Contents

Chapter 9

Tabular parsing

Tabular parsing algorithms: derive all true propositions about the input string and its constituents, storing them in a table. For grammars describing substrings, such as the ordinary grammars, given an input string $w = a_1 \dots a_n$, the possible propositions about its constituents are of the form $A(a_{i+1} \dots a_j)$, where A is a category symbol, and i and j are any positions in w. An algorithm may store such propositions in a table indexed by i and j, where the symbol A is put into an entry (i, j).

As these algorithms derive *all* propositions about a string, they are not affected by rules with multiple premises, and hence work for conjunctive and Boolean grammars in the same way as for ordinary grammars.

9.1 Cubic-time parsing

The simplest parsing method for ordinary grammars was independently discovered by Cocke, by Kasami [6] and by Younger [15], and is therefore known as the Cocke–Kasami–Younger algorithm (occasionally, Cocke–Younger–Kasami). The algorithm applies to conjunctive and Boolean grammars with minimal modifications.

9.1.1 The algorithm

Let $G = (\Sigma, N, R, S)$ be a Boolean grammar in binary normal form, let $w = a_1 \dots a_n$ be an input string. The simple cubic-time parsing algorithm constructs the *parsing table* $T \in (2^N)^{n \times n}$, with each element $T_{i,j}$ with $0 \leqslant i < j \leqslant n$ representing the set of nonterminals that generate the substring between the positions $i + 1$ and j:

$$T_{i,j} = \{ A \in N \mid a_{i+1} \dots a_j \in L_G(A) \}.$$

The elements of this table can be computed inductively on the length $j - i$ of the substring, beginning with the elements $T_{i-1,i}$, each depending only on the symbol a_i, and continuing with larger and larger substrings, until the element $T_{0,n}$ is computed. As the basis of the induction, let

$$T_{i-1,i} = \{ A \mid A \to a_i \in R \}.$$

For the induction step, consider first the set of all pairs (B, C), with $B, C \in N$, for which the concatenation BC generates the substring $a_{i+1} \dots a_j$.

$$P_{i,j} = \{ (B, C) \mid B, C \in N,\ a_{i+1} \dots a_j \in L_G(BC) \}$$

This value can be calculated as

$$P_{i,j} = \bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j},$$

where the position k represents a cutting point of the string $a_{i+1} \dots a_j$ into two substrings: $a_{i+1} \dots a_k$ generated by B and $a_{k+1} \dots a_j$ generated by C, and the information on those shorter substrings is acquired from the sets $T_{i,k}$ and $T_{k,j}$, which must have been computed before. From the set $P_{i,j}$, one can determine the set of nonterminals generating the same substring as

$$T_{i,j} = f(P_{i,j}),$$

where the function $f\colon 2^{N \times N} \to 2^N$, defined by

$$f(P) = \{\, A \mid \exists A \to BC \in R :\ (B, C) \in P \}$$

for ordinary grammars, by

$$f(P) = \{\, A \mid \exists A \to B_1 C_1 \,\&\, \dots \,\&\, B_m C_m \in R : (B_t, C_t) \in P \text{ for all } t \}$$

for conjunctive grammars, and by

$$f(P) = \{ A \mid \exists A \to B_1 C_1 \,\&\, \dots \,\&\, B_m C_m \,\&\, \neg D_1 E_1 \,\&\, \dots \,\&\, \neg D_{m'} E_{m'} \in R : \\ (B_t, C_t) \in P \text{ and } (D_t, E_t) \notin P \text{ for all } t \}$$

for Boolean grammars, represents the logic in the rules of the grammar.

Algorithm 9.1 constructs the table $T_{i,j}$ according to the above equations.

Algorithm 9.1 The Cocke–Kasami–Younger algorithm for Boolean grammars

Let $G = (\Sigma, N, R, S)$ be a Boolean grammar in the binary normal form. Let $w = a_1 \dots a_n$, where $n \geqslant 1$ and $a_i \in \Sigma$, be an input string. For all $0 \leqslant i < j \leqslant n$, let $T_{i,j}$ be a variable ranging over subsets of N, and let P be a variable ranging over subsets of $N \times N$.

```
for i = 1 to n do
        T_{i-1,i} = { A | A → a_i ∈ R }
for ℓ = 2 to n do
        for i = 0 to n − ℓ do
                let j = i + ℓ
                let P = ∅
                for all k = i + 1 to j − 1 do
                        P = P ∪ (T_{i,k} × T_{k,j})
                T_{i,j} = f(P)
accept if and only if S ∈ T_{0,n}
```

For ordinary grammars, the algorithm can be slightly simplified. In this case, the function f is defined as $f(P) = \{\, A \mid \exists (B, C) \in P :\ A \to BC \in R \}$, and it is distributive over union: $f(P \cup P') = f(P) \cup f(P')$. Therefore, it is not necessary to accummulate all pairs in the set P before applying the function f. It can be applied directly to each Cartesian product, as in Algorithm 9.2.

9.1.2 Constructing a parse tree

The algorithm has so far been defined as a recognizer, which determines whether the string is generated by the grammar or not. If the string is found to be generated by the grammar, one would typically be interested in obtaining a parse tree of this string. Since all the necessary information is in the parsing table $T_{i,j}$, one can first build this table by Algorithm 9.1 and then

Algorithm 9.2 The Cocke–Kasami–Younger algorithm for ordinary grammars

Let $G = (\Sigma, N, R, S)$ be an ordinary grammar in the Chomsky normal form. Let $w = a_1 \dots a_n$, where $n \geqslant 1$ and $a_i \in \Sigma$, be an input string. For all $0 \leqslant i < j \leqslant n$, let $T_{i,j}$ be a variable ranging over subsets of N.

for $i = 1$ to n **do**
    $T_{i-1,i} = \{ A \mid A \to a_i \in R \}$
for $\ell = 2$ to n **do**
    for $i = 0$ to $n - \ell$ **do**
        let $j = i + \ell$
        for all $k = i + 1$ to $j - 1$ **do**
            $T_{i,j} = T_{i,j} \cup f(T_{i,k} \times T_{k,j})$
accept if and only if $S \in T_{0,n}$

Algorithm 9.3 Parse tree construction procedure for ordinary grammars

Let $G = (\Sigma, N, R, S)$ be an ordinary grammar in Chomsky normal form, let $w = a_1 \dots a_n$, with $n \geqslant 1$ and $a_i \in \Sigma$, be an input string, and let the sets $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L_G(A) \}$ be available for all $0 \leqslant i < j \leqslant n$. Then, for every substring $a_{\ell+1} \dots a_m$ and for every symbol $A \in T_{\ell,m}$ generating this substring, the following procedure constructs a parse tree of $a_{\ell+1} \dots a_m$ from A.

Procedure $parse(A, \ell, m)$:

if $m - \ell = 1$ **then**
    return tree with root $A \to a_m$ connected to the leaf a_m
else
    for all rules $A \to BC \in R$ **do**
        for $k = \ell + 1$ to $m - 1$ **do**
            if $B \in T_{\ell,k}$ and $C \in T_{k,m}$ **then**
                Create a node τ labelled with $A \to BC$
                Add descendant $parse(B, \ell, k)$ to τ
                Add descendant $parse(C, k, m)$ to τ
                return τ

use another procedure to construct the parse tree according to this table. This procedure is given as Algorithm 9.3.

The running time, which is proportional to $n \cdot t$, where n is the length of the input and t is the number of nodes in the resulting tree. Since $t = \Theta(n)$, the time complexity of this step is $\Theta(n^2)$, which is smaller than the time spent constructing the parsing table.

An extension of this method to conjunctive and Boolean grammars (Theorem 9.4) in the worst case works in cubic time.

Algorithm 9.4 Parse tree construction procedure for Boolean grammars

Let $G = (\Sigma, N, R, S)$ be a Boolean grammar in Chomsky normal form, let $w = a_1 \dots a_n$, with $n \geqslant 1$ and $a_i \in \Sigma$, be an input string and let $T_{i,j} = \{ A \,|\, a_{i+1} \dots a_j \in L_G(A) \}$, with $0 \leqslant i < j \leqslant n$, be its parsing table. Once a parse tree of a substring $a_{i+1} \dots a_j$ from A is constructed, it is stored in a global variable $\tau_{A,i,j}$, and recalled whenever the same tree is needed again, in order to avoid unnecessary recomputations.
The following procedure constructs a parse tree of $a_{\ell+1} \dots a_m$ from A, assuming that $A \in T_{\ell,m}$.
Procedure $parse(A, \ell, m)$:

```
1: if τ_{A,ℓ,m} is defined then
2:         return τ_{A,ℓ,m}
3: else if m − ℓ = 1 then
4:         return tree with root A → a_m connected to the leaf a_m
5: else
6:         for all rules A → B_1C_1 & ... & B_rC_r & ¬B_{r+1}C_{r+1} & ... & ¬B_sC_s & ¬ε ∈ R do
7:                 integer p[1..s]
8:                 for t = 1 to s do
9:                         for k = ℓ + 1 to m − 1 do
10:                                if B_t ∈ T_{ℓ,k} and C_t ∈ T_{k,m} then
11:                                        p[t] = k
12:                if p[t] is defined for all t ∈ {1, ..., r} and for none of t ∈ {r + 1, ..., s} then
13:                        Create a node τ labelled A → B_1C_1 & ... & B_rC_r & ¬B_{r+1}C_{r+1} & ... & ¬B_sC_s & ¬ε
14:                        for t = 1 to r do
15:                                Add descendant parse(B_t, ℓ, p[t]) to τ
16:                                Add descendant parse(C_t, p[t], m) to τ
17:                        τ_{A,ℓ,m} = τ
18:                        return τ
```

9.2 Parsing through matrix multiplication

A parsing algorithm discovered by Valiant [14], which constructs the same table $T_{i,j}$ using Boolean matrix multiplication.

9.2.1 An opportunity to multiply matrices

Consider the cubic-time parsing algorithm from Section 9.1. The most time-consuming operation in Algorithm 9.1 is computing the sets $P_{i,j} = \bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$. If each Cartesian product is computed individually, as it is done in line 8 of the above algorithm, then spending linear time for each $P_{i,j}$ is unavoidable. The idea behind fast parsing is to rearrange the order of these operations, so that much of the work could be represented as Boolean matrix multiplication.

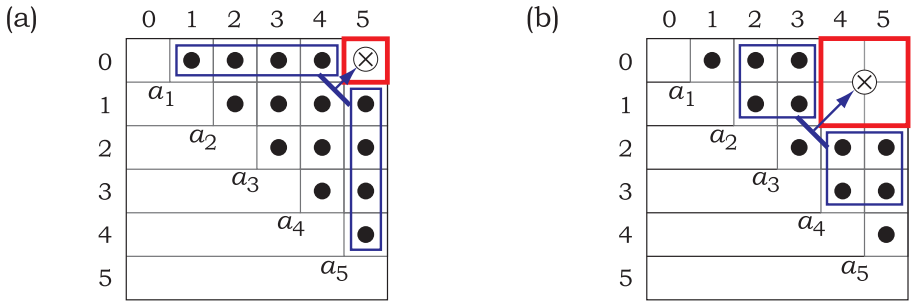

Figure 9.1: How products of submatrices of T contribute to calculating submatrices of P: (a) in the basic algorithm; (b) in Example 9.1.

To see how the work done by Algorithm 9.1 is related to matrix multiplication, consider the following adaptation of the notion of matrix product to matrices with subsets of N as elements.

Definition 9.1. *For any numbers $m, \ell, n \geqslant 1$, let $X \in (2^N)^{m \times \ell}$ and $Y \in (2^N)^{\ell \times n}$ be two matrices with subsets of N as elements. Their product $X \times Y$ is a matrix $Z \in (2^{N \times N})^{m \times n}$, with*

$$Z_{i,j} = \bigcup_{k=1}^{\ell} X_{i,k} \times Y_{k,j}.$$

Such a product can be represented as a product of $|N|^2$ pairs of Boolean matrices as follows. For every $B, C \in N$, consider the Boolean matrix Z^{BC}, where $Z^{BC}_{i,j}$ denotes the membership of the pair (B, C) in $Z_{i,j}$. Then the Boolean matrix Z^{BC} is exactly the product $X^B \times Y^C$ of the Boolean matrix X^B, which represents the membership of B in the elements of X, with the similarly defined Boolean matrix Y^C.

In the terminology of Definition 9.1, lines 6–8 of Algorithm 9.1 actually multiply a $1 \times (\ell - 1)$ submatrix of T by a $(\ell - 1) \times 1$ submatrix of T, as illustrated in Figure 9.1(a). The result is a 1×1 matrix, that is, a single set $P_{i,j} \subseteq N \times N$. Thus, matrix products used in Algorithm 9.1 are always products of row vectors with column vectors.

In order to use fast matrix multiplication, one should somehow rearrange the total bulk of operations calculated in line 8, for all applicable i, j, k, so that large blocks of Cartesian products would be calculated together as products of square matrices. The below example demonstrates such a rearrangement for input strings of length 5, which involves a product of two 2×2 submatrices of T.

Example 9.1. *Let $w = a_1a_2a_3a_4a_5$ be an input string, and consider the partially constructed parsing table depicted in Figure 9.1(b), with $T_{i,j}$ constructed for $1 \leqslant i < j \leqslant 3$ and for $3 \leqslant i < j \leqslant 5$, that is, for the substrings $a_1a_2a_3$ and $a_3a_4a_5$, as well as for their substrings.*

Then the following product of matrices of sets

$$\begin{pmatrix} T_{0,2} & T_{0,3} \\ T_{1,2} & T_{1,3} \end{pmatrix} \times \begin{pmatrix} T_{2,4} & T_{2,5} \\ T_{3,4} & T_{3,5} \end{pmatrix} = \begin{pmatrix} (T_{0,2} \times T_{2,4}) \cup (T_{0,3} \times T_{3,4}) & (T_{0,2} \times T_{2,5}) \cup (T_{0,3} \times T_{3,5}) \\ (T_{1,2} \times T_{2,4}) \cup (T_{1,3} \times T_{3,4}) & (T_{1,2} \times T_{2,5}) \cup (T_{1,3} \times T_{3,5}) \end{pmatrix} = \\ = \begin{pmatrix} X_{0,4} & X_{0,5} \\ X_{1,4} & X_{1,5} \end{pmatrix}$$

defines partial data for the following four elements of the table of pairs: $\begin{pmatrix} P_{0,4} & P_{0,5} \\ P_{1,4} & P_{1,5} \end{pmatrix}$. First of all, each of these four elements satisfies $X_{i,j} \subseteq P_{i,j}$. In particular, the set $X_{1,4}$ is exactly $P_{1,4}$. The set $X_{0,4}$ lacks the Cartesian product $T_{0,1} \times T_{1,4}$ that should be in $P_{0,4}$ by definition, and thus does not take into account the factorization $a_1 \cdot a_2a_3a_4$; actually, the set $T_{1,4}$ is not yet known

at this point, and hence the calculation of that Cartesian product has to be delayed. The element $X_{1,5}$ is symmetrically incomplete, as it lacks the Cartesian product $T_{1,4} \times T_{4,5}$ corresponding to the factorization $a_2 \cdot a_3 a_4 a_5$. Finally, $X_{0,5}$ misses two Cartesian products, $T_{0,1} \times T_{1,5}$ and $T_{0,4} \times T_{4,5}$, which can be handled only using the not yet available elements $T_{0,4}$ and $T_{1,5}$. In total, this matrix product computes 8 Cartesian products out of the 12 needed for these four elements of P, and the computation could then proceed with calculating the remaining four Cartesian products.

Already in this small example, using one product of 2×2 matrices requires changing the order of computing the elements $\{T_{i,j}\}$: the elements $T_{0,3}$ and $T_{2,5}$ need to be calculated before $T_{1,4}$. Furthermore, the subsequent computation should be arranged to take care of the four remaining factorizations, which also must be considered in a specific order, evaluating $T_{i,j} = f(P_{i,j})$ for the appropriate entries at the appropriate time. In the next section, the whole algorithm will be restated as a recursive procedure, which arranges the computation so that as much work as possible is offloaded into products of the largest possible matrices.

9.2.2 Fast matrix multiplication

Strassen's method.

A product of two 2×2 matrices is defined using 8 multiplications and 4 additions.

Lemma 9.1 (Strassen [13])**.** *Let $\mathcal{R}$ be a ring, let $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ and $B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$ be two 2×2 matrices over $\mathcal{R}$. Then their product $AB = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$ can be calculated using 7 multiplications and 18 additions in $\mathcal{R}$.*

Proof. The intended result is illustrated in Figure 9.2(left), where horizontal dashed lines represent the four elements of A, vertical dashed lines represent the four elements of B, and the intersection of any two lines represents the product of these elements. Solid lines represent sums.

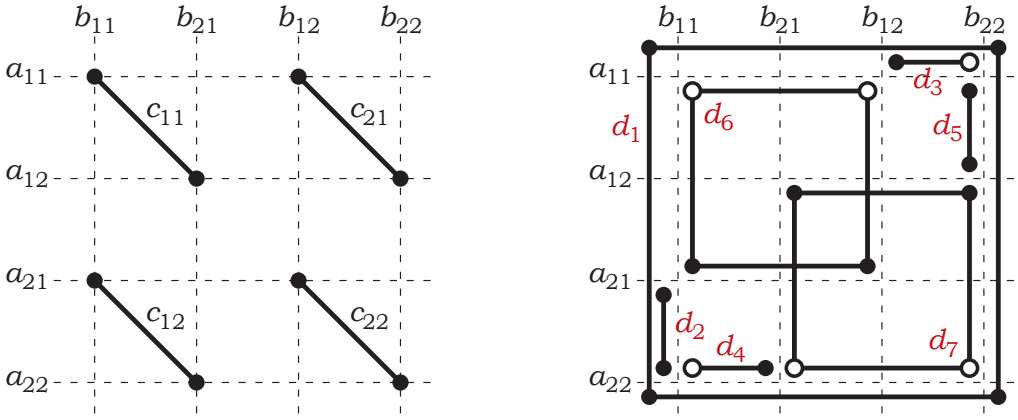

Figure 9.2: Multiplying two 2×2 matrices using 7 multiplications: (left) the eight products of elements featured in the desired matrix product; (right) Strassen's seven products.

Calculate seven products as follows:

$$\begin{aligned}
d_1 &= (a_{11} + a_{22})(b_{11} + b_{22}),\\
d_2 &= (a_{21} + a_{22})b_{11},\\
d_3 &= a_{11}(b_{12} - b_{22}),\\
d_4 &= a_{22}(b_{21} - b_{11}),\\
d_5 &= (a_{11} + a_{12})b_{22},\\
d_6 &= (a_{21} - a_{11})(b_{11} + b_{12}),\\
d_7 &= (a_{12} - a_{22})(b_{21} + b_{22}).
\end{aligned}$$

These products are illustrated in Figure 9.2(right), where a filled circle represents a product with a positive sign, while an empty circle refers to a product multiplied by -1. Then the desired matrix product AB is expressed as follows:

$$\begin{aligned}
c_{11} &= d_1 + d_4 + d_7 - d_5\\
c_{12} &= d_3 + d_5\\
c_{21} &= d_2 + d_4\\
c_{22} &= d_1 + d_3 + d_6 - d_2
\end{aligned}$$

□

Then two $2^k \times 2^k$ matrices over a ring $\mathcal{R}$ can be multiplied in 7^k operations. Consequently, two $n \times n$ matrices can be multiplied in $O(n^{\log_2 7})$ operations.

For Boolean $n \times n$ matrices: change them to 0/1 matrices in the ring of residues modulo $n+1$, then multiply them in this ring using $O(n^{\log_2 7})$ ring operations (Adleman, Booth, Preparata, Ruzzo [1]).

Improved version of Lemma 9.1 using 7 multiplications and only 15 additions, attributed to Paterson, see Fischer and Probert.

9.2.3 A combinatorial method of Boolean matrix multiplication

The method of Arlazarov, Dinitz, Kronrod and Faradzhev [3], known in the literature as *the method of Four Russians*.

Let A and B be two $n \times n$ Boolean matrices, the goal is to compute their product $C = AB$. Let k be a small number of the order of $\log_2 n$, and assume that n is divisible by k; if it is not, then n can be increased. Each row of the matrix A is split into $\frac{n}{k}$ vectors of size $1 \times k$, called *chunks*. The chunks in each i-th row are denoted by $A_{i,1}, \ldots A_{i,\frac{n}{k}} \in \mathbb{B}^{1\times k}$. The matrix B is split into $\frac{n}{k}$ submatrices of size $k \times n$, called *bands*, and denoted by $B_1, \ldots, B_{\frac{n}{k}} \in \mathbb{B}^{1\times k}$. Then each i-th row of C, denoted by $C_i \in \mathbb{B}^{1\times n}$, can be represented as follows,

$$C_i = \bigvee_{r=1}^{\frac{n}{k}} A_{i,r} B_r,$$

where each product of a chunk $A_{i,r}$ by the corresponding band B_r illustrated in Figure 9.3, is a $1 \times n$ row, and the disjunction symbol in the formula for C_i represents a bitwise disjunction of such rows.

The method works by pre-computing the products of all possible $1 \times k$ chunks with each band of B. For each chunk $(x_1, \ldots, x_k)$ and for each band B_r, the algorithm computes the vector-by-matrix product

$$D_r[x_1, \ldots, x_k] = (x_1, \ldots, x_k) \cdot B_r.$$

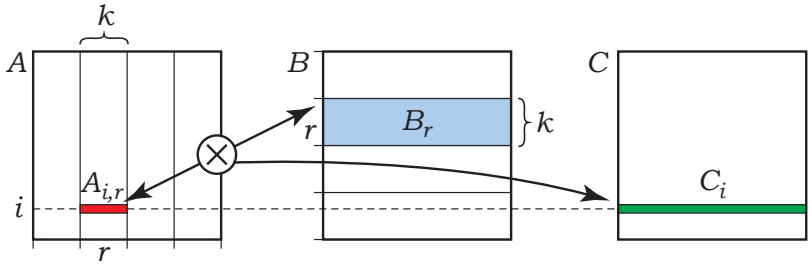

Figure 9.3: A product of a $1 \times k$ chunk $A_{i,r}$ by a $k \times n$ band B_r contributing to the i-th row of C.

and stores the resulting $1 \times n$ row in the memory. This product can actually be calculated as a bitwise disjunction of all rows of B_r corresponding to true bits of the chunk, which is a very efficient operation for typical computer hardware.

Once the pre-computation stage is complete, the algorithm calculates each i-th row of C as

$$C_i = \bigvee_{r=1}^{\frac{n}{k}} D_r[A_{i,r}],$$

that is, by looking up the pre-computed products $A_{i,r}B_r$ indexed by the contents of the chunk $A_{i,r}$ and by the band number r. This is again a bitwise disjunction of rows, which can be very efficiently implemented.

The running time of the algorithm is calculated as follows. At the pre-computation stage, there are 2^k different chunks, the matrix B contains $\frac{n}{k}$ bands, and each multiplication takes kn bit operations; therefore, the pre-computation stage requires $2^k \cdot \frac{n}{k} \cdot kn = 2^k n^2$ bit operations. The subsequent calculation of each of the n rows of C requires taking a disjunction of $\frac{n}{k}$ rows of n bits each, to the total of $\frac{n^3}{k}$ operations. Let $k = \log_2 n - \log_2 \log_2 n$. Then the overall number of operations is

$$2^k n^2 + \frac{n^3}{k} = \frac{n^3}{\log_2 n} + \frac{n^3}{\log_2 n - \log_2 \log_2 n} = O\Big(\frac{n^3}{\log_2 n}\Big).$$

A much more sophisticated method of Williams works in $O(\frac{n^3}{(\log n)^2})$ bit operations.

9.2.4 Recursive partition of matrices

Let $w = a_1 \dots a_n$ be an input string. For the time being, assume that $n+1$ is a power of two, that is, the length of the input string is a power of two minus one; this restriction can be relaxed in an implementation, which will be discussed in the next section.

The algorithm uses the following data structures. First, there is an $(n+1) \times (n+1)$ table T with $T_{i,j} \subseteq N$, as in Algorithm 9.1, and the goal is to set each entry to

$$T_{i,j} = \{\, A \mid a_{i+1} \dots a_j \in L(A) \},\qquad \text{for all } 0 \leqslant i < j \leqslant n.$$

The second table P has elements $P_{i,j} \subseteq N \times N$, each corresponding to the value of P computed by Algorithm 9.1 in the iteration $(\ell = j - i,\, i)$. The target value is

$$P_{i,j} = \{\, (B, C) \mid a_{i+1} \dots a_j \in L(B)L(C) \}\qquad \text{for all } 0 \leqslant i < j \leqslant n.$$

All entries of both tables are initialized to empty sets, and then are gradually filled by the following two recursive procedures:

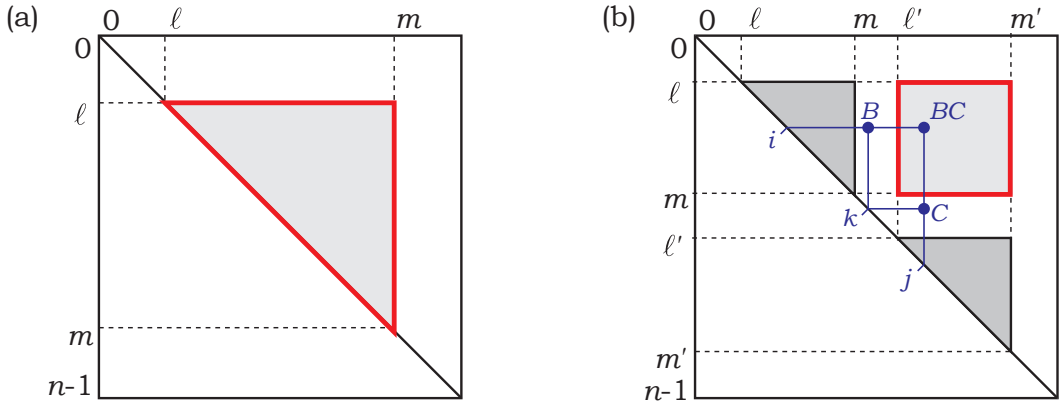

Figure 9.4: (a) The submatrix of T calculated by *compute*(ℓ, m); (b) The pre-conditions of *complete*(ℓ, m, ℓ', m') and the submatrix of T it calculates.

- The first procedure, *compute*(ℓ, m), constructs the correct values of $T_{i,j}$ for all $\ell \leqslant i < j < m$, as illustrated in Figure 9.4(a).

- The other procedure, *complete*(ℓ, m, ℓ', m'), is defined for $\ell < m \leqslant \ell' < m'$, where $m - \ell = m' - \ell'$ is a power of two. Its four arguments actually specify a submatrix of T containing all elements $T_{i,j}$ with $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$, as shown in Figure 9.4(b). The procedure assumes that the elements $T_{i,j}$ are already constructed for all i and j with $\ell \leqslant i < j < m$, as well as for all i, j with $\ell' \leqslant i < j < m'$; these are the dark grey triangles in Figure 9.4(b). It is furthermore assumed that for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$, the current value of $P_{i,j}$ is

  $$P_{i,j} = \{\, (B, C) \mid \exists k\, (m \leqslant k < \ell') :\ a_{i+1} \ldots a_k \in L(B),\ a_{k+1} \ldots a_j \in L(C) \},$$

  which is a subset of the intended value of $P_{i,j}$. This means that whenever a substring $a_{i+1} \ldots a_j$ is in $L_G(BC)$ because of a partition with a middle point between m and ℓ', then the pair (B, C) must already be in $P_{i,j}$, as illustrated in the figure.

  Under these assumptions, *complete*(ℓ, m, ℓ', m') constructs $T_{i,j}$ for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$, which is shown in Figure 9.4(b).

The algorithm is going to calculate products of square submatrices of T, putting the results to submatrices of P, and these operations shall be represented in the algorithm using the following notation. Given an $n \times n$ matrix X and a quadruple of numbers $\mathcal{A} = (\ell, m, \ell', m')$, as in the arguments for the procedure *complete*(), denote by $X_{\mathcal{A}}$ the $(m - \ell) \times (m' - \ell')$ submatrix of X formed by its rows from ℓ to $m - 1$ and its columns from ℓ' to $m' - 1$. A product of matrices of sets $X, Y \in (2^N)^{m \times m}$ is a matrix $Z \in (2^{N \times N})^{m \times m}$, defined by

$$Z_{i,j} = \bigcup_{k=1}^{m} X_{i,k} \times Y_{k,j},$$

as in the earlier Definition 9.1. Elementwise union of two matrices $Z, Z' \in (2^{N \times N})^{m \times m}$ is denoted by $Z \cup Z' = \widehat{Z}$, where $\widehat{Z}_{i,j} = Z_{i,j} \cup Z_{i,j}$.

The partitions of the matrix in *compute*() and *complete*() are illustrated in Figure 9.5. Note that $m - \ell$ is a power of two in each call to *compute*() and to *complete*(), and accordingly, if the input string is of length $2^k - 1$, then the algorithm multiplies submatrices of size 1×1, 2×2, 4×4, and so on up to $2^{k-2} \times 2^{k-2}$. Furthermore, both m and ℓ are always divisible by $m - \ell$, and hence all $2^{k-i} \times 2^{k-i}$ submatrices being multiplied are aligned over a 2^{k-i}-step grid.

Algorithm 9.5 Parsing through matrix multiplication (Valiant's algorithm)

Let $G = (\Sigma, N, R, S)$ be a Boolean grammar in the binary normal form. Let $w = a_1 \dots a_n$, where $n \geqslant 1$ and $a_i \in \Sigma$, be an input string; assume n is a power of two.

Main procedure:

1: $compute(0, n+1)$
2: Accept if and only if $S \in T_{0,n}$

Procedure $compute(\ell, m)$:

3: **if** $m - \ell \geqslant 4$ **then** /* see Figure 9.5(a) */
4: $compute(\ell, \frac{\ell+m}{2})$
5: $compute(\frac{\ell+m}{2}, m)$
6: $complete(\ell, \frac{\ell+m}{2}, \frac{\ell+m}{2}, m)$

Procedure $complete(\ell, m, \ell', m')$, which requires $m - \ell = m' - \ell'$:

7: **if** $m - \ell > 1$ **then** /* see Figure 9.5(b) */
8: denote $\mathcal{B} = (\ell, \frac{\ell+m}{2}, \frac{\ell'+m'}{2}, m')$, $\mathcal{B}' = (\frac{\ell+m}{2}, m, \ell', \frac{\ell'+m'}{2})$, $\mathcal{C} = (\frac{\ell+m}{2}, m, \ell', \frac{\ell'+m'}{2})$, $\mathcal{D} = (\ell, \frac{\ell+m}{2}, \ell', \frac{\ell'+m'}{2})$, $\mathcal{D}' = (\frac{\ell+m}{2}, m, \frac{\ell'+m'}{2}, m')$, $\mathcal{E} = (\ell, \frac{\ell+m}{2}, \frac{\ell'+m'}{2}, m')$
9: $complete(\mathcal{C})$
10: $P_{\mathcal{D}} = P_{\mathcal{D}} \cup (T_{\mathcal{B}} \times T_{\mathcal{C}})$
11: $complete(\mathcal{D})$
12: $P_{\mathcal{D}'} = P_{\mathcal{D}'} \cup (T_{\mathcal{C}} \times T_{\mathcal{B}'})$
13: $complete(\mathcal{D}')$
14: $P_{\mathcal{E}} = P_{\mathcal{E}} \cup (T_{\mathcal{B}} \times T_{\mathcal{D}'})$
15: $P_{\mathcal{E}} = P_{\mathcal{E}} \cup (T_{\mathcal{D}} \times T_{\mathcal{B}'})$
16: $complete(\mathcal{E})$
17: **else if** $m - \ell = 1$ and $m < \ell'$ **then**
18: $T_{\ell,\ell'} = f(P_{\ell,\ell'})$
19: **else if** $m - \ell = 1$ and $m = \ell'$ **then**
20: $T_{\ell,\ell+1} = \{\, A \mid A \to a_{\ell+1} \in R \}$

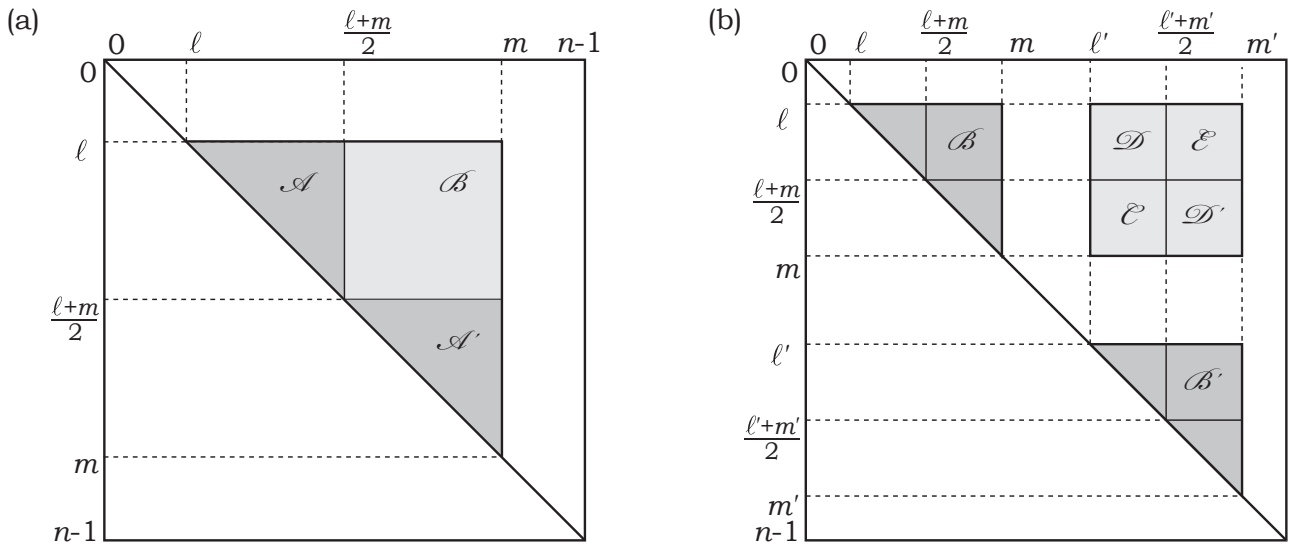

Figure 9.5: (a) Matrix partition in $compute(\ell, m)$; (b) matrix partition in $complete(\ell, m, \ell', m')$.

Lemma 9.2. *Let $\ell < m \leqslant \ell' < m'$, where $m - \ell = m' - \ell'$ is a power of two, and assume that $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ for all i and j with $\ell \leqslant i < j < m$, as well as for all i, j with $\ell' \leqslant i < j < m'$. Furthermore, assume that, for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$,*

$$P_{i,j} = \{ (B,C) \mid \exists k \, (m \leqslant k < \ell') : \; a_{i+1} \dots a_k \in L(B), \; a_{k+1} \dots a_j \in L(C) \}.$$

Then complete(ℓ, m, ℓ', m') returns with $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$.

Proof. Induction on $m - \ell$.

Basis I: $m - \ell = 1$ and $m = \ell'$. Then the algorithm has to handle a single element $T_{\ell,\ell+1}$, and this element is correctly computed in line 20.

Basis II: $m - \ell = 1$ and $m < \ell'$. Again, there is only one element to compute, and the current value of $P_{\ell,\ell'}$ is $\{ (B,C) \mid \exists k \, (\ell < k < \ell') : \; a_{\ell+1} \dots a_k \in L(B), \; a_{k+1} \dots a_{\ell'} \in L(C) \} = \{ (B,C) \mid a_{\ell+1} \dots a_{\ell'} \in L(B)L(C) \}$. Then line 18 of *complete*() computes $f(P_{\ell,\ell'}) = \{ A \mid a_{\ell+1} \dots a_{\ell'} \in L(A) \}$ and thus sets $T_{\ell,\ell'}$ correctly.

Induction step. Let $\ell < m \leqslant \ell' < m'$ with $m - \ell = m' - \ell' > 1$ and assume that *complete*$(\ell_1, m_1, \ell_2, m_2)$ works correctly for $m_1 - \ell_1 = m_2 - \ell_2 < m - \ell$. Consider the computation of *complete*(ℓ, m, ℓ', m'), which begins with the submatrices $\mathcal{B}$ and $\mathcal{B}'$ of T already computed, as in Figure 9.4(b).

The first call to *complete*$(\frac{\ell+m}{2}, m, \ell', \frac{\ell'+m'}{2})$ in line 9 requires that the current value of each $P_{i,j}$ with $\frac{\ell+m}{2} \leqslant i < m$ and $\ell' \leqslant j < \frac{\ell'+m'}{2}$ (that is, in the $\mathcal{C}$-submatrix) is $\{ (B,C) \mid \exists k \, (m \leqslant k < \ell') : \; a_{i+1} \dots a_k \in L(B), \; a_{k+1} \dots a_j \in L(C) \}$, which is true by the assumption. Then, by the induction hypothesis, this call to *complete*() computes all values of T in the submatrix $\mathcal{C}$.

The call to *product*() in line 10 adds to each $P_{i,j}$ with $\ell \leqslant i < \frac{\ell+m}{2}$ and $\ell' \leqslant j < \frac{\ell'+m'}{2}$ (in the submatrix $\mathcal{D}$), all pairs (B,C) with $a_{i+1} \dots a_k \in L(B)$, $a_{k+1} \dots a_j \in L(C)$ and $\frac{\ell+m}{2} \leqslant k < m$. Taking into account that all such pairs with $m \leqslant k < \ell'$ were already there by the assumption, $P_{i,j}$ now contains these pairs for all $\frac{\ell+m}{2} \leqslant k < \ell'$. Then the induction hypothesis is applicable to the subsequent call to *complete*$(\ell, \frac{\ell+m}{2}, \ell', \frac{\ell'+m'}{2})$ in line 11, and so it computes all values of T in the $\mathcal{D}$-submatrix.

Symmetrically, the next lines 12–13 compute all $T_{i,j}$ with $\frac{\ell+m}{2} \leqslant i < m$ and $\frac{\ell'+m'}{2} \leqslant j < m'$, that is, the submatrix $\mathcal{D}'$.

At this moment, the elements $P_{i,j}$ with $\ell \leqslant i < \frac{\ell+m}{2}$ and $\frac{\ell'+m'}{2} \leqslant j < m'$ (that is. the $\mathcal{E}$-submatrix of P) contain all pairs (B,C) with $a_{i+1} \dots a_k \in L(B)$, $a_{k+1} \dots a_j \in L(C)$ and $m \leqslant k < \ell'$. The subsequent line 14 adds to each $P_{i,j}$ all pairs with $\frac{\ell+m}{2} \leqslant k < m$, and line 15 adds pairs with $\ell' \leqslant k < \frac{\ell'+m'}{2}$. With these additions, each $P_{i,j}$ contains all pairs (B,C) satisfying $a_{i+1} \dots a_k \in L(B)$ and $a_{k+1} \dots a_j \in L(C)$ for some $\frac{\ell+m}{2} \leqslant k < \frac{\ell'+m'}{2}$. All conditions to call *complete*$(\ell, \frac{\ell+m}{2}, \frac{\ell'+m'}{2}, m')$ are now fulfilled, and, by the induction hypothesis, line 16 constructs all elements $T_{i,j}$ with $\ell \leqslant i < \frac{\ell+m}{2}$. This is the last remaining submatrix $\mathcal{E}$, and now $T_{i,j}$ is computed for all $\ell \leqslant i < m$ and $\ell' \leqslant j < m'$, which completes the proof. □

Lemma 9.3. *The procedure compute(ℓ, m), executed on any such ℓ and m that $m - \ell$ is a power of two, returns with $T_{i,j} = \{ A \mid a_{i+1} \dots a_j \in L(A) \}$ for all $\ell \leqslant i < j < m$.*

Proof. Induction on $m - \ell$.

The base case is $m - \ell = 2$, in which the procedure *compute*() makes no recursive calls to itself, and line 6 calls *complete*$(\ell, \ell+1, \ell+1, \ell+2)$. The pre-conditions for calling *complete* are met, because there exist no elements $T_{i,j}$ with $\ell \leqslant i < j < \ell + 1$ or with $\ell + 1 \leqslant i < j < \ell + 2$, and no numbers k with $\ell + 1 \leqslant k < \ell + 1$. Hence, by Lemma 9.2, this call results in $T_{\ell,\ell+1}$ correctly computed (which will actually be done in line 20). Since this is the only element $T_{i,j}$ with $\ell \leqslant i < j < m = \ell + 2$, the lemma holds for this case.

If $m - \ell \geqslant 4$, then *compute*() will first call itself to compute the values of $T_{i,j}$ for all $\ell \leqslant i < j < \frac{\ell+m}{2}$ and for all $\frac{\ell+m}{2} \leqslant i < j < m$. Then, when *complete*$(\ell, \frac{\ell+m}{2}, \frac{\ell+m}{2}, m)$ is called, the condition on $T_{i,j}$ in Lemma 9.2 is satisfied. The second condition of the lemma is that each $P_{i,j}$ contains all pairs (B, C) corresponding to *some* k *with* $\frac{\ell+m}{2} \leqslant k < \frac{\ell+m}{2}$, and since there are no such values of k, this condition is satisfied as well. Therefore, Lemma 9.2 is applicable to this call, and it asserts that $T_{i,j}$ will be correctly set for all $\ell \leqslant i < \frac{\ell+m}{2}$ and $\frac{\ell+m}{2} \leqslant j < m$, which are all the remaining values of i and j. □

In order to estimate the running time of the algorithm, it would be sufficient to use the well-known general solutions of recurrence relations. However, in order to understand what the algorithm actually does, it is more useful to determine, exactly how many times the procedures *compute*() and *complete*() are called for subproblems of each size, and how many products of matrices of each size get computed.

Lemma 9.4. *Let the input string be of length* $2^k - 1$*. Then, in the computation of the main procedure,*

i. for each $i \in \{0, \ldots, k-1\}$*,* *compute*(ℓ, m) *with* $m - \ell = 2^{k-i}$ *is called exactly* 2^i *times,*

ii. for each $i \in \{1, \ldots, k-1\}$*,* *complete*(ℓ, m, ℓ', m') *with* $m - \ell = 2^{k-i}$ *is called exactly* $2^{2i-1} - 2^{i-1}$ *times,*

iii. for each $i \in \{2, \ldots, k\}$*,* *product*() *is called for submatrices of size* $2^{k-i} \times 2^{k-i}$ *exactly* $2^{2i-1} - 2^i$ *times.*

Proof. The first claim is proved by an obvious induction on i, and the proof can be safely omitted.

Turning to the second claim, the induction here is also rather simple, and proceeds as follows. For succinctness, the phrase "*complete*() of size s" shall refer to any calls to the procedure *complete*(ℓ, m, ℓ', m') with $m - \ell = s$. Then, as the base case, $i = 1$, a call to *complete*() of size 2^{k-1} is made only once, from the top-level *compute*$(0, 2^k)$, and accordingly $2^{2i-1} - 2^{i-1} = 2^1 - 2^0 = 1$. For the induction step, assume that *complete*() of size 2^{k-i} is called $2^{2i-1} - 2^{i-1}$ times, and consider the calls to *complete*() of size 2^{k-i-1}. First, the procedure *complete*() of size 2^{k-i-1} is called 4 times from each instance of *complete*() of size 2^{k-i}, and secondly, it is called once from each instance of *compute*(ℓ, m) with $m - \ell = 2^{k-i}$. In total, this sums up to

$$4 \cdot (2^{2i-1} - 2^{i-1}) + 1 \cdot 2^i = 2^{2(i+1)-1} - 2^{(i+1)-1}$$

calls, as claimed.

Finally, each call to *complete*() of size $2^{k-(i-1)}$ makes four calls to *product*() for matrices of size $2^{k-i} \times 2^{k-i}$, and these are all the matrix products computed by the algorithm. Hence, *product*() is called for $2^{k-i} \times 2^{k-i}$ matrices $4 \cdot (2^{2(i-1)-1} - 2^{i-2}) = 2^{2i-1} - 2^i$ times. □

According to these calculations, the time spent on matrix multiplication dominates the running time, which leads to the following estimation of the algorithm's complexity.

Theorem 9.1. *Given a Boolean grammar* G *in binary normal form and a string of length* n*, Algorithm 9.5 constructs the parsing table* T *for this grammar and this string in time* $O\big(|G| \cdot \mathrm{BMM}(n) \log n\big)$*, where* $\mathrm{BMM}(n)$ *is the time needed to multiply two* $n \times n$ *Boolean matrices. Assuming* $\mathrm{BMM}(n) = n^{2+\Omega(1)}$*, the complexity is* $\Theta\big(|G| \cdot \mathrm{BMM}(n)\big)$*.*

Proof. Assume that $n = 2^k - 1$. If $n + 1$ is not a power of two, then one can construct the parsing table for a padded string, and then use only its relevant entries. The correctness of the algorithm is asserted by Lemma 9.3, according to which, the call to *compute*$(0, 2^k)$ in line 1 of the main procedure calculates all $T_{i,j}$ with $0 \leqslant i < j < 2^k$.

In order to estimate the running time, by Lemma 9.4, it is sufficient to sum up the time used for matrix multiplication. Let $\mathrm{BMM}(n) = n^{\omega} \cdot f(n)$, where $\omega \geqslant 2$ and $f(n) = n^{o(1)}$. According to Lemma 9.4(iii), for every $i \in \{2, \ldots, k\}$, the procedure *product*() is called $2^{2i-1} - 2^i < 2^{2i}$ times for matrix size 2^{k-i}, and calculates $C = O(|G|)$ products of Boolean submatrices of size $2^{k-i} \times 2^{k-i}$. The total number of operations is thus estimated as follows.

$$C \sum_{i=2}^{k} 2^{2i} \mathrm{BMM}(2^{k-i}) = C \sum_{i=2}^{k} 2^{2i} 2^{\omega k - \omega i} f(2^{k-i}) = C \cdot 2^{\omega k} \sum_{i=2}^{k} 2^{(2-\omega)i} f(2^{k-i}) \leqslant$$

$$\leqslant C \cdot 2^{\omega k} f(2^k) \sum_{i=2}^{k} 2^{(2-\omega)i} = C \cdot \mathrm{BMM}(2^k) \sum_{i=2}^{k} 2^{(2-\omega)i}$$

It remains to estimate the sum. Under the assumption that $\omega > 2$, it is bounded by a constant as a convergent geometric series.

$$\sum_{i=2}^{k} 2^{(2-\omega)i} \leqslant \sum_{i=2}^{\infty} 2^{(2-\omega)i} = \frac{2^{2(2-\omega)}}{1 - 2^{2-\omega}},$$

Therefore, the total number of operations is $O\big(|G| \cdot \mathrm{BMM}(2^k)\big) = O\big(|G| \cdot \mathrm{BMM}(n)\big)$.

On the other hand, if $\omega = 2$, then

$$\sum_{i=2}^{k} 2^{(2-\omega)i} = \sum_{i=2}^{k} 2^0 = k - 1,$$

leading to the upper bound $O\big(|G| \cdot \mathrm{BMM}(2^k) \cdot k\big) = O\big(|G| \cdot \mathrm{BMM}(n) \log n\big)$. □

9.2.5 Constructing a parse tree

For an ordinary grammar, a tree is constructed in square time from the table $T_{i,j}$ by Algorithm 9.3 in Section 9.1.2. For a Boolean grammar, Algorithm 9.4 can create a parse tree from $T_{i,j}$ as well, but this requires cubic time, and thus negates the complexity improvements in Algorithm 9.5.

In order to construct a parse tree for a Boolean grammar in subcubic time, Algorithm 9.5 is augmented to construct the tree along with the parsing table, as follows. The sets $P_{i,j} \subseteq N \times N$ are replaced with functions $P'_{i,j} \colon N \times N \to \{0, 1, \ldots, n\}$, where $P'_{i,j}(B, C) = k > 0$ indicates that $a_{i+1} \ldots a_k \in L_G(B)$ and $a_{k+1} \ldots a_j \in L_G(C)$, that is, the substring may be split as $B \cdot C$ with the splitting position k. Whereas the original sets $P_{i,j}$ were obtained by Boolean matrix multiplication, calculating the splitting positions for $P'_{i,j}$ is the problem of finding *witnesses for Boolean matrix multiplication*, solved by the algorithm by Alon and Naor [2], which uses $O(M(n) \log^5 n)$ operations in a finite ring, where $M(n)$ is the number of ring operations for matrix multiplication. Using this procedure instead of the standard Boolean matrix multiplication in Algorithm 9.5, yields an algorithm for constructing a parse tree in time $O(|G| \cdot n^{\omega})$.

Lower bounds on matrix multiplication based on the complexity of parsing. Proposed as an open problem by Harrison [5, Problem 12.7.7]. Implemented by Lee.

9.3 Square-time parsing for unambiguous grammars

The basic cubic-time parsing algorithm has a variant, discovered by Kasami and Torii [7], that works in square time on any unambiguous grammar. This algorithm also applies for conjunctive and Boolean grammars, and is presented in the version for Boolean grammars.

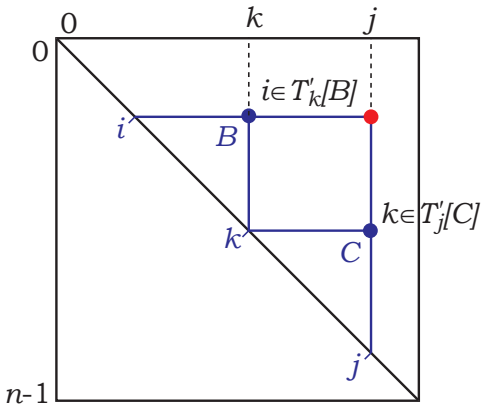

Figure 9.6: How the Kasami–Torii algorithm (Algorithm 9.6) finds all substrings ending in the position j representable as a concatenation BC. ***DRAFT***

This performance increase is achieved by using a different data structure to represent the same table $T \in (2^N)^{n \times n}$, with

$$T_{i,j} = \{\, A \in N \mid a_{i+1} \dots a_j \in L_G(A) \}.$$

The new data structure is a two-dimensional table T', indexed by positions in the input and nonterminals. Each entry of this table holds a set of positions in the input string. The element corresponding to a position k ($1 \leqslant k \leqslant n$) and a nonterminal $A \in N$ is denoted by $T'_k[A]$, and its intended value is

$$T'_j[A] = \{\, i \mid a_{i+1} \dots a_j \in L_G(A) \},$$

that is, $T'_j[A]$ should contain the initial positions of all substrings generated by A, which end at the k-th position. Then, accordingly, $i \in T'_j[A]$ if and only if $A \in T_{i,j}$. The entire string $a_1 \dots a_n$ is in $L(G)$ if and only if the position 0 is in $T'_n[S]$. Each set $T'_j[A]$ is being stored in memory as a list in an ascending order.

The algorithm processes the input string from left to right: after reading every next symbol a_j, it calculates the sets $T'_j[A]$ for all $A \in N$. For each j, this is done along with determining all concatenations $a_{i+1} \dots a_k \cdot a_{k+1} \dots a_j$, where $a_{i+1} \dots a_k \in L_G(B)$ and $a_{k+1} \dots a_j \in L_G(C)$ for some positions i, k and for some conjunct BC in any rule of the grammar. These concatenations are then stored in the variables $P_{i,j} \subseteq N \times N$, which are eventually used to insert the element i in the lists $T'_j[A]$.

The key idea of the algorithm is the particular way, in which these sets $P_{i,j}$ are filled. The algorithm performs lookups of the following form: in search for a conjunct BC, first, it traverses the list $T'_j[C]$ and reads each position k from this list; secondly, for every such position k, it traverses the list $T'_k[B]$ and considers each position i in there; and thirdly, for every such i, it adds the pair (B, C) into $P_{i,j}$. In this way, if the lists for a particular string are sparsely populated, then the algorithm will have to make only as few steps as the number of actual concatenations, rather than look for concatenations in all possible places. And if the grammar is unambiguous, one can prove that there are only $O(n^2)$ concatenations in total, and that the statement in the innermost loop of the algorithm shall be executed at most $O(n^2)$ times.

The purpose of each j-th iteration of the outer loop (line 2) is to determine, for all $A \in N$, the membership in $L_G(A)$ of substrings of the input string ending at its j-th position. This information is stored in $T'_j[A]$. The first nested loop in lines 3–7 handles substrings of length 1, that is, it records in $T'_j[A]$ whether a_j is in $L_G(A)$. Substrings of greater length ending at the j-th position are processed in the second nested loop by k (line 9).

Algorithm 9.6 The Kasami–Torii parsing algorithm for Boolean grammars

Let $G = (\Sigma, N, R, S)$ be a Boolean grammar in the Chomsky normal form, and let $\widehat{P} = \{\, BC \mid \text{there is a conjunct } BC \text{ in some rule}\}$. For every $P \subseteq \widehat{P}$, define

$$f(P) = \{A \mid \exists A \to B_1C_1 \& \ldots \& B_\ell C_\ell \& \neg D_1E_1 \& \ldots \& \neg D_mE_m \& \neg\varepsilon \in R, \\ \text{such that } (B_1, C_1), \ldots, (B_\ell, C_\ell) \in P \text{ and } (D_1, E_1), \ldots, (D_m, E_m) \notin P\}$$

Let $w = a_1 \ldots a_n$, where $n \geqslant 1$ and $a_i \in \Sigma$, be an input string. For each $j \in \{1, \ldots, n\}$, let $T'_j[A]$ be a variable ranging over subsets of $\{0, \ldots, j-1\}$; for each $i \in \{0, \ldots, n-1\}$, let a variable P_i range over subsets of $\widehat{P}$.

1: let $T'_j[A] = \varnothing$ for all $j = 1, \ldots, n$ and $A \in N$
2: **for** $j = 1$ to n **do**
3: **for all** $A \in N$ **do**
4: **if** $A \to a_j \in R$ **then**
5: $T'_j[A] = \{j-1\}$
6: **else**
7: $T'_j[A] = \varnothing$
8: let $P_i = \varnothing$ for all i $(0 \leqslant i < j-1)$
9: **for** $k = j-1$ to 1 **do**
10: **for all** $(B, C) \in \widehat{P}$ **do**
11: **if** $k \in T'_j[C]$ **then**
12: **for all** $i \in T'_k[B]$ **do**
13: $P_i = P_i \cup \{(B, C)\}$
14: **for all** $A \in f(P_k)$ **do**
15: $T'_j[A] = T'_j[A] \cup \{k-1\}$
16: accept if and only if $0 \in T'_n[S]$

Each $T'_j[A]$ is stored as a list, with elements sorted in an ascending order. The operations on this data structure are implemented as follows:

Lines 1, 5 and 7: A one-element list or an empty list is created.

Line 11: The first element in the list is checked. If it is not k, it is assumed that k is not in the list.

Line 12: The list is traversed.

Line 15: The new element is inserted in the beginning of the list.

Line 16: As in line 11, only the first element is checked.

This loop constructs an auxiliary data structure P: for each $i \in \{0, \ldots, j-2\}$, P_i is meant to contain all conjuncts BC, for which the substring beginning at the position $i+1$ and ending at the position j is in $L_G(BC)$. Every k-th iteration of this loop, denoted (j,k), considers substrings of various length starting at any position $i+1 \in \{1,2,\ldots,k\}$ and ending at the position j. The goal is to determine all such substrings, which belong to $L_G(BC)$ for some $BC \in \widehat{P}$, and in which the middle point in their partition into $u \in L_G(B)$ and $v \in L_G(C)$ is exactly $k+1$, that is, the first part u ends at the position k and the second part v starts at the position $k+1$. These substrings uv are identified by first considering the appropriate unsigned conjunct, then checking the membership of the second substring in $L_G(C)$ (line 11), and finally by enumerating all appropriate first parts using the data in $T'_k[B]$.

This is used to fill the elements P_{k-1}, P_{k-2}, …, P_0, with appropriate conjuncts. An element P_{k-1} gets completely filled in course of iteration (j,k), and at this point the set of nonterminals generating the substring starting from the position k and ending at the position j can be obtained as $f(P_{k-1})$, which is done in lines 14–15.

To verify the algorithm's correctness, there are three properties to be established: first, that the given implementation of $T'_j[A]$ by lists faithfully represents the high-level set operations. Second, it has to be shown that the algorithm is a correct recognizer, that is, it accepts w if and only if $w \in L(G)$. Third, it remains to demonstrate that the algorithm works in time $O(n^2)$ on every unambiguous grammar.

Let us see that, indeed, the lists $T'_j[A]$ stay sorted in course of the computation, and the tests in lines 11, 16 and the insertion in line 15 can be implemented as described.

Lemma 9.5. *Each list $T'_j[A]$ always remains sorted. Each time the algorithm checks the condition in line 11, every set $T'_j[A]$ does not contain elements less than k. Each time the algorithm is about to execute line 15, the set $T'_j[A]$ does not contain elements less than k.*

Proof. An element $k-1$ ($1 \leqslant k < j$) can be added to $T'_j[A]$ only at the iteration (j,k). Hence, in the beginning of each iteration (j,k) the current value of $T'_j[A]$ is a subset of $\{k, k+1, \ldots, j-1\}$. As a result, if $T'_j[A]$ is sorted before the assignment in line 15, it remains sorted after the assignment. All three claims follow. □

Let us continue with the correctness statement of the algorithm, which claims what values should the variables have at certain points of the computation.

To unify the notation, let us refer to the point before the iteration $j = 1$, that is, to the very beginning of the execution, as "after the iteration 0". Similarly, the point before the iteration $(j, k = j-1)$, that is, inside iteration j right before the loop by k is entered, will be referred to as "after the iteration (j,j)". Then the statement of correctness can be succinctly formulated as follows:

Lemma 9.6 (Correctness of Algorithm 9.6)**.** *For every Boolean grammar in the binary normal form, in the computation of the above algorithm on a string $w \in \Sigma^+$,*

i. after iteration j, for each $A \in N$ and for each $t \in \{1, \ldots, j\}$, the set $T'_t[A]$ equals

$$\{\, i \mid 0 \leqslant i < t \text{ and } a_{i+1} \ldots a_t \in L_G(A) \};\tag{9.1}$$

ii. after iteration (j,k), every $T'_j[A]$ with $A \in N$ equals

$$\{\, i \mid k-1 \leqslant i < j \text{ and } a_{i+1} \ldots a_j \in L_G(A) \};\tag{9.2}$$

iii. after iteration (j,k), every P_i with $0 \leqslant i < j$ equals

$$\{\, (B,C) \in \widehat{P} \mid \exists \ell\, (k \leqslant \ell < j) : a_{i+1} \ldots a_\ell \in L(B) \text{ and } a_{\ell+1} \ldots a_j \in L(C) \}.\tag{9.3}$$

Lemma 9.7 (Algorithm 9.6 on unambiguous grammars)**.** *Assume that all concatenations in G are unambiguous, and let w be an n-symbol input string. Then the assignment statement $P_i = P_i \cup \{BC\}$ in the inner loop is executed at most $|\widehat{P}| \cdot n^2$ times.*

Proof. Let us prove that for every j, for every concatenation $(B, C) \in \widehat{P}$ and for every i there exists at most one number k, such that iteration (j, k, BC, i) of four nested loops is executed.

Suppose there exist two such numbers, k and k'. For the inner loop in lines 12–13 to be executed, both k and k' have to be in $T'_j[C]$. Then, by Lemma 9.6(ii),

$$a_{k+1} \dots a_j \in L(C) \quad \text{and} \tag{9.4a}$$
$$a_{k'+1} \dots a_j \in L(C). \tag{9.4b}$$

Furthermore, for the corresponding iterations of the inner loop to be executed, i must be both in $T'_k[B]$ and in $T'_{k'}[B]$. By Lemma 9.6(i), this means the following:

$$a_{i+1} \dots a_k \in L(B), \tag{9.5a}$$
$$a_{i+1} \dots a_{k'} \in L(B). \tag{9.5b}$$

Combining (9.5a) with (9.4a) and (9.5b) with (9.4b), one obtains two partitions of $a_{i+1} \dots a_j$ as $u \cdot v$, where $u \in L(B)$ and $v \in L(C)$. Since the concatenation BC is unambiguous by assumption, there is at most one such partition. Therefore, the constructed partitions are the same, that is, $k = k'$. □

Theorem 9.2. *For every Boolean grammar $G = (\Sigma, N, R, S)$ in binary normal form and for every input string $w \in \Sigma^*$, Algorithm 9.6 accepts if and only if $w \in L(G)$. Implemented on a random access machine, it terminates after $O(n^3)$ elementary steps, where $n = |w|$, or after $O(n^2)$ elementary steps, if the grammar is unambiguous.*

Proof. The correctness of the algorithm is given by Lemma 9.6(i): for $j = n$ and $A = S$, the final value of $T'_j[A]$ is

$$T'_n[S] = \{\, i \mid 0 \leqslant i < n \text{ and } a_{i+1} \dots a_n \in L(G) \},$$

and therefore $0 \in T'_n[S]$ if and only if $a_1 \dots a_n \in L(G)$.

Next, note that each statement of the algorithm is executed in a constant number of machine instructions. Indeed, the only data of non-constant size are the lists $T'_j[A]$, and the implementation notes in the end of Algorithm 9.6 cover each reference to these variables in the algorithm. Then the cubic time upper bound for the execution time is evident.

These are lines 14–15 that are responsible for cubic time, and each of the rest of the statements is visited $O(n^2)$ times in any computation. Since, by Lemma 9.7, on any unambiguous grammar lines 14–15 are visited $O(n^2)$ times as well, this implies the algorithm's square-time performance on any unambiguous grammar. □

9.4 Parsing for pair-wrapping grammars

9.4.1 The direct $O(n^6)$-time algorithm

By Vijay-Shanker and Joshi.

Assume a pair-wrapping grammar $G = (\Sigma, N, R, S)$ in a normal form, where each rule is of the following form.

$$\begin{aligned}
&A \to BC && (B, C \in N)\\
&A \to B : \varepsilon && (B \in N)\\
&A \to \varepsilon : C && (C \in N)\\
&A \to a : \varepsilon && (a \in \Sigma)\\
&A \to \varepsilon : a && (a \in \Sigma)
\end{aligned}$$

Given an input string $w = a_1 \dots a_n$, the algorithm constructs a four-dimensional array T, with

$$T_{i,k,\ell,j} = \{\, A \in N \mid a_{i+1} \dots a_k : a_{\ell+1} \dots a_j \in L_G(A) \}$$

for all $0 \leqslant i < k \leqslant \ell < j \leqslant n$.

Base case (strings with a gap, of combined length 1).

$$\begin{aligned}
T_{i,i+1,j,j} &= \{\, A \in N \mid A \to a_{i+1} : \varepsilon \in R \},\\
T_{i,i,j-1,j} &= \{\, A \in N \mid A \to \varepsilon : a_j \in R \},
\end{aligned}$$

for all i, j with $0 \leqslant i < j \leqslant n$.

Transition to longer strings with a gap.

$$T_{i,k,\ell,j} = f\Big(\bigcup_{s=i}^{k} \bigcup_{t=\ell}^{j} T_{i,s,t,j} \times T_{s,k,\ell,t} \Big) \underbrace{\cup f'\Big(\bigcup_{s=i}^{k} T_{i,s,s,k} \Big)}_{\text{if } \ell = j} \underbrace{\cup f''\Big(\bigcup_{t=\ell}^{j} T_{\ell,t,t,j} \Big)}_{\text{if } i = k},$$

where the function $f \colon 2^{N \times N} \to 2^N$ is defined by $A \in f(P)$ if and only if there is a rule $A \to BC$ with $(B, C) \in P$.

f', f'' defined similarly.

Total: $\Theta(n^4)$ elements, $\Theta(n^2)$ operations for each. Hence, running time $\Theta(n^6)$.

9.4.2 Reduction to matrix multiplication

Parsing in time $O(n^{2\omega})$ by Rajasekaran and Yooseph [12]. (that is, the time of multiplying $n^2 \times n^2$ matrices)

Bibliography

[1] L. Adleman, K. S. Booth, F. P. Preparata, W. L. Ruzzo, "Improved time and space bounds for Boolean matrix multiplication", *Acta Informatica* 11:1 (1978), 61–70.

[2] N. Alon, M. Naor, "Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions", *Algorithmica*, 16:4–5 (1996), 434–449.

[3] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradzhev, "On economical construction of the transitive closure of an oriented graph", *Soviet Mathematics Doklady*, 11 (1970), 1209–1210.

[4] S. L. Graham, M. A. Harrison, W. L. Ruzzo, "An improved context-free recognizer", *ACM Transactions of Programming Languages and Systems*, 2:3 (1980), 415–462.

[5] M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

[6] T. Kasami, "An efficient recognition and syntax-analysis algorithm for context-free languages", Report AF CRL-65-758, Air Force Cambridge Research Laboratory, USA, 1965.

[7] T. Kasami, K. Torii, "A syntax-analysis procedure for unambiguous context-free grammars", *Journal of the ACM*, 16:3 (1969), 423–431.

[8] L. Lee, "Fast context-free grammar parsing requires fast Boolean matrix multiplication", *Journal of the ACM*, 49:1 (2002), 1–15.

[9] A. Okhotin, "Boolean grammars", *Information and Computation*, 194:1 (2004), 19–48.

[10] A. Okhotin, "Unambiguous Boolean grammars", *Information and Computation*, 206 (2008), 1234–1247.

[11] A. Okhotin, "Parsing by matrix multiplication generalized to Boolean grammars", *Theoretical Computer Science*, 516 (2014), 101–120.

[12] S. Rajasekaran, S. Yooseph, "TAL recognition in $O(M(n^2))$ time", *Journal of Computer and System Sciences*, 56:1 (1998), 83–89.

[13] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13 (1969), 354–356.

[14] L. G. Valiant, "General context-free recognition in less than cubic time", *Journal of Computer and System Sciences*, 10:2 (1975), 308–314.

[15] D. H. Younger, "Recognition and parsing of context-free languages in time n^3", *Information and Control*, 10 (1967), 189–208.

Index