Hauptseminar
Algorithmische Probleme kontextfreier Grammatiken

LMU München, CIS, SS 2006

Hans Leiß

3. August 2006

Zusammenfassung

Das Seminar behandelte hauptsächlich die Äquivalenz des Parseproblems für kontextfreie Grammatiken und der Boole'schen Matrixmultiplikation nach [7, 5, 4]. Vor dieser Äquivalenz wird hier nur die Reduktion des Parseproblems auf die Boole'sche Matrixmultiplikation beschrieben.

Außerdem werden Ringe und Matrixringe und damit Algorithmen über Erreichbarkeit und Abstände in gewichteten Graphen und das Erkennungsverfahren von Rytter implementiert (und korrigiert).

Das Seminar behandelte daneben das Problem der regulären oberen Schranke einer kontextfreien Sprache nach [2, 1], worauf aber hier nicht eingegangen wird.

Inhaltsverzeichnis

1 Reduktion von Problemen

Definition 1.1 Ein *Entscheidungsproblem* $P = \{\, a \in A \mid \varphi(a) \,\}$ besteht darin, aus einer gegebenen Menge A von möglichen Eingaben, den *Instanzen*, diejenigen zu erkennen, die eine gegebene Eigenschaft $\varphi(x)$ erfüllen.

Das Problem P ist *lösbar*, wenn es eine berechenbare Funktion $f : A \to \mathbb{B}$ mit

$$\varphi(a) \iff f(a) = 1$$

für alle $a \in A$ gibt. Eine solche Funktion f ist eine *Lösung* des Problems P. Gibt es kein solches f, so ist P *(rekursiv) unlösbar*.

Das Problem $P = \{\, a \in A \mid \varphi(a) \,\}$ *ist (Turing-) reduzierbar auf* das Problem $Q = \{\, b \in B \mid \psi(b) \,\}$, kurz $P \leq Q$, wenn es eine berechenbare Funktion $h : A \to B$ gibt, so daß für alle $a \in A$:

$$\varphi(a) \iff \psi(h(a)).$$

Ein solches h würde eine Lösung g von Q in eine Lösung $f := g \circ h$ von P umrechnen, da ja $1 = f(a) = g \circ h(a) = g(h(a)) \iff \psi(h(a)) \iff \varphi(a)$ wäre.

Die Probleme P und Q sind *gleich schwierig*, kurz $P \equiv Q$, wenn $P \leq Q$ und $Q \leq P$, wenn also jedes auf das andere reduzierbar ist. (Beachte, daß dazu nicht erforderlich ist, daß P und Q lösbar sind.) Der *Schwierigkeitsgrad* von P ist die Äquivalenzklasse $[P]_{\equiv}$.

Läßt man bei der Reduktion von P auf Q nicht beliebige berechenbare h zu, sondern nur solche einer gegebenen Komplexität, so erhält man feinere Reduktionsbegriffe und entsprechende Äquivalenzrelationen zwischen Problemen. Wenn möglich, sollte die Reduktionsfunktion h nicht aufwendiger sein als die Lösung g, die man für Q (eventuell) hat.

Oft interessiert man sich auch für *Berechnungsprobleme*, bei denen es darum geht zu zeigen, daß und wie eine Funktion $p : A \to B$ berchenbar ist, d.h. man sollte ein Programm angeben, das für jede Eingabe $a \in A$ den Wert $p(a)$ ausgibt. Dies führt man ein entsprechendes Entscheidungsproblem ein:

$$P_p := \{\, (a, b) \in A \times B) \mid p(a) = b \,\}.$$

Wir wollen Algorithmen betrachten, die Eigenschaften kontextfreier Sprachen an ihren Grammatiken ablesen oder kontextfreie Grammatiken transformieren. Man unterscheidet dabei, ob die Grammatik Teil der Eingabe ist oder nicht.

Beispiel 1.2 Sei Σ eine endliche Menge und *CFG* die Menge der kontextfreien Grammatiken $G = (N, \Sigma, S, P)$ über dem (Terminal-) Alphabet Σ. Das *universelle Erkennungsproblem* für kontextfreie Grammatiken (über Σ) ist

$$\{\, (G, w) \in \mathit{CFG} \times \Sigma \mid w \in L(G) \,\}.$$

Das (spezielle) *Erkennungsproblem* für die Grammatik $G \in CFG$ ist

$$\{\, w \in \Sigma^* \mid w \in L(G) \,\}.$$

Insbesondere wollen wir den Zusammenhang zwischen diesen Erkennungsproblemen und der Multiplikation von Matrizen über geeigneten Ringen betrachten und zeigen, daß das Erkennungsproblem für kontextfreie Grammatiken und das Problem der Matrixmultiplikation über dem boole'schen Ring $\mathbb{B}$ gleich schwierig sind. Diese Probleme hängen auch mit der Berechnung von Wegen in Graphen und Optimierungsproblemen zusammen, weshalb wir erst darauf eingehen.

2 Graphen und Boole'sche Matrizen

Definition 2.1 Ein *Graph* (G, E) ist eine Menge mit einer zweistelligen Relation $E \subseteq G \times G$. Ein *Weg (der Länge n)* in (G, E) ist ein Tupel $(p_1, \ldots, p_n)$, so daß $(p_{i-1}, p_i) \in E$ für $i \leq n$. Der *leere Weg* $()$ hat die Länge 0.

Man kann einen Weg $\pi = (p_1, p_2, \ldots, p_n)$ in G auch als die Folge

$$(p_1, p_2)(p_2, p_3) \cdots (p_{n-1}, p_n)$$

der *Kanten* $(p_i, p_{i+1}) \in E$ verstehen, oder als ein Wort $\pi \in E^*$.

2.1 Graphen als quadratische Boole'sche Matrizen

Einen Graphen (G, E) kann man durch seine *charakteristische Funktion* $\chi_E : G \times G \to \mathbb{B}$, mit

$$\chi_E(p, q) := \begin{cases} 1 & \text{falls } (p, q) \in E, \\ 0 & \text{sonst,} \end{cases}$$

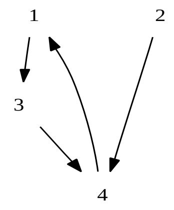
identifizieren. Ist G endlich und sind die Elemente durch eine injektive Numerierung $G = \{g_1, \ldots, g_n\}$ benannt, so stellt man χ_E durch eine $0,1$-*Matrix der Dimension* $n \times n$

$$E = \begin{pmatrix} e_{1,1} & \cdots & e_{1,n} \\ \vdots & \ddots & \vdots \\ e_{n,1} & \cdots & e_{n,n} \end{pmatrix}$$

dar, bei der in der i-ten Zeile (von oben) und j-ten Spalte (von links) der Wert $e_{i,j} := \chi_E(g_i, g_j) \in \mathbb{B}$ steht. Wir nehmen dann oft gleich $G = \{1, 2, \ldots, n\}$ an.

Man nennt dies die *Adjazenzmatrix* des Graphen (G, E), da sie die Paare (durch E) direkt benachbarter Elemente von G durch eine 1 darstellt.

Beispiel 2.2 Der Graph $(G, E) = (\{1, 2, 3, 4\}, \{(1, 3), (2, 4), (3, 4), (4, 1)\})$ oder



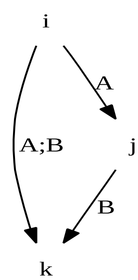
wird durch die Matrix
$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$
dargestellt.

Beachte, daß eine Kante $(i,j) \in E$ mit $i < j$ durch eine 1 oberhalb der Diagonale von $e_{1,1}$ nach $e_{n,n}$ dargestellt wird, eine Kante mit $i > j$ durch eine 1 unterhalb der Diagonale.

2.2 Relationenprodukt als Produkt Boole'scher Matrizen

Graphen $E \subseteq G \times G$ sind spezielle zweistellige Relationen; im allgemeinen Fall einer zweistelligen Relation $A \subseteq I \times J$ sind I und J nicht dieselbe Grundmenge. Man kann aber A ebensogut als $\{0,1\}$-Matrix
$$A = (a_{i,j})$$
darstellen, wobei man $i \in I$ als Zeilen- und $j \in J$ als Spaltenindex versteht.

Definition 2.3 Seien $A \subseteq I \times J$ und $B \subseteq J \times K$ zweistellige Relationen, bei denen der Wertebereich J von A gleich dem Argumentbereich von B ist[1]. Das *Relationenprodukt* von A und B ist die Relation $A\,;\,B \subseteq I \times K$ mit
$$(i,k) \in A\,;\,B \iff \exists j \in J\ (i,j) \in A \wedge (j,k) \in B. \tag{1}$$
Anschaulich:



[1] oder zumindest eine Teilmenge davon

Ist J endlich, so wird aus (1) die Summenformel für die Berechnung des Elements im Feld (i,k) der Boole'schen Matrix $C = A \,;\, B$:

$$C(i,k) := \sum_{j \in J} A(i,j) * B(j,k). \tag{2}$$

Hier sind die Summe $\sum$ und das Produkt $*$ der Boole'schen Werte $0, 1$ gemeint.

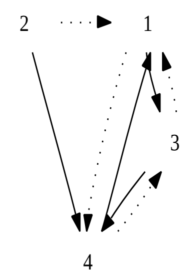
Das Relationenprodukt ist nicht kommutativ: Auch wenn $J \subseteq K \subseteq I$ und damit sowohl $A \,;\, B$ als auch $B \,;\, A$ definiert sind, ist i.a. $A \,;\, B \neq B \,;\, A$.

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} ; \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \neq \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} ; \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{3}$$

Beispiel 2.4 Ist $E \subseteq G \times G$ ein Graph, so ist $(i,j) \in (E \,;\, E) \subseteq G \times G$, wenn es einen Weg von i nach j im Graphen (G,E) der Länge 2 gibt. Für den Graphen E aus Beispiel 2.2 erhält man

$$E^2 = E \,;\, E = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} ; \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

In der Tat sieht man am Graphen, daß $E \,;\, E$ die E-Wege der Länge 2 darstellt:



Die Existenz eines Weges der Länge n von x nach y in einem Graphen (G,E) kann ausgedrückt werden durch

$$\begin{aligned} E^0(x,y) &\iff x = y, \\ E^{n+1}(x,y) &\iff \exists z (E^n(x,z) \wedge E(z,y)). \end{aligned}$$

In der Matrixdarstellung ist E^0 die *Einheitsmatrix*

$$E^0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

und die E^{n+1} erhält man aus E durch Matrixmultiplikationen: $E^{n+1} = E^n \,;\, E$. Will man die Existenz eines Weges von x nach y beliebiger Länge wissen, also

$$E^*(x,y) \quad \Longleftrightarrow \quad \exists n \in \mathbb{N}\; E^n(x,y),$$

so muß man die Summe

$$E^*(x,y) = E^0(x,y) + E^1(x,y) + E^2(x,y) + \dots$$

boole'scher Werte bilden; wenn ein Summand 1 ist, tragen die weiteren Summanden nichts mehr zum Ergebnis bei und man kann abbrechen. In einem endlichen Graphen G mit $|G| = n$ ist $E^*(x,y) = \sum_{i=0}^{n} E^i(x,y)$.

2.3 Der Algorithmus von Warshall

Sei (G,E) ein Graph mit $|G| = n$. Die Erreichbarkeit durch einen E-Weg (der Länge ≥ 0) ist nichts anderes als die (reflexive und) transitive Hülle E^+ (bzw. E^*) der Relation E.

Algorithmus von Warshall[8] zur Berechung der transitiven Hülle E^+ von E:

⟨*Warshall Pseudocode*⟩≡

```
% Initialisierung
for i = 1, ..., n
  for k = 1, ..., n
    C(i,k) := E(i,k);

for j = 1, ..., n
  for i = 1, ..., n
    for k = 1, ..., n
      C(i,k) := C(i,k) + C(i,j)*C(j,k)
```

Wir wollen zeigen, daß dieser Algorithmus korrekt ist, daß also nach der Berechnung die Matrix C die Relation E^+ darstellt:

$$\begin{aligned} &\text{Für } i,k = 1,\dots,n \text{ gilt :} \\ &C(i,k) = \begin{cases} 1, & \text{falls es einen } E\text{-Weg von } i \text{ nach } j \text{ gibt,} \\ 0, & \text{sonst.} \end{cases} \end{aligned} \tag{4}$$

Beweis: Unter Wegen werden hier die Wege der Länge ≥ 1 verstanden. Sei $E^{(j)}(i,k) \in \mathbb{B}$ genau dann 1, wenn es einen E-Weg von i nach k gibt, der außer an den Endpunkten kein Element $m \geq j$ durchläuft.

Man zeigt nun durch Induktion über j: ist die äußere Schleife von 1 bis j durchlaufen, so ist

$$C(i,k) = E^{(j+1)}(i,k) \text{ für alle } i,k \in G.$$

Nach der Initialisierung gilt diese Aussage für $j = 0$, da dann C die Adjazenzmatrix von E ist, in der im Feld (i, k) genau dann eine 1 steht, wenn $(i, k) \in E$ ist, d.h. wenn es einen E-Weg von i nach k gibt, der außer den Endpunkten gar kein $m \in G$ durchläuft.

Die Ausage gelte für alle $j' < j$. Beim Durchlauf der äußeren Schleife für j werden die Zuweisungen

$$C(i, k) := E^{(j)}(i, k) + E^{(j)}(i, j) * E^{(j)}(j, k)$$

ausgeführt: Da $C(i, k)$ nur einmal aktualisiert wird, hat der Summand $C(i, k)$ der Instruktion nach Induktionsannahme den Wert $E^{(j)}(i, k)$. Im Summand $C(i, j) * C(j, k)$ hat $C(i, j)$ den Wert $E^{(j)}(i, j)$, wenn $j \geq k$, ist, denn dann wurde $C(i, j)$ im aktuellen Durchlauf der äußeren Schleife noch nicht verändert, und (nach Induktion über k) den Wert $E^{(j+1)}(i, j)$ sonst. Ebenso wurde der Faktor $C(j, k)$ im aktuellen Durchlauf der äußeren Schleife noch nicht verändert, wenn $j \geq i$, und hat dann den Wert $E^{(j)}(j, k)$, sonst (nach Induktion) den Wert $E^{(j+1)}(j, k)$. Nun ist aber $E^{(j+1)}(j, k) = E^{(j)}(j, k)$: wenn es einen Weg *mit j als Ausgangspunkt* gibt, der nochmal j trifft, bevor er in k endet, so hat er ein Endstück von j nach k, auf dem er zwischendurch nicht durch j geht. Analog ist $E^{(j+1)}(i, j) = E^{(j)}(i, j)$. Die obige Formel drückt aber aus, daß $C(i, k)$ den Wert 1 erhält, wenn es einen Weg von i nach k gibt, der zwischendurch nicht durch $m \geq j$ geht, oder einen Weg, der einmal durch j, aber durch kein $m > j$ geht. Daher ist $C(i, k) = E^{(j+1)}(i, k)$, wie behauptet. $\square$

Man sieht am Beweis, daß man die Zuweisung bei jedem j für alle (i, k) simultan machen könnte.

3 Halbringe und gewichtete Graphen

In vielen Anwendungsfällen interessiert man sich nicht nur für die Länge von Wegen in Graphen, sondern möchte die einzelnen Kanten gewichten (oder: bewerten) und die Wege nicht nach ihrer Länge, sondern nach den Gewichten der durchlaufenen Kanten messen.

Die Gewichte müssen addiert und multipliziert werden können, damit der Verkettung von Kanten das Produkt ihrer Gewichte und einer Alternative zwischen Kanten die Summe ihrer Gewichte entspricht.

Definition 3.1 Ein *Halbring* $\mathcal{R} = (R, +, 0, \cdot, 1)$ besteht aus einer Menge R mit zweistelligen Operationen $+, \cdot : R \times R \to R$ und Elementen $0, 1 \in R$, so daß für alle $a, b, c \in R$ folgende Axiome gelten:

(i) $+$ ist assoziativ und kommutativ: $a + (b + c) = (a + b) + c$ und $a + b = b + a$,

(ii) 0 ist neutrales Element für $+$: $a + 0 = a$,

(iii) $\cdot$ ist assoziativ: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$,

(iv) 1 ist neutrales Element für $\cdot$: $a \cdot 1 = a = 1 \cdot a$,

(v) $\cdot$ distribuiert über $+$ von beiden Seiten: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ und $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$.

Definition 3.2 Ein *bewerteter* oder *gewichteter Graph* (G, Γ) besteht aus einer Menge G und einer *Bewertung* oder einem *Gewicht* $\Gamma : G \times G \to \mathcal{R}$ in einen Halbring $\mathcal{R} = (R, +^{\mathcal{R}}, 0^{\mathcal{R}}, \cdot^{\mathcal{R}}, 1^{\mathcal{R}})$. Eine *Kante* $p \xrightarrow{r} q$ von (G, Γ) ist ein Paar $(p, q) \in G \times G$ mit Gewicht $r = \Gamma(p, q) \neq 0^{\mathcal{R}}$.

Ein *Weg* π *der Länge* k in (G, Γ) (von p_0 nach p_k) ist eine Folge

$$p_0 \xrightarrow{r_1} p_1, \quad p_1 \xrightarrow{r_2} p_2, \; \ldots, \; p_{k-1} \xrightarrow{r_k} p_k$$

von k Kanten, bei denen der Endpunkt der einen mit dem Anfangspunkt der nächsten Kante übereinstimmt. Wir schreiben dafür kurz

$$\pi \;=\; p_0 \xrightarrow{r_1} p_1 \xrightarrow{r_2} p_2 \xrightarrow{r_3} \;\ldots\; \xrightarrow{r_k} p_k.$$

Das *Gewicht des Weges* π ist das Produkt des Gewichts seiner Kanten,

$$\Gamma(\pi) \quad := \quad r_1 \cdot^{\mathcal{R}} \ldots \cdot^{\mathcal{R}} r_k,$$

unter Beachtung der Reihenfolge, und $1^{\mathcal{R}}$ ist das Gewicht des leeren Wegs ϵ.

Definition 3.3 Sei $\mathit{Wege}(p, q)$ die Menge der Wege π in (G, Γ) mit Anfangspunkt p und Endpunkt q. Der *Abstand von* p *nach* q ist

$$\mathit{dist}(p, q) := \sum^{\mathcal{R}} \{\, \Gamma(\pi) \mid \pi \in \mathit{Wege}(p, q) \,\},$$

sofern diese i.a. unendliche Summe sinnvoll definiert ist.[2]

Der Abstand $\mathit{dist}(p, q) \in \mathcal{R}$ muß nicht unbedingt die Länge $\Gamma(\pi)$ eines $\pi \in \mathit{Wege}(p, q)$ sein.

Bemerkung 3.4 Die Terminologie kommt daher, daß man oft den Ring

$$\mathcal{R} = (R, +^{\mathcal{R}}, 0^{\mathcal{R}}, \cdot^{\mathcal{R}}, 1^{\mathcal{R}}) = (\mathbb{N}_{+\infty}, \min{}^{\mathbb{N}}, +\infty, +^{\mathbb{N}}, 0^{\mathbb{N}})$$

nimmt; dann ist (wegen $\cdot^{\mathcal{R}} = +$) das Gewicht eines Weges die Summe der Gewichte der Kanten, und der kürzeste Weg ist (wegen $+^{\mathcal{R}} = \min$) derjenige, dessen Gewicht im Sinne der Ordnung $\leq^{\mathbb{N}}$ minimal ist. Da jede nichtleere Menge natürlicher Zahlen ein kleinstes Element hat, existiert in $\mathcal{R}$ ein kürzester Weg von p nach q auch dann, wenn $\mathit{Wege}(p, q)$ unendlich ist.

[2] Wir brauchen nur den Fall, wo $\mathit{Wege}(p, q)$ endlich ist und es sich um eine normale Summe von endlich vielen Elementen des Rings handelt.

3.1 Standardbeispiele für Halbringe

Beispiel 3.5 Der Boole'sche Halbring $\mathbb{B} = (\{0,1\}, \max, 0, \min, 1)$, wobei das Maximum und Minimum bezüglich der von $0 < 1$ erzeugten Ordnung genommen werden.

Dieser linear angeordnete Halbring hat verschiedene Verallgemeinerungen zu diskreten und stetigen Skalen, z.B.:

Beispiel 3.6 Der Halbring $\mathbb{N}_{[0,n]} = (\{0,1,\ldots,n\}, \max, 0, \min, n)$ eines Anfangsstücks der natürlichen Zahlen, wobei Maximum und Minimum bezüglich der von $0 < 1 < \ldots < n-1 < n$ erzeugten Ordnung genommen werden.

Beispiel 3.7 Der Halbring $I\!R_{\pm\infty} = (I\!R \cup \{-\infty, +\infty\}, \max, -\infty, \min, +\infty)$, die sogenannte *Max-Min-Algebra*, wobei Maximum und Minimum bezüglich der von

$$-\infty < r <^{I\!R} s < +\infty$$

erzeugten Ordnung genommen werden. Sie ist isomorph zu

$$I\!R_{[0,1]} = ([0,1]^{I\!R}, \max, 0, \min, 1),$$

dem Intervall $[0,1]^{I\!R} := \{\, r \in I\!R \mid 0 \leq^{I\!R} r \leq^{I\!R} 1 \,\}$ der reellen Zahlen zwischen 0 und 1 und Maximum und Minimum bezüglich der reellen Zahlen. Eine Variante hiervon ist

$$I\!R^{\geq 0}_{+\infty} = (I\!R^{\geq 0}_{+\infty}, \max, 0, \min, \infty),$$

bei der man keine negativen Werte zuläßt, also 0 das kleinste Element ist.

Analog dazu sind auch die diskrete unendliche Skala $\mathbb{Z}_{\pm\infty}$ (sowie $\mathbb{N}_{+\infty} = \mathbb{Z}^{\geq 0}_{+\infty}$) und die unvollständige dichte Skala $\mathbb{Q}_{[0,1]}$ Halbringe, wenn man Maximum und Minimum als Operationen $+$ und $\cdot$ nimmt und das kleinste bzw. größte Element als das neutrale Element von $+$ bzw. von $\cdot$.

Beispiel 3.8 Der Halbring $\mathbb{N} = (\mathbb{N}, +, 0, \cdot, 1)$ mit der üblichen Addition $+$ und Multiplikation $\cdot$ von natürlichen Zahlen $\mathbb{N} = \{0,1,2,3,\ldots\}$. Die ganzen Zahlen $(\mathbb{Z}, +, 0, \cdot, 1)$ mit den üblichen Operationen bilden ebenfalls einen Halbring.

Außerdem sind die üblichen Zahlkörper, wenn man die Subtraktion und Division wegläßt, Halbringe, z.B. $(\mathbb{Q}, +, 0, \cdot, 1)$, $(\mathbb{R}, +, 0, \cdot, 1)$, $(\mathbb{C}, +, 0, \cdot, 1)$.

Beispiel 3.9 Ist $\mathcal{M} = (M, \cdot^M, 1^M)$ ein Monoid, z.B. $(\Sigma^*, \cdot, \epsilon)$, so ist

$$\mathcal{P}(\mathcal{M}) = (\mathcal{P}(M), \cup, \emptyset, \cdot, \{1^M\}),$$

die Gesamtheit der Teilmengen von M mit der "elementweisen" Multiplikation

$$A \cdot B := \{\, a \cdot b \mid a \in A,\, b \in B \,\}$$

und der Vereinigung als Addition, ein Halbring.

Beispiel 3.10 Ist M eine Menge, so ist die Potenzmengenalgebra

$$\mathcal{P}(M) = (\{ A \mid A \subseteq M \}, \cup, \emptyset, \cap, M)$$

ein Halbring.

Beispiel 3.11 Die *Min-Plus-Algebra*

$$R = (I\!\!R_{\pm\infty}, \min, +\infty, +, 0)$$

ist ein Halbring, wobei die Addition $+^R$ das Minimum bezüglich der durch $-\infty < r \leq^{I\!\!R} s < +\infty$ definierten Anordung, mit $+\infty$ als neutralem Element 0^R. Die Multiplikation und ihr neutrales Element sind im wesentlichen die Addition auf $I\!\!R$ und die $0^{I\!\!R}$:

$$\begin{array}{rcl} 1^R & := & 0, \\ x \cdot^R y & := & \begin{cases} +\infty, & \text{falls } +\infty \in \{x, y\}, \\ -\infty, & \text{sonst, falls } -\infty \in \{x, y\}, \\ x +^{I\!\!R} y & \text{sonst.} \end{cases} \end{array}$$

Für einen Graphen (G, E, v) mit einer Bewertung $v : E \to R$ in diesen Ring ist das *Gewicht eines Pfads* $\pi = (x_1, x_2) \cdots (x_{k-1}, x_k)$, das Produkt

$$v(\pi) := v(x_1, x_2) \cdot^R \ldots \cdot^R v(x_{k-1}, x_k),$$

also nicht anderes als die reelle Summe

$$v(\pi) = v(x_1, x_2) +^{I\!\!R} \cdots +^{I\!\!R} v(x_{k-1}, x_k),$$

der Kantengewichte, und die Summe der Gewichte zweier Pfade deren Minimum,

$$v(\pi_1) + v(\pi_2) = \min\{v(\pi_1), v(\pi_2)\}.$$

Ist die Menge $P(x, y)$ aller Pfade von x nach y, so ist die unendliche Summe

$$\sum \{ v(\pi) \mid \pi \in P(x, y) \}$$

wohldefiniert, die größte untere Schranke der Länge aller Wege von x nach y. Ein *Weg minimalen Gewichts* von x nach y ist ein $\tilde{\pi} \in P(x, y)$ mit

$$v(\tilde{\pi}) = \sum \{ v(\pi) \mid \pi \in P(x, y) \}.$$

Nimmt die Bewertung nur Werte in $I\!\!R_{\geq 0}$ an, so kann man sie als Längen oder Kosten verstehen und spricht vom *kürzesten Weg* oder vom *Weg mit minimalen Kosten.*

Satz 3.12 Ist $\mathcal{R} = (R, +^R, 0^R, \cdot^R, 1^R)$ ein Halbring und $n \in \mathbb{N}$, ist auch

$$Mat_{n\times n}(\mathcal{R}) = (Mat_{n\times n}(R), +, 0, \cdot, 1)$$

einen Halbring, wobei

$$Mat_{n\times n}(R) \quad := \{\, A \mid A : \{1,\ldots,n\} \times \{1,\ldots,n\} \to R \,\} \tag{5}$$

die Menge der $n \times n$-Matrizen mit Einträgen in R ist und die Operationen $+, \cdot$ und Elemente $0, 1$ wie folgt definiert sind:

$$\begin{aligned}
(A+B)(i,j) &:= A(i,j) +^R B(i,j) && (6)\\
0(i,j) &:= 0^R, && (7)\\
(A\cdot B)(i,j) &:= \sum_{k=1}^{n} A(i,k) \cdot^R B(k,j) && (8)\\
&= A(i,1) \cdot^R B(1,j) \ +^R \ldots +^R \ A(i,n) \cdot^R B(n,j) &&\\
1(i,j) &:= \begin{cases} 1^R, & \text{falls } i = j,\\ 0^R, & \text{sonst.}\end{cases} && (9)
\end{aligned}$$

Daß in $Mat_{n\times n}(\mathcal{R})$ tatsächlich die Axiome eines Halbrings gelten, kann man leicht nachrechnen. Wie man an der Definition sieht, ist $A \cdot B$ eine Verallgemeinerung der Multiplikation $A\,;\,B$ boole'scher Matrizen, d.h. des Relationenprodukts.

Bemerkung 3.13 Der Halbring der Matizen über $\mathcal{R}$ hat i.a. nicht dieselben Eigenschaften wie $\mathcal{R}$. Beispielsweise zeigt (3):

(i) die Multiplikation im Halbring $Mat_{2\times 2}(\mathbb{B})$ ist nicht kommutativ, obwohl die Multiplikation in $\mathbb{B}$ kommutativ ist.

(ii) der Halbring $Mat_{2\times 2}(\mathbb{B})$ hat *Nullteiler*, d.h. Elemente $A \neq 0 \neq B$ mit $A \cdot B = 0$, aber der Halbring $\mathbb{B}$ ist nullteilerfrei.

Bemerkung 3.14 Wie teuer ist eine Anwendung von $+$ oder $\cdot$ in $Mat_{n\times n}(\mathcal{R})$? Für $(A+B)$ braucht man $O(n^2)$ Feldzugriffe und Anwendungen von $+^R$. Für ein Feld (i,j) von $(A \cdot B)$ braucht man $O(n)$ Feldzugriffe, Anwendungen von $\cdot^R$ und Anwendungen von $+^R$, insgesamt für $(A \cdot B)$ also $O(n^3)$ Feldzugriffe und Anwendungen von $\cdot^R$ und $+^R$.

Wir nehmen an, daß man auf ein Matrixfeld in konstanter Zeit zugreifen kann. Dann braucht man für $(A+B)$ $O(n^2)$ Anwendungen von $+^R$ und für $(A \cdot B)$ $O(n^3)$ Anwendungen von $\cdot^R$ und $+^R$.

3.2 Anwendungsbeispiele

Die verschiedenen Beispiele für Halbringe braucht man für verschiedene Arten von Anwendungen. Ein häufiger Fall sind Fragen zu Wegen in Graphen.

Sei (G, E) ein endlicher Graph mit $|G| = n$, oder $E \in Mat_{n\times n}(\mathbb{B})$, und

$$Wege(x, y) = \{\, \pi \mid \pi \text{ ist ein } E\text{-Weg von } x \text{ nach } y \,\}.$$

für $x, y \in G$. Beachte, daß $Wege(x, y)$ unendlich sein kann. Die Länge eines Weges $\pi = (p_1, \ldots, p_k)$ ist die Anzahl der durchlaufenen Kanten,

$$|\pi| = |(p_1, \ldots, p_k)| := k - 1 \in \mathbb{N}.$$

Betrachtet man π als Folge von Kanten (p_i, p_{i+1}) mit Länge $1 = |(p_i, p_{i+1})|$, so ist die Länge des Wegs die Summe der Längen der durchlaufenen Kanten:

$$|\pi| = |(p_1, p_2)| + \ldots + |(p_{k-1}, p_k)| = k - 1.$$

Beispiel 3.15 (Erreichbarkeit) Die *Existenz eines Weges* von x nach y im Graphen (G, E) entspricht dem Wahrheitswert $E^*(x, y)$ der Boole'schen Matrix

$$E^* \quad := \quad \sum_{i\in\mathbb{N}} E^i = \sum_{i=0}^{|G|} E^i. \tag{10}$$

Betrachtet man E als Element von $Mat_{n\times n}(\mathbb{B})$, so rechnet man die *Verbindungsmatrix* E^* oder die *reflexive, transitive Hülle* E^* von E in $Mat_{n\times n}(\mathbb{B})$ mit $n = |G|$ aus.

$E^* \in Mat_{n\times n}(\mathbb{B})$ gibt für je zwei Punkte aus G an, ob mindestens ein E-Weg vom ersten zum zweiten führt. Hierbei braucht man nur Wege der Länge $\leq n-1$ zu berücksichtigen: wenn es überhaupt einen Weg von x nach y gibt, so auch einen, der keinen Punkt zweimal durchläuft, also einen Weg der Länge $\leq n - 1$.

Die Berechnung von E^* nach (10), genauer gesagt, nach der Rekursionsformel

$$\sum_{i=0}^{k+1} E^i \quad := \quad E^0 + (\sum_{i=0}^{k} E^i)E, \tag{11}$$

braucht $O(n)$ Anwendungen von $+$ und $\cdot$, also $O(n^4)$ Anwendungen der Ringoperationen $+^{\mathbb{B}}$ und $\cdot^{\mathbb{B}}$.

Beispiel 3.16 Die *maximale Länge* von Wegen: will man die Länge aller Verbindungen von x nach y im Graphen (G, E) nach oben abschätzen, also eine kleinste obere Schranke $s(x, y)$ für

$$\{\, |\pi| \mid \pi \in Wege(x, y) \,\},$$

so braucht man den Halbring

$$\mathbb{N}'_{+\infty} = (\mathbb{N} \cup \{+\infty\}, \max, +\infty, +, 0),$$

um mit $+$ die Kantenlängen eines Weges π zu addieren, und mit $\max\{|\pi_1|, |\pi_2|\}$ die größere Länge zweier Wege zu berechnen. Wenn $Wege(x, y)$ eine unendliche Menge ist, braucht man

$$s(x, y) = sup\{\, |\pi| \mid \pi \in Wege(x, y)\,\} = +\infty$$

als kleinste obere Schranke der Menge der Weglängen.

Wir betrachten hier die Boole'sche Matrix E als Matrix über dem Halbring $\mathbb{N}'_{+\infty}$ und berechnen $s(x, y) = E^*(x, y)$ mit $E^* \in Mat_{n\times n}(\mathbb{N}_{+\infty})$. Aber hier ist

$$E^*(x, y) = E^0(x, y) +^{\mathbb{N}'_{+\infty}} E^1(x, y) +^{\mathbb{N}'_{+\infty}} \dots$$

nicht durch eine endliche Summe ersetzbar. Bei einer Wegfortsetzung $\pi_1\pi_2$ werden die oberen Schranken addiert

Beispiel 3.17 (Optimierungsproblem) Die *minimale Länge* von Wegen: will man die kürzeste Länge aller Wegen von x und y, so betrachtet man meist Graphen (G, E), deren Kanten ein Gewicht $0 \leq E(x, y) \in \mathbb{R}_{+\infty}$ haben, mit $E(x, y) = +\infty$, wenn es keine Verbindung gibt; also ist $E \in Mat_{n\times n}(\mathbb{R}^{\geq 0}_{+\infty})$. Das Gewicht eines Weges ist die Summe der Kantengewichte:

$$w((p_1, p_2, \dots, p_k)) = \sum_{i=1}^{n-1} E(p_i, p_{i+1}).$$

Der Verkettung von Wegen wird die Summe der Einzelgewichte (Kosten) zugeordnet, einer Menge W von Wegen die kleinste untere Schranke

$$\inf\{\, w(\pi) \mid \pi \in W\,\}.$$

Man braucht also die Matrizen über dem Ring $(\mathbb{R}^{\geq 0}_{+\infty}, \min, 0, +, 0)$. Da keine negativen Gewichte zugelassen sind, können verlängerte Wege nur ein größeres Gewicht haben als die Anfangsstücke. Es muß daher reichen, wenn man

$$E^+ = E + E^2 + E^3 + \ldots + E^n$$

berechnet.

Beispiel 3.18 (Flaschenhalsprobleme, Wege maximaler Kapazität) In einem Graphen, dessen Kanten eine „Weite" haben, hat jeder Weg einen *Engpaß*, eine Kante mit minimaler Weite. Es soll in einem solchen Graphen bestimmt werden, was die größtmögliche „Bequemlichkeit", d.h. die *maximale Engpaßweite*, unter den Wegen von x nach y ist. (Z.B. soll unter allen Verbindungen eine benutzt werden, auf der die niedrigste Brücke, unter der man hindurchfahren muß, möglichst hoch ist.) Hier braucht man die Max-Min-Algebra.

3.3 Implementierung (in Standard ML)

Wir wollen Ringe und Matrizen über einem Ring so implementieren, daß die Matrizenoperationen relativ zu einem unbekannten Ring definiert sind. Durch Angabe eines konkreten Rings R und einer Dimension n sollen dann die $n \times n$-Matrizen über R als Spezialfall entstehen.

Zunächst definieren wir *Signaturen*, d.h. die Namen der jeweils gewünschten Operationen, ihre Stelligkeit und die Typen ihrer Argumente und Werte:

⟨*semiring.sml*⟩≡

```
signature MONOID =
    sig
        type t    (* Typ der Individuen *)
        val sum : t * t -> t
        val zero : t
    end

signature SEMIRING =
    sig
        type t
        val sum : t * t -> t
        val zero : t
        val prod : t * t -> t
        val unit : t
    end
```

Bei den Matrizen definieren wir eine Signatur für die Matrizen aller Dimensionen über einem Ring, aus der die einer festen Dimension ausgewählt werden:

⟨*semiring.sml*⟩+≡

```
signature MATRICES =
    sig
        structure Ring : SEMIRING
        type t
        val sum : t * t -> t
        val zero : int * int -> t
        val unit : int * int -> t
        val prod : t * t -> t
        val dimension : t -> int * int
        val sub : t * (int * int) -> Ring.t
        val map : (Ring.t -> Ring.t) -> t -> t
        val tabulate : (int * int) -> (int * int -> Ring.t) -> t
        val scale : Ring.t * t -> t
        val tran : t -> t
    end
```

```
signature MATRIX =
    sig
        structure Ring : SEMIRING
        type t
        val dim : int * int
        val sum : t * t -> t
        val zero : t
        val prod : t * t -> t
        val unit : t
        val sub : t * (int * int) -> Ring.t
        val map : (Ring.t -> Ring.t) -> t -> t
        val tabulate : (int * int -> Ring.t) -> t
        val scale : Ring.t * t -> t
        val tran : t -> t
    end
```

Hier ist `sub(A,(i,j))` der Zugriff auf das (i,j)-te Element von A, `map f A` die Anwendung der Funktion f auf alle Elemente von A, `tabulate (n,m) f` erzeugt aus f eine $n \times m$-Matrix A mit `sub(A,(i,j)) = f(i,j)` und `scale(r,A)` multipliziert alle Einträge von `A` von links mit dem Ringelement `r`.

Neben den Signaturen brauchen wir noch *Funktoren*, mit denen Ringe in Abhängigkeit von gewissen Argumenten gebildet werden.

Um die endliche, diskrete Skala $(\mathbb{N}_{[0,n]}, \max, 0, \min, n)$ zu bilden, muß man die maximale Größe n vorgeben:

$\langle$*semiring.sml*$\rangle$+≡

```
functor ScaleDisc (val max : int) =
   struct
      type t = int
      fun sum(x,y) = Int.max(x,y)
      val zero = 0
      fun prod(x,y) = Int.min(x,y)
      val unit = max
   end
```

Um den Ring $Mat_{n\times n}(R)$ der $n \times n$-Matrizen über einem Ring R zu bilden, muß man sowohl die Dimension als auch den Ring vorgeben. Der Funktor `Matrix` benutzt den Hilfsfunktor `Matrices`.

$\langle$*semiring.sml*$\rangle$+≡

```
functor Matrices (R : SEMIRING) : MATRICES =
    struct
        structure Ring = R
        type r = Ring.t
```

```
type t = r vector vector

fun dimension (S:t) =
    let
        val rows = Vector.length S
        val columns = Vector.length (Vector.sub(S,0))
        val samelengths =
            Vector.foldr (fn (v,b) => b andalso
                                          (Vector.length v = columns))
                         true S
    in
        if samelengths then (rows,columns) else raise Size
    end

fun sub (E:t,(i,j)) =
    let
        val (d1,d2) = dimension E
    in
        if 0 <= i andalso i < d1 andalso 0 <= j andalso j < d2
            then Vector.sub (Vector.sub (E,i), j)
        else raise Size
    end

fun tabulate (rows,cols) (f : int * int -> r) =
    Vector.tabulate (rows, fn i => Vector.tabulate
                                   (cols, fn k => f(i,k))) : t

fun map (f : r -> r) (A : t) =
    let
        val (n,m) = dimension (A:t)
    in
        tabulate (n,m) (fn (i,j) => f (sub (A,(i,j))))
    end

fun zero (n,m) = tabulate (n,m) (fn (i,k) => R.zero)
fun unit (n,m) =
    tabulate (n,m) (fn (i,k) =>
                   if i=k then R.unit else R.zero)

fun sum (S:t,T:t) : t =
    let
        val (ns,ms) = dimension S
        val (nt,mt) = dimension T
        fun sumV (v:r vector,w:r vector) : r vector =
            Vector.tabulate (Vector.length v,
```

```
                                  fn i => R.sum (Vector.sub(v,i),
                                                 Vector.sub(w,i)))
            in
                if (ns,ms)=(nt,mt) then
                    Vector.tabulate (ns, fn i => sumV (Vector.sub (S,i),
                                                       Vector.sub (T,i)))
                else raise Size
            end

        fun prod (S:t,T:t) : t =
            let
                val (ns,ms) = dimension S
                val (nt,mt) = dimension T
                fun prodV  (v:r vector,w:r vector) : r =
                    let
                        fun sm i = if i=0 then R.zero
                                   else R.sum(sm (i-1),
                                              R.prod (Vector.sub(v,i-1),
                                                      Vector.sub(w,i-1)))
                    in
                        sm (Vector.length v)
                    end
                fun column (T:t,j) = Vector.map (fn r => Vector.sub (r,j)) T
            in
                if ms=nt then
                    tabulate (ns,mt) (fn (i,j) => prodV (Vector.sub(S,i),
                                                         column(T,j)))
                else raise Size
            end

        fun scale (s:r, S:t) : t =
            tabulate (dimension S) (fn (i,j) => R.prod(s,sub(S,(i,j))))

        fun tran (S:t) =
            let val (n,m) = dimension S
            in
                tabulate (m,n) (fn (i,j) => sub(S,(j,i)))
            end
    end

functor Matrix (structure R:SEMIRING; val rows:int; val cols:int) : MATRIX =
    let
        structure M = Matrices(R)
    in
        struct
            structure Ring = R : SEMIRING
```

```
            val dim = (rows,cols)
            type t = M.t
            val sum = M.sum
            val zero = M.zero dim
            val prod = M.prod
            val unit = M.unit dim
            val sub = M.sub
            val map = M.map
            val tabulate = M.tabulate dim
            val scale = M.scale
            val tran = M.tran
        end
    end
```

Beispiel 3.19 (BoolRing, IntRing, Boole) Da SML schon Strukturen `Bool` und `Int` hat, können wir deren Funktionen übernehmen und der Signatur `SEMIRING` anpassen.

⟨*semiring.expl.sml*⟩≡

```
(* Compiler.Control.Print.printDepth := 30; *)

structure BoolRing : SEMIRING =
    struct
        type t = bool
        val zero = false
        fun sum (x,y) = x orelse y
        val unit = true
        fun prod (x,y) = x andalso y
    end

structure IntRing : SEMIRING =
    struct
        type t = int
        val zero = 0
        fun sum (x,y) = x + y
        val unit = 1
        fun prod (x,y) = x * y
    end
```

Um nicht mit den langen Werten `true` und `false` arbeiten zu müssen, definieren wir eine Variante zu `BoolRing`, die diskrete Skala mit den Werten 1 und 0:

⟨*semiring.expl.sml*⟩+≡

```
structure Boole = ScaleDisc (val max = 1)
```

Wenn man die Funktoren geladen hat, kann man sie auch interaktiv verwenden, z.B. um die Boole'schen Matrizen zu bilden:

Beispiel 3.20 (Interaktive Anwendung von `Matrices`) Nachdem SML gestartet ist[3], kann man mit `use` die obigen Dateien laden:

⟨*SML aufrufen und Dateien laden*⟩≡
```
Standard ML of New Jersey, Version 110.0.3, January 30, 1998
- use "semiring.sml";
- [opening semiring.sml]
GC #0.0.0.0.1.3:   (5 ms)
...
functor Matrices (R : SEMIRING) : MATRICES =
...
- use "semiring.expl.sml";
...
```

Nun kann man die Boole'schen Matrizen beliebiger Dimension bilden, und dann die früheren Beispielmatrizen A und B definieren:

⟨*Boole'sche Matrizen*⟩≡
```
- structure BoolMat = Matrices(BoolRing);

- val A = BoolMat.tabulate (2,2)
          (fn (0,0) => false | (0,1) => true
            | (1,0) => false | (1,1) => false
            | (_,_) => raise Size);
val A = #[#[false,true],#[false,false]] : BoolMat.t

- val B = BoolMat.tabulate (2,2)
          (fn (0,0) => false | (0,1) => false
            | (1,0) => false | (1,1) => true
            | (_,_) => raise Size);
val B = #[#[false,false],#[false,true]] : BoolMat.t

- BoolMat.prod(A,B);
val it = #[#[false,true],#[false,false]] : BoolMat.t
- BoolMat.prod(B,A);
val it = #[#[false,false],#[false,false]] : BoolMat.t
```

Wie man sieht, ist $A \cdot B = A \neq 0 = B \cdot A$, wie oben in (3) schon ausgerechnet.

Beim Multiplizieren von Matrizen wird die Dimension der Matrizen überprüft:

⟨*Überprüfung der Dimensionen*⟩≡
```
- val C = BoolMat.prod (BoolMat.unit (2,3),BoolMat.unit (3,2));
```

[3] im CIP-Pool der Informatik: mit `/soft/bin/sml-cm` bekommt man SML-NJ Version 110.0.6 (mit dem Compilation Manager CM)

```
val C = #[#[true,false],#[false,true]] : BoolMat.t
- val D = BoolMat.prod (BoolMat.unit (2,4),BoolMat.unit (3,2));
uncaught exception size
   raised at: semiring.sml:131.28-131.32
```

Wenn man die *langen Namen* mit dem Strukturpräfix nicht schreiben will, kann man eine Struktur öffnen und dann die *kurzen Namen* der darin definierten Typen, Werte und Unterstrukturen auf Top-Level verwenden:

⟨*Öffnen einer Struktur und Verwenden der kurzen Namen*⟩≡

```
- open BoolMat;
opening BoolMat
  structure Ring :
    sig
      type t = bool
      val sum : t * t -> t
      val zero : t
      val prod : t * t -> t
      val unit : t
    end
  type t = BoolRing.t vector vector
  val sum : t * t -> t
  val zero : int * int -> t
  val unit : int * int -> t
  val prod : t * t -> t
  val dimension : t -> int * int
  val sub : t * (int * int) -> Ring.t
  val map : (Ring.t -> Ring.t) -> t -> t
  val tabulate : int * int -> (int * int -> Ring.t) -> t

- val C = prod (unit (2,3), unit (3,2));
val C = #[#[true,false],#[false,true]] : t
```

Beispiel 3.21 (Matrizen über $\mathbb{Z}$) Die Konstruktion einer Matrix mit `tabulate (n,m) f` ist zwar bei Boole'schen Matrizen nicht naheliegend, aber dann recht bequem, wenn sich die Einträge aus den Zeilen- und Spaltennummern berechnen:

⟨*Matrizen über den ganzen Zahlen*⟩≡

```
- structure IntMat =
  Matrix (structure R = IntRing val rows = 3 val cols = 3);

- val I1 = IntMat.tabulate (fn (i,j) => i + j)
  and I2 = IntMat.tabulate (fn (i,j) => i * ~j);
val I1 = #[#[0,1,2],#[1,2,3],#[2,3,4]] : IntMat.t
val I2 = #[#[0,0,0],#[0,~1,~2],#[0,~2,~4]] : IntMat.t
```

```
- val I = IntMat.prod (I1,I2);
val I = #[#[0,~5,~10],#[0,~8,~16],#[0,~11,~22]] : IntMat.t
```

Beispiel 3.22 (Berechnung der transitiven Hülle A^+ einer Relation A) Sei $E \subseteq G \times G$ der Graph aus Beispiel 2.2. Wir implementieren E als Boole'sche Matrix, wobei wir zur besseren Lesbarkeit `0,1` statt `false,true`, also `Boole` statt `BoolRing` benutzen. (Da die Numerierung der Zeilen und Spalten in SML bei 0 beginnt, nennen wir die Elemente von G via $\{1,2,3,4\} \simeq \{0,1,2,3\}$ um.)

⟨*graph1.sml*⟩≡

```
structure BM4 = Matrix(structure R = Boole;
                       val rows = 4; val cols = 4);

val E = BM4.tabulate
  (fn (0,0) => 0 | (0,1) => 0 | (0,2) => 1 | (0,3) => 0
    | (1,0) => 0 | (1,1) => 0 | (1,2) => 0 | (1,3) => 1
    | (2,0) => 0 | (2,1) => 0 | (2,2) => 0 | (2,3) => 1
    | (3,0) => 1 | (3,1) => 0 | (3,2) => 0 | (3,3) => 0);

fun wege A n = if n = 1 then A else
    let
        val B = wege A (n-1)
    in
        BM4.sum(B, BM4.prod(B, A))
    end;

fun tc A = wege A 4;
```

Für $A \subseteq G$ mit $|G| = n$ wird hiermit die transitive Hülle von A berechnet:

$$A^+ = \sum_{i=1}^{n} A^i = \mathit{wege}\ A\ n = \mathit{tc}(A).$$

Das wird analog zu (10) gemacht, kostet also $O(n^4)$ Schritte.

⟨*Erreichbarkeit in höchstens $n \geq 1$ Schritten*⟩≡

```
- use "graph1.sml";
[opening graph1.sml]
...
- wege E 1;
val it = #[#[0,0,1,0],#[0,0,0,1],#[0,0,0,1],#[1,0,0,0]] : BM4.t
- wege E 2;
val it = #[#[0,0,1,1],#[1,0,0,1],#[1,0,0,1],#[1,0,1,0]] : BM4.t
- wege E 3;
val it = #[#[1,0,1,1],#[1,0,1,1],#[1,0,1,1],#[1,0,1,1]] : BM4.t
- wege E 4;
val it = #[#[1,0,1,1],#[1,0,1,1],#[1,0,1,1],#[1,0,1,1]] : BM4.t
```

Da durch Wegen der Länge 4 nicht mehr Punkte verbunden werden als durch Wege der Länge 3, muß das die transitive Hülle sein.

⟨*Transitive Hülle von E*⟩≡
```
- tc E;
val it = #[#[1,0,1,1],#[1,0,1,1],#[1,0,1,1],#[1,0,1,1]] : BM4.t
```

Wir hatten E^2 von Hand ausgerechnet und erhalten hier dasselbe Ergebnis:

⟨*Erreichbarkeit mit Wegen der Länge = 2*⟩≡
```
- BM4.prod(E,E);
val it = #[#[0,0,0,1],#[1,0,0,0],#[1,0,0,0],#[0,0,1,0]] : BM4.t
```

Beispiel 3.23 (Berechnung der transitiven Hülle A^+) Nach Warshall[8] kann A^+ für $A \in Mat_{n\times n}(\mathbb{B})$ in $O(n^3)$ Schritten berechnet werden. Das können wir wie folgt implementieren, wobei wir statt der Zuweisungen

$$C(i,k)^{(j+1)} := C(i,k)^{(j)} + C(i,j)^{(j)} \cdot C(j,k)^{(j)}$$

neue Matrizen $C^{(1)},\ldots,C^{(n)}$ anlegen, was nur $O(n^3)$ zusätzliche Schritte erfordert, allerdings den n-fachen Platz (oder den doppelten, bei einem klugen Compiler).

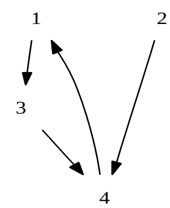
⟨*warshall.sml*⟩≡
```
structure Boole = ScaleDisc(val max = 1);
structure BM = Matrices(Boole);

fun wegew A j =
    let
        val (n,m) = BM.dimension A
        val sumR  = BM.Ring.sum
        val prodR = BM.Ring.prod
    in
        if not(n=m) then raise Size
        else
            if j = 0 then A
            else
                let
                    val C = wegew A (j-1)
                in
                    BM.tabulate (n,n)
                    (fn (i,k) => sumR(BM.sub(C,(i,k)),
                                      prodR(BM.sub(C,(i,j-1)),
                                            BM.sub(C,(j-1,k)))))
                end
    end

fun tcw A =
```

```
let
    val (n,m) = BM.dimension A
in
    if not(n=m) then raise Size else wegew A n
end
```

Damit erhalten wir für das frühere Beispiel



mit der Matrix $\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$

die folgende Berechnung:

$\langle$*Berechnung von E^+ nach Warshall*$\rangle \equiv$

```
- use "warshall.sml";
[opening warshall.sml]
...
- use "graph1.sml";
[opening graph1.sml]
...
val E = #[#[0,0,1,0],#[0,0,0,1],#[0,0,0,1],#[1,0,0,0]] : BM4.t

- wegew E 0;
val it = #[#[0,0,1,0],#[0,0,0,1],#[0,0,0,1],#[1,0,0,0]] : BM.t
- wegew E 1;
val it = #[#[0,0,1,0],#[0,0,0,1],#[0,0,0,1],#[1,0,1,0]] : BM.t
- wegew E 2;
val it = #[#[0,0,1,0],#[0,0,0,1],#[0,0,0,1],#[1,0,1,0]] : BM.t
- wegew E 3;
val it = #[#[0,0,1,1],#[0,0,0,1],#[0,0,0,1],#[1,0,1,1]] : BM.t
- wegew E 4;
val it = #[#[1,0,1,1],#[1,0,1,1],#[1,0,1,1],#[1,0,1,1]] : BM.t
- tcw E;
val it = #[#[1,0,1,1],#[1,0,1,1],#[1,0,1,1],#[1,0,1,1]] : BM.t
-
```

Beachte, daß `wegew E j` nicht die Erreichbarkeit in $1 \leq i \leq j$ Schritten darstellt, sondern die Erreichbarkeit über die Zwischenpunkte $1, \ldots, j$.

Wir haben hier allerdings `tcw` nicht innerhalb der Struktur `Matrices(Boole)` implementiert, von wo man es in die mit dem Funktor `Matrix` erzeugten Strukturen quadratischer Boole'scher Matrizen fester Dimension übertragen könnte. (Der Grund ist, daß `Matrices` auf beliebige Ringe anwendbar sein sollte.)

Beachte außerdem, daß `E:BM4.t` war, aber `tcw:BM.t -> BM.t`. Die Anwendung `tcw(E)` geht nur deshalb, weil die Signaturbeschränkung in `fun Matrices(...) : MATRICES` transparent ist und daher `BM4.t = BM.t` erkannt wird.

4 Parsing mit kontextfreien Grammatiken

Sei *CFG* die Menge aller kontextfreien Grammatiken $G = (N, \Sigma, S, P)$ mit *Kategorien* oder *Nonterminal*symbolen N, der *Hauptkategorie* oder dem *Startsymbol* $S \in N$, der endlichen *Regelmenge* $P \subseteq N \times (N \cup \Sigma)^*$ über demselben *Alphabet* Σ von Buchstaben oder *Terminal*symbolen.

Definition 4.1 Das *universelle Parseproblem für kontextfreie Grammatiken über* Σ ist das Berechnungsproblem, für die Funktion

$$parse : CFG \times \Sigma^* \to \mathcal{P}(Tree)$$

ein Berechnungsverfahren anzugeben, das zu jedem $G \in CFG$ und $w \in \Sigma$ die Menge $parse(G, w)$ aller Syntaxbäume von w bezüglich G ausgibt.

Das *spezielle Parseproblem für* $G \in CFG$ ist das Berechnungsproblem für

$$parse_G : \Sigma^* \to \mathcal{P}(Tree) \quad \text{mit} \quad parse_G(w) := parse(G, w);$$

Beachte, daß man das universelle Problem i.a. nicht lösen muß, um das spezielle Problem für G zu lösen. Wenn z.B. P in G leer ist, kann man $parse_G$, durch $parse_G(w) := \emptyset$ berechnen; dazu braucht man kein Berechnungsverfahren für das universelle Problem zu kennen.

Das spezielle Parseproblem für G ist unlösbar, wenn es eine Eingabe w gibt, für die es unendlich viele Syntaxbäume gibt, und das kann vorkommen, wenn G Kettenregeln oder Löschregeln hat.[4]

Anstatt zu zeigen, daß das Parseproblem für *CFG* und die Multiplikation Boole'scher Matrizen auf einander reduzierbare Berechnungsprobleme sind, betrachtet man nur folgende vom Parseproblem abgeleitete Entscheidungsprobleme:

Definition 4.2 Das *universelle Erkennungsproblem für kontextfreie Grammatiken über* Σ ist

$$erkenne := \{ (G, w) \in CFG \times \Sigma^* \mid w \in L(G) \}.$$

Das *spezielle Erkennungsproblem für* $G \in CFG$ ist

$$erkenne_G := \{ w \in \Sigma^* \mid w \in L(G) \}.$$

Das Erkennungsproblem ist wegen

$$w \in L(G) \iff parse(G, w) \neq \emptyset,$$

[4]Damit das Parseproblem lösbar wird, sollte man eine Äquivalenzrelation ziwschen Syntaxbäumen definieren, die nur endlich viele Äquivalenzklassen hat, und von jeder einen Repräsentanten im Ergebnis haben. Eine andere Möglichkeit ist, sich auf eine Teilklasse von Grammatiken zu beschränken, bei der das nicht vorkommt.

die Frage, ob die Eingabe überhaupt einen Syntaxbaum hat oder nicht.

Für Grammatiken, bei denen das Parseproblem lösbar ist, kann man es auf das Erkennungsproblem reduzieren, denn aus den Daten, die man zur Beantwortung des Erkennungsproblems für G bei Eingabe w berechnet, kann man auch die Syntaxbäume der Eingabe rekonstruieren.

Allerdings kann die Menge der Syntaxbäume zu w immer noch exponentiell groß in $|w|$ sein, z.B. bei

$$G = (\{A\}, \{a\}, A, \{A \to AA,\ A \to a\}),$$

wo man zu $A \Rightarrow_G^* A^n \Rightarrow a^n$ die verschiedenen „Klammerungen" der A^n verschiedene Syntaxbäume ergeben. Manchmal läßt man daher als Ausgabe von $parse_G(w)$ eine Datenstruktur zu, aus der alle Syntaxbäume mit beschränktem Aufwand rekonstruierbar sind.

4.1 Der Erkenner von Cocke, Younger und Kasami

Das einfachste Erkennungsverfahren ist das von Cocke, Younger und Kasami (1967) unabhäng von einander gefundene und naheliegende Verfahren.

Man setzt voraus, daß die Grammatik in *Chomsky-Normalform* vorliegt, d.h. nur Regeln der Form $A \to BC$ mit $A, B, C \in N$ und $A \to a$ mit $a \in \Sigma$ hat. (Insbesondere kommen keine Ketten- oder Löschregeln vor, und damit hat jede Eingabe nur endlich viele Syntaxbäume.)

Zur Eingabe $w = a_1 \dots a_n$ macht man eine Tabelle, in der man für $0 \leq i < j \leq n$ im Feld $T(i,j)$ die $A \in N$ einträgt, für die $A \Rightarrow_G^* a_{i+1} \cdots a_j$ gilt.

$\langle$*CYK-Algorithmus, Pseudocode*$\rangle\equiv$
```
Für i = 0,...,n-1
    T(i,i+1) := {A| (A -> a_i+1) in P}

Für 0 <= i < j <= n mit 1 < j-i:
    T(i,j) := {A| es gibt k mit i < k < j,
                  B in T(i,k), C in T(k,j) und (A-> BC) in P}
```

In welcher Reihenfolge man die Felder füllt, ist egal, außer daß man alle $T(i,k)$ und $T(k,j)$ mit $i < k < j$ schon gefüllt haben muß, wenn man $T(i,j)$ füllt.

Man kann sich T als eine $(n+1) \times (n+1)$-Matrix vorstellen, bei der nur die *obere Dreiecksmatrix*, die Felder oberhalb der Hauptdiagonalen, gefüllt sind.

Beispiel 4.3 Sei G die Grammatik mit den Regeln

$$\begin{array}{lllllllll}
C & \to & AB & A & \to & a & D & \to & a \\
D & \to & CE & B & \to & b & E & \to & b \\
F & \to & BE & C & \to & c & F & \to & c \\
E & \to & CF & & & & & &
\end{array}$$

und $S = E$. Für die Eingabe $w = abb = a_1a_2a_3$ erhalten wir die folgenden Matrizen: Zuerst füllen wir die Felder $T(i, i+1)$, dann die Felder $T(i, i+2)$, dann die Felder $T(i, i+3)$, jeweils für alle $i = 0, 1, 2$:

	0	1	2	3
0		A, D	?	?
1			B, E	?
2				B, E
3				

	0	1	2	3
0		A, D	C	?
1			B, E	F
2				B, E
3				

	0	1	2	3
0		A, D	C	D
1			B, E	F
2				B, E
3				

Eine andere Möglichkeit wäre, die Tabelle spaltenweise von links nach rechts zu füllen, und jede Spalte von unten nach oben:

	0	1	2	3
0		A, D	?	?
1			?	?
2				?
3				

	0	1	2	3
0		A, D	C	?
1			B, E	?
2				?
3				

	0	1	2	3
0		A, D	C	D
1			B, E	F
2				B, E
3				

Satz 4.4 Die durch den Algorithmus erstellte Matrix T hat die Eigenschaft: Für alle $0 \leq i < j \leq n$ und $A \in N$ gilt:

$$A \in T(i, j) \iff A \Rightarrow_G^* a_{i+1} \cdots a_j \tag{12}$$

Korollar 4.5 Für die durch den Algorithmus erstellte Matrix T gilt:

$$S \in T(0, n) \iff S \Rightarrow_G^* a_1 \cdots a_n \iff w \in L(G).$$

Beweis: Durch Induktion über $|j - i|$ (für die erste Art, die Tabelle zu füllen):
Fall $j - i = 1$, also $j = i + 1$: Nach Definition von T ist

$$A \in T(i, j) \iff (A \to a_{i+1}) \in P \iff A \Rightarrow_G^* a_{i+1},$$

wobei benutzt wird, daß P außer den lexikalischen Regeln keine Kettenregeln $B \to C$ und keine Löschregeln $B \to \epsilon$ hat.

Fall $j - i > 1$: Für jedes k mit $i < k < j$ ist sowohl $k - i < j - i$ als auch $j - k < j - i$, so daß nach Induktionsannahme schon gilt:

$$\begin{aligned} T(i, k) &= \{ B \in N \mid B \Rightarrow_G^* a_{i+1} \cdots a_k \} \\ T(k, j) &= \{ C \in N \mid C \Rightarrow_G^* a_{k+1} \cdots a_j \} \end{aligned}$$

Nach Definition von T ist daher

$$\begin{aligned} A \in T(i, j) \iff & \text{Für ein } i < k < j : \\ & \text{Es gibt } B \in T(i, k), C \in T(k, j) \text{ und } (A \to BC) \in P \\ \iff & \text{Für ein } i < k < j : \\ & \text{Es gibt } (A \to BC) \in P \text{ mit } B \Rightarrow_G^* a_{i+1} \cdots a_k \\ & \qquad\qquad\qquad \text{und } C \Rightarrow_G^* a_{k+1} \cdots a_j \\ \iff & A \Rightarrow_G^* a_{i+1} \cdots a_j, \end{aligned}$$

wobei im letzten Schritt wieder benutzt wird, daß es keine Lösch- und Kettenregeln gibt. □

Bemerkung 4.6 Man kann sich das Erkennen eines Wortes w bezüglich einer CNF-Grammatik auch als Berechnung kürzester Wege vorstellen, wie folgt:

Man betrachte das Wort $w = a_1 \cdots a_n$ als den Multigraphen

$$\mathcal{G}_w = (\{0, 1, \ldots, n\}, \langle \xrightarrow{c} \rangle_{c \in \Sigma}),$$

dessen Punkte die Positionen $(0), 1, \ldots, n$ der Buchstaben im Wort w sind, und der für jeden Buchstaben $a \in \Sigma$ eine Relation

$$\xrightarrow{a} = \{\, (i, i+1) \mid 0 \leq i < n,\ a_{i+1} = a \,\},$$

hat, durch die gerade die aufeinanderfolgenden Positionen i und $i+1$ verbunden werden, an denen in w der Buchstabe a vorkommt. Nun kann man die Menge der Kanten induktiv durch die Regeln

$$\frac{i \xrightarrow{a} i+1}{i \xrightarrow{A} i+1}\ ((A \to a) \in P) \qquad \frac{i \xrightarrow{B} k \quad k \xrightarrow{C} j}{i \xrightarrow{A} j}\ ((A \to BC) \in P)$$

erweitern, bis keine neuen (markierten) Kanten hinzukommen. Dann gibt es die Kante $i \xrightarrow{A} j$ genau dann, wenn $A \Rightarrow_G^* a_{i+1} \cdots a_j$. Insbesondere gibt es die Kante $0 \xrightarrow{S} n$ genau dann, wenn $w \in L(G)$.

Anstatt neue Kanten (Wege der Länge 1) einzuführen, kann man auch sagen, der Weg

$$i \xrightarrow{a_{i+1}} i+1 \xrightarrow{a_{i+2}} \cdots j-1 \xrightarrow{a_j} j$$

habe (unter anderem) das *Gewicht* A, falls $A \Rightarrow_G^* a_1 \cdots a_n$. Dann ist die Frage, ob der Weg $0 \xrightarrow{a_1} 1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} n$ das Gewicht S hat.

5 Der Algorithmus von Valiant (Skizze)

Die Hauptidee des Verfahrens von Valiant[7] ist, das Füllen der Parsematrix T als die Berechnung einer transitiven Hülle zu betrachten. Man berechnet die transitive Hülle D^+ einer Matrix D über einem „nichtassoziativen Ring" $\mathcal{R}^G$ rekursiv über die Dimension von D. Da man die Multiplikation von Matrizen über $\mathcal{R}^G$ auf die von Matrizen über $\mathbb{B}$ zurückführen kann, wird insgesamt das Erkennungsproblem für kontextfreie Grammatiken in Chomsky-Normalform auf die Multiplikation Boole'scher Matrizen reduziert.

5.1 Matrixmultiplikation im „Fastring" $\mathcal{R}^G$

Im Verfahren von Cocke, Younger und Kasami werden für den Fall $j - i > 1$ die Felder $T(i, j)$ durch

$$T(i,j) = \bigcup_{i<k<j} \{ A \mid (A \to BC) \in P,\ B \in T(i,k), C \in T(k,j) \} \tag{13}$$

definiert. Die einzelnen Mengen hierin kann man sich als ein Produkt

$$T(i,k) \cdot^P T(k,j) := P^{-1}(T(i,k) \cdot T(k,j))$$

der Mengen $T(i,k)$ und $T(k,j)$ vorstellen, dem Urbild der „elementweisen Verkettung" dieser Mengen,

$$T(i,k) \cdot T(k,j) = \{ BC \mid B \in T(i,k), C \in T(k,j) \},$$

unter der Relation $P \subseteq N \times N^+$. Betrachtet man die Mengenvereinigung als Summe $+$, so ist (13) die Berechnung der „oberen Diagonale" aller (i,j) mit $j - i = d > 1$ aus den Oberdiagonalen aller (i', j') mit $j' - i' = d' < d$.

Beispiel 5.1 (Fortsetzung von Beispiel 4.3) Definiert man die Multiplikation von Matrizen mit Mengen als Einträgen durch die übliche Formel mit $\cup$ für $+$ und $\cdot^P$ für $\cdot$ der Einträge und nennt man die Matrixmultiplikation wieder $\cdot^P$, so ergibt sich für die Berechnung der Tabelle aus Beispiel 4.3:

$$\begin{pmatrix} & A,D & ? & ? \\ & & B,E & ? \\ & & & B,E \\ & & & \end{pmatrix} \cdot^P \begin{pmatrix} & A,D & ? & ? \\ & & B,E & ? \\ & & & B,E \\ & & & \end{pmatrix} = \begin{pmatrix} & \emptyset & C & ? \\ & & \emptyset & F \\ & & & \emptyset \\ & & & \end{pmatrix}$$

und entsprechend für die nächste Oberdiagonale

$$\begin{pmatrix} & A,D & C & ? \\ & & B,E & F \\ & & & B,E \\ & & & \end{pmatrix} \cdot^P \begin{pmatrix} & A,D & C & ? \\ & & B,E & F \\ & & & B,E \\ & & & \end{pmatrix} = \begin{pmatrix} & \emptyset & C & E \\ & & \emptyset & F \\ & & & \emptyset \\ & & & \end{pmatrix}$$

Wir machen das nun allgemein nochmal, damit wir so etwas wie einen Ring $(\mathcal{P}(N), \cup, \emptyset, \cdot^P)$ bekommen.

Man kann mit der Regelmenge P eine Multiplikation $\cdot^P$ auf $\mathcal{P}(N)$ definieren,

$$N_1 \cdot^P N_2 \quad := \quad \{\, A \in N \mid (A \to BC) \in P,\ A \in N_1,\ B \in N_2 \,\},$$

und als Addition und 0 die Vereinigung und die leere Menge nehmen:

$$\begin{aligned} N_1 +^P N_2 &:= N_1 \cup N_2 \\ 0^P &:= \emptyset. \end{aligned}$$

Allerdings kann man die dadurch entstehende Struktur

$$\mathcal{R}^G := (\mathcal{P}(N), +^P, 0^P, \cdot^P)$$

nicht zu einem Ring erweitern: einerseits hat $\cdot^P$ kein neutrales Element, und andererseits ist $\cdot^P$ nicht assoziativ. Aber es gelten die Ringaxiome für $+$ und 0,

$$\begin{aligned} (X+Y)+Z &= X+(Y+Z), \\ X+Y &= Y+X, \\ X+0 = \quad X &= 0+X \\ X \cdot 0 = \quad 0 &= 0 \cdot X \end{aligned}$$

und die Distributivgesetze,

$$\begin{aligned} X \cdot (Y+Z) &= (X \cdot Y) + (X \cdot Z), \\ (X+Y) \cdot Z &= (X \cdot Z) + (Y \cdot Z). \end{aligned}$$

Wir können die Berechnung von T in Einzelschritten vornehmen, beginnend mit

$$T^{(1)}(i,j) := \begin{cases} T(i,j), & \text{falls } j = i+1, \\ \emptyset, & \text{sonst.} \end{cases}$$

Diese Matrix gibt die lexikalischen Kategorien der Buchstaben a_i an ihren Vorkommenspositionen i in w an, die in den Feldern $T(i,j)$ mit $j-1=1$ stehen, also die in der 1.ten Oberdiagonale $\{\, (i,i+1) \mid 0 \leq i < n \,\}$. Angenommen, $T^{(d)}$ gibt in $T^{(d)}(i,j)$ die Kategorien des Teilworts $a_{i+1} \cdots a_j$ für $j-i=d$ an.

Dann erhalten wir durch Matrixmultiplikation über $\mathcal{R}^G$:

$$T^{(d+1)} := \sum_{e=1}^{d} T^{(e)} \cdot^P T^{(d+1-e)},$$

wie man am Ausfüllen der Felder sehen kann: zum Ausfüllen eines Felds (i,j) der Oberdiagonale $d+1$ (also für $j-i=d+1$) muß man Felder der Oberdiagonale e mit solchen der Oberdiagonale $d+1-e$ kombinieren, z.B.

$$T^{(3)}(0,3) = T^{(1)}(0,1) \cdot^P T^{(2)}(1,3) \ \cup\ T^{(2)}(0,2) \cdot^P T^{(1)}(2,3).$$

Definition 5.2 Definiere für jede Matrix D über $\mathcal{R}^G$ und $0 < d \in \mathbb{N}$ die Matrizen $D^{(d)}$ und D^+ durch

$$D^{(1)} := D, \qquad D^{(d+1)} := \sum_{e=1}^{d} D^{(e)} \cdot D^{(d+1-e)}, \qquad D^+ := \sum_{d=1}^{\infty} D^{(d)}$$

Die Matrix D heißt *transitiv*, falls $D = D^+$. (Hier ist $+ = \cup$ und $\cdot = \cdot^P$ gemeint.)

Beachte, daß D^+ stets existiert, da die Einträge im Feld (i, j) durch Vereinigungen nicht kleiner werden und durch N beschränkt sind.

Satz 5.3 Inititalisiert man D wie im CYK-Algorithmus und berechnet man dann D^+, so erhält man dieselbe Matrix T wie durch das Verfahren von CYK.

Beweis: Zeige, daß die d-te Oberdiagionale von T mit den nichtleeren Feldern von $D^{(d)}$ übereinstimmt. (Siehe Harrison[3], Theorem 12.5.1) $\square$

Es folgt, daß $D^+ = \sum_{d=1}^{n} D^{(d)}$ durch eine endliche Summe berechnet werden kann.

5.2 Reduktion auf Multiplikation Boole'scher Matrizen

Sei $\mathcal{R}^G$ der oben definierte „Fastring“ zur Grammatik $G = (N, \Sigma, S, P)$ und einfachheitshalber $N = \{A_1, \ldots, A_s\}$. Die Multiplikation von Matrizen über $\mathcal{R}^G$ kann wie folgt auf die Multiplikation von Matrizen über $\mathbb{B}$ reduziert werden:

(i) Schritt 1: Betrachte eine Menge $X \subseteq N$ als Bitvektor $x = (x_1, \ldots, x_s) \in \mathbb{B}^s$ mit Komponenten

$$x_i := \begin{cases} 1, & \text{falls } A_i \in X, \\ 0, & \text{sonst.} \end{cases}$$

Also etwa $\{A_1, A_2, A_4\} \mapsto (1, 1, 0, 1, 0, \ldots, 0)$.

(ii) Schritt 2: Betrachte eine Matrix B über $\mathcal{R}^G$ als Matrix der entsprechenden Bitvektoren, also als Matrix über $\mathbb{B}^s$, und spalte diese in s Komponenten $(B_1, \ldots, B_s)$, also Matrizen B_i über $\mathbb{B}$ auf, z.B.

$$\begin{pmatrix} \emptyset & \{A_1\} & \{A_1, A_2\} \\ \emptyset & \emptyset & \{A_2\} \\ \emptyset & \emptyset & \emptyset \end{pmatrix} \simeq \begin{pmatrix} (0,0) & (1,0) & (1,1) \\ (0,0) & (0,0) & (0,1) \\ (0,0) & (0,0) & (0,0) \end{pmatrix}$$
$$\mapsto \left(\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \right).$$

(iii) Schritt 3: Berechne das Produkt der $n \times n$-Matrizen B, C über $\mathcal{R}^G$ so:

(a) Zerlege die Matrizen $B \mapsto (B_1, \dots, B_s)$ und $C \mapsto (C_1, \dots, C_s)$ in ihre Boole'schen Komponentenmatrizen,

(b) Berechne die s^2 Matrizen $B_p \cdot C_q$ über $\mathbb{B}$, für alle $1 \leq p, q \leq n$,

(c) Setze $D(i,j) := \bigcup_{1 \leq p,q \leq n} \{ \{A_p\} \cdot^P \{A_q\} \mid (B_p \cdot C_q)(i,j) = 1 \}$.

Mit Schritt 3 wird die Multiplikation $B \cdot^P C$ auf s^2 viele Multiplikationen Boole'scher Matrizen gleicher Größe reduziert:

Satz 5.4 Das Verfahren in Schritt 3 berechnet aus B, C die Matrix $B \cdot^P C$.

Beweis: Die Verwendung von $\cdot^P$ in (c) ist wegen

$$\{A_p\} \cdot^P \{A_q\} = \{ A \in N \mid (A \to A_p A_q) \in P \}$$

nur ein Nachsehen in der Regelmenge P. Daher ergibt sich mit

$$\begin{aligned}
A \in D(i,j) \quad &\iff \quad \bigvee_{(A \to A_p A_q) \in P} (B_p \cdot C_q)(i,j) = 1 \\
&\iff \quad \bigvee_{(A \to A_p A_q) \in P} \sum_k B_p(i,k) \cdot C_q(k,j) = 1 \\
&\iff \quad \bigvee_{(A \to A_p A_q) \in P} \bigvee_k (A_p \in B(i,k) \wedge A_q \in C(k,j)) \\
&\iff \quad A \in \bigcup_k (B(i,k) \cdot^P C(k,j)) \\
&\iff \quad A \in (B \cdot^P C)(i,j)
\end{aligned}$$

die Behauptung. □

Sei $M_{\mathcal{R}}(n)$ der Aufwand zur Multiplikation zweier $n \times n$-Matrizen über dem Ring $\mathcal{R}$ oder dem „Fastring" $\mathcal{R} = \mathcal{R}^G$. Aus Schritt 3 erhält man die Abschätzung

$$O(M_{\mathcal{R}^G}(n)) \leq c \cdot O(M_{\mathbb{B}}(n))$$

für eine Konstante c, wenn man G festhält, d.h. den Aufwand nur in der Größe $n = |w|$ des Eingabeworts mißt.

Beachte, daß in c der Faktor s^2 aus (b) und das Durchsuchen der Regelmenge P steckt, was nochmal $|P| \leq s^2$ Schritte pro Paar (p, q) von Positionen in w erfordert. Da die Multiplikation $B_p \cdot C_q$ mindestens s^2 Schritte erfordert, bedeutet das insgesamt einen Faktor $c = s^2$. Die Anzahl s der Nonterminale ist aber bei Grammatiken natürlicher Sprachen viel größer als die Satzlänge n.

5.3 Berechnung der transitiven Hülle D^+

Nun geht es um eine effiziente Berechnung der transitiven Hülle D^+ einer oberen Dreiecksmatrix D über $\mathcal{R}^G$.[5] Die Berechnung von D^+ für große Dreiecksmatrizen wird auf dasselbe Problem für kleinere reduziert.

An der Berechnung von D^+ für Dreiecksmatrizen D sieht man, daß man für das Feld $D^+(i,j)$ nur die Felder $(i,i),\ldots,(i,j-1),\ldots,(j-1,j-1)$ braucht.

Eine *zentrale Untermatrix* der $n \times n$-Matrix B ist eine $m \times m$-Matrix der Form $A = (a_{i,j}) = (b_{r+i,r+j})_{i,j<m}$ für ein r mit $r + m \leq n$.

Lemma 5.5 Sei B eine oberere Dreiecksmatrix. Ist A eine zentrale Untermatrix von B, so ist $A^{(d)}$ bzw. A^+ die entsprechende zentrale Untermatrix von $B^{(d)}$ bzw. B^+.

5.3.1 Reduktion auf Dimension 2^k

Ist D mit Dimension $n \times n$ gegeben, wo $2^{k-1} < n < 2^k$ ist, also $k = \lceil \log n \rceil$, so füllt man D durch 0-en auf die Dimension $2^k \times 2^k$ auf zu

$$\begin{pmatrix} D & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & 0 \end{pmatrix}.$$

Dies ist eine obere Dreiecksmatrix $\tilde{D}$ der Dimension 2^k, auf die man das Verfahren anwendet. Dabei erhält man

$$(\tilde{D})^+ = \begin{pmatrix} D^+ & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & 0 \end{pmatrix}.$$

Im folgenden sei $M(n)$ der Aufwand zur Berechnung des Produkts zweier oberer Dreiecksmatrizen der Dimension $n \times n$.

Vergrößert man obere Dreiecksmatrizen der Dimension $n \times n$ wie angedeutet auf solche der Dimension $2^{\lceil \log n \rceil} \times 2^{\lceil \log n \rceil}$, so kann deren Multiplikation durch die für die Ausgangsmatrizen abgeschätzt werden:

$$M(2^{\lceil \log n \rceil}) \leq c \cdot M(n)$$

Den Zeitaufwand zur Berechnung von D^+ für obere Dreiecksmatrizen D brauchen wir daher nur für $n = 2^k$ mit Hilfe von $M(n)$ abzuschätzen.

[5] Unter *oberer Dreiecksmatrix* wird immer eine Dreicksmatrix verstanden, deren Einträge in den Feldern (i,j) mit $j \leq i$, also einschließlich der Hauptdiagonalen, mit 0 besetzt sind.

5.3.2 Rekursion über die Dimension

Man definiert für Teiler k von n Operationen P_k, die die transitive Hülle B^+ einer oberen Dreiecksmatrix B der Dimension $n \times n$ unter der Voraussetzung berechnen, daß die Matrizen B_1, B_2 der Dimension $n - n/k$ in

$$B = \begin{pmatrix} B_1 & \cdots \\ \vdots & \ddots \end{pmatrix} = \begin{pmatrix} \ddots & \vdots \\ \cdots & B_2 \end{pmatrix}$$

schon transitiv sind, d.h. $B_i = B_i^+$ gilt.

Lemma 5.6 Angenommen, man hat schon eine Algorithmus P_2 mit der genannten Eigenschaft. Definiere eine Operation $\mathit{trans}(A)$ für obere Dreiecksmatrizen A der Dimension $n \times n$, wo n eine Potenz von 2 ist, wie folgt:

$$\mathit{trans}(A) = \begin{cases} A, & \text{falls } n = 2 \\ P_2 \begin{pmatrix} \mathit{trans}(A_1) & A_3 \\ 0 & \mathit{trans}(A_2) \end{pmatrix}, & \text{sonst,} \end{cases}$$

wobei $A = \begin{pmatrix} A_1 & A_3 \\ 0 & A_2 \end{pmatrix}$ für Matrizen A_i der Dimension $n/2 \times n/2$. Dann gilt:

(i) $\mathit{trans}(A) = A^+$, und

(ii) Ist $T_2(n)$ die Zeit, $P_2(B)$ für $n \times n$-Matrizen B zu berechnen, und $\mathit{TR}(n)$ die Zeit, $\mathit{trans}(A)$ für $n \times n$-Matrizen A zu berechnen, so ist

$$\mathit{TR}(n) = 2 \cdot \mathit{TR}(n/2) + T_2(n) + O(n^2).$$

Die Definition von P_2 benutzt P_3 und P_4 und wird unten angegeben. Um die Korrektheit von P_3 und P_4 zu zeigen, braucht man folgendes Lemma von Valiant.

Sei C eine $n \times n$-Matrix und $1 \leq n/2 \leq r \leq n$. Zerlege C gemäß

$$C = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}$$

in $r \times r$-Matrizen

$$B_1 := \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 4 & 5 \\ \hline \end{array} \quad \text{und} \quad B_2 := \begin{array}{|c|c|} \hline 5 & 6 \\ \hline 8 & 9 \\ \hline \end{array}.$$

Definiere $C^{+(r)}$ durch

$$C^{+(r)} := \begin{array}{|c|c|c|} \hline E_1 & 2 & E_3 \\ \hline 4 & 5 & 6 \\ \hline E_7 & 8 & E_9 \\ \hline \end{array} \quad \text{mit} \quad \begin{array}{|c|c|} \hline 1 & 3 \\ \hline 7 & 9 \\ \hline \end{array}^{+} = \begin{array}{|c|c|} \hline E_1 & E_2 \\ \hline E_3 & E_4 \\ \hline \end{array}.$$

Lemma 5.7 (Valiant) Sei B eine obere $n \times n$-Dreicksmatrix und $1 \leq n/2 \leq r \leq n$. Zerlege B wie oben C in 9 Teilmatrizen. Angenommen, B_1 und B_2 sind transitiv, und

$$E_r := \begin{array}{|c|c|c|}\hline 0 & 2 & 0 \\ \hline 0 & 0 & 6 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

Dann ist $B^+ = (B + E_r^2)^{+(r)}$, oder anschaulich

$$B^+ = \left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 0 & 0 & 2*6 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \right)^{+(r)} = \begin{array}{|c|c|c|}\hline 1 & 2 & 3+(2*6) \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}^{+(r)} .$$

Der Beweis des Lemmas ist sehr technisch.

Lemma 5.8 Berechne $P_2(B)$ für Matrizen

$$B = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array}$$

mit Teilmatrizen i der Dimension $n/4 \times n/4$ und unter der Voraussetzung, daß

$$B_1 = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 5 & 6 \\ \hline \end{array} = B_1^+ \quad \text{und} \quad B_2 = \begin{array}{|c|c|}\hline 11 & 12 \\ \hline 15 & 16 \\ \hline \end{array} = B_2^+$$

schon transitiv sind, wie folgt:

(i) Falls $n = 2$ ist, ist $B^+ = B$.

(ii) Für $n > 2$ berechne rekursiv

$$\begin{aligned}
P_2\left(\begin{array}{|c|c|}\hline 6 & 7 \\ \hline 10 & 11 \\ \hline \end{array}\right) &= \begin{array}{|c|c|}\hline 6 & 7' \\ \hline 10 & 11 \\ \hline \end{array}, \\
P_3\left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 5 & 6 & 7' \\ \hline 9 & 10 & 11 \\ \hline \end{array}\right) &= \begin{array}{|c|c|c|}\hline 1 & 2 & 3' \\ \hline 5 & 6 & 7' \\ \hline 9 & 10 & 11 \\ \hline \end{array}, \\
P_3\left(\begin{array}{|c|c|c|}\hline 6 & 7' & 8 \\ \hline 10 & 11 & 12 \\ \hline 14 & 15 & 16 \\ \hline \end{array}\right) &= \begin{array}{|c|c|c|}\hline 6 & 7' & 8' \\ \hline 10 & 11 & 12 \\ \hline 14 & 15 & 16 \\ \hline \end{array}, \\
P_4\left(\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3' & 4 \\ \hline 5 & 6 & 7' & 8' \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array}\right) &= \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3' & 4' \\ \hline 5 & 6 & 7' & 8' \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array}) \; = \; B^+
\end{aligned}$$

Der Zeitaufwand $T_2(n)$ zur Berechnung von $P_2(B)$ für $n \times n$-Matrizen B ist

$$T_2(n) = T_2(n/2) + 2 \cdot T_3(3n/4) + T_4(n).$$

Lemma 5.9 Berechne $P_3(B)$ für obere Dreiecksmatrizen B der Dimension $n \times n$ wie folgt, unter der Voraussetzung daß bei der Zerlegung

$$B = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}$$

in 9 quadratische Untermatrizen der Dimension $n/3 \times n/3$ die Untermatrizen

$$B_1 = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 4 & 5 \\ \hline \end{array} = B_1^+ \quad \text{und} \quad B_2 = \begin{array}{|c|c|}\hline 5 & 6 \\ \hline 8 & 9 \\ \hline \end{array} = B_2^+$$

schon transitiv sind:

(i) Berechne

$$C = B + E_r^2 = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 0 & 2 & 0 \\ \hline 0 & 0 & 6 \\ \hline 0 & 0 & 0 \\ \hline \end{array}^2 = \begin{array}{|c|c|c|}\hline 1 & 2 & 3' \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array},$$

(ii) Für $r = 2n/3$, berechne

$$C^{+(r)} = P_2\left(\begin{array}{|c|c|}\hline 1 & 3' \\ \hline 7 & 9 \\ \hline \end{array}\right).$$

Der Zeitaufwand $T_3(n)$ zur Berechnung von $P_3(B)$ für $n \times n$-Matrizen B ist

$$T_3(n) = M(n) + T_2(2n/3) + O(n^2).$$

Lemma 5.10 Berechne $P_4(B)$ für obere Dreiecksmatrizen B der Dimension $n \times n$ unter der Voraussetzung, daß bei der Zerlegung

$$B = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array}$$

in quadratische Untermatrizen der Dimension $n/4 \times n/4$ die Teilmatrizen

$$B_1 = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 5 & 6 & 7 \\ \hline 9 & 10 & 11 \\ \hline \end{array} = B_1^+ \quad \text{und} \quad B_2 = \begin{array}{|c|c|c|}\hline 6 & 7 & 8 \\ \hline 10 & 11 & 12 \\ \hline 14 & 15 & 16 \\ \hline \end{array} = B_2^+$$

schon transitiv sind, wie folgt:

(i) Berechne für $r = 3n/4$

$$C = B + E_r^2 = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array} + \begin{array}{|c|c|c|c|}\hline 0 & 2 & 3 & 0 \\ \hline 0 & 0 & 0 & 8 \\ \hline 0 & 0 & 0 & 12 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}^{\,2} = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4' \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array},$$

(ii) berechene für $r = 3n/4$ die Matrix

$$C^{+(r)} = P_2\left(\begin{array}{|c|c|}\hline 1 & 4' \\ \hline 13 & 16 \\ \hline \end{array}\right).$$

Der Zeitaufwand $T_4(n)$ zur Berechnung von $P_4(B)$ ist

$$T_4(n) = M(n) + T_2(n/2) + O(n^2).$$

Dann muß man noch die Komplexitäten zusammenrechnen:

Lemma 5.11 Falls es ein $\gamma \geq 2$ mit $M(2^{m+1}) \geq 2^{\gamma} M(2^m)$ für alle m gibt, gilt

(i) für $n = 2^k$ mit $k \geq 1$ und gewisse Konstanten c_1, c_2:

$$T_2(n) = \begin{cases} c_1 M(n), & \text{falls } \gamma > 2, \\ c_2 M(n) \log n, & \text{falls } \gamma = 2. \end{cases}$$

(ii) die Zeit $TR(n)$ zur Berechnung der transitiven Hülle einer Matrix der Dimension $n \times n$ ist

$$TR(n) \leq \begin{cases} O(M(n)), & \text{falls } \gamma > 2, \\ O(M(n) \log n & \text{falls } \gamma = 2. \end{cases}$$

Satz 5.12 Sei D eine obere Dreiecksmatrix über $\mathcal{R}^G$. Die Zeit zur Berechnung der transitiven Hülle D^+ von D ist

$$TR(n) \leq \begin{cases} cM_{\mathbb{B}}(n) \leq cn^{2.81}, & \text{falls } M_{\mathbb{B}}(n) \geq n^{2+\epsilon} \text{ für ein } \epsilon > 0, \\ cn^2 \log n, & \text{falls } M_{\mathbb{B}}(n) < n^{2+\epsilon} \text{ für jedes } \epsilon > 0. \end{cases}$$

Mit Satz 5.3 übertragen sich diese Schranken auf das Erkennen von Wörtern der Länge n mit kontextfreien Grammatiken in Chomsky-Normalform.

Für die Beweise siehe Harrison[3], S. 442–470.

6 Der Algorithmus von Rytter

Auch das Verfahren von Rytter reduziert das Erkennungsproblem für kontextfreie Grammatiken in Chmosky-Normalform auf die Boole'sche Matrixmultiplikation. Der Vorteil des Verfahrens von Rytter gegenüber dem von Valiant besteht darin, daß Rytter den Ring der 2-stelligen Relationen zwischen Nonterminalen von G benutzt, während Valiant einen nicht-assoziativen „Fastring" $\mathcal{R}^G$ (von Syntaxbäumen) verwendet. Das vereinfacht einen Korrektheitsbeweis erheblich (wenn man von den Fehlern der Originalarbeit absieht.) Der Zeitaufwand beider Verfahren ist gleich.

Wir definieren zur CNF-Grammatik $G = (N, T, P, S)$ und der Eingabe $w = a_1 \cdots a_n$ die Menge der *gültigen Kanten* (A, i, j) durch

$$valid(i,j) := \{ A \mid A \Rightarrow_G{}^* a_i \cdots a_j \} \in \mathcal{P}_N. \tag{14}$$

Statt $A \in valid(i,j)$ schreiben wir oft $A(i,j)$. Offenbar ist $valid(i,j) = \emptyset$ für $i \not\leq j$. Um die Eingabe zu erkennen, muß geprüft werden, ob $S \Rightarrow_G{}^* a_1 \cdots a_n$ bzw. ob

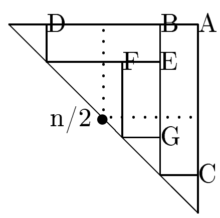
$$S \in valid(1,n).$$

Dazu soll für o.B.d.A. eine Zweierpotenz n die Berechnung der Matrix

$$VALID(1,n) = (valid(i,j))_{1 \leq i,j \leq n} \in Mat_{n \times n}(\mathcal{P}_N)$$

auf die Berechnung von $VALID(1, n/2)$ und $VALID(n/2+1, n)$ zurückgeführt werden. Eine wesentliche Beobachtung von Rytter ist folgende:

> Ein Syntaxbaum t zu $A \in valid(1,n)$ hat einen Pfad $\pi = \ldots FEBA$ vom Blatt $n/2$ zur Wurzel, so daß für jeden Teilbaum von t mit Wurzel auf π der bekannte Teil $VALID(1,n/2) \cup VALID(n/2+1,n)$ einen seiner direkten Teilbäume bzw. dessen gültige Kante enthält.



Daraus ergeben sich *bedingte Regeln zum Füllen der Felder* $valid(i,j)$, für jede Grammatikregel $(A \to BC) \in P$:

$$\frac{C \in valid(j+1, j')}{B \in valid(i,j) \Rightarrow A \in valid(i,j')} \qquad \frac{B \in valid(i,j)}{C \in valid(j+1,j') \Rightarrow A \in valid(i,j')}$$

Gibt es schon $C \in \mathit{valid}(j+1, j')$, so führen spätere Einträge $B \in \mathit{valid}(i, j)$ zu Einträgen $A \in \mathit{valid}(i, j')$ mit $j < j'$; gibt es schon $B \in \mathit{valid}(i, j)$, so führen spätere Einträge $C \in \mathit{valid}(j+1, j')$ zu Einträgen $A \in \mathit{valid}(i, j')$ mit $i < j+1$.

Jede Grammatikregel liefert also Regeln zum Auffüllen der Matrix *valid* entlang horizontaler Kanten $(i, j) \to (i, j')$ mit $j < j'$ oder vertikaler Kanten $(i', j') \uparrow (i, j')$ mit $i < i'$. Diese Regeln besagen, welche Kategorien B bzw. C im *valid*-Feld am Ausgangspunkt der Kante zu welchen Kategorien A am Zielpunkt der Kante führen; man kann die Kanten markieren: horizontale Kanten

$$(i, j) \xrightarrow{B\backslash A} (i, j') \quad \text{mit } j < j'$$

für „wenn B zu $\mathit{valid}(i, j)$ kommt, so kommt A zu $\mathit{valid}(i, j')$“, vertikale Kanten

$$(i', j') \xrightarrow{A/C} (i, j') \quad \text{mit } i < i'$$

für „wenn C zu $\mathit{valid}(i', j')$ kommt, so kommt A zu $\mathit{valid}(i, j')$“. Die Menge der sich so ergebenden Markierungen $B\backslash A$ bzw. A/C derselben Kante bildet das *Gewicht* der Kante, eine Relation $R \subseteq N \times N$, die von den schon aufgefüllten Feldern $\Pi = \mathit{VALID}(1, n/2) \cup \mathit{VALID}(n/2+1, n)$ von *valid* abhängt.

Diesen gewichteten Graphen nennt Rytter einen *Verbandsgraphen*. Aus der Beobachtung von Rytter folgt, daß die noch nicht gefüllten Felder $\mathit{valid}(i, j)$ von $\mathit{VALID}(1, n)$ die Kategorien enthalten, die man durch Anwenden der Gewichte der Wege von $(n/2, n/2)$ nach (i, j) im Verbandsgraphen auf die Kategorien im Feld $(n/2, n/2)$ erhält.

6.1 Der gewichtete Verbandsgraph $L(\Pi)$

Sei $\mathcal{R}_N = (\mathcal{P}_{N\times N}, \cup, \emptyset, ;, 1_N)$ der Ring der 2-stelligen Relationen über N.

Definition 6.1 Sei $n_2 - n_1$ durch 2 teilbar und $n_3 := n_1 + (n_2 - n_1)/2$. Der *Verbandsgraph* zu

$$\Pi = \mathit{VALID}(n_1, n_3) \cup \mathit{VALID}(n_3+1, n_2)$$

ist der Graph (L, Γ, q) mit Quelle $q \in L$ und Gewicht $\Gamma : L \times L \to \mathcal{R}_N$, so daß

$$\begin{aligned} L &:= \{ (i, j) \mid n_1 \leq i, j \leq n_2 \}, \\ q &:= (n_3, n_3), \\ \Gamma((i, j), (i', j')) &:= \{ (A, A') \mid (A, i, j) \Rightarrow (A', i', j') \}, \end{aligned}$$

wobei $\Rightarrow$ wie folgt relativ zu Π (und w und G) definiert wird:

$$\begin{aligned} (B, i, j) \Rightarrow (A, i', j') \quad &:\Longleftrightarrow \quad i = i' \text{ und } (C, j+1, j') \in \Pi \text{ für ein } (A \to BC) \in P, \\ (C, i, j) \Rightarrow (A, i', j') \quad &:\Longleftrightarrow \quad j = j' \text{ und } (B, i', i-1) \in \Pi \text{ für ein } (A \to BC) \in P \end{aligned}$$

Da jeweils eine Komponente geich bleibt, kann man die Gewichte kompakt in zwei Matrizen H und V der Dimension $(n_2 - n1) \times (n_2 - n_1)$ darstellen, wobei

$$\begin{aligned} H(j,j') &:= \{\, Y\backslash X \mid Y(i,j)\backslash X(i,j') \,\} = \{\, (Y,X) \mid Y(i,j) \Rightarrow X(i,j') \,\}, \\ V(i,i') &:= \{\, X/Z \mid X(i',j)/Z(i,j) \,\} = \{\, (Z,X) \mid Z(i,j) \Rightarrow X(i',j) \,\}. \end{aligned}$$

Mit $L(\Pi)$ ist diese Darstellung (H,V,q) des Verbandsgraphen (L,Γ,q) gemeint.

Das Gewicht $\Gamma(\pi)$ eines Weges

$$\pi = \ (i_1,j_1) \xrightarrow{R_1} (i_2,j_2) \xrightarrow{R_2} (j_3,j_3) \xrightarrow{R_3} \ \ldots \ \xrightarrow{R_k} (i_{k+1},j_{k+1})$$

ist das Produkt der Relationen R_i, die Relation $\Gamma(\pi) = R_1\,;\,R_2\,;\,\ldots\,;\,R_k$. Der ***Abstand von*** (i,j) ***nach*** (k,l) in (L,Γ) ist wegen $+^{\mathcal{R}_N} = \cup$ die Vereinigung der beteiligten Relationen $\Gamma(\pi)$,

$$\mathit{dist}((i,j),(k,l)) = \bigcup \{\, \Gamma(\pi) \mid \pi \in \mathit{Wege}((i,j),(k,l)) \,\},$$

und wird i.a. von keinem Weg angenommen. Wie man die Abstände

$$\mathit{DIST}(x) := \mathit{dist}(q,x) = \bigcup \{\, \Gamma(\pi) \mid \pi \in \mathit{Wege}(q,x),\ \pi \neq \epsilon \,\},$$

von der Quelle q zu allen $x \in L(\Pi)$ berechnet, wird in Abschnitt 6.3 gezeigt.

6.2 Berechnung der Parsematrix *VALID*

Zur gegebenen CNF-Grammatik $G = (N,T,P,S)$ und der Eingabe $w = a_1 \cdots a_n$ wird die Parsematrix durch $\mathit{VALID}(1,n)$ rekursiv über n erstellt, wobei o.B.d.A. die Länge n eine Potenz von 2 ist.

$\mathit{VALID}(i,j)$ berechnet $\mathit{valid}(k,l)$ für $i \le k \le l \le j$ wie folgt, mit $m := j - i$:

(i) Fall 1, $m = 0$: Dann ist $j = i$, und die lexikalischen Analyse ergibt

$$\mathit{valid}(i,i) = \{\, A \mid (A \rightarrow a_i) \in P \,\} \quad \text{bzw.} \quad \mathit{valid}(i,i) = \{a_i\},$$

je nachdem, ob man lexikalische Regeln erlaubt oder nicht.

(ii) Fall 2, $m = 1$, also $j = i + 1$:

   (a) berechne rekursiv $\mathit{VALID}(i,i)$ und $\mathit{VALID}(j,j)$;
   (b) berechne aus $\Pi := \mathit{VALID}(i,i) \cup \mathit{VALID}(j,j)$ den Verbandsgraphen $L(\Pi) = (H,V,q)$;
   (c) setze $\mathit{valid}(i,j) := H(i,j)(\mathit{valid}(i,i)) + V(j,i)(\mathit{valid}(j,j))$.

(iii) Fall 3, $m > 1$, und O.B.d.A. $m = 2^{k+1}$:

(a) berechne rekursiv $\mathit{VALID}(i, i+m/2)$ und $\mathit{VALID}(i+m/2+1, j)$, also $\mathit{valid}(k,l)$ mit $i \le k \le l \le i+m/2$ und $i+m/2+1 \le k \le l \le j$;

(b) konstruiere aus $\Pi := \mathit{VALID}(i, i+m/2) \cup \mathit{VALID}(i+m/2+1, j)$ den gewichteten Graphen $L(\Pi) = (H, V, q)$;

(c) berechne in $L(\Pi)$ die Abstände $\mathit{DIST}(k,l) \in \mathcal{R}_N$ und setze

$$\mathit{valid}(k,l) := \mathit{DIST}(k,l)(\mathit{valid}(q))$$

für alle Punkte (k,l) mit $i \le k \le i+m/2$ und $i+m/2+1 \le l \le j$.

Das bedeutet in der Schreibweise gültiger Kanten:

$$\mathit{VALID}(i,j) = \Pi \cup \{\, (A,k,l) \mid (X, i+m/2, i+m/2) \Rightarrow^* (A,k,l) \,\}.$$

Die Fälle 1 und 2 sind bei Rytter nicht ausgeführt. Der Zeitaufwand $T(n)$ zur Berechnung von $\mathit{VALID}(1,n)$ erfüllt die Rekursionsbedingung

$$T(n) = 2 * T(n/2) + O(T_1(n)) + O(T_2(n)), \tag{15}$$

wobei $T_1(n)$ der Zeitaufwand zur Berechnung von DIST und $T_2(n)$ der zur Berechnung von $L(\Pi)$ ist. An Definition 6.1 kann man leicht ablesen, daß $T_2(n) = O(n^2)$, wenn man die Grammatik fest läßt. Den Aufwand $T_1(n)$ schätzen wir im folgenden Abschnitt durch

$$T_1(n) = 4 * T_1(n) + O(M_{\mathbb{B}}(n))$$

ab. Unter der Annahme $M_{\mathbb{B}}(n) = O(n^{2+\epsilon})$ folgt $T_1(n) = O(n^{2+\epsilon})$, und dann für den Zeitaufwand der Berechnung der Parsematrix nach (15) analog

$$\begin{aligned} T(n) &= 2 * T(n/2) + O(T_1(n)) + O(T_2(n)), \\ &= 2 * T(n/2) + O(n^{2+\epsilon}) + O(n^2), \\ &= 2 * T(n/2) + O(n^{2+\epsilon}) \\ &= O(n^{2+\epsilon}) \\ &= O(M_{\mathbb{B}}(n)). \end{aligned}$$

6.3 Berechnung der Abstände in einem Verbandsgraphen

Wir beschreiben die Berechnung von Abständen in Verbandsgraphen unabhängig von ihrer Verwendung bei der Berechnung der Parsematrix. Dabei ist es einfacher, wenn die vertikalen Kanten „nach unten“ gehen, da man dann an einer Stelle direkt die Matrixmultiplikation verwenden kann.

Lemma 6.2 Sei $\mathcal{G} = (G, \Gamma, q)$ ein gewichteter Graph mit Größe $|G| = n^2$,

$$G = \{\, (i,j) \mid 1 \le i, j \le n \,\},$$

einem Gewicht $\Gamma : G \times G \to \mathcal{R}$ und einem Quellpunkt $q \in G$ so daß

(i) die Kantenrelation

$$E = \{ (x,y) \in G \times G \mid \Gamma(x,y) \neq 0^{\mathcal{R}} \},$$

nur aus horizontalen Kanten $\rightarrow$ und vertikalen Kanten $\downarrow$ besteht, d.h.

$$((i,j),(i',j')) \in E \implies (i = i' \wedge j < j') \vee (i < i' \wedge j = j'),$$

(ii) und es $H, V \in Mat_{n\times n}(\mathcal{R})$ gibt mit

$$\begin{aligned} \Gamma((i_1,j),(i_1,j+k)) = \Gamma((i_2,j),(i_2,j+k)) &= H(j,j+k),\\ \Gamma((i,j_1),(i+k,j_1)) = \Gamma((i,j_2),(i+k,j_2)) &= V(i,i+k), \end{aligned}$$

für alle $1 \leq i_1, i_2, j_1, j_2 \leq n$, $2 \leq i+k, j+k \leq n$, d.h. gleich lange, in derselben Spalte beginnende horizontale Kanten sowie gleich lange, in derselben Zeile beginnende vertikale Kanten haben dasselbe Gewicht.

Unter der Annahme, die Zeit zur Multiplikation zweier $n \times n$-Matrizen über dem Ring $\mathcal{R}(N)$ sei $M_{\mathcal{R}(N)}(n) = O(n^{2+\epsilon})$, kann

$$DIST(p) = \sum \{ \Gamma(\pi) \mid \pi \in Wege(q,p) \} \quad \text{für alle } p \in G$$

durch *ShortestPaths*(G) in der Zeit $O(M_{\mathcal{R}(N)}(n))$ berechnet werden.

Der Algorithmus *ShortestPaths*(X) berechnet für ein Teilquadrat X der Größe $n \times n$ von G die Abstände $DIST(p)$ für alle $p \in X$, wie folgt:

- für $n > 1$ setze $m = n/2$;
  (i) spalte X in vier Quadranten der Dimension $m \times m$ auf:

  $$X = \begin{array}{|c|c|} \hline A & D \\ \hline B & C \\ \hline \end{array};$$

  (ii) berechne *ShortestPaths*(A), d.h. berechne $DIST(x)$ für alle $x \in A$;
  (iii) berechne *ShortestPaths*(B) und *ShortestPaths*(D);
  (iv) berechne *ShortestPaths*(C);
- für $n = 1$ berechne

  $$DIST(x) := \Gamma(q,x) + \sum_{x' \in X',\, (x',x) \in E} DIST(x') \cdot \Gamma(x',x).$$

  Da E nur horizontale und vertikale Kanten hat, kann diese Operation durch Multiplikation von Matrizen berechnet werden:

  $$DIST := \Gamma + DIST_v + DIST_h = \Gamma + (DIST^t \cdot V)^t + DIST \cdot H.$$

Dabei sei X' der Bereich von G oberhalb und links von X und $DIST(x')$ für alle $x' \in X'$ schon bekannt.

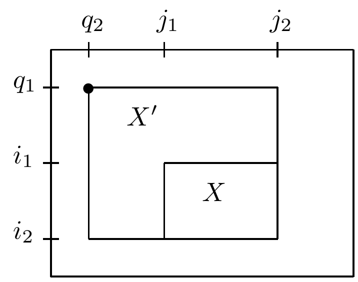
Beweis: Da es nur Kanten $\rightarrow$ und $\downarrow$ gibt, gilt $DIST(i', j') = 0$ für $j' < q_2$ oder $i' < q_1$ und für $i' = q_1 \wedge j' \leq d_j$ oder $i' \leq q_1 \wedge j' = q_2$. Sei

$$X = \{ (i, j) \mid i_1 \leq i \leq i_2, j_1 \leq j \leq j_2 \} \subseteq G$$

ein Rechteck, und

$$X' = \{ (i', j') \mid q_1 \leq i' < i_1 \wedge q_2 \leq j' \leq j_2) \vee (q_1 \leq i' \leq i_2 \wedge q_2 \leq j' < j_1) \}.$$

Um den Abstand $DIST(p)$ von $q = (q_1, q_2)$ nach $p = (i, j) \in X$ zu berechnen, nehmen wir an, daß $DIST(p')$ für $p' = (i', j') \in X'$ schon berechnet ist:



Das ist für das Rechteck $X = G$ mit $(i_1, j_1) = (0, 0)$ und $(i_2, j_2) = (n, n)$ erfüllt. Jeder Weg von q nach $p \in X$ ist entweder eine Kante mit $\Gamma(q, p) \neq 0$, oder hat ein Anfangsstück mit Punkten in X' und ein Endstück mit Punkten in X.

Man berechnet $DIST(p)$ für alle $p \in X$ durch Rekursion über Teilrechtecke von

$$X = \begin{array}{|c|c|} \hline A & D \\ \hline B & C \\ \hline \end{array}.$$

Da A' eine Teilmenge von X' ist, kann man rekursiv schon $DIST(p)$ für alle $p \in A$ berechnen. Danach ist die Induktionsvoraussetzung für $B', D' \subseteq X' \cup A$ erfüllt, so daß man $DIST(p)$ für alle $p \in B \cup D$ berechnen kann. Dadurch wird die Voraussetzung für $C' \subseteq X' \cup A \cup B \cup D$ erfüllt, sodaß $DIST(p)$ für alle $p \in C$ berechnet werden kann. Es genügt daher, den Fall $X = \{p\}$ zu behandeln.

Wir müssen $DIST(q_1, j)$ für $q_2 < j \leq j_2$ aus den horizontalen und $DIST(i, q_2)$ für $q_1 < i \leq i_2$ aus den vertikalen Kantengewichten aus H bzw. V berechnen. Man beginnt für $p = (i, j) \in X$ mit

$$DIST'(i, j) := \Gamma((q_1, q_2), (i, j)) = \begin{cases} H(q_2, j) & \text{falls } q_1 = i \wedge q_2 < j \\ V(q_1, i) & \text{falls } q_1 < i \wedge q_2 = j \\ 0^R & \text{sonst} \end{cases}$$

Da $V(i',i) = 0$ für $i' \geq i$ und $H(j',j) = 0$ für $j' \geq j$ ist, erhält man

$$\begin{aligned}
DIST(p) &:= DIST'(p) + \sum_{p' \in X', (p',p) \in E} DIST(p') \cdot \Gamma(p',p) \\
&= DIST'(i,j) + \sum_{((i',j'),(i,j)) \in E} DIST((i',j')) \cdot \Gamma((i',j'),(i,j)) \\
&= DIST'(i,j) + \sum_{i'<i} DIST(i',j) \cdot \Gamma((i',j),(i,j)) \\
&\qquad + \sum_{j'<j} DIST(i,j') \cdot \Gamma((i,j'),(i,j)) \\
&= DIST'(i,j) + \sum_{i'<i} DIST(i',j) \cdot V(i',i) + \sum_{j'<j} DIST(i,j') \cdot H(j',j) \\
&= DIST'(i,j) + \sum_{i'} DIST(i',j) \cdot V(i',i) + \sum_{j'} DIST(i,j') \cdot H(j',j) \\
&= DIST'(i,j) + (DIST^t \cdot V)(j,i) + (DIST \cdot H)(i,j) \\
&= DIST'(p) + (DIST^t \cdot V)^t(p) + (DIST \cdot H)(p)
\end{aligned}$$

(Diese Berechnung erfolgt im Programm durch `singleEdgeExtend`.)

Sei $T_1(n)$ der Zeitaufwand zur Berechnung von `ShortestPaths(G)` für $|G| = n^2$, wenn G durch (H,V,q), also durch zwei $n \times n$-Matrizen über $\mathcal{R}(N)$, gegeben ist. Da `ShortestPaths` sich rekursiv für die vier quadratischen Teile von G aufruft und bei Dimension 1 nach

$$DIST := \Gamma + DIST_v + DIST_h = \Gamma + (DIST^t \cdot V)^t + DIST \cdot H$$

zwei Additionen und Multiplikationen von $n \times n$-Matrizen über $\mathcal{R}(N)$ braucht, ergibt sich für T_1 die Rekursionsbedingung

$$T_1(n) = 4 * T_1(n/2), \qquad T_1(1) = O(M_{\mathcal{R}(N)}(n)).$$

Die Multiplikation in $\mathcal{R}(N)$ kann leicht auf die Multiplikation zweier $s \times s$-Matrizen über $\mathbb{B}$ reduziert werden, weshalb $M_{\mathcal{R}(N)} \leq c \cdot O(M_{\mathbb{B}}(n))$. $\square$

6.4 Ein Beispiel

Die Originalarbeit von W. Rytter [5] enthält kein Beispiel. Um die nachfolgende Implementierung überprüfen zu können, berechnen wir von Hand die Auffüllung der Parsematrix an einem Beispiel von E.Torres [6]. Nachträglich wurde die Berechnung der Abstände im Verbandsgraphen etwas geändert, so daß kleine Abweichungen von dem Ergebnis des Programms möglich sind.

Beispiel 6.3 Eingabe $w = BCDDCBBE$ bei der Grammatik

$S \to AB$
$A \to CD$
$B \to EF$
$C \to BA$
$C \to AE$
$E \to CB$
$E \to BC$
$F \to BE$

	1	2	3	4	5	6	7	8
1	B	E	C	A	?	?	?	?
2		C	A	–	?	?	?	?
3			D	–	?	?	?	?
4				D	?	?	?	?
5					C	E	–	B
6						B	–	–
7							B	F
8								E

Diese Tabelle $\Pi = VALID(1, 8/2) \cup VALID(8/2, 8)$ ergibt folgende Relation $\Rightarrow$,

$$\begin{aligned}
\{ (D, A) \mid D \Rightarrow A \} &= \{ (B, A) \mid B \backslash A \} \cup \{ (C, A) \mid A/C \} \\
&= \{ (B, A) \mid (A \to BC) \in P, C \in VALID(8/2, 8) \} \\
&\cup \ \{ (C, A) \mid (A \to BC) \in P, B \in VALID(1, 8/2) \}
\end{aligned}$$

die ich hier im Stil der Kategorialgrammatik in / und \ trenne:

H		V
$Y(i,j)\backslash X(i,j+k)$	$X \to YZ$	$X(i-k,j)/Z(i,j)$
$A(i,4)\backslash S(i,8)$	$S \to AB$	$S(1,j)/B(5,j)$
$A(i,5)\backslash S(i,6)$		$S(2,j)/B(4,j)$
$A(i,6)\backslash S(i,7)$		
	$A \to CD$	$A(1,j)/D(4,j)$
		$A(2,j)/D(3,j)$
$E(i,6)\backslash B(i,8)$	$B \to EF$	$B(1,j)/F(3,j)$
	$C \to BA$	$C(1,j)/A(2,j)$
$A(i,4)\backslash C(i,6)$	$C \to AE$	$C(1,j)/E(5,j)$
$A(i,7)\backslash C(i,8)$		$C(2,j)/E(4,j)$
$C(i,4)\backslash E(i,8)$	$E \to CB$	$E(1,j)/B(4,j)$
$C(i,5)\backslash E(i,6)$		$E(2,j)/B(3,j)$
$C(i,6)\backslash E(i,7)$		
$B(i,4)\backslash E(i,5)$	$E \to BC$	$E(1,j)/C(2,j)$
$B(i,4)\backslash F(i,6)$	$F \to BE$	$F(1,j)/E(2,j)$
$B(i,7)\backslash F(i,8)$		

Aus dieser Relation $\Rightarrow$ definiert man die horizontalen und vertikalen Gewichte.

Das ergibt die Matrizen H mit $H(j,k) = \{\, Y\backslash X \mid Y(i,j)\backslash X(i,k) \,\}$

H	1	2	3	4	5	6	7	8
1								
2								
3								
4					$B\backslash E$	$A\backslash C, B\backslash F$		$A\backslash S, C\backslash E$
5						$A\backslash S, C\backslash E$		
6							$A\backslash S, C\backslash E$	$E\backslash B$
7								$A\backslash C, B\backslash F$
8								

und V mit $V(i,k) = \{\, X/Z \mid X(k,j)/Z(i,j) \,\}$:

V	1	2	3	4	5	6	7	8
1								
2	$C/A, E/C, F/E$							
3	B/F	$A/D, E/B$						
4	$A/D, E/B$	$S/B, C/E$						
5	$A/B, C/E$							
6								
7								
8								

Rytter numeriert die Zeilen der Parsematrix in anderer Reihenfolge, damit man direkt auf die Matrixmultiplikation kommt, wenn man die Pfade entlangläuft.

Nun betrachten wir den Quellknoten $s = (4,4) \in L$. Es interessieren uns nur noch Wege zu den Punkten in $X = \{\, (i,j) \mid 1 \leq i \leq 4,\ 5 \leq j \leq 8 \,\}$. Für jedes $x \in X$ soll $DIST(x) \in \mathcal{R}_N$ berechnet werden. Wenn man das hat, füllt man das Feld (i,j) der Parsematrix mit

$$valid(i,j) := \{\, A \mid A \in DIST(i,j)(valid(4,4)) \,\},$$

dem Bild der Kategorienmenge am Quellknoten unter der Relation $DIST(i,j)$.

Wenn man $DIST(x)$ für alle $x \in X$ berechnet, wird nach der Konstruktion im Shortest-Paths-Lemma der Quadrant X rekursiv in Unterquadranten zerlegt:

$$X = \begin{array}{c|cc|cc|} & 5 & 6 & 7 & 8 \\ \hline 1 & & & & \\ 2 & B_1 & & C_1 & \\ \hline 3 & A_1 & & D_1 & \\ 4 & & & & \\ \hline \end{array} = \begin{array}{c|cc|cc|} & 5 & 6 & 7 & 8 \\ \hline 1 & & & & \\ 2 & B_1 & & C_1 & \\ \hline 3 & B_2 & C_2 & D_1 & \\ 4 & A_2 & D_2 & & \\ \hline \end{array}$$

Das führt hier (bei der anderen Zeilenindizierung) dazu, daß man über $A_1 = \{(3,5),(3,6),(4,5),(4,6)\}$ zu $A_2 = \{(4,5)\}$ kommt. Hier sieht man an H, daß es nur einen Weg $(4,4) \xrightarrow{B\backslash E} (4,5)$ gibt, der von der Quelle direkt in diesem Punkt landet, und daher ist $DIST(4,5) = \{B\backslash E\}$. Da die Parsematrix am Quellpunkt nur die Kategorie $D \neq B$ hat, ist ihr Feld $(4,5)$ mit $\emptyset$ zu füllen.

Nun muß man die Wege von der Quelle $(4,4)$ in den Bereich $B_2 = \{(3,5)\}$ berechnen. Das Lemma sagt nur, wie man zu den Kosten des Wegs $(4,4) \rightarrow (4,5) \uparrow (3,5)$ kommt. Die Kosten des Wegs $(4,4) \uparrow (3,4) \rightarrow (3,5)$ muß man vorher kennen; durch die Berechnung von $valid(1,4)$ im Rekursionsaufruf kennt man die Kosten der Kante $(4,4) \uparrow (3,4)$[6], was hier $V(4,3) = \emptyset$ ist. Die Kosten der Kante $(3,4) \rightarrow (3,5)$ kennt man aus der Matrix H, hier $H(4,5) = \{B\backslash E\} = \{(B,E)\}$. Insgesamt ist das Gewicht des Wegs

$$\Gamma((4,4) \uparrow (3,4) \rightarrow (3,5)) = V(4,3)\,;\, H(4,5) = \emptyset\,;\, \{B\backslash E\} = \emptyset.$$

Nun berechnen wir die Kosten zu Punkten im Bereich $D_2 = \{(4,6)\}$. Dorthin kommt man vom Quellpunkt $(4,4)$ nur durch den Weg $(4,4) \rightarrow (4,5) \rightarrow (5,6)$, woraus sich

$$\begin{aligned}
&\Gamma((4,4) \rightarrow (4,5) \rightarrow (5,6)) \\
&\quad = DIST(4,5)\,;\, H(5,6) = H(4,5)\,;\, H(5,6) = \{B\backslash E\}\,;\, \{A\backslash S, C\backslash E\} \\
&\quad = \emptyset
\end{aligned}$$

ergibt. Als nächstes können die Kosten zu Punkten im Bereich $C_2 = \{(3,6)\}$ berechnet werden. Das sind die der Wege

$$\begin{array}{l}
(4,4) \uparrow (3,4) \rightarrow (3,6), \\
(4,4) \uparrow (3,4) \rightarrow (3,5) \rightarrow (3,6), \\
(4,4) \rightarrow (4,5) \uparrow (3,5) \rightarrow (3,6), \\
(4,4) \rightarrow (4,5) \rightarrow (4,6) \uparrow (3,6).
\end{array}$$

Um den „kürzesten“ Weg nach $(3,6)$ zu berechnen, kann man die Berechnung vereinfachen, indem man die kürzesten Wege in die Bereiche B_2, D_2 benutzt. Dazu tragen wir diese $DIST_{(4,4)}(i,j)$ in eine Matrix ein:

$DIST$	5	6	7	8
1				
2	B_1		C_1	
3	$\emptyset$	C_2	D_1	
4	(B,E)	$\emptyset$		

und berechnen den Eintrag bei C_2 durch

$$\begin{aligned}
&DIST(3,6) \\
&\quad = V(4,3)\,;\, H(4,6) +^{\mathcal{R}_N} DIST(3,5)\,;\, H(5,6) +^{\mathcal{R}_N} DIST(4,6)\,;\, V(4,3) \\
&\quad = \emptyset\,;\, H(4,6) \cup \emptyset\,;\, H(5,6) \cup \emptyset\,;\, V(4,3) = \emptyset.
\end{aligned}$$

[6] i.a. eines Weges; man muß also wohl die Ergebnisse der Pfadberechnungen aufbewahren

Analog berechnen wir die Einträge in die Felder B_1, D_1 und C_1, z.B.

$$\begin{aligned}
&DIST(2,5)\\
&= V(4,2)\,;\, H(4,5) \cup DIST(4,5)\,;\, V(4,2) \cup DIST(3,5)\,;\, V(3,2)\\
&= \{(B,S),(E,C)\}\,;\,\{(B,E)\} \cup \{(B,E)\}\,;\,\{(B,S),(E,C)\} \cup \emptyset\,;\, V(3,2)\\
&= \emptyset \cup \{(B,C)\} \cup \emptyset = \{(B,C)\}
\end{aligned}$$

Wenn man das ausführt, erhält man die folgende Matrix der Abstände:

DIST	5	6	7	8
1	$\emptyset$	(D,C)	(D,E)	$(D,S),(C,E)$
2	(B,C)	$\emptyset$	$\emptyset$	(C,C)
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	(B,E)	$\emptyset$	$\emptyset$	$(A,S),(C,E)$

Wendet man die jeweiligen Relationen auf die Menge der Nonterminale an der Quelle $(4,4)$ in der Parsematrix Π bzw. *VALID* an, also auf $\{D\}$, so erhält man folgende Einträge für die fehlenden Felder der Parsematrix:

VALID	5	6	7	8
1	$\emptyset$	C	E	S
2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Die gesamte Parsematrix $VALID(1,8)$ ist daher

	1	2	3	4	5	6	7	8
1	B	E	C	A	–	C	E	S
2		C	A	–	–	–	–	–
3			D	–	–	–	–	–
4				D	–	–	–	–
5					C	E	–	B
6						B	–	–
7							B	F
8								E

6.5 Implementierung in Standard ML

Zuerst definieren wir ein Format für kontextfreie Grammatiken in Chomsky-Normalform. Damit Vektoren und Matrizen mit Nonterminalen indiziert werden können, muß man eine Liste `syncats` aller Nonterminale und eine Umrechnung `nth` von Zeilen- und Spaltennummern in zugeordnete Nonterminale angeben:

$\langle$*rytter.sml*$\rangle\equiv$

```
signature CNF =
    sig
        eqtype syncat
        eqtype lexcat
        datatype rule = SynRule of syncat * syncat * syncat
                      | LexRule of syncat * lexcat
        val rules : rule list
        val syncats : syncat list
        val nth : int -> syncat
    end
```

Zu einer Grammatik $G = (N, T, P, S)$ in Chomsky-Normalform brauchen wir

(i) den Ring $\mathcal{P}_N$ der Mengen von Nonterminalen von G,

(ii) den Ring $\mathcal{R}_N$ der zweistelligen Relationen zwischen Nonterminalen von G,

und zur Eingabe w der Länge n einige Matrizen der Dimension $n \times n$:

(i) die Parsematrix $valid \in Mat_{n\times n}(\mathcal{P}_N)$ mit Feldindizes[7] $0 \leq i, j < n$ und

$$valid(i,j) = \{ A \in N \mid A \Rightarrow_G a_i \cdots a_j \},$$

was man als Gewichtung der Kanten in $\mathcal{P}_N$ verstehen kann;

(ii) die Matrizen $H, V \in Mat_{n\times n}(\mathcal{R}_N)$, mit der wir die Relation $\Rightarrow$ zwischen Kanten (A, i, j) und (B, k, l), darstellen,

(iii) die Matrix $Dist(H, V, q) \in Mat_{n\times n}(\mathcal{R}_N)$ der kürzesten Wege im Verbandsgraphen (H, V, q).

Da in Programmiersprachen die Zeilen- und Spaltennummern bei 0 beginnen, sind gegenüber dem vorigen Abschnitt einige Indizes anzupassen.

Die Ringe $\mathcal{P}_N$ und $\mathcal{R}_N$ implementieren wir durch Matrizenringe über $\mathbb{B}$. Relativ zu einer festen Numerierung $i : \{0, \ldots, s-1\} \to N$ ist

$$\mathcal{P}_N \simeq Mat_{1\times s}(\mathbb{B}) \quad \text{und} \quad \mathcal{R}_N \simeq Mat_{s\times s}(\mathbb{B}),$$

d.h. eine Menge $M \subseteq N$ ist ein Vektor $M \in Mat_{1\times s}(\mathbb{B})$ der Länge s und eine Relation $R \subseteq N \times N$ eine Matrix $R \in Mat_{s\times s}(\mathbb{B})$ der Dimension $s \times s$.

Da $\mathcal{P}_N$ und $\mathcal{R}_N$ von G abhängen, implementieren wir sie durch Funktoren von SML. Den Ring $\mathcal{R}_N$ kann man mit dem Funktor `Matrix` als Matrixring über dem

[7] Wir passen hier auch die Numerierung der Buchstaben von w zu $w = a_0 \cdots a_{n-1}$ an. Rytter meint mit $valid(i,j)$ die Menge der Kanten für das Teilwort $a_i \cdots a_j$. Stattdessen benutzt man oft $w = a_1 \cdots a_n$ und Positionen $0, \ldots, n$ *zwischen* den a_i, sodaß $valid(i,j)$ die Kanten des Teilworts $a_{i+1} \cdots a_j$ meint, damit $B \in valid(i,j)$ und $C \in valid(j,k)$ mit $(A \to BC) \in P$ zu $A \in valid(i,k)$ der Verkettung $a_{i+1} \cdots a_j \cdot a_{j+1} \cdots a_k = a_{i+1} \cdots a_k$ passen.

Halbring `R = Boole` bilden, wenn man die Anzahl s der Nonterminale kennt; eine Matrix $R \in Mat_{s\times s}(\mathbb{B})$ läßt sich als Liste von Nonterminalpaaren anzeigen.

Die Bereiche $Mat_{n\times m}(R)$ werden mit `Matrix(R:SEMIRING n,m:int):MATRIX` definiert; da die Dimensionen nicht Teil des Typs der Matrizen sind, kann man $A \in Mat_{n\times m}(R)$ mit $R \in Mat_{m\times m}(R)$ mit der Multiplikation aus $Mat_{n\times m}(R)$ multiplizieren, weil die Ergebnismatrix wieder in $Mat_{n\times m}(R)$ liegt.

Das Bild einer Menge $A \in Mat_{1\times s}(\mathbb{B})$ unter der Relation $R \in Mat_{s\times s}(\mathbb{B})$, also

$$R[A] := \{\, b \mid \exists a(a \in A \wedge (a,b) \in R)\,\},$$

erhält man als Matrix $A \cdot R \in Mat_{1\times s}(\mathbb{B})$.

Beispiel 6.4 Für die Mengen $N \supseteq A$ und die Relation $R \subseteq N \times N$

$$\begin{aligned} N &= \{A_1,\ldots,A_4\} \\ A &= \{A_2,A_4\}, \\ R &= \{(A_1,A_2),(A_2,A_3),(A_3,A_4),(A_4,A_1),(A_4,A_4)\} \end{aligned}$$

erhält man das Ergebnis $R[A] = \{A_1,A_3,A_4\}$ in der Matrixdarstellung:

$$A \cdot R = (0,1,0,1) \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} = (1,0,1,1).$$

In SML läßt sich das wie folgt programmieren:

⟨*Anwendung einer Relation auf eine Menge*⟩≡

```
structure BooleMat = Matrix(structure R = Boole val rows = 4 val cols = 4);
structure BooleVec = Matrix(structure R = Boole val rows = 1 val cols = 4);
structure BooleVec : MATRIX
- val A = BooleVec.tabulate (fn (_,0) => 0 | (_,1) => 1
                             | (_,2) => 0 | (_,3) => 1 | _ => 0);
val A = #[#[0,1,0,1]] : BooleVec.t
- val R = BooleMat.tabulate (fn (i,j) => if j = i+1 then 1 else
                                         if i = 3 andalso (j=0 orelse j=3)
                                         then 1 else 0);
val R = #[#[0,1,0,0],#[0,0,1,0],#[0,0,0,1],#[1,0,0,1]] : BooleMat.t
- BooleVec.prod(A,R);
val it = #[#[1,0,1,1]] : BooleVec.t
```

6.5.1 Berechnung von Abständen im Verbandsgraphen

Die Berechnung der Abstände in einem quadratischen Verbandsgraphen kann über einem beliebigen Ring gemacht werden. Im Verbandsgraphen verlaufen vertikate Kanten von Zeilen i zu Zeilen j mit $i < j$ (wie bei Rytter), was durch

$i \downarrow j$ angedeutet wird, während bei der Parsematrix *valid* die vertikalen Kanten von Zeilen i zu Zeilen j mit $i > j$, also $i \uparrow j$, verlaufen.

⟨*lattice.sml*⟩≡
```
functor LatticeGraph (val dim : int; structure Ring : SEMIRING) =
    struct
       structure Mat =
       Matrix(structure R = Ring; val rows = dim; val cols = dim)

       fun singleEdgeExtend (H,V,q) D =
           let
               val sum = Mat.sum
               val prod = Mat.prod
               val tran = Mat.tran
               val F = tran(prod(tran D,V))
               val E = let val (qi,qj) = q
                       in
                           Mat.tabulate
                               (fn (i,j) => if i = qi andalso (j > qj)
                                            then Mat.sub(H,(qj,j))
                                            else if j = qj andalso (i > qi)
                                            then Mat.sub(V,(qi,i))
                                            else Ring.zero)
                       end
           in
               sum(E,sum(D,sum(F,prod(D,H))))
           end

       (* Recursively compute the distance to points in subsquare X with
          NW = (i1,j1) SE = (i2,j2) by splitting X into four subsquares *)

       fun shortestPaths (H,V,q) =
           let
               fun paths Dist (i1,j1) (i2,j2) =
                   if i2 = i1 then Dist
                   else
                       if i2 = i1+1 then singleEdgeExtend (H,V,q) Dist
                       else
                           let
                               val m = (i2+1-i1) div 2
                               val A = paths Dist (i1,j1) (i1+m-1,j1+m-1)
                               val B = paths A (i1+m,j1) (i2,j1+m-1)
                               val D = paths A (i1,j1+m) (i1+m-1,j2)
                               val C = paths D (i1+m,j1+m) (i2,j2)
                           in
                               C
```

```
                        end
                val (n,_) = Mat.dim
            in
                paths Mat.zero (0,0) (n-1,n-1)
            end
    end
```

Bemerkung 6.5 Mit der Matrix E berücksichtigen wir in `singleEdgeExtend` alle Wege, die aus einer Kante von q zum Zielpunkt p bestehen. Rytter macht das nicht und erhält deshalb auf keinen Fall die kürzesten Wege von q nach p, da er q nirgends benutzt! (Bei ihm dürfte immer $0^{\mathcal{R}}$ herauskommen.)

Rytter berechnet die Abstände noch insofern anders, als er `singleEdgeExtend` mehrfach aufruft:

$\langle$*Rytters Definition*$\rangle\equiv$

```
fun shortestPaths (H,V,q) =
    let
        fun paths Dist (i1,j1) (i2,j2) =
            if i2 - i1 = 0 then Dist
            else
                let
                    val m = (i2+1-i1) div 2
                    val A = paths Dist (i1,j1) (i1+m-1,j1+m-1)
                    val A1 = singleEdgeExtend (H,V,q) A
                    val B = paths A1 (i1+m,j1) (i2,j1+m-1)
                    val D = paths B (i1,j1+m) (i1+m-1,j2)
                    val D1 = singleEdgeExtend (H,V,q) D
                    val C = paths D1 (i1+m,j1+m) (i2,j2)
                in
                    C
                end
        val (n,_) = Mat.dim
    in
        paths Mat.zero (0,0) (n-1,n-1)
    end
```

Das ist m.M. aber völlig unnötig; in den Beispielen ergeben sich dieselben Werte.

Beispiel 6.6 Zum Testen benutzen wir den Halbring $\mathbb{N}_{+\infty}$, so daß die normale Weglänge berechnet wird. Dabei nehmen wir für $\mathbb{N}$ die (nicht negativen) ganzen Zahlen von SML, und für $+\infty$ die größte davon, `Int.maxInt = 10737418223`.

$\langle$*lattice.expl.sml*$\rangle\equiv$

```
structure NatMinPlus : SEMIRING =
    struct
        type t = int
        val zero = case Int.maxInt of
```

```
                        SOME n => n
                      | NONE   => raise Domain
        fun sum (x,y) = Int.min(x,y)
        val unit = 0
        fun prod (x,y) = (x + y) handle Overflow => zero
    end
```

Bei folgender Wahl von H und V sind alle Wege von q nach p gleich lang:

$\langle$*lattice.expl.sml*$\rangle{+}{\equiv}$

```
structure LG = LatticeGraph(val dim = 4; structure Ring = NatMinPlus)

val H = LG.Mat.tabulate (fn (i,j) => if j = i+1 then 1 else LG.Mat.Ring.zero)
val V = LG.Mat.tabulate (fn (i,j) => if j = i+1 then 1 else LG.Mat.Ring.zero)
val q = (1,1)

val sp = LG.shortestPaths (H,V,q)

val sp =
  #[#[1073741823,1073741823,1073741823,1073741823],
    #[1073741823,1073741823,1,2],
    #[1073741823,1,2,3],
    #[1073741823,2,3,4]]  : LG.Mat.t
```

Das funktioniert also für den Beispielring $\mathbb{N}_{+\infty}$. Mit `use "expl.sml";` lädt man die Datei in SML.

6.5.2 Berechnung der Parsematrix

Aus der Eingabe (w, G) soll rekursiv über die Wortlänge $n = |w|$ die Parsematrix

$$\mathit{valid}(i,j) = \{\, A \mid A \Rightarrow^* a_i \cdots a_j \,\}$$

der gültigen Kanten des Cocke-Younger-Kasami-Erkenners berechnet werden, gemäß

$$\begin{aligned}
\mathit{VALID}(0,n-1) &= \begin{pmatrix} \mathit{VALID}(0,n/2-1) & \mathit{IMPLIED}(n/2,n-1) \\ 0 & \mathit{VALID}(n/2,n-1) \end{pmatrix} \\
(H,V,q) &= \mathit{LG}(\mathit{VALID}(0,n/2-1) \cup \mathit{VALID}(n/2,n-1), \\
&\qquad (0,n-1)), \\
\mathit{IMPLIED}(n/2,n-1) &= \mathit{DIST}(H,V,q)(\mathit{valid}(n/2-1,n/2-1)),
\end{aligned}$$

wobei *DIST* aus dem Verbandsgraphen (H,V,q) die $\mathcal{R}_N$-Gewichte der Wege zu den Feldern im rechten oberen Quadranten bestimmt.

$\langle$*rytter.sml*$\rangle{+}{\equiv}$

```
functor Rytter(structure G:CNF; val word :G.lexcat list) =
```

```
struct
   val s = List.length G.syncats
   structure NTrel = Matrix(structure R = Boole; val rows = s; val cols = s)
   structure NTset = Matrix(structure R = Boole; val rows = 1; val cols = s)

   fun apply (R:NTrel.t, A:NTset.t) = NTset.prod(A,R)

   val n = List.length word
   structure PM = Matrix(structure R = NTset; val rows = n; val cols = n)
   structure LG = LatticeGraph(val dim = n; structure Ring = NTrel)

   structure MatR = Matrices(NTrel:SEMIRING)

   fun syncatToNTset (X:G.syncat) =
       NTset.tabulate (fn (i,j) => if G.nth(j) = X then 1 else 0)

   fun lexcatToNTset (a:G.lexcat) =
       let
           val cats = List.map (fn (G.LexRule(A,_)) => A)
                               (List.filter (fn (G.LexRule(_,b)) => a = b
                                                  | _ => false) G.rules)
           fun add (X::Xs) = NTset.sum(syncatToNTset X, add Xs)
             | add [] = NTset.zero
       in
           add cats
       end

   local
       val makeNTset = NTset.tabulate
       val makeNTrel = NTrel.tabulate
       fun isSynRule (k,i,j) =
           let
               val rule = G.SynRule(G.nth k,G.nth i,G.nth j)
           in
               List.exists (fn r => r = rule) G.rules
           end
   in
   val PH = Vector.tabulate
                 (s, fn C => makeNTrel (fn (B,A) => if isSynRule(A,B,C)
                                                    then 1 else 0))
   val PV = Vector.tabulate
                 (s, fn B => makeNTrel (fn (C,A) => if isSynRule(A,B,C)
                                                    then 1 else 0))
```

Bei der Berechnung von $VALID(n_1, n_2)$ behandeln wir den Fall $n_2 - n_1 = 1$ explizit, da das Feld (n_1, n_2) i.a. sowohl Elemente enthält, die horizontalen Abstand von (n_1, n_1) haben als auch solche, die vertikalen Abstand von (n_2, n_2) haben. Wir berechnen den Feldinhalt daher direkt durch

$$H(n_1, n_2)(\mathit{valid}(n_1, n_1)) + V(n_2, n_1)(\mathit{valid}(n_2, n_2)).$$

Die rekursive Berechnung des Abstands von (n_1, n_2) zu *einer* Quelle (n_1, n_1) und seine Anwendung auf diesen Quellpunkt würde hier nicht ausreichen.

Im Fall $n_2 = n_1$ wird $\mathit{VALID}(n_1, n_2)$ ebenfalls explizit berechnet, indem man alle Kategorien des jeweiligen Buchstabens a_{n_1} der Eingabe sammelt.

In den Fällen, wo $\mathit{VALID}(n_1, n_2)$ rekursiv über $|n_2 - n_1|$ berechnet wird, müssen die Abstände vom Quellpunkt

$$(n_1 + \frac{n_2 + 1 - n_1}{2} - 1, n_1 + \frac{n_2 + 1 - n_1}{2} - 1)$$

zu den Punkten (i, j) im rechten oberen Quadranten der Parsematrix berechnet und auf den Quellpunkt angewendet werden.

Dazu wird mit der Funktion `latticeGraph` aus den schon berechneten Feldern der Parsematrix ein Verbandsgraph (H, V, q) berechnet. Die Abstände zu q werden mit `LG.shortestPaths` berechnet, was einen Verbandsgraphen aus horizontalen Kanten $i \rightarrow j$ und vertikalen Kanten $i \downarrow j$ mit jeweils $i < j$ erwartet; deshalb muß die Indizierung der Zeilen angepaßt werden, indem man in (H, V, q) die Zeilennummern von V um die waagerechte Mittelachse der Parsematrix spiegelt und die Zeilennummer des Quellpunkts anpaßt.

$\langle \mathit{rytter.sml} \rangle{+}{\equiv}$

```
fun Valid (n1,n2) =
    if n2-n1 >= 1
    then
        let
            val n3 = (n2+1 - n1) div 2
            val sub = PM.sub
            val _ = TextIO.print ("Valid(" ^ (Int.toString n1)
                                  ^ ", "   ^ (Int.toString n2) ^ ")\n")
        in
            if 2 * n3 = (n2+1 - n1)
            then
                let
                    val NW = Valid (n1,n1+n3-1)
                    val SE = Valid (n1+n3,n2)
                    val M = PM.sum(NW,SE)
                    val (H,V,(q1,q2)) = latticeGraph(M,(n1,n2))
                in
                    if n2-n1 = 1
                    then
                        let
```

```
                        fun valid(i,j) =
                            if (i,j) = (n1,n1) orelse (i,j) = (n2,n2)
                            then lexcatToNTset (List.nth(word,i))
                            else if (i,j) = (n1,n2)
                            then NTset.sum(apply(LG.Mat.sub(H,(i,j)),
                                                 sub(M,(i,i))),
                                           apply(LG.Mat.sub(V,(j,i)),
                                                 sub(M,(j,j))))
                            else NTset.zero
                    in
                        PM.tabulate valid
                    end
                else
                    let (* correct row numbers for vertical edges and source node *)
                        val Vtr = LG.Mat.tabulate
                                      (fn(i,j) => LG.Mat.sub(V,(n-1-i,n-1-j)))
                        val qtr = (n-1-q1,q2)
                        val Dist = LG.shortestPaths (H,Vtr,qtr)
                        fun valid(i,j) =
                            if (n1 <= Int.min(i,j) andalso Int.max(i,j) < n1+n3)
                               orelse
                               (n1+n3 <= Int.min(i,j) andalso Int.max(i,j) <= n2)
                            then sub(M,(i,j))
                            else if n1 <= i andalso i < n1+n3
                                    andalso n1+n3 <= j andalso j <= n2
                            then apply(LG.Mat.sub(Dist,(n-1-i,j)),
                                       PM.sub(M,(q1,q2)))
                            else NTset.zero
                    in
                        PM.tabulate valid
                    end
            end
        else
            raise Size
    end
else
    let
        val _ = TextIO.print ("Valid(" ^ (Int.toString n1)
                              ^ ", "   ^ (Int.toString n2) ^ ")\n")
        fun valid(i,j) =
            if (i,j) = (n1,n1)
            then lexcatToNTset (List.nth(word,i))
            else NTset.zero
    in
        PM.tabulate valid
    end
```

Hier wird bei der Rekursion immer eine (leicht anzeigbare) $n \times n$-Matrix berechnet. Um die gewünschte Laufzeit zu erhalten, müßten die Rekursionsaufrufe wohl $n/2 \times n/2$-Teilmatrizen liefern, die man dann zusammensetzt.

6.5.3 Berechnung des Verbandsgraphen (H, V, q)

Aus den durch $VALID(0, n/2-1)$ und $VALID(n/2, n-1)$ gefüllten Feldern von *valid* muß der Verbandsgraph berechnet werden. (Die Zeilen werden wie im Beispiel numeriert, nicht wie im Funktor `LatticeGraph`!)

Sei $G = (\Sigma, N, P, S)$ mit $s = |N|$ Nonterminalsymbolen. Zur Berechnung von H und V extrahieren wir zuerst aus der Regelmenge P die Relationen

$$\begin{aligned} PH(Z) &= \{(Y,X) \mid (X \to YZ) \in P\}, \\ PV(Y) &= \{(Z,X) \mid (X \to YZ) \in P\}. \end{aligned}$$

Dann bilden wir

$$\begin{aligned} H(j,k) &= \{Y\backslash X \mid Y(i,j)\backslash X(i,k)\} \\ &= \{(Y,X) \mid (X \to YZ) \in P,\ Z \in valid(j+1,k)\} \\ &= \{(Y,X) \in PH(Z) \mid Z \in valid(j+1,k)\} \end{aligned}$$

aus dem Vektor $PH \in Mat_{1\times s}(\mathcal{R}_N)$, indem wir seine Komponenten mit den jeweiligen Bits von $valid(j,k) \in \mathcal{P}_N = Mat_{1\times s}(\mathbb{B})$ multiplizieren und die Ergebnisse in $\mathcal{R}_N$ addieren. Analog wird die Matrix V aus PV gebildet.

$$\begin{aligned} V(i,k) &= \{X/Z \mid X(k,j)/Z(i,j)\} \\ &= \{(Z,X) \mid (X \to YZ) \in P,\ Y \in valid(k,i-1)\} \\ &= \{(Z,X) \in PV(Y) \mid Y \in valid(k,i-1)\} \end{aligned}$$

$\langle rytter.sml \rangle{+}{\equiv}$

```
and latticeGraph (M:PM.t,(n1,n2):int*int) =
    let
        val _ = TextIO.print ("latticeGraph(M,(" ^ (Int.toString n1)
                              ^ "," ^ (Int.toString n2) ^"))\n")
        val n3 = (n2+1 - n1) div 2

        fun PHv(i,j) = Vector.tabulate
                       (s, fn k => if i+1<n andalso
                                      NTset.sub(PM.sub(M,(i+1,j)),(0,k)) = 1
                                   then Vector.sub(PH,k) else NTrel.zero)
        val H = LG.Mat.tabulate
                (fn (i,j) => Vector.foldr NTrel.sum NTrel.zero (PHv(i,j)))

        fun PVv(i,j) = Vector.tabulate
                       (s, fn k => if 0 <= i-1 andalso
```

```
                                            NTset.sub(PM.sub(M,(j,i-1)),(0,k)) = 1
                                        then Vector.sub(PV,k) else NTrel.zero)
        val V = LG.Mat.tabulate
                (fn (i,j) => Vector.foldr NTrel.sum NTrel.zero (PVv(i,j)))

        val q = (n1+n3-1,n1+n3-1)
    in (H,V,q) end
end
```

6.5.4 Darstellungsfunktionen

Folgende Funktionen wandeln die Matrixdarstellung in die Listendarstellung von Mengen und Relationen zwischen Nonterminalen um und werden nur zur Anzeige von Beispielen verwendet.

⟨*rytter.sml*⟩+≡

```
local
    fun collect (1::bs) j acc = collect bs (j+1) (j::acc)
      | collect (_::bs) j acc = collect bs (j+1) acc
      | collect nil j acc = List.rev acc
in
fun NTsetToSyncats (X:NTset.t) =
    let
        val bits = List.tabulate (s, fn j => NTset.sub(X,(0,j)))
    in
        map G.nth (collect bits 0 [])
    end
fun NTrelToSyncats (X:NTrel.t) =
    let
        fun bits i = List.tabulate (s, fn j => NTrel.sub(X,(i,j)))
        fun row i = map (fn j => (G.nth i, G.nth j)) (collect (bits i) 0 [])
        val rows = List.tabulate (s,fn k => k)
    in
        List.foldr (op @) [] (List.map row rows)
    end

fun showPH () = Vector.map (fn r => NTrelToSyncats r) PH
fun showPV () = Vector.map (fn r => NTrelToSyncats r) PV

fun showPM(C:PM.t) : G.syncat list vector vector =
    let
        val n = Vector.length C
        val vtab = Vector.tabulate
    in
        vtab(n, fn i => vtab(n, fn j => NTsetToSyncats
                                           (PM.sub(C,(i,j)))))
```

```
        end
    end

    fun showLG(H,V,q:int*int) =
        let
            val (n,_) = LG.Mat.dim
            fun tab f = Vector.tabulate (n,f)
            val sH = tab(fn i => tab(fn j => NTrelToSyncats
                                                (LG.Mat.sub(H,(i,j)))))
            val sV = tab(fn i => tab(fn j => NTrelToSyncats
                                                (LG.Mat.sub(V,(i,j)))))
        in
            (sH,sV,q)
        end

    fun showDist Dist =
        let
            val (n,_) = LG.Mat.dim
            fun tab f = Vector.tabulate (n,f)
        in
            tab(fn i => tab(fn j => NTrelToSyncats
                                      (LG.Mat.sub(Dist,(i,j)))))
        end
end
```

6.5.5 Effizienz

Diese Implementierung des Verfahrens von W. Rytter ist aus verschiedenen Gründen nicht effizient:

(i) da eine funktionale Implementierung angestrebt wurde, werden keine veränderbaren Matrizen benutzt, sondern jeweils neue Matrizen konstruiert;

(ii) bei den rekursiven Aufrufen werden Matrizen der Größe $n \times n$ konstruiert, wo jeweils ein Teilquadrant der Größe $n/2 \times n/2$ ausreichen würde;

(iii) Matrizen werden aus Funktionen erzeugt, was Umwandlungen von Funktionsobjekten in Vektoren erfordert.

Es ging hier nur um eine korrekte Version des Algorithmus und eine verständliche Beschreibung. Um eine schnellere Implementierung zu bekommen, sollte man `MONO-ARRAY2`, die veränderbaren 2-dimensionalen Arrays von SML, nutzen.

Beispiel 6.7 Wenn in SML durch `CM.make "rytter.cm";` die nötigen Signaturen, Strukturen und Funktoren erzeugt sind, kann man folgendes Beispiel mit `use "rytter.expl.sml";` nach SML laden.

Die Beispielgrammatik sei

⟨*rytter.expl.sml*⟩≡
```
(* CM.make "rytter.cm"; *)
val _ = Control.Print.printDepth := 30;

structure G : CNF =
  struct
    datatype syncat = S | A | B | C | D | E | F
    datatype lexcat =     a | b | c | d | e
    datatype rule = SynRule of syncat * syncat * syncat
                  | LexRule of syncat * lexcat
    val rules = [ SynRule(S,A,B), SynRule(A,C,D), SynRule(B,E,F),
                  SynRule(C,B,A), SynRule(C,A,E), SynRule(E,C,B),
                  SynRule(E,B,C), SynRule(F,B,E),
                  LexRule(A,a), LexRule(B,b),
                  LexRule(C,c), LexRule(D,d), LexRule(E,e)
                ]
    val syncats = [S, A, B, C, D, E, F]
    fun nth i = List.nth(syncats,i)
  end;
```

Der Typ der Terminalsymbole des Eingabeworts hängt von der Eingabegrammatik ab, aber die Signatur `CNF` verbirgt die Definition des Typs `lexcat`. Damit die Argumente von `Rytter` wohlgetypt sind, definieren wir zuerst die Grammatik und dann auf Top-Level den Datentyp ihrer lexikalischen Kategorien:

⟨*rytter.expl.sml*⟩+≡
```
datatype lexcat = datatype G.lexcat;
structure Cyk = Rytter(val word = [b, c, d, d, c, b, b, e] : G.lexcat list;
                       structure G = G);
```

An der Grammatik werden folgende Kandidaten für die horizontalen und vertikalen Kanten abgelesen:

⟨*Das ergibt:*⟩≡
```
Cyk.showPH();
(* val it = #[[],[(B,C)],[(A,S),(C,E)],[(B,E)],[(C,A)],[(A,C),(B,F)],[(E,B)]] *)
Cyk.showPV();
(* val it = #[[],[(B,S),(E,C)],[(A,C),(C,E),(E,F)],[(B,E),(D,A)],[],[(F,B)],[]] *)
```

Wir berechnen für die Eingabe *bcddcbbe* die Parsematrix und den Verbandsgraphen und die Abstandsmatrix (für die gesamte Eingabe):

⟨*rytter.expl.sml*⟩+≡
```
val M = Cyk.Valid(0,7);
val ppM = Cyk.showPM M;
```

```
(* val ppM =
   #[#[[B],[E],[C],[A],[],[C],[E],[S]],
     #[[],[C],[A],[],[],[],[],[]],
     #[[],[],[D],[],[],[],[],[]],
     #[[],[],[],[D],[],[],[],[]],
     #[[],[],[],[],[C],[E],[],[B]],
     #[[],[],[],[],[],[B],[],[]],
     #[[],[],[],[],[],[],[B],[F]],
     #[[],[],[],[],[],[],[],[E]]]
*)
val (H,V,q) = Cyk.latticeGraph(M, (0,7))
val ppLG = Cyk.showLG (H,V,q);
(* val it =
  (#[#[[],[(B,E)],[(B,C)],[],[],[],[],[]],
     #[[],[],[(C,A)],[],[],[],[],[]],
     #[[],[],[],[(C,A)],[],[],[],[]],
     #[[],[],[],[],[(B,E)],[(A,C),(B,F)],[],[(A,S),(C,E)]],
     #[[],[],[],[],[],[(A,S),(C,E)],[],[]],
     #[[],[],[],[],[],[],[(A,S),(C,E)],[(E,B)]],
     #[[],[],[],[],[],[],[],[(A,C),(B,F)]],
     #[[],[],[],[],[],[],[],[]]
    ],
   #[#[[],[],[],[],[],[],[],[]],
     #[[(A,C),(C,E),(E,F)],[],[],[],[],[],[],[]],
     #[[(F,B)],[(B,E),(D,A)],[],[],[],[],[],[]],
     #[[(B,E),(D,A)],[(B,S),(E,C)],[],[],[],[],[],[]],
     #[[(B,S),(E,C)],[],[],[],[],[],[],[]],
     #[[],[],[],[],[(B,E),(D,A)],[],[],[]],
     #[[(B,E),(D,A)],[],[],[],[(F,B)],[(A,C),(C,E),(E,F)],[],[]],
     #[[(F,B)],[],[],[],[],[],[(A,C),(C,E),(E,F)],[]]
    ],
   (3,3))
*)
```

Diese Abstände sind richtig, wobei die Zeilennumerierung der Parsematrix benutzt wird; so bedeutet $(E,F) \in V(2,1)$: wenn in eine Spalte der Zeile 2 ein E eingetragen wird, muß darüber(!) in Zeile 1 ein F eingetragen werden.

Um die Abstände vom Quellpunkt zu berechnen, muß man die Zeilennumerierung anpassen. Die Einträge sind Oberrelationen der von Hand berechneten Felder, da sie auf vollständigen Matrizen H und V beruhen. (In Abschnitt 6.4 wurden die für die Teile aus Π entstehenden Einträge von V und H ignoriert.)

$\langle$*rytter.expl.sml*$\rangle$+≡

```
fun ppDist (q1,q2) =
    let val n = 8
        val Vtr = Cyk.LG.Mat.tabulate
```

```
                    (fn(i,j) => Cyk.LG.Mat.sub(V,(n-1-i,n-1-j)))
        val qtr = (n-1-q1,q2)
        val Dist = Cyk.LG.shortestPaths (H,Vtr,qtr)
        fun swap X = Cyk.LG.Mat.tabulate
                       (fn(i,j) => Cyk.LG.Mat.sub(X,(n-1-i,j)))
    in
        Cyk.showDist (swap Dist)
    end
(* val ppDist q =
 #[#[[],[],[],[(B,E),(D,A),(E,E)],[(B,E)],[(B,F),(D,C)],[(A,E),(D,E)],
                                                  [(C,E),(D,S),(E,F)]],
   #[[],[],[],[(B,S),(E,C)],[(B,C)],[(B,E)],[(A,C)],[(B,B),(C,C),(E,E)]],
   #[[],[],[],[],[],[],[],[]],
   #[[],[],[],[],[(B,E)],[(A,C),(B,F)],[(A,E)],[(A,S),(C,E)]],
   #[[],[],[],[],[],[],[],[]],
   #[[],[],[],[],[],[],[],[]],
   #[[],[],[],[],[],[],[],[]],
   #[[],[],[],[],[],[],[],[]]
  ]
*)
```

Bem.: Man sollte vielleicht einen Datentyp als Argument an `Rytter` übergeben und damit die Signatur der Grammatik einschränken:

⟨*möglicherweise besser*⟩≡

```
structure Cyk = Rytter(datatype cat = a | b | c | d;
                       val word = [b:cat, c, d, d];
                       structure G = G : CNF where type lexcat = cat);
```

Hierdurch würde man wohl eine Signatur `CNF'` bekommen, die den Typ der lexikalischen Kategorie erkennen läßt.

6.5.6 Makefile

SML-NJ hat einen „Compilation Manager", der SML-Programme aus Noweb-Dokumentationsdateien extrahieren, Teile davon bei Bedarf übersetzen und Module zusammenbinden kann. In einem „Makefile" gibt man an, welche Programme wo zu finden sind und welche Strukturen[8] daraus erzeugt werden sollen.

(i) Hat man die vorliegende Dokumentationsdatei und die der Matrixprogramme, so braucht man den folgenden „Makefile":

⟨*rytter.nw.cm*⟩≡

```
Library
```

[8] Man kann nur Signaturen, Strukturen und Funktoren exportieren; Werte und Funktionen wie `singleEdgeExtend` muß man in eine Struktur verpacken.

```
    signature MONOID
    signature SEMIRING
    signature MATRICES
    signature MATRIX
    functor ScaleDisc
    functor Matrices
    functor Matrix
    structure Boole
    structure BM
    signature CNF
    functor LatticeGraph
    functor Rytter
  is
    $/smlnj-lib.cm
    $/ml-yacc-lib.cm
    $/basis.cm

    seminarskript.nw
           (subdir:.
            semiring.sml
            warshall.sml
            rytter.sml
            lattice.sml
           )
```

Man lädt diese Datei (in der SML-Version 110.45 vom Februar 2004) mit

```
- CM.make "rytter.nw.cm";
```

in SML, wodurch alles aus der Dokumentation extrahiert und, wenn nötig, neu übersetzt wird.

(ii) Hat man die Dokumentationsdatei nicht, aber die Quellprogramme, so benutze man folgende `.cm`-Datei:

⟨*rytter.cm*⟩≡

```
  Library
    signature MONOID
    signature SEMIRING
    signature MATRICES
    signature MATRIX
    functor ScaleDisc
    functor Matrices
    functor Matrix
    structure Boole
    structure BM
    signature CNF
    functor LatticeGraph
```

```
    functor Rytter
  is
    $/smlnj-lib.cm
    $/ml-yacc-lib.cm
    $/basis.cm

    semiring.sml
    warshall.sml
    rytter.sml
    lattice.sml
```

(iii) Bei früheren SML-Versionen (z.B. 110.0.3) lautet der Aufruf

```
- CM.make();
```

was implizit den „Makefile“ `sources.cm` des jeweiligen Verzeichnisses benutzt; in unserem Fall die folgende Datei:

⟨*sources.cm*⟩≡

```
  Library
    signature MONOID
    signature SEMIRING
    signature MATRICES
    signature MATRIX
    functor ScaleDisc
    functor Matrices
    functor Matrix
    structure Boole
    structure BM
    signature CNF
    functor LatticeGraph
    functor Rytter
  is
    semiring.sml
    warshall.sml
    rytter.sml
    lattice.sml
```

Die Beispieldateien kann man anschließend getrennt nach SML laden, z.B. mit

```
- use "rytter.expl.sml";
```

Literatur

[1] R. Backhouse. Regular algebra applied to language problems. *Journal of Logic and Algebraic Programming*, 66(2), 2006.

[2] O. de Moor, S. Drape, D. Lacey, and G. Sittampalam. Incremental program analysis via language factors. *to appear*, 2002. `http://web.comlab.ox.ac.uk/work/oege.de.moor/pubs.htm`.

[3] M. Harrison. *Introduction to Formal Languages*. Addison Wesley, Reading, Mass., 1978.

[4] L. Lee. Fast context-free parsing requires fast boolean matrix multiplication. In *Proceedings 35th ACL/8th EACL*, pages 9–15, 1997.

[5] W. Rytter. Context-free recognition via shortest paths computation: a version of Valiant's algorithm. *Theoretical Computer Science*, 143:343–352, 1995.

[6] E. Torres. Eine Variante von Valiants Erkennungsalgorithmus (Rytter 1995). Seminarausarbeitung, CIS, Universität München, Juni 2006.

[7] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Science*, 10:308–315, 1975.

[8] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9:11–12, 1962.