

# Indhold

Sammendrag	2
Kapitel 1. Introduktion	3
Kapitel 2. Dyck-sprogene	5
2.1. Notation	5
2.2. Egenskaber ved Dyck-sprogene	5
Kapitel 3. Modeller for dynamiske algoritmer	9
3.1. De dynamiske operationer	9
3.2. Beregningsmodeller	10
3.3. Randomiserede modeller	12
3.4. Terminologi	13
Kapitel 4. Resultater	15
Kapitel 5. Algoritmer for $D_1$ og $D'_1$	21
Kapitel 6. Monte Carlo algoritmer	25
Kapitel 7. Entydig dynamisk signatur af strenge	31
7.1. Tre-farvnings strategien	31
7.2. Blok-inddeling	34
7.3. Hierarkisk signatur kodning	35
7.4. Datastrukturen	40
Kapitel 8. Algoritmer for $D_k$ og $D'_k$ .	45
Kapitel 9. Fredman og Saks' resultat	49
Kapitel 10. Nedre grænser og simple reduktioner	59
10.1. Sammenhænge mellem problemer i <b>change</b> -modellen	61
Kapitel 11. Dynamisk prefix balancering	63
Kapitel 12. Anvendelser af dynamisk prefix balancering	67
12.1. En $\Omega\left(\frac{\log n}{(\log \log n)^2}\right)$ nedre grænse	68
Kapitel 13. Fremtidigt arbejde	71
Bilag A. Appendiks	73
Litteratur	75

## Sammendrag

Vi fremlægger en række resultater for dynamisk genkendelse af Dyck-sprogene, en klasse af sprog, der består af korrekt balancerede parenteser. Disse resultater omfatter polylogaritmiske øvre grænser for alle Dyck-sprog. Vi giver også tilhørende nedre grænser, blandt andet  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  nedre grænser for genkendelse af Dyck-sprog med 2 eller flere sæt af parenteser.

De øvre grænser for Dyck-sprog med flere typer parenteser bygger på et resultat af Mehlhorn, Sundar og Uhrig i [MSU94]. Alle viste nedre grænser bygger på et resultat af Fredman og Saks i [FS89]. For sidstnævnte resultat introducerer vi en overbygning, der er en ny reduktionsteknik vi kalder *dynamisk prefix balancering*. Denne teknik gør os i stand til at vise nedre grænser på  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  eller bedre for hovedparten af de betragtede problemer.

Specialet omfatter desuden en præsentation af nævnte datastruktur fra [MSU94], samt det benyttede resultat fra [FS89]. I begge præsentationer har vi forsøgt at give en forenklet fremstilling, og samtidigt tilpasset sætningerne til de specifikke formål i specialet.

## Introduktion

**Dynamiske algoritmer for Dyck-sprogene.** Dette speciale handler om dynamiske algoritmer knyttet til en delklasse af de kontekst-frie sprog kaldet *Dyck-sprogene*. Udgangspunktet er såkaldt dynamisk sproggenkendelse, der består i for et givet alfabet  $\Sigma$  og et formelt sprog  $L \subseteq \Sigma^*$ , at opretholde en input-instans i form af en streng  $x \in \Sigma^*$ , under opdateringer kaldet updates, således at medlemskab af  $L$  kan besvares løbende. Som eksempel kan man tænke sig input-instansen  $x$  som en vektor  $x = (x_1, x_2, \dots, x_n) \in \Sigma^n$ , hvor den dynamiske datastruktur understøtter en **change**-operation, der givet et indeks  $i$  og et symbol  $a \in \Sigma$ , ændrer  $x_i$  til  $a$ . Datastrukturen skal herudover understøtte forespørgsler, kaldet queries, f.eks. en **member**-operation, der løbende kan afgøre om  $x \in L$ . Den primære beregningsmodel vi vil arbejde med, for sådanne dynamiske problemer, vil være en unit-cost RAM med ordstørrelse  $O(\log n)$ .

Vores udgangspunkt er som nævnt Dyck-sprogene, der populært kan betragtes som de sprog, der består af strenge af korrekt balancerede parenteser, hvor en eller flere parentestyper tillades. F.eks. balancerer  $((()))$ , men ikke  $(())$ . Dyck-sprogene inddeles i to klasser kaldet de en-sidige og de to-sidige. I de en-sidige Dyck-sprog orienteres parenteserne således, at  $(())$  balancerer, men ikke  $))(($ , hvor begge dele tillades for de to-sidige sprog. I kapitel 2 defineres Dyck-sprogene formelt.

Som oplagt anvendelse af hurtige (læs polylogaritmiske) dynamiske algoritmer for Dyck-sprogene, er tekst-editorer. Brugeren vil under en editerings-interaktion ofte være interesseret i oplysninger om korrekt balancerede dele af teksten, om matchende parentespar m.v. Da en tekst under en editeringsinteraktion sædvanligvis kun ændrer sig i små skridt og brugeren samtidigt ikke ønsker for lange pauser imellem opdateringer på skærmen, vil gode dynamiske algoritmer naturligvis være specielt velegnede her. Vi har med udgangspunkt i sådanne praktiske formål også beskrevet særlige update og query-operationer som **insert /delete**, og **match**, der sigter mod sådanne praktiske formål.

**Resultater.** I [Mil92] blev emnet dynamisk sproggenkendelse introduceret og herunder dynamisk genkendelse af Dyck-sprog. I denne beskrivelse blev en dynamisk  $O(\log n)$  algoritme for en-sidige Dyck-sprog med en type parentes beskrevet. For en-sidige Dyck-sprog med flere typer parenteser blev der givet reduktioner, der essentielt viste, at dynamisk genkendelse af disse reducerer til dynamisk lighedstest af strenge, der opretholdes under update-operationer som indsættelse og fjernelse af tegn vilkårlige steder i strengene. I 1994 viste Mehlhorn, Sundar og Uhrig i [MSU94] imidlertid et overraskende resultat, der gav deterministiske polylogaritmiske tider for sidstnævnte problem. De væsentligste øvre grænser præsenteret i dette speciale bygger videre på dette grundlag. Vi præsenterer resultatet fra [MSU94] med mindre udbygninger, der er målrettet mod at give polylogaritmiske

tider for genkendelse af både en-sidige og to-sidige Dyck-sprog. Udover disse deterministiske algoritmer præsenterer vi også Monte Carlo randomiserede algoritmer, hvor vi skærer en logaritmisk faktor fra tiderne. Resultater knyttet til denne del af specialet indgår som en del af [FHM<sup>+</sup>95], som vi er medforfattere på.

Udover de øvre grænser har vi også beskæftiget os med nedre grænser. Vores arbejde kan her inddrages i to dele; reduktioner der klassificerer sværheden af forskellige dynamiske modeller/Dyck-sprog relativt til hinanden, samt reduktioner der giver nedre grænser for nogle udvalgte af de betragtede problemer. Basis for dette arbejde er et centralt nedre grænse resultat af Fredman og Saks fra 1989 publiceret i [FS89]. Dette resultat består i en  $\Omega(\frac{\log n}{\log \log n})$  grænse for et dynamisk problem, hvor opgaven er opretholdelse af en vektor  $x \in \{0, 1\}^n$ , således at pariteten af en vilkårlig prefix-sum kan besvares. Vi har valgt at præsentere beviset for denne nedre grænse i dette speciale, og har på visse punkter tilpasset/forenklet beviset til vores behov. Udover Fredman og Saks' resultat bygger flere af vores grænser tillige på en ny reduktionsteknik, kaldet *dynamisk prefix balancering*, som bliver introduceret i dette speciale. Denne reduktionsteknik betragter vi som vores væsentligste bidrag, og har sammen med de øvrige reduktioner givet grænser på mindst  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  for næsten alle de betragtede problemer. De nedre grænser, der ikke involverer dynamisk prefix balancering er også beskrevet i [FHM<sup>+</sup>95]. I kapitel 4 er der givet en overordnet oversigt over resultater fremlagt i dette speciale.

**Motivation for fremlagte resultater.** Emnet dynamisk sprog-genkendelse er som tidligere nævnt blandt andet introduceret i [Mil92] og også i [MSVT94]. I disse artikler er målet med polylogaritmiske tider i ovennævnte RAM-model introduceret som et naturligt mål for dynamiske sproggenkendelsesalgoritmer. I sidstnævnte artikel er klassen af sprog, der har sådanne algoritmer, introduceret med betegnelsen *incr-POLYLOGTIME*, og resultaterne fra dette speciale indebærer dermed det teoretisk interessante resultat, at alle Dyck-sprog tilhører denne klasse. Et naturligt spørgsmål vil være om dette mål er tilfredsstillende set i lyset af eksistensen af ikke-trivielle dynamiske datastrukturer med sublogaritmiske tider, f.eks. Van Emde Boas træer [vEB75] og Union-Find-datastrukturer (se f.eks. [GI91]) med deterministiske tider på  $O(\log \log n)$  eller bedre. Vores nedre grænser vist i dette speciale vil udelukke muligheden for sådanne eksponentielt bedre tider, og etablerer dermed, at der maksimalt er et polynomielt gab mellem vores øvre grænser og optimale løsninger.

## Dyck-sprogene

Dyck-sprogene er som nævnt sprogene omfattende korrekt balancerede parentes-strengene. De er inddelt i to klasser; de en-sidige og de to-sidige Dyck-sprog.

Lad  $\Sigma$  være et parentes-alfabetet med  $k$  sæt parenteser, defineret ved  $\Sigma = A \cup \bar{A}$ , hvor  $A = \{a_1, a_2, \dots, a_k\}$ , og  $\bar{A} = \{\bar{a}_1, \bar{a}_2, \dots, \bar{a}_k\}$ . Det *en-sidige* Dyck-sprog, betegnet  $D_k$ , defineres nu relativt til  $\Sigma$  ved følgende kontekst frie grammatik:

$$S \rightarrow SS \mid a_1 S \bar{a}_1 \mid \dots \mid a_k S \bar{a}_k \mid \epsilon.$$

Tilsvarende defineres det *to-sidige* Dyck-sprog, betegnet  $D'_k$ , ved:

$$S \rightarrow SS \mid a_1 S \bar{a}_1 \mid \bar{a}_1 S a_1 \mid \dots \mid a_k S \bar{a}_k \mid \bar{a}_k S a_k \mid \epsilon$$

**EKSEMPEL 2.1.** *I et programmeringssprog som f.eks. Pascal vil  $A = \{(\, [, \mathbf{begin}\}$  og  $\bar{A} = \{)\, ], \mathbf{end}\}$  være blandt dette sprogs parentes-symboler. For en streng  $x$ , der kun består af disse parentes-symboler fra kildeteksten (og naturligvis i samme rækkefølge), gælder der, at  $x \in D_3$ , hvis hele kildeteksten skal være en del af Pascal-grammatikken.*

De to-sidige Dyck-sprog er almindeligvis ikke på samme måde en del af et programmeringssprogs grammatik, så disse sprog er primært interessante ud fra en teoretisk synsvinkel.

### 2.1. Notation

For en streng  $u \in \Sigma^n$  indicerer vi fra 1 til  $n$ . Det  $i$ 'te tegn i  $u$  betegnes  $u_i$ .  $u_{[i..j]}$  betegner del-strengen  $u_i u_{i+1} \dots u_j$ . Den tomme streng betegnes  $\epsilon$ . Vi kan også tale om den spejlvendte streng  $u^R$ . Denne defineres for et parentes-alfabet  $\Sigma = A \cup \bar{A}$  som:

$$u^R = \bar{u}_m \bar{u}_{m-1} \dots \bar{u}_1$$

hvor  $\bar{\bar{a}}_i = a_i$ . F.eks. er  $[((\ ])^R = (]\ ])$ , og  $\epsilon^R = \epsilon$ .

### 2.2. Egenskaber ved Dyck-sprogene

I dette afsnit vil vi introducere nogle særlige reducerende afbildninger, der giver en alternativ karakterisation af Dyck-sprogene. Disse afbildninger (idet vi følger [Har78]) defineres induktivt ved:

Lad  $\Sigma = A_k \cup \bar{A}_k$  for et  $k \geq 1$ . Afbildningerne  $\mu_1, \mu_2 : \Sigma^* \rightarrow \Sigma^*$  er givet ved:

$$\begin{aligned} \mu_1(\epsilon) &= \epsilon, \\ \mu_1(x a_i) &= \mu_1(x) a_i \quad \text{for alle } i, \quad 1 \leq i \leq k, \\ \mu_1(x \bar{a}_i) &= \begin{cases} \mu_1(x) \bar{a}_i & \text{hvis } \mu_1(x) \notin \Sigma^* a_i, \quad 1 \leq i \leq k, \\ x' & \text{hvis } \mu_1(x) = x' a_i. \end{cases} \end{aligned}$$



$$\begin{aligned} h(ua_1) &= h(u) \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} & h(u\bar{a}_1) &= h(u) \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix} \\ h(ua_2) &= h(u) \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} & h(u\bar{a}_2) &= h(u) \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}. \end{aligned}$$

Afbildningen  $h$  har en meget nyttig egenskab, som vi senere benytter i vores Monte Carlo algoritmer.

**SÆTNING 2.1.** *Lad  $u \in \Sigma^*$ , med  $\Sigma = \{a_1, a_2\} \cup \{\bar{a}_1, \bar{a}_2\}$ . Da gælder*

$$u \in D'_2 \Leftrightarrow h(u) = I.$$

Denne kønne algebraiske egenskab anvendes i [LZ77] til at vise, at ordproblemet for de frie grupper kan løses i *LOGSPACE*. D.v.s. de to-sidige Dyck-sprog  $D'_k$  er i *LOGSPACE*.

Ovenstående konstruktion kan udvides til også at håndtere  $k > 2$  ved hjælp af følgende sætning.

**SÆTNING 2.2.** *Den frie gruppe genereret af  $k$  elementer  $\{a_1, a_2, \dots, a_k\}$  er isomorf med en undergruppe af den frie gruppe genereret af to elementer  $\{a_1, a_2\}$*

For et bevis se [Rot65], s. 254. Faktisk gælder, at hvis man sætter  $c_i = a_1^i a_2^i$  for  $1 \leq i \leq k$ , da vil  $c_1, c_2, \dots, c_k$  generere en fri gruppe, se [MKS66], problem 1.4.12.

Følgende sætning gør det muligt for os at udvide resultater for Dyck-sprog med to typer af parenteser til Dyck-sprog med  $k$  typer af parenteser, hvor  $k$  er en konstant.

**SÆTNING 2.3.** *Lad  $\Sigma_k = \{a_1, a_2, \dots, a_k\} \cup \{\bar{a}_1, \bar{a}_2, \dots, \bar{a}_k\}$  og  $\Sigma_2 = \{a_1, a_2\} \cup \{\bar{a}_1, \bar{a}_2\}$ . Der findes en afbildning  $\varphi: \Sigma_k^* \rightarrow \Sigma_2^*$  med følgende to egenskaber*

- $u \in D_k \Leftrightarrow \varphi(u) \in D_2$  for  $u \in \Sigma_k^*$ .
- $|\varphi(u)| = c|u|$  for en konstant  $c$ .

*Bevis.* For hvert  $a_i \in \{a_1, a_2, \dots, a_k\}$  tilknyt den binære repræsentation af  $i - 1$  med  $\lceil \log k \rceil$  bits. I disse binære tal erstattes 0 med  $a_1$  og 1 med  $a_2$ . Det betyder, at et element  $a_i$  tilknyttes en entydig streng  $b_i \in \{a_1, a_2\}^{\lceil \log k \rceil}$ . Vi tilknytter  $\bar{a}_i$  strengen  $b_i^R$ . Definer  $\varphi$  induktivt, så den tilknytter en streng  $b_j$  til hvert element  $i$  i  $x \in \Sigma_k^*$ .

Som eksempel kan man betragte  $D_4$  med de fire parenteser  $(, [, \{$  og  $\langle$

$$\begin{aligned} \varphi\left(\left(\right)\right) &= \left(\left(\right), \varphi\left(\right)\right)=) \\ \varphi\left(\left[\right]\right) &= \left([\right], \varphi\left([\right]\right)=] \\ \varphi\left(\left\{\right\}\right) &= \left[\left(\right), \varphi\left(\right)\right]=) \\ \varphi\left(\left\langle\right\rangle\right) &= \left[[\right], \varphi\left(\right)\right]=] \end{aligned}$$

Det er oplagt, at  $(\mu_1(u) = \epsilon) \Rightarrow (\mu_1(\varphi(u)) = \epsilon)$  ud fra definitionen af  $\varphi$ . Vi viser ved induktion i længden af  $u$ , at  $(\mu_1(\varphi(u)) = \epsilon) \Rightarrow (\mu_1(u) = \epsilon)$  også gælder.

Basis er oplagt, da  $\varphi(\epsilon) = \epsilon$ . Antag nu, at det holder for strenge af længde op til  $l$ . Vi skal så vise, at det også holder for strenge af længde op til  $l + 2$ . Lad  $u = u_1 u_2 \cdots u_{l+2}$ .

$$\mu_1(\varphi(u)) = \mu_1(\varphi(u_1)\varphi(u_2)\cdots\varphi(u_{l+2})) = \epsilon.$$

Det betyder, at der i strengen  $\varphi(u_1)\varphi(u_2)\cdots\varphi(u_{l+2})$  må findes et matchende par af parenteser  $a_i\bar{a}_i$ . Da denne streng er konstrueret ved hjælp af  $\varphi$ , må disse to parenteser  $a_i\bar{a}_i$  indgå i henholdsvis en blok af  $\lceil \log k \rceil$  venstre-parenteser  $b_j$  og en blok af  $\lceil \log k \rceil$  højre-parenteser  $b_{j+1}$ .

Da  $\mu_1(\varphi(u_1)\varphi(u_2)\cdots\varphi(u_{l+2})) = \epsilon$  ved vi at  $\mu_1(b_j b_{j+1}) = \epsilon$ . Ud fra konstruktionen af  $\varphi$  ved vi, at  $b_j = \varphi(u_j)$  og  $b_{j+1} = \varphi(u_{j+1})$ , samt at  $u_j = \bar{u}_{j+1}$ . Det betyder, at man kan fjerne  $u_j$  og  $u_{j+1}$  fra strengen  $u$  samt  $b_j$  og  $b_{j+1}$  fra strengen  $\varphi(u_1)\varphi(u_2)\cdots\varphi(u_{l+2})$  uden at ændre deres reducerede form. Lad  $u'$  være strengen  $u$  med  $u_j$  og  $u_{j+1}$  fjernet,  $u' = u_{[1..j-1]u_{[j+2..|u|]}}$  af længde  $l$ , hvilket betyder at

$$\begin{aligned} \epsilon &= \mu_1(\varphi(u)) = \mu_1(\varphi(u')) \\ &\Downarrow \quad (\text{induktionsantagelsen}) \\ \epsilon &= \mu_1(u') = \mu_1(u). \end{aligned}$$

Konstruktionen af  $\varphi$  giver direkte at  $|\varphi(u)| = \lceil \log k \rceil |u|$ .

□

## Modeller for dynamiske algoritmer

### 3.1. De dynamiske operationer

I introduktionen beskrev vi kort, hvilke operationer en dynamisk datastruktur for sprogenkendelsesproblemer kan have. Vi vil i dette kapitel mere detaljeret beskrive de modeller vi har taget udgangspunkt i. Udgangspunktet for alle modellerne er, at vi skal kunne foretage ændringer/opdateringer i en eller undertiden et univers af strenge over et fast alfabet  $\Sigma$ , og samtidigt kunne få besvaret forespørgsler vedrørende det aktuelle input. Vi vil tillade os at kalde opdateringer for *updates* og forespørgsler for *queries*.

**change-modellen.** Den mest enkle og samtidigt centrale model i dette speciale, er den såkaldte **change**-model, knyttet til en streng af fast længde  $n$  og kort berørt i introduktionen. For en input-instans givet ved en vektor  $x = (x_1, x_2, \dots, x_n) \in \Sigma^n$ , skal den dynamiske datastruktur understøtte een update-operation, kaldet **change**, der skal opfylde følgende:

**change**( $i, a$ ) ændrer  $x_i$  til  $a \in \Sigma$ .

Denne model skal så naturligvis samtidigt understøtte en eller flere query-operationer. For dynamisk sprogenkendelse fokuserer vi primært på følgende query-operationer:

**member**: Svarer Ja, hvis  $x \in L$ . Ellers Nej.

**prefix**( $i$ ): Svarer Ja, hvis  $x_{[1..i]} \in L$ . Ellers Nej.

**interval**( $i, j$ ): Svarer Ja, hvis  $x_{[i..j]} \in L$ . Ellers Nej.

**2-interval**( $i, j, p, q$ ): Svarer Ja, hvis  $x_{[i..j]}x_{[p..q]} \in L$ . Ellers Nej.

for et fastlagt sprog  $L \subseteq \Sigma^*$ .

Ovennævnte model er meget generel for dynamiske problemer, hvor man har fast inputstørrelse  $n$ . For dynamiske grafalgoritmer med fast antal knuder  $N$ , kan man f.eks. repræsentere grafen ved dens adjacency matrix,  $x \in \{0, 1\}^{N^2}$ , hvor indsættelse/fjernelse af en kant svarer til at sætte/slette den korresponderende bit i  $x$ . Sammenhængsegenskaber og lignende kan derefter besvares ved queryet **member**, hvor sproget  $L$  defineres passende. En af de vigtigste styrker ved den fikserede inputstørrelse  $n$ , er at den knytter sig godt til både en uniform og ikke-uniform beregningsmodel. Vi vil senere vende tilbage til dette emne.

Udover ovennævnte queries, der knytter sig til alle formelle sprog, kan man også ofte være interesseret i queries, der knytter sig til egenskaber ved det konkrete sprog. Som eksempel på dette vil vi i dette speciale betragte et query **match**, der er defineret for de en-sidige Dyck-sprog. **match** tager som parameter et index  $i$  i  $x$ , og skal så returnere et index  $j$  på en entydigt matchende parentes, d.v.s. et  $j \neq i$  så  $x_{[i..j]} \in D_k$  eller  $x_{[j..i]} \in D_k$ , hvor  $j$  er valgt, så  $|i - j|$  er minimal, og hvis en sådan parentes ikke findes, returneres værdien Udefineret.

I specialet beskæftiger vi os i forbindelse med de nedre grænser for tre andre naturlige dynamiske problemstillinger, der knytter sig til ovennævnte **change** -model. Der er her tale om følgende problemer:

**Paritet-prefix problemet:** Dette problem består i at opretholde  $x \in \{0, 1\}^n$  ved hjælp af ovennævnte **change** -operation, hvor vi tillige har queriet **prefix**( $k$ ), der returnerer  $\sum_{1 \leq i \leq k} x_i \pmod 2$ .

**Majoritet-prefix problemet:** I dette problem opretholdes  $x \in \{0, 1\}^n$  igen via **change**. Queriet **prefix**( $k$ ) returnerer Ja, hvis  $\sum_{1 \leq i \leq k} x_i \geq \lceil \frac{k}{2} \rceil$  og ellers returneres Nej.

**Signed prefix sum problemet:** Dette er en generalisering af paritet-prefix problemet, hvor  $x \in \{-1, 0, 1\}^n$  skal opretholdes via **change**, og med queriet **sum**( $k$ ), der returnerer værdien  $\sum_{1 \leq i \leq k} x_i$ .

**insert /delete -modellen.** Som nævnt er er **change**-modellen knyttet til et fast  $n$ . I mange dynamiske problemstillinger er dette ofte en for restriktiv begrænsning. F.eks. vil man under en editeringsproces have behov for at foretage indsættelser eller fjerne dele af en tekst, således at den samlede teksts længde løbende ændrer sig. Som en naturlig stærkere model, der fanger disse egenskaber, er der en model vi kalder **insert /delete** -modellen. I stedet for **change**-operationen har vi følgende operationer på en streng:

**insert**( $i, a$ ): Foretager en indsættelse af tegnet  $a \in \Sigma$  efter index  $i$  i  $x$ .

**delete**( $i$ ): Fjerner det  $i$ 'te tegn fra  $x$ .

De kompleksitetens mæssige mål er nu relativt til den aktuelle størrelse af inputtet. Denne model suppleres med query-operationer fuldstændigt i stil med **change**-modellen.

En sidste model vi betragter er en model for opretholdelse af mange strenge, hvor vi som i **insert /delete** -modellen har varierende inputstørrelse. Denne model knytter sig til datastrukturen i [MSU94], hvor vi skal opretholde en multimængde af strenge  $S$  over et alfabet  $\Sigma$ , hvor vi kan foretage operationer som:

**create**( $a$ ): Skaber strengen  $a$  og returnerer reference til denne i  $S$ .

**split**( $x, i$ ): Splitter strengen  $x = x_1 \dots x_m$  i de to strenge:  $y = x_1 \dots x_i$  og  $z = x_{i+1} \dots x_m$ , der indsættes i  $S$ , og referencer for disse returneres.

**concat**( $x, y$ ): Konkaterer strengene  $x$  og  $y$ , og returnerer reference.

**equal**( $x, y$ ): Afgør om  $x$  og  $y$  er ens.

**lcp**( $x, y$ ): Returnerer længden af største fælles prefix for strengene  $x$  og  $y$ .

For denne model vælger vi, at input-størrelsen er summen af længderne af strengene i  $S$ -universet;  $n = \sum_{x \in S} |x|$ .

### 3.2. Beregningsmodeller

**RAC'en.** Vores primære beregningsmodel er en *RAC* (Random Access Computer) defineret i [AV79]. Denne model er den velkendte *unit-cost Random Access Machine* med ordstørrelse som funktion af inputstørrelsen  $n$ . I dette speciale er ordstørrelsen altid den gængse på  $O(\log n)$  medmindre andet eksplicit er angivet.

Som motivationen for RAC'en er blandt andet, at den sikrer imod unaturligt kraftfulde operationer på ord (da ordstørrelsen f.eks. kun har størrelse  $O(\log n)$ ), og dermed lettes en tidsanalyse ved, at alle instruktioner blot koster een tidsenhed. Da der ikke er restriktioner på lageret, er denne model robust overfor forskellige mindre

afvigelse i instruktionssættet, idet vi kan have tabeller for særlige instruktionssæt liggende.

I forbindelse med sprogklasser for dynamiske problemer, er der i [MSVT94] foreslået sprogklasser som *incr-TIME*[ $t(n)$ ] og *incr-POLYLOGTIME*. I vores **change, member**-model, med ovennævnte RAC som beregningsmodel, vil sprog der accepteres i  $O(t(n))$  med polynomiel initialiseringstid ligge i klassen *incr-TIME*[ $t(n)$ ]. Klassen *incr-POLYLOGTIME* =  $\cup_{k \geq 0} \text{incr-TIME}[\log^k n]$ , definerer de sprog der kan genkendes i polylogartimisk tid, og som nævnt i introduktion er et af de fremlagte hovedresultater i dette speciale, at Dyck-sprogene tilhører denne klasse.

**Cell-probe modellen.** Den anden beregningsmodel vi beskæftiger os med, er den såkaldte cell-probe beregningsmodel indført i [Yao81]. Denne model anvendes i forbindelse Fredman og Saks nedre grænser vist i [FS89], hvoraf beviset for partitet-prefix problemet gengives i kapitel 9.

Cell-probe modellen er en ikke-uniform model, der fokuserer på antallet af læse/skrive-operationer, mens øvrige beregninger er gratis. Cell-probe modellen har ligesom RAC'en en ordstørrelse  $b$ , som parameter. En cell-probe algoritme defineres nu ved: Til hver mulig update eller query-operation (f.eks. **change(7,a)**) knytter vi et såkaldt *decision assignment* træ. Dette træ består af to typer knuder; skrive- og læsekuder. En skrive-knude har aritet 1 og er annoteret med et par  $(l, v)$ , hvor  $l$  er en hukommelseslokation for et ord i lageret, og  $v$  er en værdi fra  $\{0, \dots, 2^b - 1\}$ , der skal skrives i ordet  $l$ . En læseknude er kun annoteret med en lokation for et ord  $l$ , den skal læse. For hver af de  $2^b$  mulige værdier af en sådan læsning, har knuden en søn. For query-operationer har træerne tillige blade med de forskellige mulige query-svar, f.eks. Ja og Nej,  $0 \dots n$  etc. En "udførelse" af en cell-probe algoritme består i, at vi for en given operation gennemløber det for operationen associerede træ startende fra roden. For hver læse-knude annoteret  $l$  læses hukommelsescelle  $l$  og hvis denne har værdi  $i$ , fortsætter udførelsen til  $i$ 'te søn. Hvis der er tale om en skrive-knude annoteret med  $(l, v)$ , sættes hukommelsescelle  $l$  lig med værdien  $v$ , og udførelsen fortsætter til den efterfølgende søn. For query-operationer returneres værdien af et nået blad. Tidskompleksiteten for en cell-probe algoritme er dybden af det dybeste af træerne.

Cell-probe modellen udmærker sig bl.a. ved sin kombinatoriske natur. Samtidigt er fokuseringen på lagertilgang en anden styrke, der krystaliserer et centralt aspekt ved dynamiske problemer. Endelig er det naturligvis også vigtigt, at nedre grænser i denne model med  $b \in \Omega(\log n)$ , også holder for den realistiske RAC med samme ordstørrelse.

Generelt er cell-probe modellen begrænset m.h.t. hvilke grænser den kan vise. F.eks. er det klart at alle problemer kan klares i tid  $n$ , da vi kan lade hele inputtet  $x$  være gemt, og ved et query kan vi da scanne hele  $x$  og returnere et passende svar. I [Mil92] er dette forbedret en anelse, idet det vises, at alle funktioner kan klares i tid  $n - \log \log n + 2$ . Omvendt er denne øvre grænse for "næsten" alle funktioner meget tæt på det optimale, jævnfør følgende sætning vist ved et tælleargument i [Mil92].

**SÆTNING 3.1.** *For næsten alle funktioner, d.v.s.  $\frac{2^{2^n-1}}{2^{2^n}}$  ud af de  $2^{2^n}$  mulige funktioner  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ , skal en cell-probe algoritme, der løser  $f$ , mindst bruge tiden  $n - 2 \log n - 5$ .*

### 3.3. Randomiserede modeller

I dette speciale vil vi flere gange have behov for at analysere randomiserede algoritmer, dels i form af de øvre grænser for præsenterede Monte Carlo algoritmer, men også for nogle Las Vegas randomiserede reduktioner. Valg af model i konteksten for dynamiske algoritmer er her lidt mere kompliceret end for simple sekventielle input/output algoritmer. Vi har valgt at undlade at gå ind i videre undersøgelser af forskellige problematikker m.h.t. dette her, da det aldrig har været fokus for vores arbejde, men vil blot definere vores opfattelse af en rimelig model for Las Vegas/Monte Carlo randomiserede dynamiske algoritmer.

Generelt antager vi, at vi har fri adgang til tilfældige bits. I RAC-beregningsmodellen med ordstørrelse  $b$ , vil vi antage at den er udbygget med en særlig instruktion **rand**, der returnerer en uniformt tilfældig værdi fra  $\{0, \dots, 2^b - 1\}$ . Vi vil ofte omtale benyttelse af **rand**-instruktionen som et tilfældigt valg foretaget af algoritmen.

**Las Vegas randomisering.** For en Las Vegas randomiseret algoritme er vi interesserede i den *forventede tid* for udførelsen af en update/query-operation. Vi præciserer dette i følgende kontekst. Lad en update/query sekvens  $Q = u_1 \dots u_m$  være givet, hvor  $m$  kan være vilkårlig stor. Vi kan nu tale om den forventede udførelsestid for  $u_m$  relativt til sekvensen  $Q$ , defineret ved:

$$E(T_Q) = \sum_{t=0}^{\infty} t \cdot Pr(T_Q = t),$$

hvor  $T_Q$  betegner den stokastiske variabel, der antager værdien  $t$ , hvis de tilfældige valg Las Vegas-algoritmen foretager i sekvensen  $Q$ , bevirker at  $u_m$  har udførelsestid  $t$ . Vi mangler nu at definere den forventede tid for en Las Vegas algoritme uafhængigt af en sekvens  $Q$ . Vores valg er her, analogt til definition af worst-case deterministisk tid, at se på en sekvens  $Q$ , der giver maksimal forventet tid for den operation vi betragter. Vi støder imidlertid på et mindre teknisk problem m.h.t. inputtets størrelse, der eventuelt kan ændre sig for forskellige sekvenser  $Q$ . I **change**-modellen har vi fast input-størrelse  $n$ , hvor tidskompleksiteten uanset sekvens udtrykkes relativt til denne værdi. Vi kan derfor let definere worst-case forventet Las Vegas tid  $t(n)$  for en update-operation **change** til:

$$t(n) = \max\{E(T_Q) | Q \text{ ender med en } \mathbf{change} \text{-operation}\}$$

og tilsvarende er tiden for en fast query-operation **query** lig med:

$$\max\{E(T_Q) | Q \text{ ender med } \mathbf{query}\text{-operationen}\}.$$

For modeller, hvor de aktuelle input-størrelser varierer med forskellige update/query-sekvenser, restringerer vi ovennævnte update/query-sekvenser til dem, der knytter sig til den betragtede input-størrelse  $n$ . F.eks. vil den forventede tid  $t(n)$  for **insert**-operationen i **insert/delete**-modellen være defineret ved:

$$t(n) = \max\{E(T_Q) | Q \text{ ender med input-størrelse } n \text{ og en } \mathbf{insert} \text{-operation}\}.$$

Ovennævnte definition opfylder f.eks. at de *forventede* tider for en *deterministisk* algoritmes operationer præcis er de deterministiske worst-case tider.

**Monte Carlo randomisering.** For Monte Carlo-algoritmer er vi interesserede i at se på sandsynligheden for et forkert query-svar. Vores anvendelser er altid knyttet til **change-modellen**, og her udtaler vi os altid om fejlsandsynligheder knyttet til sekvenser af updates/queries af samme længde som inputtet  $n$ . Som før er sandsynligheder for et forkert svar defineret relativt til de tilfældige valg algoritmen har foretaget undervejs i denne sekvens.

### 3.4. Terminologi

Vi vil i resten af specialet benytte en særlig notation, der hjælper med at skelne de forskellige problemer for Dyck-sprogene i **change**-modellen. For et generelt query; **query**  $\in$  {**member**, **prefix**, **interval**, **2-interval**, **match**} og Dyck-sprog  $D$ , vil vi bruge notationen **query** $_D$  for det dynamiske problem, der i ovennævnte **change**-model er suppleret med operationen **query** defineret for Dyck-sproget  $D$ . F.eks. er **prefix** $_{D'_1}$  det dynamiske problem med update-operationen **change** og query-operationen **prefix** for  $D'_1$ .

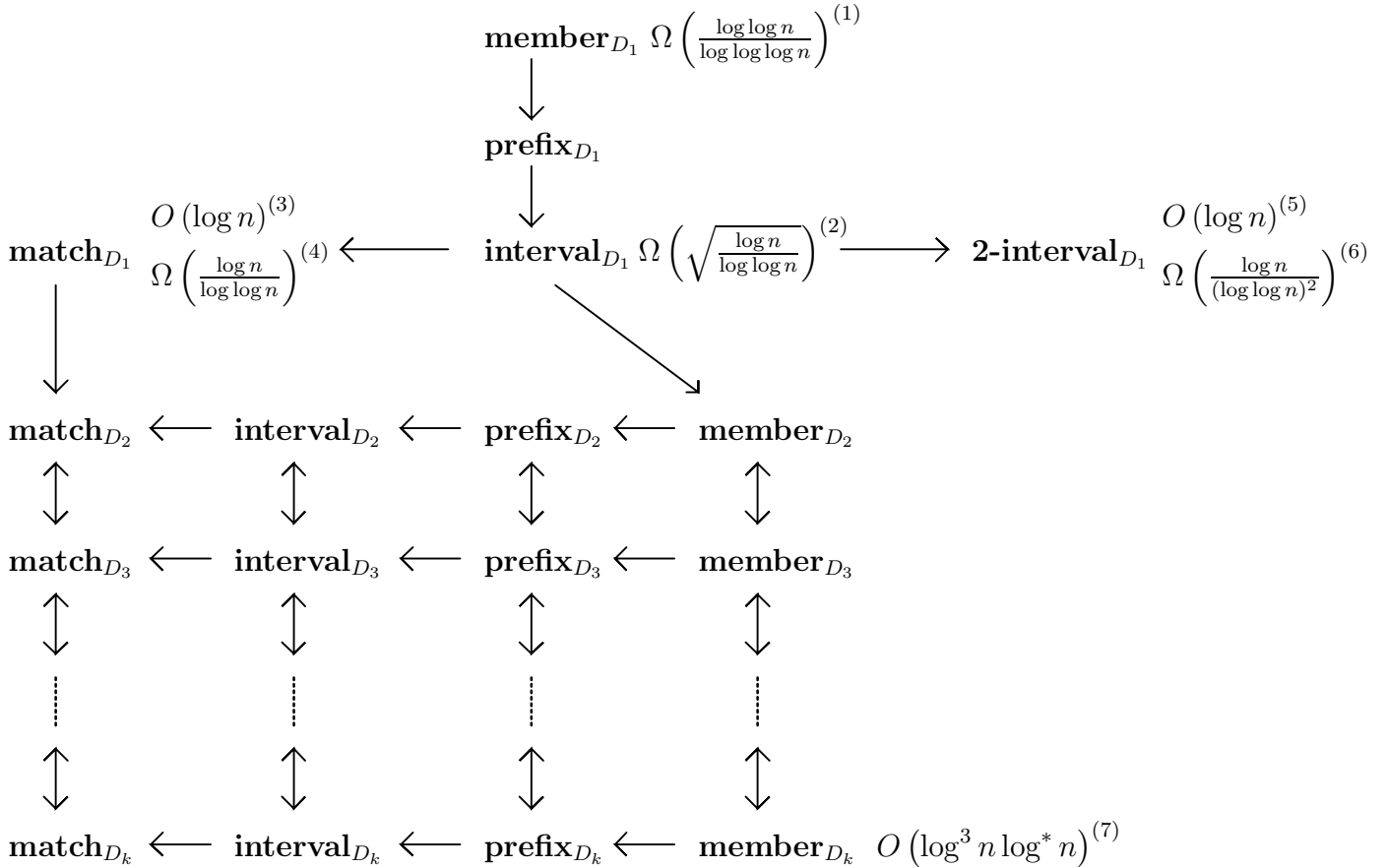
Vi vil ofte betegne problemer relativt til en konstant  $k$  i forbindelse med Dyck-sprogene  $D_k$  eller  $D'_k$ , med  $k$  parentestyper. Øvre grænser på  $O(t(n))$  for f.eks. **match** $_{D_k}$  betyder her, at problemet for vilkårlig konstant  $k \geq 1$  kan løses i tid  $O(t(n))$ , hvor  $k$ 'et kan være en skjult konstant i  $O$ -notationen. Omvendt vil en nedre grænse på  $\Omega(t(n))$  for et problem, f.eks. **member** $_{D_k}$ , betyde at der findes en konstant  $k$ , så en dynamisk algoritme for dette problem mindst skal bruge tid  $\Omega(t(n))$ .

Endeligt vil vi altid tale om deterministiske worst-case tider i nedre og øvre grænser med mindre andet er angivet.



## Resultater

**En-sidige Dyck-sprog.** For at give et overblik over sammenhængen mellem de forskellige operationer i **change**-modellen har vi lavet en hierarkisk oversigt. Figur 4.1 viser sammenhængen mellem operationerne i **change**-modellen for de en-sidige Dyck-sprog. Pilene viser reduktioner mellem de forskellige operationer.  $P \rightarrow Q$



FIGUR 4.1. Oversigt over sammenhængen mellem de forskellige operationer i **change**-modellen for de en-sidige Dyck-sprog.

betyder, at en instans af  $Q$  løser en instans af  $P$  med et konstant antal query og update operationer i instansen for  $Q$ . Traditionelt siger man, at  $P$  reducerer til  $Q$ . Det vil sige, at en øvre grænse for  $Q$  medfører den samme øvre grænse for  $P$ , og en nedre grænse for  $P$  medfører den samme nedre grænse for  $Q$ . Det betyder, at

på figur 4.1 holder de øvre grænser, når man bevæger sig imod en pil, hvorimod de nedre grænser holder, når man følger en pil. For eksempel kan man se, at der er en nedre grænse på  $\Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$  for alle operationerne på en-sidige Dyck-sprog med mere end én type parentes.

Følgende reduktioner ses alle direkte fra operationernes definitioner.

$$\begin{aligned} \mathbf{member}_{D_i} &\longrightarrow \mathbf{prefix}_{D_i} \longrightarrow \mathbf{interval}_{D_i} \longrightarrow \mathbf{2-interval}_{D_i} \\ \mathbf{match}_{D_i} &\longrightarrow \mathbf{match}_{D_{i+1}}. \end{aligned}$$

Udover disse er der en række ikke-trivielle reduktioner.

$$\begin{aligned} \mathbf{interval}_{D_i} &\longrightarrow \mathbf{match}_{D_i} \text{ (sætning 10.3).} \\ \mathbf{interval}_{D_1} &\longrightarrow \mathbf{member}_{D_2} \text{ (sætning 10.5).} \end{aligned}$$

Den sidste reduktion, som bevirker at hierarkiet “kollapser” (der findes reduktioner i begge retninger) på den ene led ved to eller flere parentes-typer, skyldes en grundlæggende egenskab ved de en-sidige Dyck-sprog, se sætning 2.3, som blandt andet medfører, at

$$\begin{aligned} \mathbf{member}_{D_k} &\longleftrightarrow \mathbf{member}_{D_2} \\ \mathbf{prefix}_{D_k} &\longleftrightarrow \mathbf{prefix}_{D_2} \\ \mathbf{interval}_{D_k} &\longleftrightarrow \mathbf{interval}_{D_2} \\ \mathbf{match}_{D_k} &\longleftrightarrow \mathbf{match}_{D_2}. \end{aligned}$$

I modsætning til de to-sidige Dyck-sprog ved vi ikke, om hierarkiet også “kollapser” på den anden led, altså om  $\mathbf{match}_{D_k}$  eller  $\mathbf{interval}_{D_k}$  reducerer til  $\mathbf{member}_{D_k}$ .

Som vist i sætning 8.1 vil en øvre grænse for  $\mathbf{interval}_{D_k}$ , også være en øvre grænse for  $\mathbf{match}_{D_k}$ . Vi har i specialet kun vist en konkret algoritme for  $\mathbf{member}_{D_k}$ , men vi mener, det er muligt at udvide denne, så den også håndterer  $\mathbf{match}_{D_k}$  eller  $\mathbf{interval}_{D_k}$ .

De påståede grænser på figur 4.1 udgør følgende sætninger:

- (1): Proposition 6.3 i [FHM<sup>+</sup>95].
- (2): Sætning 12.2.
- (3): Sætning 5.2.
- (4): Sætning 10.1.
- (5): Sætning 5.2.
- (6): Sætning 12.4.
- (7): Sætning 8.2.

**To-sidige Dyck-sprog.** På tilsvarende vis har vi lavet en oversigt over operationerne i **change**-modellen for de to-sidige Dyck-sprog. Her er de interessante reduktioner

$$\mathbf{interval}_{D'_i} \longrightarrow \mathbf{member}_{D'_{i+1}} \text{ (sætning 10.4),}$$

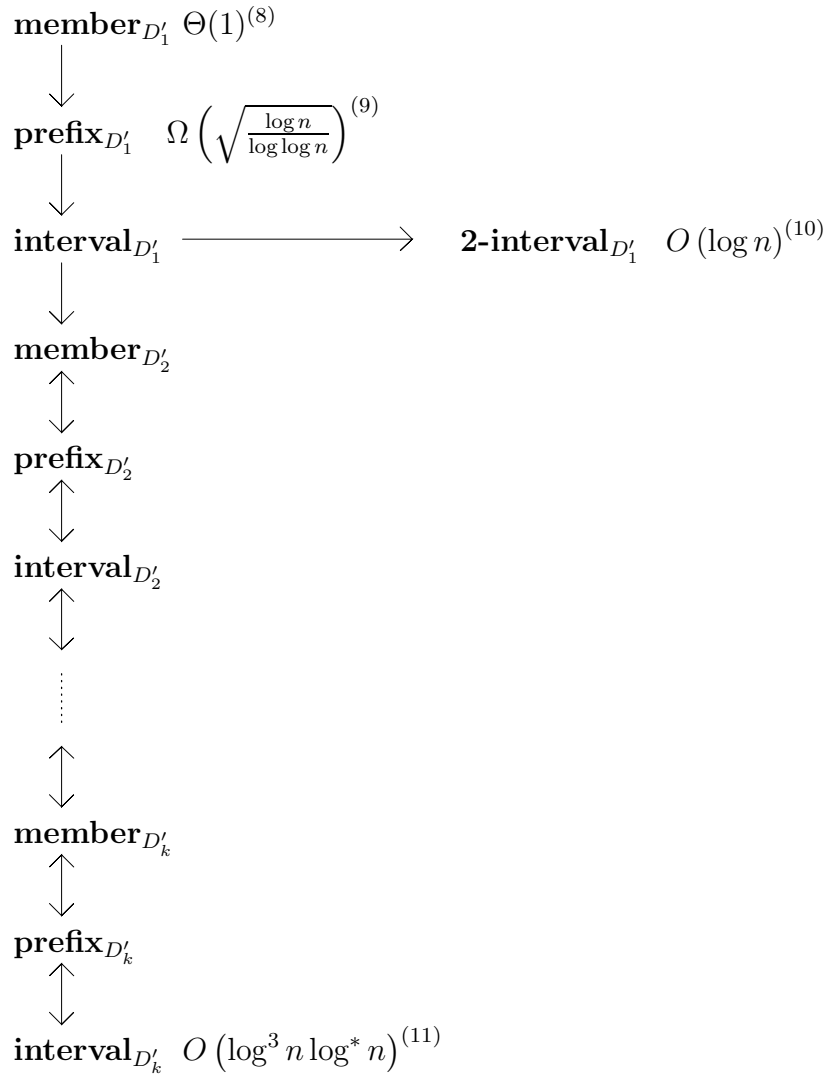
og sætning 2.2, som bevirker at

$$\mathbf{member}_{D'_k} \longrightarrow \mathbf{member}_{D'_2}.$$

I modsætning til de en-sidige Dyck-sprog bevirker dette, at hierarkiet “kollapser” fra  $\mathbf{member}_{D'_2}$  til  $\mathbf{interval}_{D'_k}$ .

De øvre grænser svarer helt til de øvre grænser for de en-sidige Dyck-sprog, dog med den forskel, at  $O(\log^3 n \log^* n)$  grænsen i denne model også er vist for  $\mathbf{interval}_{D'_k}$  ved hjælp af ovenstående reduktioner. Grænsen  $O(\log n)$  for  $\mathbf{2-interval}_{D'_1}$  er ikke vist direkte her i specialet, men kan nemt udledes af konstruktionen for  $D_1$ , se kapitel 5 sætning 5.2. Som det fremgår af figur 4.2, holder den stærke grænse

$\Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$  allerede fra **prefix** $_{D'_1}$  i det to-sidige tilfælde. Det betyder, at der for de to-sidige Dyck-sprog med en type parentes kun er et lidt mere end kvadratisk stort gab mellem øvre og nedre grænser.



FIGUR 4.2. Oversigt over sammenhængen mellem de forskellige operationer i **change**-modellen for de to-sidige Dyck-sprog.

De påståede grænser på figur 4.2 udgør følgende sætninger:

- (8): Sætning 5.1.
- (9): Sætning 12.1.
- (10): Se ovennævnte diskussion.
- (11): Sætning 8.1.

**Øvrige resultater.** Udover ovennævnte deterministiske øvre grænser har vi også enkelte Monte Carlo grænser. For **member** $_{D'_k}$  er der en  $O(\log n)$  øvre grænse (sætning 6.1) og for **member** $_{D_k}$  en  $O(\log^2 n)$  øvre grænse (sætning 6.2).

Endelig har vi to nedre grænser, der heller ikke indgår som en del af figur 4.1 og 4.2. Den ene er en  $\Omega(\frac{\log n}{\log \log n})$  grænse for **insert /delete**-modellen med **member**-operationen (sætning 10.2). For majoritet-prefix problemet, har vi som biprodukt af grænsen for **2-interval** $_{D_1}$ , også en  $\Omega(\log n / (\log \log n)^2)$  grænse for dette problem (sætning 12.3).

**4.0.1. Teknikker.** De øvre grænser vist på figur 4.1 og 4.2 bygger alle (pånær algoritmen for  $D'_1$ ) på samme grundide. Instansen  $x$  repræsenteres af et balanceret binært træ, hvor det  $i$ 'te blad repræsenterer  $x_i$ , og hver intern knude repræsenterer den delstreng, som knuden dækker over, på reduceret form. Man har altså i alle algoritmerne brug for at kunne sammensætte to reducerede strenge  $\mu_i(u)$  og  $\mu_i(v)$  til en ny reduceret streng  $\mu_i(uv)$ . Hvordan de reducerede delstrengene er repræsenteret, og hvordan de beregnes er derimod forskellig fra algoritme til algoritme.

Monte Carlo algoritmen for  $D'_k$  anvender et resultat af Lipton og Zalcstein [LZ77], der beskriver en teknik til at beregne reducerede delstrengene ved hjælp af matricer. En fingeraftryks-algoritme i stil med [KR87] giver os derefter den øvre grænse for Monte Carlo algoritmen. I det en-sidige tilfælde er tingene mere kompliceret, da  $D_k$  ikke har de samme pæne algebraiske egenskaber som  $D'_k$ , se kapitel 2.2. Vi bruger en balancerings-teknik vist i [FHM<sup>+</sup>95], som reducerer problemet til sammenligning af strenge af venstre-parenteser og strenge af højre-parenteser. Denne sammenligning kan klares med Monte Carlo algoritmen for de to-sidige Dyck-sprog.

I det deterministiske tilfælde med flere typer parenteser reducerer problemet reelt til dynamisk sammenligning af strenge. Netop dette problem beskriver Mehlhorn, Sundar og Uhrig som nævnt i [MSU94]. Ved at tilføje en operation, til deres struktur, som kan finde længste fælles prefix af strenge, er det muligt at vise en polylogaritmisk øvre grænse for genkendelse af alle Dyck-sprogene. For de en-sidige Dyck-sprog har man samme problemer, som i Monte Carlo algoritmen. Vi anvender samme løsning som i denne, men bruger istedet Mehlhorn et al strukturen til at sammenligne strengene.

I flere af algoritmerne anvender vi global rebuilding af Overmars [Ove83]. I Monte Carlo algoritmerne anvender vi den til at opbygge en ny struktur i baggrunden for at begrænse fejlsandsynligheden. I de deterministiske algoritmer  $D_k$  og  $D'_k$  anvender vi den som en form for garbage collector, igen ved hele tiden at opbygge en ny struktur i baggrunden.

Som nævnt i introduktionen er alle nedre grænser afledt af en nedre grænse af Fredman og Saks fra [FS89], der består i en  $\Omega(\frac{\log n}{\log \log n})$  grænse for paritet-prefix problemet. Grænserne for **match**-operationen og **insert /delete**-modellen er opnået ved direkte reduktioner fra paritet-prefix problemet. For de øvrige nedre grænser har vi anvendt vores reduktions-teknik *dynamisk prefix balancering*. Under anvendelsen af denne teknik udnytter vi to essentielle egenskaber ved Fredman og Saks' resultat, der er nævnt, men ikke eksplicit fremhævet i [FS89]. Den ene egenskab er, at grænsen på  $\Omega(\frac{\log n}{\log \log n})$  fremkommer som et "trade-off" imellem tiden brugt i updates versus tiden for queries. Den anden egenskab er, at resultatet kan styrkes til at holde for *forventet* query-tid for en Las Vegas randomiseret algoritme.

Den sidste egenskab er afgørende for de viste  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  grænser, hvor vi anvender den dynamiske prefix balancering i en randomiseret udgave, til at vise en særlig nedre grænse for signed prefix sum-problemet. En helt anden anvendelse af dynamisk prefix balancering gives i forbindelse med grænsen for **2-interval**-operationen. Det viser sig her, at majoritet-prefix problemet reducerer til **2-interval**<sub>D<sub>1</sub></sub>, og v.h.a. en særlig binær søgning i forbindelse med den dynamiske prefix balancering, kan vi opnå den næsten kvadratisk stærkere grænse  $-\Omega(\log / (\log \log n)^2)$  for de to problemer.

I figur 4.1 og 4.2 er der naturligvis en række tilbageværende spørgsmål. I kapitel 13 vil vi kort vende tilbage til nogle af disse uafklarede spørgsmål.

**4.0.2. Vores rolle.** Da specialet omfatter flere originale resultater, hvor flere af resultaterne er sket i et samarbejde med andre, vil vi her præcisere hvilke dele af specialet, der er vores egne bidrag, og hvilke der er sket i samarbejdet omkring artiklen [FHM<sup>+</sup>95], eller er præsentation af andres resultater.

**Resultater fra [FHM<sup>+</sup>95]:** Disse resultater, som vi har medvirket til omfatter essentielt alle præsenterede polylogaritmiske øvre grænser i forbindelse med Dyck-sprogene. De præsenterede nedre grænser for **match**-operationen i **change**-modellen og **insert /delete**-modellens grænse er også en del af denne artikel.

**Præsentation af kendte resultater:** Denne del af specialet består af vores egen fremstilling af de omtalte resultater af Mehlhorn, Sundar og Uhrig [MSU94] og Fredman og Saks resultat [FS89]. Som nævnt er disse på visse punkter udvidet/forenklet med udgangspunkt i vores anvendelser.

**Egne bidrag:** Denne del består i reduktions-teknikken dynamisk prefix balancering, samt resultater afledt heraf. Disse resultater er specielt de nedre grænser for **prefix**<sub>D<sub>1</sub>'</sub>, **interval**<sub>D<sub>1</sub></sub>, **2-interval**<sub>D<sub>1</sub></sub> og majoritet-prefix problemet, i kombination med de simple reduktioner angivet ved pilene i figur 4.1 og 4.2.



## Algoritmer for $D_1$ og $D'_1$

Vi starter med de simple algoritmer for  $D_1$  og  $D'_1$ . De to grænser er ikke direkte sammenlignelige, da algoritmen for  $D'_1$  kun kan svare på **member**-queries, mens datastrukturen for  $D_1$  indeholder så meget information, at den kan svare på alle naturlige queries. For eksempel **prefix** og **match**.

SÆTNING 5.1. **member** $_{D'_1}$  kan klares i konstant tid pr. operation.

*Bevis.* Dette problem er faktisk blot et spørgsmål om at tælle antallet af højre- og venstreparenteser. Der gælder nemlig følgende for alle  $x \in \{a, \bar{a}\}^*$

$$x \in D'_1 \iff |x|_a = |x|_{\bar{a}}.$$

Det er oplagt, at en streng i  $D'_1$  indeholder lige mange højre og venstre parenteser. Den anden vej følger af den kendsgerning, at en reduceret streng over  $\{a, \bar{a}\}^*$  (med hensyn til  $D'_1$ ) ikke kan indeholde både  $a$  og  $\bar{a}$ .

Det betyder, at man kun behøver at tælle antallet af forekomster af  $a$  og  $\bar{a}$  i  $x$ , hvilket kan gøres i konstant tid pr. update. □

Den ovenstående løsning kan *ikke* bruges til for eksempel **prefix**-queries. Faktisk viser vi en nedre grænse på  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  i sætning 12.1. For operationerne (**prefix**, **interval** og **2-interval**) findes en logaritmisk øvre grænse ved brug af et binært træ.

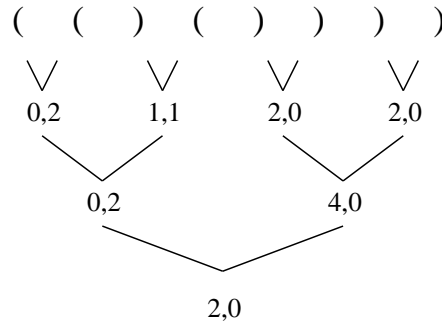
SÆTNING 5.2.  $D_1$  kan klares i tid  $O(\log n)$  pr. operation for alle operationerne i **change**-modellen.

*Bevis.* Bemærk først at for enhver streng  $x \in \{a, \bar{a}\}^*$ , gælder der, at den reducerede streng vil være på formen  $\bar{a}^r a^l$ , hvor  $r, l \geq 0$  er antallet af ikke matchede højre- og venstre-parenteser.

Dette leder naturligt hen imod en datastruktur til håndtering af problemet. Informationerne gemmes i et balanceret binært træ, hvor det  $i$ 'te blad indeholder  $x_i$ , og en intern knude repræsenterer sammensætningen af dens sønners strenge. I hver knude gemmer vi et par  $(r, l)$  af heltal, der beskriver den reducerede streng. Se figur 5.1. Bemærk at  $x \in D_1$  hvis og kun hvis roden indeholder  $(0, 0)$ . En knude kan beregnes ud fra dens to sønner efter følgende simple regel. Vi skriver  $y \equiv (r, l)$  for en knude i træet, der repræsenterer del-strengen  $y$ , og som indeholder parret  $(r, l)$  (jævnfør tidligere beskrivelse). Lad  $y \equiv (r, l)$  være en knude i træet, og lad  $u \equiv (r_1, l_1)$  og  $v \equiv (r_2, l_2)$  være dens sønner, så  $y = uv$ .

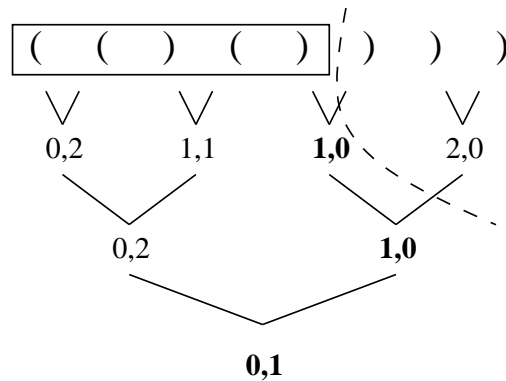
$$(r, l) = \begin{cases} (r_1, l_1 - r_2 + l_2), & \text{hvis } l_1 \geq r_2 \\ (r_1 + r_2 - l_1, l_2), & \text{ellers.} \end{cases}$$

Denne beregning kan foretages i konstant tid for hver knude. En **change**-operation, der ændrer et blad, kan således behandles i tid svarende til højden af træet nemlig



FIGUR 5.1. Træet for strengen “(()())”.

$O(\log n)$ . **member** operationen kan klares i konstant tid, da man blot behøver at undersøge roden af træet. Et **match** query er en smule mere kompliceret. Betragt tilfældet, hvor man ønsker at matche en venstreparentes i strengen (højreparenteser behandles efter samme princip). Man bevæger sig fra bladet med venstreparentesen ned i træet og holder hele tiden styr på hvor mange venstreparenteser, man mangler at matche. Dette er nødvendigt, da man kan få et bidrag af venstreparenteser, som først skal matches, fra en højre-søn. Når man en knude, hvis højre søn har et antal umatchedede højreparenteser, der er større end eller lig det antal, man leder efter, begynder man at bevæge sig op i træet. Denne søgning kan naturligvis klares i tid  $O(\log n)$ . **prefix**( $i$ )-operationen svarer til at “bortskære” den del af træet, som repræsenterer den sidste del af strengen. Se figur 5.2. Dette gøres på følgende måde. Vi definerer  $P$  som mængden af knuder, hvis højre-søn repræsenterer en del-streng,

FIGUR 5.2. **prefix**(5) for strengen “(()())”. De genberegnete knuder er markeret med **fede** typer.

der ligger efter prefix’et, og hvis venstre-søn repræsenterer en del-streng, der ligger helt eller delvist i prefix’et. Disse knuder ligger på én sti i træet, fra det  $i$ 'te blad til roden. På figur 5.2 består  $P$  af de to fede knuder, som indeholder 1, 0. Lad  $y \equiv (r, l)$  være en knude i  $P$ , og lad  $u \equiv (r_1, l_1)$  og  $v \equiv (r_2, l_2)$  være dens sønner, så  $y = uv$ . At “skære”  $v$  bort svarer til at betragte  $r_2$  og  $l_2$  som 0. Vi beregner stien fra det  $i$ 'te blad til roden, og behandler alle knuderne fra  $P$  som beskrevet ovenfor. Disse

beregninger skal naturligvis ikke ændre det oprindelige træ. Den nye rod fortæller så, om prefix'et er med i sproget. Igen bruges der tid  $O(\log n)$ .

Denne metode kan passende generaliseres til **interval** og **2-interval** operationer.

□

Ved at bruge dynamiske søgetræer kan ovenstående datastruktur let udvides til også at håndtere **insert** og **delete** operationer. Dette er nødvendigt, da en **insert/delete** operation vil ændre træets topologi. Til dette kan man bruge en af de mange strategier til at balancere dynamiske søgetræer f. eks. *rød-sortede søgetræer* [GS78], som sikrer træet en dybde af størrelse  $O(\log n)$ .



## Monte Carlo algoritmer

Vi betragter nu Dyck-sprogene med flere typer af parenteser startende med Monte Carlo algoritmerne, da disse er de simpleste.

Vi starter med algoritmen for  $D'_k$ , som er den nemmeste på grund af de algebraiske egenskaber ved de to-sidige Dyck sprog, se kapitel 2.2.

**SÆTNING 6.1. member**  $D'_k$  kan klares i tid  $O(\log n)$  pr. operation, så sandsynligheden for et forkert svar på et query i en sekvens af  $n$  operationer er  $O(\frac{1}{n})$ .

*Bevis.* Som nævnt tidligere har de to-sidige Dyck sprog en pæn algebraisk fortolkning, se kapitel 2.2. Lad  $x$  være den instans af  $\Sigma^n$  vi arbejder på. Med denne tolkning har vi nu, at  $x \in D'_2$  hvis og kun hvis  $h(x)$  er identitetsmatricen. ( $h$  er den korrespondance, som er beskrevet i kapitel 2.2.)

Dette lægger op til en randomiseret fingeraftryks-algoritme i stil med [KR87]: beregn  $h(x)$  modulo et tilfældigt primtal  $p$  og check om resultatet er identitetsmatricen.

Til det dynamiske tilfælde opbygger vi et binært træ, hvor det  $i$ 'te blad repræsenterer  $h(x_i)$  og hver knude indeholder produktet af dens to sønner modulo  $p$ . Roden af dette træ vil dermed indeholde  $h(x)$  modulo  $p$ . En **change** operation kræver så logaritmisk mange genberegninger, som alle kan klares i konstant tid ved at vælge et primtal af passende størrelse.

For at begrænse fejl-sandsynligheden er vi nødt til at kende de størst mulige værdier i matricen  $h(x)$ . En grov vurdering viser, at indgangene i matricen  $h(x)$  er begrænset af  $4^n$ , se lemma 6.1. Det betyder, at der højst kan være  $n$  forskellige primtal  $p$ , hvor  $h(x) \equiv 1 \pmod p$  og der samtidig gælder at  $h(x) \not\equiv 1$ . Ved at vælge  $p \leq n^4$  tilfældigt, og for hver  $n$  operationer vælge et nyt  $p$  (se nedenfor), kan man garantere, at sandsynligheden for et fejlagtigt svar på et query i en sekvens af  $n$  operationer er begrænset af  $O(\frac{1}{n})$ . Denne begrænsning følger af det faktum, at der er af størrelsesorden  $\frac{n^4}{4 \log n}$  primtal mindre end  $n^4$ . Se JaJa [JaJ92] s. 453 for et præcist estimat.

□

Når man vælger et nyt primtal, er det nødvendigt, at genberegne alle knuderne i træet modulo dette. Det vil naturligvis tage for lang tid at gøre som en del af en enkelt operation, så til dette anvender vi *global rebuilding* teknikken af Overmars [Ove83].

Denne teknik anvendes til at opnå worst-case grænser i visse tilfælde, hvor man har en metode, som umiddelbart ville give average-case grænser. (Average-case grænser skal her forstås som et gennemsnit over et antal operationer). Ideen bag teknikken er som følger. Som regel betragter man en dynamisk struktur, som man kan udføre updates og queries på. Denne kan være balanceret fra starten, men

efter en række updates er strukturen kommet så meget ud af balance, at operationerne tager for lang tid. Man går derfor igang med at bygge en ny struktur som er balanceret, men istedet for at bygge den på én gang (hvilket vil give en average-case grænse) spreder man det ud over en række af de efterfølgende operationer. I mellemtiden udfører vi stadig updates på den gamle struktur, så denne kan svare på queries. De updates, som udføres mens vi er ved at opbygge den nye struktur, gemmes i en buffer. Når den nye struktur så er færdig, udfører vi disse updates på den, ved for eksempel at udføre to updates i den nye struktur for hver operation i den gamle. Det betyder, at den nye struktur på et tidspunkt vil "indhente" den gamle. Den nye struktur er så klar til at tage over, når bufferen er tom.

I vores tilfælde vil det sige, at vi opbygger et træ svarende til et nyt  $p$  i baggrunden, i stedet for at standse op og beregne det for hver  $n$  operationer i parentesstrukturen. I denne model er det faktisk så simpelt, at man i hver knude i træet kan have to matricer: en for det gamle primtal og en for det nye. Man skifter så fra den ene mængde af matricer til den anden, når alle de nye matricer er beregnet.

**LEMMA 6.1.** *Givet en følge af  $2 \times 2$  matricer  $A_1, \dots, A_n$  over  $Z$  med indgange begrænset af 2. Da er indgangene i matricen  $M = A_1 \cdot A_2 \cdots A_n$  begrænset af  $4^n$ .*

*Bevis.* Lad  $M_{max}$  betegne den største indgang i matricen  $M$ . Lad  $M = A_1 \cdot A_2 \cdots A_n$ , da gælder

$$M_{max} \leq 2^{2n-1}.$$

Dette vises ved induktion i  $n$ . Basis holder, da  $M_{max} \leq 2 * 2 * 2 = 2^3$ , for  $M = A_1 \cdot A_2$ . Antag nu, at påstanden gælder for  $n \leq k$ . Lad  $M' = A_1 \cdot A_2 \cdots A_k$  og  $M = M' \cdot A_{k+1}$ . Da gælder:  $M_{max} \leq 2 * 2 * M'_{max} \leq 2 * 2 * 2^{2k-1} = 2^{2k+1} = 2^{2(k+1)-1}$ . Til sidst bemærkes at  $2^{2n-1} < 4^n$ .

□

Algoritmen for  $D_k$  er som sagt mere kompliceret, og anvender rent faktisk Monte Carlo algoritmen for  $D'_k$  til at løse delproblemer. Selve strukturen bliver også anvendt i den deterministiske algoritme for  $D_k$ .

**SÆTNING 6.2. member $_{D_k}$**  *kan klares i tid  $O(\log^2 n)$  pr. operation, så sandsynligheden for et forkert svar på et query i en sekvens af  $n$  operationer er  $O(\frac{1}{n})$ .*

*Bevis.* Lad  $x$  være den instans af  $\Sigma^n$  vi arbejder på. Igen opretholder vi et balanceret binært træ, hvor det  $i$ 'te blad repræsenterer  $x_i$ , og hver intern knude repræsenterer sammensætningen af dens sønners strenge.

Vi indfører endnu en afbildning  $\mu : \Sigma^* \rightarrow \Sigma^*$ , hvor enhver type venstreparentes annullerer enhver type højreparentes

$$\begin{aligned} \mu(\epsilon) &= \epsilon, \\ \mu(xa_i) &= \mu(x)a_i \quad \text{for alle } i, \quad 1 \leq i \leq k, \\ \mu(x\bar{a}_i) &= \begin{cases} \mu(x)\bar{a}_i & \text{hvis } \mu(x) \notin \Sigma^*A, \quad 1 \leq i \leq k, \\ x' & \text{hvis } \mu(x) \in x'A \quad 1 \leq i \leq k. \end{cases} \end{aligned}$$

For hvert  $y \in \Sigma^*$  kan vi skrive  $\mu(y)$  som  $y_{\bar{A}}y_A$  hvor  $y_{\bar{A}} \in \bar{A}^*$  og  $y_A \in A^*$ . I træ-knuden, der repræsenterer  $y$ , gemmer vi en bit, der er sand hvis og kun hvis  $\mu_1(y) \in \bar{A}^*A^*$ , altså  $\mu_1(y) = \mu(y)$ . Bemærk at  $\mu_1$  er den funktion der matcher ensidigt. Ideen er, at hvis denne bit er falsk, kan  $x$  ikke balancere da

$$\mu_1(y) \in \bar{A}^* A^* \text{ hvis og kun hvis } \exists u, v : uyv \in D_k.$$

Lad  $u$  og  $v$  betegne strengene repræsenteret af en knudes sønner, henholdsvis en venstre- og en højre-søn, og lad  $\mu(u) = u_{\bar{A}} u_A$  og  $\mu(v) = v_{\bar{A}} v_A$ . Antag at  $|u_A| \geq |v_{\bar{A}}|$  (det andet tilfælde er symmetrisk). Skriv  $u_A$  som  $u_{A,1} u_{A,2}$  hvor  $|u_{A,2}| = |v_{\bar{A}}|$ . Bemærk at  $y_A = u_{A,1} v_A$  og  $y_{\bar{A}} = u_{\bar{A}}$ . Lad  $w = u_{A,2} v_{\bar{A}}$ .

$\mu(u) = u_{\bar{A}} u_{A,1} u_{A,2}$	$\mu(v) = v_{\bar{A}} v_A$
$\mu(y) = u_{\bar{A}} u_{A,1} v_A = y_{\bar{A}} y_A$	$w = u_{A,2} v_{\bar{A}}$

FIGUR 6.1. Opbygning af knuden for strengen  $y$  ud fra dens to sønner med strengene  $u$  og  $v$ .

Ideen er nu, at vi i knuden for strengen  $y$  ønsker at håndtere de parenteser, som matches på dette niveau (strengen  $w$  ovenfor). Det betyder, at vi har to strenge af samme længde ( $u_{A,2}$  og  $v_{\bar{A}}$ ) på formen  $A^*$  og  $\bar{A}^*$ , som skal matches. Det svarer til at sammenligne de to strenge  $u_{A,2}$  og  $v_{\bar{A}}$  element for element, og se om de er ens. Det er dette check, som bruges til at sætte bit'en beskrevet ovenfor. Hvis bit'en er sand i alle knuderne i træet, og roden ikke indeholder umatchede parenteser, ved vi, at  $x \in D_k$ .

Ovenstående problem kan løses dynamisk på flere måder. I en Monte Carlo model kan man beregne fingeraftryk af de to strenge og sammenligne disse. En deterministisk algoritme til at sammenligne disse strenge er mere kompliceret, og vi viser senere, hvordan dynamisk streng-sammenligning af Mehlhorn, Sundar og Uhrig [MSU94] kan bruges til dette.

Vi vælger her at sammenligne de to strenge ved hjælp af algoritmen for  $D'_k$ , der tager en fast vektor som instans. Da  $w = u_{A,2} v_{\bar{A}} \in A^* \bar{A}^*$ , ved vi at  $w \in D'_k$  hvis og kun hvis  $w \in D_k$ . I træ-knuden for strengen  $y$  gemmer vi nu følgende information

- En bit der fortæller, om  $\mu_1(y) \in \bar{A}^* A^*$ .
- Indekserne af  $y_A$ ,  $y_{\bar{A}}$  og  $w$  i strengen  $x$  repræsenteret i tre balancerede binære søgetræer ordnet efter indekset. Selve elementerne kan naturligvis slås op i  $x$ . Længderne af  $y_A$  og  $y_{\bar{A}}$  gemmes også.
- En vektor  $w_{\#} \in (\Sigma \cup \{\#\})^{|y|}$  defineret på følgende måde: da elementerne i  $w$  stammer fra  $y$ , kan vi skrive den som,  $w = y_{i_1} y_{i_2} \dots y_{i_l}$ , hvor  $1 \leq i_1 < i_2 < \dots < i_l \leq |y|$ .  $w_{\#}$  defineres nu som

$$w_{\#} = \#^{i_1-1} y_{i_1} \#^{i_2-i_1-1} y_{i_2} \#^{i_3-i_2-1} \dots \#^{i_l-i_{l-1}-1} y_{i_l} \#^{|y|-i_l}.$$

$w_{\#}$  har altså fast længde, og svarer til de parenteser, der eventuelt matches ud på dette niveau "padded" med  $\#$  (se figur 6.2).

Lad os nu betragte operationerne. Medlemskab  $x \in D_k$  kan aflæses i roden, da

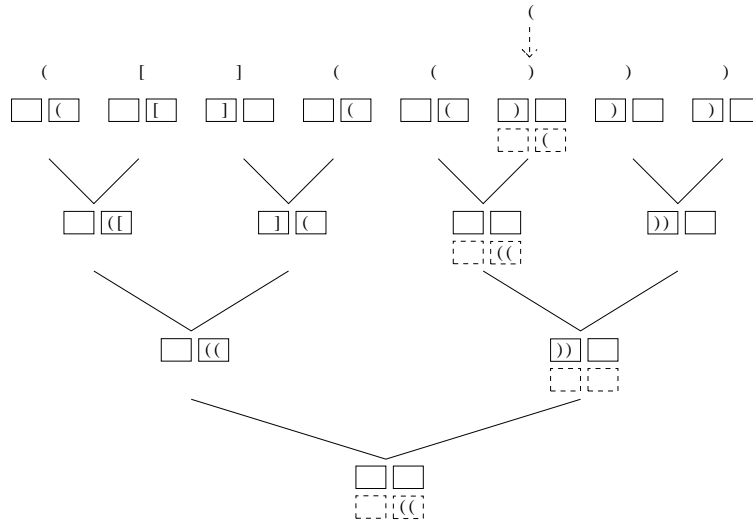
$$x \in D_k \text{ hvis og kun hvis } \mu_1(x) \in \bar{A}^* A^* \text{ og } x_A = x_{\bar{A}} = \epsilon.$$

En **change** operation er mere kompliceret og kræver en grundig analyse. Vi bruger som sagt Monte Carlo algoritmen for  $D'_k$  til at undersøge, om  $w \in D_k$ , da

$$w \in A^* \bar{A}^* \Rightarrow (w \in D'_k \Leftrightarrow w \in D_k).$$

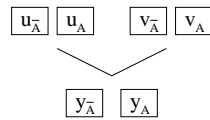
$w_{\#}$  bruges som instans, ved at lade  $h$  afbilde det ekstra tegn  $\#$  over i identitetsmatricen. Denne algoritme bruger tid  $O(\log n)$  for hver knude. Ovenstående bruges til





FIGUR 6.3. Monte Carlo datastruktur hvor strengen "(())" ændres til "(((())". De stiplede kasser indeholder  $y_A$  og  $y_{\bar{A}}$ , efter at strengen er ændret.

$y_{\bar{A}}$ . På samme måde vil én fjernet parentes i  $v_{\bar{A}}$  bevirke én ændring i enten  $y_A$  eller  $y_{\bar{A}}$ .



FIGUR 6.4. To niveauer i datastrukturen.

Hvis vi antager, at der højst er sket to ændringer i  $v_A$  og  $v_{\bar{A}}$  tilsammen, betyder det, at der højst sker to ændringer i  $y_A$  og  $y_{\bar{A}}$  tilsammen. Dette viser, at der kun sker et konstant antal ændringer på hvert niveau i træet. Ændringerne som hører til knuden for strengen  $y$  kan altså klares i tid  $O(\log n)$ , med kendskab til ændringerne på de lavere niveauer, da ændringerne foretages i balancerede træer. Dette giver den påståede tidsgrænse.

For at begrænse fejl-sandsynligheden, bemærk at vi bruger  $O(n)$  forskellige udgaver af datastrukturen fra Sætning 6.1. Ved at vælge et primtal fra en større mængde (for eksempel,  $p \leq n^5$ ) opnår man den ønskede grænse.

□



## Entydig dynamisk signatur af strenge

I forbindelse med de deterministiske algoritmer for  $D_k$  og  $D'_k$  har vi brug for at kunne sammenligne strenge af længde  $O(n)$  i en struktur, der ændrer sig dynamisk. Dette kapitel beskriver en forholdsvis ny teknik, der netop håndterer dette problem. Vi har tilføjet en ekstra operation, som er nødvendig i vores algoritmer.

Det vi ønsker, er at tilknytte en entydig signatur (et fingeraftryk) til strenge over et givet alfabet. To ens strenge skal naturligvis have den samme signatur, og signaturerne skal være så små, at de kan sammenlignes i konstant tid. Dette kan umiddelbart virke som et simpelt problem, men når strengene håndteres dynamisk kræves en god ide. Teknikken er beskrevet af Mehlhorn, Sundar og Uhrig [MSU94]. De operationer vi har brug for ser ud på følgende måde.  $S$  er universet af strenge.

**init:** Initialiserer strukturen.

**create( $a$ ):** Skaber en ny streng  $s = a \in \Sigma$ , og indsætter den i  $S$ .

**equal( $s_1, s_2$ ):** Returnerer **sand**, hvis og kun hvis  $s_1 = s_2$ .

**change( $s, i, a$ ):** Ændrer det  $i$ 'te element i strengen  $s$  til  $a \in \Sigma$ .

**concat( $s_1, s_2$ ):** Skaber en ny streng  $s_3 = s_1s_2$ , som indsættes i  $S$ .

**split( $s, i$ ):** Skaber to nye strenge  $s_1 = s_{[1\dots i]}$  og  $s_2 = s_{[i+1\dots |s|]}$ , som indsættes i  $S$ .

**lcp( $s_1, s_2$ ):** Returnerer længden af det største fælles prefix af strengene  $s_1$  og  $s_2$ .

Ideen er følgende. Strengen  $s$  inddeles i blokke af mindst to elementer, som hver tilknyttes en signatur. En ordbog sørger for, at ens blokke tilknyttes ens signaturer. Strengen  $s$  er nu reduceret til en streng af signaturer, der er mindre end halv så lang som  $s$ . Proceduren gentages så rekursivt på denne streng, indtil  $s$  er reduceret til en enkelt signatur. Det komplicerede i denne teknik er blok-inddelingen. To ens strenge skal have ens blok-inddelinger for at få samme signatur. Det lyder måske oplagt, men man skal huske på, at de to strenge kan være skabt på vidt forskellige måder. For at kunne håndtere disse blokke dynamisk er det nødvendigt, at en ændring i en streng, kun må ændre blokkene i en "lille" omegn af denne ændring. Løsningen på dette er at bestemme blok-inddelingen lokalt. Til dette bruges en deterministisk tre-farvnings teknik, der netop har denne lokale egenskab.

### 7.1. Tre-farvnings strategien

Denne teknik, som beskrevet i [MSU94], er baseret på en algoritme til tre-farvning af træer (Goldberg, Plotkin og Shannon [GPS87]), der igen er en generalisering af den såkaldte *Deterministic coin tossing* teknik af Cole og Vishkin [CV86c]. Teknikken blev oprindeligt brugt i forbindelse med parallelle algoritmer, hvor symmetri-brydning ofte er en nødvendighed, som man før kun kunne opnå med randomiserede algoritmer. Man ledte derfor efter en deterministisk metode, og det lykkedes blandt

andre Karp, Wigderson [KW84] og Luby [Lub85] at derandomisere flere af teknikkerne, hvilket dog betød at de ikke længere var optimale.

Deterministic coin tossing teknikken har for eksempel været brugt til at opnå optimale  $O(\log n)$  deterministiske EREW PRAM algoritmer for list ranking (Cole og Vishkin [CV86a]; Anderson og Miller [AM88]). For yderligere anvendelser af teknikken inden for parallelle algoritmer se Cole og Vishkin [CV86b].

Lad  $s = a_1 \dots a_n$  være en streng med  $a_i \in [0 \dots N - 1]$  og  $a_i \neq a_{i+1}$ . En  $k$ -farvning af en streng er en afbildning  $C : \{a_1, \dots, a_n\} \rightarrow \{0, \dots, k - 1\}$ . En gyldig farvning er en farvning, hvor ingen to sammenhængende elementer har samme farve.

Uformelt gør vi følgende. Fra start har vi en gyldig  $N$  farvning. Derefter erstatter vi elementerne i strengen med deres farve, og betragter mængden af farver som det nye univers. Vi betragter farverne som bitstreng, og finder den første position i denne streng hvor et elements farve  $C$  adskiller sig fra det forrige elements. Farven sættes nu til to gange denne position, som er af størrelse  $O(\log C)$ , plus værdien af denne bit. Dette sikrer en gyldig farvning. Proceduren gentages, og efter  $O(\log^* N)$  iterationer har vi en gyldig seks-farvning, som vi reducerer til en tre-farvning med en anden procedure.

Identificer hvert  $a_i$  (og dets farve) med dets binære repræsentation. Bit'ene er nummereret fra 0 (fra højre), og den  $j$ 'te bit af repræsentationen af farven for  $a_i$  skrives  $C_i(j)$ . Følgende algoritme har strengen  $s = a_1 \dots a_n$  som input og beregner en gyldig seks-farvning af  $s$ . Se figur 7.2 for et eksempel. Vi bruger  $C_i$  til at betegne farven af  $a_i$ .  $N_c$  betegner antallet af brugte farver. Se figur 7.1.

```

Procedure Seks-Farvning( $a_1 \dots a_n$ ) ;
begin
   $N_c \leftarrow N$  ;
  forall  $i \in \{1, \dots, n\}$  do
     $C_i \leftarrow a_i$  ;
  od ;
  while  $N_c > 6$  do
     $C'_1 \leftarrow C_1(0)$  ;
    forall  $i \in \{2, \dots, n\}$  do
       $j_i \leftarrow \min\{j \mid C_i(j) \neq C_{i-1}(j)\}$  ;
       $C'_i \leftarrow 2j_i + C_i(j_i)$  ;
    od ;
     $C \leftarrow C'$  ;
     $N_c \leftarrow \max\{C_i \mid i \in \{1, \dots, n\}\} + 1$  ;
  od ;
end ;

```

FIGUR 7.1. Seks-farvnings proceduren

LEMMA 7.1. *Proceduren Seks-Farvning producerer en gyldig seks-farvning af en streng  $a_1 \dots a_n$  med  $a_i \in [0 \dots N - 1]$  og  $a_i \neq a_{i+1}$  for alle  $1 \leq i \leq n$ . Proceduren bruger tid  $O(n \log^* N)$ .*

Seks-Farvning				Tre-Farvning			
111	1101111	0000001	0000001	1	1	1	1
55	0110111	0000110	0000000	0	0	0	0
31	0011111	0000111	0000001	1	1	1	1
123	1111011	0000100	0000000	0	0	0	0
27	0011011	0001010	0000011	3	2	2	2
15	0001111	0000101	0000001	1	1	1	1
109	1101101	0000010	0000000	0	0	0	0
95	1011111	0000011	0000001	1	1	1	1
47	0101111	0001000	0000000	0	0	0	0
39	0100111	0000110	0000011	3	1	1	1
43	0101011	0000100	0000010	2	2	2	2
47	0101111	0000101	0000001	1	1	1	1
31	0011111	0001001	0000100	4	4	2	2
119	1110111	0000110	0000000	0	0	0	0
123	1111011	0000100	0000010	2	2	2	2
86	1010110	0000000	0000100	4	4	0	0
42	0101010	0000100	0000101	5	5	5	1
	$N_c = 14$	$N_c = 10$	$N_c = 6$	$N_c = 6$	$N_c = 5$	$N_c = 4$	$N_c = 3$

FIGUR 7.2. Eksempel på tre-farvning af streng.  $N_C$  er antallet af brugte farver på hvert niveau.

*Bevis.* Lad os først vise, at proceduren beregner en gyldig farvning. Bemærk at farvningen inden while-løkken er gyldig, da  $a_i \neq a_{i+1}$ . Betragt nu to sammenhængende elementer  $a_i$  og  $a_{i+1}$ . I forall-løkken vælger  $a_i$  den mindst betydende bit  $j_i$ , hvor dens farve  $C_i$  er forskellig fra  $C_{i-1}$ . På tilsvarende vis vælges  $j_{i+1}$ . Hvis  $j_i \neq j_{i+1}$  bliver de to farver forskellige. Hvis de er ens, har vi, at  $C_{i+1}(j_i) \neq C_i(j_i)$  og igen bliver de to farver forskellige. Efter while-løkken har vi altså en gyldig farvning.

Lad os nu finde en øvre grænse for antallet af iterationer. Lad  $L = \lceil \log N \rceil$ , og lad  $L_k$  betegne antallet af bit's i den længste repræsentation af en farve efter  $k$  iterationer. Da gælder  $L_k \leq \lceil \log^k L \rceil + 2$ , hvis  $\lceil \log^k L \rceil \geq 2$ . Dette vises ved induktion i  $k$ . Vi har  $L_1 \leq \lceil \log L \rceil + 1$ , da man i forall-løkken har at  $j_i \leq L$ . Basis holder altså. Antag nu at påstanden holder op til og med  $k - 1$ . Skridtet holder nu da  $\lceil \log^k L \rceil \geq 2$  medfører at  $\log^{k-1} L > 2$ . Det betyder at

$$\begin{aligned}
 (1) \quad L_k &\leq \lceil \log L_{k-1} \rceil + 1 \\
 (2) \quad &\leq \lceil \log(2 \log^{k-1} L) \rceil + 1 \\
 &\leq \lceil \log^k L \rceil + 2
 \end{aligned}$$

(1) ses, da  $j_i \leq L_{k-1}$ . (2) følger af induktions antagelsen. Efter  $\log^* N + 1$  iterationer har vi at  $\lceil \log^k L \rceil = 1$  og derfor  $L_k \leq 3$ . Det betyder, at der er tre mulige værdier for indekset  $j$  og to mulige værdier af bit'en  $C(j)$ . Derfor giver næste iteration en

seks-farvning, som ifølge ovenstående er gyldig. Da hver iteration tager tid  $O(n)$  følger tids-grænsen.

□

Man kan nu let beregne en gyldig tre-farvning ved at erstatte hver farve  $C_i \in \{3, 4, 5\}$  af et element  $a_i$  med den mindste farve i  $\{0, 1, 2\}$ , som ikke er tilknyttet en af dens naboer. Der er dog en lille fælde her. Man kan ikke nøjes med et sweep, hvor farverne  $\{3, 4, 5\}$  skiftes ud, som beskrevet ovenfor. Det ses let af en streng seks-farvet på formen  $C_i 3453543$ . Hvis  $C_i = 0$ , bliver strengen tre-farvet som 01010101, og omvendt hvis  $C_i = 1$ . Dette bryder med den lokale afhængighed, som vi har brug for.

I stedet kan man bruge en anden simpel procedure, der løber strengen igennem tre gange (en for hver farve  $\{3, 4, 5\}$ ). Se figur 7.3.

```

Procedure Tre-Farvning( $a_1 \dots a_n$ ) ;
begin
  Seks-Farvning( $a_1 \dots a_n$ ) ;
   $C_0 \leftarrow \infty$  ;
   $C_{n+1} \leftarrow \infty$  ;
  for  $c = 3$  to  $5$  do
    forall  $i \in \{1, \dots, n\}$  do
      if  $C_i = c$  then
         $C_i \leftarrow \min\{\{0, 1, 2\} - \{C_{i-1}, C_{i+1}\}\}$  ;
      fi ;
    od ;
  od ;
end ;

```

FIGUR 7.3. Tre-farvnings proceduren

**LEMMA 7.2.** *Proceduren Tre-Farvning producerer en gyldig tre-farvning af en streng  $a_1 \dots a_n$  med  $a_i \in [0 \dots N - 1]$  og  $a_i \neq a_{i+1}$  for alle  $1 \leq i \leq n$ . Proceduren bruger tid  $O(n \log^* N)$ .*

*Bevis.* Først beregnes en gyldig seks-farvning. Hver af de tre iterationer fjerner så en farve, ved at erstatte et elements farve med en ny forskellig fra dens to naboers. Farvningen er tydeligvis stadig gyldig. Tiden for løkken er  $O(n)$ , og den påståede grænse følger da af lemma 7.1.

□

## 7.2. Blok-inddeling

For en streng  $a_1 \dots a_n$  definerer vi nu strengen  $d_1 \dots d_n$ , hvor  $d_i \in \{0, 1\}$ , på følgende måde. Først beregner vi en gyldig tre-farvning af  $a_1 \dots a_n$  med proceduren Tre-Farvning. Betragt nu farverne som heltallene  $\{0, 1, 2\}$ . Vi sætter så  $d_i = 1$  hvis og kun hvis farven af  $a_i$  er et lokalt maksimum i strengen af farver, og  $d_i = 0$  ellers. Strengen  $d_1 \dots d_n$  bruges nu til at inddele strengen  $a_1 \dots a_n$  i blokke, ved at starte en ny blok alle de steder, hvor  $d_i = 1$ . Lemma 7.3 viser, at ingen blok indeholder mere end fire elementer, at alle blokke (undtagen måske den første og den sidste) indeholder mindst to elementer og at denne mærkning har den lokale egenskab vi har brug for.

LEMMA 7.3. *Givet en streng  $a_1 \dots a_n$ ; værdierne  $d_1 \dots d_n$  defineret ovenfor har følgende egenskaber*

1.  $d_i + d_{i+1} \leq 1$  for alle  $1 \leq i \leq n$ .
2.  $d_i + d_{i+1} + d_{i+2} + d_{i+3} \geq 1$  for alle  $1 \leq i \leq n - 3$ .
3. Værdien af  $d_i$  afhænger kun af del-strengen  $a_{i-\log^* N-6} \dots a_{i+4}$ .

*Bevis.* Da to farver af sammenhængende elementer er forskellige i en gyldig farvning, kan de ikke begge være lokale maksima, hvorfor den første egenskab følger.

Den anden egenskab ses, da enhver streng af fire sammenhængende elementer enten indeholder farven 2, som altid er et lokalt maksimum eller farve-følgen 010, hvor 1 er lokalt maksimum.

Vi beviser den tredje egenskab i flere skridt. Først viser vi ved induktion i antallet af iterationer af while-løkken i proceduren Seks-Farvning, at for hvert  $a_i$  afhænger den beregnede farve kun af del-strengen  $a_{i-\log^* N-2} \dots a_i$ . Formelt siger påstanden, at farven af  $a_i$  efter den  $k$ 'te iteration kun afhænger af del-strengen  $a_{i-k} \dots a_i$ . Ovenfor har vi lige vist, at  $k \leq \log^* N + 2$  (se lemma 7.1). Dette ses dog let. Før første iteration afhænger farven af  $a_i$  kun af  $a_i$  selv (vi ser naturligvis ikke på afhængighed mellem selve elementerne). Antag nu at for hvert  $1 \leq i \leq n$  afhænger farven af  $a_i$  efter  $(k-1)$  iterationer kun af del-strengen  $a_{i-k+1} \dots a_i$ . I næste iteration får elementet  $a_i$  en farve, der afhænger af  $C_i$  og  $C_{i-1}$ . Ifølge antagelsen afhænger  $C_i$  kun af del-strengen  $a_{i-k+1} \dots a_i$ , og  $C_{i-1}$  afhænger kun af del-strengen  $a_{i-k} \dots a_{i-1}$ . Det betyder, at den nye farve for  $a_i$  kun afhænger af del-strengen  $a_{i-k} \dots a_i$ , hvilket viser påstanden.

Derefter vises, at for hvert  $a_i$  afhænger farven beregnet af proceduren Tre-Farvning kun af del-strengen  $a_{i-\log^* N-5} \dots a_{i+3}$ . Dette ses igen ved induktion i antallet af iterationer i proceduren. I hver iteration afhænger den nye farve af et element kun af dens naboers farver, som ikke ændrer sig. Da der kun er tre iterationer, afhænger tre-farvningen af  $a_i$  kun af seks-farvningen af  $a_{i-3} \dots a_{i+3}$  og derfor af del-strengen  $a_{i-\log^* N-5} \dots a_{i+3}$ .

Til sidst kan man bemærke, at værdien af  $d_i$  kun afhænger af farverne  $C_{i-1}$ ,  $C_i$  og  $C_{i+1}$  og derfor af del-strengen  $a_{i-\log^* N-6} \dots a_{i+4}$ .

□

### 7.3. Hierarkisk signatur kodning

Tre-farvnings teknikken kan ikke anvendes direkte på vilkårlige strenge, da nabo-elementer skal være forskellige. For at løse dette problem, slår vi ens nabo-elementer sammen i *potens-elementer*. En streng på formen  $aabbbcbbb$  bliver derfor til  $a^2b^3c^1b^3$ .

Formelt defineres proceduren  $Compress_\tau(s)$ , hvis gentagne udførelse kan betragtes som et træ, hvor elementerne fra  $s$  gemmes i bladene, og roden indeholder den entydige signatur af strengen  $s$ . Vi skriver  $Compress_\tau^j(s)$  for  $Compress_\tau$  anvendt  $j$  gange på strengen  $s$ . Til proceduren er tilknyttet en ordbog  $\tau$ , der afbilder blade, potens-elementer og *blok-elementer* over i signaturer  $C$ . Formelt er  $\tau$  en injektiv partiel afbildning på formen

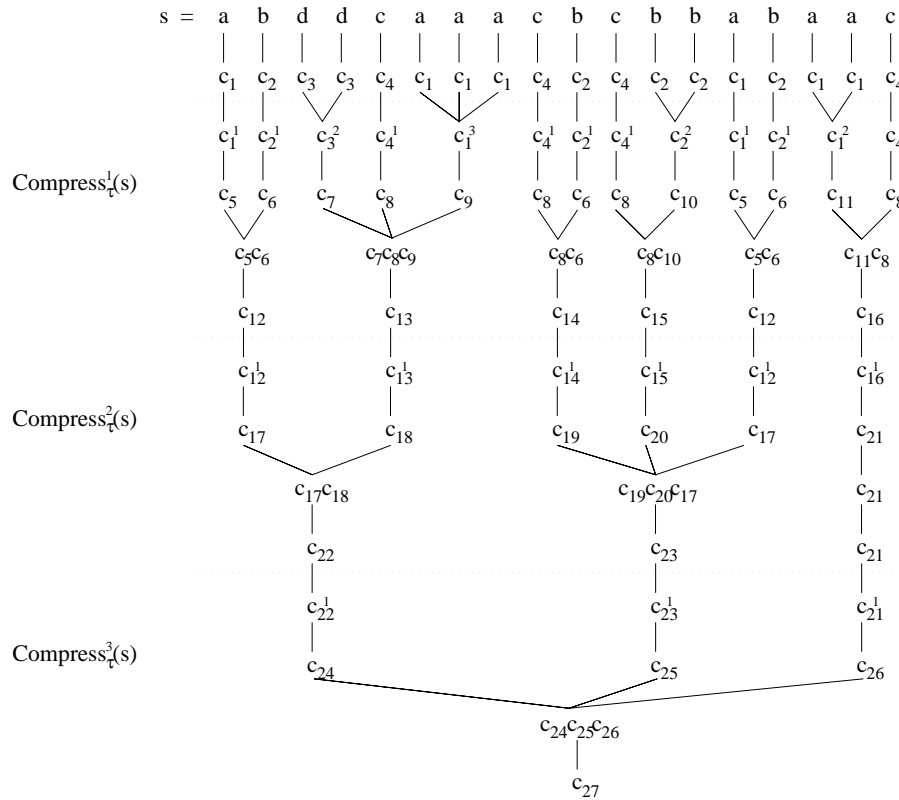
$$\tau : \{\Sigma \cup (C, \mathbb{N}) \cup C^2 \cup C^3 \cup C^4\} \hookrightarrow C.$$

$Compress_\tau(s)$  fjerner først gentagelser i strengen  $s$ , ved at slå ens elementer sammen i potens-elementer. Hvert af disse potens-elementer tilknyttes en entydig signatur. Derefter anvendes tre-farvnings proceduren til at inddele den nye streng i blokke. Hvert blok-element tilknyttes en entydig signatur. Se figur 7.4. Ved at bruge den

samme ordbog  $\tau$  ved kodning af en mængde af strenge, der kan være skabt vidt forskelligt, sikrer man, at ens strenge får ens signaturer, da træet på grund af den deterministiske blok-inddeling kun kan have en udformning.

$a \hookrightarrow c_1$	$b \hookrightarrow c_2$	$d \hookrightarrow c_3$	$c \hookrightarrow c_4$	$c_1^1 \hookrightarrow c_5$
$c_2^1 \hookrightarrow c_6$	$c_3^2 \hookrightarrow c_7$	$c_4^1 \hookrightarrow c_8$	$c_1^3 \hookrightarrow c_9$	$c_2^2 \hookrightarrow c_{10}$
$c_1^2 \hookrightarrow c_{11}$	$c_5 c_6 \hookrightarrow c_{12}$	$c_7 c_8 c_9 \hookrightarrow c_{13}$	$c_8 c_6 \hookrightarrow c_{14}$	$c_8 c_{10} \hookrightarrow c_{15}$
$c_{11} c_8 \hookrightarrow c_{16}$	$c_{12}^1 \hookrightarrow c_{17}$	$c_{13}^1 \hookrightarrow c_{18}$	$c_{14}^1 \hookrightarrow c_{19}$	$c_{15}^1 \hookrightarrow c_{20}$
$c_{16}^1 \hookrightarrow c_{21}$	$c_{17} c_{18} \hookrightarrow c_{22}$	$c_{19} c_{20} c_{17} \hookrightarrow c_{23}$	$c_{22}^1 \hookrightarrow c_{24}$	$c_{23}^1 \hookrightarrow c_{25}$
$c_{21}^1 \hookrightarrow c_{26}$	$c_{24} c_{25} c_{26} \hookrightarrow c_{27}$			

FIGUR 7.4. Ordbog  $\tau$  hørende til signaturtræ.



FIGUR 7.5. Eksempel på signaturtræ.

Det vi nu ønsker at vide er, hvor meget signaturtræet for en streng ændres når man ændrer et enkelt element i selve strengen. Hvis man i første omgang ser bort fra ens nabelementer, er problemet naturligvis blok-inddelingen, der igen er baseret på tre-farvnings proceduren. Som vist i lemma 7.3 afhænger mærkningen af et element højst af de foregående  $O(\log^* N)$  elementer og et konstant antal efterfølgende. Det betyder, at blok-inddelingen i en tilsvarende omegn kan ændre sig. Når elementerne

så er slået sammen i blokke af mindst to elementer på det næste niveau i signaturtræet, er den omegn, som måske har ændret sig, mere end halveret. Problemet er nu, at når vi tre-farver signaturkodningen af disse blokke, får vi igen tilføjet en  $O(\log^* N)$  omegn og det virker derfor som om, den ændrede omegn vokser på hvert niveau. Intuitivt kan man se, at dette ikke er tilfældet ud fra rækken

$$\log^* n \rightarrow \frac{1}{2} \log^* n + \log^* n \rightarrow \frac{3}{4} \log^* n + \log^* n \rightarrow \frac{7}{8} \log^* n + \log^* n \rightarrow \dots$$

som er begrænset af  $2 \log^* n$ . Formelt vises det af følgende meget tekniske lemma.

Vi betragter en sammensætning af to strenge og viser, at store dele af de to strenges signaturtræer kan "genbruges" i det nye træ, og kun en  $O(\log^* N)$  omegn af snittet skal beregnes på hvert niveau, for at skabe det nye træ. Vi siger, at en knude  $u$  i træet *spænder over* en anden, hvis denne er indeholdt i  $u$ 's undertræ.

**LEMMA 7.4.** *Lad  $s_1 = a_1 \dots a_l$ ,  $s_2 = a_{l+1} \dots a_n$  og  $s = s_1 s_2$  være strenge og  $j \geq 0$  et heltal. Lad  $\text{Compress}_\tau^j(s) = c_1 \dots c_r$  og lad  $i$  være sådan at  $c_i$  spænder over den del-streng, som indeholder  $a_l$ . Da gælder følgende*

1.  $c_1 \dots c_{i-6}$  er et prefix af  $\text{Compress}_\tau^j(s_1)$  og  $|\text{Compress}_\tau^j(s_1)| \leq i + 13$ .
2.  $c_{i+\log^* N+10} \dots c_r$  er et suffix af  $\text{Compress}_\tau^j(s_2)$  og  $|\text{Compress}_\tau^j(s_2)| \leq r - i + \log^* N + 11$ .

*Bevis.* Den første del af punkt 1 vises ved induktion i  $j$ . Basis er oplagt, da  $\text{Compress}_\tau^0(s) = s$  for alle strenge  $s$ . Antag nu, at det holder op til og med et  $j \geq 0$ . Lad  $\text{Compress}_\tau^j(s) = C_s = c_1 \dots c_i \dots c_r$ , hvor  $c_i$  spænder over  $a_l$ . Inddelingen af  $c$  i potens-elementer betegner vi

$$P_s = p_1^{l_1} \dots p_{i_p}^{l_{i_p}} \dots p_{r_p}^{l_{r_p}}, \text{ hvor } p_{i_p}^{l_{i_p}} \text{ spænder over } c_i,$$

for  $l_1, \dots, l_{r_p} \in \mathbb{N}$ . Hvert af disse potens-elementer tilknyttes en signatur, og denne streng betegner vi

$$C''_s = c''_1 \dots c''_{i_p} \dots c''_{r_p}, \text{ hvor } c''_{i_p} \text{ spænder over } p_{i_p}^{l_{i_p}}.$$

Denne streng inddeles i blokke ved hjælp af tre-farvnings teknikken, og disse blokke betegnes

$$B_s = b_1 \dots b_{i'} \dots b_{r'}, \text{ hvor } b_{i'} \text{ spænder over } c''_{i_p}.$$

Til sidst tilknyttes hver af disse blokke en signatur, og den resulterende streng bliver på formen

$$\text{Compress}_\tau^{j+1}(s) = C'_s = c'_1 \dots c'_{i'} \dots c'_{r'}, \text{ hvor } c'_{i'} \text{ spænder over } b_{i'}.$$

Som det fremgår af konstruktionen spænder  $c'_{i'}$  over den del-streng, som indeholder  $a_l$ . Induktionen giver, at  $c_1 \dots c_{i-6}$  er et prefix af  $\text{Compress}_\tau^j(s_1)$ . Det betyder, at beregningen af  $\text{Compress}_\tau^{j+1}(s_1)$  kan skrives på følgende måde

$$\text{Compress}_\tau^j(s_1) = C_{s_1} = c_1 \dots c_{i-6} e_1 \dots e_{r_e}.$$

Disse signaturer samles af følgende potens-elementer;

$$P_{s_1} = p_1^{l_1} \dots p_k^{l_k} g_1 \dots g_{r_g}, \text{ hvor } p_1^{l_1} \dots p_k^{l_k} \text{ er det største prefix fælles med } P_s,$$

der igen tilknyttes følgende signaturer:

$$C''_{s_1} = c''_1 \dots c''_k e''_1 \dots e''_{r_g}.$$

Disse inddeles i følgende blokke

$B_{s_1} = b_1 \dots b_{k'} d_1 \dots d_{r_d}$ , hvor  $b_1 \dots b_{k'}$  er det største prefix fælles med  $B_s$  som hver tilknyttes en signatur, og til sidst har vi

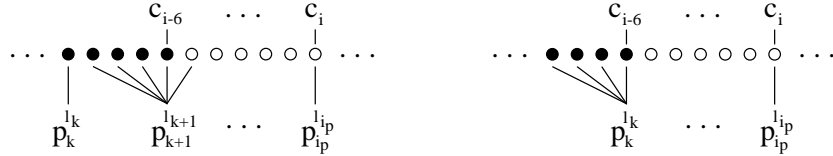
$$\text{Compress}_\tau^{j+1}(s_1) = C'_{s_1} = c'_1 \dots c'_{k'} e'_1 \dots e'_{r_d}.$$

Det ses nu let, at  $c'_1 \dots c'_{k'}$  er det størst mulige fælles prefix for  $\text{Compress}_\tau^{j+1}(s)$  og  $\text{Compress}_\tau^{j+1}(s_1)$ . For en mere grafisk fremstilling af de fælles prefix se figur 7.6.

$$\begin{aligned} C_s &= \boxed{c_1 \quad \dots \quad c_{i-6}} \dots c_i \quad \dots \quad c_r \\ C_{s_1} &= \boxed{c_1 \quad \dots \quad c_{i-6}} e_1 \quad \dots \quad e_{r_e} \\ \\ P_s &= \boxed{p_1^{i_1} \quad \dots \quad p_k^{i_k}} \dots p_{i_p}^{i_p} \quad \dots \quad p_{r_p}^{i_p} \\ P_{s_1} &= \boxed{p_1^{i_1} \quad \dots \quad p_k^{i_k}} g_1 \quad \dots \quad g_{r_g} \\ \\ C''_s &= \boxed{c''_1 \quad \dots \quad c''_k} \dots c''_{i_p} \quad \dots \quad c''_{r_p} \\ C''_{s_1} &= \boxed{c''_1 \quad \dots \quad c''_k} e''_1 \quad \dots \quad e''_{r_g} \\ \\ B_s &= \boxed{b_1 \dots b_{k'}} \dots b_{i_1} \quad \dots \quad b_r \\ B_{s_1} &= \boxed{b_1 \dots b_{k'}} d_1 \dots d_{r_d} \\ \\ C'_s &= \boxed{c'_1 \dots c'_{k'}} \dots c'_{i_1} \quad \dots \quad c'_r \\ C'_{s_1} &= \boxed{c'_1 \dots c'_{k'}} e'_1 \dots e'_{r_d} \end{aligned}$$

FIGUR 7.6. Fælles prefix under beregningen af  $\text{Compress}_\tau^{j+1}(s)$  og  $\text{Compress}_\tau^{j+1}(s_1)$ .

Idet vi husker på, at  $c'_i$  spænder over  $a_i$ , mangler vi blot at vise, at  $i' - k' \leq 6$ . Vi betragter derfor den størst mulige afstand mellem disse. Hvert af de seks elementer  $c_{i-5} \dots c_i$  kan være forskellige, og bliver derfor samlet af hvert deres potens-element. Det betyder, at  $i_p - k \leq 6$ . Figur 7.7 viser de to muligheder.

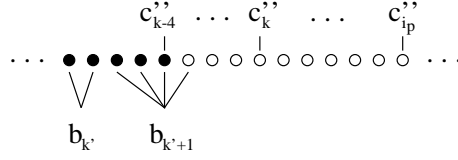


FIGUR 7.7. Eksempel på potens-streng for  $\text{Compress}_\tau^{j+1}(s)$  omkring  $c_{i-6}$ .

I følge lemma 7.3 afhænger blok-mærkningen af et element højst af de efterfølgende fire elementer. Det medfører, at mærkningen af  $c''_{k-3} \dots c''_k$  i beregningen af  $\text{Compress}_\tau^{j+1}(s)$  kan være forskellig fra mærkningen i beregningen af  $\text{Compress}_\tau^{j+1}(s_1)$ . Det betyder at blokkene, som spænder over  $c''_{k-4} \dots c''_k$  kan være forskellige. Se figur 7.8. Da hver blok spænder over mindst to elementer, ved vi at

$$i' - k' \leq \lceil (i_p - k + 5)/2 \rceil \leq 6.$$

Det viser, at  $c'_1 \dots c'_{i'-6}$  er et prefix af  $Compress_{\tau}^{j+1}(s_1)$ , hvilket viser den første del af punkt 1.



FIGUR 7.8. Eksempel på blok-streng for  $Compress_{\tau}^{j+1}(s)$  omkring  $p_k^{l_k}$ .

Vi vil nu vise anden del af punkt 1 i lemma'et. Det vises igen ved induktion i  $j$ . Beregningen af  $Compress_{\tau}^{j+1}$  for  $s$  og  $s_1$  beskrives med samme notation som før. I induktions-skridtet skal vi vise, at  $|Compress_{\tau}^{j+1}(s_1)| \leq i' + 13$  og derfor ønsker vi at betragte instanser af  $Compress_{\tau}^j(s)$  og  $Compress_{\tau}^j(s_1)$ , der giver den længst mulige  $Compress_{\tau}^j(s_1)$ , altså ingen ens nabo-elementer slået sammen i potens-elementer og kun blokke af størrelse to. Da lemma'et udtaler sig om længden af  $Compress_{\tau}^{j+1}(s_1)$  i forhold til  $i$ , har størrelsen af dette fælles prefix også betydning. Følgende lighed viser dette

$$(3) \quad |Compress_{\tau}^{j+1}(s_1)| = k' + r_d = i' + r_d - (i' - k') \leq i' + r_d.$$

Som før er  $r_d$  længden af det suffix af  $Compress_{\tau}^{j+1}(s_1)$ , der ikke er med i det fælles prefix. For at gøre beviset meget simplere vil vi se bort fra  $(i' - k')$ , da vi ved at  $0 \leq (i' - k') \leq 6$ , og kun søge at begrænse  $r_d$ . Det betyder en lidt dårligere konstant (fra 8 til 13), men det har ingen indflydelse på vores senere anvendelse af teknikken, da vi kun har brug for, at den er  $O(1)$ .

Basis er trivielt opfyldt. Antag nu, at

$$|Compress_{\tau}^j(s_1)| \leq i + 13$$

holder op til og med et  $j \geq 0$ . Det ses let, at  $r_d$  bliver maksimal, hvis vi betragter  $Compress_{\tau}^j(s_1)$  på formen

$$Compress_{\tau}^j(s_1) = c_1 \dots c_{i-6} e_1 \dots e_{6+13}.$$

Hvert af elementerne  $e_1 \dots e_{19}$  kan være forskellige, hvorfor de samles af hvert deres potens-element.  $c_{i-6}$  kan være lig med  $c_{i-5}$  i  $Compress_{\tau}^j(s)$ , hvilket betyder, at de er slået sammen i et potens-element. Dette potens-element er naturligvis ikke med i det fælles prefix, da  $c_{i-5}$  ikke er med i  $Compress_{\tau}^j(s_1)$ . I beregningen af  $Compress_{\tau}^{j+1}(s_1)$  bliver den længste potens-streng derfor

$$p_1^{l_1} \dots p_k^{l_k} g_1 \dots g_{20}$$

med tilhørende signatur-streng

$$c''_1, \dots, c''_k e''_1 \dots e''_{20}.$$

Da tre-farvningen kan afhænge af de fire efterfølgende elementer, kan mærkningen af  $c''_{k-3} \dots c''_k$  være forskellig fra  $Compress_{\tau}^{j+1}(s)$  til  $Compress_{\tau}^{j+1}(s_1)$ . Det betyder, at blokkene som spænder over  $c''_{k-4} \dots c''_k$  også kan være forskellige. Hvis vi lader det sidste element  $e''_{20}$  indgå i en blok alene, og resten af de blokke, der spænder

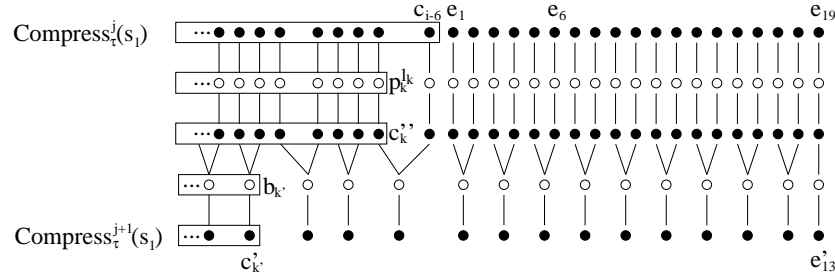
over  $c''_{k-4} \dots c''_k e''_1 \dots e''_{19}$  spænde over to elementer, bliver den længste mulighed for blok-strengen i beregningen af  $Compress_{\tau}^{j+1}(s_1)$  (se figur 7.9)

$$b_1 \dots b_{k'} d_1 \dots d_{13}.$$

Hver blok får tilknyttet en entydig signatur, så den længste mulighed for  $Compress_{\tau}^{j+1}(s_1)$  er

$$Compress_{\tau}^{j+1}(s_1) = c'_1 \dots c'_{k'} e'_1 \dots e'_{13}.$$

Da  $k' \leq i'$  har vi begrænset længden til  $i' + 13$ .



FIGUR 7.9. Den længste mulighed for  $Compress_{\tau}^{j+1}(s_1)$ . De indrammede elementer er fælles med  $Compress_{\tau}^j(s)$ .

Punkt 2 vises på helt samme måde, nu er blok-mærkningen blot afhængig af en  $\log^*$  omegn, hvilket gør analysen en smule mere kompliceret. For det fulde bevis se [MSU94].

□

#### 7.4. Datastrukturen

Selve strukturen er delt op i to: signaturtræer og en signaturordbog. Signaturordbogen skal tilknytte nye ord, som ikke findes i ordbogen i forvejen, en ny entydig signatur. Lad  $T(M)$  være den tid det tager at håndtere ordbogen for et univers af strenge  $S$ , hvor  $M$  er det totale antal af signaturer. Det viser sig senere, at  $M$  er begrænset af  $O(N)$ , hvor  $N$  er den samlede længde af alle strenge i  $S$ .

**7.4.1. Signaturtræet.** Signaturtræerne er bygget op som vist på figur 7.5. Hver knude har forbindelse både opad og nedad i træet, samt til begge naboer. Blokknuderne består blot af pointers til de knuder, de spænder over. Potensknuderne, der kan spænde over et vilkårligt antal knuder, er bygget op som balancerede binære træer ordnet efter indekset på de knuder, de spænder over. Hver knude i signaturtræet indeholder desuden information om, hvor mange blade den spænder over.

Datastrukturen til håndtering af signaturtræerne skal som nævnt underbygge følgende operationer i polylogaritmisk tid

**init:** Initialiserer strukturen.

**create( $a$ ):** Skaber en ny streng  $s = a \in \Sigma$ , og indsætter den i  $S$ .

**equal( $s_1, s_2$ ):** Returnerer **sand**, hvis og kun hvis  $s_1 = s_2$ .

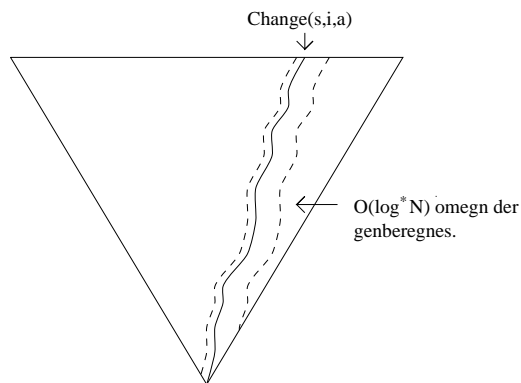
**change( $s, i, a$ ):** Ændrer det  $i$ 'te element i strengen  $s$  til  $a \in \Sigma$ .

**concat( $s_1, s_2$ ):** Skaber en ny streng  $s_3 = s_1 s_2$ , som indsættes i  $S$ .

**split**( $s, i$ ): Skaber to nye strenge  $s_1 = s_{[1..i]}$  og  $s_2 = s_{[i+1..|s|]}$ , som indsættes i  $S$ .

**lcp**( $s_1, s_2$ ): Returnerer længden af det største fælles prefix af strengene  $s_1$  og  $s_2$ .

**init** og **equal** klares i konstant tid. **create**( $a$ ) kan trivielt klares i tid  $O(T(M))$ . **change**( $s, i, a$ ) ændrer signaturtræet langs en sti fra det  $i$ 'te blad til roden. Ifølge lemma 7.4 er det nye træ magen til det gamle på nær en  $O(\log^* M)$  omegn af denne sti. Det betyder, at  $O(\log n \log^* M)$  knuder, som vi betegner  $K$ , i alt skal beregnes. Se figur 7.10. Alle knuder der blot indeholder en signatur, kan håndteres i tid  $O(T(M))$ . De kan altså sammenlagt klares i tid  $O(T(M) \log n \log^* M)$ . Vi betragter herefter to mængder af knuder, nemlig dem der indeholder et blok-element, og dem der indeholder et potens-element.



FIGUR 7.10. Eksempel på en opdatering i signaturtræet. De stiplede linier viser, hvilken omegn der skal genberegnes.

Lad os først se, hvad der sker på et niveau i signaturtræet, der består af blok-elementer. Vi antager, at niveauet ovenover er ændret. For at konstruere det nye træ er vi nødt til at beregne en ny tre-farvning omkring den sti, der kan have ændret sig. Bemærk at ifølge lemma 7.3 er det kun blokkene i en omegn af størrelse  $\log^* M$  plus en konstant  $c$ , der kan have ændret sig. Vi kender konstanten  $c$  og kan derfor anvende Tre-farvnings proceduren på denne omegn alene. Ifølge lemma 7.2 kan dette klares i tid  $O((\log^* M)^2)$ . Da signaturtræet har dybde  $O(\log n)$  kan alle knuderne fra  $K$ , der indeholder blok-elementer, beregnes i tid  $O(\log n (\log^* M)^2)$ .

For potens-elementerne på et niveau antager vi igen, at elementerne på niveauet ovenover i en  $O(\log^* M)$  omegn er ændret. Det betyder, at  $O(\log^* M)$  potens-elementer på dette niveau skal beregnes. Ved at "genbruge" gamle potens-elementer kan de nye potens-elementer beregnes i tid  $O(\log n \log^* M)$ . "genbruge" betyder her, at potens-elementer som overlapper kanten af  $O(\log^* M)$  omegnen ikke beregnes fra bunden.

Dette viser, at samtlige potens-elementer fra  $K$  håndteres i tid  $O(\log^2 n \log^* M)$ . Sammenlagt kan alle  $K$  knuder altså beregnes i tid

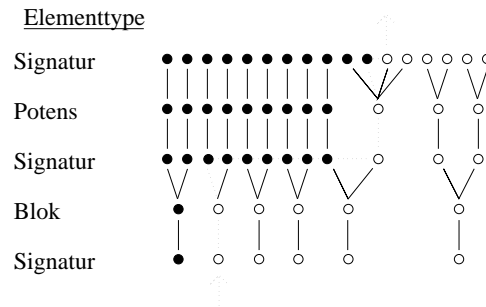
$$O(T(M) \log n \log^* M + \log n (\log^* M)^2 + \log^2 n \log^* M).$$

Ved **concat**( $s_1, s_2$ ) og **split**( $s, i$ ) skal en tilsvarende omegn beregnes, og vi får samme udførelsestid som for **change**( $s, i, a$ ). Ideen er her den samme: man genbruger

så vidt muligt det(de) gamle træ(er) og beregner kun den nye omegn. Vi vil ikke gå i detaljer med dem.

$\text{lcp}(s_1, s_2)$  operationen er begrebsmæssig simpel, men den indeholder et par detaljer, som man skal være opmærksom på. Da blokinddelingen afhænger af efterfølgende elementer på samme niveau i signaturtræet, kan man ikke blot lede efter et fælles undertræ. Ideen er, at vi først finder det største fælles (prefix) undertræ. Derefter søger vi op gennem den omegn, hvor de to træer kan være topologisk forskellige, som måske dækker over de samme elementer. Vi kalder de knuder, som er fælles for de to træers prefix, for  $P$ . Bemærk, at for alle knuder  $p$  i  $P$  gælder der, at de knuder  $p$  spænder over, også er med i  $P$ .

Vi starter med at betragte det yderste venstre blad ( $s_{i,[1]}$ ) i begge signaturtræer. Herfra bevæger man sig nedad mod roden i begge træer indtil den sidste knude, som er med i  $P$ . Fra denne begynder man nu at bevæge sig opad og til højre i træerne. (Man bevæger sig hele tiden ens i de to træer). Hvis man står i en knude, som er med i  $P$ , bevæger man sig mod højre, og hvis ikke bevæger man sig opad. Se figur 7.11 for et eksempel på et sådan gennemløb.



FIGUR 7.11. En prefix-søgning gennem et *Compress*-niveau, hvor den stiplede linie viser søgestien. De mørke knuder er ens for de to signaturtræer (med i  $P$ ).

Problemet består nu i at begrænse antallet af bevægelser til højre på et niveau i træerne. Invarianten for dette gennemløb er, at gennemløbet på hvert niveau går igennem den første knude (fra venstre), der ikke er med i  $P$ . Dette er opfyldt fra bunden af gennemløbet. Vi antager, at vi på niveauet for  $\text{Compress}_\tau^j$  befinder os i den første signaturknude (fra venstre), som ikke er med i  $P$ . Denne er markeret på figur 7.11 med en stiplede pil. Vi vil gerne vise, at vi kan finde den tilsvarende knude på niveauet for  $\text{Compress}_\tau^{j-1}$ , i  $O(\log k)$  skridt, hvor  $k$  er potensen for et potens-element, vi skal bevæge os igennem.

Først bevæger man sig op i en blokknude, som heller ikke er med i  $P$ . Fra denne bevæger man sig op i dens yderste venstre søn  $u$ . Hvis  $u$  ikke er med i  $P$ , fortsætter man blot opad til potensknuden ovenover. Hvis  $u$  er med i  $P$ , bevæger vi os mod højre, indtil vi når en knude, som ikke er med i  $P$ . Fra lemma 7.3 ved vi, at blokinddelingen højst afhænger af de efterfølgende fire elementer. Der kan altså højst være et konstant antal knuder til højre for  $u$ , som er med i  $P$ , da blokken, som spænder over  $u$ , ikke er med i  $P$ . Det betyder, at man højst skal bevæge sig et konstant antal knuder til højre på dette niveau, før man finder en knude  $v$ , som ikke er med i  $P$ .

Fra  $v$  bevæger man sig så op i den første potensknude (fra venstre), som ikke er med i  $P$ . Hvis disse potens-elementer (et for  $s_1$  og et for  $s_2$ ) spænder over forskellige typer elementer, bevæger man sig blot op i deres yderste venstre søn. Hvis de spænder over samme type elementer, bevæger man sig op i den søn, der svarer til den laveste potens af de to og herfra ét skridt til højre. Dette kan klares i tid  $O(\log n)$ . Vi befinder os nu i den første signaturknude (fra venstre), som ikke er med i  $P$  på niveauet for  $Compress_7^{j-1}$ .

Dette gennemløb vil ifølge ovenstående ende i det første element i strengene  $s_1$  og  $s_2$ , der ikke er med i det fælles prefix. Da knuderne i signaturtræerne indeholder information om, hvor mange elementer de spænder over, kan man med dette gennemløb let beregne længden af det største fælles prefix. Gennemløbet kan foretages i tid  $O(\log^2 n)$  sammenlagt for alle niveauerne. Operationerne på signaturtræerne kan altså håndteres i tiden

$$(4) \quad O(T(M) \log n \log^* M + \log n (\log^* M)^2 + \log^2 n \log^* M).$$

**7.4.2. Ordbogen.** Størrelsen af ordbogen  $\tau$  har selvfølgelig direkte indflydelse på udførelsestiderne i ovenstående struktur. Lemma 7.5 giver derfor en øvre grænse på antallet af signaturer, der kræves for at kode én streng.

**LEMMA 7.5.** *Lad  $s$  være en vilkårlig streng af længde  $n$ . Signatur-kodningen af  $s$  kræver højst  $4n$  signaturer.*

*Bevis.* Hvert element fra  $s$  tilknyttes en signatur. Derefter forgår kodningen ved først at tilknytte potens-elementerne en signatur, og derefter blok-elementerne. Den resulterende signatur-streng er højst halv så lang som den foregående, og proceduren gentages rekursivt. Følgende rekursionsligning beskriver da, hvor mange signaturer der højst kræves for at kode en streng af signaturer

$$\begin{aligned} \max_{sig}(1) &= 1, \\ \max_{sig}(n) &\leq n + \frac{n}{2} + \max_{sig}\left(\frac{n}{2}\right). \end{aligned}$$

Løsningen til denne er  $\max_{sig}(n) \leq 3n$ , der tilsammen med de første  $n$  signaturer giver det ønskede. □

Hvis vores streng-univers  $S$  har størrelse  $N$ , betyder det, at ordbogen skal håndtere  $M = O(N)$  signaturer. En simpel ordbog kunne være et balanceret træ, som medfører at  $T(M) = O(\log N)$ . Det betyder, at alle operationerne i Mehlhorn et al strukturen kan klares i tiden

$$(5) \quad O(\log^2 n \log^* N).$$



## Algoritmer for $D_k$ og $D'_k$ .

De deterministiske algoritmer for  $D_k$  og  $D'_k$  kan nu let beskrives, som en sammensætning af teknikker fra tidligere kapitler. Det eneste lille trick består i at repræsentere den inverse delstreng  $y^R$  sammen med  $y$  i knuderne.

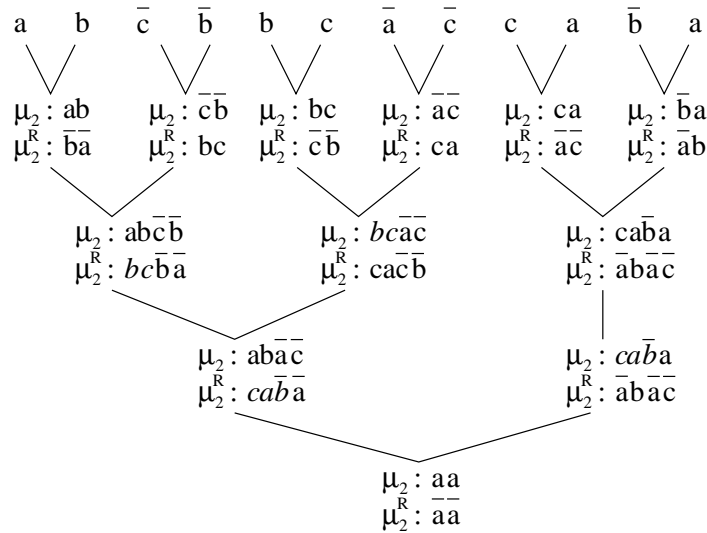
SÆTNING 8.1. **member** $_{D'_k}$  kan klares i tid  $O(\log^3 n \log^* n)$  pr. operation.

*Bevis.* Vi opretholder et balanceret binært træ, hvor det  $i$ 'te blad repræsenterer  $x_i$ , og hver intern knude repræsenterer sammensætningen af dens sønners strenge. I knuden, som repræsenterer for eksempel strengen  $y$ , gemmer vi  $\mu_2(y)$  og  $\mu_2(y^R)$ . Se figur 8.1. Til at håndtere disse strenge bruger vi datastrukturen fra kapitel 7. Bemærk at en query er simpel, da roden af træet indeholder  $\mu_2(x)$ . En **change**-operation ændrer træet på en sti fra det ændrede blad til roden. Følgende egenskab ved  $\mu_2$  ligger til grund for konstruktionen. Lad  $u, v \in \Sigma^*$  være på følgende form

$$(6) \quad \mu_2(u) = u'aw \quad \text{og} \quad \mu_2(v) = w^Rbv', \quad \text{med } \bar{a} \neq b,$$

for  $u', v', w \in \Sigma^*$  og  $a, b \in \Sigma$ . Da gælder, at

$$(7) \quad \mu_2(uv) = u'abv'.$$



FIGUR 8.1. Eksempel på træ for  $D'_3$ . De største fælles prefix er markeret med kursiv.

Betragt nu en knude, der repræsenterer strengen  $y$ . Lad dens højre- og venstre-søn repræsentere henholdsvis strengene  $u$  og  $v$ , og lad disse strenge være på samme form

som i lemma'et. Vi kan da konstruere strengene  $\mu_2(y)$  og  $\mu_2(y^R)$  med et konstant antal operationer. Først findes det længste fælles prefix af  $\mu_2(u^R)$  og  $\mu_2(v)$  ved hjælp af operationen **lcp**. På figur 8.1 er de længste fælles prefix markeret med kursiv. Et konstant antal **split** og **concat** operationer kan nu konstruere strengene  $\mu_2(y)$  og  $\mu_2(y^R)$  ud fra  $u, u^R, v$  og  $v^R$ .

For at sikre at antallet af signaturer i Mehlhorn et al signaturordbogen ikke bliver for stort, bruger vi global rebuilding til i baggrunden hele tiden at opbygge en ny struktur.

Lad  $N$  være den totale længde af alle strengene i dette træ. For at begrænse  $N$  kan man bemærke, at den samlede længde af strenge på hvert niveau i træet er  $O(n)$ , og derfor er  $N$  begrænset af  $O(n \log n)$ . En **change** operation kan da klares i tid  $O(\log^2 n \log^* n)$  i en enkelt knude, og derfor samlet i tid  $O(\log^3 n \log^* n)$ .

□

**SÆTNING 8.2.** **member** $_{D_k}$  kan klares i tid  $O(\log^3 n \log^* n)$  pr. operation.

*Bevis.* Til dette kan vi genbruge træ-strukturen fra Monte Carlo algoritmen for  $D_k$ . Ideen var, at man i hver knude repræsenterede tre strenge  $y_A, y_{\bar{A}}$  og  $w$ . Altså henholdsvis overskydende højre- og venstre-parenteser, samt de parenteser, der skulle matches på dette niveau. Problemet var, at man i hver knude i træet skulle undersøge, om to strenge ( $u_{A,2}$  og  $v_{\bar{A}}^R$ ) var ens. I stedet for en Monte Carlo algoritme ønsker vi nu at gøre dette ved hjælp af datastrukturen fra kapitel 7. Det betyder, at vi ikke gemmer strengene  $w_{\#}$ , der havde fast længde, og svarede til de parenteser, der skulle matches. Vi gemmer i stedet strengene  $u_{A,2}$  og  $v_{\bar{A}}^R$  i Mehlhorn et al strukturen.

Lad os betragte operationerne. Medlemskab af  $D_k$  kan igen aflæses direkte i roden. En **change**-operation påvirker træet i en sti fra det ændrede blad til roden. I hver knude på stien bevirker den som før højst fire ændringer til sammen i de to strenge, der skal sammenlignes. Dette kan gøres med et konstant antal operationer i Mehlhorn et al strukturen.

Håndteringen af disse strenge kan, med samme argumenter som ovenfor, klares i tid  $O(\log^2 n \log^* n)$ . Da træet har dybde  $O(\log n)$  får vi den samme grænse som i det to-sidige tilfælde:  $O(\log^3 n \log^* n)$ .

□

Følgende lidt specielle reduktion fortæller, at **match** $_{D_k}$  kan løses ved hjælp af **interval** $_{D_k}$  og **match** $_{D_1}$ . Dette betyder, sammenholdt med hierarkiet på figur 4.1, at en øvre grænse for **interval** $_{D_k}$  vil medføre øvre grænser for alle operationerne på de en-sidige Dyck-sprog i **change**-modellen.

**PROPOSITION 8.1.** **match** $_{D_k}$  kan klares i tid  $O(T_1(n) + T_2(n))$  pr. operation, hvor  $T_1(n)$  er tiden for at håndtere **interval** $_{D_k}$  og  $T_2(n)$  er tiden for at håndtere **match** $_{D_1}$ .

*Bevis.* Lad  $x \in (\{a_1, a_2, \dots, a_k\} \cup \{\bar{a}_1, \bar{a}_2, \dots, \bar{a}_k\})^n$  være vores input-instans. Til denne streng knytter vi en ny streng  $y$  med kun en type parentes defineret så enhver højre-parentes i  $x$  erstattes af  $a_1$  og enhver venstre-parentes erstattes af  $\bar{a}_1$ .  $y$  har

altså samme parentesstruktur som  $x$  men kun en type parentes. For eksempel

$$\begin{aligned} x &= a_3 a_4 \bar{a}_4 a_1 \bar{a}_1 \bar{a}_3 \\ &\Downarrow \\ y &= a_1 a_1 \bar{a}_1 a_1 \bar{a}_1 \bar{a}_1 \end{aligned}$$

Denne streng  $y$  håndteres af en struktur for  $D_1$ , med **match**-operationen. En **match**-operation på  $x_i$  klares nu ved at spørge om **match** på  $y_i$  i  $D_1$  strukturen. Antag at  $y_j$  matcher  $y_i$  ( $i < j$ ) i  $D_1$  strukturen. Det betyder, at intervallet  $y_{[i..j]}$  er balanceret i  $D_1$ . På grund af konstruktionen af  $y$ , må intervallet  $x_{[i..j]}$  også balancere modulo parentes-typerne. Man skal altså blot undersøge, om intervallet  $x_{[i..j]}$  er med i  $D_k$ . Hvis dette er opfyldt, vil  $x_j$  være det korrekte svar til **match**( $x_i$ ). Hvis det ikke er opfyldt, er svaret udefineret.

Denne konstruktion giver oplagt den ønskede grænse.

□



## Fredman og Saks' resultat

I dette kapitel gennemgår vi beviset for en nedre grænse for det beskrevne paritet-prefix problem. Beviset er som nævnt oprindeligt beskrevet i [FS89]. I vores præsentation har vi valgt at fokusere på worst-case forventet tid, og har således bortskåret de dele af beviset, der sikrede, at grænsen også holder for amortiseret kompleksitet.

Vores udgave adskiller sig herudover på to punkter. Beviset holder for en randomiseret algoritmes forventede tid for et query, og det simple lemma 9.1 erstatter 3-distance sætningen anvendt i [FS89]. Vi har til en vis grad bestræbt os på, at lægge centrale dele af notationen op ad original-udgavens for at lette en eventuel sammenligning.

Beregningsmodellen er i dette kapitel cell-probe modellen beskrevet i kapitel 3. Som omtalt i samme kapitel holder de beviste grænser også for unit-cost RAM modellen med samme ordstørrelse som cell-probe algoritmen.

**SÆTNING 9.1** (Fredman og Saks). *Lad  $r$  betegne det forventede antal af læse-operationer for et prefix-query for paritet-prefix problemet. Lad  $w$  betegne worst-case antallet af skrive-operationer for et update. Da gælder  $r \in \Omega(\frac{\log n}{\log(bw \log n)})$ , hvor  $b$  er cell probe-registrenes ordstørrelse.*

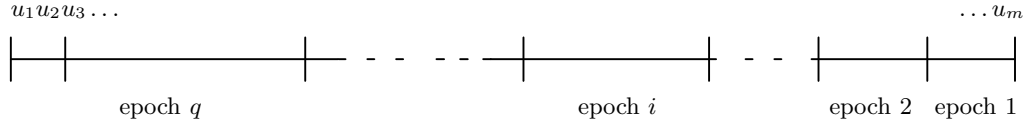
*Bevis.* Beviset består i, for en mængde af update-sekvenser, at analysere hvormange læse-operationer en deterministisk algoritms efterfølgende prefix-queries foretager gennemsnitligt, hvor gennemsnittet er over en stor mængde af update-sekvenser. Resultatet af denne analyse, kombineret med Yaos Minimax-princip sikrer os da den påståede nedre grænse for den forventede tid for en randomiseret algoritme.

Lad  $\hat{A}$  være en cell-probe algoritme for paritet-prefix problemet, hvor updates foretages i vektoren  $x = (x_1, x_2, \dots, x_n)$ . Vi antager, at algoritmen initielt har hukommelsen initialiseret, så  $x_i$  er 0 for alle  $i$ . Lad  $\mathcal{Q} = \{Q_1, \dots, Q_{2^m}\}$  betegne en mængde på  $2^m$  forskellige update-sekvenser, hvor  $m = \lfloor \frac{1}{6}n \rfloor$ . Hver sekvens  $Q_i$  består af  $m$  updates,  $u_1, \dots, u_m$ , på formen  $u_j = \text{change}(pos, val)$ . Positionen  $pos \in [1..n]$  er den samme for  $j$ 'te update i alle sekvenserne, mens værdierne ( $val \in \{0, 1\}$ ) er de eneste, der adskiller sekvenserne i  $\mathcal{Q}$  fra hinanden. Vi beskriver senere de konkrete værdier af disse størrelser.

Lad  $w$  betegne det maksimale antal skrive-operationer et update blandt sekvenserne  $\mathcal{Q}$  udfører. Lad  $r$  betegne det gennemsnitlige antal læse-operationer der udføres blandt prefix-queries efter sekvenserne  $\mathcal{Q}$ . Sagt med andre ord betegner  $r$  altså det forventede antal læse-operationer, cell-probe algoritmen udfører efter en uniformt tilfældigt valgt sekvens  $Q_i \in \mathcal{Q}$  og tilhørende prefix-query  $\text{prefix}(j)$ .

**Epoch-inddeling.** Vi inddeler hver sekvens  $Q_i \in \mathcal{Q}$  i  $q$  faser, vi kalder epochs. Epoch'ene ligger i "omvendt" orden i forhold til updates i  $Q_i$ , således at epoch 1 ligger til sidst i  $Q_i$ , og til venstre herfor følger epoch 2 frem til  $q$ . Se figur 9.1. Størrelsen af de forskellige epochs fastsættes således, at antallet af updates i epoch

1 til og med  $i$  er  $L_i = (\alpha w)^i$ . Antallet af epochs  $q$  vælges nu maksimalt så  $L_q \leq m$ . D.v.s.  $q = \lfloor \log_{\alpha w} m \rfloor^1$ .  $\alpha$  afhænger af  $n$  og  $b$  og vil blive fastsat til sidst i beviset. Intuitionen er, at  $\alpha$  skal sikre, at epoch-størrelserne vokser så hurtigt, at antallet af updates under epoch  $i$  er stort relativt til det samlede antal *skrive-operationer* under epoch  $i - 1$  til epoch 1. Dette sikrer, at effekten (på afsluttende queries) af updates under epoch  $i$  ikke kan afspejles udelukkende ved det "lille" antal skrive-operationer under de resterende operationer, og herigennem kan vi ved et tælleargument argumentere for, at et prefix-kald gennemsnitligt læser fra en vis fast brøkdel af de  $q$  epochs. Hvis parametrene  $b$  og  $w$  f.eks. er polylogaritmiske vil valget af  $\alpha$  bevirke, at  $q \in \Omega\left(\frac{\log n}{\log \log n}\right)$ , og dermed samme grænse for det gennemsnitlige antal læse-operationer for et prefix-kald.



FIGUR 9.1. Epoch-inddeling af en update-sekvens

**Update-sekvenserne.** En af de fundamentale ideer i beviset er, at updates fra hver epoch skal have en samlet stor "effekt" på afsluttende queries. Intuitivt sikres dette ved, at en update-position  $x$  ligger "langt" fra alle positioner, der efterfølgende foretages updates  $i$ .

Mere præcist er alle de  $2^m$  update-sekvenser opbygget så opdateringer blandt de  $L$  sidstliggende updates  $u_{(m-L)} \dots u_m$  opdaterer med afstand på mindst  $n/(3L)$ .

Idet vi betegner update-positionerne for  $u_m, u_{m-1}, \dots, u_1$  som henholdsvis  $p_1, p_2, \dots, p_m$  defineres disse ved:

$$\begin{aligned}
 p_1 &= \lfloor \frac{1}{2}n \rfloor \\
 p_2 &= \lfloor \frac{1}{4}n \rfloor \quad p_3 = \lfloor \frac{3}{4}n \rfloor \\
 p_4 &= \lfloor \frac{1}{8}n \rfloor \quad p_5 = \lfloor \frac{3}{8}n \rfloor \quad p_6 = \lfloor \frac{5}{8}n \rfloor \dots \\
 (8) \quad p_k &= \lfloor \frac{2(k - 2^{\lfloor \log k \rfloor}) + 1}{2^{\lfloor \log k \rfloor + 1}}n \rfloor
 \end{aligned}$$

For denne strategi gælder der:

LEMMA 9.1. *Afstanden imellem  $p_i$  og  $p_j$  for  $i \neq j$  og  $i, j \leq L \leq m$  er mindst*

$$\frac{n}{3L}.$$

*Bevis.* Betragt to punkter  $p_i$  og  $p_j$ ,  $i \neq j$  og  $i, j \leq L$ . Lad  $l$  være et maksimalt heltal så  $2^l \leq L$ . Ifølge (8) kan henholdsvis  $p_i$  og  $p_j$  skrives på formen (ved eventuelt at

<sup>1</sup> $\log_{\alpha w}$  betegner logaritmen med grundtal  $\alpha w$

forlænge brøken med en 2-potens):  $\lfloor \frac{q_i}{2^{l+1}} n \rfloor$  og  $\lfloor \frac{q_j}{2^{l+1}} n \rfloor$  for heltal  $q_i, q_j < 2^{l+1}$  og  $q_i \neq q_j$  (vises formelt ved induktion i  $L$ ). D.v.s.

$$\begin{aligned} |p_i - p_j| &= \left| \lfloor \frac{q_i}{2^{l+1}} n \rfloor - \lfloor \frac{q_j}{2^{l+1}} n \rfloor \right| \\ &\geq \frac{|q_i - q_j|}{2^{l+1}} n - 1 \\ &\geq \frac{n}{2^{l+1}} - 1 \\ &\geq \frac{n}{2L} - 1 \\ &\geq \frac{n}{3L} \end{aligned}$$

Den sidste ulighed følger af at  $L \leq m \leq \frac{1}{6}n$ .

□

Update-sekvenserne er nu defineret så de udspænder alle de  $2^m$  mulige sekvenser  $Q$ , der opdaterer på de ovennævnte positioner. Formelt lader vi  $\pi : [1..2^m] \rightarrow \{0, 1\}^m$  være en vilkårlig fastholdt bijektion. Update-sekvenserne defineres nu relativt til  $\pi$  så sekvens  $Q_i$ 's  $k$ 'te update er defineret ved:

$$u_k = \mathbf{change}(p_{(m-k+1)}, \pi(i)_k),$$

hvor  $\pi(i)_k$  er  $k$ 'te indgang i vektoren  $\pi(i)$ . Vi kalder  $\pi(i)$  update-mønstret for  $Q_i$ . Vi vil ofte få brug for at tale om updates knyttet til et mønster,  $\tau \in \{0, 1\}^*$ , kortere end  $m$ , defineret som:

$$\mathbf{change}(p_{(m)}, \tau_1), \mathbf{change}(p_{(m-1)}, \tau_2), \dots, \mathbf{change}(p_{(m-|\tau|)}, \tau_{|\tau|})$$

og betegnet  $U(\tau)$ . Sagt på en anden måde er  $U(\tau)$  et prefix af enhver sekvens  $Q_i$ , hvor der eksisterer et  $\mu \in \{0, 1\}^{m-|\tau|}$  så  $\pi(i) = \tau\mu$ . Bemærk at  $Q_i = U(\pi(i))$ .

**Query-vektorene.** I det følgende vil vi formalisere hvad vi mener med "effekten" af et update på de afsluttende query-svar. I den sags tjeneste er det nødvendigt at introducere nogle redskaber fra den lineære algebra.

Vi vil arbejde med vektorrum med koordinater og skalarer fra legemet med to elementer;  $\text{GF}(2)$  ( $\mathbb{Z} \pmod{2}$ ). For det  $n$ -dimensionale vektorrum  $V = \{0, 1\}^n$  vil vi tale om afstanden imellem to vektorer  $v, w \in V$ , givet ved  $|v - w| = |\{i | v_i \neq w_i\}|$ .  $\text{span}(w_1, \dots, w_k)$  betegner som sædvanligt vektorrummet udspændt af  $w_1, \dots, w_k$ , d.v.s.  $\{\lambda_1 w_1 + \dots + \lambda_k w_k | \lambda_i \in \text{GF}(2)\}$ .  $\text{span}(W) + c$  betegner det affine vektorrum  $\{w + c | w \in \text{span}(W)\}$ . Vi vil benytte notationen  $x_1 x_2 \dots x_n$  (uden parenteser og kommaer) for en vektor i  $\{0, 1\}^n$ , hvor  $x_i \in \{0, 1\}$ .

Vi vil ofte omtale query-svar i termer af en query-vektor i  $\{0, 1\}^n$ :

$$\mathbf{prefix}(1)\mathbf{prefix}(2) \dots \mathbf{prefix}(n),$$

hvor **prefix**-kaldene sker umiddelbart efter en fastsat sekvens af updates.

Vi kan betragte effekten af en **change**-operation på en rent algebraisk måde. Den aktuelle query-vektor med værdien  $F$  før et update **change**( $i, 1$ ), hvor  $x_i=0$ , vil efter dette update have værdien:

$$(9) \quad F + \underbrace{00 \dots 0}_{i-1} \underbrace{11 \dots 1}_{n-i+1}.$$

Lad  $F(Q_i)$  betegne query-vektoren efter hele update-sekvensen  $Q_i$ . Mængden  $\mathcal{U}_\tau$  består af query-vektorer efter sekvenser  $Q_i \in \mathcal{Q}$ , hvor  $U(\tau)$  er et prefix af  $Q_i$ . Formelt:

$$\mathcal{U}_\tau = \{F(Q_i) \mid \exists \mu : Q_i = U(\tau\mu)\}.$$

Vi har nu lagt rammerne for følgende centrale kombinatoriske lemma. Dette lemma er den eneste del af beviset, der direkte er knyttet til paritet-prefix problemets natur.

LEMMA 9.2. *Lad  $\tau \in \{0, 1\}^*$  være givet og lad  $L = m - |\tau|$ . Der gælder:*

1.  $|\mathcal{U}_\tau| = 2^L$ .
2. *Antallet af vektorer i  $\mathcal{U}_\tau$  indenfor afstand  $n/30$  fra en vilkårlig fast vektor  $C$  er begrænset af  $2^{0,9L}$ .*

*Bevis.*  $\mathcal{U}_\tau$  består af de query-vektorer, der fremkommer ved update-sekvenser på formen  $U(\tau\mu)$  for alle mulige sekvenser  $\mu \in \{0, 1\}^L$ . Updates i suffix'et af update-sekvensen  $U(\tau\mu)$  knyttet til mønstret  $\mu$  består af **change**-operationer, der alle opdaterer i positionerne  $p_1, p_2, \dots, p_L$  i  $x$ , jævnfør definitionen for update-sekvenserne. Sorter nu disse positioner så  $1 < q_1 < q_2 < \dots < q_L < n$ , hvor hvert  $q_j$  er lig en og kun en position  $p_i$ . Definer vektorene  $v_i \in \{0, 1\}^n$  for  $i \in [1..L]$  så (idet vi lader  $q_{L+1} = n + 1$ ):

$$v_i = \underbrace{00 \dots 0}_{q_i-1} \underbrace{11 \dots 1}_{q_{i+1}-q_i} \underbrace{00 \dots 0}_{n-q_{i+1}+1}.$$

og  $c = F(U(\tau 00 \dots 0))$ . Lad  $V = \{v_1, \dots, v_L\}$ . Vi påstår at  $\text{span}(V) + c = \mathcal{U}_\tau$ . Et første skridt i beviset for denne påstand er at definere endnu et sæt af vektorer:  $w_i = v_i + v_{i+1} + \dots + v_L$ , for  $i \in [1..L]$ . D.v.s.

$$w_i = \underbrace{00 \dots 0}_{q_i-1} \underbrace{11 \dots 1}_{n-q_i+1}.$$

Lad  $W = \{w_1, w_2, \dots, w_L\}$ . Da hver  $w_i$  jo er en linear-kombination af vektorer fra  $V$  har vi direkte:

$$\text{span}(V) + c = \text{span}(W) + c.$$

Vi kan derfor vise påstanden ved at vise at  $(\text{span}(W) + c) = \mathcal{U}_\tau$ . Vi viser dette ved at vise inklusion begge veje.

$(\text{span}(W) + c) \subseteq \mathcal{U}_\tau$ : Lad  $z \in (\text{span}(W) + c)$ . Vi kan skrive  $z$  på formen:  $z = \lambda_1 w_1 + \dots + \lambda_L w_L + c$ . Vi ønsker at konstruere en update-sekvens  $U(\tau\mu)$ , hvor  $z = F(U(\tau\mu))$ . Efter updates foretaget i  $\tau$  er query-vektoren for  $x$  pr. definition lig  $c$ . Lad nu  $i_1, i_2, \dots, i_k$  være indicene for de  $\lambda_i$ , der er lig 1. Betragt positionerne  $q_{i_1}, \dots, q_{i_k}$  i  $x$ . Hver af disse positioner er det muligt at opdatere i efter sekvensen  $U(\tau)$  (da de jo er en delmængde af  $\{p_1, \dots, p_L\}$ ). Mønstret  $\mu$  vælges nu, så der i disse og kun disse positioner foretages et update på formen **change**( $q_{i_j}, 1$ ), mens øvrige positioner fra  $p_1, \dots, p_L$  sættes til 0. Fra (9) og pr. definition af  $w_i$ -vektorene, vil disse updates, f.eks. **change**( $q_{i_j}, 1$ ), bevirke at den hidtidige query-vektor  $F$  ændres til  $F + w_{i_j}$ . Da vi efter  $U(\tau)$  som nævnt har query-vektor  $c$ , vil query-vektoren efter  $U(\tau\mu)$  få værdien  $F(U(\tau\mu)) = c + w_{i_1} + w_{i_2} + \dots + w_{i_k} = \lambda_1 w_1 + \dots + \lambda_L w_L + c = z$  som ønsket.

$\mathcal{U}_\tau \subseteq (\text{span}(W) + c)$ : Lad  $z \in \mathcal{U}_\tau$ . D.v.s. der findes en update-sekvens  $\tau\mu$  så  $F(\tau\mu) = z$ . Lad  $q_{i_1}, \dots, q_{i_k}$  være positioner i  $\mu$  hvor der foretages et update på

formen **change** $(q_{i_j}, 1)$ . Som før ses det at  $z = c + w_{i_1} + \dots + w_{i_k}$ , og dermed at  $z \in (\text{span}(W) + c)$ .

Vi konkluderer dermed at  $\mathcal{U}_\tau = \text{span}(W) + c = \text{span}(V) + c$ . Da  $V$  består af  $L$  ortogonale  $\{0, 1\}$ -vektorer er størrelsen af  $\mathcal{U}_\tau$  derfor  $2^L$ , som beviser punkt 1 i lemmaet.

Vi vil nu vise punkt 2. Lad  $C \in \{0, 1\}^n$  være en vilkårlig fast vektor. Lad  $v$  være en vektor i  $(\text{span}(V) + c) = \mathcal{U}_\tau$  med mindst mulig afstand til  $C$ . Skriv  $v$  som linear-kombinationen  $v = c + \lambda_1 v_1 + \dots + \lambda_L v_L$ , hvor  $\lambda = (\lambda_1, \dots, \lambda_L) \in \{0, 1\}^L$ .

Vi vil estimere, hvor mange vektorer  $w \in (\text{span}(V) + c)$ , der maksimalt er afstand  $n/30$  fra  $C$ . Lad  $v' = c + \lambda'_1 v_1 + \dots + \lambda'_L v_L$  og lad  $K = |\{i \mid \lambda'_i \neq \lambda_i\}| = |\lambda' - \lambda|$ . Der gælder da  $|v' - C| \geq K \frac{n}{6L}$ , hvilket ses af følgende ræsonnement.

Lad  $i$  være et index så  $\lambda'_i \neq \lambda_i$ . Betragt del-vektoren  $v[q_i, q_{i+1} - 1]$ . Da denne del-vektor udelukkende består af 0'er hvis  $\lambda_i = 0$  eller 1'er hvis  $\lambda_i = 1$ , og  $v$  er valgt så den har mindst afstand til  $C$ , må afstanden imellem  $C[q_i, q_{i+1} - 1]$  og  $v[q_i, q_{i+1} - 1]$  være mindre end eller lig  $\frac{1}{2}n/(3L)$ , da intervallet  $[q_i, q_{i+1} - 1]$  er af længde mindst  $\frac{n}{3L}$  ifølge lemma 9.1. For hvert  $\lambda'_i \neq \lambda_i$  i ovennævnte linear-kombinationer, er afstanden imellem  $C[q_i, q_{i+1} - 1]$  og  $v'[q_i, q_{i+1} - 1]$  derfor større end  $n/(6L)$ . De  $K$  disjunkte intervaller  $[q_{i_1}, q_{i_1+1} - 1], \dots, [q_{i_K}, q_{i_K+1} - 1]$ , hvor  $\lambda_{i_j} \neq \lambda'_{i_j}$ , i  $v'$  og  $C$ , vil derfor sikre den påståede afstand imellem disse to vektorer.

For at sikre at  $K \frac{n}{6L}$  er mindre end  $n/30$  må  $K \leq L/5$ . Antallet af vektorer  $v'$  indenfor  $n/30$  fra  $C$  må derfor være indeholdt i  $M = \{\lambda'_1 v_1 + \dots + \lambda'_L v_L + c \mid L/5 \geq |\lambda' - \lambda|\}$ .

D.v.s. antallet af vektorer  $\lambda' \in \{0, 1\}^L$  indenfor afstand  $L/5$  fra  $\lambda$  begrænser størrelsen af  $M$ . Dette antal svarer til antallet af måder man kan udvælge op til  $L/5$  indgange fra  $\lambda$  og negere deres værdi.

$$\begin{aligned} |M| &\leq \sum_{k=1}^{L/5} \binom{L}{k} \leq L/5 \cdot \binom{L}{L/5} \leq L/5 \cdot \frac{L^{(L/5)}}{(L/5)!} \\ &\leq L/5 \cdot \frac{L^{(L/5)}}{\sqrt{2\pi(L/5)}(L/5e)^{(L/5)}} \\ &< L/5\sqrt{L} \cdot \frac{L^{(L/5)}}{\left(\frac{1}{5e}L\right)^{(L/5)}} \\ &= \sqrt{L}/5 \cdot 5e^{(1/5)L} = \sqrt{L}/5 \cdot 2^{\log 5e \cdot 0,2L} \leq \sqrt{L}/5 \cdot 2^{0,8L} < 2^{0,9L}. \end{aligned}$$

Her er benyttet en egenskab ved binomial koefficienten (proposition A.1) og Stirlings formel (proposition A.2).

□

Uformelt fortæller lemma 9.2 at vektorene  $\mathcal{U}_\tau$  er "meget forskellige", d.v.s. kun få af disse vektorer (mindre end  $k \cdot 2^{0,9L}$  ud af  $2^L$  mulige) *approximeres* godt af  $k$  fastholdte vektorer. Vi præciserer senere betydningen af dette.

**Data under epoch  $i - 1$  til epoch 1.** Registrene i cell-probe-modellen inddeles nu efter, hvilket epoch de undervejs er blevet ændret i. Vi betegner et register som et *epoch  $i$ -register* hvis og kun hvis det er blevet ændret af et update forekommende i epoch  $i$ , og ikke er ændret i de resterende epochs;  $i - 1$  til 1. Vi vil nu for alle sekvenser i  $\mathcal{Q}$  og efterfølgende prefix-queries, nedad-begrænse disse queries gennemsnitlige antal læsninger, hvor gennemsnittet tages over alle kombinationer af

sekvenser i  $\mathcal{Q}$  og queries  $\mathbf{prefix}(1) \dots \mathbf{prefix}(n)$ . Ideen er at vi for hver epoch  $i$  tæller hvormange prefix-queries efter de forskellige sekvenser  $Q_i \in \mathcal{Q}$ , der nødvendigvis skal læse et register fra dette epoch. Udgangspunktet for dette er for fastholdt epoch  $i$ , at påvise at epoch  $i - 1$  til epoch 1-registrene ikke kan afspejle effekten af de forskelligeartede updates under epoch  $i$ , og dermed må nogen af prefix-query'ene være tvunget til at hente informationen direkte fra et epoch  $i$ -register! Vi starter med at estimere, hvor meget informationen, der maksimalt kan lagres *efter* epoch  $i$ . Antallet af skrive-operationer foretaget i epoch  $i - 1$  til og med epoch 1 kan begrænses til

$$(10) \quad wL_{i-1} = L_i/\alpha.$$

Disse skrive-operationer kan skrive til et potentionelt stort antal forskellige registre, men vi er her kun interesseret i de registre, der kan læses indenfor  $q$  læsninger. Intuitionen bag dette er, at målet med analysen er at nedadbegrænse antallet af læsninger for de forskellige queries til en fast brøkdel af  $q$ . Hvis et prefix-query derfor foretager en læsning af et register efter udførelsen af  $q$  læsninger, vil dette prefix-query automatisk opfylde dette mål. Vi vender senere tilbage til, hvordan dette konkret hænger sammen i analysen. Antallet af registre, der kan læses indenfor  $q$  læsninger er begrænset af

$$(11) \quad n2^{qb}$$

som er det samlede antal registre i  $n$  (et for hvert update) cell-probe-træer med dybde  $q$  og grad højst  $2^b$ .

Vi ønsker en øvre grænse for, hvor mange forskellige hukommelses-konfigurationer cell-probe-algoritmen maksimalt kan skelne indenfor  $q$  læsninger, hvor kun epoch  $i - 1$  til epoch 1 registre tillades at variere. Dette antal, som vi betegner  $\epsilon_i$ , må være begrænset af antallet af forskellige steder, registrene kan være placeret gange antallet af mulige indhold i disse. Fra (10) og (11) fås da, at  $\epsilon_i$  er begrænset ved følgende grove vurdering:

$$(12) \quad \epsilon_i \leq \binom{n2^{qb}}{L_i/\alpha} 2^{bL_i/\alpha} \leq (n2^{qb})^{L_i/\alpha} 2^{bL_i/\alpha} \leq 2^{\left(\frac{2(bq+\log n)}{\alpha}\right)L_i}$$

**Tælle-argumentet.** Fasthold nu epoch  $i$ . Lad  $\tau$  være et mønster i  $\{0, 1\}^{m-L_i}$ , d.v.s.  $U(\tau)$  omfatter updates frem til men uden epoch  $i$ . Vi "markerer" nu epoch  $i$  ved at lade prefix-queries, der læser fra et epoch  $i$ -register eller læser efter  $q$  læsninger, læse et 0 i stedet. Se figur 9.2. Vi vil i det følgende skelne mellem svar hørende til disse modificerede prefix-queries og de korrekte umodificerede queries. Lad  $J(Q_j)$  være analogien til  $F(Q_j)$ , men hvor query-vektoren er den, der fremkommer som resultat af ovennævnte modificerede læsninger. Tilsvarende lader vi  $\mathcal{M}_\tau$  betegne analogien til  $\mathcal{U}_\tau$ , blot for vektorer  $J(Q_j)$ . Formelt

$$\mathcal{M}_\tau = \{J(Q_i) | \exists \mu : Q_i = U(\tau\mu)\}.$$

Husk på  $\mathcal{U}_\tau$  som umodificerede query-vektorer og  $\mathcal{M}_\tau$  som modificerede.

Modifikationen af et prefix-query giver udslag i to effekter.

- Hvis et modificeret prefix-query adskiller sig fra det korrekte resultat, må den korrekte beregning enten have læst et epoch  $i$ -register eller have foretaget mere end  $q$  læsninger. Dette kan også betragtes som, at  $|J(U(\tau\mu)) - F(U(\tau\mu))|$  af **prefix**-kaldene  $\mathbf{prefix}(1), \dots, \mathbf{prefix}(n)$  læser fra epoch  $i$  eller foretager mere end  $q$  læsninger.

- $|\mathcal{M}_\tau| \leq \epsilon_i$ . Betragt update-sekvenserne  $\mathcal{Q}_\tau = \{U(\tau\mu) | \mu \in \{0,1\}^{L_i}\}$ , og bemærk at  $\mathcal{M}_\tau$  kan betragtes som mængden  $\{J(Q) | Q \in \mathcal{Q}_\tau\}$ .

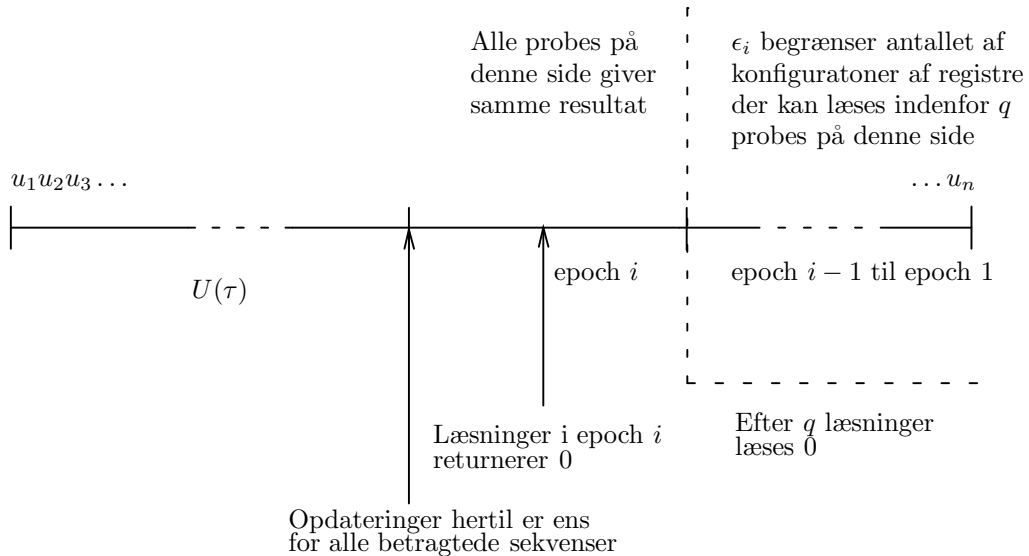
For et fast modificeret **prefix-query**, **prefix**( $k$ ), kan vi nu splitte de læste registre op i to mængder. Den ene mængde er registre der er læst indenfor  $q$  læsninger og er tilknyttet epoch  $i-1$  til epoch 1. Den anden mængde består af den resterende del af registrene. Vi påstår, at det returnerede resultat af en læsning fra et register i den sidstnævnte mængde vil være uafhængig af sekvensen  $Q \in \mathcal{Q}_\tau$ . Vi argumenterer for dette ved følgende case-analyse:

*Der læses fra et epoch  $i+1$  til epoch  $q$  register:* Alle sekvenserne i  $\mathcal{Q}_\tau$  starter med update-prefixet  $U(\tau)$ . Altså må epoch  $i+1$ -registre frem til epoch  $q$ -registre have ens indhold for alle sekvenserne, og dermed give samme resultat i dette tilfælde.

*Der læses fra et register efter  $q$  læsninger er foretaget:* Pr. definition af et modificeret query-kald læses der et 0 uanset sekvens.

*Der læses fra et epoch  $i$  register:* Der er igen tale om at et query-kald, der læser et 0 uanset sekvens.

Størrelsen af  $\{J(Q) | Q \in \mathcal{Q}_\tau\} = \mathcal{M}_\tau$  må derfor kun afhænge af epoch  $i-1$  til epoch 1 registre læst indenfor  $q$  læsninger. Dermed er denne størrelse begrænset af  $\epsilon_i$ , altså  $|\mathcal{M}_\tau| \leq \epsilon_i$ .



FIGUR 9.2. Situation ved modificeret prefix-kald efter sekvens startende med  $\tau$

Intuitionen er nu, at  $\epsilon_i$  (og dermed  $|\mathcal{M}_\tau|$ ) er så lille relativt til størrelsen af  $\mathcal{U}_\tau$ , at mange af de korrekte query-vektorer i  $\mathcal{U}_\tau$  må adskille sig meget fra enhver af vektorene i  $\mathcal{M}_\tau$ , da  $\mathcal{U}_\tau$  som før nævnt er svær at approksimere med en lille mængde af vektorer.

Mere præcist gælder der ifølge lemma 9.2 punkt 2, at højst  $2^{0,9L_i}$  af query-vektorene i  $\mathcal{U}_\tau$  er mindre end  $n/30$  fra en vilkårlig fast vektor. D.v.s. at antallet af vektorer i  $\mathcal{U}_\tau$ , der er så "tæt" på mindst een af vektorene i  $\mathcal{M}_\tau$ , er begrænset af

$|\mathcal{M}_\tau| \cdot 2^{0,9L_i} \leq \epsilon_i \cdot 2^{0,9L_i}$  eller sagt på en anden måde; Lad  $\mathcal{D}_\tau$  være de vektorer i  $\mathcal{U}_\tau$ , der er længere end  $n/30$  fra enhver vektor i  $\mathcal{M}_\tau$ . Vi får da:

$$(13) \quad \begin{aligned} |\mathcal{D}_\tau| &\geq |\mathcal{U}_\tau| - \epsilon_i \cdot 2^{0,9L_i} \\ &= 2^{L_i} - \epsilon_i \cdot 2^{0,9L_i}, \end{aligned}$$

idet  $|\mathcal{U}_\tau| = 2^{L_i}$  ifølge lemma 9.2 punkt 1.

En update-sekvens  $U(\tau\mu)$  med  $F(U(\tau\mu)) \in \mathcal{D}_\tau$  (og dermed  $|F(U(\tau\mu)) - J(U(\tau\mu))| \geq n/30$ ) må ifølge ovennævnte diskussion sikre, at mindst  $1/30$  af de  $n$  mulige prefix-kald efter  $U(\tau\mu)$  læser fra epoch  $i$  eller foretager mere end  $q$  læsninger. Definer nu:

$$\begin{aligned} i\_read(\tau\mu) &= \{k | \mathbf{prefix}(k) \text{ efter } U(\tau\mu) \text{ læser fra epoch } i \text{ og foretager maksimalt } q \text{ læsninger}\} \\ q\_read(\tau\mu) &= \{k | \mathbf{prefix}(k) \text{ efter } U(\tau\mu) \text{ foretager mere end } q \text{ læsninger}\} \end{aligned}$$

og bemærk at for sekvenser  $U(\tau\mu)$ , hvor  $F(U(\tau\mu)) \in \mathcal{D}_\tau$ , er

$$|i\_read(\tau\mu) \cup q\_read(\tau\mu)| \geq n/30.$$

Vi vil nu nedad-begrænse det totale antal læsninger, benævnt  $R$ , for *alle* de  $n \cdot 2^m$  kombinationer af de  $2^m$  update-sekvenser  $\mathcal{Q}$  og  $n$  prefix-kald på følgende lidt specielle måde:

$$(14) \quad \begin{aligned} R &\geq \left( \sum_{i \leq q} \sum_{\tau: |\tau|=m-L_i} \sum_{\mu: |\mu|=L_i} |i\_read(\tau\mu)| \right) + \sum_{\tau\mu: |\tau\mu|=m} q |q\_read(\tau\mu)| \\ &= \sum_{i \leq q} \sum_{\tau: |\tau|=m-L_i} \sum_{\mu: |\mu|=L_i} (|i\_read(\tau\mu) \cup q\_read(\tau\mu)|) \\ &\geq \sum_{i \leq q} \sum_{\tau: |\tau|=m-L_i} \left( \frac{n}{30} |\mathcal{D}_\tau| \right) \\ &\geq \sum_{i \leq q} 2^{m-L_i} \frac{n}{30} (2^{L_i} - \epsilon_i \cdot 2^{0,9L_i}), \end{aligned}$$

hvor sidste ulighed følger af (13), samt det faktum at der er  $2^{m-L_i}$  mulige kombinationer af updates  $U(\tau)$  inden epoch  $i$ .

Det gennemsnitlige antal læse-operationer  $r$  pr. prefix-kald begrænses nedad ud fra (12) og (14) ved:

$$(15) \quad \begin{aligned} r = \frac{1}{n \cdot 2^m} R &\geq \frac{1}{n \cdot 2^m} \sum_{i \leq q} 2^{m-L_i} \frac{n}{30} (2^{L_i} - \epsilon_i \cdot 2^{0,9L_i}) \\ &\geq \frac{1}{30} \sum_{i \leq q} 1 - (2^{-L_i} 2^{(\frac{2(bq+logn)}{\alpha} + 0,9)L_i}) \\ &\geq \frac{1}{30} \left( q - \sum_{i \leq q} 2^{(\frac{2(bq+logn)}{\alpha} - 0,1)L_i} \right). \end{aligned}$$

Vi er nu klar til at regne på den asymptotiske størrelse af  $r$ . Vi starter med at se på  $q$ .

$$(16) \quad q = \lfloor \log_{\alpha w} m \rfloor = \frac{\log \lfloor n/6 \rfloor}{\log \alpha w}.$$

Fastsæt  $\alpha$  til  $100b \log n$ . Vi har da (idet  $q < \log n$ ):

$$(17) \quad \frac{2(bq + \log n)}{\alpha} \leq \frac{2(b \log n + \log n)}{100b \log n} \leq \frac{1}{25} = 0,04.$$

Størrelsen af  $\alpha$  sikrer, at  $L_i \geq 100$  for alle  $i$ , og vi får nu udfra (15) og (17):

$$\begin{aligned} r &\geq \frac{1}{30} \left( q - \sum_{i \leq q} 2^{(\frac{2(bq + \log n)}{\alpha} - 0,1)L_i} \right) \\ &\geq \frac{q}{30} - \sum_{i \leq q} 2^{-0,06 \cdot L_i} \geq \frac{q}{30} - \sum_{i \leq q} 2^{-0,06 \cdot 100} \\ &\geq \frac{q}{30} - q \frac{1}{64} \end{aligned}$$

D.v.s. for det valgte  $\alpha$  er  $r \in \Omega(q)$ . Fra (16) opnår vi nu det endelige resultat:

$$r \in \Omega \left( \frac{\log n}{\log(bw \log n)} \right).$$

Til sidst er der kun at bemærke, at  $r$  er det forventede antal læse-operationer  $\hat{A}$  bruger for et uniformt tilfældigt prefix-query fra 1 til  $n$  efter en uniformt tilfældig sekvens  $Q_i$ . Yao's minimax-princip [Yao77] sikrer os da samme grænse for det forventede antal læse-operationer for en randomiseret Las Vegas algoritme.  $\square$

Teknikken anvendt i ovennævnte bevis, kaldes ofte for *time-stamp*-metoden. Dette navn kommer af ideen med at splitte registre, der skrives i under update-sekvenserne, op efter det tilhørende epoch. Denne opdeling kan betragtes som at hvert register ved skrivning til dette, får et tidsstempel lig med nummeret på det aktuelle epoch.



## Nedre grænser og simple reduktioner

I dette kapitel vil vi anvende resultatet (sætning 9.1) fra forrige kapitel til via direkte reduktioner at give  $\Omega(\frac{\log n}{\log \log n})$  nedre grænser for to af de problemer vi beskæftiger os med. Det ene problem er i **change** -modellen for  $D_1$ , hvor vi har det beskrevne **match** -query. Det andet problem er genkendelse i **insert /delete** -modellen. Udover disse to grænser vil vi også vise nogle reduktioner for sammenhængen imellem flere af de betragtede problemer for **change** -modellen, der er indgår som en del af figur 4.1 og 4.2 i kapitel 4.

Inden vi går igang med de konkrete reduktioner, indfører vi for simplicitetens skyld en mindre generalisering af Dyck-sprogene, der er praktisk i **change** -modellen. Generaliseringen består i, at vi udvider med et særligt ekstra tegn;  $\#$ . Lad  $D$  betegne et vilkårligt af Dyck-sprogene.  $D_\#$  betegner det  $\#$ -udvidede sprog, defineret ved:

$$v \in D_\# \Leftrightarrow v \setminus \# \in D$$

hvor  $v \setminus \#$  er  $v$  med  $\#$  fjernet og øvrige tegn bibeholdt i hidtidig orden. I **change** -modellen kan  $\#$ -tegnene fungere som “blankt” tegn, og dermed skabe plads for “indsættelse” imellem øvrige parenteser. Vi vil f.eks. generelt omtale  $\#$ -tegnene i termer af en position, hvor vi (via **change**) sætter en parentes, når vi ændrer  $\#$  til parentes, og fjerner en parentes, når vi ændrer til  $\#$ . Operationerne **prefix**, **interval** og **match** defineres som hidtil, idet vi vælger at **match** for et  $\#$ -tegn er tegnet selv. Endeligt definerer vi  $\#^R = \#$ .

$\#$ -udvidelsen ændrer ikke kompleksiteten for nogen af operationerne, hvilket følger af følgende simple lemma.

LEMMA 10.1. *Lad  $D$  være et Dyck-sprog (en- eller to-sidigt) over alfabetet  $\Sigma = \{a_1, \bar{a}_1, \dots, a_k, \bar{a}_k\}$ . Afbildningen  $\phi(v) : (\Sigma \cup \{\#\})^* \rightarrow \Sigma^*$  defineret ved*

$$\begin{aligned} \phi(a_i) &= a_i a_i \\ \phi(\#) &= a_1 \bar{a}_1 \\ \phi(av) &= \phi(a)\phi(v) \end{aligned}$$

*opfylder  $v_{[i..j]} \in D_\# \Leftrightarrow \phi(v)_{[2i..2j]} \in D$ . For en vektor  $x \in (\Sigma \cup \{\#\})^n$  kan vektoren  $\phi(x) \in \Sigma^{2n}$  opretholdes ved max. 2 ændringer pr. ændring af et tegn i  $x$ .*

Beviset overlades til læseren. Bemærk at alle betragtede query-operationer er defineret i termer af bestemte intervaller i opretholdt  $v$  (f.eks. også **match**), og ovennævnte 1-1 korrespondance mellem intervaller i  $v$  og  $\phi(v)$  er derfor tilstrækkelig til at aflæse svar for alle betragtede queries. Fremover lader vi opretholdsen af  $\phi$  være en implicit del af reduktioner knyttet til **change** -modellen beskrevet i dette speciale.

SÆTNING 10.1. **match** $_{D_1}$  har nedre grænse på  $\Omega(\frac{\log n}{\log \log n})$ .

*Bevis.* Beviset er som nævnt en reduktion fra paritet-prefix problemet. Lad  $x \in \{0, 1\}^n$  være en input-instans for paritet-prefix problemet.

Vi opretholder nu vektoren  $y \in D_1$  på formen:

$$y = (b_n(b_{n-1}(\dots(b_1)))\dots)\underbrace{\#\#\dots\#}_{n-m}$$

hvor

$$b_i = \begin{cases} (, & \text{hvis } x_i = 1 \\ \#, & \text{hvis } x_i = 0 \end{cases}$$

og  $m = \sum_{1 \leq u \leq n} x_u$ .

Antag vi skal afgøre pariteten af et prefix i  $x$  af længde  $k$ . Parentesen lige inden  $b_k$  i  $y$ -vektoren;  $\dots(b_k(\dots)$ , d.v.s.  $y_{2n-2k+1}$ , vil nu matche en højreparentes i sidste halvdel af  $y$ . Det kan let vises, at index'et på denne matchende parentes er  $\mathbf{match}(2n - 2k + 1) = \sum_{1 \leq i \leq k} x_i + 2n + k$ , idet  $b_i$ 'erne med  $i \leq k$ , hvor  $x_i = 1$ , vil tvinge den matchende parentes' index et mod højre. Da vi kan opretholde  $y$  ved  $O(1)$  **change**-operationer pr. **change** i  $x$ , og finde værdien af **prefix**( $k$ ) for  $x$  ved et enkelt **match**-query i  $y$ , idet  $\mathbf{prefix}(k) = (\mathbf{match}(2n - 2k + 1) + k \bmod 2)$ , har vi fra 9.1 den påståede nedre grænse. □

Den næste nedre grænse vi viser er for **insert** / **delete**-modellen. Denne grænse er mere knyttet til selve modellens natur, end genkendelsen af Dyck-sprogene. Denne nedre grænse er derfor primært interessant ud fra den synsvinkel, at dynamiske Dyck-sprogsalgoritmer anvendt i tekst-editorer eller lignende, ofte vil tage udgangspunkt i denne model. Beviset bygger på en  $\Omega(\frac{\log n}{\log \log n})$  nedre grænse for det såkaldte *list repræsentationsproblem*, der er nævnt uden bevis i [FS89]. Beviset er også beskrevet i [FHM<sup>+</sup>95].

List repræsentationsproblemet er følgende: Dynamisk skal man opretholde en streng  $x \in \{0, 1\}^*$  under følgende operationer:

**insert**( $i, x$ ): Indsætter  $x \in \{0, 1\}$  imellem  $x_{i-1}$  og  $x_i$  i  $x$ .

**delete**( $i$ ): Fjerner tegnet  $x_i$  fra  $x$ .

**value**( $i$ ): Returnerer værdien af  $x_i$ .

LEMMA 10.2. *List repræsentationsproblemt har nedre grænse på  $\Omega(\frac{\log n}{\log \log n})$ .*

Beviset er udeladt her, idet der henvises til [FHM<sup>+</sup>95].

SÆTNING 10.2. *Dynamisk genkendelse af et vilkårligt Dyck-sprog  $D$  i **insert** / **delete**-modellen kræver tid  $\Omega(\frac{\log n}{\log \log n})$ .*

*Bevis.* Beviset for denne sætning er igen en reduktion, hvor vi løser list repræsentationsproblemet v.h.a. af operationerne **insert**, **delete** og **member** for Dyck-sproget  $D$ . Lad  $x \in \{0, 1\}^*$  være strengen vi ønsker at opretholde under update-operationerne **insert** og **delete**, samt queryet **value**.

Vi opretholder nu en parentes-streng  $y$ , korresponderende til  $x$ , hvor vi for nemheds skyld definerer parentesparret så  $A = \{0\}$  og  $\bar{A} = \{1\}$ .  $y$  skal opretholdes på formen:  $y = \underbrace{00\dots0}_n x x^R \underbrace{11\dots1}_n$ , hvor  $|x| = n$ . Denne streng vil altid tilhøre

et vilkårligt Dyck-sprog, og kan nemt opretholdes ved max. 4 **insert** / **delete**-operationer pr. **insert** / **delete** i  $x$ , når  $n$  også opbevares. En **value**-operation for  $x$

simuleres nu ved ændre  $y_i$  til 1 (ved hjælp af **delete**( $i$ ) og efterfølgende **insert**( $i, 1$ )), hvorefter  $y$  stadig tilhører Dyck-sproget hvis og kun hvis  $y_i = x_i$  var 1 (da der ellers vil være forskel på antal 1'er og 0'er i  $y$ ), der dermed giver os svaret på **value**( $i$ ). Efter et **value** -query genetableres  $y$  let på nævnte form i  $O(1)$  operationer. Fra lemma 10.2 følger den påståede grænse.  $\square$

Fremover vil vi udelukkende fokusere på den svagere **change** -model.

### 10.1. Sammenhænge mellem problemer i **change**-modellen

Som nævnt vil vi her beskrive flere reduktioner imellem de forskellige problemer for Dyck-sprogene i **change** -modellen. Vi viser følgende tre reduktioner, der sammen med sætningerne 2.2 og 2.3 fra kapitel 2.2 danner basis for alle pile i figurene 4.1 og 4.2 i kapitel 4.

**interval** $_{D_k}$  reducerer til **match** $_{D_k}$ .

**interval** $_{D'_k}$  reducerer til **member** $_{D'_{k+1}}$ .

**interval** $_{D_1}$  reducerer til **member** $_{D_2}$ .

SÆTNING 10.3. **interval** $_{D_k}$  reducerer til **match** $_{D_k}$  for alle  $k$ .

*Bevis.* Lad  $x$  være en instans af  $D_k$ . Vi opretholder nu vektoren  $y$  på formen

$$y = \#x_1\#x_2\#\dots\#x_n\#.$$

Et query for intervallet  $[i, j]$  besvares nu ved at sætte to parenteser omkring intervallet, således at  $y$  ændres til

$$y' = \#x_1\#\dots(x_i\#\dots\#x_j)\dots\#x_n\#.$$

Ved at anvende **match** på den første af de to nye parenteser er det muligt at undersøge, om  $x_{[i..j]} \in D_k$ . Det indses let, at **match** returnerer den anden nye parentes, hvis og kun hvis  $x_{[i..j]} \in D_k$ . Efter queryet genoprettes  $y$ .  $\square$

SÆTNING 10.4. **interval** $_{D'_k}$  reducerer til **member** $_{D'_{k+1}}$ .

*Bevis.* Antag af parentesparret  $[, ]$  er det  $k + 1$ 'te parentespar for input-instanser for **member** $_{D'_{k+1}}$ , der ikke er en del af parenteserne for **interval** $_{D'_k}$ , og alle øvrige parentespar iøvrigt er ens for de to probleminstanser.

Lad  $x$  være en input-instans for **interval** $_{D'_k}$ . Vi opretholder nu vektoren  $y$  på formen:

$$y = \#x_1\#x_2\#\dots\#x_n\#x_n^R\#x_{n-1}^R\#\dots\#x_1^R\#$$

og det bemærkes at  $y \in D'_{k+1}$ .

Et query for intervallet  $[i, j]$  besvares nu ved at sætte to firkantede parenteser omkring intervallet, således at  $y$  ændres til:

$$y' = \#x_1\#\dots[x_i\#\dots\#x_j]\dots\#x_n\#x_n^R.$$

Det indses let, at  $y' \in D'_{k+1}$  hvis og kun hvis  $x_{[i..j]} \in D'_k$ . Efter queryet genoprettes  $y$ .  $\square$

I det en-sidige tilfælde kan vi ikke vise helt så stærkt et resultat, men må nøjes med at se på niveauet for  $D_1$  og  $D_2$ .

SÆTNING 10.5. **interval** $_{D_1}$  reducerer til **member** $_{D_2}$ .

*Bevis.* Beviset er næsten magen til forrige bevis for sætning 10.4. Lad inputinstanser for **interval** $_{D_1}$  spænde over parentesparret  $(,)$  og **member** $_{D_2}$  over parentesparrene  $(,)$  og de firkantede  $[,]$ .

Problemet i forhold til forrige bevis er, at vektoren  $y$  nu skal opretholdes, så den forbliver korrekt balanceret for det en-sidige Dyck-sprog  $D_2$ . Vi sikrer dette ved at opretholde  $y$  på formen:

$$y = \underbrace{\left( \dots \left( \#x_1\#x_2\#\dots\#x_n\#x_n^R\#\dots\#x_1^R\# \right) \right)}_n \dots$$

De  $n$  parenteser før og efter sikrer, at  $y \in D_2$ . Som før findes svaret for et intervalquery ved at sætte firkantede parenteser omkring det korresponderende interval i  $y$ .

□

Bemærk at ovennævnte bevis *ikke* generaliserer til  $D_k$  og  $D_{k+1}$ , da vi ikke kan opretholde en balancering af  $y$  ved at omkapsle med parenteser.

## Dynamisk prefix balancering

I dette kapitel introducerer vi teknikken *dynamisk prefix balancering*. Det er denne reduktionsteknik, der kombineret med det beskrevne resultat af Fredman og Saks for paritet-prefix problemet, gør os i stand til at vise forholdsvis gode grænser for mange af problemerne for Dyck-sprogene i **change**-modellen.

I første omgang viser vi, hvordan reduktionsteknikken kan bruges til at vise en særlig nedre grænse for det nye dynamiske problem vi introducerede i kapitel 3, kaldet *signed prefix sum* problemet. Det særlige ved denne nedre grænse er, at vi tillader tiden pr. prefix-query at afhænge af den returnerede sums størrelse. Igennem denne grænse fanger vi en fælles egenskab for de Dyck-sprogsproblemer vi senere viser nedre grænser for, og samtidigt beskrives teknikken dynamisk prefix balancering bedst i denne kontekst.

Det er klart, at signed prefix sum problemet har nedre grænse på  $\Omega\left(\frac{\log n}{\log \log n}\right)$  pr. operation, idet problemet er en generalisering af paritet-prefix problemet (sætning 9.1). Som nævnt er vi derimod interesseret i en anden grænse, hvor tiden for **sum** må afhænge af den returnerede sums størrelse. Dette er udtrykt i følgende sætning.

**SÆTNING 11.1.** *Lad  $A$  være en algoritme for signed prefix sum problemet, der bruger worst-case tid  $O(t(n))$  for **change** og forventet  $O(t(n)|s|)$  tid for **sum**( $k$ ), hvor  $s = \sum_{1 \leq i \leq k} x_i$ , d.v.s. værdien der returneres af **sum**( $k$ ). Da er  $t \in \Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ .*

*Bevis.* Beviset består i en reduktion, hvor vi antager vi har en implementation, der løser signed prefix sum problemet i worst-case tid  $O(t(n))$  for en **change**-operation og forventet tid  $O(t(n)|h|)$  for queryet **sum**( $k$ ) =  $h$ , og vi ønsker nu at løse paritet-prefix problemet via denne implementation. Lad  $x = (x_1, x_2, \dots, x_n)$  være en instans af paritet-prefix problemet, hvori der fortløbende foretages opdateringer. Vi vil nu opretholde vektoren  $y \in \{-1, 0, 1\}$  så:

$$(18) \quad y_i = \begin{cases} 0 & \text{hvis } x_i = 0 \\ 1 \text{ eller } -1 & \text{hvis } x_i = 1. \end{cases}$$

Et paritet-prefix query **prefix**( $k$ ) kan herved direkte besvares ved at returnere **sum**( $k$ ) mod 2, idet  $-1 = 1 \pmod{2}$ . Ideen er, at vi ønsker at begrænse tiden for et query-svar **sum**( $k$ ) mod 2, ved at sørge for, at denne sum er *forventet* tæt på 0, samtidigt med at korrespondancen (18) opretholdes. Hvis det sikres, at summen  $|\sum_{1 \leq i \leq k} y_i|$  forventet er mindre end  $f(n)$ , ved for hver ændring i  $x$  at foretage maksimalt  $O(\log^k n)$  ændringer i  $y$ , vil de samlede tider for denne reduktion være:

- En **change**-operation i  $x$  simuleres i tid  $O(\log^k n \cdot t(n))$  v.h.a.  $\log^k n$  **change**-operationer i  $y$ .
- Et **prefix**-query i  $x$  simuleres i forventet tid  $O(f(n)t(n))$

Ud fra sætning 9.1 vil tiden  $t(n)$  derfor være nedad-begrænset af:

$$(19) \quad \Omega(\log / (f(n) \log \log n)).$$

Det tilbagestående problem er altså at begrænse  $f(n)$ . Det er dette problem vi omtaler som *dynamisk prefix balancering*. Vores resultat er her en randomiseret strategi, der sikrer at  $f \in O(\sqrt{\frac{\log n}{\log \log n}})$ , der dermed ifølge (19) giver os den påståede grænse for  $t$ , nemlig  $t \in \Omega(\sqrt{\frac{\log n}{\log \log n}})$ . Det nævnte resultat udgør sætning 11.2 herunder. □

**SÆTNING 11.2** (Randomiseret dynamisk prefix balancering). *Lad  $x \in \{0, 1\}^n$  være en input-instans og  $y \in \{-1, 0, 1\}^n$  den korresponderende vektor, der altid skal opfylde korrespondancen (18) (og dermed  $\sum_{1 \leq i \leq k} y_i = \sum_{1 \leq i \leq k} x_i \pmod 2$  for alle  $k$ ).*

*Der findes en randomiseret strategi, så et **change** i  $x$  kan afspejles ved worst-case  $O(\frac{\log^2 n}{\log \log n})$  **change-operationer** i  $y$  og samtidigt opfylde, at alle prefix-summer  $|\sum_{1 \leq i \leq k} y_i|$  forventet er  $O(\sqrt{\frac{\log n}{\log \log n}})$  store.*

*Bevis.* Beviset består i en struktur, der er en række lister  $l_1, l_2, \dots, l_h$  af aftagende længde. Listerne består af indgange  $y_i$  fra  $y$ , hvor  $x_i = 1$ , d.v.s. de indgange i  $y$  der skal være 1 eller -1. Listens indgange er ordnet efter indeksernes orden i  $y$ , d.v.s.  $l_k = (y_{i_1}, y_{i_2}, \dots, y_{i_m})$ ,  $i_1 < i_2 < \dots < i_m$ . Den første liste  $l_1$  omfatter alle indgangene  $y_i$ , hvor  $x_i = 1$ . Når der foretages et **change**( $k$ ) i  $x$ , svarer dette derfor til indsættelse/fjernelse af  $y_k$  i  $l_1$ . Til hver liste tilknyttes en såkaldt *parring* af  $y$ -indgangene i listen. En parring er en mængde af nabo-par fra  $l_k$ , hvor hver indgang fra  $l_k$  højst forekommer i et enkelt par i parringen. Et element, der ikke forekommer i noget par i parringen kalder vi en *oversidder* fra  $l_k$ . Parringerne vil senere danne basis for, hvorvidt en specifik indgang i  $y$  sættes lig 1 eller -1.

Listerne og de tilhørende parringer skal opfylde følgende invariant:

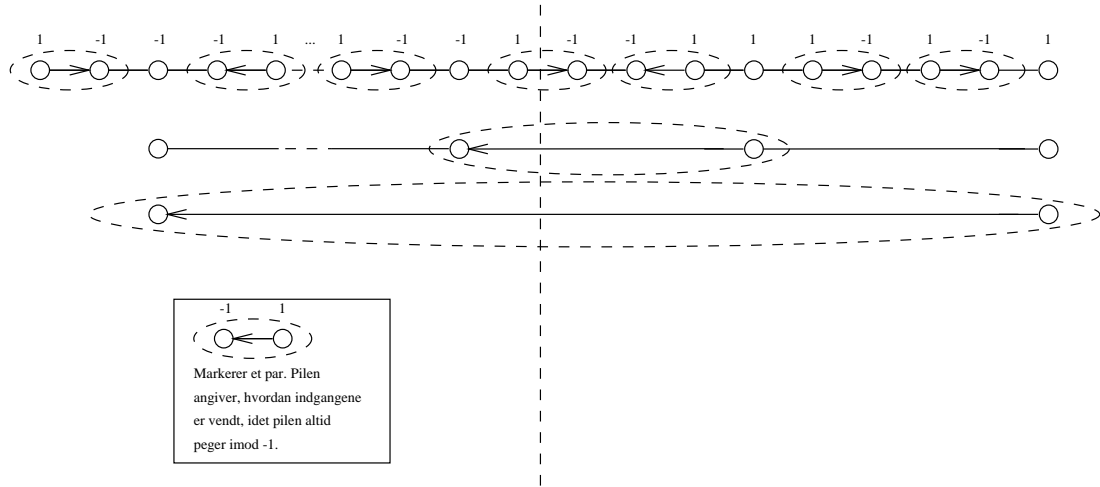
- Oversiddere i en liste  $l_k$  skal være adskilt af mindst  $\log n$  elementer i listen.
- Alle oversiddere fra en liste  $l_k$  udgør præcis elementerne i liste  $l_{k+1}$ .
- Endeligt skal der gælde det tekniske krav at en liste med et enkelt element ikke viderefører dette element som en oversidder

I figur 11.1 ses et eksempel med 3 lister. Parrene i en liste er markeret med stiplede omkransning. Knuder uden omkransning er oversidderne, som er direkte videreført i næste liste.

Ovennævnte invariant sikrer, at liste  $l_{k+1}$  højst har størrelse  $\lceil |l_k| / \log n \rceil$ , da dette er det maksimale antal oversiddere i  $l_k$ . Heraf følger at der højst er  $O(\frac{\log n}{\log \log n})$  lister. Samtidigt bemærkes, at enhver indgang i  $y$  er parret på et og kun et niveau, på nær eventuelt en sidste enlig indgang  $y_i$ . Af tekniske årsager vælger vi at betragte denne indgang, som parret med en særlig dummy-indgang  $y_{n+1}$ , der ligger efter alle de øvrige indgange, og udelukkende tjener formål for analysen. Vi betegner med  $P$  foreningsmængden af parrene fra alle niveauer.

Invarianten skal opretholdes under to operationer på listerne; *indsættelse* og *fjernelse* af et element fra listen. Betragt listen  $l_k = (z_1, \dots, z_m)$ .

**Indsættelse:** Lad elementet  $e$  være et nyt element, der skal indsættes imellem  $z_i$  og  $z_{i+1}$ . Betragt nu elementerne i en  $\log n$  omegn af  $z_i$ . Hvis der ingen



FIGUR 11.1. Lister og tilknyttede parringer

oversiddere er i denne omegn, kan vi blot lade  $e$  være en oversidder i  $l_k$  og rekursivt indsætte dette element i  $l_{k+1}$ -listen. Hvis der derimod er en oversidder  $z_D$  mindre end  $\log n$  fra  $z_i$ , er vi nødsaget til at foretage en “om-parring” i  $l_k$ . Ifølge invarianten er der i en  $\log n$  omegn af  $z_D$  kun parrede elementer. Disse par af elementer, samt  $z_D$  og  $e$ , kan nu omparres to og to, så hverken elementet  $z_D$  eller  $e$  er oversiddere.  $z_D$  udtages derfor rekursivt fra  $l_{k+1}$ . Samtidigt er det klart at afstanden mellem to oversiddere i  $l_k$  kun kan være forøget og invarianten derfor er bibeholdt. Endeligt bemærkes det, at maksimalt  $O(\log n)$  elementer i  $l_k$  blev omparret og maksimalt 1 element blev fjernet eller indsat i  $l_{k+1}$  under denne proces.

**Fjernelse:** Denne duale operation, hvor et element  $z_i$  skal udtages fra  $l_k$  kan ligeledes nemt realiseres. Hvis  $z_i$  er en oversidder fjernes  $z_i$  blot direkte fra  $l_k$  og rekursivt fra  $l_{k+1}$ . Hvis  $z_i$  derimod er parret med en nabo, f.eks.  $z_{i+1}$ , betragter vi det blot som en indsættelse af  $z_{i+1}$  i  $l_k$  uden parret  $(z_i, z_{i+1})$  jævnfør ovennævnte indsættelsesstrategi. Denne proces vil altså igen maksimalt omparre  $O(\log n)$  elementer i  $l_k$ , og foretage maksimalt 1 indsættelse eller fjernelse af et element i  $l_{k+1}$ .

Lad som nævnt  $h$  betegne antallet af lister. Vi vil nu kræve, at alle par  $(y_i, y_j) \in P$  skal opfylde kravet  $y_i + y_j = 0$ . Da en indgang kun forekommer i et enkelt par i  $P$ , kan dette krav naturligvis opfyldes. Der er to måder det kan ske på; enten sættes  $y_i = -1$  og  $y_j = 1$  eller også sættes  $y_i = 1$  og  $y_j = -1$ . Ved dannelsen af et par vælges en af disse to muligheder tilfældigt med lige stor sandsynlighed. I den tidligere beskrevne figur 11.1 har vi valgt at illustrere valget ved en orienteret kant imellem et par, således at kanten peger imod den indgang, der er -1.

Betragt nu et prefix af  $y$ ;  $(y_1, y_2, \dots, y_j)$ . Vi vil nu opad-begrænse den forventede størrelse af prefix-summen  $|\sum_{1 \leq i \leq j} y_i|$ . Vi kan inddele  $y_i$ 'erne ( $1 \leq i \leq k$ ) i tre grupper. Første gruppe er de  $y_i$ , der er 0. Næste gruppe er de  $y_i$ , der er parret med et  $y_k$ , hvor  $k \leq j$ . Den sidste gruppe benævnt  $M$ , består af de  $y_i$ , der er parret med en indgang efter  $y_j$ . Det er klart, at den nævnte sum kun afhænger af  $M$ .

På figur 11.1 illustrerer den stiplede linie et snit ved et prefix. Betragt de orienterede kanter snittet passerer. De indgange (knuder), der hører til disse kanter og ligger til venstre for snittet, udgør  $M$  for dette prefix. Da kanterne peger mod indgangen med værdi  $-1$ , vil prefix-summen ved et snit netop være antallet af kanter, der vender mod højre fratrukket antallet, der peger mod venstre. På figuren er prefix-summen ved det illustrerede snit derfor  $-1$ . Intuitivt gælder der, at enhver prefix-sum svarer til et snit i figuren, der passerer maksimalt  $h$  kanter som med lige stor sandsynlighed bidrager med  $-1$  eller  $1$  til prefix-summen. Herved kan vi begrænse den forventede absolutte størrelse af en prefix-sum, som kvadratroden af antallet af kanter snittet passerer, der maksimalt kan være antallet af niveauer. Mere præcist gælder der:

For ethvert prefix er  $|M| \leq h$ , hvor  $h$  er antal lister. I modsat fald må der være mindst to indgange  $y_i$  og  $y_{i'}$  med  $i < i'$ , der begge er parret med en indgang, der ligger efter  $y_j$  på samme niveau. Men da både  $y_i$  og  $y_{i'}$  vil ligge i samme liste, med  $y_i$  før  $y_{i'}$  er  $y_i$  dermed parret med en indgang, der ikke kan være en nabo - en modstrid. Da ingen af indgangene i  $M$  er parret med en anden indgang også i  $M$ , antager de alle værdien  $-1$  eller  $1$  med lige stor sandsynlighed og uafhængigt af hinanden. Ifølge lemma A.4 gælder der da:

$$E\left(\left|\sum_{y \in M} y\right|\right) = O(\sqrt{|M|}) = O(\sqrt{h}) = O\left(\sqrt{\frac{\log n}{\log \log n}}\right).$$

Vi er nu klar til at analysere, hvor mange **change**-kald for  $y$ , der kræves for hver opdatering i  $x$ . En opdatering i  $x$  svarer som beskrevet til en indsættelse/fjernelse af et element i  $l_1$ . Ifølge ovennævnte indsættelsesstrategi resulterer dette i  $O(\log n)$  omparringer for hvert af i alt  $O\left(\frac{\log n}{\log \log n}\right)$  niveauer, d.v.s. maksimalt  $O\left(\frac{\log^2 n}{\log \log n}\right)$  par dannes ved en sådan ændring i  $x$ , og for hver dannet par udføres maksimalt to **change**-operationer i  $y$ . Heraf følger de påståede grænser i sætningen. □

Sætning 11.2 sikrer altså at enhver prefix-sum (eller endda interval-sum) af  $y$  forventet er tæt ved nul. I et senere kapitel har vi behov for en lidt anden strategi, hvor vi i stedet ønsker, at prefix-summerne aldrig er mindre end 0, men til gengæld tillader vi, at prefix-summen godt må være op til  $O(\log n)$  stor. I ovennævnte bevis, bestod den randomiserede del i, at par af knuder  $(y_i, y_j)$  enten blev vægtet mod "venstre" eller "højre", d.v.s.  $y_i$  sættes lig  $-1$  og  $y_j$  lig  $1$  eller omvendt. Hvis vi i stedet vælger *altid* at "vende" parrene mod højre, d.v.s. sætte  $y_i = 1$  og  $y_j = -1$ , vil enhver prefix sum af  $y$  naturligvis være ikke-negativ og summens størrelse vil samtidigt stadig være begrænset af antallet af niveau'er, d.v.s.  $O(\log n / \log \log n)$ . Vi sammenfatter denne diskussion i nedestående sætning.

**SÆTNING 11.3** (Deterministisk dynamisk prefix balancering). *Lad  $x \in \{0, 1\}^n$  være en input-instans og  $y \in \{-1, 0, 1\}^n$  den korresponderende vektor, der altid skal opfylde korrespondancen (18) (og dermed  $\sum_{1 \leq i \leq k} y_i = \sum_{1 \leq i \leq k} x_i \pmod{2}$  for alle  $k$ ).*

*Der findes en deterministisk strategi, så et **change** i  $x$  kan afspejles ved worst-case  $O\left(\frac{\log^2 n}{\log \log n}\right)$  **change**-operationer i  $y$  og samtidigt opfylde:*

$$\forall k : 0 \leq \sum_{1 \leq i \leq k} y_i = O\left(\frac{\log n}{\log \log n}\right).$$

## Anvendelser af dynamisk prefix balancering

Vi er nu klar til at vise nedre grænser for Dyck-sprogene i **change**-modellen, hvor vi vil anvende forrige kapitels resultater. Vi starter med at benytte signed prefix sum-problemet til at vise grænser for **interval**<sub>D<sub>1</sub></sub> og **prefix**<sub>D<sub>1</sub></sub>.

SÆTNING 12.1. **prefix**<sub>D<sub>1</sub></sub> har nedre grænse på  $\Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ .

*Bevis.* Lad  $t(n)$  betegne tiden for **change**, **prefix**-operationerne for  $D'_1$ . Lad  $x = (x_1, x_2, \dots, x_n)$ ,  $x_i \in \{-1, 0, 1\}$  være en instans af signed prefix sum problemet som vi ønsker at løse via **change** og **prefix**-operationerne for  $D'_1$ . Vi opretholder nu to ens vektorer  $y^-$  og  $y^+$ .

$$y^\pm = b_1 \underbrace{\#\#\dots\#}_{n^{q^1}} b_2 \underbrace{\#\#\dots\#}_{n^{q^2}} \dots b_n \underbrace{\#\#\dots\#}_{n^{q^n}}$$

$$\text{hvor } b_i = \begin{cases} ( & \text{hvis } x_i = 1 \\ ) & \text{hvis } x_i = -1 \\ \# & \text{hvis } x_i = 0 \end{cases}$$

$q^1, \dots, q^n$  betegner delvektorer initielt bestående af  $n$   $\#$  tegn. Ideen er, at vi kan beregne **sum**( $k$ ) ved at sætte venstreparenteser i  $q^k$  i  $y^-$  og tilsvarende højreparenteser i  $y^+$  en efter en og parallelt indtil der forekommer et match lige efter  $q^k$  i en af vektorene. Hvis f.eks. **sum**( $k$ ) =  $h$ ,  $h \geq 0$  må prefixet frem til  $q^k$  i  $y^+$  udfra konstruktionen have et overskud på præcist  $h$  venstre-parenteser. D.v.s. at prefixet inklusive  $q^k$  i  $y^+$  først matcher, idet der er sat  $h$  parenteser, mens prefixet i  $y^-$  aldrig vil matche. Symmetrisk for  $h < 0$ . Det følger, at vi ved  $O(|h|)$  **change** og **prefix**-operationer kan beregne  $h$ . Bagefter "ryddes op" i det samme antal operationer, så  $y^+$  og  $y^-$  genetableres.

Reduktionen sikrer derved tiden  $O(t(n)|h|)$  for queriet **sum**( $k$ ) =  $h$  og  $O(t(n))$  for **change**. Fra sætning 11.1 følger hermed at  $t(n) \in \Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ .

□

SÆTNING 12.2. **interval**<sub>D<sub>1</sub></sub> har nedre grænse på  $\Omega\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ .

*Bevis.* Beviset er som før en reduktion af signed prefix sum-problemet. Lad  $t(n)$  betegne tiden for operationerne **change**, **interval** for  $D_1$  og lad  $x \in \{-1, 0, 1\}^n$  være en instans af signed prefix sum problemet. Vi opretholder vektoren:

$$y = b_n b_{n-1} \dots b_1 \underbrace{)) \dots)}_{2n}$$

$$\text{hvor } b_i = \begin{cases} ((, & \text{hvis } x_i = 1 \\ (\#, & \text{hvis } x_i = 0 \\ \#\#, & \text{hvis } x_i = -1 \end{cases}$$

Ideen er, at et query  $\mathbf{sum}(k)$  kan besvares ved at betragte en række intervaller, der alle starter ved positionen lige før  $b_k$  i  $y$ . Lad som før  $\mathbf{sum}(k) = h$ . Lad  $I(j)$  betegne delstrengen  $y_{[2(n-k)+1..2n+(k+j)]}$ , for  $j \in [-n..n]$ , d.v.s.

$$I(h') = b_k \dots b_1 \underbrace{)) \dots)}_{k+h'}$$

Da  $k+h$  netop er antallet af venstre-parenteser i  $b_k \dots b_1$ , jvf. korrespondancen til  $x_1 \dots x_k$ , gælder der,  $I(h') \in D'_1$  hvis og kun hvis  $h = h'$ . Dette anviser direkte en strategi for, hvorledes vi i  $O(|h|)$  queries kan returnere summen  $h$ . Vi tester fortløbende intervallerne for strengene  $I(0)$ ,  $I(-1)$ ,  $I(1)$ ,  $I(-2)$ ,  $I(2)$ ,  $\dots$  indtil et af disse intervaller  $I(h')$  matcher, og dermed giver os det ønskede resultat i tiden  $O(t(n)|h|)$ . Fra sætning 11.1 følger den påståede nedre grænse for  $t$ . □

### 12.1. En $\Omega\left(\frac{\log n}{(\log \log n)^2}\right)$ nedre grænse

I dette afsnit præsenterer vi en anden anvendelse af dynamisk prefix balancering, hvor vi opnår tæt på kvadratisk bedre grænser, nemlig  $\Omega(\log n / (\log \log n)^2)$  grænser for **2-interval** $_{D_1}$  og majoritet-prefix problemet som nævnt i kapitel 4. Grænsen for **2-interval** $_{D_1}$  belyser samtidigt en af forskellene ved de en-sidige og to-sidige Dyck-sprog som vi ikke hidtil har benyttet i nogen nedre grænser, hvilket vi vender tilbage til lidt senere.

SÆTNING 12.3. *Majoritet-prefix problemet har nedre grænse på  $\Omega\left(\frac{\log n}{(\log \log n)^2}\right)$ .*

SÆTNING 12.4. **2-interval** $_{D_1}$  *har nedre grænse på  $\Omega\left(\frac{\log n}{(\log \log n)^2}\right)$ .*

Grænsen for **2-interval** er direkte afledt af grænsen for majoritetsproblemet, og bibringer derfor en interessant sammenhæng imellem det velstuderede majoritets-problem og  $D_1$  i vores dynamiske kontekst. Samtidigt er denne reduktion den eneste af vores reduktioner, der udnytter asymmetrien ved de en-sidige Dyck-sprog, og kan derfor ikke benyttes for de to-sidige sprog.

*Bevis.*[Sætning 12.3] Beviset består i en reduktion, der benytter dynamisk prefix balancering på en ny måde. For at adskille operationerne for majoritet-prefix problemet og paritet-prefix problemet fra hinanden, vælger vi at betegne operationerne for paritet-prefix problemet ved; **change** $_{mod2}$  og **prefix** $_{mod2}$ , og for majoritet-prefix problemet ved; **change** $_{maj}$  og **prefix** $_{maj}$ .

Lad  $x \in \{0, 1\}^n$  være en input-instans for paritet-prefix problemet. Ideen er, at vi via lemma 11.3 kan simulere **change** $_{mod2}$  ved  $O(\log^2 n / \log \log n)$  kald af **change** $_{maj}$ , således at vi blot skal bruge  $O(\log \log n)$  kald af **prefix** $_{maj}$  for at simulere **prefix** $_{mod2}$ .

Et første skridt i reduktionen består i at opretholde vektoren  $y \in \{-1, 0, 1\}^n$ , så der for en konstant  $c$  og for alle  $k$  gælder:

$$0 \leq \sum_{1 \leq i \leq k} y_i \leq c \frac{\log n}{\log \log n}$$

og

$$(20) \quad \sum_{1 \leq i \leq k} y_i \pmod 2 = \sum_{1 \leq i \leq k} x_i \pmod 2,$$

hvor der pr. ændring i  $x$  foretages  $O(\log^2 n / \log \log n)$  ændringer i  $y$  jævnfør lemma 11.3.

Vi opretholder nu  $z \in \{0, 1\}$  korresponderende til  $y$  på følgende vis:

$$z = b_1 Q b_2 Q b_3 \dots Q b_n$$

hvor

$$b_i = \begin{cases} 00, & \text{hvis } y_i = -1, \\ 01, & \text{hvis } y_i = 0, \\ 11, & \text{hvis } y_i = 1 \end{cases}$$

$$\text{og } Q = \underbrace{00 \dots 0}_{2c \log n} \underbrace{11 \dots 1}_{2c \log n}.$$

Vektoren  $z$  kan altså dynamisk opretholdes for  $x$  ved  $O(\log^2 n / \log \log n)$  kald af **change**<sub>maj</sub> pr. **change**<sub>mod2</sub> kald for  $x$ . Lad nu **prefix**<sub>mod2</sub>( $k$ ) være et prefix-kald for  $x$  vi ønsker pariteten for. Betragt de  $c \log n$  prefix'er af  $z$ :

$$\begin{aligned} s_0 &= b_1 Q b_2 Q \dots b_k \\ s_1 &= b_1 Q b_2 Q \dots b_k 00 \\ &\dots \\ s_i &= b_1 Q b_2 Q \dots b_k \underbrace{0 \dots 0}_{2i} \\ &\dots \\ s_{c \log n} &= b_1 Q b_2 Q \dots Q b_k \underbrace{00 \dots 0}_{2c \log n} \end{aligned}$$

Ideen er nu, at fra et vist trin  $h$ ,  $0 \leq h \leq c \log n$ , har prefix'erne  $s_i$ ,  $i > h$  ikke en majoritet af 1-taller, mens prefixer  $s_i$ ,  $i \leq h$  alle har. Vi argumenterer som følger:

For et prefix  $s_i$  kan vi se bort fra mellemliggende  $Q$ -blokke, da de alle har lige stor forekomst af 0'er og 1'er. Tilbage er  $b_1 \dots b_k$  samt  $2i$  0'er. Det kan let vises (pr. definition af  $b_i$ 'erne), at antallet af 1'taller blandt  $b_i$ 'erne netop er  $k + Y$ , og dualt er antallet af 0'er  $k - Y$ , hvor  $Y = \sum_{1 \leq i \leq k} y_i \geq 0$ . Det følger heraf, at  $s_i$  har flere eller ligeså mange 1'er som 0'er hvis og kun hvis  $i \leq Y$ . Da  $Y$  samtidigt er mindre end eller lig  $c \log n / \log \log n < c \log n$ , findes prefixer  $s_i$ ,  $i > Y$ , der ikke har en majoritet af 1'er. Ved hjælp af **prefix**<sub>maj</sub> kan vi afgøre fra hvilket trin  $h$ , der gælder at  $s_i$ ,  $i > h$  ikke længere har majoritet af 1'taller, d.v.s. hvor  $h = Y$ . Idet vi benytter binær søgning efter dette  $h$  blandt de  $c \log n$  prefixer  $s_i$ ,  $0 \leq i \leq c \log n$ , skal vi for at finde dette  $h$  kun benytte  $O(\log \log n)$  kald af **prefix**<sub>maj</sub>. Da pariteten af  $Y$  er lig med et korrekt svar for **prefix**<sub>mod2</sub> jævnfør (20), konkluderer vi, at hvis  $t(n)$  er worst-case tiden for majoritet-prefix problemet, så kan vi løse paritet-prefix problemet i tid  $O(t(n) \log^2 n)$  for **change**<sub>mod2</sub>, og  $O(t(n) \log \log n)$  for **prefix**<sub>mod2</sub>. Fra sætning 9.1 følger da, at  $t \in \Omega(\log n / (\log \log n)^2)$  som ønsket. □

Vi vil nu vise sætning 12.4 ved en reduktion fra ovennævnte majoritet-prefix problem.

*Bevis.*[Sætning 12.4] Lad  $x \in \{0, 1\}^n$  være en input-instans for majoritet-prefix problemet. Oprethold  $y$  så

$$y = b_n b_{n-1} \dots b_1 \underbrace{)) \dots)}_n \underbrace{(( \dots (}_{n} b_n^R \dots b_1^R$$

hvor

$$b_i = \begin{cases} \#, & x_i = 0 \\ (, & x_i = 1 \end{cases}.$$

Svaret for majoritets-queriet  $\mathbf{prefix}(k)$  kan nu besvares ud fra  $y$  ved queriet:

$$(21) \quad \mathbf{2-interval}(n - k + 1, n + \lceil \frac{k}{2} \rceil + 1, 3n - \lceil \frac{k}{2} \rceil + 1..3n + k + 1),$$

idet

$$y_{[n-k+1..n+\lceil \frac{k}{2} \rceil+1]} y_{[3n-\lceil \frac{k}{2} \rceil+1..3n+k+1]} = b_k b_{k-1} \dots b_1 \underbrace{)) \dots)}_{\lceil \frac{k}{2} \rceil} \underbrace{(( \dots (}_{\lceil \frac{k}{2} \rceil} b_k^R \dots b_1^R \in D_1,$$

hvis og kun hvis antallet af venstreparenteser i  $b_k \dots b_1$  er større end eller lig  $\lceil \frac{k}{2} \rceil$ . Da der er ligeså mange 1'er i  $x_{[1..k]}$ , som venstreparenteser i  $b_k \dots b_1$ , må svaret på (21) korrespondere med svaret for  $\mathbf{prefix}(k)$  (der jo også svarer Ja, hvis antallet af 1'taller er større end eller lig  $\lceil \frac{k}{2} \rceil$ ). Fra sætning 12.3 følger den påståede nedre grænse. □

## Fremtidigt arbejde

Her til sidst vil vi kort beskrive flere interessante problematikker, hvor der efter vores mening eventuelt kan ligge muligheder for forbedring/udbyggelse af beskrevne resultater.

**De øvre grænser.** De deterministiske øvre grænser for dynamisk genkendelse af Dyck-sprog med flere sæt parenteser har alle tider på  $O(\log^3 \log^* n)$ . En faktor på  $O(\log^2 n \log^* n)$  skyldes her brugen af datastrukturen af Mehlhorn et al, og det er her fristende at formode, at dette kan forbedres betydeligt i forbindelse med en Las Vegas randomiseret udgave af denne datastruktur.

En anden spændende problematik er de dynamiske algorithmer for Dyck-sprogene med en type parentes. For f.eks.  $\mathbf{prefix}_{D'_1}$  har vi et lidt mere end kvadratisk stort gab mellem den præsenterede nedre grænse på  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  og den øvre grænse på  $O(\log n)$ . En udforskning af dette gab vil være interessant.

**Nedre grænser.** Figurene 4.1 og 4.2 i kapitel 4 lægger op til flere mulige forbedringer. En interessant problematik er problem-hierakiet:

$$\mathbf{member}_{D_1} \rightarrow \mathbf{prefix}_{D_1} \rightarrow \mathbf{interval}_{D_1} \rightarrow \mathbf{2-interval}_{D_1}$$

De har alle den fælles øvre grænse på  $O(\log n)$ , mens vores nedre grænser styrkes trin for trin. For  $\mathbf{member}_{D_1}$  og  $\mathbf{prefix}_{D_1}$  har vi ikke vist nogen nedre grænser i dette speciale, men har henvist til grænsen på  $\Omega(\frac{\log \log n}{\log \log \log n})$  vist i [FHM<sup>+</sup>95]. Herefter sker der et spring til  $\Omega(\sqrt{\frac{\log n}{\log \log n}})$  grænsen for  $\mathbf{interval}_{D_1}$ , og endnu et spring til grænsen for  $\mathbf{2-interval}_{D_1}$ , der kun lader et gab på  $O((\log \log n)^2)$  stå tilbage mellem denne og den øvre grænse. Det er klart, at det her er specielt interessant at se på, hvorvidt vores reduktioner er for svage - d.v.s. er det muligt at få dele af dette hieraki til at kollapse?

Et andet lidt pudsigt emne er, at den præsenterede nedre grænse for  $D'_1$  med  $\mathbf{prefix}$ -operationen er langt bedre end den postulerede for  $D_1$ , mens det omvendt er  $D_1$  vi har vist den stærkeste nedre grænse for med hensyn til  $\mathbf{2-interval}$ -operationen. Dette kan virke unaturligt, og giver blandt andet anledning til at overveje eventuelle sammenhænge imellem de en-sidige og to-sidige Dyck-sprog.

Endeligt er der den dynamiske prefix balancerings. Det er naturligvis interessant om der er andre anvendelser end de beskrevne for Dyck-sprogene og majoritet-prefix problemet, eventuelt via raffinering eller generalisering af teknikken.

**Andre tilbageværende projekter.** Som nævnt i kapitel 4, har vi ikke beskrevet algoritmer for f.eks. **match** $_{D_k}$ , **prefix** $_{D_k}$  og **interval** $_{D_k}$ . Vores forventning er, at disse operationer kan realiseres ved lette generaliseringer af datastrukturen for problemet **member** $_{D_k}$ . Et andet emne vi heller ikke har berørt, er en undersøgelse af datastrukturens praktiske værdi, f.eks. i tekst-editorer.

**Acknowledgement.** Vi ønsker først og fremmest at rette en stor tak til vores vejleder Gudmund S. Frandsen, der under hele forløbet har udvist stor interesse og engagement for projektet. Vi takker også Peter Bro Miltersen og Thore Husfeldt for et godt samarbejde under arbejdet med [FHM<sup>+</sup>95]. Endelig retter vi en tak til Gerth Stølting Brodal for hans interesse og mange berigende diskussioner.

## Appendiks

PROPOSITION A.1. *Lad  $n \geq k \geq 0$ . Da vil  $\binom{n}{k} \leq \frac{n^k}{k!}$ .*

PROPOSITION A.2. **(Stirlings formel)**

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right).$$

PROPOSITION A.3. **(Chebychevs ulighed)**

*Lad  $X$  være en stokastisk variabel med middelværdi  $\mu$  og varians  $\sigma^2$ . Da gælder for alle  $t > 0$ ,*

$$Pr(|X - \mu| > t \cdot \sigma) \leq \frac{1}{t^2}.$$

PROPOSITION A.4. *Lad  $X_1, X_2, \dots, X_n$  være  $n$  parvis uafhængige stokastiske variable, der antager værdien  $-1$  eller  $1$  med lige stor sandsynlighed. Da gælder:*

$$E\left[\left|\sum_i X_i\right|\right] = O(\sqrt{n}).$$

*Bevis.* Lad  $X = \sum_i X_i$ . Da  $E[X_i] = 0$  og  $Var(X_i) = 1$  for alle  $i$  gælder der, at  $E[X] = 0$  og  $Var(X) = n$ , da variableerne er parvis uafhængige.

Lad  $p_k$  betegne sandsynligheden  $Pr(|X| > k\sqrt{n})$ . Fra Chebychevs ulighed har vi  $p_k \leq \frac{1}{k^2}$  for  $k \geq 1$ .

$$\begin{aligned} E[|X|] &\leq \sum_{k=0}^n (k+1)\sqrt{n}Pr(k\sqrt{n} \leq |X| < (k+1)\sqrt{n}) \\ &\leq \sqrt{n} \sum_{k=0}^n (k+1)(p_k - p_{k+1}) \\ &= \sqrt{n}(1(p_0 - p_1) + 2(p_1 - p_2) + 3(p_2 - p_3) + \dots + n(p_n - p_{n+1})) \\ &\leq \sqrt{n} \left(\sum_{k=0}^n p_k\right) \\ &\leq \sqrt{n} \left(1 + \sum_{k=1}^n \frac{1}{k^2}\right) \\ &< \sqrt{n} \left(1 + \frac{\pi^2}{6}\right). \end{aligned}$$

□



## Litteratur

- [AM88] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In John H. Reif, editor, *1988 Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, pages 81–90. Springer-Verlag, 1988.
- [AV79] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. of Comput. Syst. Sci.*, 18(2):155–193, 1979.
- [CV86a] R. Cole and U. Vishkin. Approximate parallel scheduling. part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. Ultracomputer Note 110, October 1986. Also Computer Science Technical Report 244. *SIAM Journal on Computing*, 17(1988), 128-142.
- [CV86b] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proc. 18th ACM Symposium on Theory of Computing (STOC)*, pages 206–219, 1986.
- [CV86c] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.
- [FHM<sup>+</sup>95] Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum. Dynamic algorithms for the dyck languages. Research Series RS-95-1, BRICS, Department of Computer Science, University of Aarhus, January 1995. 21 pp.
- [FS89] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Ann. Symp. on Theory of Computing (STOC)*, pages 345–354, 1989.
- [GI91] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319, September 1991.
- [GPS87] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. of the 19th Ann. Symp. on Theory of Computing (STOC)*, pages 315–324, 1987. Journal version: [GPS88].
- [GPS88] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal of Discrete Mathematics*, 1(4):434–446, 1988.
- [GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 8–21. IEEE Computer Society, 1978.
- [Har78] Michael A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, 1978.
- [JaJ92] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, Massachusetts, 1992.
- [KR87] Richard M. Karp and Michael O. Rabin. Efficient randomised pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, March 1987.
- [KW84] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proc. 16th ACM Symposium on Theory of Computing (STOC)*, pages 266–272, 1984.
- [Lub85] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proc. 17th ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1985.
- [LZ77] Richard J. Lipton and Yechezkel Zalcstein. Word problems solvable in logspace. *Journal of the ACM*, 24(3):522–526, 1977.
- [Mil92] Peter Bro Miltersen. On-line reevaluation of functions. Technical Report DAIMI PB-380, Computer Science Department, Aarhus University, 1992.
- [MKS66] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial Group Theory*, volume 13 of *Pure and Applied Mathematics*. Interscience Publishers, 1966.

- [MSU94] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *Proc. 5th Ann. Symp. on Discrete Algorithms (SO-DA)*, pages 213–222. ACM-SIAM, 1994.
- [MSVT94] Miltersen, Subramanian, Vitter, and Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130, 1994.
- [Ove83] Mark H. Overmars. *The design of dynamic data structures*, volume 156 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1983.
- [Rot65] J. Rotman. *The theory of groups: An introduction*. Allyn and Bacon. Boston, 1 edition, 1965.
- [vEB75] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Symposium on Foundations of Computer Science (16th)*, pages 75–84, 1975.
- [Yao77] A. C. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.
- [Yao81] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.