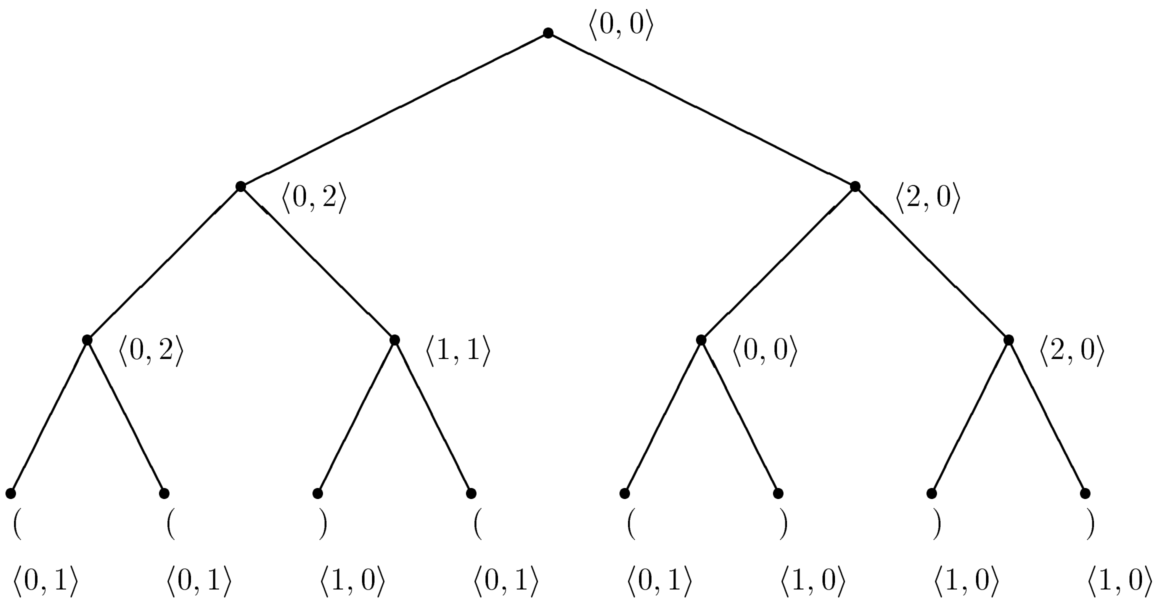

Figure 2.3 A Sample Parenthesis Tree

parenthesis is achieved with the help of two auxiliary routines GO-UP and GO-DOWN as described below.

FIND-ENCLOSE(l_i, r_i)
  Let BA be a global variable
  Find the next leaf q to the right of r_i
  if q is a right parenthesis **then return** q
  Initialize BA $\leftarrow \langle 0, 1\rangle$
  return GO-UP(q)

For two BALANCEs $\langle i, j\rangle$ and $\langle k, l\rangle$, we say $\langle i, j\rangle < \langle k, l\rangle$ whenever $j < k$.

GO-UP(q)
  $p \leftarrow parent(q)$
  if $p = nil$ **then return** nil
  if q is p's right child **then return** GO-UP(p)
  Suppose r is p's right child
  if BA $<$ balance(r) **then**
    GO-DOWN(r)

else

BA $\leftarrow$ BA $\oplus$ BALANCE(r)

GO-UP(p)

GO-DOWN(p)

if p is a leaf **then return** p

Let q, r be p's left and right children respectively

if BA $<$ balance(q) **then return** GO-DOWN(q)

BA $\leftarrow$ BA $\oplus$ BALANCE(q)

GO-DOWN(r)

The BALANCE information tells whether the enclosing parenthesis, that we are seeking, is contained in the subtree. We start our search for the enclosing parenthesis of an open parenthesis *op* by executing a call to GO-UP, with balance information $\langle 0, 1\rangle$. Intuitively, we are looking for a subtree immediately to the right of *op* that contains sufficient number of unmatched close parentheses so that one of them is the enclosing right parenthesis that we seek. If the subtree contains sufficient number of close parentheses, we go down the subtree to get to the enclosing parenthesis using GO-DOWN. If the subtree does not contain sufficient number of close parentheses, then we have to go further up the tree and continue the search. In the process, we need to incorporate any change in the balance information because the subtree may balance some of the existing unmatched open parentheses, and may add some more unmatched open parentheses. We use GO-UP for this purpose.

We can extend the scheme of Güting and Wood to work with trees with multiple arity (> 2), by observing that the $\oplus$ operator is associative. In fact, we use this fact in Chapter 4 to build a parenthesis tree using a-b trees.

Chapter 3

Multiple Matching with Strings: Algorithm

3.1 Introduction

Amir, Farach, Galil, Giancarlo, and Park (AFGGP) initiated the study of the *dynamic dictionary matching problem* [9, 12]. We are given a collection of patterns $D = \{P_1, P_2, \ldots, P_s\}$, called the *dictionary*, that can change over time. The basic matching operation is to *search* a text $T[1, t]$ and report all occurrences of dictionary patterns in the text. The dictionary can be changed by *inserting* or *deleting* individual patterns.

The *static dictionary matching problem*, in which inserts and deletes are not supported, was addressed in earlier papers by Aho and Corasick [1] and Commentz-Walter [29]. Algorithms for both the static and dynamic problems have applications to database searches and molecular biology [1, 12].

The two algorithms for static dictionary matching can be summarized as follows. We use d to denote the total size of all the patterns in the dictionary; we use Σ to denote the alphabet and σ to denote the number of distinct characters that appear in the dictionary. The Aho-Corasick algorithm (AC) builds the dictionary D in time $O(d \log \sigma)$ and searches a text $T[1, t]$ in $O(t \log \sigma + tocc)$ time, where $tocc$ is the total number of occurrences reported. The time complexity of this algorithm is considered linear as the $\log \sigma$ factor is usually implicit while stating the time bounds.

The AFGGP algorithm for dynamic dictionary matching is based on using a suffix tree [63, 57, 28] as an automaton. Each state corresponds to a substring of some pattern. The AFGGP algorithm is able to insert or delete a pattern $P[1, p]$ in time $O(p \log d)$, and it performs a search in time $O((t + tocc) \log d)$. If inserts and deletes are frequent enough, this algorithm is better than the simple alternative of using the AC algorithm for searches and rebuilding the dictionary in $O(d \log \sigma)$ time at each update. It may be a little surprising that the AFGGP time bounds appear not to depend on the alphabet size; actually, the $\log d$ factor is short for $\log d + \log \sigma$, but the $\log \sigma$ factor is subsumed by the $\log d$ factor since $\sigma \leq d$.

In this chapter, we present another algorithm for dynamic dictionary matching that addresses three questions raised by the AFGGP algorithm and time bounds.

First, is the idea of failure functions and the AC automaton of any use at all for dynamic dictionary matching? Our algorithm is based on the AC automaton. With a suitable choice of underlying data structures, our method achieves a search time of $O((t + tocc) \log d)$ and an update time of $O(p \log d)$ matching the AFGGP bounds. While the suffix tree approach is quite novel and interesting, perhaps we should not abandon the idea of failure functions in dynamic dictionary matching.

Second, are other tradeoffs between search and update times possible? One would expect that in some applications, the updates are relatively infrequent or the text strings are much longer than the patterns. Under either condition we would prefer a search time better than $O((t + tocc) \log d)$ while tolerating an update time worse than $O(p \log d)$. It is interesting to ask: how good can we make the update time if we insist that the search time match the AC bound of $O(t \log \sigma + tocc)$ for the static problem?

We show that our algorithm can also use a different data structure such that for any constant $k \geq 2$, it can achieve search time $O(t(k + \log \sigma) + tocc\ k)$ and update time $O(p(kd^{1/k} + \log \sigma))$. We thereby match the static search time of [1] and have a sublinear update time if the patterns are not very long relative to d.

Third, can one match the $O(d)$ preprocessing time of the AC algorithm? The AFGGP algorithm builds the initial dictionary by repeated insertion of patterns in time $O(d \log d)$. We would like to avoid these insertions, especially if we are going to increase their cost. We show that regardless of the choice of the data structure, we can build our initial dictionary in $O(d \log \sigma)$ time.

One advantage of our approach is that it generalizes to two dimensions, while the AFGGP algorithm does not seem to generalize. In Chapter 5, we describe how to combine our dynamic dictionary framework with some other ideas, to solve the dynamic dictionary matching problem with square patterns, achieving the same time bounds for insertion, deletion, and search as in one dimension. Later in Chapter 6, we extend our framework to solve the general two dimensional multiple pattern matching problem, where patterns can be of any rectangular shape.

In sum, we show that it is possible to extend the AC approach to dynamic dictionary matching and achieve the same preprocessing and search times that they do, while achieving an update time sublinear in d. Alternatively, we can match the search and update times of the AFGGP algorithm and improve the dictionary construction time to $O(d \log \sigma)$. Our framework can be extended to two dimensional dictionary matching.

The rest of this chapter is organized as follows. In Section 3.2, we develop an automata-based framework for dynamic dictionary matching. We use this framework to develop two separate algorithms for the dynamic dictionary matching. In Section 3.3, we present a simple and practical algorithm. In Section 3.4, we present an alternate algorithm, which can be modified to improve the search time. In Section 3.5, we show how to modify the underlying data structures, to improve the search time. In Section 3.6, we describe how to construct the initial dictionary in $O(d \log \sigma)$ time.

3.2 An Automaton for Dynamic Dictionary Matching

Let $D = \{P_1, \ldots, P_s\}$ be a dictionary of patterns, where each P_i, $1 \leq i \leq s$, is a string over a finite alphabet Σ. For convenience, we assume that the empty string ϵ is always a pattern in the dictionary. We append to each pattern a special symbol \$ that does not occur elsewhere in any pattern or text. We shall henceforth assume that $\$ \in \Sigma$ and that \$ is the largest symbol in the lexicographic order. Throughout this chapter, we generally use w, x, y, z to denote a prefix of some pattern, and a, b to denote a character of Σ. We use the following dictionary as an example to explain various definitions and concepts of our automaton.

> **Example 3.1** Suppose $\Sigma = \{a, b, \$\}$. Let $D = \{\$, b\$, aab\$\}$ be a sample dictionary where every pattern is appended with the special symbol \$.

We make the same assumptions on the character set Σ as in the papers on suffix trees [63, 57, 28, 9, 12]:

> **Assumption 3.1** Each character is represented by a constant number of bytes.

> **Assumption 3.2** The relative order of any two characters can be determined in constant time.

Recall the description of the AC algorithm presented in Chapter 2. The AC algorithm builds an automaton on the patterns and uses it to search a text. Each state in the automaton corresponds to a prefix of some pattern in D. We also defined two important (partial) functions, *goto* and *fail*, that describe the transitions in the automaton. The basic search loop is:

$state = \epsilon$
while not the end of text **do**
  while $goto(state, symbol)$ is undefined **do**
    $state \leftarrow fail(state)$
  $state \leftarrow goto(state, symbol)$
  if $state \in finalstates$ **then**
    report all the matches
  $symbol \leftarrow nextsymbol$

For any given choice of *state* and *symbol* we may have to take the *fail* transition repeatedly, but this shortens the length of the new *state*. The total time needed to search a text T using this algorithm is $O(t(g+f))$, where $g+f$ is the time needed to make one evaluation of *goto* and *fail* [1].

Like AC, we store the *goto* function as a directed rooted tree, where the nodes correspond to automaton states (pattern prefixes). The edges of the tree are directed away from the root and every edge is labeled with a character from Σ. The fact that the dictionary is dynamic has no impact on the way the *goto* function is computed, and therefore, we can still keep it as tree, as done in the AC algorithm. We organize the outgoing edges of an internal node into a balanced search tree with the edge labels as the keys for search, as suggested in [1]. The *degree*, or the maximum number of children of an internal node, of a *goto* tree is bounded by σ. With this modification, computing each *goto* takes $O(\log \sigma)$, which is $O(\log d)$. We also store with each node a pointer to its parent node. For each prefix x of a pattern, we keep a count of how many patterns have x as a prefix.

We call a string a *normal prefix* if it is a prefix of some pattern in D. For each proper prefix x we also define an *extended prefix*, $x\$$, by appending the character \$; the extended prefixes help in detecting patterns. The corresponding states are called normal or extended. In Example 3.1, the set of normal prefixes is $\{\epsilon$, a, b, \$, aa, $b\$$, aab, $aab\$\}$ and the set of extended prefixes is $\{\$$, $a\$$, $b\$$, $aa\$$, $aab\$\}$. Notice that some prefixes are both normal and extended. We extend the definition of *fail* to accommodate the extended prefixes as follows: Let w be a prefix (normal or extended). $fail(w) = x$ such that $|x| < |w|$ and x is the longest suffix of w such that

x is a normal prefix. In Example 3.1, $fail(aab\$) = b\$$ but $fail(aa\$) \neq a\$$ since $a\$$ is not a normal prefix.

We recognize patterns as follows. When we reach a position of a text, we pretend that the next symbol is a \$. If we can make a transition to some normal prefix ending with a \$, then we know that a pattern has been matched at that position, since any normal prefix ending with a \$ must be a pattern in the dictionary. By applying $fail$ repeatedly, we can report all the matching patterns in the order from the longest pattern to the shortest.

Suppose we are searching the text *abaabba* for the occurrences of the patterns in the dictionary D of Example 3.1. After reading the prefix *abaab* we will be in the normal state *aab*. When we pretend to read \$ as the next symbol, we will *temporarily* enter a state $aab\$$ and since this is a normal state in the dictionary we report that a pattern *aab* is recognized at the current location of the text. If we take $fail(aab\$) = b\$$ we see that we have matched another (smaller) pattern b. Again, if we take $fail(b\$) = \$$ we realize that no more patterns can be matched as this corresponds to the empty pattern ϵ. Since we keep track of the state *aab*, we can continue our search by reading the next symbol b from the text.

In [1], $fail$ is stored as a directed, rooted tree with the edges pointing towards the root. If $fail(s) = t$ then there will be an edge $s \to t$. The degree of the $fail$ tree is unbounded. Unlike the *goto* function, the $fail$ function cannot be maintained efficiently as a tree, when the dictionary is dynamic. Therefore, we use a new method to store $fail$ that enables us to insert new patterns, updating the $fail$ function. Before describing our representation of $fail$, we explain some auxiliary prefixes and data structures that we use.

Let $* \notin \Sigma$ be a new symbol such that $* > a$ for any $a \in \Sigma$ in lexicographic comparison ($* > \$$). For every prefix $w \in \Sigma^*$ we define $*w$ as the *complement* of w. We call w a *regular prefix*, and $*w$ a *complementary prefix*. For convenience, we assume that $fail(*x) = fail(x)$ for every regular prefix x. We define a total ordering on the set of prefixes and their complements and call it the *inverted order* denoted by $<_{inv}$. For two distinct strings w and x, $w <_{inv} x$ if w^R comes before x^R in the lexicographic ordering, where x^R is the reverse of the string x.

Example 3.2 Consider the list of prefixes of D in Example 3.1 in the inverted order. It can be represented as: $S = \epsilon,\ a,\ aa,\ *aa,\ *a,\ b,\ aab,\ *aab,\ *b,\ \$,\ a\$,\ aa\$,\ aa\$,\ *a\$,\ b\$,\ aab\$,\ *aab\$,\ *b\$,\ *\$,\ *$.

Amir and Farach point out that there is an equivalent, but more algorithmic, description of $<_{inv}$ and the complementary prefixes, which some readers may find more intuitive. Take all the prefixes, reverse them, and then sort them in lexicographic order. Now define a compacted trie with all the reversed prefixes as nodes and conduct an Euler tour of that trie, listing the prefix nodes as they are visited. The Euler tour order is precisely the $<_{inv}$ order if we adopt the convention that the incoming visit to x^R in the trie corresponds to x and the outgoing visit from x^R in the trie corresponds to $*x$.

The number of extended prefixes is at most equal to the number of normal prefixes, so the number of regular prefixes is $O(d)$. The number of complementary prefixes is exactly equal to the number of regular prefixes. Therefore, the total number of prefixes in our dictionary structure is only $O(d)$, independent of Σ.

Let S denote the set of all normal and extended, regular and complementary prefixes of patterns in D. An important property of S is that for any string x, if S contains x then S contains every prefix of x. We use this property later in our insertion algorithm. All string comparisons are made with respect to $<_{inv}$ ordering unless stated otherwise. For a nonempty prefix $x \in S$, $pred(x)$ is the largest prefix in S smaller than x. We need to compute $pred(x)$ when inserting a new prefix to know where the new prefix lies in the $<_{inv}$ order. In Example 3.2, $pred(aab) = b$ and $pred(b) = *a$. The following lemmas state some important properties of the prefixes in S that follow directly from the definition of $<_{inv}$ and the character $*$.

Lemma 3.1 Let $w, x \in S$ be arbitrary regular prefixes. Let $y \in S$ be any prefix.
1. $w <_{inv} *w$; w is smaller than its complement.
2. $w <_{inv} x <_{inv} *w$ if and only if $w <_{inv} *x <_{inv} *w$; if we replace a regular prefix with a '(' and its complement with a ')' then the prefixes of S in the $<_{inv}$ order yield a list of well balanced parentheses.
3. If $w <_{inv} y <_{inv} *w$ then $y = y'w$ for some y'; w and $*w$ are respectively the smallest and largest prefixes in S with the suffix w.

Lemma 3.2 Let xa be the prefix we are inserting into S. Suppose $yb \neq xa$ is a prefix already in S. Then the following relations hold:
1. $\epsilon <_{inv} xa$.
2. If $b <_{inv} a$ then $yb <_{inv} xa$. Similarly if $a <_{inv} b$ then $xa <_{inv} yb$.

3. If $b = a$ then $yb <_{inv} xa$ if and only if $y <_{inv} x$. Since x and y must already be in S we can determine whether $y <_{inv} x$ from S alone.

Lemma 3.2 suggests a way to obtain the relative order of two prefixes without making a complete lexicographic comparison. We utilize Lemma 3.2 to compute the *pred* function by building an auxiliary search tree, called ST, on the top of all the prefixes of S. The elements of ST are basically pointers to the states of the *goto* tree sorted by the inverted order of the corresponding prefixes. We actually store ST as an a-b tree [58] with all the prefixes as leaves. For any state corresponding to a prefix yb we can determine the character b and the state with the prefix y in constant time as y is the parent of yb in the *goto* tree and hence this information is available with the parent pointer of yb. To utilize Lemma 3.2 we need to know the relative order of all the existing prefixes in S. Since we implement ST as an a-b tree, we can determine the relative order of two prefixes in $O(\log d)$ time.

We now describe one way to compute $pred(xa)$ for a nonempty prefix xa not yet in S. Our aim is to find the largest prefix in ST smaller than xa. We start the search at the root of ST and proceed towards the leaves. Suppose yb is the key associated with an internal node of ST. At the next level we take the right child of the node if $yb <_{inv} xa$ and the left child otherwise. With this scheme we need $O(\log d)$ tree comparisons to find $pred(xa)$. From now on we assume that we have a routine FINDPRED that computes the *pred* of a new prefix. We can conclude that:

Lemma 3.3 Let xa be a prefix to be inserted into S. We can compute $pred(xa)$ using FINDPRED(xa) in $O(\log^2 d)$ time.

Proof We can determine $pred(xa)$ with $O(\log d)$ tree comparisons. Since ST is organized as an a-b tree, the relative order of two existing prefixes can be determined in $O(\log d)$ time. Therefore, by Lemma 3.2 each tree comparison takes $O(\log d)$ time. From this it follows that we can determine $pred(xa)$ in $O(\log^2 d)$ time. □

To reduce the time to compute *pred*, we use the data structure of Dietz and Sleator [32], which was presented in Section 2.4. We store the prefixes of S in an auxiliary list data structure that we call the DS list, or DSL. Using DSL we can determine the relative order of two prefixes in constant time instead of $O(\log d)$ time as achieved by using only ST.

Lemma 3.4 Let x and y be two prefixes in S. By building DSL for all the prefixes of S, we can determine the relative order of x and y in constant time. Hence, we can compute *pred* in $O(\log d)$ time.

When we insert a new prefix xa, we need to find the value of $fail(xa)$. Below we show how to use $pred(xa)$ to compute $fail(xa)$ quickly.

Lemma 3.5 Suppose w is a normal regular prefix, and $w <_{inv} x <_{inv} *w$. Then $fail(x) = w$ if and only if there is no normal regular y such that $w <_{inv} y <_{inv} x <_{inv} *y <_{inv} *w$.

Proof By Lemma 3.1, w is a suffix of x. If there is a normal regular prefix $y \in S$ such that $w <_{inv} y <_{inv} x <_{inv} *y <_{inv} *w$ then y must be a suffix of x longer than w. By definition, $fail(x) \neq w$ because of the presence of y. On the other hand, if there is no such y then $fail(x) = w$. □

In Example 3.2, $fail(aab) = b$ since there is no normal y such that $b <_{inv} y <_{inv} aab <_{inv} *y <_{inv} *b$, but $fail(aab) \neq \epsilon$ since b can play the role of y in that case.

The above lemma gives us a way to compute $fail$ indirectly by keeping regular and complementary prefixes in the inverted order. In the next two sections, we use the framework developed in this section to provide efficient solutions to the dynamic dictionary matching problem.

3.3 Algorithm using a Parenthesis Tree

In this section, we present our first solution to the dynamic dictionary matching problem. This solution is simple to understand and implement. In fact, we have implemented this algorithm, and we shall see the implementation details in Chapter 4.

In the last section, we noted that we need an efficient way to compute the $fail$ function to make a dictionary dynamic. Lemma 3.5 provides a way to compute $fail$ when the prefixes, both regular and complementary, of a dictionary are maintained in the inverted order.

Consider the set S of regular and complementary prefixes, sorted in the inverted order. We define a *parenthesis mapping* on S as follows: In S, replace each regular prefix with a '(', and its complement with a ')'. The result is a set of balanced parentheses, with each matching pair corresponding to a regular prefix and its complement, by Lemma 3.1.

For a prefix $x \in S$, let $paren(x)$ denote the parenthesis to which x is mapped under the parenthesis mapping. We can derive from Lemma 3.5 and the definition of parenthesis mapping that:

Lemma 3.6 Let w, x be two regular prefixes such that $fail(x) = w$. Then $paren(w)$ is the *nearest enclosing parenthesis* of $paren(x)$.

Lemma 3.6 gives a way to compute the $fail$ function. Recall from the last section that we build a balanced search tree ST on the top of all the prefixes. ST is organized as an *a-b* tree, where all the leaves correspond to prefixes in S, in the inverted order. We transform this tree into a parenthesis tree by augmenting it with the parenthesis mapping information. In other words, each leaf of the tree is considered either an open or a close parenthesis, depending on whether it corresponds to a regular or a complementary prefix. We also build the BALANCE information described in Section 2.5 for parenthesis trees. We call the resulting tree a parenthesis tree, denoted by PT, which is essentially ST augmented with the parenthesis mapping information.

Under the above scheme, computing the $fail$ of a prefix x is equivalent to finding the nearest enclosing parenthesis of $paren(x)$, say $paren(w)$, and taking the associated prefix w, using Lemma 3.6.

We now give the pseudocode for searching a text, inserting a pattern, and deleting a pattern. We use prefixes instead of states for clarity.

Algorithm 1 Code for searching a text.

```
SEARCH(T = t_1 ... t_n)
    state ← ϵ
    for i ← 1 to n do
        while goto(state, t_i) is undefined
            state ← fail(state)
        state ← goto(state, t_i)
        /* Pretend a $ is read to check if any patterns match */
        temp ← goto(state, $)
        if temp is not normal then temp ← fail(temp)
        while temp ≠ $ do /* Report all non-empty patterns */
            /* Since temp ends in $ we have matched a pattern */
            Print the pattern associated with temp
            temp ← fail(temp) /* See if any smaller patterns match */
```

Algorithm 2 Code for inserting a pattern into a dictionary.

INSERT$(P = p_1 \ldots p_m)$ /* $p_m = \$$ */

Suppose $p_1 \ldots p_j$ is the longest prefix of P shared by some other pattern.
Increment the reference count for the prefixes of $p_1 \ldots p_j$.
for $i \leftarrow j+1$ **to** m **do**
1 Let $x = p_1 \ldots p_{i-1}$. Let $a = p_i$. /* xa to be inserted, x already in S */
2 Compute $y = pred(xa)$ using FINDPRED.
3 *insert* xa into PT after y.
4 *L_Insert* xa into DSL after y.
5 Compute $z = pred(*xa)$ using FINDPRED
6 *insert* $*xa$ into PT after z.
7 *L_Insert* $*xa$ into DSL after z.
8 Repeat steps 1–7 to insert $xa\$$ and $*xa\$$. /* Extended prefixes */

Algorithm 3 Code for deleting a pattern from a dictionary.

DELETE$(P = p_1 \ldots p_m)$ /* $p_m = \$$ */

Suppose $p_1 \ldots p_j$ is the longest prefix of P shared by some other pattern.
Decrement the reference count for the prefixes of $p_1 \ldots p_j$.
for $i \leftarrow m$ **downto** $j+1$ **do**
  Let $x = p_1 \ldots p_i$ /* x is a normal prefix */
  if $x\$$ is still in S **then**
    delete $x\$$ and $*x\$$ from PT.
    L_Delete $x\$$ and $*x\$$ from DSL.
  delete x and $*x$ from PT.
  L_Delete x and $*x$ from DSL.

The correctness and running time bounds for the above procedures follow from the previous lemmas.

Theorem 3.1 Let D be a dictionary of patterns over an alphabet Σ. Using the algorithm based on a parenthesis tree, we can search a text T in time $O(t(\log\sigma + \log d) + tocc \log d)$, where $tocc$ is the total number of patterns reported. We can insert or delete a pattern P in time $O(p(\log\sigma + \log d))$. Moreover, we require only $O(d)$ space to store the automaton.

3.4 Algorithm using a Forest of Fail Trees

In this section, we present another solution to the dynamic dictionary matching problem. In the previous section, we used a parenthesis tree to compute the $fail$ function. In this section, we instead use a forest of fail trees to compute the $fail$ function. This approach has the advantage that it provides a trade-off between search and update times as we shall see in the next section. We start with a lemma which follows from Lemma 3.5.

Lemma 3.7 Let x be a regular prefix. Suppose $pred(x) = w$. If w is normal and regular then $fail(x) = w$ else $fail(x) = fail(w)$.

Proof If w is normal and regular then $w <_{inv} x <_{inv} *w$. From the definition of $pred$, there is no normal regular y such that $w <_{inv} y <_{inv} x <_{inv} *w$. Hence by Lemma 3.5, $fail(x) = w$. Suppose that w is not both normal and regular, and that $fail(w) = z$. Then from Lemma 3.5, $z <_{inv} w <_{inv} *z$ and there is no normal regular y such that $z <_{inv} y <_{inv} w <_{inv} *y <_{inv} *z$. Since x is right after w in the $<_{inv}$ order the same conditions hold for x. Hence $fail(x) = z = fail(w)$. □

In Example 3.2, $fail(aab) = b$ since $b = pred(aab)$ is normal and regular. Similarly $fail(b) = \epsilon = fail(*a)$ and one may observe that $pred(b) = *a$.

The routine for searching a text is the same as the SEARCH algorithm given at the end of the previous section, except that the $fail$ function is computed differently. We show later how INSERT, and DELETE can be implemented using the $fail$ and $pred$ functions on the set S. We first describe how $fail$ is represented. We store $fail$ as a forest of a-b trees, which we call fail trees. Amir and Farach [9] use 2-3 trees in their suffix-tree automaton for a similar purpose. Each tree is called a *fail tree*, and the forest is called a *fail forest*. There is a one-to-one correspondence between the set of a-b trees and the set of normal regular prefixes of S. If w is a normal regular prefix, then T_w denotes the a-b tree associated with w and contains as leaves all prefixes x such that $fail(x) = w$. In Example 3.2, T_a contains $\{aa, *aa\}$ as leaves, whereas T_{aab} is an empty tree. The root of the tree T_x contains a pointer to x. Each prefix $x \in S$ is a leaf in exactly one a-b tree, T_y, where $y = fail(x)$. The leaves of any a-b tree are sorted in $<_{inv}$ order. For any x, $fail(x)$ can be computed by starting at the leaf x finding the root of the tree T_y and taking the pointer to y.

When we insert a new pattern P into D, we insert its prefixes and extended prefixes into S, in increasing order of length. Thus when we insert a prefix x into S, we have already inserted all the prefixes of x. Using Lemmas 3.3 and 3.4 we can find $w = pred(x)$. From Lemma 3.7 we can obtain $y = fail(x)$. If $w = y$ insert x into T_y as the leftmost leaf. Otherwise, insert x into T_y right after w. We can similarly find $z = pred(*x)$ (z could be x) and insert $*x$ into T_y right after z. We keep a separate bidirectional link between x and $*x$. Similarly when we delete a pattern P from D, we delete its prefixes in the order of decreasing lengths.

After inserting x and $*x$ into S we must create T_x. For this we must first identify those prefixes whose $fail$ value changes to x. By Lemma 3.5, these are the prefixes with a suffix x and whose current $fail$ value is y. But these are exactly the leaves of T_y properly enclosed between x and $*x$. We change $fail$ for these nodes by a special *split* of T_y into T_y and T_x. We can similarly handle the case when a normal prefix x with $fail(x) = y$ becomes extended as a result of deletion of some pattern. In this case we fuse T_y and T_x into T_y with a special *concatenate*. These special operations were used similarly in [9].

As an example, consider inserting a new pattern $ab\$$ into the dictionary D of Example 3.1. For this we need to insert the prefixes a, $a\$$, ab, and $ab\$$ into the set S of Example 3.2. Since a and $a\$$ are already present in S we simply increment the reference count for them. We insert ab after $pred(ab) = b$, and $*ab$ after $pred(*ab) = *aab$. After this we have $S = \{\ldots, b, ab, aab, *aab, *ab, *b, \ldots\}$. T_b contained $\{aab, *aab\}$ as leaves before the insertion of ab and $*ab$. We create T_{ab} by splitting T_b as described above. After this step, T_{ab} contains $\{aab, *aab\}$ as leaves, and T_b contains $\{ab, *ab\}$ as leaves. We similarly insert $ab\$$ and $*ab\$$.

We now give the pseudocode to describe our procedures for inserting and deleting a pattern from a dictionary. We use prefixes instead of states for clarity.

Algorithm 4 Code for inserting a pattern into a dictionary.

INSERT($P = p_1 \ldots p_m$) /* $p_m = \$$ */

Suppose $p_1 \ldots p_j$ is the longest prefix of P shared by some other pattern.
Increment the reference count for the prefixes of $p_1 \ldots p_j$.
for $i \leftarrow j + 1$ **to** m **do**
Let $x = p_1 \ldots p_{i-1}$. Let $a = p_i$. /* xa to be inserted, x already in S */
Compute $y = pred(xa)$ using FINDPRED.
Compute $w = fail(xa)$ using Lemma 3.7.
if $w \neq y$ **then**
insert xa into T_w right after y.

Else *insert* xa into T_w as the leftmost leaf.
5 *insert* xa into ST after y.
6 *L_Insert* xa into DSL after y.
7 Compute $z = pred(*xa)$ using FINDPRED /* $w = fail(*xa)$. */
8 *insert* $*xa$ into T_w right after z.
9 *insert* $*xa$ into ST after z.
10 *L_Insert* $*xa$ into DSL after z.
11 Repeat steps 1–10 to insert $xa\$$ and $*xa\$$. /* Extended prefixes */
12 Form T_{xa} by a *split* of T_w into T_w and T_{xa}.

Algorithm 5 Code for deleting a pattern from a dictionary.

DELETE($P = p_1 \ldots p_m$) /* $p_m = \$$ */
Suppose $p_1 \ldots p_j$ is the longest prefix of P shared by some other pattern.
Decrement the reference count for the prefixes of $p_1 \ldots p_j$.
for $i \leftarrow m$ **downto** $j + 1$ **do**
Let $x = p_1 \ldots p_i$ /* x is a normal prefix */
if $x\$$ is still in S **then**
delete $x\$$ and $*x\$$ from their fail tree.
delete $x\$$ and $*x\$$ from ST.
L_Delete $x\$$ and $*x\$$ from DSL.
Let $y = fail(x)$. Fuse T_x and T_y into T_y by a *concatenate.*
delete x and $*x$ from T_y.
delete x and $*x$ from ST.
L_Delete x and $*x$ from DSL.

The correctness and running time bounds for the above procedures follow from the previous lemmas.

Theorem 3.2 Let D be a dictionary of patterns over an alphabet Σ. Using the algorithm based on fail trees, we can search a text T in time $O(t(\log\sigma + \log d) + tocc \log d)$, where $tocc$ is the total number of patterns reported. We can insert or delete a pattern P in time $O(p(\log\sigma + \log d))$. Moreover, we require only $O(d)$ space to store the automaton.

3.5 Linear Time Searching

In this section, we show how to improve the search time. From Theorem 3.2 in the last section, a text T can be searched in $O(t(\log\sigma + \log d) + tocc \log d)$ time. The $\log\sigma$ factor comes from the computation of each *goto* and the $\log d$ factor from the computation of each $fail$. We show that it is possible to speed up the computation of $fail$ and achieve a faster searching algorithm. When σ is small and finite, or when $\sigma \ll d$, this is better than the searching algorithm of Section 3.4. It does, however, slow down the update times of the dictionary.

In Section 3.4, we used a-b trees to store every fail tree. In an a-b tree, the number of children or the *degree* of a nonroot internal node v, denoted by $\delta(v)$, must be in the range $[a, b]$, and must be in the range $[2, b]$ if v is a root. In this section, we use a variant of a-b trees, which we call *hybrid a-b trees*, to store the fail trees. In a hybrid a-b tree different nodes may have different ranges for the number of children permitted. The ranges depend on the number of leaves in the fail forest.

Our hybrid a-b trees depend on a parameter, which is a fixed integer independent of d. For any $k \geq 2$, the hybrid trees will allow us to perform a *findroot* in $O(k)$ time, and any update operation in $O(k\ n^{1/k})$ time, where n is the total number of leaves in the forest of fail trees. Recall that n is twice the number of states in the automaton.

Let $\alpha \geq 16$ be the smallest power of 2 such that $\alpha^k \leq n \leq (2\alpha)^k$. Each internal node is designated as *small* or *big*, but may change its designation during the algorithm. A small nonroot v has $\delta(v) \in [\alpha, 2\alpha]$, a small root v has $\delta(v) \in [2, 2\alpha]$, a big nonroot v has $\delta(v) \in [2\alpha, 4\alpha]$, and a big root v has $\delta(v) \in [2, 4\alpha]$. We maintain lists of small and big nodes using a separate link. Let *#small* and *#big* denote the number of such nodes. Our ranges imply that any nonroot has $\Theta(n^{1/k})$ children and there will be $O(k)$ levels in any hybrid tree.

The operations *insert, delete, concatenate, and split* can be implemented in a similar fashion to that used for regular a-b trees. Each operation visits or modifies at most b nodes at each level of the tree and may cause at most a constant number of nodes per level to violate the constraints on the number of children allowed.

In the rest of the section we show how to handle overflows and underflows in the number of children of a node of a hybrid tree. Define $excess = n - \alpha^k$, and $m = (\alpha^k + (2\alpha)^k)/2$. We maintain two invariants:

$$\#small + excess \leq (2\alpha)^k - \alpha^k \tag{3.1}$$

$$\#big \leq excess \tag{3.2}$$

Note that all nodes will be small when $n = \alpha^k$, and big when $n = (2\alpha)^k$. If n goes above $(2\alpha)^k$, we redefine all the big nodes as small nodes and start operating in the interval $(2\alpha)^k \leq n \leq (4\alpha)^k$ with small nonroot nodes having the range $[2\alpha, 4\alpha]$ and big nonroot nodes having the range $[4\alpha, 8\alpha]$. We redefine *excess* and m accordingly. We can do a similar thing when n falls below α^k. The following two lemmas simplify invariant maintenance.

> **Lemma 3.8** If there are n leaves in the fail forest, then the number of internal nodes of the fail trees is at most $3n/5$.

Proof The number of fail trees is equal to the number of normal regular prefixes. Since there is a complementary prefix for every regular prefix, the total number of prefixes n is at least twice the number of normal regular prefixes. This implies that the number of fail trees is at most $n/2$. Thus the number of root nodes is at most $n/2$.

Since $\alpha \geq 16$, the degree of any nonroot internal node in a fail tree must be at least 16. This means that the number of nonroot internal nodes can be at most $n/15$. Hence the total number of internal nodes of all fail trees is at most $n/2 + n/15$ which is less than $3n/5$. □

> **Lemma 3.9** In the middle of any operation, if $n \leq m$ then invariant 3.1 cannot be violated. Similarly if $n \geq m$ then invariant 3.2 cannot be violated.

Proof If $n \leq m$ then $excess \leq ((2\alpha)^k - \alpha^k)/2$. From Lemma 3.8 it follows that $\#small \leq 3n/5 \leq \frac{3}{5}\,\frac{\alpha^k + (2\alpha)^k}{2}$. Hence $\#small + excess \leq \frac{4}{5}(2\alpha)^k - \frac{\alpha^k}{5} \leq (2\alpha)^k - \alpha^k$; the last inequality holds provided $k \geq 2$. One can similarly verify that $\#big \leq excess$ whenever $n \geq 5\alpha^k/2$, which holds if $n \geq m$ and $k \geq 2$. □

Lemma 3.9 implies that when $n \leq m$ we only have to control $\#big$, as invariant 3.1 is always satisfied in this case, and the value of $\#small$ has no effect on invariant 3.2. Similarly when $n \geq m$, we only have to control $\#small$.

If $n \leq m$, we do any update operation in such way that $\#big$ never increases. Any update operation can increment or decrement the value n by at most one. If invariant 3.2 is satisfied before a *delete* but violated afterwards, we have to decrement

#big by only one to maintain invariant 3.2. Similarly, when $n \geq m$, we update in such a way that *#small* never increases. We may have to decrement *#small* by at most one after an *insert*. The operations *concatenate and split* do not change the value of n.

Two primitive operations needed for implementing hybrid trees are those that handle overflow and underflow of an internal node. Those are the cases when the degree of an internal node goes one above or below its declared range. We show how overflow and underflow can be eliminated without violating the invariants. We annotate each line of pseudocode by an ordered pair (i, j) implying that *#small* changes by i and *#big* by j after executing the line.

overflow(v):

Suppose we are controlling $\#small$.

  if v is small **then** redesignate it as a big node. $\{(-1, 1)\}$

  if v is big **then** break it into two big nodes. $\{(0, 1)\}$

Suppose we are controlling $\#big$.

(*) **if** v is small **then** break it into two small nodes. $\{(1, 0)\}$

(*) **if** v is big **then** break it into one big and one small node. $\{(1, 0)\}$

underflow(v):

Suppose v is a root node.

  if $degree(v) = 1$ **then**

    Remove v.

    Name its only child as the new root. $\{(-1, 0)$ or $(0, -1)\}$

Suppose v is a nonroot node with an immediate sibling w.

  Let $[a, b]$ be the range of w.

  if $\delta(w) \geq a + 1$ **then** transfer one child of w to v. $\{(0, 0)\}$

  else

    if v and w are both big(small) **then**

      Fuse them into one big(small) node. $\{(0, -1)$ or $(-1, 0)\}$

    if one of them is big and the other small **then**

      Fuse them into one big node. $\{(-1, 0)\}$

One can check that each change in *#small* or *#big* does not violate the invariants. Also *underflow(v)* never increases *#small* or *#big*. If an invariant is violated by 1 as a result of an *insert* or *delete*, applying *overflow* or *underflow* does not exacerbate the violation, and we restore the violated invariant as explained below.

A single overflow or underflow can be handled in $O(\alpha)$ time. Correcting the overflow(underflow) of a node may cause an overflow(underflow) of its parent. This is the way any update operation is implemented; we make changes to an internal node and correct any overflows or underflows that may propagate all the way to the root. Since there are $O(k)$ levels in any tree, we can implement any update operation in $O(k\ \alpha)$ time.

Finally we show how to decrement *#small* or *#big* by at least one in $O(k\ \alpha)$ time. These operations are necessary to restore the invariants that may be violated after an *insert* or *delete* operation. There is no problem with the invariants if the following operations cause *#small* or *#big* to go down by more than one.

decreasesmall(): Pick some small node v. If v is a root, redesignate it as a big node. If v is not a root, it must have an immediate sibling w. If w is small, then fuse v and w into a big node. If w is big, fuse v and w into one or two big nodes depending on $\delta(v)+\delta(w)$. This may propagate underflows to the ancestors of v which can be handled without increasing *#small* as we noted earlier that *#small* does not increase during an *underflow* operation.

decreasebig(): Pick some big node v. Split v into one or two small nodes depending on $\delta(v)$. Any propagated overflows can be handled without increasing *#big* as can be observed from the lines marked (*) of *overflow*.

> **Theorem 3.3** For a fixed integer $k \geq 2$, we can search a text T in $O(t(k + \log\sigma) + tocc\ k)$ time. Furthermore we can insert or delete a pattern P into a dictionary D in $O(p(k\ d^{1/k} + \log\sigma))$ time.

Proof For any state v, *fail(v)* can be obtained with a *findroot* operation. Since this takes only $O(k)$ time on a hybrid *a-b* tree we can search T in $O(t(k+\log\sigma)+tocc\ k)$ time. Any update operation will be accompanied by at most one *decreasesmall or decreasebig* operation. Each of these operations take $O(k\ \alpha)$ time. Since α is $O(d^{1/k})$, it follows that P can be inserted or deleted in the specified time. □

3.6 Building D in Linear Time

In this section, we show how to build the initial dictionary in $O(d\log\sigma)$ time. This is better than building the dictionary by repeated insertion of patterns which requires

$O(d \log d)$ (as in [9, 12]) or $O(d^{1+1/k})$ time depending on the data structure used. We can build the dictionary in linear time using either a parenthesis tree or regular a-b trees or hybrid a-b trees to store the $fail$ function.

Let $D = \{P_1, \ldots, P_s\}$ be the initial set of patterns given. Our first major goal is to sort all the (normal and extended, regular and complementary) prefixes of S in $<_{inv}$ order. For this purpose we partition S into two disjoint sets S_1 and S_2. S_1 contains those prefixes of S ending in a \$ and S_2 contains the rest of S.

We start by building a sorted list of the prefixes of S_2. These prefixes are exactly the prefixes of the patterns $P_1, \ldots, P_s$ and $*P_1, \ldots, *P_s$ with the \$ stripped from their right ends. We build a suffix tree for the *reverses* of these $2s$ prefixes using the suffix tree construction of [57] as modified by [12]. There will be a one-to-one correspondence between the leaves of the suffix tree and the prefixes of S_2 as proved in Lemma 2 of Section 2 in [12]. If we sort the children of every internal node of the suffix tree by the labels of the edges connecting the children and the parent, then the left-to-right order of the leaves of the suffix tree is the $<_{inv}$ order of the prefixes of S_2. We can build the suffix tree in $O(d \log \sigma)$ time. Rearranging the children and scanning the leaves in the left-to-right order takes only $O(d \log \sigma)$ time.

The prefixes of in S_1 (and S_2) occur consecutively in S (this follows from the assumptions made on \$ in Section 3.2). Furthermore the prefixes of S_1 are in one-to-one correspondence with the prefixes of S_2, and occur in exactly the same $<_{inv}$ order as the corresponding prefixes of S_2. Thus we can build a sorted list of the prefixes of S_1 by scanning the list of the prefixes of S_2, and for each prefix x of S_2 inserting (in the same relative order) the prefix $x\$$ into S_1. Finally we concatenate both S_2 and S_1 to obtain S in the $<_{inv}$ order. This takes $O(d)$ time.

In our dictionary structure, we used both a balanced search tree and a DS list on the sorted list of prefixes. Both of these can be built in $O(d)$ time (see the discussion on building a-b trees below).

Our second major goal is to compute the *fail* function. Specifically, to each prefix w, we want to associate those prefixes x, such that $fail(x) = w$. We temporarily keep these prefixes associated with w in a sorted list in $<_{inv}$ order, which we call w's *fail list*. The prefixes on the fail list will become the leaves of the fail tree T_w. By Lemma 3.5, these are precisely the prefixes x such that $w <_{inv} x <_{inv} *w$, and there is no y such that $w <_{inv} y <_{inv} x <_{inv} *y <_{inv} *w$. Intuitively, if we think of each regular-complementary pair of a normal prefix as a pair of matching parentheses, we want to find the deepest pair of parentheses containing x; if w is the left parenthesis in that pair, then $fail(x) = w$. The natural way to do this is to scan the list S keeping track of the unmatched normal prefixes on a stack STK. We use the following scanning rules in the order below:

if $x \neq \epsilon$ is regular (i.e. $fail(x) = top(STK)$), **then**
    Append x to the fail list for $top(STK)$.
if x is normal and regular **then** $push(x)$ onto STK.
if x is normal and complementary **then** $pop(STK)$.
if $x \neq \$$ is complementary (i.e. $fail(x) = top(STK)$), **then**
    Append x to the fail list for $top(STK)$.

These rules ensure that $fail(x) \neq x$ and $fail(x) = fail(*x)$ for every regular prefix x. Scanning the list takes constant time per prefix, so this step takes $O(d)$ time.

Finally we show how we can organize each fail list into a fail tree. For each w, we have computed the leaves of T_w in sorted order as w's fail list. To build T_w we build a search tree with the elements of the fail list as leaves. We can build any standard a-b tree given the sorted list of leaves in linear time [58]. To build hybrid a-b trees, let n be the total number of leaves in all trees. Given $k \geq 2$, choose α as described in Section 3.5 and define the value m. Build the trees with all small nodes if $n \leq m$, and with all big nodes if $n > m$. The correctness of this construction follows from Lemma 3.9. This also takes only $O(d)$ time.

We can summarize our linear time construction algorithm as follows:

Algorithm 6 Code for preprocessing a dictionary

BUILD($D = \{P_1, \ldots, P_s\}$)
Build suffix tree on the *reverses* of $P_1, \ldots, P_s, *P_1, \ldots, *P_s$ to order S_2.
Scan S_2 and build S_1.
Concatenate S_2 and S_1 to obtain S.
Build ST (or PT) and DSL for the sorted list of prefixes.
Scan the sorted list and compute the fail list of each prefix.
Convert each fail list into a (regular or hybrid) a-b tree.

The entire process takes linear time as every step has been shown to take $O(d \log \sigma)$ time. We can summarize:

Theorem 3.4 The initial dictionary D can be constructed in $O(d \log \sigma)$ time.

Chapter 4

Multiple Matching with Strings: Implementation

One of the secondary themes of this thesis is to demonstrate the simplicity and correctness of our solutions to multiple pattern matching problems by implementing one of them in a real language. Though the main objective of this thesis is to study and develop a theoretical framework for dynamic multiple pattern matching problems, it is our belief that some of our ideas are also practical. In this regard, we have implemented the dynamic dictionary matching algorithm presented in Section 3.3. In this chapter, we present the details of our implementation.

4.1 Overview

Recall the algorithm presented in Section 3.3. To implement this algorithm, we need two data structures: (1) The Dietz-Sleator list, and (2) *a*-*b* trees. To the best of our knowledge, there is no previous implementation of the Dietz-Sleator list [62]. We have not found any implementation of *a*-*b* trees suitable for our needs.

Our implementation has two phases. In the first phase, we implement the DS list as a *separate* package. As mentioned in Section 2.4, the Dietz-Sleator list has applications to several other problems. By making it a separate package, independent of the dictionary matching, we expect researchers working on these problems to use and benefit from our package. In the second phase, we implement the dynamic dictionary matching algorithm, along with a customized version of *a*-*b* trees. In this phase, we use the operations of the DS list implementation in the first phase. All programs are written in ANSI C.

We now describe the details of both the phases in two different sections.

4.2 The Dietz-Sleator List

We implement the amortized version of the DS list described in Section 2.4. As mentioned earlier, we implement it as a separate package which can be used in any application. We first describe the interface to the package. We then describe the overall organization of the program, followed by specific implementation details.

Interface The interface is very simple. It is assumed that the DS list is built on the top of some other existing data structure for an application written in ANSI C. Let `node` denote the structure representing an individual node or element in the application. For example, a state or a prefix represents an individual element in the dictionary matching problem. In order to build a DS list on the top of these nodes, the `node` structure should be augmented with an extra field of type `ds_element` as shown below.

```
struct node {
  /* Other fields of the structure go here */
  ds_element ⟨element⟩;
};
```

where ⟨*element*⟩ is a field name of user's choice. The reason for this kind of interface is that the DS list implementation is oblivious to any application using it. It merely creates elements of type `ds_element` as specified, and it is the responsibility of the application to make its `node`s point to the specific elements in the DS list. That means, to insert a node `node2` after another node `node1`, an application should insert `node2.`⟨*element*⟩ after `node1.`⟨*element*⟩ in the DS list.

Prior to any list operation, a DS list must be created using the statements:

```
ds_list ⟨list⟩;
⟨list⟩ = ds_create();
```

The operation `ds_create()` returns a pointer to a newly created DS list, initially containing a dummy element of type `ds_element`. The dummy element can be accessed using the routine

```
ds_dummy(⟨list⟩).
```

Once a list is created, a node `node2` can be inserted into the list *after* an existing node `node1` using

```
node2.⟨element⟩ = ds_insert(node1.⟨element⟩);
```

The routine `ds_insert()` creates a new element after `node1.`⟨*element*⟩, and returns a pointer to it. By assigning this pointer to `node2.`⟨*element*⟩, we effectively insert `node2` after `node1` into the list. If we want to insert a node `node1` as the very first node in the list, we can do so by using `node1.`$\langle element \rangle$ `= ds_insert(ds_dummy(`$\langle list \rangle$`));`

which creates a new element after the dummy element. Since the dummy element is always the first element that does not count, we have effectively made `node1` as the first node in the list.

An existing element can be deleted from its DS list by using

`node.`$\langle element \rangle$ `= ds_delete(node.`$\langle element \rangle$`);`

which deletes the element from the DS list, and returns `NULL`. Since the old value of `node.`$\langle element \rangle$ is no longer valid we must set it to `NULL` as shown above.

Two *distinct* nodes `node1` and `node2` can be compared using

`ds_order(node1.`$\langle element \rangle$`, node2.`$\langle element \rangle$`)`

which returns `TRUE` if `node1` comes before `node2` in the DS list and `FALSE` otherwise. Finally, a DS list can be destroyed using

`ds_destroy(`$\langle list \rangle$`);`

Organization The DS list is implemented in two layers. At the first layer, it is organized as a list of sublists, and at the second layer, each sublist contains the individual elements. This is the same scheme described in Section 2.4. As a result of this, we define three auxiliary routines `ds_subinsert()`, `ds_subdelete()`, and `ds_suborder()`, which are the corresponding operations on sublists.

Let n be the number of elements, and m be the number of sublists in a DS list at any time. For the implementation, we make certain changes to the algorithm as suggested in the full paper of [32].

First, we always assign the label 0 to the dummy sublist in the list, which is always the first sublist of the list. This obviates the need to subtract the label of the base while comparing labels.

Second, we change the relabeling scheme so that there is always at least one unused label between the labels of two consecutive sublists in the list, or equivalently, we insert a new record into the first gap and then relabel.

Third, we keep the arena size M such that $M > 2m^2$ instead of $M > m^2$. This will enable us to avoid modular arithmetic while assigning labels. For example, consider an operation $y =$ `ds_subinsert(`x`)`, which inserts sublist y after sublist x. When renumbering, we first determine if $v(x) < M/2$. If so, we use the algorithm as it is. Otherwise, we replace x with y and relabel in the opposite direction, starting at y. This way, we never wrap around, and therefore, no modular arithmetic will be necessary.

The elements of a sublist are given labels in the range $[0, maxint]$, where $maxint$ is the greatest unsigned integer that fits in a full word of a computer (typically $2^{32}-1$ on most computers).

Suppose we perform the operation $y =$ `ds_insert(`x`)`. Let L_i be the sublist containing x. We insert y after x into the sublist L_i. If x is previously the rightmost element, we make the average of the label of x and $maxint$ as the label of y. Otherwise, if there is an unused label between the labels of x and its previous successor, we assign it to y. Otherwise, we relabel all the elements of L_i. Let $minlist$ be a predefined constant. If L_i contains more than $2\max(minlist, \log n)$ elements, we split L_i into many sublists such that each newly created sublist (except the last one) contains $\max(minlist, \log n)$ elements. Also, elements in each new sublist will be relabeled.

To perform the operation $x =$ `ds_delete(`x`)`, we first delete x from its sublist L_i. We then check if L_i contains more than $2\max(minlist, \log n)$ elements. If so, we split L_i into sublists exactly as described above.

There are two differences between our strategy described above, and the strategy of [32]. First, we introduce $minlist$ as a practical means to avoid frequent splitting of sublists. We found empirically that the performance of a DS list gets better with increasing values of $minlist$ up to a certain value (around 10). Second, we use a somewhat different relabeling strategy for the elements of a sublist. We set the gap between the labels of consecutive elements to at least $\max(2^{minlist}, n)$ rather than $|L_i|$ (length of the sublist) as suggested in [32]. These changes do not affect the running time of any operation as explained below.

When we split a sublist in `ds_insert()` or `ds_delete()`, we call `ds_subinsert()` to insert the newly created sublists. Each call to `ds_subinsert()` takes $O(\log n)$ amortized time as noted in Section 2.4. However, there are only $O(n/\log n)$ such calls because each sublist contains at least $\log n$ elements. Therefore, the total time taken by all calls to `ds_subinsert()` is $O(n)$, which contributes an $O(1)$ amortized time per each call to `ds_insert()` or `ds_delete()`. It only remains to show that the time taken by splitting or relabeling a sublist takes only $O(1)$ amortized time per an insertion or a deletion.

We use the following potential function which is a slight modification of the one given in the full paper of [32].

$$\Phi = c\sum_{i=1}^{m} \max(0, |L_i| - \max(minlist, \log n)) + \max(0, |L_i| - \log g_i),$$

where $|L_i|$ is the number of elements in i^{th} sublist L_i, g_i is the size of minimum gap between the labels of any consecutive elements in L_i, and c is a suitably chosen constant. As one can see, there are two major terms in the potential function Φ. The left term controls the splits whereas the right term controls the relabelings.

Consider a call to `ds_insert()`. If it does not cause a split or a relabeling, then it takes constant worst-case time. The left term in Φ cannot go up by more than one, and the right term cannot go up by more than two. Thus the increase in Φ is at most 3c. Therefore, the amortized time per insertion is $O(1)$. If the call causes a split, then we spend $O(|L_i|)$ worst-case time. In this case, $|L_i| > 2\max(minlist, \log n)$. After this operation, the left term goes down by $c(|L_i| - \max(minlist, \log n))$. The right term can only go down (but not up). If we set c to at least 2, then the decrease in Φ will be at least $|L_i|$ which will give us $O(1)$ amortized time per insertion. If a call causes a relabeling, then we spend $O(|L_i|)$ worst-case time. In this case, $|L_i| \leq 2\max(minlist, \log n)$. After this operation, the right term goes down by $c\max(minlist, \log n)$ if $|L_i| \geq \max(minlist, \log n)$, or by $c|L_i|$ if $|L_i| < \max(minlist, \log n)$. The left term is unaffected. Again setting c to at least 2 gives us $O(1)$ amortized time per operation.

A similar analysis can be used to show that each call to `ds_delete()` takes $O(1)$ amortized time.

Implementation Details The program is contained in two files: one is the header file `ds_list.h`, and the other is the source file `ds_list.c`. The header file contains all the important data structures, and all constant and type definitions. The header file must be included in any application using the DS list package. The source file can be compiled and archived into a library, which can be loaded at the execution time.

Before we describe the data structures, we need to explain how we store the labels of sublists and elements in a DS list. As explained above, the label of an element in a sublist is in the range $[0, maxint]$. Therefore, we declare the label of an element to be of type `unsigned int`. We need a more complex data type for storing the label of a sublist. Recall that we need to maintain the constraint $M > 2m^2$, on the arena size and the number of sublists. This means that, if we declare the label to be of type `unsigned int`, then on most computers, $M = 2^{32}$, and $m < 2^{15}$. In other words, we can only support at most 32K sublists in a list, which is a very small number for many applications. In order to support a larger arena for labels, we use the GNU Multiple Precision Arithmetic Library [45], called GNU-MP. This package is written in ANSI C, and supports arbitrary multiple precision arithmetic operations and therefore provides us with a very large arena for labels. With the aid of this package, we declare the label of a sublist to be of type `MP_INT`. We also replace each basic arithmetic operation on labels with a procedure call in the GNU-MP package.

There are two constants `DS_MINLIST` and `DS_MAXINT` that correspond to *minlist* and *maxint* respectively, which can be tuned to specific needs. There are three data structures `ds_list`, `ds_sublist`, and `ds_element`. The `ds_list` structure contains the following fields.

```
struct {
  MP_INT elements;
  MP_INT sublists;
  MP_INT maxlabel;
  ds_sublist first;
};
```

where `elements` gives the number of elements n in the list, `sublists` gives the number of sublists m in the list, and `maxlabel` gives the current arena size M. `first` points to the first sublist in the list.

The `ds_sublist` structure has the following fields.

```
struct {
  unsigned elements;
  MP_INT label;
  ds_list list;
  ds_sublist prev, next;
  ds_element first;
};
```

where `elements` gives the number of elements in the sublist, `label` is the label associated with the sublist, `list` points to the parent list containing the sublist, `prev` and `next` point to the previous and next siblings respectively, and `first` points to the first element in the sublist.

The `ds_element` structure has the following fields.

```
struct {
  unsigned label;
  ds_sublist sublist;
  ds_element prev, next;
};
```

where `label` is the label associated with the element, `sublist` points to the parent sublist containing the element, and `prev` and `next` point to the previous and next siblings respectively.

Finally we give a brief description of the procedures. `ds_create()` creates a new structure for a DS list, creates a dummy first sublist, and a dummy first element. Finally, it returns a pointer to this new structure. `ds_destroy()` takes as input a pointer to a DS list, returns all the used space, and destroys the entire list. The three routines, `ds_suborder()`, `ds_subdelete()`, and `ds_subinsert()` perform the intended operations on sublists, while the three routines, `ds_order()`, `ds_delete()`, and `ds_insert()` perform the operations on elements. Finally, the auxiliary routine `ds_aux_rearrange()`, which is called by `ds_delete()` and `ds_insert()`, splits a sublist into many sublists of appropriate size as described earlier.

4.3 Dynamic Dictionary Matching

We implement the dynamic dictionary matching algorithm presented in Section 3.3, which is based on the parenthesis tree. As part of the implementation, we implement a customized version of *a*-*b* trees. We also make use of the DS list implementation presented in the last section. We first describe the interface to the program. We then describe the overall organization of the program, followed by specific implementation details.

Interface The program is written as a separate package which can be included in any application. In fact, we have built a very simple interactive dynamic dictionary matching system using the interface that we are going to describe now. We include a sample session of this interactive system in Appendix A.

The program accepts any null-terminated ASCII string as a pattern or a text. We set a constant `MAXCHAR` to 128, which is the size of ASCII alphabet. We can change the value of `MAXCHAR` to handle other types of strings as well. For convenience, we assume that the alphabet is $\{0, 1, \ldots, \texttt{MAXCHAR} - 1\}$. We *need not* (and should not) terminate each pattern with a special symbol \$, as assumed in Chapter 3. In Chapter 3, we have made that assumption to facilitate the explanation of how we recognize matched patterns at each location in a text. Later, we will explain how we recognize patterns without making that assumption. The maximum number of patterns in a dictionary at any time is bounded by a constant MAXPATTERN, which can be changed to specific needs.

A dictionary can be created using

```
dictionary ⟨dict⟩;
⟨dict⟩ = is_create();
```

The routine `is_create()` creates a dictionary containing a dummy pattern ϵ, and returns a pointer to the dictionary. Once a dictionary is created, we can insert patterns into it using

```
pattern ⟨dict⟩;
is_insert(⟨dict⟩, ⟨pat⟩);
```

where $\langle pat \rangle$ is a pattern to be inserted, which is basically a pointer to a string representing the pattern. Similarly, a pattern $\langle pat \rangle$ can be deleted from a dictionary using

```
is_delete(⟨dict⟩, ⟨pat⟩);
```

Finally, we can search a text using

```
text ⟨text⟩;
is_search(⟨dict⟩, ⟨text⟩);
```

which prints all the matched (non-null) patterns at each location in $\langle text \rangle$, which is a pointer to a string representing the text. We can easily change this routine to report only the longest matched patterns instead.

Organization We implement the dynamic dictionary matching algorithm presented in Section 3.3, which makes use of the parenthesis tree. We make two changes to the algorithm as described below.

First, we organize the *goto* tree in a different way for practical considerations. In Chapter 3, we assumed that the outgoing edges of a node in the *goto* tree are organized into a balanced search tree. For the implementation, however, we take a practical approach. We partition the set of nodes of the *goto* tree into two classes, depending on the way the outgoing edges of a node are accessed. We define a constant GOTOTHRESHOLD $\leq$ MAXCHAR, which controls the type of access. In *direct access*, we store the outgoing edges of a node x in an array of size MAXCHAR. To compute $goto(x, i)$, we simply take the i^{th} entry of the array. In *indirect access*, we store the outgoing edges of a node x in an array of size GOTOTHRESHOLD. To compute $goto(x, i)$, we scan the entries of the array until we find a transition with the label i. The access of the start state ϵ is always direct as it is the most frequently used state. For other states, the access is indirect until their degree (that is, the number of outgoing edges) exceeds `GOTOTHRESHOLD`, after which their access becomes direct forever. We take this strategy under the assumption that any state, whose degree exceeds GOTOTHRESHOLD once, is frequently used and merits direct access. The value `GOTOTHRESHOLD` can be tuned to individual needs.

Second, we remove the assumption made in Chapter 3 that each pattern string ends with a special symbol \$. By removing this assumption, we obviate the need for extended prefixes whose sole purpose is to facilitate finding matched patterns. This cuts down the total number of prefixes in the automaton by half. Therefore, the total space requirement is also cut down by half. However, we need another way to find matched patterns.

In Section 3.3, we use the parenthesis tree to compute the $fail$ function. We find a novel way to find matched patterns, by extending the parenthesis tree in an interesting way. We start with a definition.

A *bracket tree* is a parenthesis tree, in which some of the parenthesis pairs are *also* treated as *brackets*, denoted by '[' and ']'. They behave like ordinary parentheses for the operation *find nearest enclosing parenthesis*. In addition, we define a new operation on a bracket tree. The operation *find nearest enclosing bracket* finds the nearest enclosing bracket, ignoring any other nearer enclosing ordinary parentheses. In other words, a bracket is also treated as a parenthesis, but a parenthesis is not treated as a bracket.

We define *bracket mapping* as follows: It is the same as parenthesis mapping defined in Section 3.3, except that it maps a regular prefix which is also a *full* pattern to a '[', and its complement to a ']'. Similarly, we use $paren(x)$ to denote the corresponding parenthesis or bracket to which x is mapped.

> **Example 4.1** Suppose $D = \{\epsilon, b, aab\}$ is a dictionary. The prefixes of S in the inverted order are:
>
> $\epsilon, a, aa, *aa, *a, b, aab, *aab, *b, *.$
>
> The bracket mapping of S yields:
>
> $[, (, (,),), [, [,],],].$

We build a bracket tree on the normal prefixes of a dictionary, in the place of the parenthesis tree on the normal and extended prefixes as done in Section 3.3. Suppose we are in a state x at some location in a text. Computing $fail(x)$ is the same as finding the nearest enclosing parenthesis of $paren(x)$ as before. We can report all matched patterns at the current location of the text as follows. Let P be the longest pattern matched at the location. We first check whether x itself corresponds to a full pattern. If so, $P = x$ is the longest pattern matched at the current location. Otherwise, we find the nearest enclosing *bracket* of $paren(x)$, and take the corresponding prefix (actually a pattern), which is P. Since ϵ is always considered a pattern in the dictionary, such a P always exists. To find any other shorter patterns matched at the location, we start with P and keep finding the nearest enclosing brackets all the way to the end.

We also implement a customized version of *a*-*b* trees to build the bracket tree. The bracket tree initially consists of a pair of brackets corresponding to the null string ϵ and its complement $*$. The leaves of the tree in the left-to-right order correspond to the (normal) prefixes of the dictionary in the inverted order. The tree is actually an `A-(B-1)` tree, where `A` and `B` are predefined constants. It is possible to choose different values for `A` and `B` provided `A` ≥ 2, and `B` is an even number such that `B` $\geq$ `2A`. We define several auxiliary routines on these trees. `ab_insert()` inserts a prefix into a bracket tree at the right place. It uses `ab_findpred()` to find the predecessor of the prefix in the tree, and `ab_insertafter()` to do the actual insertion of the prefix after its predecessor. The routine `ab_delete()` is used to delete a prefix from a bracket tree. It uses another routine `ab_auxdelete()` for the purpose. These routines call another routine `ab_addbalance()` to update the BALANCE information.

Two routines `ab_go_up()` and `ab_go_down()` are used to find the nearest enclosing parenthesis, while `ab_match_up()` and `ab_match_down()` are used to find the nearest enclosing bracket. Finally, a special routine `ab_makepattern()` is used to deal with the case where an existing prefix becomes a pattern, while another special routine `ab_unmakepattern()` deals with the converse case.

Implementation Details The program is contained in two files: one is the header file `is_dict.h`, and the other is the source file `is_dict.c`. The header file contains all the major data structures, and other constant and type definitions. The header file also includes the header file for the DS list implementation.

The structure `dictionary` contains the following fields.

```
struct {
  state start;
  state root;
  ds_list list;
```

```
    pattern pattern[MAXPATTERN];
    short numpat;
  };
```

where `start` points to the starting state ϵ of the automaton, `root` points to the root of the bracket tree, and `list` points to the associated DS list. `pattern[]` stores the character representations of patterns in the dictionary, and `numpat` gives the number of patterns in the dictionary.

The most fundamental data structure is `state`, which is defined below.

```
  struct {
    stateinfo stateinfo;
    failtree failtree;
    gototree gototree;
  };
```

The information contained in a `state` depends on whether it corresponds to a leaf or an internal node of the bracket tree. In the case of a leaf, the information again depends on whether it corresponds to a regular prefix or a complementary prefix.

The first field `stateinfo` is defined only for leaves of the bracket tree; that is, only for prefixes. It has two subfields, as defined below.

```
  struct {
    state complement;
    ds_element element;
  };
```

The first subfield `complement` points to the complement of the prefix. The second subfield `element` is of type `ds_element` which is used for inserting the prefix into the associated DS list.

The second field of `state` is `failtree` which stores the information about the bracket tree. It is defined for all nodes of the bracket tree. The subfields of `failtree` are:

```
  struct {
    state parent;
    ptroffspring offspring;
    short arity;
    struct balance {
      short open;
      short close;
```

```
    } prefix, pattern;
};
```

The subfield `parent` points to the parent node in the bracket tree. `offspring` is a pointer to an array of pointers to the children nodes, also of type `state`. It is defined only for internal nodes. `arity` gives the number of children of an internal node. The structure `balance` stores the BALANCE information defined in Section 2.5. BALANCE has two components, `prefix` and `pattern`, both of which are defined for all internal nodes. In the case of leaves, `prefix` is defined for both parentheses and brackets, but `pattern` is defined only for brackets. Recall that a bracket is treated as a parenthesis, but not vice versa.

Finally, the third field `gototree` stores the information about the *goto* tree. It is defined for only regular prefixes. It has the following structure.

```
struct {
  state parent;
  state *child;
  unsigned short patternnumber;
  unsigned short refcount;
  unsigned char label;
  unsigned char arity;
  unsigned char childarraysize;
};
```

The subfield `parent` points to the parent of the state or prefix in the *goto* tree. `label` is the label associated with the edge from the parent to the state. For example, if the state is xa then `parent` points to x, and a is its label. `child` points to an array containing the children of a state in the *goto* tree. The size of the array is contained in `childarraysize`, whose value is either `MAXCHAR` or `GOTOTHRESHOLD` depending on the access type, as discussed earlier. `arity` gives the actual number of children, and `refcount` gives the number of patterns containing the prefix. `patternnumber` gives the pattern associated with this state (if any), which is an index to the `pattern[]` array of the dictionary.

We have tested the code with several examples and found that it works correctly. The code is available as a package that can be used in any application. We hope that other researchers will use it and find it useful.

Chapter 5

Multiple Matching with Square Patterns

In this chapter, we solve the two dimensional dynamic dictionary matching problem with square patterns. We are given a dictionary of square patterns $D = \{P_1, P_2, \ldots, P_n\}$, that can change over time. The basic matching operation is to search a rectangular text $T[1 \ldots n_r, 1 \ldots n_c]$ and report all occurrences of patterns in the text. The dictionary can be changed by inserting or deleting individual patterns.

We present an algorithm which is based on the one dimensional algorithms presented in Section 3.3 and Section 3.4. We arbitrarily choose any of these algorithms, and refer to it as the IS (Idury-Schäffer) algorithm from now on. The time bounds for our algorithm are:

Dictionary Preprocessing: $O(d \log d)$, where d is the size of the dictionary.

Dictionary Update: $O(p \log d)$, where p is the area of the square pattern to insert or delete.

Text Searching: $O((t + tocc) \log d)$, where t is the area of the text, and $tocc$ is the total number of pattern occurrences reported.

In [11], Amir and Farach presented a (static) multiple matching algorithm with similar preprocessing and searching complexities of:

Dictionary Preprocessing: $O(d \log n)$, where n is the number of patterns.

Text Searching: $O((t + tocc) \log n)$, where t is the area of the text.

Giancarlo [42] has recently developed a data structure called the *L-suffix tree* that generalizes the suffix tree to two dimensional strings. Two related application of the L-suffix tree are: a different method to solve the two dimensional static problem with the same time bounds as in [11], and a solution to the two dimensional dynamic problem with slightly worse time bounds than are presented here.

The main ideas behind our algorithm are: a linearization of two dimensional square matrices along the main diagonal to produce a one dimensional dictionary of strings over an alphabet of subrow/subcolumn pairs and a new method to manipulate efficiently the failure links over the non-standard subrow/subcolumn alphabet.

The rest of the chapter is organized as follows. In Section 5.1, we make certain modifications and augmentations to the IS algorithm presented in Chapter 3. We use these modifications in presenting our algorithm for the two dimensional dictionary problem in this chapter. We present an overview of the two dimensional algorithm in Section 5.2. We describe a search automaton for square patterns from a conceptual point of view. We show the similarities and differences of this automaton from its one dimensional counterparts. We follow this by describing the details of the searching algorithm in Section 5.3. Finally, we describe the details of updating algorithms in Section 5.4.

5.1 Modifications to the IS Algorithm

In this chapter, we use the IS algorithm for one dimensional dynamic dictionary matching as a backbone for our two dimensional algorithm. Recall that the IS algorithm supports three main operations INSERT, DELETE, and SEARCH. In this section, we define two additional operations to the IS algorithm, which we need for the two dimensional algorithm. In addition, we show how the IS algorithm can be used to solve certain prefix/suffix queries, which are needed in this chapter.

The INSERT operation inserts a new pattern, by first finding the longest common prefix with some other pattern, and then by inserting the remaining characters, *one character at a time*. Specifically, when we insert a new pattern P into D, we insert its prefixes into S, in increasing order of lengths. Thus when we insert a string x into S, we have already inserted all the prefixes of x. For each prefix $x \in S$, the algorithm keeps a reference count of how many patterns have x as a prefix. This means that if we are inserting a new prefix xa where x is already a prefix in the dictionary, we first get to x and then add a new state (or prefix) from there.

Similarly the DELETE operation deletes an existing pattern, by deleting characters in the reverse direction, *one character at a time*, until we reach the longest common prefix with some other pattern. That is, when we delete a pattern P from D, we delete its prefixes in decreasing order of lengths. Those prefixes of P that are not prefixes of any other pattern are deleted. For those prefixes that are prefixes of another pattern, the IS deletion algorithm simply decrements the reference count.

As a consequence, we can define the following new operations on the dictionary, which we use later in this chapter. The IS algorithm supports these operations in $O(\log d)$ time each.

EXTEND(x, a): Add a prefix xa into the dictionary starting with a pointer to x. Return the pointer to xa.

TRUNCATE(xa): Delete the prefix xa from the dictionary starting with a pointer to xa. Return the pointer to x.

A key component in the static two dimensional dictionary algorithm [11] is a subroutine that answers queries of the form: "Is x a prefix of y?", where x and y are suffixes of dictionary patterns in constant time. In [11] the prefix query is converted into a least common ancestor query on a suffix tree; least common ancestor queries on fixed trees can be answered quickly with enough preprocessing. One of the obstacles to making the algorithm of [11] dynamic is that it is not known how to efficiently answer the least common ancestor queries in a dynamic setting.

Therefore, we find a different way to handle the prefix queries in a dynamic setting. Recall that the IS algorithm builds a DS list on the top of all the prefixes of a dictionary. As a consequence of that, we can use the IS algorithm to solve the above problem. First, we convert the prefix queries on pattern suffixes into suffix queries on pattern prefixes. To answer a suffix query, we can use the following lemma (closely related to Lemma 3.1):

Lemma 5.1 x is a suffix of y if and only if $x <_{inv} y <_{inv} *y <_{inv} *x$.

The middle $<_{inv}$ ordering always holds and the outer two can be tested in constant time using the *L_Order* operation. This works even when the lists are dynamically changing due to insertions and deletions.

5.2 Overview of the Algorithm

We describe an automaton for recognizing square patterns in a rectangular text. This automaton is very similar to the AC and IS automata for strings. Consequently, the algorithms for searching a text, and inserting and deleting a pattern from the dictionary will be very similar too. We also extend the notion of $goto$ and $fail$ functions to square patterns as done in [11]. Throughout this chapter we use M to denote our automaton for square patterns.

Each square pattern is conceptually divided into an upper triangular (henceforth u-t) half and a lower triangular (henceforth l-t) half. For searching purposes, each square pattern is conceptually divided into ⌟-shaped linear strips centered around characters on the main diagonal. Each ⌟-shaped strip consists of a column c of the u-t half and a row r of the l-t half, both aligning at the main diagonal. Each column (row) is read from the top (left) edge of the pattern towards the diagonal. Consider, for example, the following 3×3 pattern along with its u-t and l-t halves:

Example 5.1

$$\begin{array}{ccc} a & b & c \\ c & a & a \\ d & b & b \end{array} \qquad \begin{array}{ccc} a & b & c \\ & a & a \\ & & b \end{array} \qquad \begin{array}{ccc} a & & \\ c & a & \\ d & b & b \end{array}$$

In this pattern, the columns in the u-t half are the one dimensional strings a, ba, cab, and the corresponding rows in the l-t half are a, ca, dbb. The three ⌟-shaped strips, each represented as an ordered pair $\langle c, r \rangle$, are $\langle a, a \rangle$, $\langle ba, ca \rangle$ and $\langle cab, dbb \rangle$.

We define the *prefix* and *suffix* of a square pattern. For convenience, we first define the prefix and suffix of upper and lower triangular patterns. Then we extend the definitions to square patterns as well.

Let $U[1 \leq i \leq j \leq m]$ be an upper triangular pattern. Then for any $1 \leq l \leq m$, $U[1 \leq i \leq j \leq l]$ is a prefix of U, and $U[l \leq i \leq j \leq m]$ is a suffix of U. The empty upper triangular pattern ϵ is both a prefix and suffix of every upper triangular pattern. We can similarly define the prefix and suffix of a lower triangular pattern.

Let $P[1 \ldots m,\ 1 \ldots m]$ be a square pattern. Then for any $1 \leq i \leq m$, $P[1 \ldots i,\ 1 \ldots i]$ is a prefix of the pattern P. Similarly, for any $1 \leq i \leq m$, $P[i \ldots m,\ i \ldots m]$ is a suffix of P. The empty square pattern ϵ is both a prefix and suffix of every square pattern.

The above definitions enable us to linearize square patterns into ⌟-shaped strips. With this linearization, it is possible to view each square pattern as a *one dimensional* string where each character corresponds to a ⌟-strip. A new pattern is inserted into the dictionary one ⌟-shaped strip at a time, starting with the shortest strip. Similarly, patterns are deleted from the dictionary one strip at a time, starting with the longest strip. Consequently, a text is searched diagonally along all its diagonals.

In the example above we insert the pattern as three strips in the order $\langle a, a\rangle$, $\langle ba, ca\rangle$, $\langle cab, dbb\rangle$. Similarly, we delete the pattern as the same three strips, but in the reverse order.

The automaton M for the set of square patterns in D consists of states, where each state corresponds to a prefix of some square pattern in D. This is exactly like in the AC automaton except that we are dealing with the prefixes of a square pattern. As before, we will use the prefix to mean its state, and vice versa, as long as there is no ambiguity. We define *goto* for M as follows:

Let $Q[1 \ldots i, 1 \ldots i]$ and $R[1 \ldots i+1, 1 \ldots i+1]$ be two arbitrary prefixes in M, such that Q is a prefix of R. Then $goto(Q, \langle c, r\rangle) = R$, where $\langle c, r\rangle$ is the $(i+1)^{\mathrm{st}}$ strip of R. If there is no such R, then $goto(Q, \langle c, r\rangle)$ is undefined.

For any prefix R in M, $fail(R) = Q$, where Q is the largest prefix in M which is a proper suffix of R. If there is no such Q then $fail(R) = \epsilon$. In other words, the $fail$ of a square pattern prefix R is the largest square pattern prefix Q smaller than R, that aligns with the lower right hand corner of R.

We can extend the notion of *goto* and $fail$ to upper and lower triangular patterns in a similar way. In order to avoid confusion with the *goto* and $fail$ of a one dimensional automaton for strings, we sometimes use $goto^{\Box}$ and $fail^{\Box}$ ($\Box$ for square) to denote the transitions of M. We similarly use $goto^{u}$ and $fail^{u}$ (u for upper) for an automaton for upper triangular patterns.

As in the AC and IS automata, the *goto* and $fail$ functions can be represented by directed rooted trees in the automaton M. The *goto* tree is a labeled tree where each edge is labeled with a $\lrcorner$-strip, which is an ordered pair of a (sub)column and a (sub)row, denoted by $\alpha = \langle c, r\rangle$. Our searching algorithm is similar to the static dictionary matching searching algorithm of [11] and has the same structure as the AC and IS algorithms (the pseudocode for our searching algorithm is shown at the end of next section). The main difference, however, is in the details of how we search for a suitable transition to take in the *goto* tree, which is explained below.

We first need to define a special operator $\leq_s$ as done in [11]. $\leq_s$ is defined by:

$$x_1 \leq_s x_2 \qquad \text{if } x_1 \text{ is a suffix of } x_2.$$

$$\langle c_1, r_1\rangle \leq_s \langle c_2, r_2\rangle \qquad \text{if } c_1 \leq_s c_2 \text{ and } r_1 \leq_s r_2.$$

We extend the definition of $<_{inv}$ as follows:

$$\langle c_1, r_1 \rangle <_{inv} \langle c_2, r_2 \rangle \qquad \text{if } (c_1 <_{inv} c_2) \text{ or } (c_1 \leq_s c_2 \text{ and } r_1 <_{inv} r_2).$$

Let s be a state in M. We use $label(s)$ to denote the set of labels of the *goto* transitions out of the state s in M. That is $label(s) = \{\langle c, r \rangle \mid goto(s, \langle c, r \rangle) \text{ is defined}\}$.

We now explain how to take transitions in the goto tree. Recall the basic AC search loop shown in Section 3.2. Suppose we are searching a rectangular text T, and are currently in state x of automaton M and at position $T[i,j]$. Suppose that x represents a square sub-pattern $x[1 \ldots t, 1 \ldots t]$.

> **Fact 5.1** Under the above conditions, being in state x signifies that $x[1 \ldots t, 1 \ldots t] = T[i - t + 1 \ldots i, j - t + 1 \ldots j]$; x indicates how much is matched.

Let $col[i,j] = T[1 \ldots i, j]$ denote the j^{th} subcolumn of the text T running from top and ending in the i^{th} position. Similarly let $row[i,j] = T[i, 1 \ldots j]$ denote the i^{th} subrow of T running from left and ending in the j^{th} position. To move from x to the next state in searching, the question we need to answer is: Is there an $\alpha = \langle \alpha_c, \alpha_r \rangle$ such that the goto tree of M has a transition out of state x with label α such that

$$\langle \alpha_c, \alpha_r \rangle \leq_s \langle col[i+1, j+1], row[i+1, j+1] \rangle\, ? \tag{5.1}$$

Figure 5.1 shows an example of such a scenario. The text is shown with each character in a separate (dashed) tile. The solid box shows the current state x. The larger $\lrcorner$-strip corresponds to $\langle col[4,4], row[4,4] \rangle$. If we set α to the smaller $\lrcorner$-strip, then Equation 5.1 will be satisfied because the smaller strip is enclosed by the larger strip.

Notice that this is similar to the question we ask in the one dimensional automata except that we replace the relation '$=$' by the relation '$\leq_s$'. The next lemma helps us decide quickly and uniquely if there is a suitable *goto* transition.

> **Lemma 5.2** There is at most one α that satisfies the condition $\alpha \leq_s \langle col[i+1, j+1], row[i+1, j+1] \rangle$. Moreover, the set of labels of all the *goto* transitions out of state x, $label(x)$, is totally ordered under the relation $<_{inv}$.

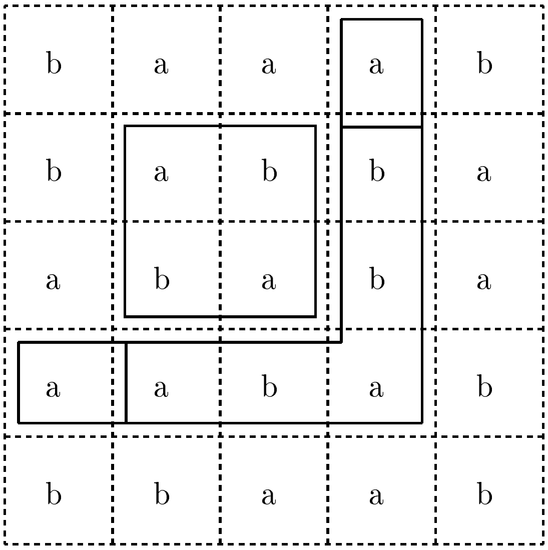

Figure 5.1 Taking a $goto^{\square}$ Transition

Proof From Fact 5.1, being in x signifies that a square of size exactly $t \times t$ has been matched. Thus all the goto transitions out of x correspond to adding an $\lrcorner$ shaped strip of size exactly $2t+2$, with a column of length $t+1$ and a row of length $t+1$. It follows from the definition of $\leq_s$ that for any two strings x, y of equal length, $x \leq_s y$ (or $y \leq_s x$) implies that $x = y$. Thus between any two different transition labels it is not possible for one to be $\leq_s$ to the other one. Thus at most one transition label can be $\leq_s$ to the ordered pair $\langle col[i+1, j+1], row[i+1, j+1] \rangle$.

Let $\alpha_1 = \langle c_1, r_1 \rangle$, and $\alpha_2 = \langle c_2, r_2 \rangle$ be the labels of two distinct transitions out of x. Since c_1 and c_2 (and similarly r_1 and r_2) are of the same length, either $\alpha_1 <_{inv} \alpha_2$ or $\alpha_2 <_{inv} \alpha_1$ holds. From this, it follows that all the labels are totally ordered under $<_{inv}$. $\square$

When we search a text using an AC (or IS) automaton, we are always in some state with a prefix, say x. On the next input character a, we look for a *goto* transition out of x with label a. If there is such a transition then we take it and enter the state xa. Otherwise we move to the state $fail(x)$ and try again. Since there may be multiple transitions (equal to the size of the alphabet in the worst case) out of each state in the automaton, we build a balanced binary search tree on the top of the transitions out of each state, with the label of each edge as its key. With this scheme we can check and find a transition with a given label in $O(\log \sigma)$ tree comparisons.

We use the same scheme to organize $label(x)$ for every state $x \in M$. We can do this since the labels are totally ordered under the $<_{inv}$ relation by Lemma 5.2. Since the labels in the two dimensional case are $\lrcorner$-strips, there could be $O(d)$ of them. Therefore, the "alphabet" is $O(d)$ in this case. With this scheme, finding a suitable label α (if it exists) takes $O(\log d)$ tree comparisons.

5.3 Text Searching

The searching algorithm implicitly linearizes the patterns into $\lrcorner$-strips as explained above. We search the text along each diagonal, considering the pair of the subcolumn and the subrow ending at each position of the text as the character to be matched. We also slide the pattern index (corresponding to the lower right corner) diagonally over the text using the dictionary automaton M to keep track of what is matched.

In order to facilitate text searching and dictionary updates, we build five auxiliary *one dimensional* dynamic dictionaries on the top of M using the IS algorithm. Three of them will be used for updates and will be described in the next section. We now describe the other two dictionaries which will be used in text searching.

The first dictionary is called C^{up} and contains all the column strings in the u-t halves of all patterns. There is a symmetric counterpart called R^{lo} that contains all the row strings in the l-t halves of all patterns.

In Example 5.1, the dictionary C^{up} contains the three columns a, ba, cab of the u-t half, and the dictionary R^{lo} contains the three rows a, ca, dbb of the l-t half.

There are two major steps in searching a text $T[1 \ldots n_r, 1 \ldots n_c]$:

Step 1: Preprocess the text once by columns and once by rows using C^{up} and R^{lo} respectively.

Step 2: Search the text using M reporting matched patterns.

In the preprocessing step (**Step 1**) our goal is to label each text position $T[i, j]$ with two labels $C[i, j]$ and $R[i, j]$. The label $C[i, j]$, representing a string of length q say, is the longest u-t pattern column in the dictionary C^{up} that matches $col[i, j]$ ending at position $T[i, j]$; that is to say $C[i, j] = T[i - q + 1, j] \ldots T[i, j]$. Recall that our u-t pattern columns go towards the diagonal, which is also top-to-bottom order.

Similarly $R[i,j]$ is the longest l-t pattern row in the dictionary R^{lo} that matches $row[i,j]$ ending at position $T[i,j]$. Again we read the rows towards the diagonal, which is also left-to-right order.

Once we label each text location $T[i,j]$ with $C[i,j]$ and $R[i,j]$, we can replace $\langle col[i+1,j+1], row[i+1,j+1]\rangle$ with $\langle C[i+1,j+1], R[i+1,j+1]\rangle$ in Test 5.1 because $C[i+1,j+1]$ and $R[i+1,j+1]$ are the longest matching column and row in their respective halves.

Since C^{up} and R^{lo} are one dimensional dynamic dictionaries built by using the IS algorithm, we can search each column and row of the text $T[1\ldots n_r, 1\ldots n_c]$ treating each column as a one dimensional string to be matched with dynamic dictionary C^{up}, and each row as a one dimensional string to be matched with R^{lo}. We can conclude from Theorem 3.2 that:

> **Corollary 5.1** All the labels $C[i,j]$ and $R[i,j]$ can be computed in $O(n_r n_c \log(|C^{up}| + |R^{lo}|)) = O(t \log d)$ time.

The labels $C[i,j]$ and $R[i,j]$ are stored as addresses of states in the appropriate automaton. These addresses will be used as characters later. This completes the description of **Step 1**.

Step 2 is searching using M, which has been outlined in the previous section. The pseudocode for the searching algorithm is given at the end of this section. The main components in the searching algorithm are computing *goto* and $fail$. We assume for now that computing $fail$ takes $O(\log d)$ time, which we will prove in the next section. We now show that computing *goto* takes $O(\log d)$ time.

> **Lemma 5.3** Suppose we are in the state x of M and at the position $T[i,j]$ of the text. We can choose $\alpha = \langle \alpha_c, \alpha_r\rangle \in label(x)$ such that $\alpha \leq_s \langle C[i+1,j+1], R[i+1,j+1]\rangle$, in $O(\log d)$ time.

Proof We noted at the end of previous section that we can determine α in $O(\log d)$ binary tree comparisons. Since α_c and $C[i+1,j+1]$ (α_r and $R[i+1,j+1]$) are pointers in C^{up} (R^{lo}), each binary tree comparison takes $O(1)$ time by Lemma 3.1 and Lemma 5.1. Therefore we can choose α in $O(\log d)$ time. □

The detection of matched patterns can be done as in Chapter 3. We assume that each pattern in D is appended with a \$-strip such that \$ does not occur in any pattern. Having computed the next state after x, we pretend to read a \$-strip, which is the end of pattern symbol. We repeatedly follow the fail tree reporting each match along the way. Note that following each fail link may take $O(\log d)$ time. The pseudocode for searching a text appears below. The text is assumed to be a square for simplifying the code, though the algorithm works for any rectangular text.

Algorithm 7 Code for searching a text.

```
2D-SEARCH(T[1...n, 1...n])
    Create C[1...n, 1...n] by searching every column of T in C^up.
    Create R[1...n, 1...n] by searching every row of T in R^lo.
    for i ← −(n − 1) to (n − 1) do
        Let C_i[1...n − |i|] be the i^th diagonal of C.
        Let R_i[1...n − |i|] be the i^th diagonal of R.
        state ← ε
        for j ← 1 to n − |i| do
            while there is no α ∈ label(state) such that α ≤_s ⟨C_i[j], R_i[j]⟩ do
                state ← fail□(state)
            state ← goto□(state, α)
            /* Pretend a $-strip is read to check if any patterns match */
            temp ← goto□(state, $-strip)
            if temp is not a pattern then temp ← fail□(temp)
            while temp ≠ $-strip do /* Report all non-empty patterns */
                Print the pattern associated with temp
                temp ← fail□(temp)
```

We summarize the time complexity of searching a rectangular text:

Theorem 5.1 Given M, C^{up}, and R^{lo}, we can search a text T and report all matched patterns in time

$$O((t + tocc)(\log(|M| + |R^{lo}| + |C^{up}|)) = ((t + tocc)(\log d)).$$

Proof The first line of the pseudocode takes $O(t\ \log|C^{up}|)$ time, which follows from Theorem 3.2. Similarly the second lines takes $O(t\ \log|R^{lo}|)$ time. The rest of the code is the same as the AC algorithm except that we are running with a linearized alphabet. So the running time of it is $O((t + tocc)\log|M|)$. Since the size of the dictionary is $O(d)$ the claimed result follows. $\square$

5.4 Inserting and Deleting Patterns

We now describe how square patterns are inserted into or deleted from a dictionary. As described earlier, the *goto* function can be implemented easily with the help of two auxiliary one dimensional dynamic dictionaries C^{up} and R^{lo}. In this section, we use another three auxiliary one dimensional dictionaries to implement $fail$. The IS algorithm for one dimensional strings relies on the properties of simple characters for implementing the $fail$ function. Since we are dealing with complex $\lrcorner$-strip characters, we do not know of any way to use the same techniques applicable to one dimensional dictionaries. The problem can be explained as follows:

For a one dimensional string w of length n, let x be its prefix of length k and y be its suffix of length $n-k$. Then w can be written as xy. In other words x and y *together* have all the information of w. Unfortunately, the same property does not hold for the case of square patterns. This property is crucial for using the techniques of the IS algorithm. Therefore we need an indirect way of computing $fail^{\square}$.

We describe two of the three auxiliary dictionaries. The first dictionary, called R^{up}, contains all the *rows* in the u-t halves of the patterns. This is different from C^{up} which stores the *columns* of the u-t halves. Another difference is that these rows are read *away from the diagonal.* In Example 5.1, R^{up} contains abc, aa, b. Similarly, the second dictionary, called C^{lo}, contains all the *columns* in the l-t halves of the patterns, read *away from the diagonal.* In Example 5.1, C^{lo} contains acd, ab, b. The use of these dictionaries will be apparent soon.

As mentioned earlier, a new pattern is inserted one $\lrcorner$-strip at a time, starting with the shortest strip. Each new strip may contribute one new column to C^{up} and one new row to R^{lo}. Each new strip of length $2p+2$ (diagonal character duplicated) may add a single one character string (the diagonal character) to R^{up} and C^{lo} as well as extending p strings in each of those dictionaries by one character per string.

The insertions can be done as in Chapter 3. The IS algorithm inserts a pattern P in $O(p \log d)$ time. Moreover, the IS algorithm does insertions one character at a time from left to right at a cost of $O(\log d)$ per character. Thus the entries in R^{up} and C^{lo} that need to be extended by one character can be extended in $O(\log d)$ time per character using the EXTEND operation.

Deletions are done similar to the insertions, but in the reverse order. A pattern is deleted one $\lrcorner$-strip at a time, starting with the longest strip. Each such strip may delete one column from C^{up} and one row from R^{lo}. Each strip of length $2p+2$ may delete a single one character string (the diagonal character) from R^{up} and C^{lo} as well as truncating p strings in each of those dictionaries by one character per string.

The deletions can be done as in Chapter 3. The IS algorithm deletes a pattern P in $O(p \log d)$ time. Moreover, the IS algorithm does deletions one character at a time from right to left at a cost of $O(\log d)$ per character. Thus the entries in R^{up} and C^{lo} that need to be truncated by one character can be truncated in $O(\log d)$ time per character using the TRUNCATE operation.

In the rest of this section we see the details of implementing $fail^{\square}$.

To facilitate the explanation we treat the u-t and l-t halves of our square patterns separately. Later we extend this to complete square patterns. In what follows we concentrate on u-t patterns. The l-t patterns are treated similarly. We use the following 4×4 u-t pattern for illustration.

$$\begin{array}{cccc} a & b & a & b \\ & b & b & c \\ & & a & b \\ & & & b \end{array}$$

Let U be a u-t pattern of width m. Let a_{ij}, for $1 \leq i \leq j \leq m$, be the characters of U. We treat the pattern as a one dimensional string $U_C = C_1; \ldots; C_m$ where each C_i stands for the i^{th} u-t column representing the string $a_{1i} \ldots a_{ii}$. We use semi-colons to separate individual columns (rows) when we write a two dimensional pattern as a one dimensional pattern of columns (rows). In the example above $U_C = a; bb; aba; bcbb$. Alternatively we can treat U as a one dimensional string $U_R = R_1; \ldots; R_m$ where each R_i stands for the i^{th} u-t row representing the string $a_{ii} \ldots a_{im}$. In the example above $U_R = abab; bbc; ab; b$. Our aim is to insert U_C into an automaton which we use for two dimensional matching.

We will pretend momentarily that our two dimensional automaton M stores only u-t patterns; later we generalize to full square patterns. We use a *one dimensional* dynamic dictionary M', which is our last auxiliary dictionary, to help compute the $fail$ function for M. For any position (i, j) of U we define a sub-row associated with that position. This is precisely the sub-row of U ending at that position, namely $r_{ij} = a_{ii} \ldots a_{ij}$. In the example above $r_{13} = aba$ and $r_{23} = bb$. For every U_C inserted into M, we insert a corresponding $U'_C = C'_1; \ldots; C'_m$ into M' defined as follows. If

$C_i = a_{1i} \ldots a_{ii}$ then $C'_i = r_{1i}; \ldots; r_{ii}; \$$, where $\$$ is a special character distinct from any r_{ij}. In the example above $C'_3 = aba; bb; a; \$$. If we insert the rows of U_R into a separate one dimensional automaton R^{up} as described earlier, each r_{ij} will be a prefix in R^{up} and hence will have a corresponding state in that automaton. We can then treat each r_{ij} as a *maha-character* (*maha* means *super* in Sanskrit) which is basically the address of the state in the automaton R^{up}. We call each transformed column C'_i a *maha-column*. Each maha-column encodes a whole prefix of a u-t pattern. More specifically C'_i represents the entire prefix $C_1; \ldots; C_i$ of U_C in a succinct way. It is important to note that a maha-column is a one dimensional string. In our example $C_1; \ldots; C_3 = a; bb; aba$ and the corresponding $C'_3 = aba; bb; a; \$$. It is not difficult to see that these represent the same u-t sub-pattern.

Now we insert each maha-column as a one dimensional string into the dictionary M' and compute the fail functions treating each maha-character as a simple character. Since each maha-character r_{ij} stands for a prefix in R^{up} we can use the $<_{inv}$ order of the prefixes to order the maha-characters. The relative order of two maha-characters is same as the relative $<_{inv}$ order of the corresponding prefixes in R^{up}. From Lemma 5.1, we can determine this in constant time. The following lemma is crucial to the correctness of our construction.

Let $ut(P)$ ($lt(P)$) denote the upper (lower) triangular half of a square subpattern P. Let $fail_M$ and $fail_{M'}$ denote the $fail$ functions in M and M' respectively.

> **Lemma 5.4** Let $fail^u$ denote the fail function for a dictionary of u-t patterns. Let $U_1[1 \leq i \leq j \leq m_1]$ and $U_2[1 \leq i \leq j \leq m_2]$ be two u-t subpatterns and $U'_1 = C^1_1; \ldots; C^1_{m_1}; \$$ and $U'_2 = C^2_1; \ldots; C^2_{m_2}; \$$ be the maha-patterns associated with U_1 and U_2 respectively. Then $fail^u_M(U_1) = U_2$ if and only if $fail_{M'}(U'_1) = U'_2$.

Proof

($\Longrightarrow$) Suppose $fail^u_M(U_1) = U_2$. From the definition of $fail^u$, U_2 is the largest subpattern such that U_2 is a suffix of U_1. In other words, $m_2 < m_1$ is the largest index such that $U_2[1 \leq i \leq j \leq m_2] = U_1[m_1 - m_2 + 1 \leq i \leq j \leq m_1]$. From the construction of M', it follows that $m_2 < m_1$ is the largest index such that $C^2_1; \ldots; C^2_{m_2}; \$ = C^1_{m_1-m_2+1}; \ldots; C^1_{m_1}; \$$. This implies that $fail_{M'}(U'_1) = U'_2$.

($\Longleftarrow$) Suppose $fail_{M'}(U'_1) = U'_2$. From the definition of $fail$, U'_2 is the largest subpattern in M' and a suffix of U'_1. In other words, $m_2 < m_1$ is the largest index such that $C^2_1; \ldots; C^2_{m_2}; \$ = C^1_{m_1-m_2+1}; \ldots; C^1_{m_1}; \$$. From the construction of M', it follows that $U_2[1 \le i \le j \le m_2] = U_1[m_1 - m_2 + 1 \le i \le j \le m_1]$. This implies that $fail^u_M(U_1) = U_2$. $\square$

Thus we have found an indirect way of computing $fail^u_M$ by changing every u-t pattern into an equivalent maha-pattern and computing $fail_{M'}$. We can do a similar procedure for building an automaton for l-t patterns by defining L, L_R, c_{ij}, L'_R.

We now generalize the way in which we used M and M' to store u-t patterns to have M and M' store square patterns instead. For this we make another transformation on U'_C and L'_R before inserting them into M'. Let $C'_i = r_{1i}; \ldots; r_{ii}; \$$ and $R'_i = c_{i1}; \ldots; c_{ii}; \$$ be the i^{th} maha-column and maha-row aligning at the i^{th} position on the main diagonal. We *shuffle* them into a single string $CR_i = r_{1i}; c_{i1}; \ldots; r_{ii}; c_{ii}; \$; \$$ (interleaving the maha-column and the maha-row) and insert it into M'. The interleaved string is always of even length. In this case we treat each pair of characters—one belonging to the maha-column and the other to the maha-row—as a single character and insert it into M'. We can prove similarly to Lemma 5.4 that:

Theorem 5.2 Let $P_1[1 \ldots i, 1 \ldots i]$ and $P_2[1 \ldots j, 1 \ldots j]$ be two square sub-patterns and $P'_1[1 \ldots i+1] = CR^1_1; \ldots; CR^1_i; \$\$$ and $P'_2[1 \ldots j+1] = CR^2_1; \ldots; CR^2_j; \$\$$ be the maha-patterns associated with P_1 and P_2 respectively. Then $fail^{\square}_M(P_1) = P_2$ if and only if $fail_{M'}(P'_1) = P'_2$.

Proof

($\Longleftarrow$) Suppose $fail_{M'}(P'_1) = P'_2$. From the definition of $fail$, P'_2 is the longest string in M' such that $CR^2_1; \ldots; CR^2_j; \$\$ = CR^1_{i-j+1}; \ldots; CR^1_i; \$\$$. From the construction of M', it follows that j is the largest index such that *both* $C^2_1; \ldots; C^2_j; \$ = C^1_{i-j+1}; \ldots; C^1_i; \$$ and $R^2_1; \ldots; R^2_j; \$ = R^1_{i-j+1}; \ldots; R^1_i; \$$ hold. From Lemma 5.4, j is the largest index such that $ut(P_2)$ is a suffix of $ut(P_1)$ and $lt(P_2)$ is a suffix of $lt(P_1)$. This means that j is the largest index such that P_2 is a suffix of P_1. From the definition of $fail^{\square}$ it follows that $fail^{\square}_M(P_1) = P_2$.

($\Longrightarrow$) Suppose $fail^{\square}_M(P_1) = P_2$. From the definition of $fail^{\square}$, P_2 is the largest square subpattern of M that aligns with the lower right hand corner of P_1. In other words, j is the largest index such that P_2 is a suffix of P_1, or $ut(P_2)$ is a suffix of $ut(P_1)$ and $lt(P_2)$ is a suffix of $lt(P_1)$. From Lemma 5.4, j is the largest index such that $C^2_1; \ldots; C^2_j; \$ = C^1_{i-j+1}; \ldots; C^1_i; \$$ and $R^2_1; \ldots; R^2_j; \$ = R^1_{i-j+1}; \ldots; R^1_i; \$$. From the construction of M', it follows that $CR^2_1; \ldots; CR^2_j; \$\$ = CR^1_{i-j+1}; \ldots; CR^1_i; \$\$$. Therefore $fail_{M'}(P'_1) = P'_2$. $\square$

We now give the pseudocode for inserting or deleting a pattern.

Algorithm 8 Code for inserting a pattern into a dictionary.

2D-INSERT$(P = a_{ij}, 1 \leq i, j \leq m)$
1 Insert the upper columns of P into C^{up}
2 Insert the lower rows of P into R^{lo}
3 Insert the upper rows of P into R^{up}, creating $r_{ij}, 1 \leq i \leq j \leq m$
4 Insert the lower columns of P into C^{lo}, creating $c_{ij}, 1 \leq j \leq i \leq m$
5 Let $P' = r_{ij}, 1 \leq i \leq j \leq m; c_{ij}, 1 \leq j \leq i \leq m$ be the maha-pattern for P
6 Insert column-row pairs of P into M
7 Insert interleaved column-row pairs of P' into M'

Algorithm 9 Code for deleting a pattern from a dictionary.

2D-DELETE$(P = a_{ij}, 1 \leq i, j \leq m)$
1 Create the maha-pattern P' by searching P in R^{up} and C^{lo}
2 Delete column-row pairs of P from M
3 Delete interleaved column-row pairs of P' from M'
4 Delete upper rows of P from R^{up}
5 Delete lower columns of P from C^{lo}
6 Delete upper columns of P from C^{up}
7 Delete lower rows of P from R^{lo}

The $fail^{\Box}$ function for M is implicitly computed as follows: From Theorem 5.2, we can replace the use of $fail^{\Box}$ at all places in 2D_SEARCH by $fail_{M'}$. Since M' is a one dimensional dictionary, $fail_{M'}$ is implicitly updated at line 7 of 2D_INSERT and at line 3 of 2D_DELETE.

We summarize the time complexity of inserting (or deleting) a new pattern of size $m \times m$ into M and M':

Theorem 5.3 A square pattern of size $m \times m$ can be inserted or deleted from M and M' in $O(m^2 \log d)$ time.

Proof All the lines in 2D_INSERT as well as 2D_DELETE are simple insertions and deletions into one dimensional dictionaries using the IS algorithm. Hence the stated time bounds follow directly from the bounds in Chapter 3. $\Box$

Chapter 6

Multiple Matching with Rectangular Patterns

6.1 Introduction

In this chapter we study the problem of Two-Dimensional Multiple Pattern Matching (2MPM), which is an extension of the 2FPM problem defined in Chapter 1. Our main result is the first efficient algorithm for matching multiple rectangular patterns of arbitrary sizes and aspect ratios in a rectangular text. Our result addresses a longstanding open problem posed by T. P. Baker in 1978 [21].

The 2FPM problem was solved by Bird [23] and independently by Baker [21]. The basic idea in their algorithms is to *linearize* the pattern by treating each column as a single character in a new derived alphabet. Bird and Baker use the AC algorithm as a subroutine to transform the columns of pattern and text into these special characters and then run the KMP algorithm to search the modified text. The preprocessing time is $O(p \log \sigma)$ and the search time is $O(t \log \sigma)$. Other algorithms that seem to perform better on random patterns have been described by Zhu and Takaoka [67] and Baeza-Yates and Régnier [19]. Recent papers by Amir, Benson, and Farach [5] and Galil and Park [40] give new algorithms whose running times are linear without depending on the alphabet size.

Since he used an algorithm for MPM to solve the 2FPM problem, it is not surprising that Baker [21] observed that it is natural to combine the two paradigms and define the 2MPM problem. Template-based computer vision is one natural motivation for 2MPM. We can think of a seeing being as having a mental dictionary of known templates (patterns) that are quickly matched against a new scene (text). For other applications of two dimensional matching and a general survey with lots of references see Section 7.3.2 of [44].

If all the patterns in an instance of 2MPM are of fixed height, then Baker [21] noted that his algorithm can be adapted to solve 2MPM simply by replacing the use of the KMP algorithm for text searching with a use of the AC algorithm. The search time for one height or width becomes $O(t \log(n + \sigma) + tocc)$. When the number of different heights, widths, and aspect ratios is large, we would like an algorithm whose performance depends as little as possible on the number of different pattern sizes. Baker explicitly left open this question of how to solve 2MPM efficiently when the patterns have varying heights and widths [21].

We use a different approach to solve 2MPM for rectangular patterns. Rectangular patterns of different heights, widths, and aspect ratios cannot be aligned at a corner. Thus there is no natural way to define the "biggest suffix" of the text that matches a "prefix" of some pattern.

Our first basic idea is to split the patterns into two subpatterns each, so that the number of different subpattern heights and widths is small. In the text searching phase we recognize occurrences of the subpatterns and then combine occurrences of subpatterns that represent an occurrence of a full pattern. The most interesting feature of our algorithm is a connection between two dimensional pattern matching and some multidimensional range searching problems. Our 2MPM algorithm also includes new adaptations of three techniques from algorithms for *other* two dimensional string matching problems: splitting patterns into two overlapping pieces of fixed height [43], focusing on text columns that are (numbered) 0 mod q for some appropriate q (see [16, 19]), and the smaller matching with tree partial order paradigm, which was defined and studied in [10]. For other, different uses of computational geometry in string matching see [55, 16].

The preprocessing time, text searching time, and space of our 2MPM algorithm depend in complicated ways on the distribution of patterns among the different sizes because this affects which known range searching algorithm is best to use. Following the general trend in string matching, we restrict ourselves to variations of our algorithm that use linear, i.e. $O(d)$, space, and get the best time possible using linear space. We do not count the $O(t)$ space needed to store the input text.

Define the *linear size* $b(P)$ of a rectangular pattern P to be the *smaller* of its height and width. Let B be the biggest linear size of any pattern. To express the possible running times of our algorithm we give one definition.

> **Definition 6.1** We call a dictionary *size-diverse* if for some fixed $k > 1$, the number of patterns of any fixed linear size $b \geq \log d$ is $O(b^k)$.

Size-diversity characterizes one extreme of dictionaries, where the number of distinct linear sizes of patterns is large. For size-diverse dictionaries our algorithm achieves the following time bounds using linear space:

Dictionary Preprocessing: $O(d \log(n + \sigma))$.

Text Searching: $O(t \log d \log(B + n + \sigma) + tocc)$.

There are many dictionaries that are not size-diverse for which the above time bounds still hold. For any dictionary, we can achieve the combination:

Dictionary Preprocessing: $O(d \log(n + \sigma))$.

Text Searching: $O(t \log^2 d \log(B + n + \sigma) + tocc)$.

It is surprising that our algorithm may achieve better times for size-diverse dictionaries because this looks like the hardest case from the perspective of the Bird-Baker algorithm.

The rest of this chapter is organized as follows. In Section 6.2 we give basic definitions and an overview of our method. In Section 6.3 we present the connection between pattern matching and computational geometry showing how to use the connection to preprocess the patterns and do the final stage of matching against the text. In Section 6.4 we give more details of the text searching algorithm and prove the resource bounds. In Section 6.5, we show how our algorithm can be extended to a dynamic setting where the set of patterns can change over time.

6.2 Definitions and Algorithm Overview

In this section we give some basic definitions, introduce some auxiliary data structures, and give an overview of the text searching algorithm. The algorithm begins by making two classifications into cases. First, we assume that all patterns have height $\leq$ width; the opposite case can be handled in the same way after transposing all patterns and the text. Second, we partition the set of patterns into $\lfloor \log B \rfloor$ sets $\mathcal{P}_0, \mathcal{P}_1, \ldots$ such that:

> **Definition 6.2** The patterns in set $\mathcal{P}_g$ have a height h in the range $2^g < h \leq 2^{g+1}$.

For each g, we search the text for just the patterns in $\mathcal{P}_g$ in one phase. Since there are at most $\lfloor \log B \rfloor$ sets of patterns, this contributes a multiplicative factor of $O(\log B)$ to the text searching time, but does not change the asymptotic preprocessing time or space.

From now on we focus on one specific $\mathcal{P}_g$ and assume we are just interested in the patterns in that set. We use the following example to illustrate various definitions.

Example 6.1 Consider a dictionary of two 3×4 patterns. Here, $g = 1$ and $h = 3$.

$$\begin{array}{cccc} a & b & c & a \\ c & a & a & b \\ a & b & b & a \end{array} \qquad\qquad \begin{array}{cccc} b & c & a & b \\ a & a & b & c \\ b & b & a & b \end{array}$$

Our first key idea is to divide each pattern into two pieces, such that each piece has height 2^g and full width. For a pattern of height h, the top piece includes rows $1, \ldots, 2^g$, and the bottom piece includes rows $h - 2^g + 1, \ldots, h$. The two pieces share $2^{g+1} - h$ rows and will overlap unless $h = 2^{g+1}$.

Definition 6.3 We call the pattern pieces *half patterns*; the top pieces are called *upper half patterns*, and the bottom pieces are called *lower half patterns*.

In Example 6.1, the upper half patterns are:

$$\begin{array}{cccc} a & b & c & a \\ c & a & a & b \end{array} \qquad\qquad \begin{array}{cccc} b & c & a & b \\ a & a & b & c \end{array}$$

And the lower half patterns are:

$$\begin{array}{cccc} c & a & a & b \\ a & b & b & a \end{array} \qquad\qquad \begin{array}{cccc} a & a & b & c \\ b & b & a & b \end{array}$$

Definition 6.4 Let C be the set of columns that occur in some half pattern.

As in the Bird-Baker algorithm, each column is assigned a number (memory address) that we think of as a character, and then we view each half pattern as a one dimensional string of column characters. In Example 6.1, the set of columns is $C = \{ab, ac, ba, bc, ca, cb\}$, where each column is written as a string written top-to-bottom. For convenience, we write a two dimensional pattern as a one dimensional string of columns, where each column is separated by a vertical bar. With this notation, we can write the first pattern of Example 6.1 as $aca|bab|cab|aba$, its upper half pattern as $ac|ba|ca|ab$, and its lower half pattern as $ca|ab|ab|ba$.

Since all half patterns are of the same height, we could use the Bird-Baker algorithm to recognize all occurrences of half patterns. To do the efficient synthesis of half patterns into full patterns, we need extra preprocessing.

> **Definition 6.5** A *prefix* (*suffix*) of a rectangular pattern is a prefix (suffix) of its string-of-column-character representation. The *reverse* of a rectangular pattern is the reverse of its string-of-column-character representation (i.e, the pattern read right-to-left).

In Example 6.1, $bab|cab|aba$ is a prefix of the second pattern and a suffix of the first pattern, whereas $bcb|aba|cab|bab$ is the reverse of the second pattern.

We maintain two extra dictionaries of upper half patterns.

> **Definition 6.6** The dictionary $H^{f,u}$ (H for half, f for forward, u for upper) contains the 2^g *longest* prefixes of each upper half pattern. The dictionary $H^{b,u}$ (b for backward) contains the 2^g *smallest* non-empty prefixes of the reverse of each upper half pattern; in other words, $H^{b,u}$ contains the last 2^g non-empty suffixes of each upper half pattern, with the prefixes stored in reverse fashion. We also maintain two similar dictionaries $H^{f,l}$ and $H^{b,l}$ for lower half patterns.

In Example 6.1, $H^{f,u} = \{ac|ba|ca|ab,\ ac|ba|ca,\ ba|ca|ab|bc,\ ba|ca|ab\}$, $H^{f,l} = \{ca|ab|ab|ba,\ ca|ab|ab,\ ab|ab|ba|cb,\ ab|ab|ba\}$, $H^{b,u} = \{ab,\ ab|ca,\ bc,\ bc|ab\}$, and $H^{b,l} = \{ba,\ ba|ab,\ cb,\ cb|ba\}$.

The reason for storing the prefixes and reverse prefixes is that our final searching pass will look only in text columns that are numbered 0 mod 2^g. Following [16], we call these *power columns*. If there is a match spanning columns $[c_1 \ldots c_2]$ of the text, we will find it in the rightmost power column $c \leq c_2$. The pattern occurrence must include at least one power column because its width is assumed to be at least as big as its height, which is at least $2^g + 1$. The part of the match in text columns $c_2, c_2 - 1, \ldots, c$ (read from right to left) will have its upper half in $H^{b,u}$ and its lower half in $H^{b,l}$. The part of the match in text columns $c_1, c_1 + 1, \ldots, c$ (read from left to right) will have its upper half in $H^{f,u}$ and its lower half in $H^{f,l}$.

Let P be a pattern of width w. Since we cannot predict where in P an occurrence will first intersect a power column, we define:

Definition 6.7 An *avatar** of pattern P is an ordered pair (P_f, P_b) such that P_f is a prefix of P of length at least $w - 2^g + 1$ and P_b is the reverse of a suffix of P of length at most 2^g. The strings P_f, P_b overlap in exactly one column character. We say the P_f is the *forward projection* of the avatar, and P_b is the *backward projection* of the avatar.

Our intention is that a match of pattern P can be viewed as that avatar, (P_f, P_b), where P_f is the part to the left of and including power column and P_b is the part to the right of and including the power column. Each pattern has 2^g avatars. In Example 6.1, the avatars of the first pattern are $(aca|bab|cab|aba, aba)$ and $(aca|bab|cab, aba|cab)$. The avatars of the second pattern are $(bab|cab|aba|bcb, bcb)$ and $(bab|cab|aba, bcb|aba)$.

For every h, $2^g < h \leq 2^{g+1}$, we maintain two one dimensional dictionaries on the avatars of all patterns of height h.

Definition 6.8 The dictionary F_h contains the forward projections of the avatars of all the patterns. Similarly B_h contains all the backward projections of the avatars of all the patterns.

In Example 6.1, $F_3 = \{aca|bab|cab|aba, aca|bab|cab, bab|cab|aba|bcb, bab|cab|aba\}$, and $B_3 = \{aba, aba|cab, bcb, bcb|aba\}$.

We also extend the definitions of avatar and projections to half patterns. Notice that we have set up the auxiliary dictionaries so that the forward projection of any avatar of any upper half pattern is in $H^{f,u}$, the backward projection of any avatar of any upper half pattern is in $H^{b,u}$, and similarly for lower half patterns.

Our text searching algorithm has two basic parts, a text preprocessing step and a matching step. In the preprocessing step our goal is to find for text location $T[i,j]$ in power column j, the widest elements of $H^{f,u}$ and $H^{f,l}$ that match the text when their upper *right* corner is placed at $T[i,j]$. The answers are placed in the auxiliary arrays FU, FL. Similarly, we seek the widest elements in $H^{b,u}$ and $H^{b,l}$ that match the text when their upper *left* corner is placed at $T[i,j]$. These answers are stored in the arrays BU, BL.

In the text searching phase, we ask for each location $T[i,j]$ in a power column and each height $h \in (2^g, 2^{g+1}]$: are there any matches of patterns such that the top row is

*The word avatar comes from Sanskrit and originally means an incarnation of a Hindu deity. In English, avatar also can mean a variant phase of a continuing basic entity (Webster's Dictionary).

i, the rightmost power column is j, and the height of the pattern matched is h? It is important to see that we ask as many as 2^g queries at each power column position, but there are at most $\lfloor t/2^g \rfloor$ power column positions, so the total number of queries is $O(t)$.

To answer a query of height h at $T[i,j]$ we use the information in $FU[i,j]$, $FL[i,j]$, $BU[i+h-2^g,j]$, and $BL[i+h-2^g,j]$. In the next section we show how to transform our queries into geometric problems.

6.3 Preprocessing and Matching Patterns

In this section we investigate how to synthesize projections and subpattern matches into full pattern matches. Our synthesis uses a geometric approach. We describe how to use our geometric pattern representations.

The synthesis from projections and subpattern matches to full pattern matches is carried out in three steps as follows:

Step 1: For a text location $T[i,j]$ on a power column j, we compute $w_f[i,j]$ which is the widest forward projection of a pattern of height h matching at $T[i,j]$. For this we take the values $FU[i,j]$ and $FL[i+h-2^g,j]$, which are pointers into $H^{f,u}$ and $H^{f,l}$ respectively, and transform them into $w_f[i,j]$ which is a pointer in F_h.

Step 2: We do a similar procedure on $BU[i,j]$ and $BL[i+h-2^g,j]$ and obtain the widest backward projection $w_b[i,j]$ of a pattern of height h matching at $T[i,j]$.

Step 3: We use $w_f[i,j]$ and $w_b[i,j]$ to report all matching patterns at the subrow $T[i,j]$ through $T[i,j+2^g-1]$.

We now define a computational geometric problem and one variant of it. We reduce each of the above steps to the variants.

- **Rectangle Enclosure Reporting (RER) [35]:** Given a set V of rectangles in the plane and another such query rectangle Q, the *rectangle enclosure reporting* problem asks for reporting *all* rectangles in V that enclose Q.

- **Nested Rectangle Enclosure Searching (NRES):** We guarantee that the set V is *nested* which means that for $v_1, v_2 \in V$, if v_1 and v_2 intersect then one of them encloses the other. In this case, we report the *smallest* rectangle v in V that encloses Q. Since V is nested the choice of v is unique.

The rectangle enclosure reporting problems can be solved efficiently using multidimensional range searching algorithms [54, 35, 37]. In the rest of the section, we show the reductions from steps $1, 2, 3$ to RER or NRES.

To perform the reductions we define a *dominance* relation on strings as follows:

Definition 6.9 For any two strings u_1 and u_2, u_2 *dominates* u_1 or u_1 is *dominated by* u_2, denoted $u_1 \leq_s u_2$, if u_1 is a suffix of u_2. We also extend the definition of dominance to a pair of strings as follows: $(u_1, v_1) \leq_s (u_2, v_2)$ if $u_1 \leq_s u_2$ and $v_1 \leq_s v_2$.

For example, $cbc \leq_s abcbc$ and $bab|cab|aba \leq_s aca|bab|cab|aba$. Similarly $(cbc, aba) \leq_s (abcbc, bcaba)$ and $(bab|cab|aba, aba) \leq_s (aca|bab|cab|aba, bcb|aba)$. Notice the similarity between the definition of dominance and the definition of $\leq_s$ in Chapter 5.

Definition 6.10 Let D be a one dimensional dictionary automaton constructed using the AC algorithm [1]. For any prefix u in D, define $fail(u)$ as the longest proper suffix v of u such that v is also a prefix in D. The $fail$ link of u points to v.

Fact 6.1 The $fail$ links of a dictionary D form a tree which we call the *fail tree* of the dictionary, denoted by $failtree(D)$.

Figure 6.1 shows $failtree(H^{f,u})$ (left) and $failtree(H^{f,l})$ (right) of Example 6.1. Patterns are shown in bold font. The $fail$ of a node is its parent in the fail tree. We can conclude from Definition 6.10 and Fact 6.1 that:

Fact 6.2 [1, 10] If u_1 and u_2 are prefixes of some patterns in a dictionary D, then $u_1 \leq_s u_2$ if and only if u_1 is an ancestor of u_2 in $failtree(D)$.

Fact 6.3 When we perform an Euler tour of a tree starting at the root we visit each node twice, once in preorder and once in postorder. If we replace the first visit by a left parenthesis and the second visit by a right parenthesis we get a list of balanced parentheses.

We use Fact 6.3 to transform the fail tree of a one dimensional dictionary D with m prefixes to an equivalent list of $2m$ balanced parentheses by performing an Euler tour of $failtree(D)$. Each matching pair of parentheses corresponds to a node in the tree

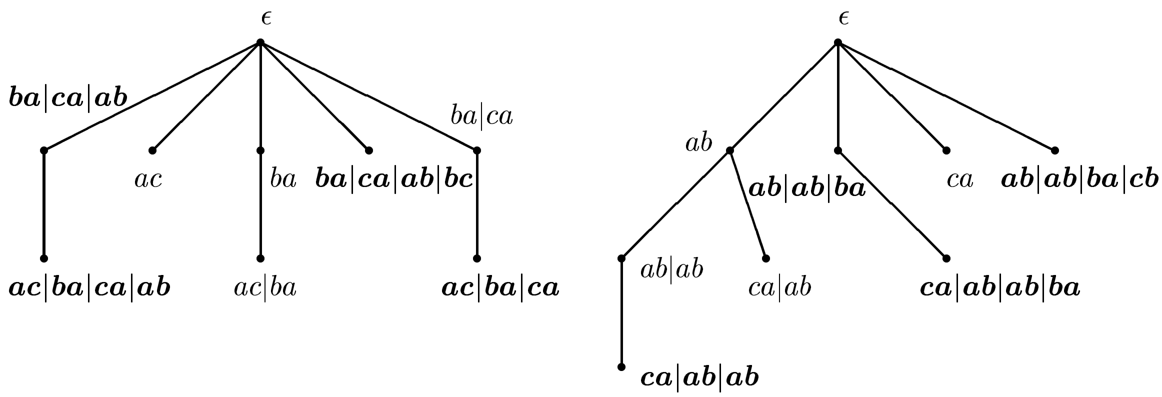

Figure 6.1 Prefixes of $failtree(H^{f,u})$ and $failtree(H^{f,l})$

which also corresponds to a prefix of some pattern. If we do a bijective mapping of the list of $2m$ parentheses to the integer points $0 \ldots 2m-1$ on the real line by assigning to each parenthesis its rank in the list, then each prefix of a pattern corresponds to an interval on the real line, with endpoints corresponding to the ranks of its associated parenthesis pair. It may be noted that these intervals are nesting because the corresponding parentheses are balanced. We call this scheme the *line-mapping* of a dictionary D on to the real line, wherein we map $failtree(D)$ as described above.

Definition 6.11 For a pattern $P \in D$, we use $line(P)$ to denote the interval to which P is mapped. More generally, if u is a prefix of some pattern in D, then $line(u)$ denotes the interval to which u is mapped. Similarly we use $line(D)$ to denote the set of intervals formed.

Figure 6.2 shows the corresponding intervals of the trees in Figure 6.1. Once again, intervals of patterns are shown in bold font. The following lemma follows from the above definition:

Lemma 6.1 Suppose $u, v \in D$. Then $u \leq_s v$ if and only if $line(u)$ encloses $line(v)$.

If u_1, v_1 are prefixes in a dictionary D_1 and u_2, v_2 are prefixes in a dictionary D_2, then $(u_1, u_2) \leq_s (v_1, v_2)$ if and only if u_1 is ancestor of v_1 in $failtree(D_1)$ and u_2 is an ancestor of v_2 in $failtree(D_2)$. If we line-map D_1 on x-axis and D_2 on y-axis,

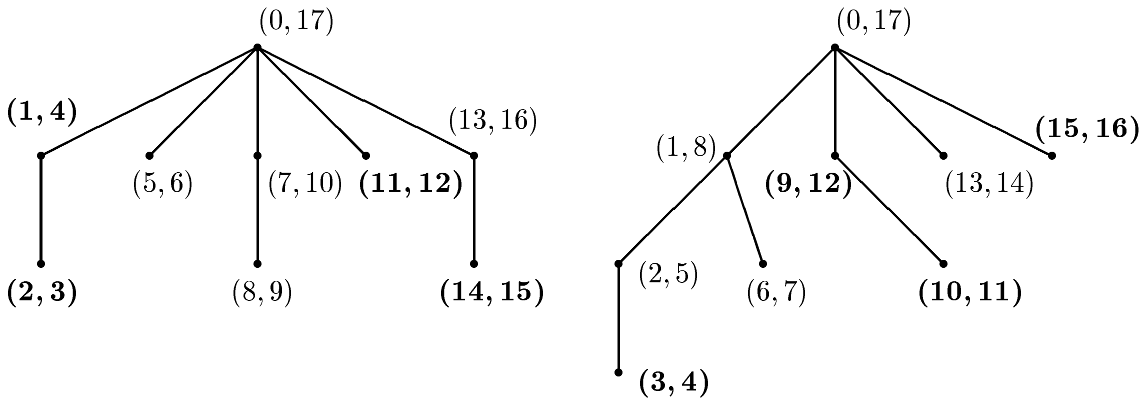

Figure 6.2 Intervals of $failtree(H^{f,u})$ and $failtree(H^{f,l})$

then each pair (u_1, u_2) is mapped to two intervals—$line(u_1)$ on x-axis and $line(u_2)$ on y-axis—which induce a rectangle. We call this scheme the *rectangle-mapping* of a pair of dictionaries (D_1, D_2) on to the plane.

Definition 6.12 For a pair of patterns $P_1 \in D_1$ and $P_2 \in D_2$, we use $rect(P_1, P_2)$ to denote the rectangle to which (P_1, P_2) is mapped. More generally, if u_1 is a prefix in D_1 and u_2 is a prefix in D_2 then $rect(u_1, u_2)$ denotes the rectangle to which (u_1, u_2) is mapped. Similarly we use $rect(D_1, D_2)$ to denote the set of rectangles formed.

In Example 6.1, if we line-map $H^{f,u}$ along x-axis, and $H^{f,l}$ along y-axis, the patterns of F_3 are rectangle-mapped as shown in Figure 6.3. Notice that the rectangles in this case are nesting. The following lemma follows from the above definition:

Lemma 6.2 Suppose $u_1, v_1 \in D_1$ and $u_2, v_2 \in D_2$. Then $(u_1, u_2) \leq_s (v_1, v_2)$ if and only if $rect(u_1, u_2)$ encloses $rect(v_1, v_2)$.

We are now ready to show the reductions:

Theorem 6.1 We can reduce both **Step 1** and **Step 2** to NRES.

Proof We first reduce **Step 1** to NRES. For every forward projection P_f of a pattern of height h in F_h, its upper half pattern P_f^u is in $H^{f,u}$ and its lower half pattern P_f^l is in $H^{f,l}$. Under the rectangle-mapping of $(H^{f,u}, H^{f,l})$, $P_f = (P_f^u, P_f^l)$ is

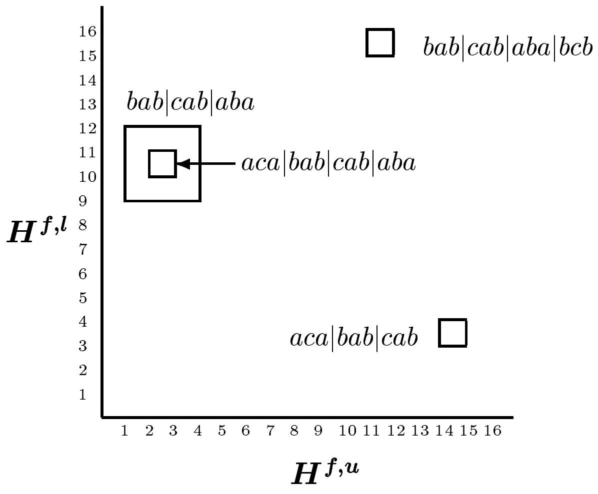

Figure 6.3 Rectangle-mapping of patterns in F_3

transformed into a rectangle in the plane. Let $V_h^f = \{rect(P_f^u, P_f^l) \mid P_f \in F_h\}$ be the set of rectangles corresponding to the patterns in F_h. We want to set $w_f[i,j]$ to the widest P_f such that $(P_f^u, P_f^l) \leq_s (FU[i,j], FL[i+h-2^g, j])$. But since the pair $(FU[i,j], FL[i+h-2^g,j])$ is also transformed into another rectangle in the plane, we have that $rect(P_f^u, P_f^l)$ is the *smallest* rectangle in V_h^f that encloses $rect(FU[i,j], FL[i+h-2^g,j])$ by Lemma 6.2.

Recall that F_h contains the forward projections of patterns of height h. Since we linearize the patterns along their columns, F_h is actually a one dimensional dictionary. Therefore the choice of the widest pattern P_f in F_h is unique. We can infer from this unique choice that the set of rectangles $rect(H^{f,u}, H^{f,l})$ is nesting. Therefore the choice of the widest pattern P_f in F_h, or equivalently the smallest rectangle $rect(P_f^u, P_f^l)$ in V_h^f is unique.

We can reduce **Step 2** to NRES using a similar procedure to the above. We define V_h^b similarly. We set $w_b[i,j]$ to the widest $P_b = (P_b^u, P_b^l)$ in B_h such that $(P_b^u, P_b^l) \leq_s (BU[i,j], BL[i+h-2^g,j])$. □

Theorem 6.2 We can reduce **Step 3** to RER.

Proof The reduction is similar to the one used in Theorem 6.1. Let D_h be the set of patterns of height h in $\mathcal{P}_g$, where $2^g < h \leq 2^{g+1}$. Consider a pattern $P \in D_h$. P has 2^g avatars with each avatar having its forward projection in F_h and backward projection in B_h. Under the rectangle-mapping of (F_h, B_h), each avatar of $P = (P_f, P_b)$ is mapped into a rectangle in the plane. Let $V_h = \{rect(P_f, P_b) \mid P \in D_h\}$ be the set of rectangles corresponding to the patterns in D_h. Similarly $(w_f[i,j], w_b[i,j])$ is also mapped into a rectangle. A pattern P is matched at some location in the subrow $T[i,j]$ through $T[i, j + 2^g - 1]$ if and only if it has an avatar (P_f, P_b) such that $(P_f, P_b) \leq_s (w_f[i,j], w_b[i,j])$. It follows from the rectangle-mapping of (F_h, B_h) that $rect(P_f, P_b)$ encloses $rect(w_f[i,j], w_b[i,j])$ by Lemma 6.2. To find all the matches with their top row in text row i and j as their rightmost power column, we perform a query of reporting *all* enclosing rectangles of $rect(w_f[i,j], w_b[i,j])$ in V_h. □

We now express the time and space required to preprocess patterns of height h in $\mathcal{P}_g$ as a function of the time and space needed for RER and NRES.

> **Lemma 6.3** Let n_h be the number of patterns in D_h, and let d_h be the total size of all patterns in D_h. Suppose the preprocessing time and space bounds for either RER or NRES with m rectangles are $O(m\ T_{rect}(m))$ and $O(m\ S_{rect}(m))$ respectively. Given C, we can build $F_h, B_h, V_h^f, V_h^b, V_h$ in time $O(d_h \log n_h + h\ n_h\ T_{rect}(h\ n_h))$, and space $O(d_h + h\ n_h\ S_{rect}(h\ n_h))$.

Proof Consider a pattern P of height h in D_h. Let $(P_f^1, P_b^1), \ldots, (P_f^{2^g}, P_b^{2^g})$ be the avatars of P, such that $P_f^1 = P$. From the definition of an avatar, P_f^i is a prefix of P_f^{i-1} for $1 < i \leq 2^g$. Similarly P_b^i is a prefix of P_b^{i+1} for $1 \leq i < 2^g$. Since any dictionary automaton containing a pattern P contains all the prefixes of P, there is no penalty in terms of time and space to include a prefix of P as a new pattern. Following this reasoning, the avatars of P can be added in F_h or B_h without any extra cost. Therefore the resource bounds for building F_h and B_h are $O(d_h \log(n_h + \sigma))$ time and $O(d_h)$ space respectively which follow from the bounds of the AC algorithm. The reason the time is $O(d_h \log n_h)$ is that in AC goto tree, each state can have only n_h outgoing edges.

Let us bound the number of rectangles in V_h^f, V_h^b, and V_h. The number of avatars of P are 2^g. Since $2^g < h$, the total number of avatars is bounded by the sum of heights of the patterns in D_h which is $h\ n_h$. Since the size of the image of any mapping is at most equal to the size of the domain of the mapping, it follows that the number of rectangles generated in any reduction is at most equal to the number of avatars. The time and space bounds to reduce each step to either RER or NRES follow. □

We are now ready to present the pseudocode for the preprocessing of patterns:

Algorithm 10 Code for preprocessing patterns.

PREPROCESS($\mathcal{P}_g$)
NOTE $\forall p \in \mathcal{P}_g$, $height(p) \leq width(p)$ and $2^g < height(p) \leq 2^{g+1}$
1 Build the dictionary C of columns of height 2^g
2 Build $H^{f,u}, H^{f,l}, H^{b,u}, H^{b,l}$
3 **for each** h, $2^g < h \leq 2^{g+1}$
4   Build the dictionary of columns of height h
5   Let $D_h \subseteq \mathcal{P}_g$ be the set of patterns with height h
6   Assume w.l.o.g. that D_h is non-empty. Build D_h
7   **for each** pattern $P \in D_h$
8   **for each** avatar (P_f, P_b) of P
9   Add $rect(P_f^u, P_f^l)$ to V_h^f
10   Add $rect(P_b^u, P_b^l)$ to V_h^b
11   Add $rect(P_f, P_b)$ to V_h
12   Build the data structure needed to solve NRES on V_h^f and V_h^b
13   Build the data structure needed to solve RER on V_h

We are now ready to prove the time and space bounds for the preprocessing of patterns.

Theorem 6.3 Let p_g denote the total size of all patterns in $\mathcal{P}_g$. Suppose the preprocessing time and space for either RER or NRES with m rectangles are $O(m\ T_{rect}(m))$ and $O(m\ S_{rect}(m))$ respectively. Let $\pi_g = \sum_h(h\ n_h)$ be the total number of avatars generated by patterns in $\mathcal{P}_g$. We can preprocess $\mathcal{P}_g$ in time $O(p_g \log(n+\sigma) + \pi_g\ T_{rect}(\pi_g))$, and space $O(p_g + \pi_g\ S_{rect}(\pi_g))$.

Proof Using analysis similar to that in the previous proof, it follows that the dictionaries C, $H^{f,u}$, $H^{f,l}$, $H^{b,u}$, and $H^{b,l}$ can be built in $O(p_g \log(n+\sigma))$ time and $O(p_g)$ space using the AC algorithm. From Lemma 6.3, steps 3–13 take time $O(\sum_h(h\ n_h\ T_{rect}(h\ n_h)))$. From this we can conclude that steps 1–13 take time $O(p_g \log(n+\sigma) + \pi_g\ T_{rect}(\pi_g))$. Similarly we can prove that steps 1–13 take space $O(p_g + \pi_g\ S_{rect}(\pi_g))$. Therefore the preprocessing time and space bounds follow. $\square$

6.4 Preprocessing and Searching the Text

In this section we give some more details of the searching algorithm outlined in the previous section, summarize the searching algorithm in pseudocode, and prove the time bounds we claimed in Section 6.1.

The pseudocode for the searching algorithm is given below. We discuss three details of it that are not apparent from the description in Section 6.3.

First, our use of the AC algorithm to report all half-pattern columns occurring in a text column is a little unusual. We use the AC algorithm searching down each column seeking to report at each location $T[i,j]$ whether a half-pattern column matches starting at $T[i,j]$ and if so, which column matches. Normally, the AC algorithm records its matches at the end of the match, where they are detected. In our application, each half pattern column is of height 2^g. Thus there can be at most one match ending (starting) at a given position in a text column and the length of the match is known. Therefore, we can modify the AC algorithm to record the column matches at the starting position, by subtracting $2^g - 1$ from the ending position.

Second, although it is possible to use our geometric approach for any distribution of patterns, we get better bounds *with linear space* by using the Bird-Baker algorithm for small pattern heights. Specifically, we define a height $h_{\min}$ such that for patterns of height $\leq h_{\min}$ we use the Bird-Baker algorithm for each separate height. This takes time $O(h_{\min}\, t \log(n + \sigma) + tocc)$. We choose suitable values of $h_{\min}$ later.

Third, from the description of the algorithm it may appear that we use $O(t)$ extra space to store the arrays T_c, FU, FL, BU, BL because they store one item per text character. There is a standard trick to keep the space to $O(d)$, when $d \ll t$. Split the text into overlapping patches of size $2d \times 2d$ and run the algorithm separately on each patch. Each text position occurs in at most 4 patches.

Example 6.2 We use the following section of a text T to explain the searching process. We use the dictionary of Example 6.1 for the purpose. Later we show how we match patterns for locations $T[i,j]$ through $T[i,j+1]$. Here j is a power column.

	$j-3$	$j-2$	$j-1$	j	$j+1$
i	a	b	c	a	b
$i+1$	c	a	a	b	c
$i+2$	a	b	b	a	b

We are now ready to give the pseudocode for the searching algorithm:

Algorithm 11 Code for searching a text with $\mathcal{P}_g$.

SEARCH(T)

0 Suppose $g > h_{\min}$. Otherwise run Bird-Baker algorithm directly
1 Search each column of T top-to-bottom with C, giving T_c
2 Search each row of T left-to-right with $H^{f,u}$, giving FU
3 Search each row of T left-to-right with $H^{f,l}$, giving FL
4 Search each row of T right-to-left with $H^{b,u}$, giving BU
5 Search each row of T right-to-left with $H^{b,l}$, giving BL
6 **for each** h, $2^g < h \leq 2^{g+1}$
7   **for each** power column j (i.e. j mod $2^g = 0$) of T
8   Set $w_f[i,j]$ to the smallest rectangle in V_h^f
                enclosing $rect(FU[i,j],\ FL[i+h-2^g,j])$
9   Set $w_b[i,j]$ to the smallest rectangle in V_h^b
                enclosing $rect(BU[i,j],\ BL[i+h-2^g,j])$
10   Let $M \in V_h$ be the set of rectangles enclosing $rect(w_f[i,j], w_b[i,j])$
11   Report the corresponding patterns of M

We now illustrate the searching algorithm, under the assumption that $h_{\min} = 0$, using Example 6.2. Since $g = 1$ in this case, we are operating in the range $2 < h \leq 4$. We show how we find all the matching patterns for the subrow $T[i,j]$ through $T[i,j+1]$ with a *single* query at location $T[i,j]$.

After step 5, we have $FU[i,j] = ac|ba|ca|ab$, $FL[i+1,j] = ca|ab|ab|ba$, $BU[i,j] = bc|ab$, and $BL[i,j] = cb|ba$. At step 6, we deal with the case $h = 3$ since the patterns in Example 6.1 are of height 3. At steps 8 and 9, we set $w_f[i,j] = aca|bab|cab|aba$, and $w_b[i,j] = bcb|aba$. Finally at steps 10 and 11, we report the actual matches as follows. The first pattern $aca|bab|cab|aba$ is matched at location $T[i,j]$ since it has an avatar $(aca|bab|cab|aba, aba)$ dominated by $(w_f[i,j], w_b[i,j])$. From Theorem 6.2, it follows that the rectangle of this avatar is in M at step 10. Similarly the second pattern $bab|cab|aba|bcb$ is matched at location $T[i,j+1]$ since it has an avatar $(bab|cab|aba, bcb|aba)$ dominated by $(w_f[i,j], w_b[i,j])$. Accordingly, its rectangle is also in M at step 10.

The correctness of the algorithm follows from Section 6.3. It remains to prove the running time bounds and make sure that we use only $O(d)$ space. We first state two important results from previous papers.

Lemma 6.4 [54, 35] The RER and NRES problems can be reduced to a 4-dimensional range searching problem.

Proof Let each rectangle R have extreme x values $(x_{\min}(R), x_{\max}(R))$ and extreme y values $(y_{\min}(R), y_{\max}(R))$. Then rectangle R encloses the query rectangle Q if and only if the following conditions hold.

$$x_{\min}(R) \leq x_{\min}(Q)$$

$$x_{\max}(R) \geq x_{\max}(Q)$$

$$y_{\min}(R) \leq y_{\min}(Q)$$

$$y_{\max}(R) \geq y_{\max}(Q).$$

Thus if we represent each rectangle by its four extreme coordinates, we seek in the RER problem to report all rectangles R that satisfy the four constraints above with respect to Q. To solve $NRES$ we seek the enclosing rectangle with maximum $x_{\min}$ and $y_{\min}$. □

In fact, the problems are equivalent in some sense [35]. Those readers familiar with range searching will notice that in the above reduction the four constraint intervals for Q are unbounded on one side. This enables us to use a result of Gabow-Bentley-Tarjan [37] who already noted its applicability to the counting variant of RER.

Lemma 6.5 [37] There is a data structure that enables us to store m rectangles in two dimensions with $S_{rect} = T_{rect} = O(\log^2 m)$ and answer RER queries in time $O(\log^2 m + nrect)$, where $nrect$ is the number of rectangles reported. For the NRES problem, the additive term $nrect$ is dropped.

We now give our first result on time bounds:

Theorem 6.4 We can solve the 2MPM problem using $O(d)$ space and times:

Preprocessing: $O(d \log(n + \sigma))$.

Text Searching: $O(t \log^2 d \log(B + n + \sigma) + tocc)$.

Proof We analyze the algorithm SEARCH. The multiplicative factor of $\log(B+n+\sigma)$ includes the $\log B$ term for the different sizes when we use our algorithm and the $\log(n+\sigma)$ term from the Bird-Baker algorithm. There is an extra multiplicative factor of 2 to handle the symmetric case where width $<$ height.

Regardless of whether the dictionary is size-diverse or not, steps 1–5 take time $O(t \log d)$ using the AC algorithm. Note that this is counted as part of the searching time, not the dictionary preprocessing time.

We take $h_{\min} = \log^2 d$. This means that the use of the Bird-Baker algorithm in step 0 (over all heights) takes time $O(t \log^2 d \log(n+\sigma) + tocc)$.

Since there are 2^g possible heights h and and $\lfloor t/2^g \rfloor$ power columns, the number of instances of RER and NRES queries that we need to solve in the loop at steps 6–11 is $O(t)$. Since each query takes $O(\log^2 d)$ time by Lemma 6.5, the searching time bound follows.

We now prove the time and space bounds for the preprocessing. Recall from the last section that the number of avatars inserted is π_g, which is also the number of points inserted in any geometric structure. From Lemma 6.5, the space and time used by the Gabow-Bentley-Tarjan structures is $O(m \log^2 m)$ for m patterns. To show that this amount is $O(d)$, it suffices to show that $\pi_g = O(d/\log^2 d)$. The key point is that we do not use the avatars of short patterns with height below $h_{\min}$. Thus $h \geq h_{\min} \geq \log^2 d$ by our choice. Therefore the following inequalities hold:

$$\pi_g \leq \sum_{h \geq h_{\min}} (h\ n_h) \leq (\sum_{h \geq h_{min}} (h^2\ n_h))/h_{\min} = d/\log^2 d.$$

From the above argument and Theorem 6.3, the time and space bounds for the preprocessing follow. $\square$

We now give our second result on time bounds:

Theorem 6.5 For size-diverse dictionaries we can improve the bounds to:

Preprocessing: $O(d \log(n+\sigma))$.

Text Searching: $O(t \log d \log(B+n+\sigma) + tocc)$.

Proof In the case where the patterns are size-diverse, we replace the Gabow-Bentley-Tarjan structure for range searching with a data structure of Bentley and

Maurer [22] that achieves query time $O(\log m)$ at the expense of having $S_{rect} = T_{rect} = O(m^{1+\epsilon})$, for $\epsilon > 0$. In the size-diverse case we use $h_{\min} = \log d$. The improved searching time bound follows as in the general case. To prove the space and preprocessing time bounds, it suffices to show that if the patterns are size-diverse, $\pi_g^{1+\epsilon} = O(d)$.

We choose ϵ small enough, so that the patterns are size-diverse with exponent $k = (1-\epsilon)/\epsilon$. By the definition of size-diverse, we have $n_h \leq h^{(1-\epsilon)/\epsilon}$ and $n_h^\epsilon \leq h^{1-\epsilon}$. Refining the above inequalities one more time, we get:

$$(\pi_g)^{(1+\epsilon)} \leq \sum_{h \geq h_{min}} (h\ n_h)^{(1+\epsilon)} \leq \sum_{h \geq h_{min}} (h^{1+\epsilon}\ h^{1-\epsilon}\ n_h) = \sum_{h \geq h_{min}} (h^2\ n_h) = O(d).$$

□

6.5 Dynamic Dictionary Matching of Rectangular Patterns

One of the themes of this thesis is to combine different pattern matching paradigms such as multiple matching and two dimensional matching. Thus, it is interesting to ask: to what extent can our algorithm for multiple matching of rectangular patterns be extended to encompass other paradigms? In this section we extend our algorithm to further combine it with the dynamic dictionary matching paradigm.

We will describe a modification and augmentation of the approach in Sections 6.3 and 6.4 that proves:

Theorem 6.6 Dynamic dictionary matching of rectangular patterns in a rectangular text can be solved in the following time bounds:

Preprocessing: $O(d \log^4 d)$.

Insertion/Deletion: $O(p \log^4 d)$, where p is the area of the pattern.

Text Searching: $O(t \log^4 d \log B + tocc)$.

Our dynamic algorithm retains most of the structure of the static algorithm presented in Sections 6.3 and 6.4. We summarize the features of the dynamic algorithm shared by the static algorithm along with the modifications.

Firstly, the pattern representation and the searching algorithm for the dynamic case are very similar to the static case. We maintain Bird-Baker dictionary structures on the top of the patterns, one for each distinct height. We use them to run separately for each height below a threshold height h_{min} as in the static case. Of course, these dictionary structures are now dynamic. Similarly the columns of patterns are inserted into a one dimensional dynamic dictionary using the IS algorithm. This also means that, we modify the IS algorithm to report the beginning of the matches rather than the end of the matches, as we modified the AC algorithm in Section 6.4.

Secondly, we maintain the Euler tour information for the pattern trees as done in the static case. This information is needed to map pattern prefixes to rectangles using the reduction described in Section 6.3. However, we cannot use the rank order of the associated parentheses because the rank of a parenthesis keeps changing in a dynamic setting. We therefore assign a *different* set of numbers to the parentheses such that the sorted order of these numbers is the same as the rank order of the parentheses. To achieve this, we build a DS list, described in Section 2.4, on the top of the parentheses in the rank order. Since a DS list assigns labels to the parentheses such that the sorted order of the labels is the same as the rank order, we can use these labels to maintain the Euler tour information.

Thirdly, as one might expect, the avatars need to be made dynamic. This means that we insert avatars as points into a dynamic range searching scheme. To make the range searching dynamic we use a data structure of Willard and Lueker [66] instead of either the Bentley-Maurer structure or the Gabow-Bentley-Tarjan structure that were preferable for the static case. The Bentley-Maurer structure cannot be easily used because the validity of the assumption that the dictionary is size-diverse may change over time. The Gabow-Bentley-Tarjan structure cannot be used because it relies on fast processing of least-common-ancestor queries in a static tree.

The following fact follows from using the data structure of Willard-Lueker:

> **Fact 6.4** The Willard-Lueker data structure on d points uses $O(d \log^3 d)$ space, and allows us to solve RER in time $O(\log^4 d + nrect)$ and NRES in time $O(\log^4 d)$. Insertions and deletions take time $O(\log^4 d)$.

Finally, we need to use a special technique in the spirit of [61] to keep the space $O(d)$, as d changes. These details will be explained later.

Everything else in the text searching algorithm stays the same, except that we may alter $h_{\min}$ (the height at which we switch from Bird-Baker to our algorithm) to reduce the space requirement. Recall that in the static case we set $h_{\min} = \log^2 d$, so that the space required by the Gabow-Bentley-Tarjan range searching structure would be $O(d)$. Using the Willard-Lueker data structure, the (tempting) natural analog is to put $h_{\min} = \log^3 d$, but d is changing over time. In fact, we will make $h_{\min} = O(\log^3 d)$ at all times but the exact value of $h_{\min}$ might vary. With this choice, the use of the Bird-Baker algorithm takes time $O(t \log^4 d + tocc)$. The $\log n$ term from the static time is replaced by $\log d$ in the dynamic case because we are using *dynamic* one dimensional dictionaries. We call the data structure that stores the avatars of patterns of height h the *Willard-Lueker structure* for height h.

In the remainder of the section, we describe our insertion and deletion algorithms. Assume for the rest of the section that all logarithms are base 2. At any given time define $c = 2^{\lceil \log \log d \rceil}$. That is, c is a power of 2 and d is in the range $[2^c, 2^{2c})$. Assume also that $d \geq 4$, and therefore $c \geq 2$. We define a *threshold* point for each range, which is set to 2^{2c-1} for the range $[2^c, 2^{2c})$. Now classify possible pattern heights by:

- Lower Domain: $[1, c^3]$
- Middle Domain: $(c^3, 8c^3]$
- Upper Domain: $(8c^3, \infty)$

The main motivation for defining the above domains is to keep the overall space linear. We build Willard-Lueker structures *completely* for the upper domain. We *do not* build them for the lower domain. We keep the structures *partially* built for the middle domain. The middle domain serves as a buffer for making smooth transitions from one range to the other.

Our idea is to keep the Willard-Lueker structures for the middle range *partially* built, making progress towards completion whenever a pattern deletion occurs, and towards depletion whenever a pattern insertion occurs. By "partially built" we mean that for some patterns, all the avatars have been inserted as points in the structure, for some patterns no avatars have been inserted, and for at most one pattern some avatars are in and others are out. When the dictionary size is above the threshold, we keep deleting the avatars of the patterns in the middle domain in anticipation for a shift from the present range to the next range. We cannot afford to use any space for Willard-Lueker structures for the avatars in the current middle domain once we shift as they will belong to the lower domain in the next range. Similarly when the dictionary size is below the threshold, we keep inserting the avatars of the patterns in the middle domain.

We maintain the following invariants in our insertion/deletion algorithms:

1. For every h, all patterns of height h are kept in a Bird-Baker dictionary for height h, which is used in searching for heights that are smaller than the current value of $h_{\min}$.

2. For every h, all patterns of height h in the upper domain are kept in a Willard-Lueker structure for height h.

3. No patterns of heights in the lower domain are kept in a Willard-Lueker structure.

4. $h_{\min}$ is the largest height in the middle domain; i.e., $8c^3 = \Theta(\log^3 d)$.

As a result of invariants 3 and 4, we can show similarly to Theorem 6.4 that the amount of space needed for the Willard-Lueker structures is $O(d)$ (because they only store information for patterns whose height is $\Omega(\log^3 d)$). A key point is that although we cannot afford the space to keep the Willard-Lueker structures for heights in the lower domain, we *can* afford the space for the middle domain.

We also assume, without loss of generality, that no single pattern is so big that its insertion or deletion causes c to change by more than a factor of 2. If a pattern is big enough so that its insertion or deletion causes c to change by more than a factor of 2, then its area A must be $\Omega(d)$. Therefore, we can rebuild the entire dictionary from scratch, having known the final value of d in advance. The cost of doing so can be paid by the pattern being inserted or deleted.

For reasons explained later, we define L as a *queue* of avatars of patterns in the middle domain which are not inserted as points into any Willard-Lueker structure. In the following algorithm description we do not distinguish between a pattern avatar (P_f, P_b), the rectangle that the avatar is mapped to, and the 4-dimensional point that describes the extreme coordinates of that rectangle.

We now give the pseudocode for insertion and deletion of a pattern:

Algorithm 12 Pseudocode for inserting a pattern.

INSERT(P)
0 Let h and A be the height and area of P respectively
1 $d \leftarrow d + A$
2 **if** $d \geq 2^{2c}$ **then**
3 $\quad c \leftarrow 2c$

Set $h_{\mathsf{min}} \leftarrow 8c^3$, and change the three domains accordingly
Insert P into D_h
for each avatar (P_f, P_b) of P
    Insert $P_f^u, P_f^l, P_b^u, P_b^l$ into $H^{f,u}, H^{f,l}, H^{b,u}, H^{b,l}$
    if P is in upper domain **then** add (P_f, P_b) to Willard-Lueker structure
    if P is in middle domain **then** add (P_f, P_b) to L.
if $d \geq 2^{2c-1}$ (threshold) **then**
    Delete up to $2A$ avatars of patterns in middle domain from
        Willard-Lueker structures and add them to L

Algorithm 13 Pseudocode for deleting a pattern.

DELETE(P)
Let h and A be the height and area of P respectively
$d \leftarrow d - A$
if $d < 2^c$ **then**
    $c \leftarrow c/2$
    Set $h_{\mathsf{min}} \leftarrow 8c^3$, and change the three domains accordingly
Delete P from D_h
for each avatar (P_f, P_b) of P
    Delete $P_f^u, P_f^l, P_b^u, P_b^l$ from $H^{f,u}, H^{f,l}, H^{b,u}, H^{b,l}$
    Delete the avatars of P (if any) from Willard-Lueker structure and L
if $d < 2^{2c-1}$ (threshold) **then**
    Insert up to $2A$ avatars of patterns in L into
        Willard-Lueker structures and delete them from L

The above invariants imply that the search and update bounds in Theorem 6.6 can be achieved using the insertion and deletion algorithms above, as long as the domain boundaries stay the same. If the domain boundaries change, this causes $h_{\min}$ to change. We consider separately the cases where the shift in boundaries is caused by an increase or decrease in d.

An insertion of a pattern can cause the dictionary size to increase, so that the new lower domain becomes $[1, 8c^3]$, the new middle domain becomes $(8c^3, 64c^3]$, and the new upper domain becomes $(64c^3, \infty)$. Invariants 1 and 2 are preserved by the insertion algorithm. To preserve invariant 4, we change $h_{\min}$ to $64c^3$. To preserve invariant 3, we must have deleted all Willard-Lueker structures for heights $(c^3, 8c^3]$, which used to be the middle domain and is now part of the lower domain. Note that we keep the Willard-Lueker structures for the new middle domain; this is important because we will need them immediately if a pattern deletion causes the domains to shift back to previous values.

A deletion of a pattern can cause the dictionary size to decrease, so the new lower domain becomes $[1, c^3/8]$, the new middle domain becomes $(c^3/8, c^3]$, and the new upper domain becomes (c^3, ∞). Again, invariant 1 is preserved by the deletion algorithm, invariant 3 is preserved because the lower domain decreases in size, and we preserve invariant 4 by adjusting $h_{\min}$ to c^3. To preserve invariant 2, we must have Willard-Lueker structures ready for heights $(c^3, 8c^3]$, which used to be the middle domain and is now part of the upper domain. Note that we do not have any Willard-Lueker structures for the new middle domain; this is important because if a pattern insertion causes the domains to shift back to previous values, this domain will become part of the lower domain.

Our strategy for building the middle domain Willard-Lueker structures is as follows. We do extra work in the insertion (deletion) algorithm, so that whenever a pattern is inserted (deleted) and the dictionary size goes above (below) the threshold value, we also make progress towards deleting (inserting) from (into) the Willard-Lueker structures the avatars of middle domain patterns in L. To delete (insert) a pattern of area A, we are allowing time $O(A \log^4 d)$. Recall that each pattern of area A has at most A avatars. To delete (insert) these avatars from (into) the Willard-Lueker structures also takes time $O(A \log^4 d)$. Therefore, when a pattern of area A is inserted (deleted) and the dictionary size goes above (below) the threshold, we delete (insert) up to $2A$ avatars of patterns in L with no asymptotic time penalty.

The following lemma shows that invariant 2 can be preserved under deletions.

Lemma 6.6 Suppose we use the above strategy of rebuilding during insertions and deletions. When a deletion causes the middle domain to drop from $(c^3, 8c^3]$ to $(c^3/8, c^3]$, there will be no more avatars of patterns in L with heights in the old middle domain after that deletion is complete. That is, the Willard-Lueker structures for heights $(c^3, 8c^3]$ will be complete when $(c^3, 8c^3]$ is shifted to the upper domain.

Proof Suppose that the dictionary size never exceeded the threshold for the old range. Whenever we enter a range from the range below it (as a result of some insertion), the Willard-Lueker structures for its middle domain will be fully built because its middle domain was part of the upper domain for the range below it. We could not have deleted any avatars from the Willard-Lueker structures during the insertions of patterns because the dictionary size never exceeded the threshold. Therefore, its L list will continue to be empty as desired.

Suppose that the dictionary size exceeded the threshold for the old range. Consider the last time α that the dictionary size is above the threshold for the old range, and let $d \geq 2^{2c-1}$ be the size of the dictionary at time α. Let d' be the size of the dictionary just after the middle domain drops to $(c^3/8, c^3]$ at time β. By the definitions of c and threshold, and by construction of the domains, $d \geq 2d'$. Let $S = x + y$ be the total area of patterns that are both inserted (with total area x) and deleted (with total area y) during the period $[\alpha, \beta]$. Thus the total area of the deleted patterns during this period is $y = d + x - d'$, which is at least $x + d'$. During those deletions we could insert into the Willard-Lueker structures all avatars of patterns whose areas total $2(x + d')$. During this period we could have deleted at most $2x$ avatars from the Willard-Lueker structures because of insertions of patterns. Therefore the amount we need to insert is at most $d' + 2x \leq 2(x + d')$. □

Finally we show that invariant 3 can be preserved under insertions. This completes the proof of Theorem 6.6.

Lemma 6.7 Suppose we use the above strategy of rebuilding during insertions and deletions. When an insertion causes the middle domain to rise from $(c^3, 8c^3]$ to $(8c^3, 64c^3]$, the Willard-Lueker structures for heights $(c^3, 8c^3]$ will be completely deleted when $(c^3, 8c^3]$ is shifted to the lower domain.

Proof Let α be the last time that the dictionary size is below the threshold for the old range, and let $d \geq 2^{2c-1}$ be the size of the dictionary at time α. Let d' be the size of the dictionary just after the middle domain rises to $(8c^3, 64c^3]$ at time β. By the definitions of c and threshold, and by construction of the domains, $d' \geq 2d$. Let $S = x + y$ be the total area of patterns that are both inserted (with total area x) and deleted (with total area y) during the period $[\alpha, \beta]$. Thus the total area of the inserted patterns during this period is $x = d' + y - d$, which is at least $y + d$.

During those insertions we could delete from the Willard-Lueker structures all avatars of patterns whose areas total $2(y+d)$. During this period we could have inserted at most $2y$ avatars into the Willard-Lueker structures because of deletions of patterns. Therefore the amount we need to delete is at most $d + 2y \leq 2(y+d)$. $\square$

Chapter 7

Conclusions

Pattern matching is currently a very active area of research, as can be evidenced by the number of recent papers in the literature. In the previous chapters, we have seen the details of the results we have obtained in this area. In this chapter, we identify three directions of research in pattern matching, discuss our results in each direction, and provide suggestions for future research.

Combining Subproblems One of the roles of theoretical computer science is to develop an algorithmic theory for various application domains. One approach to developing such a theory is by using a divide-and-conquer approach. We divide an application problem into several abstract subproblems such that each subproblem can be solved independently. We then combine the solutions to these subproblems to produce a solution to the original problem.

Using the above approach, we can divide the pattern matching domain, in general, into the following subproblems. The list is only suggestive of the nature of difficulties involved, and is by no means complete. For each subproblem, we give some situations where it occurs.

Local Errors: These can be caused by typing errors, noise in transmission of data, digitization of pictures, and so on.

Scaling: The relative size of the image of an object changes depending on the distance or angle from which it is viewed. Similarly, the sizes of letters in a script vary depending on the font.

Compression: In many applications, the data is compressed to save space and communication costs. It will be desirable for a pattern matching problem to work on the compressed data itself.

Rotation: The orientation of a camera in relation to an object may cause the image of the object to rotate.

Multiple Patterns: Very often, the number of patterns will be more than one, as in a bibliographic search.

Multiple Dimensions: The dimensionality of patterns and texts varies from application to application. The dimensionality is one in a DNA sequencing problem, and two in an image processing application.

Parameterization: In software development projects, a piece of code may be duplicated with renaming of variables, which is also known as parameterization.

In this thesis, we have developed an algorithmic framework for the multiple pattern matching problem. We have extended this scheme in two important directions. First, we have combined it with the multiple dimensions problem, by providing solutions to the multiple pattern matching problem in two dimensions. Second, we have made the problem dynamic, by allowing insertions and deletions of patterns.

There has been a lot of progress in combining several subproblems. Amir and Landau provided a fast solution for approximate matching of multidimensional arrays [15], thus combining local errors and multiple dimensions. Amir, Landau, and Vishkin solved the scaled matching of arrays [16], combining scaling and multiple dimensions. Recently, Amir and Calinescu solved the scaled dictionary matching problem [8], combining scaling, multiple patterns, and multiple dimensions. Amir, Benson, and Farach provided an optimal solution to a restricted form of compressed matching of arrays [7], combining compression and multiple dimensions. In addition to these, parallel and/or alphabet-independent solutions have provided to some of these problems [5, 40, 8, 7, 30, 6, 59].

Recently, Baker defined and provided efficient solution to the problem of parameterized matching [20]. At present we are making progress on parameterized matching with multiple patterns [49].

There are several open problems in this direction. There is still nothing interesting known about matching with rotation. One can work on combining several combinations of subproblems as shown above. An important combination from a practical viewpoint is multiple patterns with local errors, which has applications in computational biology. Alternatively, one can define new important subproblems as done by Baker above, and provide efficient solutions to them.

Relating to Other Problems We have shown in Chapter 3 that the problem of maintaining the $fail$ function in a dynamic dictionary can be reduced to the problem of maintaining a dynamic set of balanced parentheses. This means that any better algorithm for the parenthesis problem can be directly used to improve the algorithm for the dynamic dictionary problem, as demonstrated in [13]. Similarly, we have reduced the 2MPM problem to the problem of rectangle enclosure searching problem in computational geometry. Any improved solutions for rectangle enclosure searching problem can be readily used to provide a better algorithm for the 2MPM problem.

Other examples of such reductions in the literature include the paper by Fischer and Paterson [36], where they reduced the FPM problem to convolutions on two vectors. As a side benefit, they showed that the FPM problem with wild-card characters can be solved with the same reduction to convolutions, with the same time bounds. Another example can be found in the paper by Amir, Landau, and Vishkin [16], where they reduced the *longest common prefix* query of two substrings to a *lowest common ancestor* query of two nodes in a tree. This reduction has been used in other pattern matching problems to obtain efficient solutions.

Relating a problem to other problems is as important as solving the problem. This has two advantages. First, it provides a basis for unification of several problems. Second, solutions for one problem can be used to obtain solutions for the other problems. We believe that there is a lot of scope for doing research in this direction.

Implementing Algorithms The area of pattern matching has applications to several real world problems such as text editing, information retrieval, DNA sequencing, and computer vision. As we have seen so far, several algorithms have been developed for solving different varieties of the pattern matching problems. Applying these algorithms to real world problems poses formidable implementation challenges.

Many solutions to the FPM problem, such as the KMP algorithm [52], Boyer-Moore algorithm [25], and Karp-Rabin algorithm [51], have been directly implemented and used in several applications. Also, there are implementations which make some adjustments to these original algorithms [17, 47, 18]. In addition to these, there are other implementations which use a combination of these algorithms. In two dimensions for example, the 2FPM problem has two practical implementations. Zhu and Takaoka [67] used a combination of the Karp-Rabin algorithm and the KMP algorithm (or Boyer-Moore algorithm) to implement a solution for the 2FPM problem. Later, Baeza-Yates and Régnier [19] used a combination of the AC algorithm, the Boyer-Moore algorithm, and some other ideas to implement a solution for the 2FPM problem that runs better than the Zhu-Takaoka algorithm on many inputs. Recently, Baker [20] implemented part of her algorithm for the parameterized pattern matching and tested it on real applications.

We have implemented part of our one dimensional dynamic dictionary matching algorithm, to demonstrate its correctness and practicality, and to use it in future on other pattern matching problems. We intend to use it in our ongoing work on parameterized matching with multiple patterns [49]. As a side benefit, we have also implemented the Dietz-Sleator list as a separate package, which can be used in any other application. We hope that other researchers will use our implementation and benefit from it.

We envision a great potential for research in the implementation of pattern matching algorithms.

Bibliography

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18:333–340, 1975.

[2] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. *Proc. of the First Ann. ACM–SIAM Symp. on Discrete Algorithms*, pages 32–42, 1990.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[4] A. Amir. Multidimensional pattern matching: A survey of serial deterministic algorithms. Technical Report GIT-CC-92/29, Georgia Institute of Technology, 1992.

[5] A. Amir, G. Benson, and M. Farach. Alphabet independent two-dimensional matching. *Proc. of the 24th Ann. ACM Symp. on Theory of Computing*, pages 59–68, 1992.

[6] A. Amir, G. Benson, and M. Farach. Optimal parallel two-dimensional pattern matching. *Proc. of the Fifth Ann. ACM Symp. on Parallel Algorithms and Architectures*, 1993.

[7] A. Amir, G. Benson, and M. Farach. Witness-free dueling: The perfect crime! (or how to match in a compressed two-dimensional array). Manuscript, 1993.

[8] A. Amir and G. Calinescu. Fa{s,t}ter scaled matching. Manuscript, 1993.

[9] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 760–766, 1991.

[10] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of the Second Ann. ACM–SIAM Symp. on Discrete Algorithms*, pages 212–223, 1991.

[11] A. Amir and M. Farach. Two dimensional dictionary matching. *Information Processing Letters*, 1993. To appear.

[12] A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. Manuscript, 1991.

[13] A. Amir, M. Farach, R. M. Idury, J. A. La Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Proc. of the Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 392–401, 1993. Full version available as DIMACS Tech. Report 92-33.

[14] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. In *Proc. of the Third Symp. on Combinatorial Pattern Matching*, pages 262–275, 1992.

[15] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[16] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *J. Algorithms*, 13:2–32, 1992.

[17] R. Baeza-Yates. String searching algorithms. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 10, pages 219–240. Prentice Hall, 1992.

[18] R. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. In *Proc. of the Third Symp. on Combinatorial Pattern Matching, Lecture Notes Comp. Sci. 644*, pages 185–192, 1992.

[19] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching. *Information Processing Letters*, 45:51–57, 1993.

[20] B. S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proc. of the 25th Ann. ACM Symp. on Theory of Computing*, pages 71–80, 1993.

[21] T. P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp.*, 7:533–541, 1978.

[22] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[23] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6:168–170, 1977.

[24] J. Blumer, A. Blumer, E. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[25] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[26] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106:21–60, 1992.

[27] D. Chase. An improvement to bottom-up tree pattern matching. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, 1987.

[28] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[29] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. 6th ICALP*, pages 118–132, 1979.

[30] M. Crochemore, Z. Galil, L. Gasieniec, S. Muthukrishnan, K. Park, H. Ramesh, and W. Rytter. Constant time optimal parallel algorithm for 2d-pattern matching. Manuscript, 1993.

[31] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38:651–675, 1991.

[32] P. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 365–372, 1987. To appear in *J. Comp. and Syst. Sci.*

[33] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comp. and Syst. Sci.*, 38:86–124, 1989.

[34] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 145–150, 1990.

[35] H. Edelsbrunner and M. H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14:124–127, 1982.

[36] M. J. Fischer and M. S. Paterson. String matching and other products. *Complexity of Computation, R. M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[37] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 135–143, 1984.

[38] Z. Galil. Open problems in stringology. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12, pages 1–8. NATO ASI Series F: Computer and System Sciences, 1985.

[39] Z. Galil and G. F. Italiano. A note on set union with arbitrary deunions. *Information Processing Letters*, 37:331–335, 1991.

[40] Z. Galil and K. Park. Truly alphabet-independent two-dimensional pattern matching. *Proc. of the 33rd IEEE Annual Symp. on Foundation of Computer Science*, pages 247–256, 1992.

[41] Z. Galil and J. I. Seiferas. Time-space optimal string matching. *J. Comp. and Syst. Sci.*, 26:280–294, 1983.

[42] R. Giancarlo. The suffix tree of a square matrix with applications. *Proc. of the Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 402–411, 1993.

[43] G. Gonnet. Efficient two-dimensional searching. *Proc. of the 3rd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 621*, page 317, 1992. Abstract of an invited talk.

[44] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd edition, 1991.

[45] T. Granlund. Gnu mp: The gnu multiple precision arithmetic library, edition 1.2. Manual, December 1991.

[46] R. H. Güting and D. Wood. The parenthesis tree. *Information Sciences*, 27:151–162, 1982.

[47] R. N. Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10:501–506, 1980.

[48] R. M. Idury and A. A. Schäffer. Dynamic dictionary matching with failure functions. In *Proc. of the Third Symp. on Combinatorial Pattern Matching, Lecture Notes Comp. Sci. 644*, pages 276–287, 1992.

[49] R. M. Idury and A. A. Schäffer. Multiple matching of parameterized patterns. In Preparation, 1993.

[50] R. M. Idury and A. A. Schäffer. Multiple matching of rectangular patterns. In *Proc. of the 25th Ann. ACM Symp. on Theory of Computing*, pages 81–90, 1993.

[51] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Res. and Dev.*, pages 249–260, 1987.

[52] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[53] S. R. Kosaraju. Efficient tree pattern matching. *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 178–183, 1989.

[54] D. T. Lee and C. K. Wong. Finding intersections of rectangles by range search. *J. Algorithms*, 2:337–347, 1981.

[55] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, 1991.

[56] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *Proc. of the First Ann. ACM–SIAM Symp. on Discrete Algorithms*, pages 319–327, 1990.

[57] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[58] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984.

[59] S. Muthukrishnan and K. Palem. Highly efficient dictionary matching in parallel. Manuscript, 1992.

[60] E. W. Myers. An overview of sequence comparison algorithms in molecular biology. Technical Report 91-29, Department of Computer Science, Univ. Arizona, 1991.

[61] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag Lecture Notes in Computer Science 156, 1983.

[62] D. Sleator. Personal Communication, 1992.

[63] P. Weiner. Linear pattern matching algorithm. *Proc. of the 14th IEEE Annual Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[64] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proc. of the 14th Ann. ACM Symp. on Theory of Computing*, pages 114–121, 1982.

[65] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *ACM SIGMOD 86*, pages 251–260, 1986.

[66] D. E. Willard and G. S. Lueker. Adding range restriction capabilities to dynamic data structures. *J. ACM*, 32:597–617, 1985.

[67] R. F. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Comm. ACM*, 32:1110–1120, 1989.

Appendix A

DYNDICT: A Dynamic Dictionary System

In this appendix, we show a sample dynamic dictionary matching system that we have built using the interface described in Section 4.3. The program is contained in a file called `dyndict.c`. The file contains the `main()` routine, and two auxiliary routines `printgoto()` and `printfail()`. These routines are used to print the *goto* and $fail$ trees respectively. We also include the header file `is_dict.h`. After compiling the program with other files, we get an executable file `dyndict`.

To use this sample system, simply type `dyndict` at the shell prompt as follows:

% `dyndict`

The program responds as follows:

Welcome to the Sample Dynamic Dictionary Matching System

Implemented by Ramana M. Idury

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >>

The program creates a dictionary initially containing the dummy null pattern. The dictionary can contain up to 128 patterns. The program accepts simple ASCII strings typed from the terminal as patterns or texts. At this stage, we have several options to choose from. We can insert or delete a pattern from the dictionary by typing 1 or 2 respectively. We can search a text by choosing 3. By choosing 4 or 5, we can print the current *goto* or $fail$ tree respectively, in a simple format. Finally, we can quit the system by typing 6.

We now show a simple session. We use sans serif font to show the responses of the program, and *italic* font to show what we type from the terminal. We first show how we insert three patterns `fred`, `rick`, and `red`.

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *1*

Print a pattern string: *fred*

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *1*

Print a pattern string: *rick*

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *1*

Print a pattern string: *red*

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >>

At this stage, the dictionary contains three patterns `fred`, `rick`, and `red`. We now show the *goto* and *fail* trees.

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *4*

```
 f r e d$
 r i c k$
   e d$
```

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *5*

[( ) [[ ]] (( )) ( ) ( ) [ ] (( ))]

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >>

The `printgoto()` routine uses a simple indentation mechanism to print the *goto* tree in a depth first order. In addition, it prints the first child of each node on the same line as the node to reduce the number of lines of output. The `printfail()` routine simply prints the leaves of the bracket tree in the left-to-right order, replacing each prefix with the corresponding parenthesis or bracket. As one can see above, there are four pairs of matching brackets corresponding to the four patterns (including the null pattern) of the dictionary.

We can now search a text for occurrences of all patterns in the dictionary as follows:

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *3*

Print a text string: *alfredrick*

```
a
l
f
r
e
d fred red
r
i
c
k rick
```

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >>

The program prints each text character on a separate line, along with all matched patterns (excluding ϵ) ending at that position. As shown above, two patterns `fred` and `red` are printed on the same line as `d`. As one can see, these two patterns end at that position. The program prints the longest matched pattern first. On the last line of the output, the program prints `rick` as that is the only pattern matched at that location.

We now delete the pattern `fred` as shown below. We also print the *goto* and $fail$ trees to reflect the change.

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *2*

Print a pattern string: *fred*

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *4*

r i c k$

e d$

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *5*

[( ) [ ] ( ) ( ) [ ] ( )]

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >>

Finally, we search the same text `alfredrick` to show the change in matched patterns.

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >> *3*

Print a text string: *alfredrick*

a

l

f

r

e

d red

r

i

c

k rick

Type 1: Insert, 2: Delete, 3: Search, 4: Goto , 5: Fail , 6: Quit >>