[19] 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. To appear in Theor. Comp. Sci.

[20] 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. To appear in Inf. and Comp.

[21] Z. M. Kedem, G. M. Landau, and K. V. Palem. Optimal parallel suffix-prefix matching algorithm and application. *Proc. of the First Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 388–398, 1989.

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

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

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

[25] B. Meyer. Incremental string matching. *Information Processing Letters*, 21:219–227, 1985.

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

[27] D. Sleator and R. Tarjan. A data structure for dynamic trees. *J. Comput. Systems Sci.*, 24:362–391, 1983.

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

[29] J. Westbrook. Fast incremental planarity testing. *Proc. of 19th International Colloquium on Automata Languages and Programming*, pages 342–353, 1992.

[30] 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.

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

References

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

[2] 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.

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

[4] A. Amir and M. Farach. Two dimensional dictionary matching. *Information Processing Letters*, 44:233–239, 1992.

[5] A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Science*, 1991. To appear.

[6] 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.

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

[8] 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.

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

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

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

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

[13] 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.

[14] 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.*

[15] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comp. and Syst. Sci.*, 30:209–221, 1985.

[16] 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.

[17] 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.

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

$CR_1^2; \ldots; CR_j^2; \$\$ = CR_{i-j+1}^1; \ldots; CR_i^1; \$\$$. From the construction of M', it follows that j is the largest index such that *both* $C_1^2; \ldots; C_j^2; \$ = C_{i-j+1}^1; \ldots; C_i^1; \$$ and $R_1^2; \ldots; R_j^2; \$ = R_{i-j+1}^1; \ldots; R_i^1; \$$ hold. From Lemma 4.7, 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 $fail_M^{\square}(P_1) = P_2$.

($\Longrightarrow$) Suppose $fail_M^{\square}(P_1) = P_2$. 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 4.7, j is the largest index such that $C_1^2; \ldots; C_j^2; \$ = C_{i-j+1}^1; \ldots; C_i^1; \$$ and $R_1^2; \ldots; R_j^2; \$ = R_{i-j+1}^1; \ldots; R_i^1; \$$. From the construction of M', it follows that $CR_1^2; \ldots; CR_j^2; \$\$ = CR_{i-j+1}^1; \ldots; CR_i^1; \$\$$. Therefore $fail_{M'}(P_1') = P_2'$. ∎

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

Algorithm 3 Code for inserting pattern into 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 4 Code for deleting pattern from dictionary.

2D-DELETE($P = a_{ij}, 1 \leq i, j \leq m$)
1 Create the maha-pattern P' by scanning 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 $fail^{\square}$ function for M is implicitly computed as follows: From Theorem 4.8, we can replace the use of $fail^{\square}$ at all places in 2D_SCAN 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 4.9 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 [19]. ∎

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 2.8, 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 4.7 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 m_2 is the largest index such that $U_2[1 \leq i \leq j \leq m_2] = U_1[m_1 - m_2 + 1 \leq i \leq j \leq m_1]$. This implies that $fail^u_M(U_1) = U_2$. ∎

Thus we have found an indirect way of computing $fail^u_M$ by transforming every u-t pattern into an equivalent maha-pattern and computing $fail_{M'}$. We can do a similar procedure for building the automaton for l-t patterns by defining L, L_R, c_{ij} and 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 $CR_i = c_{1i}; r_{i1}; \ldots; c_{ii}; rc_{ii}; \$; \$$ (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 4.7 that:

Theorem 4.8 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

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 4.1, 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 4.1, C^{lo} contains acd, ab, b. The use of these dictionaries will be apparent soon.

As mentioned earlier, a new pattern is inserted one $\perp$-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 [19]. 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 $\perp$-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 [19]. 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 subsection 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 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 detection of matched patterns can be done as in [19]. 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 scanning 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 2 Code for scanning 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 ← λ
        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)
            temp ← goto^□(state, $ − strip) /* Pretend a $-strip is read to check if any patterns match
*/
            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 scanning a rectangular text:

Theorem 4.6 Given M, C^{up}, and R^{lo}, we can scan the text T and report all matched patterns in time $O((t + tocc)(\log(|M| + |R^{lo}| + |C^{up}|))=((t + tocc)(\log d))$ time.

Proof: The first line of the pseudocode takes $O(t \log |C^{up}|)$ time, which follows from Theorem 2.14. 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. ∎

4.3 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 subsection, 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 $\sqcup$-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 $fail^{\square}$.

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.

To facilitate text scanning 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 subsection. We now describe the other two dictionaries which will be used in text scanning.

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 4.1, 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 scanning a text $T[1 \ldots n_r, 1 \ldots n_c]$:

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

2. Scan 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 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 conclude from Theorem 2.14 that:

Corollary 4.4 We can scan 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}. 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.

The second step is scanning using M, which has been outlined in the previous subsection. The pseudocode for the scanning algorithm is given at the end of this subsection. The main components in the scanning 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 subsection. We now show that computing *goto* takes $O(\log d)$ time.

Lemma 4.5 Suppose we are in the state x of M and at the position $T[i,j]$ of the text. Let $\alpha = \langle \alpha_c, \alpha_r \rangle$ be a $\perp$strip in $label(x)$ such that $\alpha =_s \langle C[i+1,j+1], R[i+1,j+1]\rangle$. We can choose α in $O(\log d)$ time.

Proof: We noted at the end of previous subsection 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 2.7 and Lemma 2.8. Therefore we can choose α in $O(\log d)$ time. ∎

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 Subsection 2.2.2. Suppose we are scanning 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 4.2 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 scanning, 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

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

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

Lemma 4.3 There is at most one α that satisfies the condition $\alpha =_s \langle col[i+1, j+1], row[i+1, j+1]\rangle$. Moreover, the set of labels of all the *goto* transitions out of state x, $label(x)$, is totally ordered under the relation $<_{inv}$.

Proof: From Fact 4.2, 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 $\sqcup$ 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 $=_s$ that for any two strings x, y of equal length, $x =_s y$ (or $y =_s x$) implies that $x = y$. Thus between any two different transition labels it is not possible for one to be $=_s$ to the other one. Thus at most one transition label can be $=_s$ 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 $<_{inv}$. ∎

When we scan 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 4.3. Since the labels in the two dimensional case are $\sqcup$strips, there could be $O(d)$ of them. Therefore, the "alphabet" is of size $O(d)$ in this case. With this scheme, finding a suitable label α (if it exists) takes $O(\log d)$ tree comparisons.

4.2 Text Scanning

The scanning algorithm implicitly linearizes the patterns into $\sqcup$strips as explained above. We scan the text along each diagonal, considering the pair of the subcolumn and the subrow ending at each position of

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 $\lrcorner$-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 λ 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 λ is both a prefix and suffix of every square pattern.

The above definitions enable us linearize square patterns into the $\lrcorner$-shaped strips. With this linearization, it is possible to view each square pattern as a *one dimensional* string where each character corresponds to a $\lrcorner$-strip. A new pattern is inserted into the dictionary one $\lrcorner$-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 scanned 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 $(i+1)^{\text{st}}$ strip of R. If there is no such R, then $goto(Q, \langle c, r\rangle)$ is undefined.

We define $fail$ as follows: For any prefix R in M, $fail(R) = Q$, where Q is the largest prefix in M that is a suffix of R. If there is no such Q then $fail(R) = \lambda$. 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 $goto^{\square}$ and $fail^{\square}$ ($\square$ 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 scanning algorithm is similar to the static dictionary matching scanning algorithm of [4] and has the same structure as the AC and IS algorithms (the pseudocode for our scanning algorithm is shown at the end of next subsection). 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 $=_s$ as done in [4]. $=_s$ is defined by:

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

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

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

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

Adding/Deleting P: $O(p(\frac{\log d}{\log\log d} + \log\sigma))$

Scanning text T: $O((t + tocc)(\frac{\log d}{\log\log d}) + t\log\sigma)$

Proof: First, plug in the bounds for dynamic parenthesis maintenance in Lemma 3.36 into the reduction from the nearest marked ancestor problem to parenthesis maintenance given in Lemma 3.5. Then plug the bounds for nearest marked ancestor updates and queries into the reduction of the dynamic dictionary prefix lookup problem to the nearest marked ancestor problem given in Lemma 3.1. This shows that we can solve the dynamic dictionary prefix lookup problem in time bounds:

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

Insert/Delete P: $O(p(\frac{\log d}{\log\log d} + \log\sigma))$

Lookup (j,k): $O(|out_{j,k}|(\frac{\log d}{\log\log d} + \log\sigma))$

Plug these bounds for the dynamic dictionary prefix lookup problem into Lemma 2.4 to complete the proof. ∎

4 Two Dimensional Dynamic Dictionary Matching

In this section, we solve the two dimensional dynamic dictionary matching problem. We are given a dictionary of square patterns $D = \{P_1, P_2, \ldots, P_k\}$, that can change over time. The basic matching operation is to scan 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 first present an overview of the algorithm. 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 scanning and updating algorithms.

4.1 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 scanning 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 [4]. Throughout this section 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 scanning 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 4.1

a	b	c
c	a	a
d	b	b

a	b	c
	a	a
		b

a		
c	a	
d	b	b

constant, but sufficiently large number of nodes that actually belong in the tree, ignoring the already deleted leaves. If we do this sufficiently quickly, we can have the new tree ready after the next $n_{\max}/10$ deletions. When the new tree is ready we start using it and reset $n_{\max}$.

Lemma 3.35 The total time needed to insert or delete a pair of matching parentheses is $O(\log n/\log\alpha + \alpha)$.

Proof: By Lemma 3.32, the time needed to update the tables in a split is $O(\alpha)$. The time needed to do the direct and housekeeping splits is $O(\alpha)$ since there are at most three such splits per insertion and the number of parent-child pointers that need to be changed is $O(\alpha)$ by Lemma 3.34.

The bottleneck is the time needed to compute which node in the upper tree has greatest overflow. We do this using the same method as in [14]. We keep a priority queue for the nodes at each level of the tree, where only nodes that have overflowed are in the queue. At any particular h, a single node has its overflow incremented per leaf insertion, and that increment is by a "unit of" $1/2\alpha^{h(v)-3}$. When we delete the node of maximum overflow, the two resulting nodes have no overflow and are not reinserted in the priority queue. Therefore, we can implement all queue operations in $O(1)$ time [14]. To pick the overall maximum, we pick from the maximum at each level in $O(\log n/\log\alpha)$ time.

Deletion requires updates to the tables as explained in Subsection 3.4.2, and possibly, copying of a constant number of nodes. Therefore the bound of $O(\log n/\log\alpha + \alpha)$ time that we proved before, when we did not worry about the branching factor, still holds. ∎

3.4.4 Time Complexities

The complexities depend on the choice of α. We make the standard assumption that a word can hold $O(\log d)$ bits. This assumption is basic to most papers using suffix trees, including [5], because suffix tree nodes store indices into the string, which is of length d.

By picking α to be $\theta(\frac{\log n}{\log\log n})$, we make sure that NL and NR fit within a word, and have overall time complexities of $O(\log n/\log\log n)$ for both updates and queries.

Lemma 3.36 We can maintain a sequence of d well-balanced parentheses in the time bounds:

Preprocessing: $O(d)$

Insertion/Deletion: $O(\log d/\log\log d)$

Find enclosing parentheses: $O(\log d/\log\log d)$

Proof: Lemma 3.20 provides the time bounds for queries. The preceding discussion in this subsection proves the bounds for updates.

For preprocessing, we first build the initial balanced tree, which can be done in linear time [24]. We then take an Euler tour of the balanced tree to obtain the LCA of every pair of matching parentheses. Then we do a bottom-up traversal to fill in the near cover and far cover bitmaps and the far cover tables. ∎

Our parenthesis maintenance scheme can be applied as in the semi-dynamic case to the one dimensional dynamic dictionary algorithms improving the bounds to:

Theorem 3.37 The one dimensional dynamic dictionary matching problem can be solved within the following time bounds:

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

Proof: For an ancestor node v in the upper tree, the increment in $OF(v)$ is no greater than $1/(2\alpha^{h(v)-3})$. The sum of all such increments is bounded by 1. Apply Theorem 2.1 to the OF values of all upper tree nodes; for those nodes that are created during splits, we treat their OF value as 0 until they are created. The number of upper tree nodes is less than the number of parentheses, n. If we split the node with the greatest overflow each time we insert a new leaf, then Theorem 2.1 implies that no overflow will ever exceed $H_{n-1}+1$. If $n \geq 5$, then $H_{n-1}+1 < 2\log n$. ∎

Lemma 3.34 If $\alpha^3 \geq 8\log n$, then the above rules for splits can be implemented to preserve the following three properties:

1. In any direct or housekeeping split, we can split the node into (just) two nodes that have no overflow.
2. For any internal node v, $L(v)$ is in the range $[\frac{1}{2}\alpha^{h(v)}, \frac{3}{2}\alpha^{h(v)}]$,.
3. The number of children of each internal node is in the range $[\frac{1}{3}\alpha, 3\alpha]$.

Proof: We do the proof inductively assuming that all three properties hold after $j-1$ insertions and show that they can be preserved on the j-th insertion.

Suppose we want to split node w of height h. Since we want to split w, $OF(w) > 0$. Therefore, w has at least α^h leaf descendants. Let $w_1, w_2, \ldots$ be the children of w in left to right order. Since we assumed that α is non-constant, no child of w can account for more than one quarter of the leaf descendants of w. Thus we can begin summing $L(w_1) + L(w_2)\ldots$ and we must reach a j such that the number of leaf descendants from the first j children all together is in the range $[\frac{1}{2}\alpha^h, \alpha^h]$.

Since w has at most $\frac{3}{2}\alpha^h$ leaf descendants, the sum of leaf descendants among all the remaining children is also in the range $[\frac{1}{2}\alpha^h, \alpha^h]$. Thus we can split w into two nodes, where the first node inherits the first j children and the other node inherits the remaining children. Both new nodes will have a number of leaf descendants in the range $[\frac{1}{2}\alpha^h, \alpha^h]$. Thus the two new nodes have no overflow (preserving property 1), and the number of leaf descendants satisfies the lower bound in property 2.

By Lemma 3.33, the maximum value of OF is $2\log n$. A little algebraic manipulation of the definition for OF shows that $L(v) \leq \frac{3}{2}\alpha^{h(v)}$ if $8\log n \leq \alpha^3$. Thus we preserve property 2 for all the nodes that are not split after the j-th insertion.

At height 1 (one level above the leaves), the number of leaf descendants is the same as the number of children. We proved above that the number of leaf descendants for height 1 is in the range $[\frac{1}{2}\alpha, \frac{3}{2}\alpha]$, which is within the branching factor range needed for property 3.

For v of height $h > 1$, the maximum number of children possible is achieved, when v has the *maximum* number of leaf descendants possible, and all the children of v have the *minimum* number of leaf descendants possible. From the above bounds, this occurs when v has $\frac{3}{2}\alpha^h$ leaf descendants and each child of v has $\frac{1}{2}\alpha^{h-1}$ leaf descendants. Since $\frac{3}{2}$ divided by $\frac{1}{2}$ is 3, this implies that v can have at most 3α children.

Similarly, the minimum number of children possible is achieved when v has $\frac{1}{2}\alpha^h$ leaf descendants and each child has $\frac{3}{2}\alpha^{h-1}$ leaf descendants. Since $\frac{1}{2}$ divided by $\frac{3}{2}$ is $\frac{1}{3}$, v must have at least $\frac{1}{3}\alpha$ children. ∎

Deletions are handled as in [14] using standard techniques for dynamic data structures (see e.g., [26]). Whenever a leaf is deleted, the leaf is marked as deleted and the tables are updated as described above, but the leaf is not actually removed from the tree. This only causes a problem if n shrinks so much that the time bounds expressed in terms of n are not valid because there are many more than $2n$ deleted leaves in the tree.

To prepare for shrinking n, we keep track of a recent maximum $n_{\max}$. Whenever, n shrinks to say $n_{\max}/10$, we start copying the current parenthesis tree to a new tree. At each deletion, we copy a

Proof: A far cover provided by v_l must also be a far cover provided by v. In fact, any far covers for v is a far cover for v_l unless in the old tree $A(k) = v$, but in the new tree $l_k \prec v_l$ and $r_k \prec v_r$. Since $farcover(v, b)$ is the rightmost far cover for descendants of b, there cannot be another such far cover both of whose parentheses descend from v_l because the parentheses nest. Therefore we set, $farcover(v_l, b)$ to NIL, if both the numbered conditions hold. ∎

Lemma 3.32 Near and far cover information can be updated under node splitting in time $O(\alpha)$.

Proof: The algorithm is described above in Lemmas 3.28, 3.29, 3.30, and 3.31 and justified in their proofs. The time complexity follows from the fact that we do constant work to update each far and near cover entry, and that there are $O(\alpha)$ such entries. ∎

3.4.3 Maintaining the Number of Children of Nodes at $\Theta(\alpha)$

In modifying the tree, we allow the branching factor of all internal nodes to vary, but keep it $\Theta(\alpha)$. We will be mainly concerned with controlling the branching factor during insertions; we will handle shrinking during deletions by standard techniques for dynamic data structures.

To maintain the branching factor we will sometimes need to split a node into multiple nodes, dividing up its children. We will prove as part of our analysis (Lemma 3.34, property 1) that when we do a split, we can always split into just *two* nodes.

There are two separate issues that must be considered regarding node splits. One is how to correctly update the NL, NR, and far cover tables associated with the nodes; the table update algorithms do not depend on when the splits are done. This issue was addressed in Subsection 3.4.2. The second issue is how to time the node splits so that we can maintain the branching factor at $\Theta(\alpha)$ and do the splits, so that each insertion takes time $O(\log n / \log \alpha + \alpha)$.

Our strategy for timing the splits is very similar to the strategy that Dietz and Sleator [14] used in their list data structure described in Subsection 2.1

To describe our rules for node splits, we begin with some definitions. The *lower tree* consists of three lowest levels of nodes, including the leaf level. The *upper tree* consists of all the other internal nodes. A *direct split* is a split of an internal node in the lower tree. A *housekeeping split* is a split of an internal node in the upper tree.

We define $L(v)$ to be the number of leaves which are descendants of node v. For each node v in the tree, we keep track of $L(v)$. Since our tree has branching factor α, we would expect $L(v)$ to be $\alpha^{h(v)}$, where the height $h(v)$ of a node v is 0 if v is a leaf, and $1 + h(c(v, 1))$ otherwise.

Similarly to [14] we define an *overflow function* on node v in the upper tree as

$$OF(v) = \max(0, \frac{\alpha^3}{2}(\frac{L(v)}{\alpha^{h(v)}} - 1)).$$

When we add a new leaf l to the tree, we increase $OF(v)$ for the root and all upper tree nodes v such that $l \prec v$. We say that a node v *overflows* if $OF(v)$ is positive.

Here are the rules for doing splits after a new leaf is inserted:

1. If the insertion overflows the parent of the new node, we do a direct split on the parent.
2. If a split of the parent overflows the grandparent, we do a direct split of the grandparent.
3. At each insertion, split one overflowed upper tree node, if there is one.

Lemma 3.33 If in rule 3, we always choose to split a node that has maximum value of OF, then every value of OF will remain $< 2 \log n$, provided $n \geq 5$.

Proof: For any child $c \neq v_l, v_r$ the set of parentheses enclosing its descendants and their ancestors do not change, so the far cover situation cannot change.

However, one of v_l or v_r, but not both, may have a nearer far cover as follows. Suppose there exists a parenthesis pair (l_j, r_j) such that $l_j \prec v_l$, $r_j \not\prec v_r$, but $A(j) = w$. Then (l_j, r_j) could not provide a far cover for v, but does provide a far cover for v_r. Furthermore, the rightmost such left parenthesis provides the nearest enclosing far cover for v_r.

To find the correct new far cover, we need find the rightmost left near cover of v to the left of $c(v, b)$, that is, we find the $rml(c(v, b'))$ for the maximum b' such that $b' \leq b$ and $NL_v[b'] = 1$. Let such a $rml(c(v, b')) = (l_j, r_j)$. Then if $A(j) = w$, we have found a far cover for v_r, and thus the nearest one.

If however, $w \prec A(j)$, then, by the nesting of parentheses, any left near cover l_k of v_l must have $w \prec A(k)$, and thus v_r has the same far cover at w that v did. A symmetric situation holds for v_l and the right near covers of v to the right of $b + 1$. ∎

The next two Lemmas show how to fix the tables for one of the new nodes, v_l.

Lemma 3.30 Let d be the rank of a child of v in the old tree that becomes a child of v_l after the split of v. If there is an entry (l_k, r_k) in the *farcover* table for v such that l_k descends from d and r_k descends from a child of v_r, then $NL_{v_l}[d] = 1$, and $rml(v_l, d) = l_k$. Otherwise, $NL_{v_l}[d] = NL_v[d]$, and if there was a value of $rml(v, d)$ it is retained as $rml(v_l, d)$.

Proof: A left near cover of v_l is a parenthesis l_j such that $l_j \prec v_l \prec A(j)$. Any left near cover of v which is a descendant of v_l is a left near cover of v_l. This explains why we retain the value if we cannot find a suitable former far cover.

By Lemma 3.8, there could be a far cover (l_k, r_k) with left parenthesis descending from d further to the right than $rml(v, d)$. Because of the split of v, it may be the case that in the old tree $A(k) = v$, but in the new tree $l_k \prec v_l$ and $r_k \prec v_r$. Such a pair will provide a new left narrowing bracket to v_l and a new right narrowing bracket to v_r. What we need to show is that any such far cover is an entry in the far cover table.

The proof that the far covers that become rightmost left narrowing brackets can be found in the *farcover* table is partially inductive. First, observe that all such far covers have their left parentheses descending from a child of v_l and their right parentheses descending from a child of v_r, which is further right that any of the left parentheses. Therefore, all the far covers we want to find form a nesting set of parentheses.

To find the innermost such far cover if any, we look in $farcover(v, q)$, where we recall that q is the rightmost child of v inherited by v_l. If q is provided a far cover at all, that far cover must have the right parenthesis right of child q and hence, descending from a child of v_r. Since we store only innermost suitable far covers in the table, the entry $farcover(v, q)$ is the innermost far cover that becomes a rightmost left narrowing bracket after the split.

Suppose that we found a pair in $farcover(v, q)$ and the left parenthesis of that pair descends from child $q_1 < q$. Since the parentheses we want nest, the next one must be the innermost far cover for child q_1 and can be found in $farcover(v, q_1)$. If that entry is non-empty and has its left parenthesis descending from child $q_2 < q_1$, then the next far cover we want is in $farcover(v, q_2)$ and so on. We continue the iterative process until reaching a q_j such that $farcover(v, q_j) = \text{NIL}$. ∎

Lemma 3.31 Let b be the rank of any child of v_l, $farcover(v_l, b)$=$farcover(v, b)$, unless:

1. $farcover(v, b) = (l_k, r_k)$ and
2. in the new tree l_k descends from v_l and r_k descends from v_r.

If both conditions hold, then $farcover(v_l, b)$ is NIL.

We first note that we will have to re-rank all descendants of w, v_l, and v_r since we use the rank of a node to look up near and far cover information. Reranking takes $O(\alpha)$ time. We now focus on the table updates caused by the split of v.

We must update six tables, the near and far cover information for w, v_l and v_r. We will describe only the procedures for w and v_l, since v_r can be handled symmetrically. If v is a child of w, let $rml(v)$ be the rightmost left narrowing bracket for w that descends from v, if any.

Lemma 3.28 To update the near cover information for the parent w, there are three cases, depending on the value of $NL_w[v]$ in the tree before splitting:

$NL_w[v] = 0$: Then $NL_w[v_l] = NL_w[v_r] = 0$.

$NL_w[v] = 1$ and $rml(v) \preceq v_l$: Set $NL_w[v_r] = 0$, $NL_w[v_l] = 1$, and $rml(v_l) = rml(v)$.

$NL_w[v] = 1$ and $rml(v) \preceq v_r$: Set $NL_w[v_r] = 1$, and $rml(v_r) = rml(v)$. If $NL_v[1 \ldots b] = 0$ then $NL_w[v_l] = 0$.

If some entry in the subarray $NL_v[1 \ldots b]$ is 1, then let b' be the biggest index $\leq b$ such that $NL_v[b'] = 1$. Let $l_k = rml(c(v, b'))$. Set $NL_w[v_l] = 1$ and $rml(v_l) = l_k$.

Proof:

$NL_w[v] = 0$: If no descendant of v is a left narrowing bracket for w, this will not change after the split of v, since the new nodes do not add any new descendants.

$NL_w[v] = 1$ and $rml(v) \preceq v_l$: Then v_r provides no left narrowing bracket for w, and thus $NL_w[v_r] = 0$. The former $rml(v)$ now descends from v_l, so we set $NL_w[v_l] = 1$, with $rml(v_l) = rml(v)$.

$NL_w[v] = 1$ and $rml(v) \preceq v_r$: Then $rml(v)$ provides a left narrowing bracket for the descendants of the new node v_r, so $NL_w[v_r] = 1$, and $rml(v_r) = rml(v)$.

How to set $NL_w[v_l]$ is a more subtle question.

To answer this question we consider the narrowing brackets of v. If the entire subarray $NL_v[1 \ldots b] = 0$ then we should set $NL_w[v_l] = 0$, since a narrowing bracket for a parent (w) is also a narrowing bracket for the child (v).

If some entry in the subarray $NL_v[1 \ldots b]$ is 1, then let b' be the biggest index $\leq b$ such that $NL_v[b'] = 1$. Let $l_k = rml(c(v, b'))$. By the definition of narrowing brackets $v \prec A(k)$. This means that $r_k \not\preceq v$, but recall that $rml(v) \prec v$. Let $rml(v) = l_m$. Now l_k encloses l_m. Since l_m is a narrowing bracket for w, $w \prec A(m) \preceq A(k)$. So l_k is a left narrowing bracket for w. We should therefore, set $NL_w[v_l] = 1$ and $rml(v_l) = l_k$. The latter choice of left parenthesis is correct since any bracket to the right of l_k would also have been a bracket for v, thus violating the leftmostness of l_k.

∎

Lemma 3.29 For any child, $c \neq v_l, v_r$, of w the far cover information for c is unchanged.

If v had a far cover at w, then v_l and v_r will also be covered by the same parentheses, but we may need to change the far cover for one of v_l or v_r.

Suppose we can find an $rml(c(v, b'))$ for some b' such that $b' \leq b$ and $NL_v[b'] = 1$. If so, let (l_j, r_j) be $rml(c(v, b'))$ for the maximum such b'. Then if $A(j) = w$ replace $farcover(w, v_r) = rml(c(v, b'))$. A symmetric situation holds for v_l and the right near covers of v to the right of $b + 1$.

Proof: In the first case, once (l_i, r_i) is removed, (l_j, r_j) will become the NEP for all parentheses that had (l_i, r_i) as their NEP. Therefore, we need to replace all *farcover* entries that are (l_i, r_i) with the value (l_j, r_j).

In the second case, we know that no alternate pair node can be far cover for $c(A(i), e)$ since if $(l_k, r_k) \neq (l_j, r_j)$ were such a far cover, (l_k, r_k) would enclose (l_i, r_i) and would be nearer than (l_j, r_j). So we unset the far cover information for all such $c(A(i), e)$. ∎

We describe how to update the left near cover information. The right near cover information is updated symmetrically.

Lemma 3.26 Let v be a node on the path up the tree from l_i to $A(i)$. We have two cases:

$v \prec LCA(l_i, l_j)$**:** Let b be such that $C(v, l_i) = c(v, b)$. If l_i is the rightmost left parenthesis associated with $c(v, b)$, then set $NL[b] = 0$. If l_i is not the rightmost left parenthesis associated with $c(v, b)$, do nothing.

$LCA(l_i, l_j) \preceq v \prec A(i)$**:** Once again, let b be such that $C(v, l_i) = c(v, b)$. $NL[b]$ will remain set to 1 since l_j provides a left near cover at v.

If l_i is the rightmost left parenthesis associated with $c(v, b)$, set l_j to be the rightmost left parenthesis.

Proof: In the first case, suppose l_i is the rightmost left narrowing bracket associated with $c(v, b)$. Suppose, seeking to establish a contradiction, that $l_k \neq l_j$ is a descendant left parenthesis of $c(v, b)$ that is a left near cover at v. If l_k is to the right of l_i, this contradicts l_i's rightmostness. If l_k is to the left of l_i, it would enclose l_i, thus contradicting the fact that l_j is $El(l_i)$. Thus, after the deletion of l_i, $NL[b]$ should be 0.

If l_i is not the rightmost left narrowing bracket associated with $c(v, b)$, deleting l_i still leaves v with a left near cover at $c(v, b)$ further to the right, so we do not need to change and NL or table entries.

In the second case, we set $NL[b]$ to 1 since l_j provides a left near cover at v. If l_i was rightmost, then l_j replaces it. Otherwise, there is a leaf l_k to the right of l_i which is a left cover for the descendants of $c(v, b)$. Removing l_i does not affect this, so we need not update any information. ∎

Lemma 3.27 When deleting a pair (l_i, r_i), the near and far cover information can be updated in time $O(\log n / \log \alpha + \alpha)$.

Proof: The correctness of the algorithm is covered in Lemmas 3.25 and 3.26. The time follows from observing the we do constant time work per node on the path from l_i and r_i to $A(i)$, and $O(\alpha)$ work at $A(i)$. The only other task is to find $NEP(l_i)$, which can be achieved within the same bounds. ∎

Updating the tree after splits. In Subsection 3.4.3, we will show how to balance the tree under insertions and deletions. Deletions are handled as in [14] using the standard technique for dynamic data structures (see e.g., [26]) of inserting into a parallel data structure. Thus for correcting the tables we need only worry in this subsection about how to update near and far covers under insertions.

In the next subsection, we will show that an insertion sometimes causes some nodes to be split. The rules for splits will be such that any individual insertion requires at most a constant number of splits and we will prove (cf. Lemma 3.34) that each split results in a node being split into exactly two new nodes. We must provide a method for efficiently updating the near and far cover information at a node and its parent when we split the node.

More concretely, suppose we wish to split node $v = c(w, b)$ into nodes $v_l = c(w, b)$ and $v_r = c(w, b+1)$. Suppose that v_l inherits children of v with ranks $1, 2, \ldots q$, and v_r inherits the children with ranks $q+1, \ldots$.

In the second case, we do not know if there was a previous far cover for $c(v,e)$. If there was a previous far cover, (l_k, r_k), then (l_k, r_k) is known not to enclose (l_i, r_i); this means that (l_k, r_k) is nearer and should remain as the far cover. However, if there was no previous far cover for $c(v,e)$, then the new parentheses (l_i, r_i) provide one. This means that we only update the *farcover* entry to (l_i, r_i) if the entry was previously NIL. ∎

The next Lemma tells us how to update left near cover information when inserting (l_i, r_i) and describes an algorithm for making the changes. The right near cover updates are symmetric.

Lemma 3.23 Let v be a node on the path up the tree from l_i to $A(i)$. We have two cases:

$v \prec LCA(l_i, l_j)$**:** Let b be such that $C(v, l_i) = c(v, b)$. If $NL[b] = 0$ then we should set $NL[b]$ to 1 and set l_i to be the rightmost left parenthesis for child b that is a narrowing bracket at v. If $NL[b] = 1$, do nothing.

$LCA(l_i, l_j) \preceq v \prec A(i)$**:** Once again, let b be such that $C(v, l_i) = c(v, b)$. $NL[b]$ will be 1 since l_j provides a left near cover at v. If l_j is the rightmost left parenthesis associated with $c(v, b)$, set l_i to be the rightmost left parenthesis. Otherwise, some other l_k is associated with $c(v, b)$, but then l_i encloses l_k, so we do nothing.

Proof: In the first case, l_i is a left narrowing bracket for v. If v does not previously have a left narrowing bracket descending from b, then l_i is the first such narrowing bracket. Therefore, we set $NL[b]$ and make l_i the rightmost left narrowing bracket for child b. If $NL[b]$ was already 1, then this must be caused by by some parenthesis $l_k \neq l_j$. Since l_k is distinct from l_j, l_k does not enclose l_i and instead,l_i encloses l_k. This means that $l_i < l_k$ so we should not change the rightmost left narrowing bracket descending from b.

The proof in the second case is similar. The definition of the case already explains that $NL[b]$ must be 1; therefore, it does not need to change. We change the rightmost left parenthesis only if it is now l_i. This is true, if and only if it was l_j before. As in the first case, if the rightmost left narrowing bracket is $l_k \neq l_j$, then l_i encloses l_k and l_i is not rightmost. ∎

Lemma 3.24 When inserting a pair (l_i, r_i), the near and far cover information can be updated in time $O(\log n / \log \alpha + \alpha)$.

Proof: The correctness follows from Lemmas 3.22 and 3.23. The time follows from observing the we do constant time work per node on the path from l_i and r_i to $A(i)$, and $O(\alpha)$ work at $A(i)$. The two lemmas assume we have found $NEP(l_i)$, which can be achieved within the same bounds. ∎

Leaf Pair Deletion. The deletion of a pair (l_i, r_i) is largely symmetric to its insertion. Again, let $NEP(l_i) = (l_j, r_j)$. The next two Lemmas describe how to update the far cover and near cover information that involves (l_i, r_i). Recall that the parenthesis pair (l_i, r_i) can only be provided as a far cover by the LCA, $A(i)$.

Lemma 3.25 The far cover update rules are separated into two cases:

$A(i) = A(j)$**:** Replace all far cover table entries that are (l_i, r_i) with (l_j, r_j) instead. Do not set any of the *farcover* entries to NIL.

$A(i) \prec A(j)$**:** Let e be such that the $farcover(c(A(i), e)) = (l_i, r_i)$. Set $farcover(c(A(i), e) =$ NIL.

}

3 /* Narrowing Bracket Case */
Choose largest (smallest) $k < j$ ($k > j$) such that $NL_v[k] = 1$ ($NR_v[k] = 1$).
Return the rightmost (leftmost) left (right) parenthesis associated with $c(v, k)$.

Lemma 3.20 The total time for a query is $O(\log n / \log \alpha + \alpha)$.

Proof: Follows from corollaries 3.15 and 3.19. ∎

3.4.2 Updates

When a pair of parentheses is inserted or deleted, we must deal with three issues: near cover information, far cover information, and updating the tree to accommodate the nodes inserted or deleted.

We first consider the case of simply inserting or deleting a matching pair of leaves, l_i, r_i, without modifying any internal nodes.

Leaf Pair Insertion. In the following discussion assume that we are inserting the pair (l_i, r_i), that (l_i, r_i) will become the new NEP for some set S of parentheses, and that (l_i, r_i) will have some pair (l_j, r_j) as its NEP. Before (l_i, r_i) are inserted, the pair (l_j, r_j) is the NEP of all parentheses in S. The pair (l_i, r_i) may replace (l_j, r_j) in various near and far cover tables.

When inserting (l_i, r_i) we must first find the NEP, (l_j, r_j) of (l_i, r_i). Then we update the near and far cover tables.

Lemma 3.21 Let p be the parenthesis immediately to the left of l_i. If p is a left parenthesis, then $p = El(l_i)$. If p is a right parenthesis, then, $NEP(p) = NEP(l_i)$.

Proof: If p is a left parenthesis, it is the closest left parenthesis outside of l_i. Since well-balanced parentheses nest, p and its mate must enclose (l_i, r_i), and are therefore the nearest enclosing parentheses.

If p is a right parenthesis, then $Er(p)$ must be to the right of l_i. Since well-balanced parentheses nest, $Er(p)$ must also be to the right of r_i. Thus $(El(p), Er(p))$ also enclose (l_i, r_i). More generally any pair of parentheses that enclose p, will also enclose (l_i, r_i) and vice-versa, so $NEP(p) = NEP(l_i)$. ∎

The next Lemma tells us in which *farcover* table entries we need to insert (l_i, r_i). It also describes a simple algorithm for making the changes. Recall that the parenthesis pair (l_i, r_i) can only be provided as a far cover by the LCA, $A(i)$.

Lemma 3.22 The far cover update rules are separated into two cases:

$A(i) = A(j)$**:** Let $v = A(i)$, and let $a \leq b < c \leq d$ be such that $C(v, l_j) = c(v, a)$, $C(v, l_i) = c(v, b)$, $C(v, r_i) = c(v, c)$, and $C(v, r_j) = c(v, d)$. Then for all $b < e < c$, if the far cover for $c(v, e)$ is (l_j, r_j), change the far cover to be (l_i, r_i).

$A(i) \prec A(j)$**:** Once again, let $v = A(i)$, and let $b < c$ be such that $C(v, l_i) = c(v, b)$ and $C(v, r_i) = c(v, c)$. For any $b < e < c$, if $c(v, e)$ has NIL as a *farcover* value, then use (l_i, r_i) as the far cover.

Proof: In the first case, the *farcover* entry for $c(v, e)$ must be non-NIL because (l_j, r_j) is a far cover, but the nearest enclosing parentheses recorded in the far cover table may be either (l_j, r_j) or some nearer pair (l_k, r_k) enclosed by (l_j, r_j). If the far cover for $c(v, e)$ is (l_j, r_j), the new pair (l_i, r_i) is enclosed by (l_j, r_j). Furthermore, the pair (l_i, r_i) will enclose all parentheses below $c(v, e)$, so it should be the new far cover. If the previous far cover is some other $(l_k, r_k) \neq (l_j, r_j)$, then (l_i, r_i) enclose (l_k, r_k), and are therefore not the nearest enlosing parentheses of any parenthesis enclosed by (l_k, r_k).

1. $C(w, p_1) = C(w, p_2)$, call the child x,

2. no node below w provides a near cover for p_1 or p_2, or their matching parentheses, and

3. w provides a far cover for p_1 or p_2.

Then w provides a far cover for *both* p_1 and p_2 and it is the same; i.e., $El(p_1) = El(p_2)$.

Proof: If both p_1 and p_2 are left parentheses or both are right parentheses, then the corollary is just a restatement of Lemma 3.16 in which the parenthesis that is guaranteed the far cover by Assumption 3, plays the role of l_i or r_i in the lemma.

Otherwise, assume without loss of generality that p_1 is a left parenthesis and p_2 is a right parenthesis. Let r_1 be the mate of p_1 and let l_2 be the mate of p_2. There are four possible orderings of the parentheses p_1, r_1, l_2, p_2. We consider them separately:

$p_1 < l_2 < p_2 < r_1$. Since l_2 is in between p_1 and p_2, it must also descend from $C(w, p_1)$. Thus we can apply Lemma 3.16 to the two left parentheses p_1 and l_2.

$l_2 < p_1 < r_1 < p_2$. Since r_1 is in between p_1 and p_2, we can apply Lemma 3.16 to the two right parentheses r_1 and p_2.

$p_1 < r_1 < l_2 < p_2$. Same as the preceding case.

$l_2 < p_2 < p_1 < r_1$. Since neither p_2 nor p_1 has a near cover lower in the tree, any parentheses that descend from w and enclose either p_2 or p_1 cannot descend from the child x. Therefore, any enclosing left parenthesis must be left of p_2 and any enclosing right parenthesis must be right of p_1. This means that any parentheses that descend from w and enclose p_1 or p_2 also enclose the other of p_1 or p_2. Therefore, the nearest enclosing parentheses for one are also the nearest enclosing parentheses for the other. ∎

Lemma 3.18 Given nodes v and $C(v, l_i)$, we can find the nearest enclosing parentheses for (l_i, r_i) which are a far cover at v in $O(1)$ time.

Proof: By Corollary 3.17, we need keep track at each node v the nearest enclosing parentheses which provide a far cover for each of its children. Then we can look up, for an node $C(v, l_i)$, if the descendants of $C(v, l_i)$ have a far cover at v, and if so, which is the nearest enclosing far cover. ∎

We will refer to the parentheses that provide the nearest far cover for descendants of $c(v, k)$ as $farcover(c(v, k))$. We assume that any children of v that do not have a far cover for their descendants have a NIL entry in the $farcover$ table.

Corollary 3.19 The total time to find a far cover is $O(\log n / \log \alpha)$.

We summarize the query algorithm as follows:

Algorithm 1 Finding nearest enclosing parentheses

QUERY$((l_i, r_i))$
1 $v_l \leftarrow l_i; v_r \leftarrow r_i;$
2 While $(v_l \neq root)\{$
$v_l \leftarrow parent(v_l); v_r \leftarrow parent(v_r);$
Let j_l be such that $c(v, j_l) = c(v, l_i)$.
Let j_r be such that $c(v, j_r) = c(v, r_i)$.
If $farcover(c(v, j_l)) \neq NIL$, then return $farcover(c(v, j_l))$.
If $NL_v[k] = 1$ for any $1 \leq k < j_l$ then goto step 3.
If $NR_v[k] = 1$ for any $j_r < k \leq degree(v)$ then goto step 3.

with $NL[j] = 1$ ($NR[j] = 1$) and $c(v, j)$ is left (right) of the child of v that is an ancestor of l_i (r_i). We know which child of v is the ancestor of l_i (r_i) because it is the predecessor of v on the path.

Lemma 3.14 Given a node v and one of its children $C(v, l_i)$ such that v provides a left near cover for l_i, we can find in time $O(\alpha)$ the nearest parenthesis l_j such that v provides the left near cover l_j for l_i. A symmetric property holds for right near covers.

Proof: At each child $c(v, k)$, we keep a pointer to the rightmost left parenthesis which is a descendant of $c(v, k)$ and is a left narrowing bracket for v. Similarly we keep a pointer to the leftmost right parenthesis which is a descendant of $c(v, k)$ and is a left narrowing bracket for v.

Let v be the lowest node with the property that some $NL[j] = 1$ and $c(v, j)$ is left of the child of v that is an ancestor of l_i. By Lemma 3.13, the only possible candidates for $El(l_i)$ are the rightmost left parentheses that are narrowing brackets for v, stored at each of $c(v, j-1), \ldots, c(v, 1)$. The rightmost amongst these will be nearest to l_i and is therefore the enclosing left parenthesis we seek. ∎

Corollary 3.15 The total time to find a near cover is $O(\log n / \log \alpha + \alpha)$.

Far Covers By Lemma 3.8, we need to check at each node if it provides a far cover before we check if it has a near cover. The following lemma and its corollary reduce the work in checking for far covers.

Lemma 3.16 Suppose that node w provides a far cover for (l_i, r_i). Every parenthesis pair (l_1, r_1) with the properties that:

1. $C(w, l_i) = C(w, l_1)$ or $C(w, r_i) = C(w, r_1)$,
2. no node below w provides a near cover for (l_1, r_1)

has $(El(l_i), Er(r_i))$ as its nearest enclosing parentheses.

Proof: We do the proof for the case where $C(w, l_i) = C(w, l_1)$; the case where $C(w, r_i) = C(w, r_1)$ is symmetric.

Since w provides a far cover for (l_1, r_1), we have that $LCA(El(l_1), Er(r_1)) = w$. Furthermore, since we do not have a lower near cover, $C(w, El(l_i)) \neq C(w, l_i) = C(w, l_1) \neq C(w, El(l_1))$. Also, $C(w, Er(r_i)) \neq C(w, r_i)$ and $C(w, Er(r_1)) \neq C(w, r_1)$.

Now there are two cases depending on whether the parentheses pairs (l_i, r_i) and (l_1, r_1) nest or do not nest.

Suppose that (l_i, r_i) and (l_1, r_1) nest. Neither left enclosing parenthesis is a descendant of $C(w, l_i) = C(w, l_1)$. Hence both $El(l_i)$ and $El(l_1)$ must be descendants of a child of w that is further left than $C(w, l_i)$. Hence, both $El(l_i)$ and $El(l_1)$ are to the left of both l_i and l_1. Since well-balanced parentheses nest $(El(l_i), Er(r_i))$ must enclose both (l_i, r_i) and (l_1, r_1). Also, $(El(l_i), Er(r_i))$ must enclose both (l_i, r_i) and (l_1, r_1). Hence $El(l_i) = El(l_1)$.

Suppose that (l_i, r_i) and (l_1, r_1) do not nest. Assume without loss of generality that l_i is to the left of l_1. Since $C(w, l_i) = C(w, l_1)$., we must also have $C(w_i, l_i) = C(w_i, r_i)$. As in the nesting case, both $El(l_i)$ and $El(l_1)$ are both to the left of both l_i and l_1. Since l_1 is to the right of r_i, $(El(l_1), Er(r_1))$ must enclose both (l_i, r_i) and (l_1, r_1). Since $C(w, Er(r_i)) \neq C(w, r_i) = C(w, l_1)$, the enclosing right parenthesis, $Er(r_i)$, must be to the right of l_1. Thus $(El(l_i), Er(r_i))$ encloses both (l_1, r_1) and (l_i, r_i). Since well-balanced parentheses nest, one of $(El(l_1), Er(r_1))$ and $(El(l_i), Er(r_i))$ encloses the other. Whichever pair is inside is the nearest enclosing parentheses of *both* (l_i, r_i) and (l_1, r_1). ∎

Corollary 3.17 Let w be a node. Let p_1 and p_2 be two single non-matching parentheses such that

Lemma 3.10 The parentheses (l_i, r_i) have a left (right) near cover at v_l (v_r) iff v_l (v_r) has a narrowing bracket l_j (r_j) such that $C(v_l, l_j)$ ($C(v_r, r_j)$) is to the left (right) of $C(v_l, l_i)$ ($C(v_r, r_i)$).

Proof: We do the proof in each direction for l_i and left narrowing brackets. The proofs for r_i and right narrowing brackets are symmetric.

(If)Let (l_j, r_j) be a left narrowing bracket for v_l such that $LCA(l_i, l_j) = v$ and such that $C(v, l_i)$ is to the right of $C(v, l_j)$. Then, since $v \prec A(j)$, $l_i < r_j$ and so $l_j < l_i < r_i < r_j$. In other words, (l_j, r_j) enclose (l_i, r_i). But now $LCA(l_j, l_i) \prec A(j)$, so (l_j, r_j) provides a left near cover at v_l.

(Only if) Suppose that (l_i, r_i) have a near cover because $LCA(l_j, l_i)) \prec A(j)$. Let $v_l = LCA(l_j, l_i)$. Then, (l_j, r_j) is a left narrowing bracket for v_l because $l_j \prec v_l \prec A(j)$ ∎

Lemma 3.11 Suppose l_i (r_i) has an ancestor v such that v has a left (right) narrowing bracket l_j (r_j) with l_j (r_j) to the left (right) of l_i (r_i). Then (l_j, r_j) encloses (l_i, r_i).

Proof: We do the proof for left narrowing brackets. By definition of narrowing bracket $LCA(l_j, l_i) \preceq v$. Also, $v \prec LCA(l_j, r_j) = A(j)$. Thus r_j must be to the right of l_i. Since well-balanced parentheses nest r_j must also be to the right of r_i. Hence (l_j, r_j) encloses (l_i, r_i). ∎

Lemma 3.12 We can detect if a node provides a near cover in $O(1)$ time.

Proof: To help find narrowing brackets, at each node v, we keep two bit vectors $NL[]$ and $NR[]$ where $NL[k] = 1$ if $c(v, k)$ has a descendant left parenthesis which is a left narrowing bracket for v. Similarly, $NR[k] = 1$ if $c(v, k)$ has a descendant right parenthesis which is a right narrowing bracket for v.

We will prove later, in Lemma 3.34, that each tree node has at most 3α children, so the size of NL and NR should be 3α. If α is small enough, then we can encode NL (and also NR) into one machine word. We can check by table lookup [15, 29] if some left parenthesis with ancestor $c(v, k)$ has an enclosing left parenthesis that is a narrowing bracket for v: this is the case precisely when, for some $1 \leq j < k$, $NL[j] = 1$. Similarly, we can check if some ancestor of a right parenthesis has a right narrowing bracket by consulting NR. ∎

At the end of the preceding proof we use $1 \leq j < k$ rather than $1 \leq j \leq k$ because if $j = k$ then both l_i and the narrowing bracket would descend from the same child of v, and therefore that narrowing bracket would already be a near cover at a descendant of v.

With the NL and NR tables built, we can trace the path to the root from l_i and r_i and check in constant time per ancestor node if (l_i, r_i) have a near cover.

Suppose we are at a node v that provides a near cover. The next two lemmas tell us how to choose the descendant parenthesis that is half of the nearest enclosing parenthesis pair and find it quickly. In both cases we do the proof for left narrowing brackets and left enclosing parentheses.

Lemma 3.13 Suppose (l_i, r_i) has a near cover because $LCA(El(l_i, l_i)) \prec A(i)$. Then the *rightmost* left parenthesis to the left of l_i that is a narrowing bracket for an ancestor of l_i is $El(l_i)$. The corresponding fact holds for r_i and right parentheses.

Proof: By Lemma 3.10, $(El(l_i), El(r_i))$ is a narrowing bracket for an ancestor of l_i. Suppose that $El(l_i)$ is *not* the rightmost left narrowing bracket for an ancestor l_i. Then there would be another left narrowing bracket l_1 such that $El(l_i)$ is to the left of l_1 and l_1 is to the left of l_i. By Lemma 3.11, the pair (l_1, r_1) would enclose (l_i, r_i), and be nearer to l_i than the pair $(El(l_i), Er(r_i))$. This contradicts the definition of *nearest enclosing parentheses.* ∎

To find $(El(l_i), Er(l_i))$ for parentheses (l_i, r_i) that have a near cover we trace up the tree towards $A(i)$. We can stop when we find, by means of table lookup, the first node v such that there is some j

Proof: Assume $E(i)$ does not provide a far cover for l_i. Since $A(i) \preceq E(i)$, then either $LCA(El(l_i), l_i) \prec E(i)$ or $LCA(Er(l_i), r_i) \prec E(i)$. If so, then $LCA(El(l_i), l_i)$ ($LCA(Er(l_i), r_i)$) provides a left (right) near cover for l_i, contradicting the assumption. ∎

Let $\mathcal{E}_l(l_i) = \{l_j | l_j < l_i < r_i < r_j\}$ be the set of left parentheses that enclose l_i. Let $\mathcal{N}_l(l_i) = \{LCA(l_j, l_i) | l_j \in \mathcal{E}_l(l_i)\}$ be the set of nodes that provide left near covers and far covers. Symmetrically define $\mathcal{E}_r$ and $\mathcal{N}_r$. Each node $v \in \mathcal{N}_l(l_i)$ ($v \in \mathcal{N}_r(l_i)$) is an ancestor of l_i (r_i).

We will derive our algorithm from the following lemma about finding the *nearest* enclosing parentheses from amongst $\mathcal{E}_l(l_i)$ and $\mathcal{E}_r(l_i)$.

Lemma 3.8 Suppose v provides the far cover (l_j, r_j) for (l_i, r_i) and the near cover (l_k, r_k). Then $l_k < l_j < l_i < r_i < r_j < r_k$. That is, if the same node provides both a far cover and near cover, the far cover will always be a nearer enclosing parenthesis pair.

Proof: We do the proof assuming that (l_k, r_k) is a left near cover; the case where it is a right near cover is symmetric.

Since (l_k, r_k) is a left near cover at v, $LCA(l_k, l_i) \prec A(k)$. Since (l_j, r_j) is a far cover at v, we have $v = A(j) = E(i) = LCA(l_j, l_i) = LCA(r_j, r_i)$.

Putting the above facts together we see that v is an ancestor of l_k, l_j, l_i, r_i, and r_j, but v is not an ancestor of r_k because $LCA(l_k, r_k) = A(k)$ is a proper ancestor of v.

Let us determine the left to right ordering of the right parentheses r_i, r_j, and r_k. Since r_i is enclosed by (l_j, r_j) and by (l_k, r_k), r_i is the leftmost of the 3 right parentheses. Since r_j is a descendant of v, but r_k is not a descendant of v, r_j must be closer to r_i. Hence the right to left order is $r_i < r_j < r_k$.

Because well-balanced parentheses nest, the complete order on the 6 parentheses must be $l_k < l_j < l_i < r_i < r_j < r_k$. ∎

The Algorithm: The search algorithm traces up from a pair of query parentheses (l_i, r_i) to the root looking for a far or near cover at each node on the path. By Lemma 3.8, the algorithm should be structured so that at each node v it checks first for a far cover provided by v and then for a near cover provided by v. When we find the lowest cover, we look up the nearest enclosing parentheses at that node and report those parentheses.

The algorithm is summarized in pseudocode at the end of this subsection. To fill in the details, we describe efficient solutions to the following subtasks:

- How do we determine if a node v provides a near cover? (cf. Lemmas 3.10 and 3.12).
- How do we find the nearest parenthesis that provides a near cover at v? (cf. Lemmas 3.13 and 3.14)
- How do we determine if a node v provides a far cover?
- How do we find the nearest parenthesis that provides a far cover at v?

The first two questions are answered directly by the Lemmas listed above. The last two questions are answered by a *farcover* table data structure discussed after Corollary 3.17.

Near Covers

Definition 3.9 A pair of matching parentheses (l_j, r_j) is a *narrowing bracket* for v if $v \prec A(j)$ and either $l_j \prec v$ or $r_j \prec v$. In the first case, we say that l_j is a *left narrowing bracket* for v and in the second case r_j is a *right narrowing bracket* for v.

The following lemma shows the relationship between narrowing brackets and near covers.

with a matching ')'. Replacing this matching pair with pairs "()" preserves property 1. By replacing the output of the previously marked v with the output for an unmarked node, we preserve property 2.

This completes the induction step and proves that the the four update operations can be implemented to preserve the correspondence between tree S and parenthetical tour B.

For the time analysis, observe that each tree operation requires $O(1)$ pointer lookups and updates, and a constant number of parenthesis operations, each costing $O(N)$ time. ∎

We now concentrate on obtaining efficient algorithms for the nearest enclosing parenthesis problem.

3.4 Fully dynamic parenthesis maintenance

In our algorithm for fully dynamic balanced parenthesis maintenance, we build a balanced α-ary tree on the sequence of parentheses, where α is a parameter to be defined below. The degree of all internal vertices is anticipated to be roughly α, and all leaves have the exact same depth. All the parentheses are stored as leaves; the internal nodes store information to assist in searching and updating the tree. If we can make the degree of every node $\Theta(\alpha)$ the height of such a tree is $O(\log n / \log \alpha)$.

We begin with some notation. Let $A(i)$ be the least common ancestor of the matching parentheses l_i and r_i in the tree. In general, let $LCA(u, v)$ be the least common ancestor of nodes u and v. We use $c(v, k)$ to denote the kth child of v. Let $C(v, x)$ be $c(v, j)$ if $c(v, j)$ is an ancestor of x, and undefined otherwise. Let $El(x)$ be the left nearest enclosing parenthesis of x, let $Er(x)$ be the matching right parenthesis of $El(x)$, and let $E(i)$ be $LCA(El(l_i), Er(l_i))$.

We denote by $\preceq$ the "is a descendant of" order, and by $\prec$ the "is a proper descendant of" order. We write $p_1 < p_2$ if parenthesis p_1 is left of p_2 in the left-to-right ordering of leaves in the tree of parentheses.

If we consider the parentheses $EL(l_i), l_i, r_i$, and $Er(l_i)$, they occur left to right in that order. Therefore the LCA of the first two and the LCA of the last two can occur no higher in the tree than the LCA of the outer two, $EL(l_i)$ and $El(r_i)$. Using this observation we distinguish two cases:

1. $LCA(El(l_i), l_i) \prec E(i)$ or $LCA(Er(l_i), r_i) \prec E(i)$.
2. $E(i) = LCA(El(l_i), l_i) = LCA(Er(l_i), r_i)$.

In general, let (l_j, r_j) enclose (l_i, r_i). If $LCA(l_j, l_i) \prec A(j)$ or $LCA(r_j, r_i) \prec A(j)$, we say that (l_j, r_j) is a *near cover* for (l_i, r_i). If $LCA(l_j, l_i) \prec A(j)$, then we say that $LCA(l_j, l_i)$ *provides a left near cover* for l_i. If $LCA(r_j, r_i) \prec A(j)$, then we say that $LCA(r_j, r_i)$ *provides a right near cover* for l_i. This corresponds to case 1 above.

If $A(j) = LCA(l_j, l_i) = LCA(r_j, r_i)$, we say that (l_j, r_j) is a *far cover* for (l_i, r_i) and that $A(j)$ *provides a far cover* for l_i. This corresponds to case 2 above.

Thus, $(El(l_i), Er(l_i))$ is either a near or far cover for (l_i, r_i).

3.4.1 Queries

The following lemmas provide some structure to near and far covers and provide the basis of our algorithm for finding enclosing parentheses.

Fact 3.6 By the definition of far covers, no node v, $l_i \prec v \prec E(i)$, provides a far cover for parenthesis l_i; a similar fact holds for r_i.

Lemma 3.7 If no node v, $l_i \prec v \prec E(i)$, provides a left near cover for l_i, and no node w, $r_i \prec w \prec E(i)$, provides a right near cover for r_i, then $E(i)$ provides a far cover for l_i.

Proof: Let S be a marked tree. By Lemma 3.4 we can answer nearest marked ancestor queries on S in time $O(N)$, if we can maintain a parenthetical tour B of S and the pointers that indicate the correspondence between parentheses and tree nodes. We show how each marked tree update operation corresponds to an update in B.

Let $parent(v)$ be the parent of v in S. Let $left(v)$ and $right(v)$ be its (possibly empty) left and right siblings, respectively. Finally, let

$$LPT(v) = \begin{cases} \mathcal{P}^{\uparrow}(left(v)) & \text{if } left(v) \text{ exists} \\ \mathcal{P}^{\downarrow}(parent(v)) & \text{otherwise} \end{cases}$$

$$RPT(v) = \begin{cases} \mathcal{P}^{\downarrow}(right(v)) & \text{if } right(v) \text{ exists} \\ \mathcal{P}^{\uparrow}(parent(v)) & \text{otherwise} \end{cases}$$

$LPT(v)$ and $RPT(v)$ are either single parentheses or a matching pair of parentheses that do not enclose anything. Observe that $LPT(v)$ and $RPT(v)$ become neighbors once v is deleted. If we keep $parent$, $right$, and $left$ pointers, then $LPT(v)$ and $RPT(v)$ can be found in $O(1)$ time.

Here is how we update B for each tree operation:

Insert leaf v**:** Insert "()()" between $LPT(v)$ and $RPT(v)$. Update $left$, $right$, and $parent$ pointers.

Delete leaf v**:** Delete $\mathcal{P}^{\downarrow}(v)$ and $\mathcal{P}^{\uparrow}(v)$. Update $left$, $right$, and $parent$ pointers.

Mark node v**:** Replace the adjacent pair of parentheses $\mathcal{P}^{\downarrow}(v)$ with '('. Replace the pair $\mathcal{P}^{\uparrow}(v)$ with ')'. This is implemented via two parenthesis pair deletions and one insertion.

Unmark node v**:** Reverse the replacements for marking a node.

Nearest marked ancestor of v **:** Compute $\mathcal{N}(NEP(\mathcal{P}(v)))$.

We prove that the four update algorithms for Insert, Delete, Mark, and Unmark preserve the following two properties:

1. B is well-balanced.
2. B is a parenthetical tour of S.

The proof is by induction on the number of updates. Property 2 holds initially by definition. Property 1 holds initially by Lemma 3.4. Assume as inductive hypothesis that both properties hold after $i-1$ update operations, and consider the i-th update.

Insert Leaf v. Inserting "()" always keeps a well-balanced list well-balanced, so Property 1 is preserved. In a parenthetical tour, v produces output immediately after $LPT(v)$ and immediately before $RPT(v)$, so the place of insertion for the parentheses preserves property 2.

Delete Leaf v. Deleting pairs of parentheses that match from a well-balanced list keeps it well-balanced, so property 1 is preserved. The parentheses deleted are precisely those output by v in a parenthetical tour; therefore, what remains after their deletion is a parenthetical tour of the updated tree.

Mark v. The substring of B from $\mathcal{P}^{\downarrow}(v)$ to $\mathcal{P}^{\uparrow}(v)$ is a parenthetical tour of the subtree rooted at v. Therefore, by Lemma 3.4 that substring is well-balanced. The substring begins with "()" and ends with "()". Replacing these pairs with a single pair of matching parentheses preserves property 1. By replacing the output of the previously unmarked v with the output for a marked node, we preserve property 2.

Unmark v. The substring of B from $\mathcal{P}^{\downarrow}(v)$ to $\mathcal{P}^{\uparrow}(v)$ is a parenthetical tour of the subtree rooted at v. Therefore, by Lemma 3.4 that substring is well-balanced. The substring begins with '(' and ends

The dynamic parenthesis maintenance problem was previously considered by Güting and Wood [18] who gave a solution where each operation takes $O(\log d)$ time on a parenthesis string of size d. Güting and Wood applied their solution to problems in computational geometry. We will improve the bounds for parenthesis maintenance to $O(\log d / \log\log d)$ per operation (cf. Lemma 3.36).

To key to the reduction from marked trees to parentheses is the following definition. Let a *parenthetical tour* B of a marked tree S be a sequence of parentheses obtained by the following rules. Starting with *cur* set to the root of S, we traverse S in depth first search order and produce S by:

- If *cur* is unmarked,
  - If *cur* is being visited for the first or last time in the DFS traversal, append to B a matching pair ().
- Otherwise [*cur* is marked],
  - If *cur* being visited for the first time, append a '(' to B.
  - If *cur* being visited for the last time, append a ')' to B.

A parenthetical tour differs slightly from what Westbrook [29] calls "the standard parentheses encoding of a tree" in that we do not want the output parentheses for an unmarked node to enclose anything, so we double them. We say that the parentheses that are output on visits to a node v, *represent* v. We define some pointer functions that maintain the correspondence between the parentheses and the tree nodes. For any node v in S, $\mathcal{P}(v)$ is the first '(' in B that represents v. We extend this to define $\mathcal{P}^{\downarrow}(v)$ to mean the output on the first, downward visit to v and $\mathcal{P}^{\uparrow}(v)$ to mean the output on the last, upward visit to v. For any parenthesis x in v, $\mathcal{N}(x)$ is the tree node that x represents.

Let $NMA(v)$ be the nearest marked ancestor of node v. Let $NEP(i)$ be the nearest enclosing parenthesis of parenthesis i.

Lemma 3.4 For any marked tree S, the sequence of parentheses, B, produced by the above rules is well-balanced. For any unmarked node $v \in S$, $NMA(v) = \mathcal{N}(NEP(\mathcal{P}(v)))$.

Proof: The fact that B is well-balanced can be established by a straightforward inductive proof on S.

To show that for any unmarked $v \in S$, $NMA(v) = \mathcal{N}(NEP(\mathcal{P}(v)))$, we must show three things: that $\mathcal{N}(NEP(\mathcal{P}(v)))$ is marked, that it is an ancestor of v, and that it is the nearest such ancestor.

$\mathcal{N}(NEP(\mathcal{P}(v)))$ is marked because only marked nodes can produce enclosing parentheses. Unmarked nodes produce two pairs of parentheses that do not enclose anything.

Suppose $\mathcal{N}(NEP(\mathcal{P}(v))) = w$. The parentheses in B that $\mathcal{P}^{\downarrow}(w)$ and $\mathcal{P}^{\uparrow}(w)$ enclose represent tree nodes in the subtree rooted at w because of the depth first traversal order. Therefore, w is an ancestor v.

Furthermore, for any marked node x, the matching parentheses $\mathcal{P}^{\downarrow}(x)$ and $\mathcal{P}^{\uparrow}(x)$ enclose all the parentheses that represent other nodes in the subtree rooted at x. Thus the parentheses representing any node are enclosed by the parentheses representing each of its marked ancestors. Since "is an ancestor of" ordering is total over the ancestors of v, the nearest enclosing parentheses of v represent the nearest marked ancestor of v. ∎

Lemma 3.5 Let N be the (uniform) cost of satisfying requests for the dynamic parenthesis maintenance problem. Then marked tree update and query requests for the nearest marked ancestor query problem can be satisfied within time $O(N)$.

Inserting Leaf v: $O(1)$ amortized

Marking Node v: $O(1)$ worst-case

Nearest Marked Ancestor of v: $O(1)$ worst-case

In the extended abstract of this paper [6], we presented a very similar data structure that achieves the same time bounds, but Westbrook's is more general.

Theorem 3.3 The one dimensional semi-dynamic dictionary matching problem can be solved within the following time bounds:

Dictionary Preprocessing: $O(d \log \sigma)$ worst-case.

Adding pattern P: $O(p \log \sigma)$ amortized.

Scanning text T: $O(t \log \sigma + tocc)$ worst-case.

Proof: Plug in Westbrook's solution for the nearest marked ancestor problem in Lemma 3.2 into the reduction of the dictionary prefix lookup problem to the nearest marked ancestor problem given in Lemma 3.1. This shows that we can solve the semi-dynamic prefix dictionary lookup problem in time bounds:

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

Insert P: $O(p \log \sigma)$ amortized.

Lookup (j,k): $O(|out_{j,k}|)$

Plug these bounds for the semi-dynamic prefix lookup problem into Lemma 2.4 to complete the proof. ∎

3.3 Reducing the marked ancestor problem to parenthesis maintenance

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).

In our application, the fact that the root of the trie is always marked will imply that B will always have an outermost pair of matching parentheses that enclose everything else, and are never the arguments of Find queries. Thus every Find query will retrieve an enclosing parenthesis pair.

while cur is not the root.

Let cur be the node representing $P_i[1,\ldots,j]$. Then the longest pattern prefix of $P_i[1,\ldots,j]$ is the nearest marked ancestor of cur. This is so because all and only nodes representing patterns are marked, and because the "is an ancestor" relation in tries is the "is a prefix of" relation in strings. The following algorithm will therefore implement the prefix lookup:

```
LOOKUP(i, j)
    Let cur be the node with label P_i[1,...,j].
    While cur is not the root of S {
        Let m be the nearest marked ancestor of cur.
        If m is not the root {
            Output the pattern that m represents.
            cur ← parent(m).
        }
    }
```

Lemma 3.1 Let C be the (uniform) cost of satisfying marked tree update and query requests. Then dictionary prefix lookup update and query requests can be satisfied in times:

Inserting/Deleting P: $O(p(\log\sigma + C))$.

Lookup (i,j): $O(|out_{(i,j)}|C)$.

Proof: The insertion and deletion algorithms are the standard algorithms for updating a trie, except for the marking and unmarking. The correctness of the lookup algorithm follows from the correspondence between the "is a prefix of" relation on strings and the "is an ancestor of" relation on tries.

For the time analysis, note that each insertion (deletion) of a pattern P begins by tracing through the trie to find where P belongs. This takes time $O(p\log\sigma)$. Then we must insert (delete) up to p leaves, each such operation taking time $O(C)$. Finally, we must mark (unmark) a single node. This operation also takes time $O(C)$. The prefix lookup complexity follows from the fact that every time we output a prefix pattern, we expend time $O(C)$ in a single lookup. ∎

If we use cutting and linking trees [27], for example, we can solve the *nearest marked ancestor query problem* directly. This solution is implicit in [5] and it achieves $C = O(\log d)$. We will reduce the nearest marked ancestor problem to a parenthesis maintenance problem in order to achieve $C = O(\log d/\log\log d)$.

3.2 Semi-dynamic dictionary matching

In this section we give some new results on the semi-dynamic dictionary matching problem. Our main tool is a data structure recently developed by Westbrook [29] to solve the semi-dynamic nearest marked ancestor query problem. It is interesting that Westbrook developed his data structure for the problem of incremental planarity testing, which seems rather unrelated to string matching. Westbrook's data structure supports some more operations besides Insert, Mark, and Nearest Marked Ancestor, but we have no use for them here, so we summarize the parts of his results that we need for dictionary matching.

Lemma 3.2 ([29]) There is a data structure to solve the semi-dynamic nearest marked ancestor problem that achieves the following time bounds:

Building the Data Structure: $O(d\log\sigma)$, where d is the number of nodes and σ is the degree of the tree

maintenance problem. Finally, in Subsection 3.4, we will give an improved solution for the dynamic parenthesis maintenance problem.

3.1 Reducing the prefix lookup problem to nearest marked ancestor queries

The dynamic *nearest marked ancestor query problem* is the on-line problem of satisfying the following requests on a rooted tree S

Insert leaf v**:** Insert leaf v into tree S.

Delete leaf v**:** Delete leaf v from tree S.

Mark node v**:** Mark node v in tree S.

Unmark node v**:** Unmark node v in tree S.

Nearest marked ancestor of v **:** Return the node w in tree S such that w is a marked ancestor of v, and no internal node on the shortest path from w to v is marked.

A node is an ancestor of itself. We assume that the root is always marked, so that every node has a nearest marked ancestor. In the semi-dynamic version, Delete and Unmark are not supported.

Given an algorithm to solve the *nearest marked ancestor query problem*, the pseudocode below shows how to compute each operation of the *dictionary prefix lookup problem.* We initialize by letting S be a tree with just one marked root node and no edges.

We will maintain the invariant that S is a trie of the dictionary. A nonroot node in S will be marked precisely when the pattern prefix it represents is a dictionary pattern. In particular, each leaf will be marked; if pattern P is a prefix of another pattern Q, the internal node representing the (shorter) pattern P will also be marked.

```
INSERT(P)
    Search down the trie S to find V, the longest common prefix of P with a dictionary pattern.
    Let node v be the node with label V.
    i ← depth(v) + 1,
    cur ← v
    while i ≤ |P| do {
        Insert (using the algorithm for marked tree insertions) a new leaf l below cur.
        Label the edge from cur to l with P[i].
        i ← i + 1.
        cur ← l.
    }
    MARK(cur)

DELETE(P)
    Search down the trie S to find cur as the node whose label is P.
    UNMARK(cur).
    do {
        If cur is marked, exit do loop.    /* cur is another pattern*/
        If cur is not a leaf, exit do loop.    /* cur is a prefix of another pattern*/
        par ← parent(cur).
        Delete cur from T.
        cur ← par.
    }
```

When we reach a position of a text, we pretend that the next symbol is a \$. If we can make a *goto* 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 from longest pattern to the shortest.

Here is the pseudocode for searching from [19].

```
SCAN(T[1...t])
    state ← λ
    for i ← 1 to n do
        While goto(state, t_i) is not defined {
            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)
        /* Report all non-empty patterns */
        While temp ≠ $ do {
        /* Since temp ends in $ we have matched a pattern */
            Print the pattern associated with temp
        /* See if any smaller patterns match */
            temp ← fail(temp)
        }
```

Suppose we are searching the text *abaabbbbb* for the occurrences of the patterns in the dictionary $\hat{D}$ of Example 2.13. After reading the prefix *abaab* we will be in the normal state *aab*. When we pretend to read \$ as the next symbol, we will set $temp \leftarrow aab\$$. 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. Since we keep track of the state *aab*, we can continue our search by reading the next symbol *b* from the text.

The main result we need for Section 4 is:

Theorem 2.14 [19] The IS algorithm solves the dynamic dictionary problem in time bounds:

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

Update of Pattern P: $O(p \log d)$

Scanning a Text T: $O(t + tocc) \log d)$

3 Parenthesis Maintenance Algorithms

As noted in Section 2, the bottleneck in suffix tree based dynamic dictionary computation is the dictionary prefix lookup problem. In the following subsections, we reduce the dictionary prefix lookup problem to the *dynamic parenthesis maintenance* problem and give a faster solution to the latter problem. We first introduce an intermediate problem called the *nearest marked ancestor query* problem. In Subsection 3.1, we will reduce the dictionary prefix lookup problem to the nearest marked ancestor query problem. In Subsection 3.2, we use the previous reduction to derive results for the semi-dynamic dictionary problem. In Subsection 3.3, we will reduce the nearest marked ancestor query problem to the dynamic parenthesis

2. $fail(x) = w$ if and only if the parenthesis pair corresponding to w and $*w$ are the nearest enclosing parentheses of the $'('$ corresponding to x.

It follows from Lemma 2.11 that the computation of $fail$ in the main search loop of the IS algorithm can be reduced in constant time to a nearest enclosing parenthesis query. Furthermore, the improved parenthesis maintenance algorithm we will give in Section 3 directly improves the search time of the IS algorithm. However, it does not improve the insertion or deletion times of the IS algorithm. The connection between the AFGGP algorithm and parenthesis maintenance is less direct, but once we establish the connection in Section 3, we can get improvements in *both* the search time and the update time.

We will not need most of the details of the insertion and deletions algorithms of [19], but we will need to know a little bit about their structure. 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 in $x \in S$, 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. As a consequence, we can define the following new operation on the dictionary, which we will use later in two dimensional matching:

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

The IS algorithm can support the above operation in $O(\log d)$ time.

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.

We can similarly add the following new operation which can be supported by the IS algorithm in $O(\log d)$ time. We use this later in two dimensional matching.

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

From the analysis of insertion and deletion in [19] we obtain:

Corollary 2.12 The worst-case time to insert $P[1 \ldots i-1]$ is $O(\log d)$ per character. If we append an i^{th} character and wish to extend the insertion, the time needed for the extra insertion is $O(\log d)$. The worst-case time to delete $P[1 \ldots i-1]$ is $O(\log d)$ per character.

Finally we describe how the IS algorithm recognizes patterns. Suppose that similarly to the AFGGP algorithm we append a \$ to each full pattern when storing it. The character \$ is not used except at the end of strings. Define a string to be a *normal prefix* if it is a prefix of some pattern in D. For each normal prefix x that is not a full pattern, also define an *extended prefix*, $x\$$, by appending the character \$. We define $goto(x, \$) = x\$$, but the $goto$ function is not defined when the first argument is an extended prefix. The $fail$ function is defined for extended prefix arguments, so that $fail(x\$)$ is either the longest pattern that is a suffix of $x\$$ or the special string \$ if no pattern is a suffix. Given D, let $\hat{D}$ be the normal, extended, and complementary prefixes for D.

Example 2.13 Suppose that $D = \{b\$, aab\$\}$, where we have appended a \$ to each pattern. The list of normal prefixes and complementary prefixes of $\hat{D}$ in the inverted order is:

$$\lambda, a, aa, *aa, *a, b, aab, *aab, *b, b\$, aab\$, *aab\$, *b\$, *.$$

The list of the extended prefixes and their complements in inverted order is:

$$\$, a\$, aa\$, *aa\$, *a\$, *\$.$$

By definition $fail(aa) = a$ and $fail(aab\$) = b\$$.

```
while goto(state, symbol) is not defined do {
    state ← fail(state)
}
state ← goto(state, symbol)
symbol ← nextsymbol
```

In [19], Idury and Schäffer noted that the key to making the AC algorithm dynamic is to quickly compute and update the failure function. We summarize the aspects of their algorithm that will be relevant in Section 4.

Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ with each $s_i \in \Sigma^*$, define a set of *complementary strings* $S^* = \{*s_1, *s_2, \ldots, *s_k\}$ where $* \notin \Sigma$ and $*$ is the lexicographically greatest element of $\{*\} \cup \Sigma$. For strings α and β, we write that $\alpha < \beta$ if α is lexicographically less than β. Let α^R be the reverse of string α. Also define the total order $<_{inv}$ so that $\alpha <_{inv} \beta$ if and only if $\alpha^R < \beta^R$.

Example 2.6 Suppose $\Sigma = \{a, b\}$, with $a < b$. Suppose $S = \{aa, bb, aaa, abb, aba\}$. Then the set $S \cup S^*$ listed in $<_{inv}$ order is:

$$aa, aaa, *aaa, *aa, aba, *aba, bb, abb, *abb, *bb.$$

In the IS algorithm, the set S is the set of all pattern prefixes. The null prefix, λ, corresponding to the start state is included in S, The null prefix and its complementary string $*$, are correspondingly the least and greatest elements in the order $<_{inv}$ and enclose all other strings. The following lemmas summarize the properties of the $<_{inv}$ order that we will need later.

Lemma 2.7 *x is a proper suffix of y if and only if $x <_{inv} y <_{inv} *y <_{inv} *x$.*

Proof: Suppose x is a proper suffix of y. Then x^R is a proper prefix of y^R. Since any character in y is considered less than $*$, we have $x <_{inv} y <_{inv} *y <_{inv} *x$.

Suppose $x <_{inv} y <_{inv} *y <_{inv} *x$. By definition of $<_{inv}$, we have $x^R < y^R < y^R* < x^R*$. Since y^R is lexicographically between x^R and x^R*, and $*$ is the maximum character, x^R must be a proper prefix of y^R. Equivalently, x is a proper suffix of y. ∎

Lemma 2.8 By storing $S \cup S^*$ under the $<_{inv}$ total order in a Dietz-Sleator list, we can determine the relative order of two prefixes in $O(1)$ time. More importantly, we can test whether x is a suffix of y in $O(1)$ time. This holds even when the sets are dynamically changing due to insertions and deletions.

Proof: Using the characterization of Lemma 2.7 it suffices to test whether $x <_{inv} y <_{inv} *y <_{inv} *x$. The middle $<_{inv}$ ordering always holds, and the outer two can be tested in $O(1)$ time using the *L_Order* operation. ∎

Lemma 2.9 ([19]) Suppose we keep the union of the pattern prefixes in S and complementary prefixes in S^* stored in $<_{inv}$ order. Then for $x \in S$, $fail(x) = w$ if and only if $w <_{inv} x <_{inv} *x <_{inv} *w$ and there is no pattern prefix $y \in S$ such that $w <_{inv} y <_{inv} x <_{inv} *x <_{inv} *y <_{inv} *w$.

Lemma 2.10 ([19]) There is a representation of the $<_{inv}$ order that allows *fail* to be computed in $O(\log d)$ time.

The following Lemma connects the $<_{inv}$ order and failure function computations to the parenthesis maintenance problem.

Lemma 2.11 ([19]) 1. If we replace a regular prefix with a '(' and its complementary prefix with a ')', then the prefixes of $S \cup S^*$ in the $<_{inv}$ order yield a list of well-balanced parentheses.

Lookup (j,k)**:** $O(|out_{(j,k)}| \log d)$, where $|out_{j,k}|$ is number of patterns in the output.

using the method of [5]. Comparing the above time bounds to the time bounds for the dictionary representative problem, we see that these time bounds have an extra multiplicative factor of $\log d$. Thus we can improve the overall time for solving the dynamic dictionary problem solely by finding a faster solution to the the dictionary prefix problem.

In section 3, we will present faster algorithms for the dictionary prefix problem. We summarize the effect of such algorithms on the running time for (semi-)dynamic dictionary matching as follows.

Lemma 2.4 Suppose the (semi-)dynamic dictionary prefix problem can be solved in times:

Dictionary Preprocessing: $O(T_\beta(d,\sigma))$

Insert/Delete P**:** $O(T_\delta(p,d))$

Lookup (j,k)**:** $O(T_\lambda(|out_{(j,k)}|,d))$.

Then the (semi-)dynamic dictionary matching problem can be solved in times:

Dictionary Preprocessing: $O(d\log\sigma + T_\beta(p,\sigma))$

Insert/Delete P**:** $O(p\log\sigma + T_\delta(p,d))$

Scan T**:** $O(t\log\sigma + \sum_{i=1}^{t} T_\lambda(|out_{h_i}|,d))$, where $out_{h_i} = out_{(j,k)}$, when $h_i = P_j[1\ldots k]$.

Combining the previous lemma and the bounds given above from [5] for the dynamic dictionary prefix problem, the main result of that paper follows:

Corollary 2.5 ([5]) The dynamic dictionary matching problem can be solved in time

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

Insert/Delete P**:** $O(p\log d)$

Scan T**:** $O((t+tocc)\log d)$,where $tocc$ is the total number of pattern occurrences.

The semi-dynamic dictionary matching problem is not considered in [5], but of course, their results for the fully dynamic problem apply to the semi-dynamic case.

2.2.2 Automata Methods

In [1], Aho and Corasick, AC for short, solved the static dictionary matching problem — in which the dictionary is given initially and cannot change — via an automaton method which extended the work for single string matching of Knuth, Morris and Pratt [22]. The AC algorithm constructs an automaton in which each state represents some prefix of a pattern in D, the dictionary. Throughout Section 4, we will use the prefix to mean its state, and vice versa, as long as there is no ambiguity. The prefix x at the current state is the longest pattern prefix matching a suffix of the text scanned so far.

Aho and Corasick defined two important partial functions, $goto$ and $fail$, that describe the transitions in the automaton. $goto(x,a) = xa$ if xa is a prefix of some pattern in the dictionary. For any non-null prefix x, $fail(x) = w$ where w is the longest prefix of some pattern such that w is a proper suffix of x. The basic AC search loop is:

Insert/Delete P: $O(p \log \sigma)$

Scan T: $O(t \log \sigma)$

using suffix trees [5]. The AFGGP algorithm includes a subroutine called SEARCH that finds $h_1, h_2, \ldots$.

We need to change the representation that SEARCH uses to report h_i. To explain the change we need to recall a few definitions from [5].

Given two strings x and y such that x is a prefix of y, define $y - x$ to be the suffix of y obtained by deleting x. If the suffix tree T_D has a node α with $L(\alpha) = x$, then α is called the *locus* of x. For any substring x, the *contracted locus* of x is the longest prefix of x that is the label of some node in T_D.

The AFGGP algorithm solves the dictionary representative problem in increasing order of the index i. The answer h_i is represented by a pair $(clocus, s)$ such that:

1. The internal node *clocus* is the contracted locus of h_i.
2. The suffix $s = h_i - L(clocus)$.

We instead would like h_i reported as a pair (j, k) such that $h_i = P_j[1, \ldots, k]$. To enable the change we can store with each edge $\alpha \rightarrow \beta$ of T_D a pair (a, b) such that:

1. $L(\alpha) = P_a[1 \ldots b]$
2. The label on the edge $\alpha \rightarrow \beta$ gives the characters $b + 1, \ldots$ of pattern P_a.

In constructing the pair $(clocus, s)$, SEARCH identifies a suffix tree node, call it α, such that $clocus = L(\alpha)$ and an outgoing edge $\alpha \rightarrow \beta$ such that s is a prefix of the label on that edge. If (a, b) is the pair associated with $\alpha \rightarrow \beta$, then we amend SEARCH to report h_i as the pair $(a, b + |s|)$ instead of as the pair (α, s).

The SEARCH subroutine, and the corresponding parts of the dynamic dictionary insert and delete, use *only the suffix tree* without referring to the forest F_D. The forest is used to actually find the patterns that are prefixes of the dictionary representative; the AFGGP algorithm updates F_D in pattern insertion and deletion only because F_D is needed for finding pattern occurrences.

We now turn to the second subproblem, which is that of finding the dictionary patterns which are prefixes of the representative h_i. We define the *dictionary prefix* problem as follows:

Preprocess $P_1, \ldots, P_k$: Insert all patterns P_i into an initially empty dictionary D.

Insert P: Insert pattern P into the (possibly empty) dictionary D.

Delete P: Delete pattern P from the dictionary D.

Lookup (j, k): Let P be the jth dictionary pattern (under some standard numbering of dictionary patterns). Output all dictionary patterns which are prefixes of $P[1, \ldots, k]$.

In the semi-dynamic version, Delete is not supported.

Example 2.3 Suppose $D = \{P_1, P_2, P_3\}$ where $P_1 = aba, P_2 = aa, P_3 = aaba, P_4 = abaab$. Then Lookup$(3, 4) = \{P_3, P_2\}$, Lookup$(4, 4) = \{P_1\}$, and Lookup$(4, 2) = \emptyset$

The dictionary prefix problem does not involve the text at all. The dictionary prefix problem can be solved in time:

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

Insert/Delete P: $O(p \log d)$

A *compacted trie* T' is obtained from T by collapsing paths of internal nodes with only one child into a single edge and by concatenating the labels of the edges along the path to form the label of the new edge. The label of an edge in T' is a nonempty substring of some s_j, and it can be succinctly encoded by the starting and ending positions of an occurrence of the substring. The number of nodes of a compacted trie is $O(r)$.

Let $S[1,m] = a_1 a_2 \cdots a_{m-1}\$$ be a string, where the special character \$ is not in Σ. The *suffix tree* T_S of S is a compacted trie for all suffixes of S. Since \$ is not in the alphabet, all suffixes of S are distinct and each suffix is associated with a leaf of T_S. There are three well-known papers that describe linear time algorithms for building suffix trees [28, 23, 13]. The AFGGP algorithm uses the variant of suffix trees proposed by McCreight [23].

The AFGGP algorithm maintains two major data structures. One is a McCreight suffix tree, called T_D, for the concatenated string $P_1\$_1 P_2\$_2 \ldots P_n\$_n$, where n is the number of patterns. The subscript D refers to the dictionary.

The other data structure is a forest of trees F_D in which each tree is isomorphic to a particular subtree of the suffix tree. Our improvement replaces the F_D structure with a trie on the set of patterns $\{P_1, P_2, \ldots, P_n\}$. We will represent the trie implicitly based on its Euler tour. Thus we will replace the AFGGP operations on F_D with operations on our trie. The operations on T_D will not change, except that we will use an alternative notation for pattern prefixes.

In the AFGGP algorithm, the problem of dynamically maintaining a dictionary and matching the dictionary against a text is implicitly broken down into two subproblems. We call the first problem the *dictionary representative problem* and the second problem the *dictionary prefix problem.* As we shall see below, the time bottleneck in the AFGGP algorithm is the dictionary prefix problem, and that is the problem we solve differently. These two problems were not explicitly defined in [5], but within each of the insert, delete, and search procedures therein, the two problems were addressed separately. The division into two problems will facilitate the presentation of Section 3.

Define the *dictionary representative problem* to be that of satisfying the following on-line requests:

Preprocess $P_1, \ldots, P_k$: Insert all patterns P_i into an initially empty dictionary D.

Insert P: Insert pattern P into the dictionary D.

Delete P: Delete pattern P from the dictionary D.

Match T: Find, for each text location i, the pair (j,k) such that $P_j[1,\ldots,k]$ is the *longest* pattern prefix that matches T at position i.

In the semi-dynamic version, deletion is not supported.

We call the longest matching pattern prefix the *representative* for a text location i. In [5] this representative is denoted h_i. When the representative h_i is the empty string (corresponding to the root of the suffix tree), it can be denoted by the pair $(j,0)$ for any pattern number j. If the representative is the prefix of multiple patterns, then there are choices as to which pattern index j we use in the first component. Our algorithm does not depend on how the choice of pattern index j is made.

Example 2.2 Suppose $D = \{P_1, P_2, P_3\}$ where $P_1 = aba, P_2 = aa, P_3 = aaba$. Suppose our text is $T = aaabaabbaa$, then the matches we seek are: $h_1 = (2,2)$, $h_2 = (3,4)$, $h_3 = (1,3)$, $h_4 = (1,0)$, $h_5 = (3,3)$, $h_6 = (1,2)$, $h_7 = (1,0)$, $h_8 = (1,0)$, $h_9 = (2,2)$, $h_{10} = (1,1)$.

The *dictionary representative problem* can be solved in time:

Dictionary Preprocessing: $O(d \log \sigma)$, where d is the sum of all the pattern lengths.

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.

Finally we state a theorem from [14] which will be used in Section 3. Dietz and Sleator use it to show that their scheme of handling overflows achieves the desired time bounds. We use the theorem similarly to obtain the time bounds for our scheme of handling overflows in trees for parentheses maintenance.

Theorem 2.1 ([14]) Let $x_1, \ldots, x_n$ be n real valued variables, all initially zero. Repeatedly perform the following procedure:

1. Find an i, $1 \leq i \leq n$, such that $x_i = \max_j \{x_j\}$. Set x_i to zero.
2. Pick n nonnegative reals $a_1, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i = 1$.
3. For $i = 1, \ldots, n$, set x_i to $x_i + a_i$.

No x_i will ever exceed $H_{n-1} + 1$, where $H_k = \sum_{i=1}^{k} i^{-1}$, the kth harmonic number.

2.2 Review of dictionary matching algorithms

Two basic approaches have been used in previous work on dictionary matching: suffix tree methods [3, 5], and automata methods [1, 4, 19]. For our purposes, it suffices to review some aspects of one algorithm from each group; we choose the AFGGP algorithm [5] from the suffix tree group and the IS algorithm [19] from the automata group. Our improved algorithm for one dimensional matching is a modification of the AFGGP algorithm. Our algorithm for two dimensional matching is based on the IS algorithm. We do not know if it is possible to obtain both improvements using only one approach.

2.2.1 Suffix tree based algorithms

A *trie* T for a set of strings $\{s_1, \cdots, s_r\}$ is a rooted directed tree satisfying:

1. Each edge is labeled with a character, and no two edges emanating from the same node have the same label.
2. Each node v is associated with a string, denoted by $L(v)$, obtained by concatenating the labels on the path from the root to v.
3. There is a node v in T if and only if $L(v)$ is a prefix of some string s_j in the set.

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

In [4], Amir and Farach presented a (static) multiple matching algorithm with similar preprocessing and scanning complexities of:

1. **Dictionary Preprocessing:** $O(d \log n)$, where n is the number of patterns
2. **Text Scanning:** $O((t + tocc) \log n)$, where t is the area of the text.

Giancarlo [17] 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 [4], 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, as done in [4], 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 nonstandard subrow/subcolumn alphabet.

In a more recent paper, Idury and Schäffer [20] considered a more general two dimensional problem where the patterns can be rectangles of varying heights, widths, and aspect ratios. They obtained the first non-trivial bounds in both the static and dynamic cases, but the bounds are not as good as those presented here for the case of square patterns.

The rest of the paper is structured as follows. In Section 2 we review the relevant aspects of previous work on dynamic dictionary matching and parts of a data structure paper by Dietz and Sleator [14] that we use later on. In Section 3 we describe a new algorithm for balanced parenthesis maintenance which is used to speed up the one-dimensional dynamic dictionary matching algorithm; we also explain how a known data structure for semi-dynamic balanced parenthesis maintenance can be used to solve semi-dynamic dictionary matching. In Section 4, we present an algorithm for two dimensional dynamic dictionary matching.

2 Description of previous relevant work

2.1 The Dietz-Sleator data structure

We use the data structure of Dietz and Sleator [14] for the *order maintenance problem* in solving both the balanced parentheses maintenance problem and the two dimensional dynamic dictionary matching 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 distinguish them from the dictionary operations.

$L\_Insert(x, y)$: Insert a new element y 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 [14] provide a 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*.

We now give a brief description of the data structure of Dietz and Sleator. 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 [30, 31] on the children of the root. The algorithm assigns file addresses to children

A parallel algorithm for this problem was also given.

In a related full paper Amir, Farach, Galil, Giancarlo and Park (AFFGP for short) presented a modified version of the AF algorithm which improved the deletion time to *worst case* time $O(p \log d)$ [5].

Idury and Schäffer (IS for short) showed that the failure function approach and basic scanning loop of the AC algorithm can be adapted to dynamic dictionary matching [19]. They improved the initial dictionary preprocessing time to $O(d \log \sigma)$. The insertion, deletion and text scanning times are as in [5]. They also showed that faster search time can be achieved at the expense of slower dictionary update time. For fixed and constant k, the text scanning can be done in linear time $O(t(k + \log \sigma) + tocc \cdot k)$ and the dictionary update in time $O(p(kd^{1/k} + \log \sigma))$.

The two main contributions of this paper are: 1) A faster algorithm for dynamic dictionary matching with bounded alphabets, and 2) A two dimensional dynamic dictionary matching algorithm.

The first contribution is based on an algorithm that solves the general problem of maintaining a sequence of well-balanced parentheses under the operations insert, delete, and find nearest enclosing parenthesis pair. The parenthesis maintenance problem was previously considered by Güting and Wood [18], who applied it to problems in computational geometry. Our results improve theirs slightly.

We show that our parenthesis maintenance algorithm also provides an improved solution to the problem of maintaining a tree with marked nodes under the operations: insert leaf, delete leaf, mark node, unmark node, and find nearest marked ancestor. This tree maintenance problem is closely related to a data structure problem that arises in bottom-up tree pattern matching [11, 12]. It is also closely related to data structures problems that arise in the maintenance of dynamic graphs [29].

Faster string dictionary matching: All previous dynamic dictionary matching algorithms have a text scanning time of $O(t \log d)$ with a dictionary update time of $O(p \log d)$. We show that for a bounded alphabet this time can be improved by a $\log \log d$ factor to:

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

Dictionary Update: $O(p(\log d / \log \log d + \log \sigma))$

Text Scanning: $O(t(\log d / \log \log d + \log \sigma) + tocc \cdot \log d / \log \log d)$

For an unbounded alphabet, even the AC algorithm, which does not support dictionary updates, takes time $O(t \log d)$ in the worst case for text scanning. If we assume that the characters in the unbounded alphabet are represented by some bit pattern, it may take as much as $O(\log d)$ time to decode each character. Thus our main hopes for improving the previous dynamic dictionary matching bounds lie in the bounded alphabet case.

Two Dimensional Dictionary Matching: The two dimensional problem we consider has a rectangular text and square patterns. It can be defined by:

- *INPUT*: A set of square patterns $P_1, P_2, \ldots, P_n$ of total area d, and a rectangular text T of area t, all over an alphabet Σ.
- *UPDATE*: Insert or delete a pattern P_i.
- *SEARCH OUTPUT*: All ordered triples (r, c, j) such that pattern P_j matches the text when the upper left corner of P_j is matched with the row r and column c of the text.

In this paper we present a two dimensional dynamic dictionary matching algorithm with the following time bounds:

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

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

1 Introduction

The classical *pattern matching problem* is that of searching for a single pattern in a given text. When the pattern and text are strings, the *fixed string matching* problem can be defined as:

- *INPUT*: A *fixed* pattern $P = p_1, \ldots, p_m$ and a text $T = t_1, \ldots, t_n$ over an alphabet Σ.
- *OUTPUT*: All text locations i such that $t_{i+k} = p_{1+k},\ k = 0, \ldots, m-1$.

The goal is to preprocess the pattern quickly so that the text search is fast. The fixed string matching problem is one of most studied problems in computer science and has many different linear time solutions (e.g., [10, 22] and many others). The two dimensional version of this problem in which strings of symbols are replaced with matrices is also well studied [9, 8, 7, 2, 16].

One generalization of fixed pattern matching is *multiple matching*. The input is a *set* of patterns D and a text. The object is to find all occurrences of all patterns from the set D that appear in the text. We can define *multiple string matching* by:

- *INPUT*: A set of patterns $P_1, P_2, \ldots, P_k$ of total length d, and a text $T = t_1, \ldots, t_n$ all over an alphabet Σ.
- *OUTPUT*: All ordered pairs (i, j) such that pattern P_j matches the piece of text beginning at location t_i.

An early motivation for multiple string matching was given by Aho and Corasick [1]. They solved a bibliographic search problem where D was a set of words. An implicit assumption in [1] is that the text is relatively long, whereas the set of patterns is relatively small and used only once. Aho and Corasick(AC for short) gave an $O((t+d)\log\sigma + tocc)$ time algorithm for the multiple string matching problem, where the text is of length t, the sum of all pattern lengths is d, the total number of pattern occurrences reported is $tocc$, and σ is the number of distinct characters that occur in D. A parallel algorithm for multiple string matching where all the patterns are of equal length was given by Kedem, Landau and Palem [21].

Recently there has been interest in a further generalization of multiple matching called *dynamic dictionary matching*. As in multiple matching, we seek all appearances of any of a set of patterns (the *dictionary*) in an input text. In dictionary matching, the set of patterns D can change over time by the insertion or deletion of single patterns. Semi-dynamic dictionary matching, in which only insertions are allowed, was introduced by Meyer [25].

The AC algorithm solves part of the desired dynamic dictionary matching problem. They preprocess the pattern set in time $O(d\log\sigma)$ and subsequently search any given length-t text in time $O(t\log\sigma + tocc)$. However, they allow only *static* dictionaries; that is, adding or deleting a pattern string requires reprocessing the entire dictionary in time $O(d\log\sigma)$. It will always be the case that $d \geq \sigma$, so terms depending on σ are sometimes omitted.

Amir and Farach (AF for short) showed a suffix tree based method for solving the dynamic dictionary string matching problem [3]. The complexity of their algorithm is:

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

Adding a pattern P to the dictionary: $O(p\log d)$, where $p = |P|$

Deleting a pattern P from the dictionary: Amortized time $O(p\log d)$

Scanning a text T: $O((t+tocc)\log d)$

Improved Dynamic Dictionary Matching

Amihood Amir [*] Georgia Tech

Martin Farach [†] DIMACS

Ramana M. Idury [‡] Rice University

Johannes A. La Poutré [§] Princeton University

Alejandro A. Schäffer [¶] Rice University

November 29, 1993

Abstract

In the dynamic dictionary matching problem, a dictionary D contains a set of patterns that can change over time by insertion and deletion of individual patterns. The user also presents text strings and asks for all occurrences of any patterns in the text.

The two main contributions of this paper are: 1) A faster algorithm for dynamic string dictionary matching with bounded alphabets, and 2) A dynamic dictionary matching algorithm for two dimensional texts and patterns. The first contribution is based on an algorithm that solves the general problem of maintaining a sequence of well-balanced parentheses under the operations insert, delete, and find nearest enclosing parenthesis pair. The main new idea behind the second contribution is a novel method to efficiently manipulate failure links for two-dimensional patterns.

Address for correspondence: Martin Farach; Department of Computer Science, Rutgers University; Piscataway, NJ 08855 USA

[*] amir@cc.gatech.edu; Partially supported by NSF grant IRI-9013055.

[†] farach@cs.rutgers.edu; Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.

[‡] idury@hto-b.usc.edu; Partially supported by a grant from the W. M. Keck Foundation.

[§] hanlp@cs.ruu.nl; Supported by a NATO Science Fellowship awarded by N.W.O., and partially supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.

[¶] schaffer@cs.rice.edu; Partially supported by NSF grant CCR-9010534.