Lange Perioden in Subtraktions-Spielen

Dissertation zur Erlangung des Grades eines Dr. math.
der Fakultät für Mathematik der Universität Bielefeld

vorgelegt von

Dipl.-Math. Achim Flammenkamp

im November 1996

Gewidmet den
Neugierigen

Zusammenfassung

Lange Perioden in Subtraktions-Spielen*

Diese Arbeit legt meine Ergebnisse der Untersuchung von Subtraktions-Spielen mit Blickrichtung auf lange Perioden dar. Schärfere untere und obere Schranken der Periodenlängen, als bisher bekannt sind, werden bewiesen. Der Begriff der symmetrischen Zugmengen wird eingeführt und deren Bedeutung herausgestellt. Ein Struktur-Verifikations-Algorithmus wird vorgestellt, auf einem Rechner implementiert und zum erstmaligen Beweis von Gewinn/Verlust-Folgen einiger Familien von Subtraktions-Spielen benutzt. Viele Vermutungen über Periodenlängen werden formuliert und gut fundiert. Insbesondere gibt es starke Indizien, daß Familien von Subtraktions-Spielen existieren, deren Periodenlängen exponentiell in einem freien Parameter $n \in \mathbb{N}$ wachsen, aber deren Zugwerte nur linear mit diesem n anwachsen.

*Ein Teil der Ergebnisse dieser Arbeit ist zur Veröffentlichung in der Zeitschrift "Theoretical Computer Science" eingereicht worden.

Vorwort

Das Zustandekommen dieser Arbeit, insbesondere das Ergründen dieser Thematik, verdanke ich unserem gemeinsamen Interesse an Nim-Spielen. Während für Ingo Althöfer das Thema „Lange Perioden in Subtraktions-Spielen“ eine Fortführung einer von ihm betreuten Diplomarbeit war, sagte mir dieses Gebiet nicht nur von seiner zahlentheoretischen und kombinatorischen Ebene her zu. Es war auch eine Herausforderung, auf einem für mich neuen Gebiet, dem der Subtraktions-Spiele, nach sinnvollen Strukturen zu suchen und diese, soweit es in meinen Fähigkeiten stand, zu analysieren und die gefundenen Eigenschaften zu beweisen. Ich möchte noch anmerken, daß ich mich bereits in der Vergangenheit intensiv mit „Grundys-Spiel“ beschäftigt hatte und Resultate zu „Whythoffs-Spiel“ erzielt hatte.

Besonderer Dank gebührt meinem Betreuer, Prof. Dr. Ingo Althöfer, der meine Forschung mit wertvollen Anregungen unterstützte. Auch beeinflußte er den Stil dieser Arbeit maßgeblich. Ferner möchte ich noch den weiteren Helfern, namentlich Ingolf Koch und Udo Sprute, für ihr intensives Korrekturlesen der Arbeit danken. Und “last but not least” möchte ich Dr. Torsten Sillke hervorheben, der zu verschiedenen Vorversionen dieser Arbeit seine Meinung mit mir ausdiskutierte.

Jena, im Oktober 1996

Inhaltsverzeichnis

Einführung

Einleitung

Die Menschen haben sich seit altersher die Zeit mit Denk-Spielen vertrieben. Diese Spiele kann man nach ihrem den Spielern verfügbaren Informationsgehalt unterteilen. Z. B. reine Wettspiele, wie Pferderennen, Glücksspiele wie Roulette, psychologische Geschicklichkeitsspiele wie Papier-Stein-Schere ein bekanntes darstellt, Kartenspiele wie Skat und reine Strategiespiele, wie Schach, in denen alle Spieler vollständige Information über den jeweiligen Spielzustand haben. Ich möchte mich hier mit einer besonders einfachen Klasse von Strategiespielen beschäftigen. Nur zwei Spielern nehmen an dem Spiel teil, die Spielregeln sind für beide Spieler gleich und sie haben vollständige Information über den aktuellen Spielzustand. Ferner ist es unnötig sich zurückliegende Spielzustände zu merken, aber dennoch soll das Spiel anspruchsvoll, sprich nichttrivial sein. Dem Mathematiker ist bekannt, daß einfachste Fragestellungen, deren Beantwortung beliebig schwer ist, meistens in der Zahlentheorie auftreten. So ist es nicht verwunderlich, daß auch ein klassisches „Wegnehm“-Spiel, welches schon im Mittelalter bekannt war [Bac1612] [BC39, p.28, p. 36–40], sich elementar zahlentheoretisch formulieren läßt.

Die klassische Variante ist folgende: Beide Spieler haben jeweils einen genügend großen Vorrat an, sagen wir, Steinchen. Von diesen legen sie abwechselnd auf einen gemeinsamen, zu Anfang leeren Haufen eine beliebige Anzahl, aber höchstens elf. Gewonnen habe der Spieler, der als erster die Steinchenzahl im gemeinsamen Haufen auf sagen wir 100 erhöht. Der Leser wird erkennen, das unabhängig von den hier gewählten Werten 100 und elf, das Spiel doch recht trivial ist, sobald er die Gewinnstrategie „Ergänze stets zu dem nächst größeren Wert der Form $100 - (1 + 11)n$“ erkannt hat. Dies liegt im wesentlichen daran, daß man ein lückenloses Spektrum von Zügen von Eins bis zu einem Maximum, in unserem Fall elf, hat. Ändern wir dies, in dem wir eine feste, aber willkürliche Menge von zulässigen Zugzahlen vorgeben, wird dieses äußerst einfache Spiel sofort sehr schwierig, was das Auffinden seiner Gewinnstrategie angeht. Nur eine technische Änderung ist dagegen die Konvention, statt mit einem leeren Haufen anzufangen, der bis zu einer vorgegebenen Größe anwächst, mit einem Haufen vorgegebener Größe zu starten und dann zu versuchen diesen Haufen durch abwechselndes Steinchenentfernen als erster Spieler am Zug zu leeren.

Grundlagen

Betrachte das folgende Zwei-Personen-Spiel: Gegeben sei ein Haufen der Größe $n \in \mathbb{N}_0$ und k Zahlen $s_1 < s_2 < ... < s_k \in \mathbb{N}$, welche die zulässigen Züge beider Spieler definieren. Die Spielregel besagt, daß die Spieler abwechselnd die momentane Haufengröße um eine Zahl, gewählt aus der Zugliste $L = (s_1, s_2, \ldots, s_k)$, verringern müssen. Der Buchstabe L wird ebenso für die Menge der Züge $L = \{s_1, s_2, \ldots, s_k\}$ benutzt, wenn die Ordnungsrelation der Züge unwichtig oder die Sprechweise mit Mengen allgemein üblich ist. Des weiteren möchte ich mit $\max L$ generell den größten Zug in L bezeichnen.

Das Spiel endet, wenn die Haufengröße beim nächsten Zug negativ würde, und der Spieler, der nun am Zug ist, wird als Verlierer bezeichnet. Dies kann man anschaulich als „Man kann nicht mehr entfernen, als vorhanden ist.“ interpretieren. Solch ein Spiel wird als Subtraktions-Spiel bezeichnet.

Wenn es optimal gespielt wird, das heißt, wenn der Spieler am Zug auf eine Verlustposition des Gegners zieht, wann immer er kann, dann bestimmt bereits die Haufengröße zu Beginn, ob der anziehende oder der nachziehende Spieler das Spiel gewinnen wird. Für eine gegebene Menge L nennen wir die Haufengröße n eine $\mathcal{P}$- bzw. $\mathcal{N}$-Position, je nachdem, ob der erste Spieler am Zug einen Gewinn erzwingen kann (*„positiv“*) oder nicht(*„negativ“*) [BCG82, p. 83, V 1].[1] Betrachten wir nun die Folge dieser $\mathcal{P}$- und $\mathcal{N}$-Positionen, wobei n die natürlichen Zahlen durchläuft. Wenn wir eine $\mathcal{P}$-Position als 1 und eine $\mathcal{N}$-Position als 0 kodieren und mit $v_L(n)$ den Wert der Haufengröße n bei gegebener Zugmenge L bezeichnen, können wir

Beobachtung 1

$$v_L(n) \quad = \quad 1 - \prod_{\substack{s \in L: \\ s \leq n}} v_L(n-s) \qquad \textit{für alle } n \in \mathbb{N}_0$$

berechnen.[2] Dies läßt sich elegant in Boolescher Logik als `NAND`-Operator formulieren und kann daher mit Hilfe des logischen `AND` berechnet werden.

[1] Genau genommen steht im original angloamerikanischen Text $\mathcal{P}$ für “previous” und $\mathcal{N}$ für “next” und bezeichnet einen von zwei Spieler am Zug. Diese Abkürzung gibt leider intuitiv noch gar keine Auskunft über den Gewinn- oder Verluststatus des Spielers. Ferner versagt diese Veranschaulichung völlig, wenn man auch Positionen mit einem Unentschieden hat. Daher wählte ich die einprägsamere, sprachübergreifendere und verallgemeinerungsfähige *„positiv“/ „negativ“* Interpretation.

[2] Das leere Produkt ist als 1 definiert. Generell liefert ein elementweise definierter, assoziativer und kommutativer Operator: $D \times D \to D$, der auf endliche Teilmengen seines Definitionsbereiches verallgemeinert wird, angewendet auf die leere Menge, sein neutrales Element. Z. B. hat daher der ggt von $\{\}$ den Wert 0 oder AND$\{\}$ ist `TRUE`. Andererseits

Statt das Produkt der Zahlen $v_L(n-s)$ zu berechnen, werden die entsprechenden Wahrheitswerte, `TRUE` $\equiv 1$ und `FALSE` $\equiv 0$, mit *und* verknüpft. Sobald man auf einen Wert `FALSE` stößt, das heißt eine Verlustposition, kann das Verknüpfen natürlich beendet werden, da der zu berechnende Wert an der Position n damit schon feststeht. Diese Überlegungen werden relevant, wenn man, wie in dieser Arbeit, solche Folgen $(v_L(n))$ mit einem Computer berechnet, und dies effizient geschehen soll.

Die Folge $(v_L(n))_{n=0}^{\infty}$ muß nach höchstens $2^{\max L}$ Werten periodisch werden [Knu69, p. 25–32]. Dies ist einsichtig, da $v_L(n)$ höchstens von den $\max L$ vorherigen Werten vor der Position n abhängt und diese jeweils nur 2 mögliche Werte annehmen können.

Definition 2
Die Periodenlänge $p(L)$ des Spiels L oder der Folge v_L ist definiert als die kleinste Zahl $p \in \mathbb{N}$, so daß es ein $q \in \mathbb{N}_0$ gibt mit $v(n+p) = v(n)$ für alle $n \geq q$.

Definition 3
Das minimale q zu einem $p(L)$ aus Definition 2 wird die Vorperiodenlänge $q(L)$ der Folge v_L genannt, und als Vorperiode wird die Teilfolge der ersten q Werte dieser Folge bezeichnet.

Als Beispiel diene das Subtraktions-Spiel $L = \{3, 7, 8\}$:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$v(n)$	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1	1	0	0	1	1
Gewinnzüge	–	–	–	3	3	3	–	7	7,8	3,7,8	8	–	–	7	3,8	3	–	–	7	3,7,8
	Vorperiode								Periode					Periode					Pe	

n	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
$v(n)$	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	1
Gewinnzüge	3,8	–	–	7	3,7,8	3,8	–	–	7	3,7,8	3,8	–	–	7	3,7,8	3,8
	riode	Periode					Periode					Period				

Wie man hier außerdem sieht, setzt die Periodizität mit $p = 5$ in den Gewinnzuglisten erst später als nach der Vorperiodenlänge q von 8 ein.[3] Ferner liegt ab $n \geq q + p + \max L - 1 = 20$ für die Berechnung des $v(n)$-Wertes dieselbe Situation wie für die Berechnung des Wertes $v(n-5)$

liefert $\max\{\}$ auf $\mathbb{N}$ definiert den Wert 1, ist aber undefiniert für $\mathbb{Z}$.

[3]Sie muß natürlich nach spätestens $q(L) + \max L$ einsetzen.

vor. Somit greift ab jetzt die Aussage unmittelbar vor Definition 2 mit $\max L = 8$ für unser Beispiel.

Da auf dieser Seite noch so viel Platz war, hier zum Experimentieren ein C-Quellcode, welcher **nicht effizient**, aber dafür an unsere Notation und theoretischen Ausführungen angelehnt ist. Kompiliert und mit den Zügen als seinen Parametern aufgerufen, gibt es $(v_L(n))_{n=0}^{q+p+\max L}$, $p(L)$ und $q(L)$ explizit aus.

```
#include <stdio.h>
#define  MAX_POS   100000
#define  MAX_CARD     100
unsigned  v[MAX_POS];

int main(unsigned argc, char *argv[])
{
   unsigned  i, k, n, p, q;
   unsigned  s[MAX_CARD];
   /***  read all moves ***/
   s[0]=0;
   for (k=1;k<argc;k++)
      if (k >= MAX_CARD)
         return  fprintf(stderr,"sorry, at most %u moves allowed\n",MAX_CARD);
      else if (1 != sscanf(argv[k],"%u",&s[k]))
         return  fprintf(stderr,"parameter no number: %s\n",argv[k]);
      else if (s[k] <= s[k-1])
         return  fprintf(stderr,"%u. move violates orderrelation\n",k);
   if (s[k-=1] >= MAX_POS)
      return  fprintf(stderr,"sorry, moves must be < %u\n",MAX_POS);
   /*** k different ordered moves in L = s[] ***/

   for (n=0;n<MAX_POS;n++)
   {  /***  compute next value ***/
      v[n]=0;
      for (i=1;i<=k;i++)
         if (s[i] > n)
            break;
         else if (v[n-s[i]] == 0)
         {  v[n]=1;  break;
         }
      fprintf(stderr,"%u",v[n]);
      /***  check all possible periodlengths ***/
      for (p=1;p+s[k]-1<=n;p++)
      {  for (i=0;i<s[k];i++)
            if (v[n-i] != v[n-p-i])
               break;
         if (i==s[k])
         { /*** period found ***/
            q= n-p-(i-1);
            fprintf(stdout," p(L)=%u  q(L)=%u\n",p,q);
            return 0;
         }
      }
   }
   fprintf(stdout,"sorry, p+q+s[k] > %u\n",MAX_POS+2);
   return 1;
}
```

Der Fall $|L| \leq 2$

Im Fall $|L| = 1$ ist das Spiel trivial, da kein Spieler eine Zugwahlmöglichkeit hat. Es ist $p(L) = 2 \max L$ und $q(L) = 0$, weil die Folge der $\mathcal{P}$- und $\mathcal{N}$-Positionen von Anfang an abwechselnd aus $\max L$ Verlust-, gefolgt von $\max L$ Gewinnpositionen besteht.

Auch für $|L| = 2$ sind diese Spiele vollständig analysiert [BCG82, p. 69 – 70, V 3]. Die wesentlichen Aussagen sind:

Satz 4
Für alle $L = (s_1, s_2)$ gilt:

- $q(L) = 0$
- $p(L) = \begin{cases} 2s_1 & \textit{falls} \quad 2s_1 \mid s_2 + s_1 \\ s_2 + s_1 & \textit{falls} \quad 2s_1 \nmid s_2 + s_1 \end{cases}$ [4]
- *Die Periode der Länge $p(L)$ besteht in ihren ersten $p(L) - s_1$ Positionen abwechselnd aus s_1 $\mathcal{N}$- und s_1 $\mathcal{P}$-Positionen, gefolgt von s_1 $\mathcal{P}$-Positionen am Schluß.*

Wenn s_2 kein ungerades Vielfaches von s_1 ist, haben alle Gewinn- und Verlustabschnitte der $\mathcal{P}$-$\mathcal{N}$-Folge die Länge s_1 — bis auf den vorletzten Abschnitt, der die Länge $s_2 \bmod s_1$ hat.

[4] Hier bedeutet $a|b$, daß es ein $z \in \mathbb{Z}$ gibt mit $a \cdot z = b$ — meint also die zahlentheoretische „teilt"-Relation. Lese ebenso $\nmid$ als „teilt nicht".

Der Fall $|L| = 3$

Der Unterfall $s_1 + s_2 = s_3$ ist ebenfalls in [BCG82, p. 70 – 72, V 3] gelöst. Eine Analyse der Folge der Gewinn- und Verlustpositionen ergibt:

Satz 5
Für alle $L = (s_1, s_2, s_3 = s_1 + s_2)$ *mit* $r := (s_3 \bmod 2s_1) - s_1$ *gelten die folgenden drei Aussagen:*

- $q(L) = 0$
- $p(L) = \begin{cases} 2s_2 - r & r \le 0 \\ \frac{s_1(2s_2+r)}{\mathrm{ggt}(s_1,s_2)} & r > 0 \end{cases}$
- *Die $\mathcal{N}$-Positionen sind genau diejenigen mit Index* $i + \lfloor \frac{i}{s_1} \rfloor s_1 + \lfloor \frac{2i}{s_2+|r|} \rfloor s_2$ *für alle* $i \in \mathbb{N}_0$.

Daher nehme ich für alle weiteren Aussagen mit $|L| = 3$ an, daß $s_1 + s_2 \neq s_3$ ist. Ich untersuchte alle Spiele mit $s_3 < 256$ und viele weitere mit größerem s_3. Aus den daraus resultierenden Werten der Perioden und Vorperioden dieser Spiele wurde ich angeregt, einige Vermutungen aufzustellen.

Vermutung 6
Für alle $L = (s_1, s_2, s_3)$ *mit* $s_1 + s_2 \neq s_3$ *gilt*

- $q(L) < s_2 s_3$
- $p(L) \mid s_1 + s_2$ *oder* $p(L) \mid s_1 + s_3$ *oder* $p(L) \mid s_2 + s_3$
- *Aus* $p(L) \neq s_1 + s_2$ *und* $p(L) \neq s_1 + s_3$ *und* $p(L) \neq s_2 + s_3$ *folgt, daß*
  entweder p genau zwei von jenen Summen teilt
  und gleich dem ggt *dieser beiden Summen ist*
  oder p alle drei Summen teilt und gleich $2\,\mathrm{ggt}(s_1, s_2, s_3)$ *ist.*
  In diesem Fall müssen s_1, s_2 und s_3 ungerade sein.

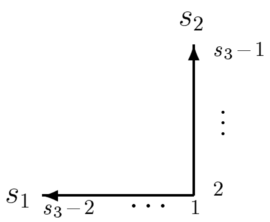
Um einen Eindruck der Verteilung der Periodenlängen von L zu bekommen, habe ich exemplarisch für einige s_3 die auftretenden Periodenlängen für alle $s_1 < s_2 < s_3$ dargestellt. Dazu habe ich die folgenden fünf Fälle unterschieden:

0. $s_1 + s_2 = s_3$

1. – 4. $s_1 + s_2 \neq s_3$ und

1. $p = s_1 + s_2$
2. $p = s_1 + s_3$
3. $p = s_2 + s_3$
4. $p \notin \{s_1 + s_2, s_1 + s_3, s_2 + s_3\}$

Für alle diese „Bitmap-Bilder“ ist auf der horizontalen Achse s_1 nach links wachsend und auf der vertikalen Achse s_2 nach oben zunehmend wie folgt



aufgetragen:

Am folgenden Bild der Vorperiodenlängenverteilung, kann man gut den Beweisaufwand mittels algebraischer Entwicklung in s_1, s_2 und s_3 für die Struktur der Folge der $v(n)$-Werte für ein festes „Gebiet“ abschätzen. Für den einfachsten Fall der weißen Gebiete links der Geraden $2s_1 = s_2$, dort ist $q(L) = 0$, konnte ich so die Struktur der $\mathcal{P}$-$\mathcal{N}$-Folge von Hand beweisen, da hier die Fallunterscheidungsanzahl sehr klein und konstant bleibt.

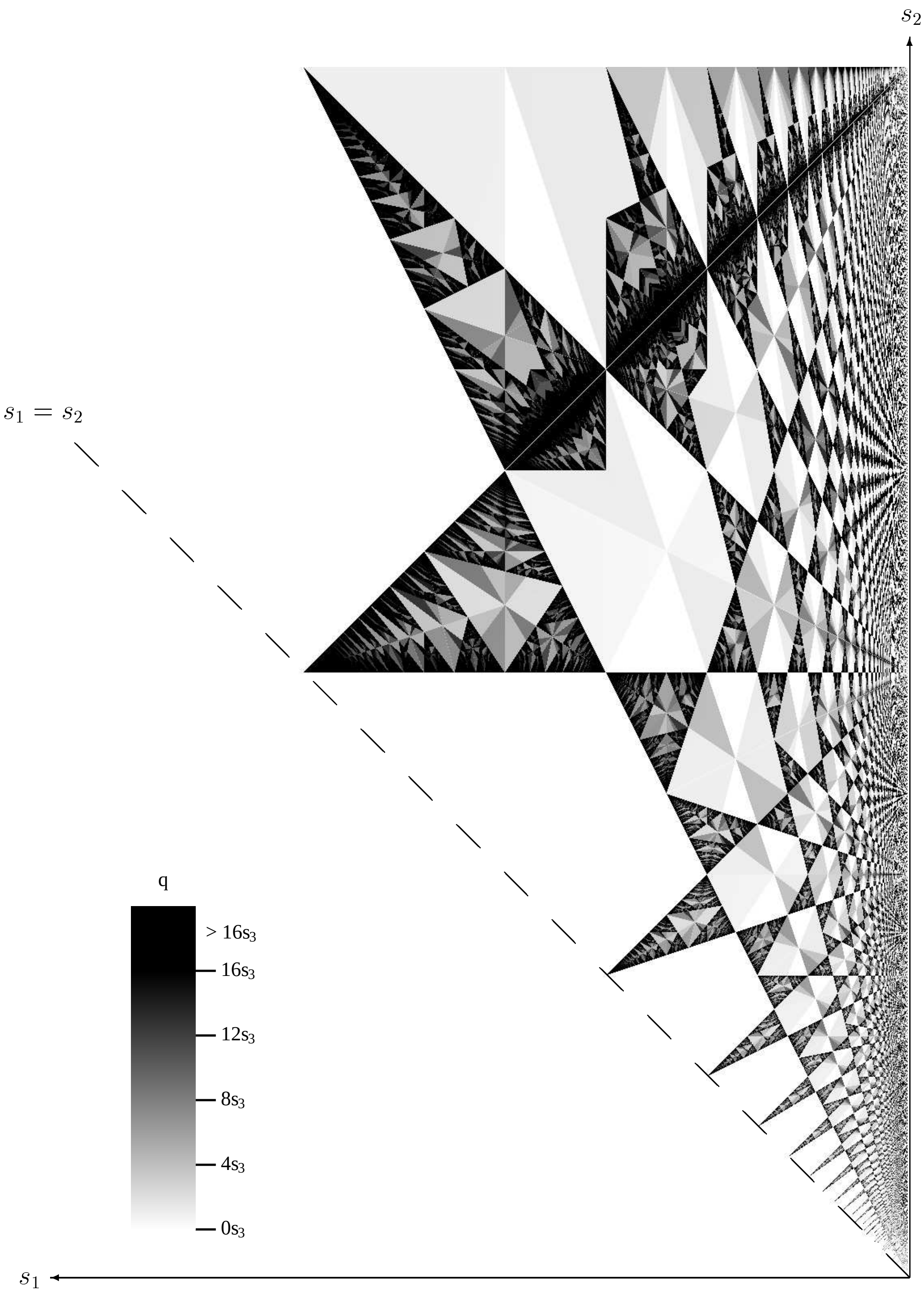


Bild i $s_3 = 1499$ wachsende Vorperiodenlängen sind linear als dunkler werdende Grauwerte aufgetragen.

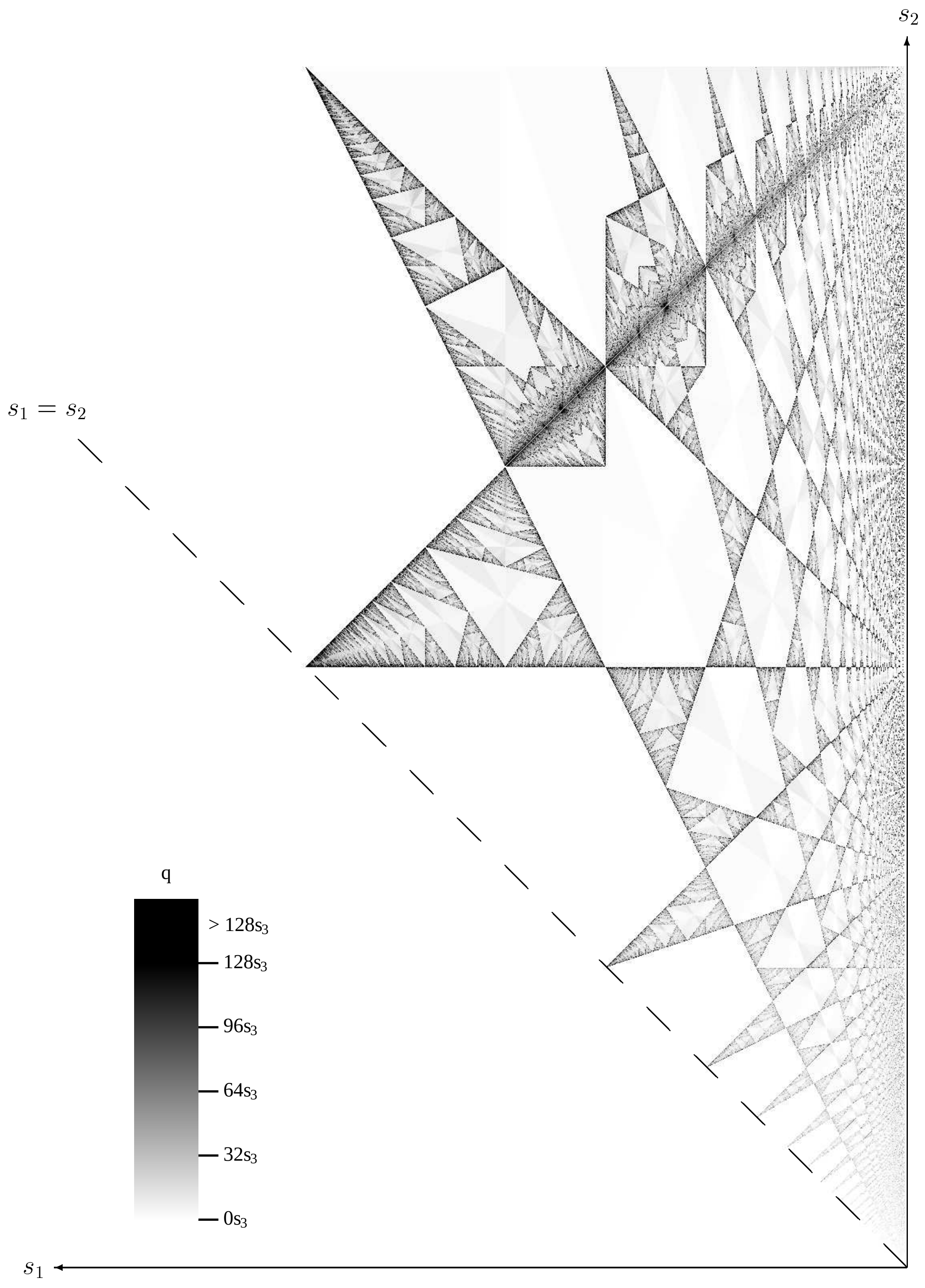


Bild ii $s_3 = 1499$ wachsende Vorperiodenlängen sind linear als dunkler werdende Grauwerte aufgetragen.

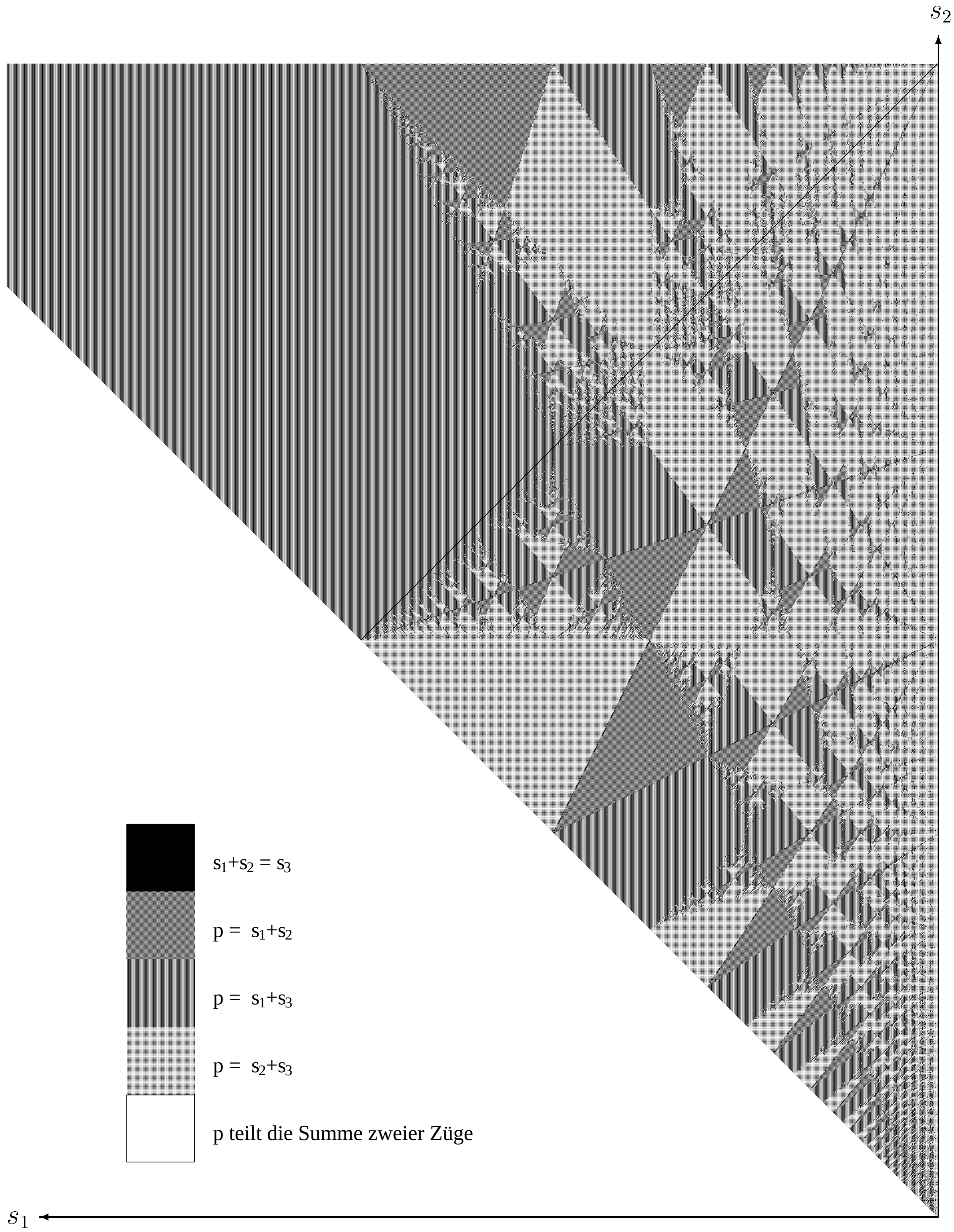


Bild iii $s_3 = 1024$ Die verschiedenen Bereiche der Periodenlängen. Hellgrau ist $p = s_2 + s_3$, grau ist $p = s_1 + s_3$, dunkelgrau ist $p = s_1 + s_2$, schwarz ist die Diagonale $s_1 + s_2 = s_3$, und alle anderen Werte sind weiß .

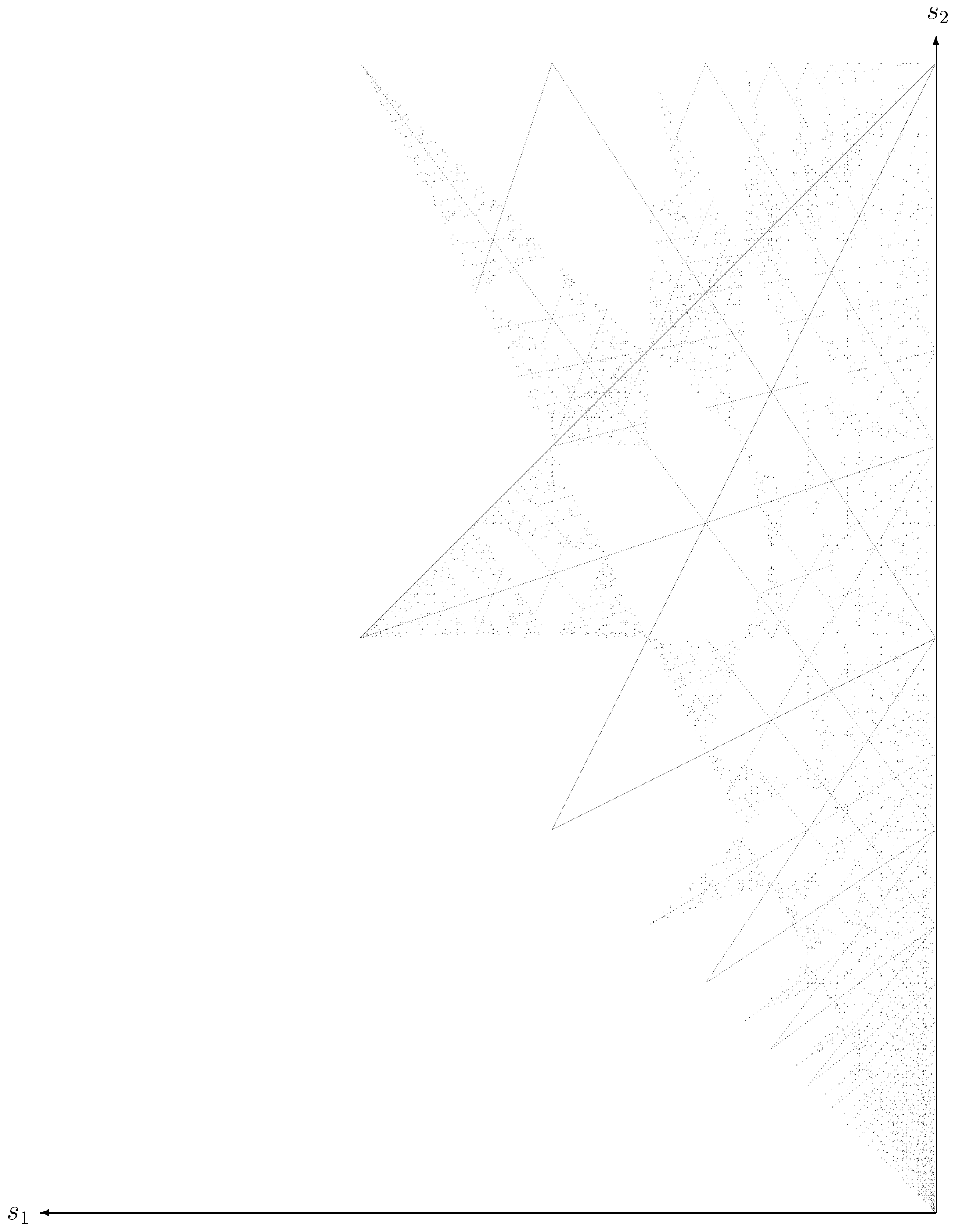


Bild iv $s_3 = 2048$ Periodenlängen ungleich der Summe von zwei Zugwerten.

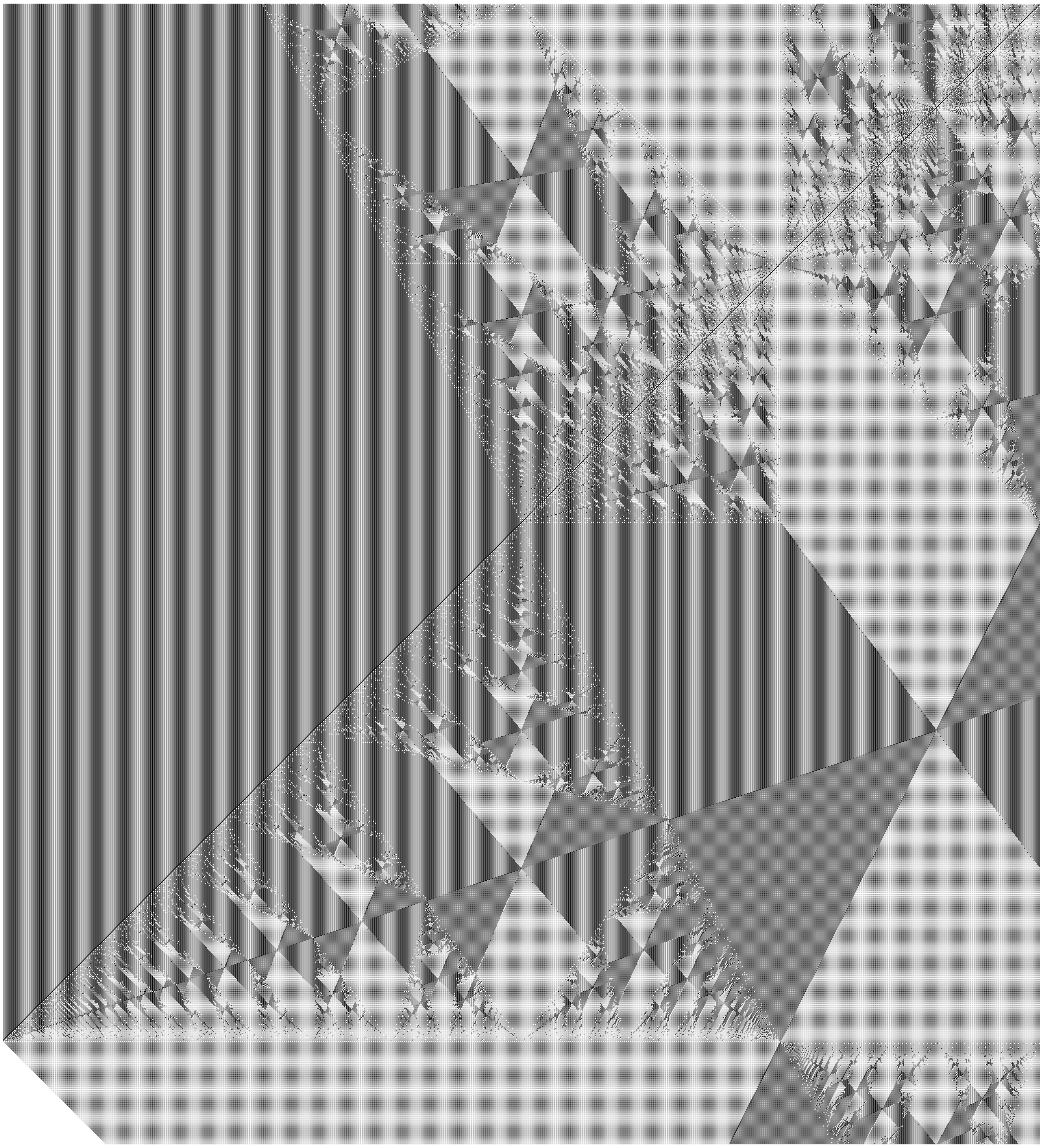


Bild v $s_3 = 3001$ Periodenlängen im Bereich von $500 \leq s_1 \leq 1500$ und $900 \leq s_2 \leq 2000$.

$s_1+s_2 = s_3$

$p = s_1+s_2$

$p = s_1+s_3$

$p = s_2+s_3$

p teilt die Summe zweier Züge

Die sich andeutende fraktale Struktur scheint in ihrer Form unabhängig von s_3 und mit wachsendem s_3 immer stärker ausgeprägt zu sein. Es würde daher für alle 3-Zugspiele ein Diagramm mit den Achsen $\frac{s_1}{s_3}$ und $\frac{s_2}{s_3}$ genügen, da sich ja $p(L)$ linear mit dem $\text{ggt}(s_1, s_2, s_3)$ skaliert. Ich habe für Zweierpotenz- und verschieden Primzahlwerte s_3 den relativen Flächenanteil der Periodenlängentypen bestimmt. Der 4. Fall wurde noch so unterteilt, daß $p(L)$ entweder zwei oder drei der Summen teilt. Die Spalte *gesamt* gibt die Anzahl aller 3-Zug-Spiele mit $\max L = s_3$ an.

s_3	gesamt	$s_1+s_2=s_3$	$p=s_1+s_3$	$p=s_2+s_3$	$p=s_1+s_2$	$p \mid 2\,Sum$	$p \mid 3\,Sum$
32	465	0.032258	0.477419	0.305376	0.058065	0.126882	0.000000
64	1953	0.015873	0.494624	0.343574	0.079877	0.066052	0.000000
128	8001	0.007874	0.504937	0.353206	0.087989	0.045994	0.000000
256	32385	0.003922	0.513015	0.358715	0.096835	0.027513	0.000000
512	130305	0.001957	0.518292	0.363440	0.100073	0.016239	0.000000
1024	522753	0.000978	0.520510	0.367018	0.102769	0.008725	0.000000
2048	2094081	0.000489	0.521866	0.368390	0.104098	0.005158	0.000000
127	7875	0.008000	0.500698	0.329397	0.088508	0.045460	0.027937
257	32640	0.003922	0.508946	0.346078	0.094485	0.025092	0.021477
509	128778	0.001972	0.514405	0.356039	0.099512	0.013690	0.014381
1021	519690	0.000981	0.517949	0.361040	0.102134	0.008030	0.009865
1999	1995003	0.000501	0.520501	0.364690	0.103536	0.004863	0.005910
3001	4498500	0.000333	0.520731	0.365859	0.104105	0.003346	0.005626

Für $s_3 \to \infty$ scheint das Verhältnis der Gebietsflächen mit $p = s_1 + s_3$, $p = s_2 + s_3$, $p = s_1 + s_2$ gegen $\approx 0.525 - \varepsilon : 0.37 + \varepsilon : 0.105$ zu streben[5] und Gebiete mit anderen Werten bilden eine Menge vom Maß 0. Ich habe nicht mehr versucht, die Hausdorff-Dimension der Mengen der beiden letzten Fälle genauer zu bestimmen, die deutlich > 1 ist.

Ich berechnete auch die Vorperioden aller 3-Zug-Subtraktions-Spiele für alle $s_3 < 550$. Auf Grund dieser Daten bin ich sicher, daß die extremen Werte der Vorperioden mit wachsendem größten Zug für $s_3 > 100$ von den ersten sechs dieser acht Zug-Familien gestellt werden:

[5] $\pm\varepsilon$ soll anzeigen ob dieser Wert eher zu erhöhen oder zu erniedrigen ist. Die Ungenauigkeit der beiden ersten Werte beträgt etwa $1/200$ und die des kleinsten circa $1/500$.

$L_n = \{s_1, s_2, s_3\}$	Periode	Vorperiodenlänge	Typ
$\{5n-2, 10n-3, 10n+2\}$	5	$45n^2 - 10n$	$s_3 - s_2 = 5$
$\{5n-2, 5n+3, 10n+2\}$	5	$45n^2 - 1$	$s_2 - s_1 = 5$, $s_1 + s_2 = s_3 - 1$
$\{4n-1, 10n-2, 14n-2\}$	$6n-1$	$88n^2 - 88n + 13$	$s_1 + s_2 = s_3 - 1$
$\{4n+1, 10n+2, 14n+2\}$	$6n+1$	$88n^2 - 14n - 4$	$s_1 + s_2 = s_3 + 1$
$\{4n-1, 10n-3, 14n-5\}$	2	$88n^2 - 102n + 25$	$s_1 + s_2 = s_3 + 1$
$\{4n+1, 10n+3, 14n+5\}$	2	$88n^2 - 9$	$s_1 + s_2 = s_3 - 1$
$\{6n+2, 15n+5, 21n+6\}$	$27n+8$	$174n^2 + 92n + 13$	$s_1 + s_2 = s_3 + 1$
$\{6n+2, 15n+4, 21n+5\}$	$36n+9$	$174n^2 + 75n + 9$	$s_1 + s_2 = s_3 + 1$

Es fällt auf, daß sie fast alle der Gleichung $s_1 + s_2 = s_3 \pm 1$ genügen und der Quotient $88/(14 \cdot 10) < q(L_n)/(s_2 s_3) < 45/(5 \cdot 10) = 0,9$ bleibt. Somit dominiert letztlich die Vorperiodenlänge der 3-Zug-Familie $\{5n-2, 5n+3, 10n+2\}$ alle anderen Vorperiodenlängen. Die Periodenlänge der beiden ersten Familien läßt sich als $(s_1 + s_3)/(3n)$ und die der 3. und 4. Familie als $(s_2 + s_3)/4$ auffassen. Für die 5. und 6. Familie tritt dagegen der Fall der Vermutung 6 ein, daß die Periodenlänge alle drei möglichen Zugsummen teilt. Und die beiden letzten Familien illustrieren den Fall, daß sich die Periodenlänge genau als Summe zweier Züge darstellen läßt.

Ein ausgewähltes 4-Zug-Subtraktions-Spiel

Für $|L| = 4$ habe ich die Suche nach Subtraktions-Spielen mit langen Perioden begonnen und einparametrige Familien $(L_n)_{n\in\mathbb{N}}$ der Art betrachtet, daß $L_n = (a_1n + b_1, a_2n + b_2, \ldots, a_4n + b_4)$ ist, um Beispiele mit langer Periode und/oder Vorperiode zu bekommen.

Diese 4-Zug-Subtraktions-Spiele wurden schon von anderen Mathematikern untersucht. Althöfer und Bültermann [AB95] berichteten von einer Analyse der Subtraktions-Spiel-Familie $L_n = (n, 4n, 12n + 1, 16n + 1)$, die für $n \leq 26$ die Periodenlängen $56n^3 + 52n^2 + 9n + 1$ besitzt. Ihre vollständige Analyse legt die dort beschriebene Struktur auch für alle weiteren n nahe. Da sie für ihre Familie keinen Beweis gaben, daß diese kubisches Wachstum in ihren Periodenlängen für alle $n \in \mathbb{N}$ hat, suchte ich mir eine andere Subtraktions-Spiel-Familie von der ich auch kubisches Wachstum vermutete, die aber eine leichter zu beschreibende und damit zu beweisende Struktur haben sollte.[6]

Dazu schaute ich als eine Verfeinerung der $\mathcal{P}$-$\mathcal{N}$-Folge die Folge der Sprague-Grundy-Werte (SGW), die sogenannte Nim-Folge [BCG82, p. 82, V 1], dieser Spiele an. Hierbei werden die Verlustpositionen mit 0 bezeichnet und die Gewinnpositionen nehmen als Werte natürliche Zahlen an.

Definition 7

$$SGW(n) = \min \ \mathbb{N}_0 \setminus \bigcup_{\substack{s\in L:\\ s\leq n}} SGW(n-s) \qquad \textit{für alle } \ n \in \mathbb{N}_0$$

Diese Definition hat ihren Ursprung in der Berechnung der Gewinn- und Verlustwerte von mehreren parallel gespielten neutralen Spielen, also der Summe von Spielen [Gru39] [Spr35]. Ferner bleiben die SGW durch $|L|$ beschränkt.

Als Illustration diene das schon in der Einführung benutzte Subtraktions-Spiel $L = \{3, 7, 8\}$:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$v(n)$	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1
$SGW(n)$	0	0	0	1	1	1	0	2	2	1	3	0	0	2	1
$SGW(n-s_i)$	–	–	–	0	0	0	1	1,0	1,0,0	0,0,0	2,1,0	2,1,1	1,1,1	3,0,1	0,2,0

[6] Mittlerweile habe ich auch die Struktur der Familie von Althöfer und Bültermann für $n > 26$ bewiesen. Siehe dazu weiteres im Kapitel „Konsistenzbeweis für gewisse ...“.

n	15	16	17	18	19	20	21	22	23	24
$v(n)$	1	0	0	1	1	1	0	0	1	1
$SGW(n)$	1	0	0	2	1	1	0	0	2	1
$SGW(n-s_i)$	0,2,2	2,1,2	1,3,1	1,0,3	0,0,0	0,2,0	2,1,2	1,1,1	1,0,1	0,0,0

Als meine Zugmenge eines Kandidaten für kubisch wachsende Periodenlängen wählte ich $L_n = (2n, 5n, 6n+1, 8n+1)$. Diese hat, meiner Meinung nach, eine der einfachsten Strukturen von Subtraktions-Spielen mit kubisch wachsender Periodenlänge. Faßt man die SGW als Buchstaben eines Alphabetes auf, kann man von endlichen, sich nicht überlappenden Abschnitten dieser Folge als Worten sprechen. Im einfachsten Fall bestehen die Worte aus maximal langen Abschnitten gleicher Buchstaben. Die Folge der SGW meiner Zugmenge ist in Worte mit gleichen Nimwerten aufgebrochen worden; jedes dargestellt in der Form SGW*Anzahl*.

Hierbei ist der besseren Lesbarkeit wegen der SGW 0 durch einen Unterstrich `_` dargestellt und die anderen Sprague-Grundy-Werte $1, 2, 3, 4$ durch die Großbuchstaben `A`, `B`, `C`, `D`. Des weiteren unterteile ich den Strom dieser Worte jedesmal in Zeilen, wenn ein `_`-Wort anfängt.

Anzahl ist die Länge eines solchen Wortes, gegeben als ein linearer Term in i, j und n.
Die Werte der Variablen i, j werden durch Schleifen der Form `FOR` *variable* `FROM` ... `UPTO` ... `REPEAT` ... `ENDREP` gesteuert. Die äußerste Schleife `REPEAT` ... `ENDREP` umschließt die gesamte Periode. An dieser Stelle möchte ich auch den Begriff des „Blocks“ einführen: Darunter verstehe ich eine Folge von SGW, die durch Schlüsselworte wie `FOR`, `REPEAT` oder `ENDREP` eingeschlossen sind.[7] Damit bekommen wir diese[8] aus vier aufeinanderfolgenden Blöcken bestehende Beschreibung:

[7] Diese bildet nicht notwendigerweise einen Schleifenrumpf, da die einschließenden Schlüsselworte nicht logisch zusammen gehörig sein müssen.

[8] Der erste Block geht vom allerersten `REPEAT` bis zum ersten `FOR`. Der zweite bildet den Rumpf der ersten `j`-Schleife, der dritte den Schleifenrumpf der zweiten `j`-Schleife und der vierte Block schließt daran an bis zum vorletzten `ENDREP`, dem der `i`-Schleife.

Satz 8
Die Nim-Folge von $L_n = (2n, 5n, 6n+1, 8n+1)$ kann für alle $n \in \mathbb{N}$ wie folgt geschrieben werden:

```
REPEAT
   _(2n)    A(2n)
   _n       Bn        An   @1        C(n-1)   B(2n)   C(n+1)
   _n       B(n-1)    D1   An
   _(2n)    A(2n)     B1
   _(n-1)   Bn        C1   A(n-1)    Cn       Bn      D1     B(n-1) C1
   FOR i FROM 1 UPTO n-1 REPEAT
      FOR j FROM i UPTO n-1 REPEAT
         _(j)       C(n-1-j)  B(j-i+1)
         _(n-j+i)   A(j)      B(n-1-j)    C1
         _(j-i)     A(n-j+i)
         _n         A(j-i)    Bi
         _(n-j)     An
         _(j-i+1)   C(i-1)    A(n-j)      B(j+1)
         _(n-1-j)   A(j-i+1)  B(n-1-j+i)  C(j+1)    A(n-1-j)  B(j-i+1)
         _i         C(n-1-j)  Bn          C(j-i+1)  Ai        B(n-1-j)  C1
      ENDREP
      FOR j FROM 0 UPTO i-1 REPEAT
         _n         Aj        B(n-i)
         _(i-j)     An
         _(n-i+j)   A(i-j)    Bj
         _(n-j)     A(n-i+j)  B(i-j)    Cj      A(n-j)  B(j+1)
         _(n-i)     C(i-1-j)  Bn        C(j+1)  A(n-i)  B(i-1-j)  C1
         _j         C(n-1-j)  B(n+1-i+j)
         _(i-j)     Aj        B(n-1-j)  C1
         _(j+1)     C(n-1-i)  A(i-j)
      ENDREP
      _n        Ai        B(n-1-i)  C1        An
      _n        Bi
      _(n-i)    An        Ci        A(n-i)    B(i+1)
      _(n-1-i)  Bn        C(i+1)    A(n-1-i)  D1
      _i        C(n-1-i)  Bn        C1        Ai      B(n-1-i)  C1
   ENDREP
ENDREP
```

Der Klammeraffe `@` im sechsten Wort steht für `C` nur beim ersten Schleifendurchlauf und danach, falls $n \geq 2$ ist, immer für den Wert `D`. Für $n = 1$ nimmt er dagegen stets den Wert `C` an, da bei $n = 1$ keine `FOR`-Schleife durchlaufen wird und somit, im Gegensatz zu $n > 1$, der erste Block immer auf sich selbst folgt.[9] An dieser Eigenart kann man auch leicht die minimale Vorperiode von $7n+1$ dieser Nim-Folge ausmachen. Wenn man _-Worte der Länge $2n$ sucht — diese treten nur als erstes und vierzehntes Wort in der äußersten Schleife auf — erkennt man, daß sowohl die Folge der SGW, als auch die $\mathcal{P}$-$\mathcal{N}$-Folge keine echte Unterperiode außer denjenigen, die diese Darstellung nahelegt, haben kann.

Folgerung 9

Dieses Spiel hat für alle $n \in \mathbb{N} \quad q(L_n) = 0$ *und*
$p(L_n) = (n-1)(n(14n+1) + (13n+1)) + (24n+2) = 14n^3 + 11n + 1$.

Für $n = 1$ erhalten wir daher:

```
REPEAT
   __  AA
   _  B  A  C  BB  CC
   _  D  A
   __  AA  B
   B  C  C  B  D  C
ENDREP
```

oder für $n = 2$ die folgende Struktur:

```
REPEAT
   ____  AAAA
   __  BB  AA  @  C  BBBB  CCC
   __  B  D  AA
   ____  AAAA  B
   _  BB  C  A  CC  BB  D  B  C
   _  B
   __  A  C
   AA
   __  B
   _  AA
   _  A  BB
   A  B  CC  B
   _  BB  C  A  C
   __  B
```

[9]Dies ist nur ein typographischer Kniff um eine Umstrukturierung der Beschreibung und Einführung eines neuen Blockes vor der äußersten Schleife zu vermeiden. Syntaktisch wäre hier ein `IF` ... `THEN` ... `ELSE` ... `FI` nötig.

```
    _  AA
    _  A
    __  A  B  AA  B
    _  BB  C  A  C
    C  BB
    _  B  C
    _  A
    __  A  C  AA
    __  B
    _  AA  C  A  BB
    BB  CC  D
    _  BB  C  A  C
ENDREP
```

Zur besseren Gewöhnung hier auch noch die Darstellung der Nim-Folge für $n = 3$ — die Leerzeilen trennen die Blöcke voneinander:

```
REPEAT
_6 A6
_3 B3 A3 @1 C2 B6 C4
_3 B2 D1 A3
_6 A6 B1
_2 B3 C1 A2 C3 B3 D1 B2 C1

_1 C1 B1
_3 A1 B1 C1 A3
_3 B1
_2 A3
_1 A2 B2
_1 A1 B2 C2 A1 B1
_1 C1 B3 C1 A1 B1 C1

_2 B2
_2 A2 C1
_1 A2
_3 A1 B1
_1 A3
_2 A1 B3
A2 B1 C3 B2
_1 B3 C2 A1 C1

_3 B2
_1 A3
```

```
_2 A1
_3 A2 B1 A3 B1
_2 B3 C1 A2 C1
C2 B3
_1 B2 C1
_1 C1 A1

_3 A1 B1 C1 A3
_3 B1
_2 A3 C1 A2 B2
_1 B3 C2 A1 D1
_1 C1 B3 C1 A1 B1 C1

_2 B1
_3 A2 C1
A3
_3 B2
_1 A3
_1 C1 A1 B3
A1 B2 C3 B1
_2 B3 C1 A2 C1

_3 B1
_2 A3
_1 A2
_3 A1 B2 A3 B1
_1 C1 B3 C1 A1 B1 C1
C2 B2
_2 B2 C1
_1 A2

_3 A1 B1
_1 A3
_2 A1 B1
_2 A2 B1 C1 A2 B2
_1 B3 C2 A1 C1
_1 C1 B3
_1 A1 B1 C1
_2 A1

_3 A2 C1 A3
_3 B2
_1 A3 C2 A1 B3
```

```
B3 C3 D1
_2 B3 C1 A2 C1
ENDREP
```

Die Nim-Folge ist als eine endliche Beschreibung von Aneinanderhängungen und Wiederholungen von SGW in Form von Worten bekannter Länge gegeben. Wichtige Voraussetzungen für den Beweis des Satzes:

- Alle Wortlängen sind *linear* in den Variablen i, j und n.
- Die obige Beschreibung ist zerlegbar in endlich viele Blöcke von SGW-Worten — bei der untersuchten Familie in vier Blöcke:
  - In den der äußeren Schleife vorangehenden Block der Länge $24n+2$ mit max. 24 Worten,
  - den der ersten inneren Schleife von $14n+1$ SGW mit max. 31 Worten,
  - den der zweiten inneren Schleife von $14n+1$ SGW mit max. 31 Worten,
  - und den nachfolgenden Block im Schleifenrumpf der i-Schleife der Länge $13n+1$ aus max. 24 Worten.
- All diese vier Längen sind linear in n. Günstigerweise sind ihre Einzellängen noch größer als $\max L$.

Den **Beweis** kann man nun führen, wie nachfolgend schematisch dargelegt wird: Im Prinzip schaut man für jedes SGW-Wort in der Beschreibung in Satz 8 nach, in welche Positionen die Züge aus L das Wort bringen würden. Der gerade überprüfte SGW muß der kleinste Wert sein, der nicht von den 4 Zügen aus L getroffen wird. In unserem Fall läuft dies auf die Untersuchung von weniger als $4 \cdot 8 \cdot \max\{24, 31, 31, 24\}$ einzelnen Tests hinaus, da jede der zwei inneren Schleifen aus 31 SGW-Worten besteht und der vorausgehende und nachfolgende Block aus je 24 SGW-Worten besteht. Der Faktor „8" zählt die Anzahl der verschiedenen Übergänge von aufeinanderfolgenden Blöcken. Ein einzelner Test bedeutet das Untersuchen einer Zugmöglichkeit eines Wortes der Beschreibung, die im allgemeinen auf mehrere Worte der Beschreibung zurückführt. Die Buchführung wird durch die Tatsache vereinfacht, daß die Positionen der Grenzen der SGW-Worte häufig Vielfache von n sind und diese mehr oder weniger gleichmäßig über die Beschreibung verteilt auftreten. Ferner sind die auftretenden Blocklängen alle unabhängig von i und j.

Um für beliebige Darstellungen dieser Form zu zeigen, daß eine solche keine echten Unterperioden der einzigen stets vorhandenen `REPEAT` ... `ENDREP` Schleife hat, muß man geeignete Positionen mit bestimmten Worten auswählen und überprüfen. Für meine obige Nim-Folge war das Wort `_(2n)` angemessen, da es nur im ersten Block auftritt. Die genaue Bestimmung der minimalen Vorperiode an Hand einer solchen Beschreibung ist im allgemeinen dagegen aufwendiger. Man könnte ein Computerprogramm schreiben, welches solche Beschreibungen überprüft und damit einen Beweis ihrer Korrektheit gibt oder Ausnahmen, das heißt Widersprüche, für bestimmte Werte von n entdeckt. Ein Beispiel eines solchen Programmes, welches kombinatorische Strukturen mit Hilfe von Systemen von linearen Ausdrücken in freien Variablen symbolisch beweist, war auch der Kern des „Additionsketten"-Teils meiner Diplomarbeit [Fla91].

Um nicht einen viele Seiten Papier verschlingenden Beweis nach diesem Schema führen zu müssen — dies wäre auch noch für weitere Subtraktions-Spiele im Rahmen dieser Arbeit nötig gewesen — habe ich mich entschlossen, dies mittels eines in einer höheren Programmiersprache formulierten Algorithmus', sprich Computerprogrammes, zu tun. Mittlerweile konnte damit nicht nur ein vollständiger Beweis der Korrektheit meiner Beschreibung des ausgewählten Spieles erbracht werden, sondern auch die ältere Strukturanalyse des Spieles von Althöfer und Bültermann [AB95] für alle $n \geq 2$ verifiziert werden. Auf diesen Algorithmus, seine Leistungsfähigkeit und technische Details möchte ich aber erst in dem späteren Kapitel „Konsistenzbeweis für gewisse Subtraktions-Spiele" ausführlicher eingehen.

Der allgemeine Fall $|L| = 4$

Sillke teilte mir mit, daß Koschnick [Kos93] etliche Familien entdeckte, deren Perioden wie Polynome vom Grad 4 anzuwachsen scheinen, und für eine von ihm entdeckte Familie, $(n, 4n, 32n+1, 36n+1)$, Eintrag Nr. 15 in der folgenden Liste der Subtraktions-Spiele, scheint die Periodenlänge sogar wie ein Polynom vom Grad 5 zu wachsen. Sillke [Sil93] hat auch die folgenden zwei 2-parametrigen Familien mit $L_{n,d} = (n, 4n, (10d+17)n+1, (10d+21)n+1)$ und $L_{n,d} = (n, 4n, (20d+12)n+1, (20d+16)n+1)$ konstruiert, deren Periodenlängen ebenfalls wie Polynome 5. Grades in n zu wachsen scheinen.

Um an die Ergebnisse von Koschnick und Sillke anzuknüpfen, folgt eine Liste etlicher Subtraktions-Spiel-Familien. Diese haben im von mir untersuchten Bereich Periodenlängen, die durch Polynome vom Grad 5 oder 6 wiedergegeben werden, und zwei Familien besitzen Vorperiodenlängen, die durch Polynome vom Grad 4 im selbigen Bereich beschrieben werden können. Dazu betrachte ich die folgenden fünf Teilmengen von 4-Zug-Subtraktions-Spielen:

Klasse	Bedingung
I	$s_1 + s_3 = s_4$
II	$s_2 + s_3 = s_4$
III	$s_1 + s_2 = s_4$
IV	$s_1 + s_2 = s_3$
O	keine der vorherigen

Diese Unterteilung wurde durch die Beobachtung motiviert, daß so eine additive Beziehung zwischen verschiedenen Zügen von L dazu neigt, die Periode wesentlich zu vergrößern. Anmerkungen: Für Zug-Familien $(L_n)_{n\in\mathbb{N}}$ bei 4-Zug-Spielen, die mindestens zwei unterschiedlichen Klassen $\neq$ O angehören, sind nur die beiden folgenden Kombinationen von Klassen möglich: Entweder $L = (s_1, s_2, s_1+s_2, 2s_1+s_2)$ in Klasse I und IV oder $L = (s_1, s_2, s_1+s_2, s_1+2s_2)$ in Klasse II und IV. Diese scheinen ähnlich wie die 3-Zugspiele mit $s_1 + s_2 = s_3$ höchstens quadratisches Wachstum in ihren Periodenlängen zu haben. Empirisch zeigen Familien aus den Klassen I und II stärkeres Wachstum als die in Klasse III und IV oder gar Klasse O. Dies kann man gut an den durchschnittlichen Periodenlängen in diesen fünf Klassen für festes s_4 beobachten.

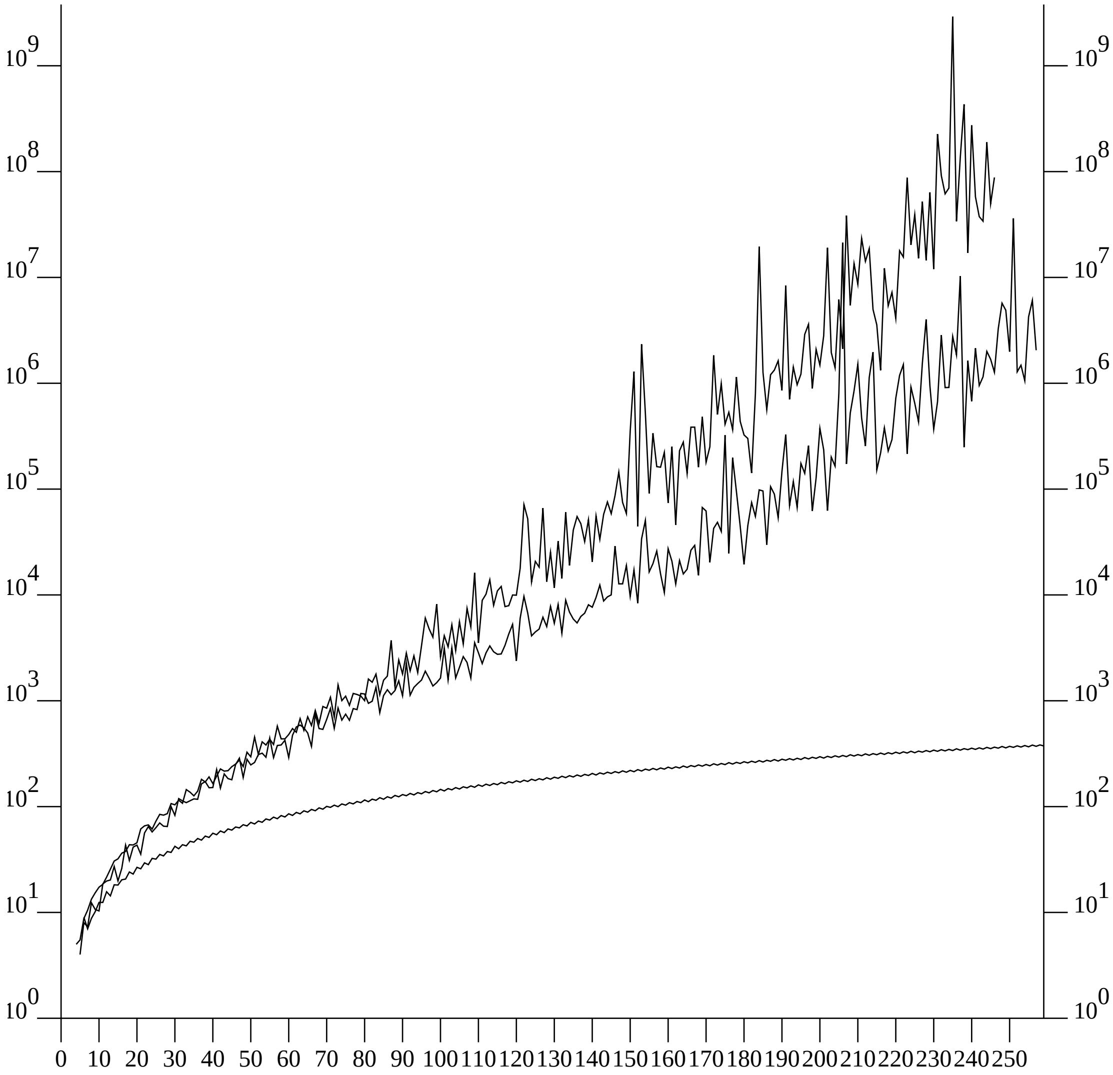


Bild vi Durchschnittliche Periodenlängen für s_4 bis etwa 256 in den Klassen I,II,O (Kurven sind diesen von oben nach unten zugeordnet)

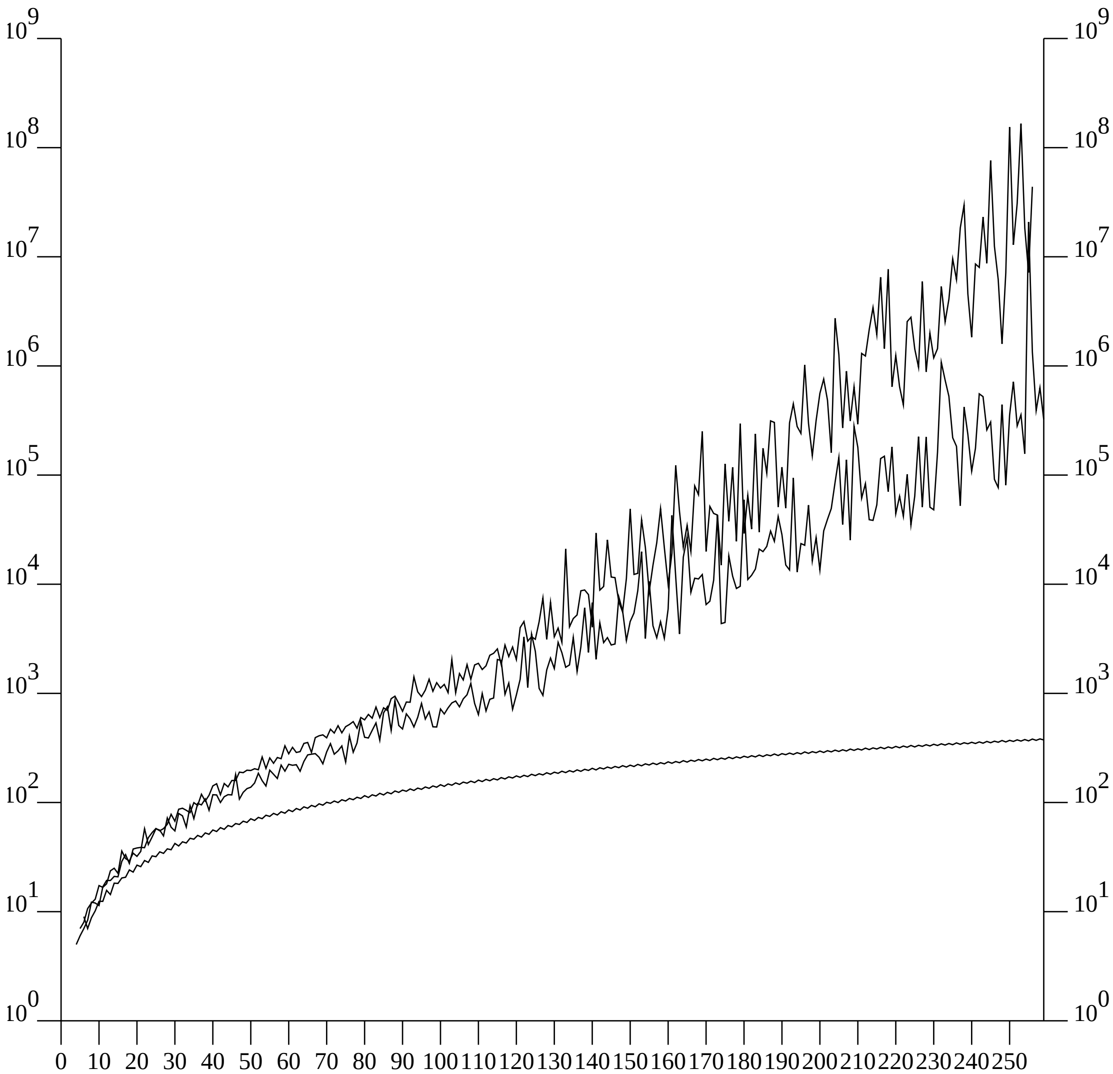


Bild vii Durchschnittliche Periodenlängen für s_4 bis etwa 256 in den Klassen IV,III,O (Kurven sind diesen von oben nach unten zugeordnet)

Fast jeder Eintrag in der nachfolgenden Liste hat die folgende Form:

Nr. Klasse Liste der zulässigen Züge zulässiger Bereich für n, Anzahl der Zeugen
`p` = Periodenlänge als Polynom in n
`q` = Vorperiodenlänge als Polynom in n

Nr. ist eine laufende Numerierung, die nur zur Bezugnahme dient. *Klasse* bezeichnet die Klasse dieses Subtraktions-Spiels gemäß obiger Einteilung. *Liste der zulässigen Züge* definiert die Zugmenge L. *zulässiger Bereich für n* gibt den niedrigsten zulässigen Index und den größten geprüften Index n dieser Zug-Familie an. Im zulässigen Bereich von n wird die Periodenlänge durch ein Polynom *p(n)* und die Vorperiodenlänge durch ein Polynom *q(n)* beschrieben. Die *Anzahl der Zeugen* ist die Zahl

$$(\text{größtes geprüftes } n - \text{kleinstes zulässiges } n \ +1) - (\text{Grad des Polynoms } p \ +1)\,,$$

die als der Grad der Überbestimmtheit des Polynoms[10] interpretiert werden kann. Bei den Einträgen, beispielsweise Eintrag 19, deren (Vor-)Periodenlängen von $n \bmod C$ mit festem $C \in \mathbb{N}$ abhängen,[11] fehlt die *Anzahl der Zeugen*, da man sie für jede Restklasse separat ermitteln müßte. Die formelle Bestimmung dieser Anzahl würde in meinen Fällen sogar zu negativen Zahlen führen, also nicht als Interpretation der Evidenzstärke geeignet sein. Diese Zerlegung in Restklassen spiegelt aber nicht die Verwandtschaft der sich ergebenden endlich vielen Familien wider, aus der ich zusätzliche Information, wie z.B. gleiche Koeffizienten für gewisse Summanden, zur Polynomapproximation ziehe.

Betrachte z.B. Eintrag 1: Dort ist das *kleinste zulässige* $n = 2$, das *größte geprüfte* $n = 14$, der Grad von $p(n)$ ist 5 und somit erhalten wir für die *Anzahl der Zeugen* $(14 - 2 + 1) - (5 + 1) = 7$.

```
 1 I { 1n+0 , 14n-1 , 51n-3 , 52n-3 }  2 <= n <= 14   ,  7
p = 1915n^5 + 2149n^4 - 3035n^3 - 598n^2 + 49n + 0
q = 2454n - 144

 2 I { 2n+0 , 26n-3 , 38n-5 , 40n-5 }  4 <= n <= 10   ,  1
p = 7488n^5 - 14592n^4 + 9348n^3 - 1466n^2 - 386n + 60
q = 936n^2 - 16n - 10

 3 I { 4n+1 , 12n+1 , 44n+3 , 48n+4 }  1 <= n <= 13   ,  7
p = 2208n^5 - 600n^4 - 744n^3 + 398n^2 + 156n + 8
q = 276n^2 + 225n + 16

 4 I { 4n+0 , 40n-3 , 56n-4 , 60n-4 }  2 <= n <= 8   ,  1
p = 178176n^5 - 334080n^4 + 172032n^3 - 49536n^2 + 6928n - 280
```

[10] Bei den Einträgen 56 – 59 ist das Polynom $q(n)$ als Referenz gewählt, da es einen größeren Grad als das Polynom $p(n)$ der Periodenlänge hat.

[11] Der dyadische Operator mod, der von $\mathbb{Z} \times \mathbb{N}$ nach $\mathbb{N}_0$ abbildet, ist hier mit % bezeichnet. Dies ist nicht mit dem Äquivalenzrelationszeichen $\equiv$, kongruent, zu verwechseln, bei der man die gesamte Relation, dann Kongruenz genannt, noch mit $\pmod{n}$ für ein festes $n \in \mathbb{N}$ präzisieren muß. Inhaltlich ordnet der %-Operator einem Paar (z, n) einen Repräsentanten der getroffenen Äquivalenzklasse zu, der kleiner als n ist, wogegen die $\equiv$-Relation zwei Zahlen x, y als kongruent betrachet, wenn sie in derselben Äquivalenzklasse liegen, wobei die Zerlegung durch $\pmod{n}$ spezifiziert wird und kein Repräsentant ausgezeichnet ist. Letzteres könnte man auch stets als $n|\ x-y$ schreiben.

```
q = 464n^2 - 132n + 8

 5 I {  4n-1 , 25n-6 , 35n-9 , 39n-10 }  3 <= n <= 19   ,  11
p = 410n^5 - 2103.5n^4 + 4002n^3 - 3641.5n^2 + 1592n - 227
q = 577n - 144

 6 I { 6n+0 , 16n+0 , 42n-1 , 48n-1 }  2 <= n <= 14   ,  7
p = 1776n^5 + 1208n^4 - 248n^3 - 784n^2 + 96n - 3
q = 174n^2 + 200n + 1

 7 I { 8n+4 , 26n+12 , 36n+17 , 44n+21 }  2 <= n <= 10   ,  3
p = 32832n^5 + 39168n^4 - 31896n^3 - 13328n^2 + 5496n + 995
q = 544n + 260

 8 I { 8n-4 , 38n-18 , 44n-21 , 52n-25 }  2 <= n <= 10   ,  3
p = 11776n^5 - 18368n^4 - 4528n^3 + 16356n^2 - 7456n + 985
q = 756n - 362

 9 I { 9n-1 , 10n-1 , 24n-3 , 33n-4 }  2 <= n <= 18   ,  11
p = 456n^5 - 206n^4 - 103n^3 + 432n^2 - 217n + 20
q = 376n - 43

10 I {  9n+2 , 28n+6 , 42n+9 , 51n+11 }  2 <= n <= 11   ,  4
p = 5265n^5 + 2280.5n^4 - 1634n^3 - 1204.5n^2 - 265n -20
q = 735n + 160

11 I { 12n+4 , 31n+10 , 40n+13 , 52n+17 }  4 <= n <= 16   ,  7
p = 896n^5 - 1386n^4 + 1491n^3 - 502n^2 + 45n + 140
q = 772n + 246

12 I { 12n-3 , 32n-8 , 45n-11 , 57n-14 }  1 <= n <= 14   ,  7
p = 1428n^5 - 476n^4 - 541n^3 + 1274n^2 - 673n + 97
q = 1154n - 282

13 I { 15n+3 , 16n+3 , 39n+8 , 54n+11 }  2 <= n <= 9   ,  2
p = 2604n^5 + 1774n^4 + 418n^3 - 805n^2 - 291n - 24
q = 963n + 196

14 I { 15n-4 , 22n-6 , 40n-11 , 55n-15 }  1 <= n <= 10   ,  4
p = 9940n^5 - 8144n^4 - 2693n^3 + 4030n^2 - 1203n + 113
q = 103n - 27

15 II { 1n+0 ,  4n+0 ,  32n+1 ,  36n+1  }  2 <= n <= 16   ,  9
```

```
p = 1632n^5 + 2384n^4 + 916n^3 - 20n^2 - 25n - 1
q = 0

16 II {  1n+0 ,  4n+0 , 57n+1 , 61n+1 }  2 <= n <= 11   ,  4
p = 10384n^5 + 4944n^4 - 4624n^3 - 260n^2 + 45n + 1
q = 0

17 II { 2n+1 , 8n+3 , 26n+9 , 34n+12 }  2 <= n <= 12   ,  5
p = 3776n^5 + 17184n^4 + 30056n^3 - 24680n^2 + 9398n - 1306
q = 1416n^2 /9 + (188+80(n%3))n/3 + { 5 , -493/3 , -277/3 }

18 II { 2n+1 , 8n+2 , 26n+5 , 34n+7 }  4 <= n <= 12   ,  3
p = 3776n^5 - 6048n^4 - 1280n^3 + 4816n^2 + 996n + 14

19 II { 2n+1 , 8n+1 , 26n+1 , 34n+2 }  3 <= n <= 23   ,
p = (3776n^5 - 13792n^4 + 8360n^3 + 12064n^2 - 2066n - 80)       n%3 = 0
p = (3776n^5 - 13792n^4 + 9128n^3 + 21496n^2 - 5738n - 290)/81   n%3 = 1
p = (3776n^5 - 13792n^4 + 8360n^3 + 12064n^2 - 2066n - 80)       n%3 = 2

20 II { 2n+0 , 8n+0 , 54n+1 , 62n+1 }  1 <= n <= 8   ,  2
p = 18560n^5 + 18144n^4 - 5552n^3 - 92n^2 + 36n + 1
q = 0

21 II { 2n+0 , 24n-1 , 40n-1 , 64n-2 }  2 <= n <= 10   ,  3
p = 9504n^5 - 7140n^4 - 1276n^3 + 1351n^2 + 48n - 4
q = 312n - 7

22 II { 3n-1 , 10n-3 , 45n-13 , 55n-16 }  2 <= n <= 14   ,  7
p = 2520n^5 - 7479n^4 + 7736n^3 - 3959n^2 + 938n - 80
q = 738n - 213

23 II { 4n-1 , 15n-4 , 48n-13 , 63n-17 }  1 <= n <= 9   ,  3
p = 16320n^5 - 9944n^4 - 5437n^3 + 5453n^2 - 1404n + 118
q = 452n - 121

24 II { 5n+0 , 35n-2 , 40n-3 , 75n-5 }  1 <= n <= 11   ,  5
p = 5760n^5 - 3282n^4 - 1665n^3 + 947n^2 + 181n - 15
q = 320n^2 - 49n + 5

25 II { 6n+1 , 16n+2 , 20n+3 , 36n+5 }  2 <= n <= 18   ,  11
p = 832n^5 - 1112n^4 - 186n^3 - 114n^2 - 98n - 12
q = 194n + 29
```

```
26 II { 6n+0 , 20n+1 , 28n+1 , 48n+2 }  2 <= n <= 15   ,  8
p = 1360n^5 - 2092n^4 + 150n^3 - 254n^2 - 120n - 4
q = 256n + 11

27 II { 8n-2 , 13n-3 , 40n-9 , 53n-12 }  1 <= n <= 16   ,  10
p = 846n^5 + 132n^4 - 784.5n^3 + 853n^2 - 449.5n + 21
q = 239n - 53

28 II { 9n+3 , 14n+5 , 21n+8 , 35n+13 }  1 <= n <= 19   ,  13
p = 294n^5 + 1879n^4 + 687n^3 - 799n^2 - 147n + 58
q = 161n + 61

29 II { 9n+4 , 14n+7 , 21n+11 , 35n+18 }   2 <= n <= 22   ,  16
p = 147n^5 + 1597.5n^4 - 367n^3 - 3195.5n^2 - 832n + 262
q = 161n + 74

30 II { 9n-2 , 14n-3 , 21n-4 , 35n-7 }  1 <= n <= 17   ,  11
p = 588n^5 + 972n^4 - 1963n^3 + 703n^2 + 67n - 27
q = 161n - 31

31 II { 12n-4 , 18n-6 , 30n-11 , 48n-17 }  1 <= n <= 8   ,  2
p = 16200n^5 - 35424n^4 + 29407.5n^3 - 11550n^2 - 2113.5n - 139
q = 306n^2 - 189n + 30

32 II { 12n-3 , 28n-7 , 35n-9 , 63n-16 }  2 <= n <= 9   ,  2
p = 2184n^5 - 580n^4 - 1748n^3 + 1312n^2 - 279n + 15
q = 672n - 170

33 II { 16n-6 , 21n-8 , 35n-13 , 56n-21 }  3 <= n <= 11   ,  3
p = 1386n^5 + 5190n^4 - 20880n^3 + 20655n^2 - 7515n + 903
q = 483n - 180

34 II { 9n-m , 2(7n-m)-1 , 3(7n-m)-1 , 5(7n-m)-2 }   m >= -2  n >= 3m+7
p = 294n^5 + (2929+742m)n^4 - (11869 + 9660m + 1876m^2)n^3 +
    (10152 + 14781m + 6874m^2 + 1036m^3)n^2 -
    (1747 + 4898m + 4373m^2 + 1604m^3 + 210m^4)n +
    (75 + 343m + 538m^2 + 369m^3 + 117m^4 + 14m^5)
q = 23(7n-m) - 8

35 II { 9n-m , 2(7n-m)+1 , 3(7n-m)+2 , 5(7n-m)+3 }   m >= 3  n >= 3m-6
p = 588n^5 + (1484m - 2290)n^4 - (3752m^2 - 11480m + 9666)n^3 +
    (2072m^3 - 8596m^2 + 12258m - 5940)n^2 -
    (420m^4 - 2280m^3 + 5058m^2 - 5512m + 2265)n +
```

```
    (28m^5 - 186m^4 + 538m^3 - 824m^2 + 603m - 161)
q = 23(7n-m) + 5

36 III { 9n-4 , 24n-11 , 26n-12 , 33n-15 }  2 <= n <= 31   ,  24
p = 243n^5 - 491n^4 - 27.5n^3 + 281.5n^2 + 19n - 48
q = 236n - 106

37 III { 10n+1 , 40n+5 , 44n+5 , 50n+6 }  2 <= n <= 9   ,  2
p = 3840n^5 + 752n^4 - 12368n^3 + 4752n^2 + 2592n + 223
q = 320n^2 + 414n + 44

38 III { 10n-4 , 40n-15 , 44n-17 , 50n-19 }  2 <= n <= 14   ,  7
p = 1920n^5 - 4424n^4 - 2136n^3 + 9816n^2 - 5306n + 794
q = 320n^2 + 94n - 83

39 III { 11n+1 , 25n+2 , 34n+3 , 36n+3 }  2 <= n <= 21   ,  14
p = 301.5n^5 - 309.5n^4 + 576.5n^3 - 261.5n^2 - 169n - 12
q = 745n + 64

40 III { 14n-5 , 35n-13 , 39n-14 , 49n-18 }  2 <= n <= 13   ,  6
p = 3465n^5 - 4414.5n^4  + 1764n^3 - 694.5n^2 + 235n - 23
q = 550n - 200

41 III { 16n+8 , 40n+19 , 45n+22 , 56n+27 }   2 <= n <= 15   ,
p = 76032n^5 - 123576n^4 + 66880n^3 - 15878n^2 + 1586n - 44    n%2 = 1
p = 76032n^5 + 66504n^4 + 9808n^3 - 4474n^2 - 1852n - 214    n%2 = 0
q = 381n + 185

42 III { 17n-8 , 38n-18 , 44n-21 , 55n-26 }  3 <= n <= 13   ,  5
p = 4416.5n^5 - 12653.5n^4 + 15378n^3 - 9806n^2 + 3229n - 432
q = 644n^2 - 664n + 171

43 III { 18n-4 , 48n-11 , 52n-12 , 66n-15 }  1 <= n <= 7   ,  1
p = 7776n^5 - 7856n^4 - 220n^3 + 1126n^2 + 38n - 48
q = 472n - 106

44 III { 19n-5 , 41n-11 , 45n-12 , 60n-16 }  1 <= n <= 7   ,  1
p = 11972n^5 - 15300n^4 + 5268.5n^3 + 243n^2 - 367.5n + 42
q = 1024n^2 - 392n + 33

45 IV { 10n-5 , 30n-14 , 40n-19 , 54n-26 }  1 <= n <= 8   ,  2
p = 12160n^5 - 32160n^4 + 33936n^3 - 17104n^2 + 4012n - 341
q = 0
```

```
46 IV { 10n-5 , 30n-13 , 40n-18 , 54n-25 }  3 <= n <= 13   ,  5
p = 3040n^5 - 7720n^4 + 7476n^3 - 2794n^2 + 234n + 38
q = 0

47 IV { 12n-2 , 30n-4 , 42n-6 , 50n-7 }  2 <= n <= 11   ,  4
p = 6336n^5 - 3168n^4 - 2760n^3 + 1452n^2 - 166n + 2
q = 354n - 50

48 IV { 12n+4 , 30n+8 , 42n+12 , 48n+13 }  1 <= n <= 14   ,  8
p = 1416n^5 + 2008n^4 - 410n^3 - 378n^2 + 22n + 16
q = 276n^2 + 262n + 53

49 IV { 13n-2 , 27n-4 , 40n-6 , 49n-7 }  3 <= n <= 16   ,  8
p = 936n^5 + 1032n^4 - 4306.5n^3 + 3055.5n^2 - 645n + 42
q = 655n - 95


50 I { 4n+1 , 13n+3 , 28n+6 , 32n+7 }  2 <= n <= 21   ,  13
p = 928n^6 - 1168n^5 - 114n^4 - 382n^3 - 172n^2 - 62n - 10
q = 116n^2 + 103n + 20

51 I { 8n+3 , 19n+7 , 28n+10 , 36n+13 }  2 <= n <= 17   ,  9
p = 1536n^6 - 1048n^5 - 536n^4 - 1754n^3 - 1448n^2 - 193n + 36
q = 509n + 186

52 I { 8n+1 , 28n+3 , 40n+3 , 48n+4 }  2 <= n <= 16   ,  8
p = 2816n^6 - 4240n^5 + 530n^4 - 1045n^3 - 284n^2 - 74n - 5
q = 2816n^4 - 256n^3 + 228n^2 + 301n + 27

53 I { 4n+0 , 22n+0 , 52n-1 , 56n-1 }  2 <= n <= 14   ,  6
p = 10176n^6 - 21440n^5 + 11544n^4 - 8696n^3 + 4998n^2 - 936n + 23
q = 612n - 7

54 II { 18n-2 , 28n-3 , 44n-5 , 72n-8 }  1 <= n <= 10   ,  3
p = 40032n^6 - 23176n^5 - 20808n^4 + 29350n^3 - 11833n^2 + 1942n - 107
q = 332n - 36

55 II { 16n+7 , 26n+12 , 80n+37 , 106n+49 }  1 <= n <= 8   ,  1
p = 108288n^6 + 423792n^5 + 615304n^4 + 438988n^3 + 169074n^2 + 34884n + 3152
q = 478n + 222

56 I { 1n+0 , 7n+1 , 20n+2 , 21n+2 }   1 <= n <= 30   ,  26
```

```
p = 120n^2 - 1n - 1
q = 240n^3 + 118n^2 + 13n + 2

57 I { 3n-1 , 15n-3 , 24n-5 , 27n-6 }   1 <= n <= 18   ,  13
p = 96n^3 + 79n^2 - 78n + 12
q = 96n^4 + 175n^3 + 1n^2 - 48n + 9

58 II { 2n+0 , 12n-3 , 24n-4 , 36n-7 }   5 <= n <= 31   ,
p = 48n - 10
q = 600n^4 - 1385n^3 + 987.5n^2 - 7.5n - 34    n%4 = 0
q = 600n^4 - 1385n^3 + 987.5n^2 - 7.5n - 34    n%4 = 1
q = 600n^4 - 1385n^3 + 987.5n^2 +16.5n - 39    n%4 = 2
q = 600n^4 - 1385n^3 + 987.5n^2 +16.5n - 39    n%4 = 3

59 II { 4n+2 , 14n+5 , 28n+12 , 42n+17 }  3 <= n <= 28   ,
p = 112n^2 + 16n - 12
v = 560n^4 + 1060n^3 + 420n^2 - 13n - 19    n%2 = 0
v = 560n^4 + 1060n^3 + 420n^2 + 15n - 8     n%2 = 1
```

Die Einträge 34 und 35 beschreiben zwei 2-parametrige Familien, deren Mitglieder ich nur berechnete, solange die Periodenlänge kleiner als 10^9 blieb. Bei diesen beiden Familien könnte man auch die Parameterbereichs-Beschreibung

```
n >= 1   -2 <= m <= (n-7)/3   und  m aus Z
n >= 3    3 <= m <= (n+6)/3   und  m aus Z
```

wählen, die vielleicht manchem natürlicher erscheint, obwohl sie länglicher ist. Nach längerem Grübeln kam ich zu dem Schluß, daß die „natürlichste“ Darstellung für beide Zug-Familien erst durch die Umparametrisierung $7n - m \to m$ und z.B. $2n \to n$ erreicht wird. Da beim Umrechnen auf diese neue Parametrisierung leider viele echte Brüche in den Periodenlängen entstehen — allerdings nicht in den Vorperiodenlängen — habe ich von der Durchführung dieser Transformation wieder Abstand genommen.

Falls man in Nr. 34 m auf -3 oder -4 setzt, erhält man Nr. 28 bzw. Nr. 29. In letzterem Fall ist die Periodenlänge für alle n, die durch 4 teilbar sind, immer um einen Faktor 2 kleiner, als nach $p(n)$ zu erwarten ist. Solche Einbrüche werden in einigen Familien beobachtet, wenn die vorausgesagte Periode $p(n)$ im Vergleich zur tatsächlichen Periode für bestimmte n durch einen gewissen Faktor geteilt wird. Hier zur Übersicht alle von mir erkannten Einbrüche in meinen Familien:

Nr.	Faktor	Bedingung
10	2	n%4 = 2 , n ≠ 2
16	5	n%5 = 1
19	81	n%3 = 1
29	2	n%4 = 0
34 , m=2	5	n%5 = 2
35 , m=4	5	n%5 = 0
35 , m=7	11	n%4 = 0 , n ≠ 16
39	2	n%4 = 0
39	4	n%8 = 4 , n ≠ 4
41	5	n%10 = 1
41	11	n=12
53	19	n=11
54	31	n=8

Nr. 55 ist für ungerades n mittels der Substitution $n \to 2n + 1$ aus { `8n-1 , 13n-1 , 40n-3 , 53n-4` } konstruiert worden. Für gerades n ergeben sich mit $n \to 2n$ drei weitere Familien, abhängig von $n \bmod 8$, deren Periodenlängen wie Polynome vom Grad 6 zu wachsen scheinen. Im Detail:

```
II { 16n-1 , 26n-1 , 80n-3 , 106n-4 }  1 <= n <= 13
{  15, 25,  77, 102 }  p=        85564   q=  358   f=2
{  31, 51, 157, 208 }  p=      9484094   q=  730
{  47, 77, 237, 314 }  p=     24966253   q= 1102   f=4
{  63,103, 317, 420 }  p=    534997682   q= 1474
{  79,129, 397, 526 }  p=    988556680   q= 1846   f=2
{  95,155, 477, 632 }  p=    443980606   q= 2218   f=13
{ 111,181, 557, 738 }  p=   3577616531   q= 2590   f=4
{ 127,207, 637, 844 }  p=  31471553882   q= 2962
{ 143,233, 717, 950 }  p=  31569338516   q= 3334   f=2
{ 159,259, 797,1056 }  p= 117797508110   q= 3706
{ 175,285, 877,1162 }  p=  51802580057   q= 4078   f=4
{ 191,311, 957,1268 }  p= 347168548354   q= 4450
{ 207,337,1037,1374 }  p= 279161658976   q= 4822   f=2
 P = 108288n^6 + 98928n^5 - 38096n^4 - 2860n^3 + 5328n^2 - 470n + 10
p = P(n)       n%2 = 0
p = P(n)/2     n%4 = 1
p = P(n)/4     n%4 = 3
q = 372n - 14
```

Die drei Familien unterscheiden sich nur in ihrem „Einbruchfaktor“ von 1, 2, bzw. 4. Ferner tritt ein „sporadischer“ Einbruch bei $n = 6$ um den Faktor 13 auf, der wahrscheinlich den Beginn einer weiteren Unterfamilie charakterisiert. Vielleicht treten auch noch weitere Unterfamilien für größere Modulo-Teiler auf.

Eine Bemerkung zu den Subtraktions-Spiel-Familien mit $L_n = (a_1n+b_1, a_2n+b_2, \ldots, a_4n+b_4)$ deren Periodenlängen wie Polynome zu wachsen scheinen: Da die Häufigkeit von solchen polynomartig in ihren Periodenlängen wachsenden Familien stark mit dem Grad des Polynoms abnimmt, war ich gezwungen, an wenigen, typischerweise drei bis vier Werten abzuschätzen, ob dieses Wachstum durch ein Polynom hohen Grades dargestellt wird. Da dies bei so wenigen „Stützstellen" theoretisch natürlich unmöglich ist, mußte ich mich auf gewisse empirisch entwickelte Heuristiken verlassen. Die hilfsreichste war die sogenannte „Quotientenbedingung". Hierzu betrachte man die Werte $|b_1|/a_1, |b_2|/a_2, |b_3|/a_3, |b_4|/a_4$ falls $a_1, a_2, a_3, a_4 \neq 0$. Falls diese alle im durch $|b_1 \pm 1|/a_1$ definierten Intervall liegen, erhöht dies stark die Wahrscheinlichkeit, daß der Polynomgrad dieser Familie hoch ist. Wie der Leser an der vorangehenden Liste von Subtraktions-Spielen leicht nachprüfen kann, trifft diese „Quotientenbedingung" auch schon für viele der Familien zu, deren Periodenlängen wie Polynome von 5. Grad zu wachsen scheinen.

Für die Beantwortung der Frage (i) in [AB95] nach der Existenz von Familien mit superpolynomialem Wachstum fand ich keine konkreten Beispiele unter den untersuchten 4-Zug-Spielen. Auch konnte ich keine Familien identifizieren, deren Periodenlängen wie Polynome mit einem Grad größer als 6 anwachsen. Die Periodenlängen der Rekordhalter für festes s_4 variieren stark, wenn ihr größter Zug, s_4, anwächst. Als Tabelle seien hier nur die Rekordhalter mit wachsendem s_4 für $80 < s_4 \leq 245$ aufgeführt:

```
{ 4, 54, 77, 81}  p=          190763  q=  287    0.21656    2.7669
{ 5, 23, 77, 82}  p=          393906  q=  818    0.22668    2.9237
{ 4, 33, 83, 87}  p=          776902  q= 1154    0.22491    3.0370
{ 7, 58, 80, 87}  p=        2 369755  q= 1332    0.24341    3.2867
{ 7, 64, 89, 96}  p=        5 756171  q= 1061    0.23392    3.4103
{ 8, 58,101,109}  p=       19 914037  q= 1176    0.22245    3.5825
{11, 51,111,122}  p=      225 217076  q= 1841    0.22743    4.0034
{10, 63,137,147}  p=      370 089094  q= 2524    0.19363    3.9534
{ 8, 29,142,150}  p=      434 777878  q= 1653    0.19130    3.9696
{11,100,139,150}  p=     1299 116811  q= 2791    0.20183    4.1881
{ 6,104,145,151}  p=     6847 546988  q= 5395    0.21638    4.5138
{ 8,102,145,153}  p=    12666 517709  q= 7153    0.21935    4.6243
{11, 51,173,184}  p=   159289 306371  q= 2771    0.20224    4.9462
{22, 71,180,202}  p=   171664 537002  q= 5245    0.18476    4.8733
{11,137,196,207}  p=   370194 612538  q= 4897    0.18565    4.9951
```

```
{ 7, 81,216,223}  p=    771125 532284  q=10718     0.17708    5.0620
{12,154,219,231}  p=  2 247087 757401  q=13109     0.17762    5.2257
{21,103,214,235}  p= 38 733953 041818  q=17114     0.19208    5.7308
```

Die linke Spalte gibt die Zugmenge an, gefolgt von `p=` Periodenlänge, gefolgt von der Vorperiodenlänge und zwei Zahlen, die den Exponenten von 2^{s_4} und s_4 spezifizieren, um die Periodenlänge als Exponentialfunktion in s_4 bzw. Monom in s_4 zu berechnen. Die letzte Zeile bedeutet somit: $p(L = (21, 103, 214, 235)) = 38\,733\,953\,041\,818 = 2^{235^{0.1920\ldots}} = 235^{5.7308\ldots}$ und $q(L) = 17114$.

Ein etwas allgemeinerer Ansatz ist es, zu festvorgegebenem $k \in \mathbb{N}$ die 4-Zug-Subtraktions-Spiele deren Periodenlängen wie Polynome von Grad k wachsen, zu untersuchen. Auch hier sind die Subtraktions-Spiel-Familien von besonderem Interesse, die bei festem Polynomgrad relativ stark wachsen. Für $k = 1$, daß heißt die Periodenlängen wachsen nur linear mit n an, verhalten sich die Koeffizienten solcher Familien ziemlich ungleichmäßig. Für Zug-Familien mit quadratischem Wachstum ihrer Periodenlängen ($k = 2$) entdeckte ich aber dieses extreme Beispiel:

Für alle $n \geq 2$ und für alle $d \geq 2$ sei $L_{n,d} = ((2d+3)n + d, 3(2d+3)n + 5d + 1, 6(2d+3)n + 8d + 3, 7(2d+3)n + 9d + 3)$. Dann gilt

$$
\begin{aligned}
f(d)\, p(L_{n,d}) = \quad & 4(2d+1)(2d+3)(16d^3 - A(d \bmod 3)d^2 - B(d \bmod 6)d + C(d \bmod 6)\,)n^2 \\
& + \tfrac{1}{6}(1600d^5 + D(d, d \bmod 6))n + \tfrac{1}{6}(416d^5 + E(d, d \bmod 6)), \\
\sim \quad & 256d^5n^2
\end{aligned}
$$

wobei $f(d)$ ein d-abhängiger Einbruchfaktor ist,

$$
\begin{aligned}
A(x) &= 4((x-1) \bmod 3) + 8 \\
B(x) &= 11 + (2x) \bmod 3 + 4(x \bmod 2) \\
C(x) &= (4 - x \bmod 2)((3 - x) \bmod 3) + 6 - 3(x \bmod 2)
\end{aligned}
$$

und $D(d, .)$ und $E(d, .)$ zwölf Polynome vom Grad 4 in d sind, mit

$$
\begin{aligned}
D(d,0) &= 738 - 177d - 1016d^2 - 2348d^3 + 128d^4 \\
D(d,1) &= 459 + 285d - 1124d^2 - 2148d^3 + 928d^4 \\
D(d,2) &= 654 + 292d - 604d^2 - 2016d^3 + 528d^4 \\
D(d,3) &= 585 - 429d - 1716d^2 - 2812d^3 + 128d^4
\end{aligned}
$$

$$
\begin{aligned}
D(d,4) &= 570 + 761d - 192d^2 - 1684d^3 + 928d^4 \\
D(d,5) &= 522 - 72d - 1420d^2 - 2480d^3 + 528d^4
\end{aligned}
$$

$$
\begin{aligned}
E(d,0) &= 108 + 228d - 367d^2 - 214d^3 - 304d^4 \\
E(d,1) &= 54 + 147d - 175d^2 - 346d^3 - 96d^4 \\
E(d,2) &= 84 + 206d - 224d^2 - 216d^3 - 200d^4 \\
E(d,3) &= 90 + 177d - 393d^2 - 350d^3 - 304d^4 \\
E(d,4) &= 60 + 184d - 81d^2 - 210d^3 - 96d^4 \\
E(d,5) &= 72 + 162d - 284d^2 - 348d^3 - 200d^4
\end{aligned}
$$

für alle $d \geq 3$. Außerdem ist

$$
\begin{aligned}
q(L_{d,n}) = (\ &(96d^2 + 72d - 75)n^2 + (100d^2 + 290d + 153)n + \\
&(26d^2 - 529d - 478 - d \bmod 2)\ )/2 \quad \text{für alle } d \geq 4\ .
\end{aligned}
$$

Die Periodenlänge dieser Spiele ist durch den von d abhängigen Einbruchfaktor $f(d)$ gekennzeichnet, um den ich die Gleichung $p(L_{n,d}) = \ldots$ zur Vermeidung eines langen Bruchstriches erweitert habe. Für kleine Werte d gibt folgende Tabelle einen Überblick über $f(d)$:

	0	1	2	3	4	5	6	7	8	9
0	-	-	.	.	3	.	.	.	.	2
1	.	3	.	2	.	.	.	.	.	39
2	.	2	45	.	.	4	.	5	5	3
3	.	5	.	2	.	.	6	.	.	.

Die Spalten sind mit $d \bmod 10$ numeriert und die Zeilen mit $d/10$ — hierbei wurde die Konstante 10 aus praktischen Erwägungen gewählt. Der Wert 1 wurde zur besseren Übersicht durch . dargestellt und die undefinierten Werte von $f(0)$ und $f(1)$ mit einem Strich. Bei dieser Familie tritt das Einbruchphänomen, dargestellt durch $f(d) \neq 1$, auch auf.

Um das Auftreten eines solchen Einbruchsfaktors f besser zu verstehen, untersuchte ich die Familien $L_{a,b} = (a, b, 2b - a, 2b)$ aus Klasse I. Nach der Analyse der Struktur dieser Spiele für $b \bmod 2a \in \{0, a, a+1\}$ und der Berücksichtigung aller Periodenwerte für $b \leq 160$ vermute ich:

Vermutung 10
Für alle $a < b \in \mathbb{N}$ sei $L_{a,b} = (a, b, 2b-a, 2b)$. Dann gilt:

$$p(L) = ((b-a)^2 + (2b+a-r)\tfrac{b-a+r}{2})/f \quad \textit{mit} \quad r = |b \bmod 2a - a| \quad \textit{und}$$
$$f = \begin{cases} a & \textit{wenn } b \bmod 2a = 0 \\ \frac{b-a}{2} & \textit{wenn } b \bmod 2a = a \\ \max\{\, k \in \mathbb{N} : k < a,\ \Delta(k) < a,\ k|\Delta(k)\,\} & \textit{sonst} \end{cases}$$
$$\textit{wobei} \quad \Delta(n) := |b \bmod \tfrac{2an}{\mathrm{ggt}(a,n)} - a| \quad \textit{für alle } n \in \mathbb{N} \textit{ ist.}$$

Insbesondere erhält man $f|\ b-a$ und $f|\ r$ für alle $L_{a,b}$ und außerdem $p(L) = 4b - a$, wenn $b \bmod 2a = a$.

Hier erzeugen die Teilbarkeitsverhältnisse von b und den Vielfachen von $2a$ Unterperioden der zu erwartenden (Grund-)Periode.

Maximales Wachstum der Vorperioden

Abschließend noch eine Bemerkung zum Wachstum der Vorperioden von 4-Zug-Spielen: Wie man an nachfolgender Tabelle der Rekordhalter mit steigendem s_4 sieht, wachsen diese im Gegensatz zu den Vorperioden der 3-Zug-Spiele stark an. Die letzte Spalte der Tabelle gibt den Exponenten x an, so daß $\max L^{\,x} = q(L)$ gilt. Vorperioden-Rekordhalter mit $20 < s_4 < 185$:

{ 1, 8, 20, 21}	p=	189	q=	278	1.848
{ 1, 8, 22, 23}	p=	118	q=	349	1.867
{ 1, 10, 26, 27}	p=	431	q=	566	1.923
{ 1, 12, 28, 29}	p=	377	q=	615	1.907
{ 1, 10, 28, 29}	p=	206	q=	817	1.991
{ 1, 12, 30, 31}	p=	162	q=	1055	2.027
{ 1, 12, 32, 33}	p=	236	q=	1422	2.076
{ 1, 12, 34, 35}	p=	318	q=	1581	2.071
{ 2, 20, 35, 37}	p=	481	q=	1909	2.092
{ 1, 14, 38, 39}	p=	356	q=	2498	2.135
{ 1, 16, 40, 41}	p=	376	q=	3669	2.210
{ 1, 18, 44, 45}	p=	502	q=	7269	2.335
{ 1, 20, 50, 51}	p=	11661	q=	12096	2.390
{ 1, 22, 54, 55}	p=	724	q=	21132	2.485
{ 1, 26, 64, 65}	p=	1114	q=	25024	2.426

{ 1, 24, 64, 65}	p=	984	q=	25044	2.426
{10, 26, 57, 67}	p=	77	q=	47982	2.563
{ 1, 34, 84, 85}	p=	1798	q=	60066	2.476
{ 1, 36, 86, 87}	p=	1842	q=	75305	2.514
{ 1, 32, 86, 87}	p=	1838	q=	92067	2.559
{ 1, 38, 94, 95}	p=	2200	q=	103185	2.535
{ 1, 40, 96, 97}	p=	2248	q=	123552	2.562
{ 1, 36, 96, 97}	p=	2436	q=	129041	2.572
{ 1, 38,102,103}	p=	2588	q=	139988	2.556
{ 1, 40,104,105}	p=	2640	q=	141743	2.548
{ 1, 40,106,107}	p=	2902	q=	147970	2.547
{ 1, 42,108,109}	p=	2742	q=	155924	2.548
{ 1, 46,110,111}	p=	156109	q=	203726	2.595
{ 1, 44,116,117}	p=	3408	q=	240195	2.601
{ 1, 44,118,119}	p=	3466	q=	279249	2.623
{ 1, 44,120,121}	p=	3764	q=	298106	2.628
{ 1, 46,122,123}	p=	3828	q=	301033	2.621
{16, 51,109,125}	p=	141	q=	310574	2.619
{ 1, 46,126,127}	p=	4204	q=	358378	2.640
{ 1, 48,128,129}	p=	4272	q=	366199	2.636
{ 1, 50,132,133}	p=	4406	q=	431932	2.653
{ 1, 50,134,135}	p=	4740	q=	715527	2.748
{25,112,114,137}	p=	12626	q=	782147	2.758
{ 1, 52,142,143}	p=	5306	q=	782291	2.734
{ 1, 54,144,145}	p=	5670	q=	1 048197	2.785
{ 1, 56,150,151}	p=	10 452549	q=	2 375646	2.926
{ 1, 60,160,161}	p=	6940	q=	2 915588	2.929
{15,116,168,183}	p=	3429	q=	2 963148	2.860

Abgesehen von $L = \{25, 112, 114, 137\}$ genügen die Vorperioden-Rekordhalter alle der Bedingung $s_1 + s_3 = s_4$ von Klasse I. Und fast alle haben $min\ L = 1$. Mit diesen beiden Einschränkungen setzte ich meine Suche nach Rekordhaltern bis $s_4 < 299$ fort und erhielt noch:

```
{ 1, 76,210,211}    p=     12046     q= 3 457879    2.813
{ 1, 78,214,215}    p=     12704     q= 3 581799    2.809
{ 1, 82,218,219}    p=     12944     q= 6 936436    2.923
{ 1, 88,234,235}    p=     14830     q=10 292912    2.957
{ 1,108,264,265}    p=     16740     q=22 601104    3.034
{ 1,110,292,293}    p=     22594     q=34 820328    3.057
```

Generell könnte man dieses Wachstum in der Vorperiode als polynomielles Wachstum interpretieren — im Gegensatz zu den Perioden-Rekordhaltern, wo auch exponentielles Wachstum vorliegen könnte. Denkbar wäre noch ein Wachstum $\Theta(s_4^{\log s_4})$ der Vorperiodenlängen-Rekordhalter. Leider sind ab $\max L \approx 160$ die Rekordhalter dünner gesät und eine Extrapolation schwierig. Bemerkenswert ist, daß bei fast allen Rekordhaltern sowohl s_2 als auch s_3 gerade ist und deren Quotient $\frac{s_2}{s_3}$ um $\approx \frac{3}{8}$ schwankt. Ferner fällt auf, daß die Periodenlängen der Vorperioden-Rekordhalter in der Regel relativ klein sind und scheinbar wie $\approx \max L^{1.76}$ wachsen.

Symmetrische Zugmengen in Subtraktions-Spielen

Um ein Gefühl für die erreichbaren Periodenlängen bei Zugmengen mit mehr als 4 Zügen zu bekommen, berechnete ich die Periodenlängen aller Zugmengen mit $\max L < 30$. Die Liste der Rekordhalter mit anwachsendem $\max L$ enthält 44 Zugmengen:

1	{ 1 }	p=	2	q=	0	
1	{ 2 }	p=	4	q=	0	
1	{ 3 }	p=	6	q=	0	
1	{ 4 }	p=	8	q=	0	
1	{ 5 }	p=	10	q=	0	
1	{ 6 }	p=	12	q=	0	
1	{ 7 }	p=	14	q=	0	
3	{ 2, 5, 7 }	p=	22	q=	0	*
4	{ 1, 4, 7, 8 }	p=	25	q=	0	*:
4	{ 2, 5, 7, 9 }	p=	26	q=	0	*
5	{ 2, 3, 6, 7, 9 }	p=	28	q=	0	**
3	{ 3, 7,10 }	p=	45	q=	0	*
3	{ 3, 8,11 }	p=	54	q=	0	*
5	{ 3, 4, 8, 9,12 }	p=	54	q=	0	**
4	{ 1, 6,11,12 }	p=	61	q=	0	*:
3	{ 4, 9,13 }	p=	76	q=	0	*
4	{ 2, 3,11,14 }	p=	79	q=	0	*
5	{ 3, 5, 9,11,14 }	p=	94	q=	0	**
3	{ 4,11,15 }	p=	100	q=	0	*
4	{ 4, 6,11,15 }	p=	113	q=	0	*
6	{ 3, 8, 9,10,12,15 }	p=	168	q=	0	*
6	{ 3, 4, 9,12,14,15 }	p=	168	q=	0	*
6	{ 2, 3, 9,12,15,16 }	p=	168	q=	0	
5	{ 3, 8, 9,14,17 }	p=	204	q=	0	**
6	{ 2, 6, 9,12,16,18 }	p=	238	q=	0	**:
7	{ 2, 6, 7,11,12,16,18 }	p=	314	q=	0	***
5	{ 4, 7,14,17,21 }	p=	444	q=	0	**
7	{ 3, 5, 9,13,17,19,22 }	p=	494	q=	0	***
7	{ 2, 7,10,12,15,20,22 }	p=	516	q=	0	***

6	{ 5,11,12,13,19,24 }	p= 689	q= 0	**:
5	{ 3,11,13,21,24 }	p= 767	q= 0	**
9	{ 3, 4, 8,10,14,16,20,21,24 }	p= 790	q= 0	****
6	{ 3,11,12,21,22,24 }	p= 854	q= 0	*:
6	{ 1, 3,12,21,23,24 }	p= 1107	q= 0	**:
5	{ 3,12,14,23,26 }	p= 1215	q= 0	**
5	{ 3,11,15,23,26 }	p= 1256	q= 0	**
7	{ 5,12,13,14,15,22,27 }	p= 1276	q= 0	***
5	{ 2, 8,19,25,27 }	p= 1892	q= 0	**
9	{ 2, 3, 8,13,14,19,24,25,27 }	p= 1937	q= 0	****
5	{ 6, 9,19,22,28 }	p= 2639	q= 0	**
6	{ 5,13,14,15,23,28 }	p= 3424	q= 0	**:
7	{ 4, 7,13,15,21,24,28 }	p= 3624	q= 0	***
5	{ 1,12,17,28,29,}	p= 4896	q= 0	**
6	{ 4,10,15,20,26,30 }	p= 5296	q= 0	**:

Die erste Zahl in jeder Zeile gibt die Mächtigkeit der Zugmenge L an, dann folgt explizit die Zugmenge, deren Elemente der Größe nach steigend sortiert sind, so daß sie mit $\max L$ abschließt. Dann folgen die Periodenlänge, die Vorperiodenlänge und eventuell mehrere *- und/oder :-Zeichen. Dabei steht je ein $*$ für das Vorhandensein eines weiteren Indexpaares (i, j) mit $i < j$ aus den Indizes der Zugliste, so daß gilt: $s_i + s_j = s_{\max L}$. Der Doppelpunkt bedeutet: Es existiert ein Index i aus der Menge der Indizes der Zugliste, so daß gilt: $2s_i = s_{\max L}$.

Es fällt auf, daß Zugmengen, deren Züge nur mit den Eigenschaften $*$ und : charakterisiert werden können, die überwiegende Mehrzahl bilden. Wenn man noch bedenkt, daß ein Spiel mit $|L|$ Zügen höchstens $(|L| - 1)/2$ Indexpaare haben kann, die $*$ erfüllen und nur die Zugmengen mit geradem $|L|$ und durch 2 teilbarem $\max L$ die Eigenschaft : aufweisen können, ist diese Häufigkeit noch bemerkenswerter.[12] Solche „symmetrischen“ Zugmengen will ich genauer analysieren und benötige daher die

Definition 11
Eine endliche Menge $L \subset \mathbb{N}$ heiße symmetrisch *falls für alle $s \in L$ gilt, daß $\max L - s \in L \cup \{0\}$ ist.*

[12] Die in Vermutung 10 betrachtete 2-parametrige Familie von Subtraktions-Spielen beschreibt schon alle diese symmetrischen Zugmengen mit 4 Zügen.

Somit kann eine symmetrische Zugliste L, mit $s_k \equiv \max L$,
als $L = (s_1, s_2, \ldots, s_k - s_2, s_k - s_1, s_k)$ geschrieben werden.

Beobachtung 1 ermöglicht es, aus $\max L$ aufeinanderfolgenden $v(n)$-Werten den nächsten zu berechnen. Unter der Rückwärtsberechnung möchte ich die Berechnung des vorangehenden Wertes einer $\max L$ Werte langen Folge von $v(n)$-Werten verstehen. Mit dieser Begriffsbildung können wir die symmetrischen Zugmengen anschaulich auffassen:

Satz 12
Die symmetrischen Zugmengen sind genau diejenigen, die eine eindeutige Rückwärtsberechnung aus jedem $\max L$ *Werte langen Abschnitt ihrer* $\mathcal{P}$-$\mathcal{N}$-*Folge gestatten.*

Um diesen Satz zu beweisen, führe ich das Konzept der 0-kollisions-freien 0-1-Folge in Bezug auf eine Zugmenge L ein.

Definition 13
Für jedes $m \in \mathbb{N}$ *wird* $(v_1, v_2, \ldots, v_m) \in \{0,1\}^m$ *0-kollisions-frei für* $L \subset \mathbb{N}$ *genannt, falls für alle* $1 \leq i < j \leq m$ *mit* $v_i = 0$ *und* $v_j = 0$ *gilt, daß* $j - i \notin L$.

Folgerung: *Jeder Abschnitt einer durch eine Zugmenge* L *erzeugten* $\mathcal{P}$-$\mathcal{N}$-*Folge ist 0-kollisions-frei für* L.

Beweis des Satzes:
Gegeben seien eine Zugmenge L und eine 0-kollisions-freie Folge $v = (v_1, v_2, \ldots, v_{\max L})$ der Länge $\max L$. Wenn es ein $s \in L$ mit $v_s = 0$ gibt, so folgt $v_0 = 1$, weil s als Verlustzug von der Position s aus auf eine Gewinnposition führen muß. Anderenfalls gilt für alle $s \in L$, daß $v_s = 1$ ist — insbesondere auch $v_{\max L} = 1$. Da nach Definition 11 jeder Zug von Position $\max L$ auf Positionen der Menge $L \cup \{0\}$ führt, wir aber mindestens eine Verlustposition für den Gewinnzug von der Position $\max L$ aus brauchen, kann dies nur v_0 sein. Somit ist v_0 eindeutig zu 0 bestimmt. Die Folgerung im letzten Fall ist für nicht-symmetrische Zugmengen falsch, und v_0 kann, abhängig von den anderen Werten $i \notin L$, sowohl 0, als auch 1 sein. □

Als eine Konsequenz der Rückwärtsberechenbarkeit erhalten wir sofort die

Folgerung 14
Aus „L ist symmetrisch" folgt $q(L) = 0$.

Man sollte große Werte für die Periodenlängen solcher symmetrischer Zugmengen erwarten, da keine 0-1-Werte in Vorperioden „verschwendet" werden. Dies erscheint mir plausibel, da die durchschn. Zykluslänge für Randomabbildungen einer n-elementigen Menge in sich nur $\sim \sqrt{2\pi n}/4$ beträgt [Knu69, p. 519], aber die durchschn. Länge der Zyklen bei Random-bijektionen wie $\sim 2n/\ln n$ wächst [Knu68, p. 518]. Des weiteren wird unter dem Gesichtspunkt der praktischen Berechnung der Periodenlängen die Erkennung der Periode wesentlich durch die Tatsache vereinfacht, daß man nur auf das erste Erscheinen von $\max L$ aufeinanderfolgenden $\mathcal{P}$-Positionen achten muß.

Von den Rekordhaltern der 44 Zugmengen mit $\max L < 30$ waren nur sieben nicht symmetrisch, der letzte mit $\max L = 24$. Daher beschloß ich, noch nach Rekordhaltern unter den symmetrischen Zugmengen bis $\max L = 60$ zu schauen, mit folgendem Ergebnis:

```
6    { 4,10,15,20,26,30 }                    p=    5296
5    { 6,15,17,26,32 }                       p=    8530
8    { 2, 5,11,17,23,29,32,34 }              p=   10419
5    { 6,14,21,29,35 }                       p=   10766
5    { 5,12,23,30,35 }                       p=   11693

5    { 4, 9,28,33,37 }                       p=   21945
5    { 3,18,20,35,38 }                       p=   23340
5    { 3,11,27,35,38 }                       p=   25285
5    { 7,17,22,32,39 }                       p=   34461
5    { 3,19,23,39,42 }                       p=   48965

9    { 2, 3, 8,13,29,34,39,40,42 }           p=   50968
5    { 9,14,29,34,43 }                       p=   57928
5    { 5,12,31,38,43 }                       p=   62552
5    { 8,17,27,36,44 }                       p=  142522
6    { 8, 9,24,39,40,48 }                    p=  168542

5    { 5,19,30,44,49 }                       p=  168819
9    { 5, 6,18,19,30,31,43,44,49 }           p=  200208
5    { 5,21,28,44,49 }                       p=  213168
5    { 6,23,27,44,50 }                       p=  394814
5    {13,19,33,39,52 }                       p=  413557

5    { 4,17,36,49,53 }                       p=  460606
5    {12,17,38,43,55 }                       p=  469851
5    { 5,21,34,50,55 }                       p=  512336
15   { 5, 6,15,16,17,18,27,28,37,38,39,40,49,50,55 }
```

```
                                                    p=  955307
5   { 7,25,31,49,56 }                               p= 1400106
5   {11,26,33,48,59 }                               p= 1851120
5   {11,27,33,49,60 }                               p= 2722797
```

Unter diesen 27 Rekordhaltern sind nur sechs mit $|L| \neq 5$. Deshalb konzentrierte sich meine Forschung nun auf $|L| = 5$ in der Hoffnung, besonders stark wachsende Familien von Subtraktions-Spielen zu finden.

Der Fall symmetrischer L mit $|L| = 5$

In [AB95] wurde $L_n = (n, 8n, 30n+1, 37n+1, 38n+1)$ als ein Kandidat für superpolynomielles Periodenlängenwachstum in n genannt. Dies scheint mir hochspekulativ, da nur drei Werte von Periodenlängen in dieser Familie berechnet werden konnten.[13] Diese Familie wurde aus der Zugmenge $(1, 8, 31, 38, 39)$ mit Periodenlänge 11757 abgeleitet, die angeblich die längste Periode in ihrem untersuchten Bereich von $s_1 = 1$, $s_2 < 10$ und $s_3, s_4, s_5 < 50$ hätte. Dabei übersahen Althöfer und Bültermann das Subtraktions-Spiel $L = (1, 8, 40, 47, 48)$, welches die Periodenlänge 106901 hat. Empirisch zeigt sich, daß, wenn man $s_1 = 1$ setzt, s_2 gerade und mindestens 8 sein muß, um lange Perioden einer symmetrischen Zugmenge $(1, s_2, s_5 - s_2, s_5 - 1, s_5)$ zu erreichen. Als zwei weitere Kanditaten für superpolynomielles Wachstum, aber mit wesentlich kleineren Koeffizienten in der Darstellung ihrer Zugmenge, möchte ich $(2n, 5n-1, 5n, 8n-1, 10n-1)$ und $(n, 2n+2, 3n+1, 4n+3, 5n+3)$ nennen. Der erste wurde für $1 \leq n \leq 14$ und der zweite für $1 \leq n \leq 27$ berechnet:

```
{ 2, 4, 5,  7,  9}   p=                 11
{ 4, 9,10, 15, 19}   p=                156
{ 6,14,15, 23, 29}   p=                390
{ 8,19,20, 31, 39}   p=              16495
{10,24,25, 39, 49}   p=              55144
{12,29,30, 47, 59}   p=             231919
{14,34,35, 55, 69}   p=             587034
{16,39,40, 63, 79}   p=          12 813100
{18,44,45, 71, 89}   p=         226 715886
{20,49,50, 79, 99}   p=         316 411174
{22,54,55, 87,109}   p=        8023 143286
{24,59,60, 95,119}   p=       26323 881364
{26,64,65,103,129}   p=      644924 208430
{28,69,70,111,139}   p=  8 219454 272099

{ 2, 6, 7, 11, 13}   p=                 42
{ 3, 8,10, 15, 18}   p=                 84
{ 4,10,13, 19, 23}   p=                422
{ 5,12,16, 23, 28}   p=                701
{ 6,14,19, 27, 33}   p=               4263
{ 7,16,22, 31, 38}   p=               2548
```

[13] Ich habe berechnet, daß $p(L_4) > 10^{13}$ ist.

{ 8,18,25, 35, 43}	p=	3212
{ 9,20,28, 39, 48}	p=	26378
{10,22,31, 43, 53}	p=	29532
{11,24,34, 47, 58}	p=	599922
{12,26,37, 51, 63}	p=	560436
{13,28,40, 55, 68}	p=	1 808116
{14,30,43, 59, 73}	p=	9 434870
{15,32,46, 63, 78}	p=	10 040454
{16,34,49, 67, 83}	p=	51 215550
{17,36,52, 71, 88}	p=	34 849013
{18,38,55, 75, 93}	p=	90 930044
{19,40,58, 79, 98}	p=	2072 869447
{20,42,61, 83,103}	p=	4829 161440
{21,44,64, 87,108}	p=	1617 701526
{22,46,67, 91,113}	p=	44131 069814
{23,48,70, 95,118}	p=	10480 817688
{24,50,73, 99,123}	p=	154031 648296
{25,52,76,103,128}	p=	185509 191703
{26,54,79,107,133}	p=	520949 268553
{27,56,82,111,138}	p=	690411 987124

Um mehr Daten zu erhalten, berechnete ich die Periodenlängen aller symmetrischen 5-Zugmengen L mit $\max L = s_5 \leq 117$. Unter diesen Werten entdeckte ich viele Familien, deren Periodenlängen linear, quadratisch und einige kubisch, wie z.B. $p((n, 3n{+}1, 8n{+}1, 10n{+}2, 11n{+}2)) = 18n^3{+}22n^2{+}5n$, mit n zu wachsen scheinen. Aber alle stärker wachsenden Familien scheinen nicht polynomiell und auch nicht monoton zu wachsen. Es folgt zum Vergleich die Liste der Rekordhalter mit wachsendem $\max L$, wenn der größte Zug > 60 ist:

{13,22,39, 48, 61}	5 425890	0.3667
{14,25,40, 51, 65}	5 641324	0.3450
{ 1,20,46, 65, 66}	6 349943	0.3424
{13,24,42, 53, 66}	8 265820	0.3482
{11,27,41, 57, 68}	8 371535	0.3382
{ 4,17,52, 65, 69}	8 445262	0.3335
{ 5,31,38, 64, 69}	9 590030	0.3361
{16,33,36, 53, 69}	14 727438	0.3451
{ 3,16,56, 69, 72}	19 008790	0.3358

{ 7,32,40, 65, 72}	19 324100	0.3362
{ 3,25,48, 70, 73}	37 223762	0.3445
{ 5,22,53, 70, 75}	55 733746	0.3431
{13,29,47, 63, 76}	82 377265	0.3460
{15,38,40, 63, 78}	84 004408	0.3375
{ 6,24,55, 73, 79}	84 488980	0.3333
{10,36,43, 69, 79}	103 216390	0.3370
{16,28,51, 63, 79}	103 625196	0.3370
{ 5,37,43, 75, 80}	159 734125	0.3406
{17,30,51, 64, 81}	234 999993	0.3433
{ 1,14,68, 81, 82}	297 893623	0.3433
{17,31,51, 65, 82}	369 025660	0.3471
{ 7,17,66, 76, 83}	593 117616	0.3511
{19,30,57, 68, 87}	721 385889	0.3382
{ 8,18,71, 81, 89}	726 568568	0.3307
{ 3,22,68, 87, 90}	808 940992	0.3288
{ 7,39,51, 83, 90}	894 130513	0.3304
{17,39,51, 73, 90}	948 307620	0.3313
{18,35,56, 73, 91}	1113 591430	0.3302
{ 2,37,55, 90, 92}	1429 129954	0.3306
{23,33,59, 69, 92}	1490 009193	0.3312
{21,28,65, 72, 93}	2195 335756	0.3337
{26,32,61, 67, 93}	2237 357606	0.3340
{ 9,34,60, 85, 94}	2557 855512	0.3325
{ 3,29,66, 92, 95}	3373 467882	0.3332
{22,27,69, 74, 96}	4509 107445	0.3341
{19,40,57, 78, 97}	6893 465976	0.3369
{ 6,21,79, 94,100}	7555 433327	0.3281
{19,33,67, 81,100}	8648 484533	0.3301
{ 6,26,75, 95,101}	9212 033734	0.3278
{21,38,63, 80,101}	13258 137589	0.3329
{ 7,11,91, 95,102}	14101 385829	0.3305
{ 6,47,56, 97,103}	17362 165502	0.3302
{21,36,67, 82,103}	31118 313300	0.3384

{ 9,30,76, 97,106}	48579 415696	0.3349
{22,45,63, 86,108}	55297 937922	0.3304
{ 3,35,75,107,110}	58442 750785	0.3251
{21,49,61, 89,110}	65312 217509	0.3266
{ 6,43,69,106,112}	82100 824716	0.3237
{ 7,17,95,105,112}	107065 938457	0.3271
{ 9,22,91,104,113}	114539 466901	0.3251
{11,42,72,103,114}	118590 803273	0.3227
{25,52,62, 89,114}	135069 271955	0.3243
{15,27,88,100,115}	147429 129464	0.3226
{22,37,78, 93,115}	191943 666687	0.3259
{ 4,55,62,113,117}	224315 744884	0.3223
{23,48,69, 94,117}	312750 374565	0.3264
{25,42,75, 92,117}	320442 802924	0.3267

Der erste Eintrag in jeder Zeile dieser Tabelle ist die Zugmenge nach wachsenden Zügen geordnet, gefolgt von der Periodenlänge des zugehörigen Subtraktions-Spiels L, und eine Zahl c mit $2^{c \cdot \max L} \approx p(L)$. Man kann den Eindruck gewinnen, daß diese Zahl c asymptotisch zwischen 0.32 und $\approx 0.33 + \varepsilon$ variiert.

Wie im Fall der Erforschung der Zugmengen mit $|L| = 3$ zeichnete ich Diagramme der Periodenlängen für festes s_5. Als horizontale Achse wählte ich $\frac{s_1}{s_5}$ von rechts nach links zunehmend und als vertikale Achse $\frac{s_2}{s_5}$ von unten nach oben ansteigend. Allerdings unterschied ich nur Periodenwerte $p \leq s_1 + s_5$, $s_1 + s_5 < p \leq s_4 + s_5$, $s_4 + s_5 < p \leq {s_5}^2$ und $p > {s_5}^2$.

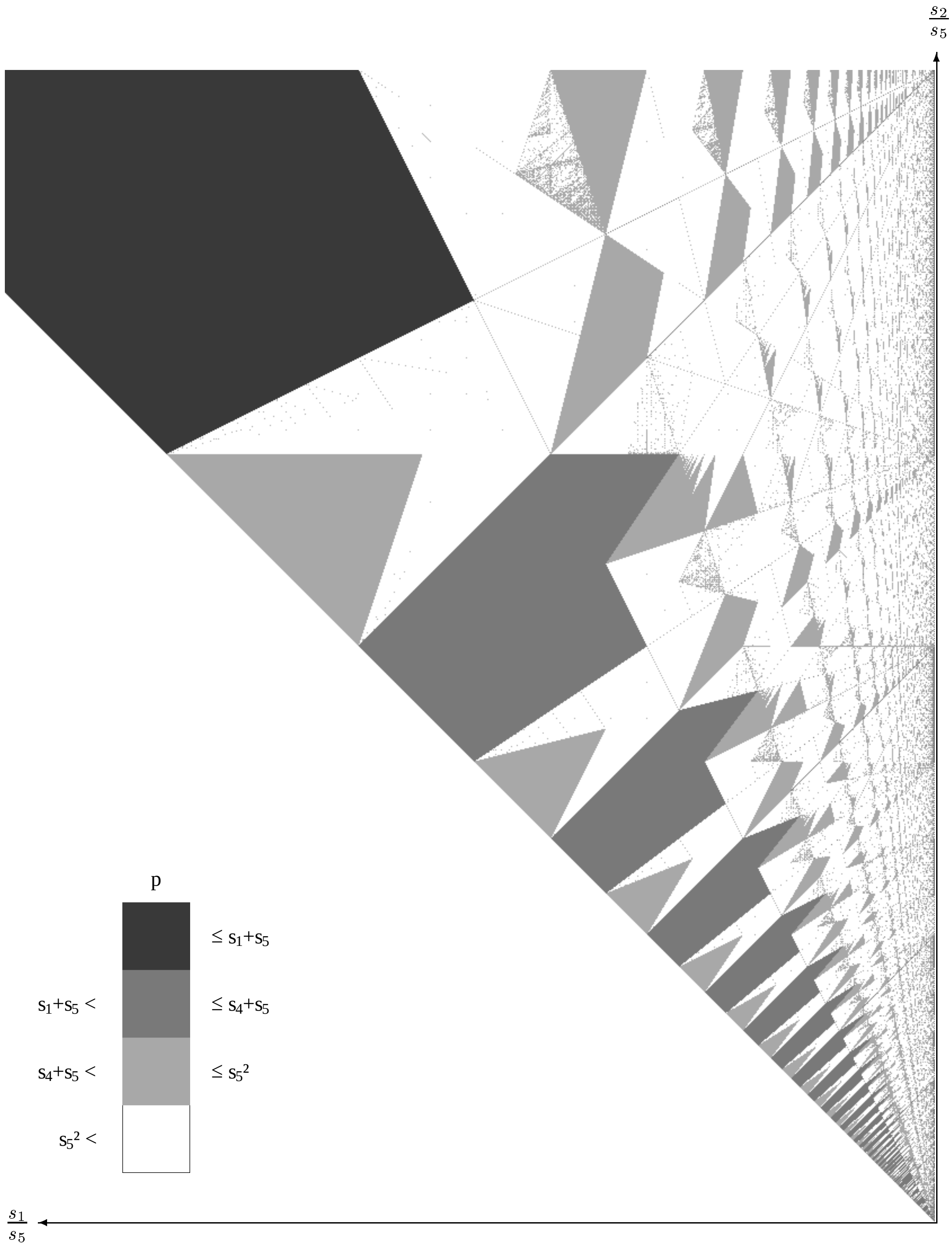


Bild viii $s_5 = 2048$ Periodenlängen der symmetrischen 5-Zug-Spiele wie beschrieben in vier Bereiche eingeteilt und durch verschiedene, in der Helligkeit ansteigende, Grauwerte dargestellt.[14]

[14]Die Zuordnung der Grauwerte zu den Bereichen ist so, daß insbesondere für $s_1 \to 0$ maximale visuelle Auflösung erreicht wird.

Vergleicht man dieses Diagramm mit ähnlichen für andere Werte von s_5, stellt man fest, daß die Verteilung der 4 Gebietsarten unabhängig von s_5 nach dieser Normierung ist. Betrachtet man die Werte $p(L)$, die in den dunklen Gebieten links der Geraden $2s_1 = s_2$ auftreten, so wird man zu folgender Aussage geführt:

Satz 15
Für jene Subtraktions-Spiele L, für die es ein $n \in \mathbb{N}$ gibt mit $2s_1 \geq s_2$ und $\frac{s_5}{n} \geq s_1 + s_2 \geq \frac{s_5+s_1}{n+1}$, ist die Periodenlänge $p(L) = s_5 + s_1 + (n-1)(s_1 + s_2) \leq s_5 + s_3$.

Der Beweis ergibt sich unmittelbar aus der Strukturanalyse. Man beachte dazu, daß für jedes symmetrische 5-Zug-Spiel $(s_1, s_2, s_5 - s_2, s_5 - s_1, s_5)$ mit $n := \lfloor \frac{s_5}{s_1+s_2} \rfloor$ und für das $2s_1 \geq s_2$ und $(n+1)(s_1+s_2) \geq s_5+s_1$ gilt, folgt:

1. $s_5 - s_1 \leq s_5 - s_2 + s_1$
2. $n(s_1 + s_2) \leq s_5$
3. $(n-1)(s_1 + s_2) + s_1 \leq s_5 - s_2$
4. $n(s_1 + s_2) \geq s_5 - s_2$

Dies ist für die mögliche Existenz des „eventuell nicht existenten“ Intervalls entscheidend, ebenso wie für mögliche Überlappungen seiner beiden Nachbarintervalle. Dies wird an der $\mathcal{P}$-$\mathcal{N}$-Folge solch eines Subtraktions-Spieles verdeutlicht, welches bis zur Position $s_5 + (n-1)(s_1 + s_2) + s_1$ folgende Struktur hat:

Anfangsindex	Länge	Gewinnzug	Kommentar
0	s_1	$-$	
s_1	s_1	s_1	
$2s_1$	$s_2 - s_1$	s_2	
$s_1 + s_2$	s_1	$-$	
$2s_1 + s_2$	s_1	s_1	
$3s_1 + s_2$	$s_2 - s_1$	s_2	
...	...	...	
$i(s_1{+}s_2)$	s_1	$-$	
$i(s_1{+}s_2) + s_1$	s_1	s_1	
$i(s_1{+}s_2) + 2s_1$	$s_2 - s_1$	s_2	
...	...	...	
$(n{-}1)(s_1{+}s_2)$	s_1	$-$	
$(n{-}1)(s_1{+}s_2) + s_1$	s_1	s_1	even. gewinnt noch
$(n{-}1)(s_1{+}s_2) + 2s_1$	$s_2 - s_1$	s_2	$s_5{-}s_2$ oder $s_5{-}s_1$
$n(s_1{+}s_2)$	$s_5{-}s_2{+}s_1{-}n(s_1{+}s_2)$	$s_5 - s_2$	even. nicht existent
$s_5 - s_2 + s_1$	$s_2 - s_1$	$s_5 - s_1$	even. verkürzt
s_5	s_1	s_5	
$s_5 + s_1$	s_1	$s_5 - s_2$	
$s_5 + 2s_1$	$s_2 - s_1$	$s_5 - s_1$	
$s_5 + s_1 + s_2$	s_1	s_5	
...	...	...	
$s_5 + (i{-}1)(s_1{+}s_2) + s_1$	s_1	$s_5 - s_2$	
$s_5 + (i{-}1)(s_1{+}s_2) + 2s_1$	$s_2 - s_1$	$s_5 - s_1$	
$s_5 + i(s_1{+}s_2)$	s_1	s_5	
...	...	...	
$s_5 + (n{-}2)(s_1{+}s_2) + s_1$	s_1	$s_5 - s_2$	
$s_5 + (n{-}2)(s_1{+}s_2) + 2s_1$	$s_2 - s_1$	$s_5 - s_1$	
$s_5 + (n{-}1)(s_1{+}s_2)$	s_1	s_5	

Falls $s_5{-}s_2{+}s_1 \leq n(s_1{+}s_2)$ ist, existiert das ab Position $n(s_1{+}s_2)$ aufgeführte Intervall nicht, weil es von seinem nachfolgenden Intervall ab Position $s_5 - s_2 + s_1$ übergedeckt wird. Ferner könnte dieses Intervall von dem vorvorherigen Intervall ab Position $(n-1)(s_1{+}s_2) + 2s_1$ teilweise abgedeckt und damit verkürzt sein. In jedem Fall sind alle letzten $s_5 \equiv \max L$ Positionen alles Gewinnpositionen. Daher muß sich diese Struktur immerzu wiederholen, sprich wir haben die Periode beschrieben. Auch kann keine echte Unterperiode vorliegen und somit ist die behauptete Periodenlänge bewiesen. $\square$

Als nächstes schaute ich, wie im Fall $|L| = 4$, die Werte $p(L)$ der Rekordhalter für wachsendes s_5 an. Diese können gut durch eine Funktion $\exp(as_5 + b)$ angenähert werden.

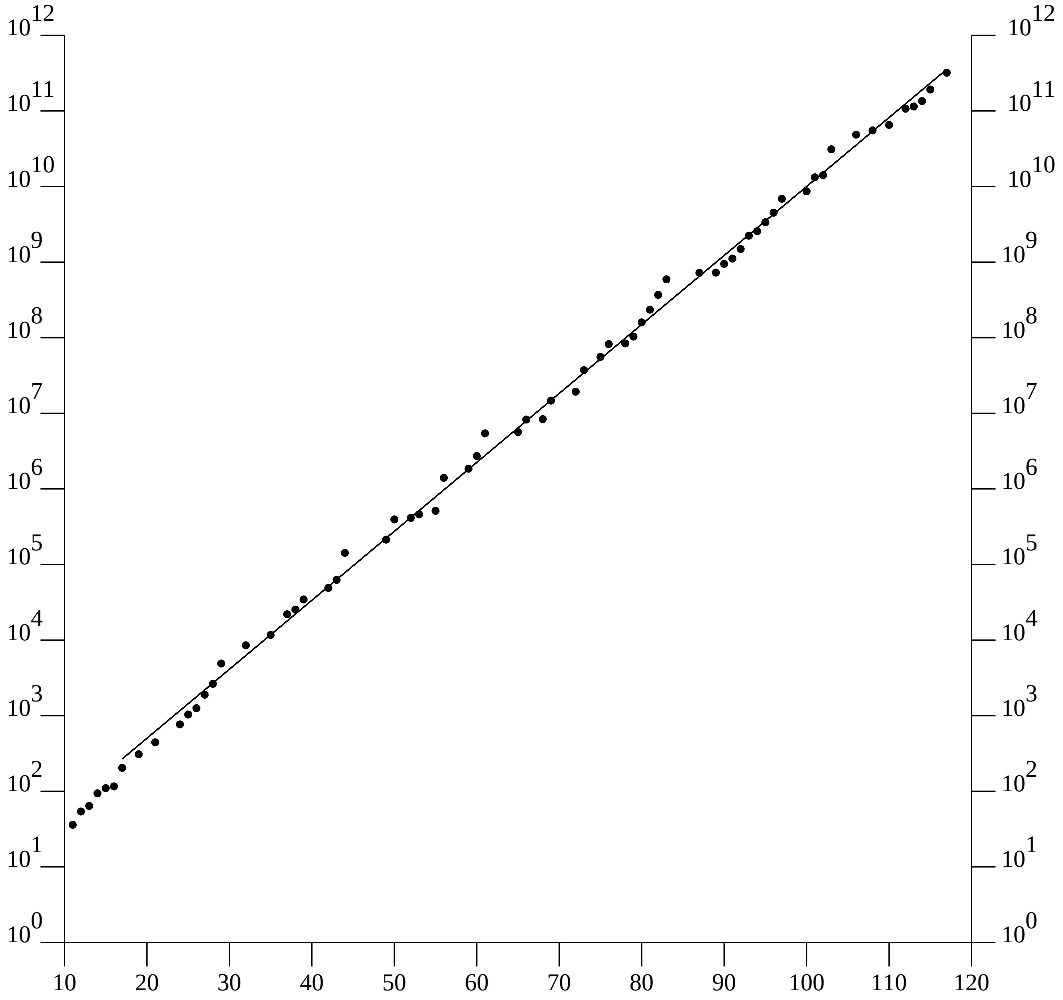


Bild ix s_5 gegen Werte der Rekordhalter von $p(L)$ logarithmisch aufgetragen.

Wenn man die besten Werte $a \approx 0.2102$ und $b \approx 2.014$ für den Bereich $16 < s_5 \leq 117$ wählt, wird der Korrelationskoeffizient ≈ 0.999.

Hier tabellarisch die Daten für zeilenweise festes $s_5 \geq 40$:

40	196821	171	1151.00	26538	0.1762
41	311133	190	1637.54	30912	0.1805
42	242445	190	1276.03	48965	0.1703
43	580841	210	2765.91	62552	0.1843
44	757219	210	3605.80	142522	0.1861

45	679167	231	2940.12	95208	0.1775
46	899884	231	3895.60	138037	0.1797
47	1 364407	253	5392.91	130210	0.1828
48	1 155486	253	4567.14	113828	0.1756
49	1 726139	276	6254.13	213168	0.1784
50	3 127347	276	11330.97	394814	0.1867
51	2 248854	300	7496.18	190404	0.1749
52	3 926019	300	13086.73	413557	0.1823
53	5 977413	325	18392.04	460606	0.1853
54	4 184167	325	12874.36	419097	0.1752
55	9 248930	351	26350.23	512336	0.1851
56	12 478794	351	35552.12	1 400106	0.1871
57	12 959836	378	34285.28	1 066690	0.1832
58	11 875997	378	31417.98	1 101605	0.1785
59	27 658723	406	68124.93	1 851120	0.1886
60	33 954855	406	83632.65	2 722797	0.1889
61	48 096983	435	110567.78	5 425890	0.1904
62	48 655288	435	111851.24	3 698041	0.1875
63	60 124856	465	129300.77	4 237374	0.1868
64	73 347042	465	157735.57	3 977697	0.1870
65	114 195411	496	230232.68	5 641324	0.1900
66	111 094917	496	223981.69	8 265820	0.1867
67	166 773511	528	315858.92	7 412007	0.1890
68	178 590272	528	338239.15	8 371535	0.1872
69	252 956665	561	450903.15	14 727438	0.1887
70	276 092479	561	492143.46	13 443110	0.1872
71	463 783555	595	779468.16	13 583914	0.1911
72	473 773586	595	796258.13	19 324100	0.1887
73	964 797292	630	1 531424.27	37 223762	0.1951
74	652 103373	630	1 035084.72	35 775691	0.1872
75	1306 190551	666	1 961247.07	55 733746	0.1932
76	1426 206055	666	2 141450.53	82 377265	0.1918
77	2374 543162	703	3 377728.54	76 878197	0.1952
78	1616 514682	703	2 299451.89	84 004408	0.1878
79	3408 369103	741	4 599688.40	103 625196	0.1942
80	3512 118963	741	4 739701.70	159 734125	0.1921
81	4895 336676	780	6 276072.66	234 999993	0.1932
82	5309 110612	780	6 806552.07	369 025660	0.1919
83	10490 433177	820	12 793211.19	593 117616	0.1972
84	7767 210003	820	9 472207.32	265 827603	0.1912
85	14795 461218	861	17 184043.23	339 540604	0.1960
86	15868 568511	861	18 430393.16	658 686708	0.1945
87	19834 816286	903	21 965466.54	721 385889	0.1943

88	21930 437496	903	24 286198.78	693 540293	0.1932
89	33508 117443	946	35 420842.96	726 568568	0.1953
90	34982 865999	946	36 979773.78	948 307620	0.1936
91	51796 421423	990	52 319617.60	1113 591430	0.1953
92	62343 280585	990	62 973010.69	1490 009193	0.1952
93	81535 623576	1035	78 778380.27	2237 357606	0.1955
94	90349 582916	1035	87 294283.01	2557 855512	0.1945
95	158461 454412	1081	146 587839.42	3373 467882	0.1979
96	132607 009543	1081	122 670684.13	4509 107445	0.1940
97	214159 851176	1128	189 858024.09	6893 465976	0.1965
98	263830 607963	1128	233 892382.95	6846 085105	0.1966
99	305054 253391	1176	259 399875.33	5074 081436	0.1957
100	373409 071146	1176	317 524720.36	8648 484533	0.1958
101	572613 415870	1225	467 439523.16	13258 137589	0.1977
102	464927 610429	1225	379 532743.21	14101 385829	0.1937
103	889836 195692	1275	697 910741.72	31118 313300	0.1977
104	937506 267315	1275	735 299033.19	15626 042249	0.1963
105	1 112513 107340	1326	838 999326.80	17163 718254	0.1957
106	1 358237 640524	1326	1024 311946.10	48579 415696	0.1957
107	2 366066 833912	1378	1717 029632.74	46656 422556	0.1987
108	2 152544 003691	1378	1562 078377.13	55297 937922	0.1960
109	3 561615 837809	1431	2488 899956.54	49428 195356	0.1985
110	3 343655 394493	1431	2336 586578.96	65312 217509	0.1961
111	4 030463 983569	1485	2714 117160.65	50374 489706	0.1957
112	6 097433 195465	1485	4106 015619.84	107065 938457	0.1976
113	8 669102 047969	1540	5629 287044.14	114539 466901	0.1987
114	7 229441 604509	1540	4694 442600.33	135069 271955	0.1953
115	13 451114 556005	1596	8428 016639.10	191943 666687	0.1987
116	13 143040 202704	1596	8234 987595.68	173480 964790	0.1968
117	19 939038 614081	1653	12062 334309.79	320442 802924	0.1984

Die erste Spalte gibt s_5 an, die nächste die Summe aller Periodenlängen, die dritte deren Anzahl, die vierte den Mittelwert $\bar{p}(s_5)$, der im nachfolgenden Bild aufgetragen ist, die fünfte das Maximum der Periodenlänge und die letzte den Exponenten x, so daß $e^{s_5 x} = \bar{p}(s_5)$ ist.

Die durchschnittliche Periodenlänge für festes s_5 trug ich gegen s_5 auf und war überrascht von einer ziemlich gleichmäßig, streng monoton anwachsenden Kurve dieser Werte.

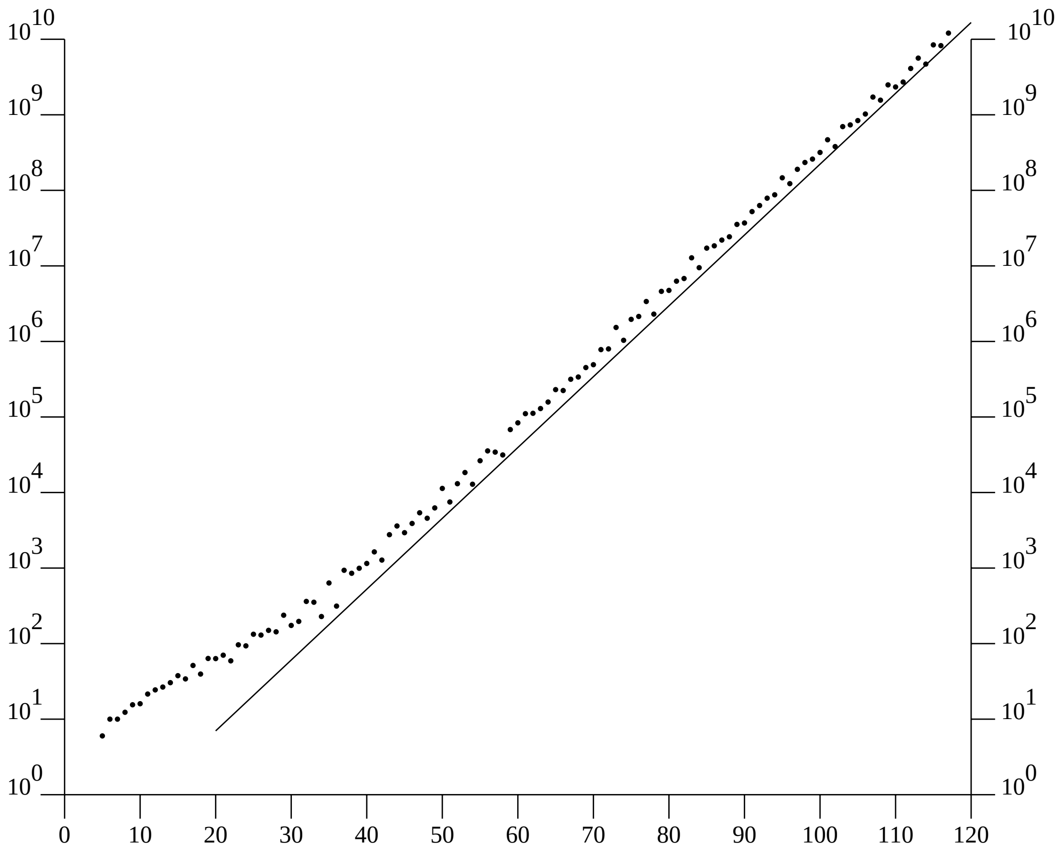


Bild x s_5 gegen das arithmetische Mittel der Werte $p(L)$ logarithmisch aufgetragen.[15] Vergleichsfunktion: $\frac{s_5}{150}\, e^{0.198 s_5}$

Es fällt eine Periodizität der Länge 6 am Ende der Kurve auf, die von der Teilbarkeit von s_5 durch 3 und 2 herrührt. Daher hier die — im Sinne der l_2-Norm bestimmten Konstanten a und b — beste Approximation durch eine Funktion $s_5 \exp(as_5 + b)$ für die Werte $57 < s_5 \leq 117$ in jeder Restklasse modulo 6 — dies sind jeweils zehn Datenpunkte:

[15] Optisch ist zu berücksichtigen, daß sich die Datenpunkte „von oben“ und stark bei kleinen Werten s_5 an die Vergleichsgerade „herankrümmen“.

s_5 mod 6	a	b	Korrelationskoeff.
0	0.194786	-4.672162	0.999396
1	0.197316	-4.600527	0.999707
2	0.199106	-4.954152	0.999513
3	0.197369	-4.756622	0.999713
4	0.202296	-5.267421	0.999686
5	0.198818	-4.701564	0.999745

Wenn wir a unabhängig von $s_5 \bmod 6$ unterstellen — bei b erscheint dies eher fragwürdig — sagt eine Kurvenanpassung im Bereich $37 < s_5 \leq 117$ eine Funktion $\Theta(s_5^{\varepsilon}) s_5 \exp(0.198 s_5)$, für festes $\varepsilon \approx 0$, als eine ausgezeichnete Approximation an diese Durchschnittswerte voraus, die vorzügliche asymptotische Qualität unter Berücksichtigung der Restklassen s_5 mod 6 hat. Dabei ist zu bedenken, daß vielleicht auch ein zu $\log s_5$ proportional vorhandener Faktor unberücksichtigt geblieben ist und daher die Bedeutung der Konstanten 150 in der Vergleichsfunktion, wie auch die von b in den Approximationsfunktionen, nicht überschätzt werden sollte.

Nichtsdestotrotz ist dies alles ein starkes Indiz für die Existenz von Subtraktions-Spiel-Familien, deren Periodenlängen exponentiell anwachsen, während ihr jeweils größter Zug nur linear wächst.

Der Fall symmetrischer Mengen L mit $|L| \geq 6$

Familien $(L_n)_{n\in\mathbb{N}}$, deren Zugmengen $L_n = (a_1n+b_1, a_2n+b_2, \ldots, a_{|L|}n+b_{|L|})$ mehr als fünf Elemente enthalten, können asymptotisch exponentiell wachsende Periodenlängen haben. Diese Formulierung wird nahegelegt, wenn man den Durchschnittswert der Periodenlängen von Subtraktions-Spielen für festes $\max L$ und festes $|L|$ der symmetrischen Zugmengen betrachtet. Diese Funktionsschar $\bar{p}_{|L|}(\max L)$ scheint sich ab $|L| = 7$ aus „fast gleichmäßig“ ansteigenden Kurven zusammenzusetzen, deren Anstieg mit wachsendem $|L|$ schwächer ist. Ich betrachte zur Vereinfachung nur ungerade Werte von $|L|$, da ansonsten, bei symmetrischen Mengen, $\max L$ gerade sein muß und der „mittlere“ Zug eine Sonderrolle einnimmt. Zuerst ein Diagramm, in dem nur die Kurven für $|L| = 1, 3, 5$ aufgetragen sind, deren zugehörige Subtraktions-Spiele schon in vorherigen Kapiteln besprochen wurden.

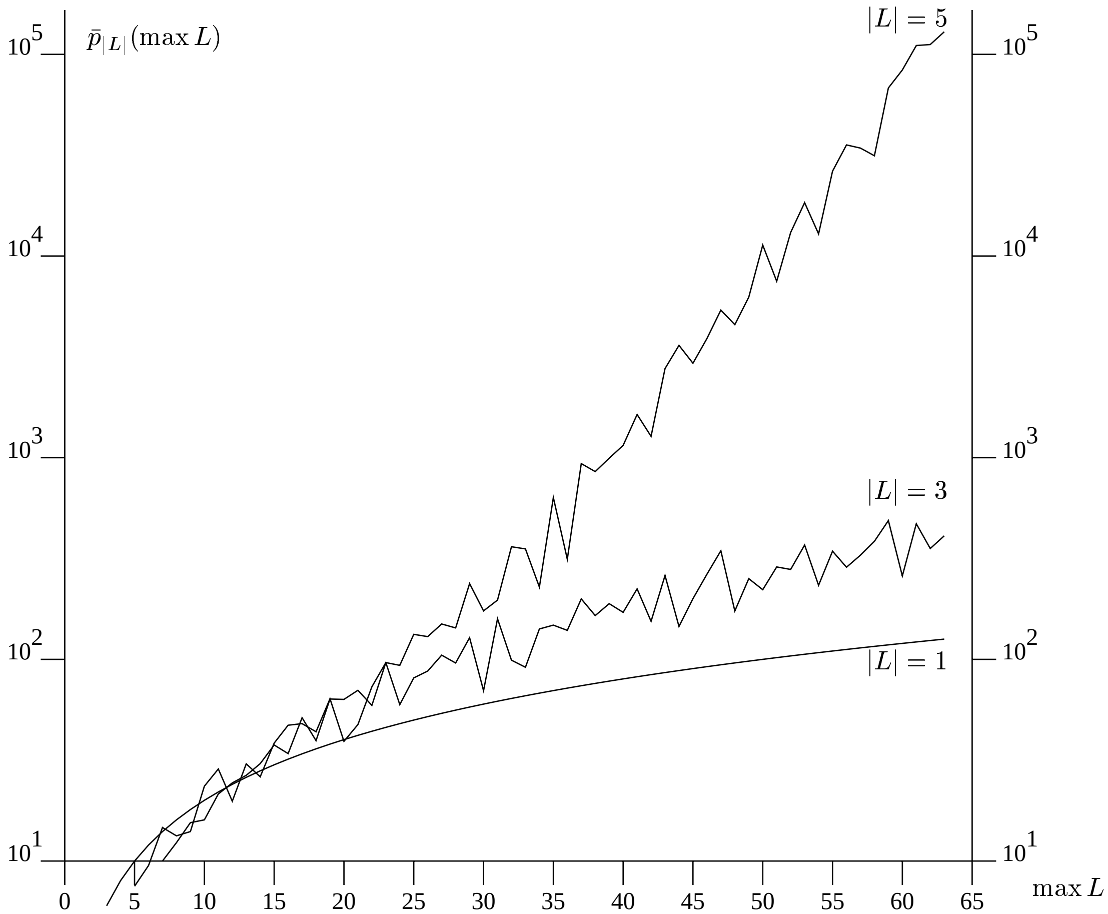


Bild xi Für kleine Werte von $|L|$ treten abgesehen vom Trivialfall $\bar{p}_1(\max L) = p(\{\max L\})$ starke Schwankungen auf.

Nun im selben Maßstab die Kurven für den $|L|$-Bereich $7, 9, 11, 13, \ldots$.

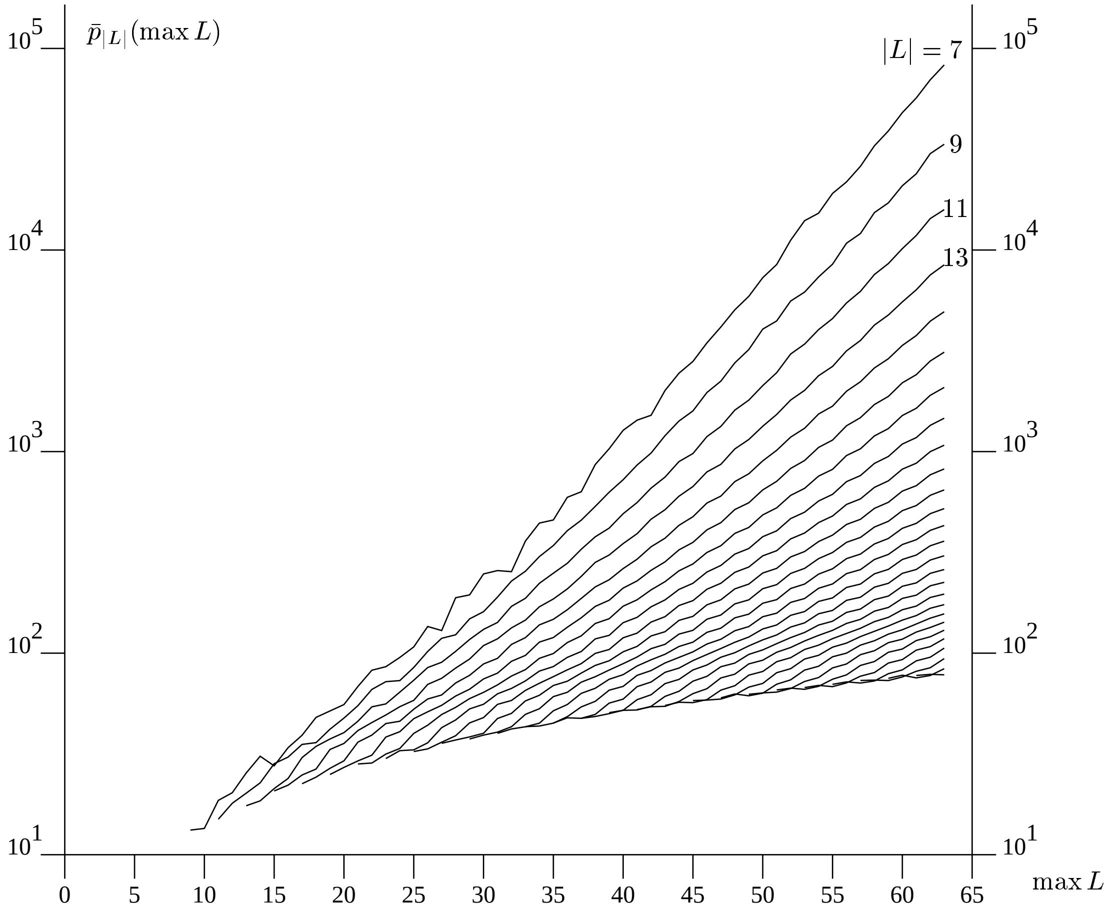


Bild xii Fast linear ansteigende Kurven für $|L| > 5$ kennzeichnen bei logarithmischer Achse exponentielles Wachstum.

Im letzten Bild fallen noch die Schwankungen der Kurve für $|L| = 7$ auf, sowie die Einhüllende der Mittelwerte von $p(L)$ für $\max L - 2 = |L|$, die eine lineare Abhängigkeit offenbart. Die linearen Kurvenanfänge für $|L| = \max L - 1$ und $\max L$ sind der Übersichtlichkeit halber nicht eingezeichnet.[16]

Um gegen den exponentiell anwachsenden Suchaufwand anzukommen, schränkte ich meinen Bereich durch $|L| \leq 9$ ein und schrieb ein spezielles C-Programm, welches die Folge der $v(n)$-Werte auf Bitpositionen von Rechnerworten kodiert und mittels 64-Bitarithmetik dann $\min\ L \cup \{64\}$ Werte

[16] In diesen beiden Fällen ist die Zugmenge bereits eindeutig bestimmt und $\bar{p}_{\max L-1}(\max L) = \frac{3}{2} \max L + 1$ und $\bar{p}_{\max L}(\max L) = \max L + 1$.

parallel ausrechnen kann. Leider wächst die Rechenzeit immer noch proportional zu $|L|$.[17]

In dem nun untersuchten Bereich symmetrischer Mengen L bis $\max L \leq 80$ und $|L| \leq 9$ war die letzte Ausnahme von $|L| = 5$ der Rekordhalter $L = (5, 6, 18, 19, 30, 31, 43, 44, 49)$. Aufgrund einer Idee, die von dem Ausreißer mit $|L| = 15$ herrührte, startete ich eine neue Durchsuchung der Zugmengen bis $\max L \leq 75$ und $|L| \leq \frac{1}{3} \max L$ unter der Einschränkung, daß die Differenzen von aufeinanderfolgenden Zügen aus der geordneten Zugliste höchstens drei verschiedene Werte annehmen dürfen.

Hierzu entwickelte ich einen neuen Algorithmus, der die jeweils letzten $\max L$ Werte der $\mathcal{P}$-$\mathcal{N}$-Folge als 0-1-Folge in einigen 64-Bit-Registern des Rechners hält. Ebenso wird die Zugmenge als Bitmaske in weiteren $\lceil (\max L)/64 \rceil$ CPU-Registern gehalten. Der neue $v(n)$-Wert kann nun durch *Undieren* der entsprechenden Register berechnet werden und das Hochzählen des Indexes n wird durch ein *Shiften* der Register simuliert. Die Laufzeit dieses Verfahrens ist nun proportional zu $\lceil (\max L)/\mathit{bits\_per\_register} \rceil$ [18] und ist bereits für die von mir untersuchten 5-Zugmengen effizienter als der „min L parallele“ Algorithmus.[19]

Das Ergebnis meines Programmlaufes mit $|L| \leq \frac{1}{3} \max L$ und der Zugdifferenzeinschränkung ergab:

```
{13,22,39,48,61 }                              p= 5 425890
{15,18,47,50,65 }                              p= 5 574318
```

[17] Um $p(\{113, 264, 377, 428\}) = 2\,521\,713\,304\,557$ mit $q = 6300$ zu bestimmen, benötigte ich 77 CPU-Stunden auf einem DEC Alpha-Server 2100 4/275 (Chip 21064 AXP) mit 4 MB Sec-Level Cache.

[18] Konkret kann ich auf einem DEC Alpha-Server 2100 z.B. in 95 sec $p(\{21, 44, 64, 87, 108\})$ zu $1\,617\,701\,526$ bestimmen, oder habe in 22 CPU-Stunden $p(\{27, 47, 81, 101, 128\})$ zu $1\,328\,462\,314\,506$ bestimmt. Für die in der nachfolgenden Beobachtung 16 erwähnte Zugmenge L_{10} werden mit einer „fein getunten“ Implementation dieses Verfahrens pro Minute $3 \cdot 10^9$ $v(n)$-Werte errechnet.

[19] Diese Aussage war nur bis Ende Oktober gültig, als ich eine algorithmisch weiterentwickelte Version des bit-parallel Verfahrens implemetierte, die sich das Auftreten langer $\mathcal{P}$-Folgen in der 0-1-Folge zunutze macht und $p(\{21, 103, 214, 235\}) = 38\,733953\,041818$ in circa 72 CPU-Stunden bestimmte — das heißt fast $9 \cdot 10^9$ $v(n)$-Werte pro CPU-Minute. Hierbei wurden 18 Positionen parallel ausgerechnet und nach einer Teilfolge von 72 aufeinanderfolgenden Gewinnpositionen gesucht. Die nachfolgende Bestimmung der Vorperiode dauert höchstens nochmal solange, aber bei korrekter Vorperiodenlängenschätzung — in diesem Fall habe ich q als ziemlich klein vermutet — weniger als eine CPU-Minute, da $q = 17114$ ist.

```
{14,25,40,51,65 }                                p= 5 641324
{13,30,36,53,66 }                                p= 7 735252
{13,24,42,53,66 }                                p= 8 265820
{11,27,41,57,68 }                                p= 8 371535
{16,33,36,53,69 }                                p=14 727438
{ 6, 7, 8,19,20,21,22,23,34,35,36,47,48,49,50,51,62,63,64,70 }
                                                 p=16 107110
```

Glücklicherweise brach der Rechner aufgrund eines Stromausfalls erst bei $\max L = 71$ zusammen, und meine Geduld wurde nicht weiter auf die Probe gestellt, denn ein weiteres Mitglied der Familie, dessen erstes aufgefallenes $L = (5, 6, 15, 16, 17, 18, 27, 28, 37, 38, 39, 40, 49, 50, 55)$ war, war soeben gefunden worden. Beide sind Mitglieder der Familie

Beobachtung 16 $(\overbrace{}^{n}\underbrace{|||||}_{n-2}\overbrace{}^{2n}\underbrace{||||||}_{n}\overbrace{}^{2n}\underbrace{|||||}_{n-2}\overbrace{}^{2n}\underbrace{||||||}_{n}\overbrace{}^{2n}\underbrace{|||||}_{n-2}\overbrace{}^{n}|)$
$L_n = (n+1, \ldots, 2n-2, 4n-1, \ldots, 5n-2, 7n-1, \ldots, 8n-4, 10n-3, \ldots, 11n-4, 13n-3, \ldots, 14n-6, 15n-5)$, *die eine Zug-Mengen-Familie mit* $|L| = 5n-5$ *und einer hochinteressanten Blockstruktur ist. Ihre zulässigen Züge sind zu fünf Blöcken von aufeinanderfolgenden Zahlen angeordnet, dazu die einzelne Zahl* $\max L$ *am Schluß. Die fünf Blöcke haben alternierend den Umfang von* $n-2$ *und* n *Zügen und sind durch* $2n$ *verbotene Züge getrennt. Ihre berechneten Periodenlängen sind:*

$p(L_2)$	$=$	260	$\max L_2 = 25$
$p(L_3)$	$=$	1 600	$\max L_3 = 40$
$p(L_4)$	$=$	955 307	$\max L_4 = 55$
$p(L_5)$	$=$	16 107 110	$\max L_5 = 70$
$p(L_6)$	$=$	192 628 021	$\max L_6 = 85$
$p(L_7)$	$=$	8 406 111 561	$\max L_7 = 100$
$p(L_8)$	$=$	604 771 076 188	$\max L_8 = 115$
$p(L_9)$	$=$	573 330 264 682	$\max L_9 = 130$
$p(L_{10})$	$>$	650 172 149 268 480	$\max L_{10} = 145$

Wie in jeder von mir gefundenen Familie mit stark anwachsenden Periodenlängen, beobachtet man auch hier Abweichungen von der Monotonie der Periodenlängen. Ist $\max L = 55, 70$ oder 115, übertreffen die Werte von $p(L)$ diejenigen der Zugmengen L mit $|L| = 5$ und vielleicht auch unendlich viele weitere für $n \geq 10$.

Verschiedene Ergebnisse

An dieser Stelle möchte ich einen einfachen Satz erwähnen, der unabhängig von $|L|$ gilt:

Satz 17
Für die Zuglisten L für deren Zugindizes i mit $2 \leq i \leq |L|$ *gilt* $s_i - s_{i-1} \leq s_1$ *folgt* $p(L) = s_1 + s_{|L|} \equiv \min L + \max L$. *Die* $\mathcal{P}$-$\mathcal{N}$*-Folge solcher Subtraktions-Spiele hat die Vorperiodenlänge* $q(L) = 0$ *und besteht aus einer endlosen Wiederholung von* s_1 $\mathcal{N}$*-Positionen gefolgt von* $s_{|L|}$ $\mathcal{P}$*-Positionen.*[20]

Beweis: Man schaue sich die $\mathcal{P}$-$\mathcal{N}$-Folge der Länge $p(L)$ ab $n = 0$ an. Die ersten s_1 Positionen müssen trivalerweise Verlustpositionen sein, da sie kleiner als der kleinste zulässige Zug s_1 sind. Ab dann gibt es stets einen Gewinnzug in das vorhergehende Verlustintervall. Dazu wähle für Position n mit $n < s_1 + s_{|L|}$ den Zug s_i, so daß $s_{i-1} \leq n - s_1 < s_i$ ist.[21] Aufgrund der Voraussetzung $s_i - s_{i-1} \leq s_1$ führt er in das letzte Verlustintervall und ist damit ein Gewinnzug. So haben wir ein Gewinnintervall der Länge $s_{|L|}$ konstruiert. Da nach $\max L$ Gewinnpositionen generell $\min L$ Verlustpositionen kommen müssen, haben wir wieder s_1 Verlustpositionen zu erwarten. Ersetzen wir n durch $n \bmod p(L)$ bei der Zugwahl, können wir nun den Beweis über die erste Periode hinaus mit den gleichen Argumenten stets wiederholen und somit die Struktur vollständig begründen. $\square$

Gewinn/Verlust-Folge versus Nim-Folge

Die Analyse der $\mathcal{P}$-$\mathcal{N}$-Folge für ein gegebenes L kann vereinfacht werden, indem man seine zugehörige Folge der Sprague-Grundy-Werte untersucht [BCG82, p. 58]. Allerdings besteht die Möglichkeit, daß sich die Perioden- oder Vorperiodenlänge dieser Nim-Folge von der ursprünlichen $\mathcal{P}$-$\mathcal{N}$-Folge unterscheidet. Für symmetrische Zugmengen L kann dies erst geschehen, falls $|L| \geq 5$ ist, und für nichtsymmetrische Zugmengen erst, wenn $|L| \geq 4$ ist. Beispiele mit minimaler Zugzahl und kleinstem $\max L$ sind $L = (4, 6, 11, 14)$ und $L = (5, 7, 14, 17)$, deren Periodenlängen als Nim-Folge betrachtet um einen Faktor 2 vergrößert sind. Für symmetrische Zugmengen mit minimalem $\max L$ bzw. kleinstem $|L|$ sind $L = (2, 3, 6, 9, 10, 12)$ und

[20] Berlekamp, Conway und Guy deuteten an, daß sie diesen Sachverhalt auch kennen [BCG82, p. 86].
[21] Technische Anmerkung: Sei $s_0 := 0$.

$L = (2, 5, 10, 13, 15)$ die ersten Beispiele deren Nim-Folge eine größere Periode als ihre $\mathcal{P}$-$\mathcal{N}$-Folge hat — hier ebenfalls um den Faktor 2.

Die Struktur der Familie $L_n = (n, n+3, 3n-1, 3n+3)$ für alle $n \geq 5$ wurde bestimmt und ihre Nim-Periodenlänge zu $12n+6$ mit Vorperiodenlänge $4n+3$ per Hand bewiesen, im Gegensatz zu ihrer $\mathcal{P}$-$\mathcal{N}$-Periode von $4n+2$ und Vorperiode von $3n$. Um einen deutlicheren Fall zu sehen, betrachte nun die folgende Zugmenge:

Beobachtung 18 *Für* $L_n = \{2, 4n-1, 4n+1, 4n+5, 8n-2\}$ *mit* $n \geq 2$ *kann die Nim-Folge in der Notation von Satz 8 geschrieben werden als:*

```
 <__AA>(4n)
 <BBCC>(4n)
  _DAE
 <__AA>(4n-4)
  _DAE
 <BBCC>(4n-4)
FOR i FROM 2 UPTO n-1 REP
   <_BAC>(4i-4)
    _DAE
   <__AA>(4n-4i)
    _CAD
   <_BAC>(4i-4)
   <BBCC>(4n-4i)
PER
REPEAT
   <_BAC>(4n-4)
    _DAB
    _CAD
   <_BAC>(4n-4)
ENDREP
```

Als eine Verallgemeinerung der auf Seite 16 vorgestellten Syntax können SGW-Worte aus verschiedenen SGW bestehen. Diese Werte werden in spitzen Klammern eingeschlossen und zyklisch von links nach rechts durchlaufen, bis die Gesamtlänge des Wortes aufgebaut ist. In unserem Beispiel ist die Wortlänge zwar stets ein Vielfaches von 4, jeder auftretenden Zykluslänge, allgemein braucht dies aber nicht zu gelten. Eine fehlende *Anzahl* nach einem SGW ist als 1 zu lesen.

Der Beweis wird dadurch erbracht, daß die SGW in dieser Syntax überprüft werden. Dies ist ein mittlerer Aufwand, da beide Schleifenrümpfe genau $8n >$

$\max L_n$ Werte in dieser Beschreibung enthalten und nur 6 Übergänge von 3 Blöcken zu verifizieren sind. Dies konnte tatsächlich mit einer späteren Version des `verify` Programms[22] erbracht werden. Somit haben wir:

Folgerung 19
Das Subtraktions-Spiel $L_n = \{2, 4n-1, 4n+1, 4n+5, 8n-2\}$ *besitzt die* $\mathcal{P}$-$\mathcal{N}$*-Periodenlänge* 4 *und Vorperiodenlänge* $8n^2-3$*, aber eine Periodenlänge von* $8n$ *und eine Vorperiodenlänge von* $8n^2-1$ *in seiner Nim-Folge.*

Beobachtung 20 *Mittels des Programms konnte auch die Struktur des ungeraden „Partners"* $L_n = \{2, 4n+1, 4n+3, 4n+7, 8n+2\}$ *verifiziert werden als:*

```
 <__AA>(4m)
  _BAC
 <BBCC>(4m)
  D_EA
 <__AA>(4m)
  DBEC
 <BBCC>(4m-4)
FOR i FROM 1 UPTO m-1 REP
   <B_CA>(4i)
    D_EA
   <__AA>(4m-4i)
    C_DA
   <B_CA>(4i-4)
   <BBCC>(4m-4i)
PER
REPEAT
   <B_CA>(4m)
    D_BA
    C_DA
   <B_CA>(4m-4)
ENDREP
```

Folgerung 21
Dieses Subtraktions-Spiel besitzt die $\mathcal{P}$-$\mathcal{N}$*-Periodenlänge* 4 *und Vorperiodenlänge* $8n^2+12n+2$*, aber eine Periodenlänge von* $8n+4$ *und eine*

[22] Quellcode im Programmanhang ab Seite 110

Vorperiodenlänge von $8n^2 + 12n + 4$ in seiner Nim-Folge.

Die letzten Folgerungen ergeben:

Satz 22
Der Quotient der Nim-Periodenlänge zu der ursprünglichen Periodenlänge kann für gewisse Zugmengen L beliebig groß werden.

Ein noch eindrucksvolleres Beispiel — die $\mathcal{P}$-$\mathcal{N}$-Periodenlänge ist nicht konstant, und auch die Vorperiode ist stark erhöht — stellt die Familie $L_n = (n, 2n+1, 4n+2, 5n+3, 6n+3)$ symmetrischer Zugmengen dar, deren Nim-Periodenlänge zu $10n^2 + 4n$ für alle $n \in \mathbb{N}$ bewiesen ist, im Gegensatz zur Periodenlänge von $10n + 4$ der ursprünglichen $\mathcal{P}$-$\mathcal{N}$-Folge. Deren Nim-Folge hat für $n \geq 3$ die grobe Form:

```
FOR i FROM 0 UPTO n+(n-3)/2 REP
   _n  An
   _1  X(n-1) A1
   _n  An  Y(n)
   Cn  Z(n)
   C1  U(n)
   Cn  V(2)
PER
REPEAT
   FOR h FROM 0 UPTO n-1 REP
      _n  An
      _1  X(n-1) A1
      _n  An  Y(n)
      Cn  Z(n)
      C1  U(n)
      Cn  V(2)
   PER
ENDREP
```

Dabei bezeichnen die Großbuchstaben `X,Y,Z,U` und `V` Worte von SGW als Funktionen in `i` bzw. `h` und n, aber von ihnen werden nur die Werte $2, 4$ und 5 bzw. `B,D` und `E` getroffen. Offensichtlich hängt die Länge dieser fünf Worte nur von n ab. Die genaue Definition dieser Funktionen für alle $n \geq 3$ und mit `j` $:=$ `i` mod $n =$ `h` mod n lautet:

```
      Definition von X                Variable    Bereich
------------------------------------------------------------------
                           B(n-1)         i       0
                        E1 B(n-2)         i       1
      B1       <DE>(j-1)   B(n-1-i)       i       2 ... n-1
 E(j) B1 E(j)  <DE>(n-2-2j)               i       n ... n+(n-3)/2
 E(j) B1 E(n-2-j)                         h       (n-1)/2 ... n-2
 E(n-1)                                   h       n-1
 E(j) B1 E(j) D1 E(n-3-2j)                h       0 ... (n-3)/2
```

```
      Definition von Y                Variable    Bereich
------------------------------------------------------------------
 Bn                                       i       0
 B1 D1 B(n-2)                             i       1 ... n-1
 Bn                                       i       n ... n+(n-3)/2
 Bn                                       h       (n-1)/2 ... n-2
 B1 D1 B(n-2)                             h       n-1
 Bn                                       h       0 ... (n-3)/2
```

```
      Definition von Z                Variable    Bereich
------------------------------------------------------------------
                                Dn        i       0
            D2    B1   <DE>(j)  D(n-3-j)  i       1 ... n-3
            D2    B1   <DE>(n-3)          i       n-2
            B1 D1 B1   <DE>(n-3)          i       n-1
            B1 D(2j+2) <DE>(n-3-2j)       i       n ... n+(n-3)/2
            B1 D(n-1)                     h       (n-1)/2 ... n-3
               Dn                         h       n-2
 B1 D1      B1 D(n-3)                     h       n-1
 B1 D(2j+3) E1 D(n-5-2j)                  h       0 ... (n-5)/2
 B1 D(n-1)                                h       (n-3)/2
```

```
      Definition von U                Variable    Bereich
------------------------------------------------------------------
           E1 B1            E(n-3)   D1   i       0
           E1 D1  B1        E(n-4)   D1   i       1
           E1 D1  E1 B(j-1) E(n-3-j) D1   i       2 ... n-3
           E1 D1  E1 B(n-3)               i       n-2
           B1 D1  E1 B(n-3)               i       n-1
           B(j+2) E1 B(n-3-j)             i       n ... n+(n-3)/2
           B(j+2) E1 B(n-3-j)             h       (n-1)/2 ... n-4
           B(n-1) D1                      h       n-3
 E1        B(n-1)                         h       n-2
```

```
 B1 D1     B(n-2)                          h       n-1
 B(j+2) E1 B(n-3-j)                        h       0 ... (n-3)/2

      Definition von V                 Variable    Bereich
-------------------------------------------------------------------
 D1 B1                                     i       0 ... n-2
 D2                                        i       n-1 ... n+(n-3)/2
 D2                                        h       (n-1)/2 ... n-3
 D1 B1                                     h       n-2
 D2                                        h       n-1
 D2                                        h       0 ... (n-3)/2


AUSNAHMEN  im Fall  n = 3
---------------------------------
Wenn i=n ,  dann sei X = B1 E1 .
Wenn i=1 ,  dann sei U = E1 D1 B1 .
Wenn i=n-1, dann sei U = B1 D2 .
```

Für gegebenes `i` bzw. `h` und n schaue man in der jeweiligen Zeile der gesuchten wortwertigen Funktion `X,Y,Z,U` oder `V` nach und berechne gegebenenfalls `j`. Die minimale Vorperiodenlänge dieser Nim-Folge ist: $(10n+4)(n+(n+(n \bmod 2)-6)/2)+3n+(n \bmod 2)-1 \sim 15n^2$ für alle $n \geq 5$, im Gegensatz zur Vorperiodenlänge von 0 der $\mathcal{P}$-$\mathcal{N}$-Folge.

Obere Schranken für Periodenlängen von Subtraktions-Spiele

Ein anderer Gesichtspunkt ist es, eine $\mathcal{P}$-$\mathcal{N}$-Folge, die durch eine Zugmenge L definiert ist, als ein Schieberegister der Länge $\max L$ mit $|L|$ Argumenten und der Operation `NAND` anzusehen. Das Schieberegister bildet die Menge aller zulässigen Schieberegisterzustände auf sich selbst ab. Für symmetrische L wirkt es als eine Bijektion auf der Menge der 0-kollisions-freien 0-1-Folgen der Länge $\max L$. Die Größe der Menge der 0-kollisions-freien Folgen hängt im allgemeinen stark von $|L|$ und $\max L$ ab. Sei bis zum Ende dieses Abschnitts $s := \max L$.

Eine triviale obere Schranke der Anzahl der 0-1-Folgen ist 2^s. Ich möchte nun diese Abschätzung für viele Mengen L auf eine andere exponentielle Schranke in s mit kleinerer Basis verbessern. Da wir, statt die Periodenlänge abzuschätzen, den Zustandsraum des Schieberegisters abschätzen werden, der eine obere Schranke der Periodenlänge liefert, benötigen wir

Definition 23
Sei S_L die Menge der 0-kollisions-freien Folgen der Länge $\max L$ zu einer Zugmenge L.

Relativ leicht gewinnen wir

Satz 24
Für alle L mit $|L| \geq 2$ gilt: $|S_L| \leq \sqrt{3}^s \approx 1.732^s$

O.b.d.A. sei $s_1 \leq s/2$ — ansonsten ist Satz 17 anwendbar und $p(L)$ durch $2s$ beschränkt. Nun betrachten wir die Folgenpositionen i und $i + s_1$ paarweise, wobei i von eins an $\mathbb{N}$ durchläuft solange $i + s_1 \leq s$ ist. Diese beiden Positionen fassen wir zu Paaren zusammen, solange i nicht bereits durch ein Paar mit $i + s_1$ erfaßt ist. So erhalten wir genau $s_1 \lfloor \frac{s}{2s_1} \rfloor$ Paare. Die verbleibenden $s \bmod 2s_1$ Positionen — diese bezeichnen die letzten Folgenpositionen — benutzten wir zum Erweitern von gewissen Paaren zu Tripel der Form $i, i + s_1, i + 2s_1$. Wir haben also letzlich eine disjunkte Zerlegung in $s \bmod 2s_1$ 3-elementige Tupel und $\frac{1}{2}(s - 3(s \bmod 2s_1))$ 2-elementige Tupel der Folgenpositionen geschaffen. Alle Tupel müssen 0-kollision-frei bezüglich s_1 sein. Damit können die Paare nur 3 von 2^2 und die Tripel nur 5 von 2^3 0-1-Wertebelegungen annehmen. Also ist der Zustandsraum der gesamten Folge — wegen $\sqrt[3]{5} < \sqrt{3}$ — durch $\sqrt{3}^s$ beschränkt. $\Box$

Um die Basis in der exponentiellen Schranke weiter zu drücken, benötigen wir

Definition 25
Sei $F(n) := \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \approx 1.6180^n$ die n-te Fibonacci-Zahl, die die Gleichung $F(n+2) = F(n+1) + F(n) \quad$ für alle $n \in \mathbb{Z}$ erfüllt.

Nun können wir

Satz 26 formulieren:

Wenn es ein $s_i, s_j \in L$ *gibt, mit* $\mathrm{ggt}(s_i, s_j) = 1$ *und* $s_i + s_j \leq s$*, dann gilt, daß* $|S_L| \leq F(s+2)$ *ist und somit auch* $p(L) < 2\left(\frac{\sqrt{5}+1}{2}\right)^s$ *ist.*

Der Beweis des Satzes wird nun sukzessive aus den vier Fällen/Bedingungen

1. $1 \in L$,
2. es gibt ein $s_i, s_j \in L :\ \mathrm{ggt}(s_i, s_j) = 1$ und $s_i + s_j = s$,
3. es gibt ein $s_i, s_j \in L :\ \mathrm{ggt}(s_i, s_j) = 1$ und $s_i + s_j = s \pm 1$,
4. es gibt ein $s_i, s_j \in L :\ \mathrm{ggt}(s_i, s_j) = 1$ und $s_i + s_j < s$.

aufgebaut.

Beweis: Wenn $1 \in L$, dann kann kein Element aus S_L zwei aufeinanderfolgende Nullen enthalten. Die Anzahl der 0-1-Folgen der Länge s, die dieser Einschränkung genügen, ist $F(s+2)$ (Beweis durch Induktion nach s): Sei $a_0(s)$ die Anzahl der 0-1-Folgen der Länge $s \in \mathbb{N}$, die auf 0 enden und $a_1(s)$ die Anzahl der 0-1-Folgen der Länge $s \in \mathbb{N}$, die auf 1 enden.[23] Offensichtlich ist $a_0(1) = a_1(1) = 1$. Ferner kann man alle 0-1-Folgen der Länge $s+1$ aus den 0-1-Folgen der Länge s gewinnen, in dem eine weitere Ziffer angehangen wird. Endet die 0-1-Folge der Länge s allerdings auf 0, darf nur eine 1 angehängt werden im Gegensatz zu beiden Möglichkeiten der Fortsetzung falls die Ausgangsfolge mit 1 endet. Formal bedeutet dies: $a_1(s+1) = a_1(s) + a_0(s)$ und $a_0(s+1) = a_1(s)$. Setzt man die zweite Rekursionsgleichung nach der Indexsubstituierung $s+1 \to s$ in die erste ein, erhält man die Rekursionsgleichung, der auch die Fibonnacci-Zahlen genügen. Unter Beachtung des Induktionsanfangs ist somit $a_1(s) = F(s+1)$ und $a_0(s) = F(s)$. Also ergibt sich die Gesamtzahl der 0-1-Folgen als disjunkte Vereinigung dieser beiden Folgenarten, sprich zu $F(s+1) + F(s) = F(s+2)$.

Im restlichen Beweis wird die 0-1-Folge der Länge $\max L$ mit einem auf $\max L$ Knoten definierten 0-1-knotengefärbten Graphen identifiziert.[24]

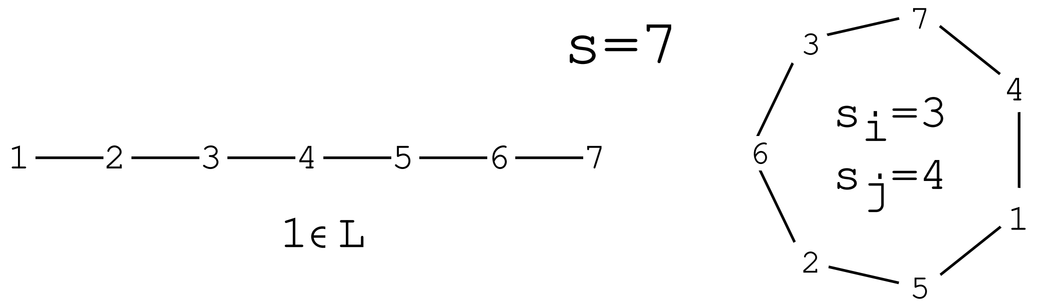
Wenn eine der drei anderen Bedingungen erfüllt ist, können keine zwei Nullen in einem Element aus S_L den Abstand s_i oder s_j haben. Die Bedingung $\mathrm{ggt}(s_i, s_j) = 1$ garantiert in diesen Fällen, daß jede Position innerhalb einer 0-1-Folge mit Schrittweiten aus $\{s_i, s_j\}$ von jeder anderen

[23] Die letzte Definition müßte man für $s \in \mathbb{N}_0$ als „... die nicht auf 0 enden." formulieren.

[24] Im vorherigen Fall hat man einen Pfad der Länge $\max L$ betrachtet.

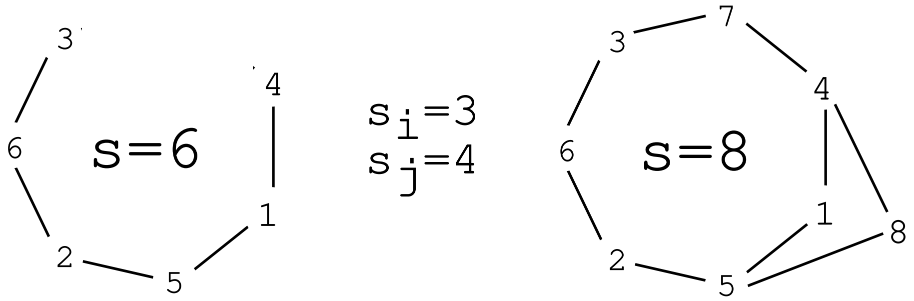
Position aus erreicht werden kann.

Mit der Bedingung im zweiten Fall sind genau zwei verschiedene Positionen innerhalb jeder 0-1-Folge durch Vorwärts- und Rückwärtsschreiten von jeder Position dieser Folge von S_L aus erreichbar. Dies läßt sich wie folgt als Graph auffassen: Die s möglichen Positionen in der 0-1-Folge bilden die Knoten des Graphen und ein möglicher „Schritt“ von einer Position, sprich Knoten, zu einer anderen wird durch eine Kante dargestellt. Da dieser Graph in jedem Knoten den Grad zwei hat und zusammenhängend ist, stellt er einen Kreis dar.



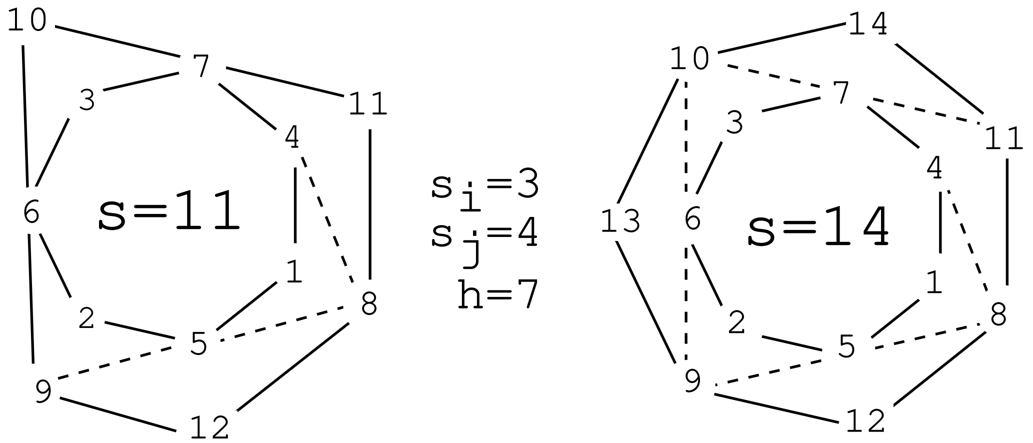
Alle möglichen Wertebelegungen der Knoten dieses Graphen müssen nur sicherstellen, daß keine zwei Nullen durch eine Kante verbunden sind, sprich aufeinander folgen.[25] Wir haben daher die Anzahl der zyklischen 0-1-Folgen der Länge s zu bestimmen, bei denen keine zwei Nullen aufeinander folgen. Dies ist nur eine leichte Verschärfung der Aussage, die die Bedingung $1 \in L$ nach sich zieht. Wir müssen somit zusätzlich darauf achten, mit welchem Wert unsere 0-1-Folge anfängt und falls dies 0 ist, die Anzahl der 0-1-Folgen die mit 0 endet von diesen noch abziehen. Wie man sich leicht klarmacht, ist die Anzahl der Folgen der Länge s die mit Null anfangen $F(s)$ und ihr Komplement $F(s+1)$. Die $F(s)$ mit Null beginnenden 0-1-Folgen zerfallen in $F(s-1)$ auf Eins endende und in $F(s-2)$ auf Null endende 0-1-Folgen. Also ist die Gesamtzahl der zweiten Bedingung genügenden 0-1-Folgen $F(s+2) - F(s-2) \equiv F(s+1) + F(s-1)$. Die zweite Summe kann man auch so interpretieren: Entweder ist eine beliebige aber feste Position im Zyklus 1, dann können die restlichen $s-1$ Positionen auf $F(s-1+2)$ Arten belegt werden, oder die ausgewählte Position ist 0. Im letzten Fall erzwingt dies auf der Vorgänger- und Nachfolgerposition jeweils eine 1 und die verbleibenden $s-3$ Positionen können auf $F(s–3+2)$ Arten belegt werden.

[25]Graphentheoretiker werden jetzt die Verwandtschaft der Begriffe 0-kollisions-frei und „unabhängige Menge“ bemerken. Die Bestimmung von $|S_L|$ ist äquivalent zur Bestimmung der Anzahl der unabhängigen, auch stabil genannt, Mengen eines Graphen, die den mit Null bewerteten Knoten entsprechen.



In dem Fall, daß $s_i + s_j = s + 1$ ist, entfernen wir nur den Knoten $s + 1$ mit seinen beiden Kanten aus dem Graphen und haben damit einen Pfad der Länge s vorliegen. Wie wir aus dem Beweis für die erste Bedingung wissen, besitzt solch ein Graph genau $F(s+2)$ zulässige 0-1-Belegungen. Ist dagegen $s_i + s_j = s - 1$ fügen wir quasi zu einem Kreis mit $s - 1$ Knoten noch einen weiteren Knoten s hinzu, der mit zwei Kanten zu den Knoten $s - s_i$ und $s - s_j$ verbunden ist, also „parallel" zum Knoten der Position 1 eingefügt worden ist. Wir haben dann $F(s-1+1) + F(s-1-1) + 2F(s-1-1)$ mögliche 0-1-Folgen der Länge s, was sich zu $F(s+2) - 2F(s-3)$ umformen läßt, also für $s > 3$ ebenfalls weniger als $F(s+2) - F(s-2)$ Möglichkeiten.

In dem Fall, daß $s_i + s_j < s$ ist, unterteilen wir die gesamte Folge in Stücke der Länge $s_i + s_j$ und eine kürzere Restfolge. Das Anwenden unserer Methode des Vor- und Zurückschreitens auf jedes dieser Stücke soll nun $|S_L| \le F(s \bmod (s_i + s_j) + 2)F(s_i + s_j + 2)^{\lfloor \frac{s}{s_i+s_j} \rfloor}$ ergeben. Um dies einzusehen, müssen wir im wesentlichen nur die zulässigen 0-1-Belegungen der Graphen mit $h + r$ Knoten korrekt abschätzen, wobei $h := s_i + s_j$ und $r := s \bmod (s_i + s_j) > 0$ ist. Der Fall, daß $\lfloor \frac{s}{s_i+s_j} \rfloor \ge 2$ ist, kann auf mehrere unabhängige Kreise abgebildet werden, weil die Formel ebenso $F(s_i + s_j + 2)$ als Faktor enthält, und die Forderung $r \neq 0$ ist dann keine Einschränkung mehr.



Die h Knoten müssen einen zulässigen Kreis bilden, an den die r Knoten in einer oder mehreren „Brücken“ angedockt sind. Für den Beweis werden wir es uns stets leisten, diese Brücken unabhängig zu betrachten. Wie wir uns bereits im Fall $s_i + s_j = s - 1$ überlegten — hier hat die Brücke die Länge Eins — werden wir nun allgemein eine zulässige 0-1-Folge irgendwo an dem zulässigen Kreis der Länge h andocken, so daß der neue Graph zulässig bleibt und die „Parallelität“ erhalten bleibt. Letzteres bedeutet, wenn wir das eine Pfadende mit einem Knoten des Kreises verbinden, muß das andere Ende des Pfades mit einem Knoten des Kreises im Abstand Pfadlänge+1 verbunden werden. Beide Wege, der entlang des Kreises und der entlang der Brücke sind also gleichlang. Dies wird durch den Umstand erzwungen, daß die Knotennummern in der Brücke um $s_i + s_j$ größer sind als ihre korrespondierenden im Kreis.
Nach diesen grundsätzlichen Überlegungen können wir die Anzahl die zulässigen 0-1-Belegungen eines Kreises der Länge h an den eine Brücke der Länge b angedockt ist, zu $F(b)(F(h+1) + F(h-1)) + 2F(b-1)F(h+1) + F(b-2)F(b+2)F(h-b)$ berechnen. Hierbei sind die Brücken jenachdem unterschieden, ob sie an beiden Enden eine 1, nur an einem oder an keinem Ende eine 1 haben. Diese Anzahl läßt sich zu $(F(h+1)+F(h-1))F(b+1)+F(h)F(b-1)+F(b+2)F(h-b)F(b-2)$ umformen. Wenn wir ferner wüßten, daß $2F(h-1)F(b-1) \geq F(b+2)F(h-b)F(b-2)$ gilt, könnten wir die Anzahl direkt zu $\leq (F(h+1) + F(h-1))(F(b+1) + F(b-1))$ abschätzen. Damit wäre die Multiplikativität in der Knotenzahl, sprich der Folgenlänge, hergeleitet. Tatsächlich auftretende Abhängigkeiten zwischen den verschiedenen Brücken der r Knoten könnten die Gesamtzahl der zulässigen 0-1-Belegungen höchstens vermindern und wir hätten den Satz bewiesen. Eine genaue Analyse zeigt nun, daß die Bedingung $2F(h-1)F(b-1) \geq 3F(h-1)F(b-2) \geq F(b+2)F(h-b)F(b-2)$ für alle $b \geq 4$ und $b = 2$ erfüllt ist. Im Fall daß $b = 3$ ist, vereinfacht sich unsere unterstellte Ungleichung zu $2F(h-1) \geq 5F(h-3)$. Dies ist für alle $h > 4$ korrekt. Den Spezialfall $h = 4$ und $b = 3$ können wir explizit abzählen und erhalten $26 < 28 = (F(5) + F(3))(F(4) + F(2))$ zulässige 0-1-Belegungen — weniger als die multiplikative Schranke fordert. Es verbleibt letztlich nur noch der Fall $b = 1$. Hier können wir $(F(h+1) + F(h-1))(F(2) + F(0))$ leider nicht erreichen, aber bleiben immerhin noch unter $2(F(h+1) + F(h-1)) < F(b+1)F(h+2)$, so daß auch in diesem Fall der Satz gilt. $\square$

Lassen wir die Forderung $\mathrm{ggt}(s_i, s_j) = 1$ im 2. Fall fallen, müssen wir die Basis

1.618 nur geringfügig vergrößern und erhalten auf diesem Beweis aufbauend

Satz 27
Ist $|L| \geq 3$ *und* L *symmetrisch, so folgt,*
daß $S_L \leq \sqrt[6]{18}^{\,s} \approx 1.6189^{\,s} < \sqrt[4]{7}^{\,s} \approx 1.6266^{\,s}$ *ist.*

Beweis: Im Fall einer symmetrischen Zugmenge L mit $|L| \geq 3$ gibt es immer ein s_i und $s_j \neq s_i$, so daß $s_i + s_j = s$ ist. Ist nun $\mathrm{ggt}(s_i, s_j) > 1$, so zerfällt der Kreis in Bedingung Zwei in ggt-viele, die alle jeweils $s/\mathrm{ggt}(s_i, s_j)$ Knoten im Graphen, sprich Positionen der 0-1-Folge umfassen. Die Länge h eines solchen Kreises ist aber mindestens 3, da $s_i \neq s_j < s$ gilt. Somit ist die Gesamtzahl $(F(h+2) - F(h-2))^{\mathrm{ggt}(s_i,s_j)}$ oder umgeformt: $\sqrt[h]{F(h+2) - F(h-2)}^{\,s}$. Die Basis dieser Potenz konvergiert mit wachsendem h monoton alternierend gegen $\frac{\sqrt{5}+1}{2}$ und nimmt für $h \geq 3$ ihr Maximum bei 4 an. Da $F(4+2) - F(4-2) = 7$ ist, ergibt sich der Satz 27. Diese Konstante $\sqrt[4]{7}$ läßt sich mit weiteren Überlegungen auf $\sqrt[6]{18}$ drücken. Erstens läßt sich für $|L| \leq 4$ stets $\mathrm{ggt}(s_i, s) = 1$ erreichen oder es gilt $\mathrm{ggt}\, L > 1$. Im letzten Fall haben wir $|S_L| = \mathrm{ggt}(L) S_{\bar{L}}$, wobei $\bar{L}$ die Elemente von L, dividiert durch $g := \mathrm{ggt}\, L$, enthält. Ferner gilt $g \leq x^g$ für alle $g \in \mathbb{N}$ und für alle $x \geq \sqrt[3]{3} < \frac{\sqrt{5}+1}{2}$. Somit können wir von nun an wie im Fall $\mathrm{ggt}(s_i, s) = 1$ zu Ende argumentieren. Es verbleibt der unbeschränkte Fall $|L| \geq 5$. Aufgrund der Überlegung der alternierenden Konvergenz der Zahlen $\sqrt[h]{F(h+2) - F(h-2)}$ können wir nur dann oberhalb von $\frac{\sqrt{5}+1}{2}$ liegen, falls nur Zyklen gerader Länge auftreten. Da wir aber zwei verschiedene $s_i < s/2$ haben — und Zyklen der Länge 2 hier nicht in Betracht kommen — muß in diesem Fall einer von diesen beiden Zügen auf $\mathrm{ggt}(s, s_i) \leq s/6$ führen. Dies heißt es gibt einen geraden Zyklus der Mindestlänge sechs und wir erhalten die etwas kleinere Konstante $\sqrt[6]{18}$. □

Unter Ausnutzung von Zugrelationen mit $s_i + s_j < s$ kann man die Konstante $\sqrt[6]{18}$ noch näher an $\frac{\sqrt{5}+1}{2}$ bringen. Da die absolute Verbesserung aber immer winziger wird und eine scharfe obere Schranke vermutlich unter 1.5 liegen wird, wäre dies für unseren Fall nur von beweistechnischem Interesse.

Für den Fall symmetrischer 4-Zugspiele kann man $|S_L|$ noch explizit bestimmen. Sei $L = (s_1, s_2, 2s_2 - s_1, 2s_2)$ mit $\mathrm{ggt}(s_1, s_2) = 1$ und $c(s_2) := |S_{L_{2s_2}}|$. Dann gilt $c(n+3) = c(n+2) + 3c(n+1) + c(n)$ mit $c(0) = 1, c(1) = 3, c(2) = 5$. Diese lineare Rekursionsgleichung hat die Lösung: $c(n) = (1+\sqrt{2})^n + (1-\sqrt{2})^n - (-1)^n$ für alle $n \in \mathbb{N}_0$ und das Maximum von $\sqrt[2n]{c(n)}$ ist $\sqrt[6]{15}$ bei $n = 3$.

Für den allgemeinen Fall symmetrischer Mengen — hier hat man mit $s := \max L$ und $d := |L| - 1$ einen d-regulären Graphen auf s Knoten vorliegen — möchte ich die folgende Vermutung von T. Sillke erwähnen: $|S_L| \leq (2 \cdot 2^d - 1)^{\frac{s}{2d}}$ wenn $s \geq 2d$. Gleichheit gilt, falls $2d|s$ und der Graph in $s/(2d)$ bipartite Graphen $G_{d,d}$ zerfällt.[26]

Für festes d wird das Maximum bei $s = 2d$ angenommen und mit $d \to \infty$ ließe sich die Basis der oberen Schranke in Satz 27 bis auf $\sqrt{2}$ drücken.

Der Einfluß der Startfolge auf die Periodenlänge

An dieser Stelle sei auf die etwas andere Definition der v_L-Werte durch Althöfer und Bültermann [AB95] eingegangen. Eine Feinheit ist die unterschiedliche Definition des Spielendes: Während ich das Spiel als beendet ansehe, sobald man keinen Zug mehr anwenden kann, dies heißt, man würde bei der Anwendung irgendeines Zuges immer den Definitionsbereich $\mathbb{N}_0$ der Haufenpositionen verlassen, definieren Althöfer und Bültermann explizit $\max L$ aufeinanderfolgende Haufenpositionen aus $\mathbb{Z}$ als verloren für den dort am Zug befindlichen Spieler — „Haufenpositionen ≤ 0 gelten als verloren“. Sie erhalten damit eine Startfolge von $\max L$ Gewinnwerten für die Haufenpositionen $1, 2, \ldots, \max L$ und wenden für alle größeren Haufenpositionen dann eine zu Beobachtung 1 äquivalente Berechnungsvorschrift an. Wie die beiden Autoren selbst bemerken, und in ihrer Definition der Vorperiodenlänge auch berücksichtigen, entspricht dies einer Verschiebung der $\mathcal{P}$-$\mathcal{N}$-Folge um $\max L + 1$ Positionen nach rechts. Als Veranschaulichung der unterschiedlichen Definitionen diene wieder die Folge des Subtraktions-Spieles $(3, 7, 8)$, wobei $w(n)$ den Wert der Haufenposition nach Althöfer und Bültermann angibt:

n	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$v(n)$	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1	1
$w(n)$	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0

Für negative Haufenpositionen ist $v(n)$ ja undefiniert, könnte aber stets zu 1 definiert werden, wobei dann in Beobachtung 1 das Produkt immer über alle Züge aus L laufen könnte, was algorithmisch vorteilhaft ist.

[26] Für $s < 2d$ kann man eine ähnliche Vermutung aufstellen, indem man den Graphen als einen verallgemeinerten k-partiten Graphen mit $k = s/(s-d)$ Knoten vom Grad $d = (k-1)(s-d)$ erkennt, die vermutlich eine obere Schranke von $\frac{s}{s-d}(2^{s-d} - 1) + 1$ für die Anzahl der unabhängigen Mengen ergibt.

Die Definition von Althöfer und Bültermann verhindert eine Verallgemeinerung auf unbeschränkte Zugmengen — die im nächsten Abschnitt betrachtet werden — , führt aber dafür auf die Frage, wie sich die Periodenlängen von 0-1-Folgen in Abhängigkeit von einer beliebigen $\max L$ langen Startfolge verhalten. Letzteres möchte ich an den Rekordhaltern der Periodenlängen der symmetrischen Zugmengen exemplarisch darstellen. In der nachfolgenden Tabelle sind die spaltenweisen Einträge:

- $|L|$,
- Zugmenge L,
- Anzahl der spiegelsymmetrischen 0-kollisions-freien Folgen bezüglich L mit Länge $\max L$,
- $p(L)$,
- Anzahl aller 0-kollisions-freien Folgen bezüglich L mit Länge $\max L$,
- Anzahl der Bahnen[27] der 0-kollisions-freien Folgen und für kleine Bahnanzahlen, in `( )` eingeschlossen, die Länge der einzelnen Bahnen.

Z.B. bedeutet die erste Zeile in dieser Tabelle, daß das 1-Zug-Spiel, was nur aus dem Zug 7 besteht, 16 spiegelsymmetrische 0-kollisions-freie Folgen der Länge 7 enthält, die Periodenlänge der Iteration begonnen mit der reinen 0- oder 1-Folge 14 beträgt und es insgesamt 128 0-kollisions-freie Folgen der Länge 7 gibt, die sich auf 10 Bahnen verteilen, unter denen 9 die Periodenlänge 14 besitzen und eine die Periode 2.

```
1 { 7 }                16     14     128   10(9x14,2)
3 { 2, 5, 7 }           5     22      29    3(22,4,3)
4 { 1, 4, 7, 8 }        3     25      33    3(25,5,3)
5 { 2, 3, 6, 7, 9 }     5     28      37    3(28,5,4)
3 { 3, 7,10 }           5     45     123    5(45,31,17,17,13)

3 { 3, 8,11 }          13     54     199    8(2x54,2x19,2x16,14,7)
5 { 3, 4, 8, 9,12 }     7     54     121    4(54,31,20,16)
4 { 1, 6,11,12 }        7     61     197    7(61,40,23,23,20,17,13)
3 { 4, 9,13 }          21     76     521   13(2x84,76,2x58,42,3x22,
```

[27] Für jede Abbildung $f : D \to D$ und für jedes $x \in D$ kann die Folge der Iterierten $(f^{(i)}(x))_{i=0}^{\infty}$ gebildet werden. Diese heißt auch Bahn von x unter f. Die Zerlegung des Definitionsbereichs einer Abbildung f unter Iteration derselbigen in disjunkte Mengen, bezeichnet man als Bahnen dieser Abbildung. In unserem Fall läßt sich f als Schieberegister auffassen, welches auf einem $\max L$ langen Abschnitt einer 0-kollisions-freien 0-1-Folge operiert. Ist f bijektiv, ist jede Bahn ein (eventuell einelementiger) Zyklus. Dann kann man auch von der Länge einer Bahn, sprich eines Zykluses, reden.

				2x17,11,8)
5 { 3, 5, 9,11,14 }	5	94	339	9(94,48,43,41,3x28,25,4)
3 { 4,11,15 }	34	100	1364	23(2x122,5x100,92,2x70, 46,35,6x26,2x19,13,8)
5 { 3, 8, 9,14,17 }	25	204	1123	16(204,179,150,91,84,3x60, 57,36,31,26,23,2x22)
6 { 2, 6, 9,12,16,18 }	25	238	901	16(238,186,88,84,71,30, 2x28,25,2x24,2x22,15,11,5)
7 { 2, 6, 7,11,12,16,18 }	19	314	721	11(314,74,73,60,44,2x30, 28,24,2x22)
5 { 4, 7,14,17,21 }	58	444	4288	35(722,444,2x260,230,190,159, 2x158,154,142,2x128,...)
7 { 3, 5, 9,13,17,19,22 }	10	494	4810	79(494,329,280,203,170,132, 80,67,55,66x44,41,26,25,4)
7 { 2, 7,10,12,15,20,22 }	37	516	2201	25(516,182,159,154,133,132, 123,109,107,65,3x62,...)
6 { 5,11,12,13,19,24 }	35	689	8481	63(689,678,666,556,414,347, 310,283,262,260,249,...)
5 { 3,11,13,21,24 }	41	767	18719	164(2x848,767,682,619,541, 437,410,402,357,346,...)
9 { 3, 4, 8,10,14,16,20,21,24 }		790	2089	23(790,236,144,2x93,8x48, 3x44,3x40,32,2x28,9)
6 { 1, 3,12,21,23,24 }	33	1107	10337	61(1107,875,782,620,465,...)
5 { 3,12,14,23,26 }	127	1215	37753	123(1835,1551,1448,1403,1346, 1278,2x1215,1025,1017,...)
5 { 3,11,15,23,26 }	63	1256	41707	179(2019,1842,1674,1669,1386, 1261,1256,1214,1209,...)
7 { 5,12,13,14,15,22,27 }	87	1276	16939	68(1276,1065,1022,942,760, 651,596,558,500,483,...)
5 { 2, 8,19,25,27 }	193	1892	56407	171(3140,2673,2284,2236,1987, 1892,1834,1826,1494,...)
9 { 2, 3, 8,13,14,19,24,25,27 }		1937	13834	69(1937,1139,875,803,490, 2x482,458,435,393,...)
5 { 6, 9,19,22,28 }	187	2639	80701	216(4504,2737,2703,2639,...)
6 { 5,13,14,15,23,28 }	71	3424	50573	154(3715,3424,3237,1940,...)
7 { 4, 7,13,15,21,24,28 }	73	3624	28449	85(3624,1937,1544,1436,...)
5 { 1,12,17,28,29 }	303	4896	131603	236(6210,4896,4674,4519,...)
6 { 4,10,15,20,26,30 }	239	5296	78373	196(5296,2937,2928,2592,...)
5 { 6,15,17,26,32 }	427	8530	449413	381(11625,9394,9298,9219, 8587,8530,7413,6508,...)
8 { 2, 5,11,17,23,29,32,34}	153	10419	123761	189(10419,6277,3432,2395,...)
5 { 6,14,21,29,35 }	977	10766	1514671	1026(21397,18755,18618,...)

5	{ 5,12,23,30,35 }	1025	11693	1531839	1250
5	{ 4, 9,28,33,37 }	1513	21945	3465347	1378
5	{ 3,18,20,35,38 }	1459	23340	5285573	1570
5	{ 3,11,27,35,38 }	708	25285	5616954	4075
5	{ 7,17,22,32,39 }	2310	34461	8194164	

Diese Tabelle vermittelt zum einen einen Eindruck, wie gut die Abschätzung aufgrund von Satz 27 auf die Anzahl der 0-kollisions-freien Folgen der Länge $\max L$ anwendbar ist, zum anderen welche Bedeutung die Bahn der 1-Startfolge unter allen möglichen Bahnen hat.[28] Je größer $|L|$ ist, um so wahrscheinlicher scheint es mir zu sein, daß die 1-Startfolge die längste Bahn bildet.

Für beliebige 0-kollisions-freie Startfolgen kann man sogar im Fall $|L| = 3$ superpolynomielle Periodenlängen konstruieren, indem man Familien von Zugmengen $(L_n)_{n\in\mathbb{N}}$ auswählt, deren $\operatorname{ggt} L_n$ eine anwachsende Funktion in n ist. Man wählt einfach $\operatorname{ggt} L_n$ viele 0-1-Folgen der Länge $\max L_n \,/\, \operatorname{ggt} L_n$ mit teilerfremden Periodenlängen. Nun setzt man die benötigte Startfolge als eine alternierende Folge dieser kürzeren Folgen zusammen. Da L unabhängig auf diesen $\operatorname{ggt} L$ vielen Teilfolgen wirkt, erhalten wir als Periodenlänge unserer Startfolge das kleinste gemeinsame Vielfache der Unterperiodenlängen.

Z.B. besitzt das Subtraktions-Spiel $L = (12, 28, 40)$, welches $\operatorname{ggt} L = 4$ hat, 228 886 641 0-kollisions-freie 0-1-Folgen der Länge 40, die in 84 644 Bahnen zerfallen. Von den 29 verschiedenen auftretenden Periodenlängen ist die maximale 1 233 180, die durch die Startfolge 1111101011111111011111101111111111111 erreicht wird. 1 233 180 ist gleich $(45 * 31 * 17 * 13) * 4$, wobei die vier Faktoren in der Klammer von den vier möglichen Periodenlängen des Spieles $(3, 7, 10)$ stammen, die sich im Spiel L jeweils an den Haufenposition mit fester Restklasse modulo 4 widerspiegeln. In diesem Sinne vielversprechend ist z.B. die Familie $L_n = (2n - 1, 2n + 1, 4n)$, die nur Bahnen mit ungerader Länge hat. Für $n = 5$ erreicht man bei zehn Basisperioden für das zusammengesetzte Subtraktions-Spiel mit $\max L = 200$ und geeigneter Startfolge bereits $p(L) = (239*221*199*181*177*163*137*53*47*31)*10 > 2^{0.36 \max L} > 200^9$.

Aber dies habe ich nicht im Sinn, wenn ich nach langen Perioden von Subtraktions-Spielen Ausschau halte.

[28] Man kann die Bahnen auch noch danach klassifizieren, ob diese zwei, eine oder keine spiegelsymmetrische Folge der Länge $\max L$ oder $\max L + 1$ enthalten.

Der Fall $|L| = \infty$

Bei unbeschränkten Zugmengen warten auch interessante zu entdeckende Strukturen. Wählt man z.B. $L = \{1, 4, 9, \ldots, n^2, \ldots\}$[29], L als die Zugmenge die Menge der positiven Fibonacci-Zahlen $F(n)$[30] oder als zulässige Züge $s_i = i!$, das Fakultäts-Subtraktions-Spiel[31], stellen wir fest, daß ihre $\mathcal{P}$-$\mathcal{N}$-Folge nicht periodisch ist. Dies ist eine einfache Folgerung des Satzes:

Satz 28
Kein Zug eines Subtraktionsspieles kann ein Vielfaches der Periodenlänge sein, wenn diese ≥ 2 ist. Oder — für alle $s \in L$ gilt: $p(L) \nmid s$ oder $p(L) = 1$.

Beweis: Entweder $p(L) = 1$ oder es gibt mindestens eine Verlustposition in der Periode. Betrachte eine solche Null in der Periode einer $\mathcal{P}$-$\mathcal{N}$-Folge. Da diese Position mit jeder um Vielfache von $p(L)$ größeren Position keine 0-Kollision bilden darf, darf kein Vielfaches von $p(L)$ in L als Zug auftreten. $\square$

Der Spezialfall $p(L) = 1$ kann nur bei unbeschränkten Zugmengen auftreten.[32] Das einfachste Beispiel mit $p(L) = 1$ ist $L = \mathbb{N}$ — das klassische Nim-Spiel [Bou02].

Folgerung 29
Besteht die Zugmenge aus Potenzen aller natürlichen Zahlen, wie bei dem Quadrat-Wegnehm-Spiel, ist sie die Menge der Fakultäten oder der Fibonacci-Zahlen, dann kann dieses Subtraktions-Spiel nicht periodisch werden.

Beweis: Trivial bis auf den Fall das $L = \{F(i) \,|\, i \in \mathbb{N}\}$ ist. Die Fibonacci-Zahlen bilden ein additives Schieberegister mit höchstens k^2 Zuständen mo-

[29] C.A.B. Smith behauptet in [BC39, p. 38], daß $p(L) = 5$ sei, dies ist aber nur für Haufenpositionen kleiner 25 der Fall. David Bush hat die Folge der v_L bis $4 \cdot 10^7$ berechnet und zahlreiche Kuriositäten in der Verteilung der Gewinn- und Verlustwerte bemerkt. Z.B. ist unter den ersten 180 000 Werten mit $v_L(n) = 0$ und $i \bmod 10 = 6$ einzig $n = 11356$ vorhanden.

[30] Nach Berechnung der ersten 10^8 Positionen wirft T. Sillke die Frage auf, ob die Spargue-Grundy-Werte für das Fibonacci-Subtraktions-Spiel beschränkt bleiben, nachdem erst $v(63331966)$ das erstemal gleich 16 ist.

[31] Hier hat man eine „quasiperiodische" $\mathcal{P}$-$\mathcal{N}$-Folge $(v_L(n))_{n=0}^{\infty}$ =0110111011011101101110111011011101101110110111011011.... der Pseudoperiodenlänge 25.

[32] Dann muß für alle Züge gelten: $s_{k+1} - s_k \leq q(L)$. Mein Betreuer warf daher die interessante Fragestellung auf „Wie schnell dürfen mit steigendem Zugindex k die Zugwerte s_k anwachsen, damit die asymptotische Dichte der Verlustpositionen gleich 0 ist?"

dulo k betrachtet. Wegen $F(0) = 0$ und der eindeutigen „Rückwärtsberechnung“ der Fibonacci-Zahlen muß $F(i) \bmod k = 0$ immer wieder auftreten. Das heißt aber, für jede natürliche Zahl k gibt es ein $s_i = F(i)$ mit $k|s_i$. $\square$

Wählt man $L = \{1, 2, 4, \ldots, 2^n, \ldots\}$ als Zugmenge, so ist $q(L) = 0$ und $p(L) = 3$. Dieses Spiel ist äquivalent zum Spiel $L = \{1, 2\}$, woran man sieht, daß fast alle Züge dieses Exponential-Spieles irrelevant sind. Ein noch einfacheres Beispiel für überflüssige Züge ist das „Ungerade“-Spiel mit $L = \{1, 3, 5, \ldots\}$, welches schon zu $L = \{1\}$ mit $p(L) = 2$ äquivalent ist. [GS56] [BCG82, p. 84, V. 1]

Ist dagegen L die Menge der Primzahlen, so scheint es völlig offen, ob ihre 0-1-Folge periodisch wird oder nicht.[33]

[33] $(v_L(n))_{n=0}^{\infty}$ =00111111100111111111111110111111110011111111111110111110. . . . Gibt es nur fünf Verlustpositionen mit geradem Index?

Konsistenzbeweis für gewisse Subtraktions-Spiel-Beschreibungen*

Um die Konsistenz von SGW-Folgen mit freien Variablen zu beweisen, habe ich eine Grammatik definiert, die diese Nim-Folgen effizient beschreibt. Links in der nachfolgenden tabellarischen Grammatikdefinition steht stets ein `-` oder `+`, gefolgt vom Namen des zu definierenden Objekts, der durch : abgeschlossen ist. Dann folgt seine Definition, die durch eine Leerzeile beendet wird. Das `+` bzw. `-` gibt den *scantype* des Namens an. Dabei bedeutet `-`, daß die Subobjekte dieses Objekts nicht durch `whitechars` — dies sind spezielle durch die Grammatik festgelegte Zeichen wie z.B. die Leerstelle(`SP`), die der Formatierung dienen — getrennt sein dürfen, und `+` bedeutet, daß alle Subobjekte dieses Objekts durch mindestens ein `whitechar` getrennt sein müssen. Man hätte natürlich auch explizit bei den `+` *scantypen* in ihre Definition an allen Stellen `whitechar*` einfügen können, dies hätte aber die Lesbarkeit der Grammatik verringert.

Definition 30
Grammatik† für Subtraktions-Spiele mit freien Parametern :=

Für die Verkettung der Objekte innerhalb der Definition ist relevant, daß

`[ ]`	*Objekte einschließt, die optional vorhanden sein können,*
obj1 obj2	*das hintereinander Auftreten von obj2 nach obj1 bezeichnet — ob dazwischen* `whitechars` *sein müssen oder nicht sein dürfen, hängt vom* scantype *ab,*
,	*ausschließende Alternativen trennt,*
`*`	*für eine endliche Wiederholung seines linken Ausdrucks steht und somit mindestens einmal vorhanden sein muß.*

Diese vier Objekttrenner sind nach fallender Priorität aufgelistet.

Als nicht (wohl) zu definierendes Elementarobjekt wird das Byte angesehen. Es muß mindestens die 78 Zeichen‡ `0`, `1`, `2`, …, `9`, `a`, `b`, `c`, …, `z`, `A`, `B`, `C`, …, `Z`, `_`, `+`, `-`, `(`, `)`, `>`, `<`, `=`, `!`, `SP`, #, `LF`, `CR`, `TAB`, `FF`, `NUL` *darstellen können.*

```
-char:   irgendein Byte außer   NUL,LF,CR,FF
```

*Eine gewisse Vertrautheit mit Sprachgrammatiken und existierenden Zeichenkodierungen auf Rechnern ist zum Verständnis der ersten Seiten dieses Kapitels von Vorteil.

†*Release 0.4* vom Sep. 96

‡Als Referenzcode kann der ISO-Latin-1-Code oder der ASCII-Code dienen.

```
-whitechar:   SP, LF, CR, TAB, FF

-comment:  # [char*] EOL

-digit:  0,1,2,3,4,5,6,7,8,9

-number:  digit*

-variable: a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z

-sign:  +,-

-expression:  [sign] number [ [sign] [number] variable * ] ,
              [sign] [number] variable [ [sign] [number] variable * ]

-value:  _,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
         +,- ,
         _EXTERN_DEFINED_LIST_OF_CHARS_

-wordvalue:  value [ [number] value * ]

-word:  (wordvalue)variable , (wordvalue)[number] , (wordvalue)(expression) ,
        <wordvalue>variable , <wordvalue>[number] , <wordvalue>(expression) ’
             value variable ,      value [number] ,       value(expression)

-keytoken:   FOR, FROM, UPTO, REPEAT, REP, ENDREP, PER,
             IF, THEN, ELIF, ELSE, FI, ENDIF, LET

-orderrelation:  >= , <=

-relation:  >= , <= , == , !=

+letstatement:  LET variable=expression

+wordblock:  [comment*] [letstatement*] word *

+loophead:  [comment*] FOR variable FROM expression UPTO expression

+conditon:  expression relation expression

+branchstatement:  [comment*] IF condition THEN [branchstatement*] wordblock
                   [ ELIF condition THEN [branchstatement*] wordblock *]
```

```
                    [ ELSE wordblock ] FI

+unit:  wordblock [unit*] , [loophead] REPEAT [unit*] ENDREP , branchstatement

+loop:  [loophead] REPEAT [unit*] ENDREP

+orderconditon:  variable orderrelation expression

+movelist:  { expression* }

+moves: [ordercondition*] movelist

+description: moves unit*
```

Das `_EXTERN_DEFINED_LIST_OF_CHARS_` *ist eine Alternative von* `char`*-Objekten die der Benutzer als mögliche Wortwerte definieren kann. Diese darf natürlich weder die Zeichen* `#`*, noch* `(`*,* `)`*,* `<`*, oder* `>` *enthalten.* `EOL` *ist ein "Environment"-spezifisches Zeichen, welches das logische Zeilenende markiert, typischerweise* `LF` *oder* `CR`*.* □ [34]

Faustregel zur Verwendung von `whitechars`:
Außer in `keytokens`, `relations`, `expressions` oder `words` dürfen `whitechars` beliebig auftreten, und haben, abgesehen von der möglichen Beendigung eines Kommentars, keinerlei syntaktische Bedeutung.
Bei der Definition von `word` wurde die `number` als optionales Subobjekt definiert, um eine „Benutzergewohnheit" zu befriedigen, indem eine fehlende `number` mit `1` gleichgesetzt wird.
Die Schlüsselworte `REPEAT` bzw. `REP` und `ENDREP` bzw. `PER` haben identische semantische Bedeutung und ihre alternative Verwendungsmöglichkeit, genauso wie die von `ENDIF` bzw. `FI`, ist nur zur Bequemlichkeit des Benutzers vorhanden.
Die „nichtdarstellbaren" Zeichen, `LF, CR, TAB, FF`, dienen in Form von `EOL` als Kommentarbeendigung und ansonsten nur dem Benutzer zur logischen Gliederung der Grammatik in Form von Spalten(`TAB`), Zeilen(`LF,CR`) und Seiten(`FF`).

Notwendige Voraussetzungen der Darstellbarkeit der Struktur eines Subtraktions-Spiels mittels einer solchen Beschreibung sind natürlich deren Endlichkeit, sowie die Abhängigkeit aller auftretenden Blocklängen in der

[34] Das Ende einer `description` darf man explizit durch `NUL` kennzeichnen, welches nicht mehr als zur `description` gehörig zählt.

Beschreibung von den freien Variablen, wie wir noch sehen werden.

Wie man erkennt, gehört syntaktisch zu einer vollständigen Beschreibung (`description`) eines Subtraktions-Spieles nicht nur die eigentliche Struktur, die durch `unit*` bezeichnet ist, sondern stets auch die Züge(`moves`). Die dort vorhandenen `orderconditions` müssen für jede freie Variable in der Zugmenge(`movelist`) mindestens eine `>=` Relation beinhalten. Eine `unit` stellt entweder einen reinen `wordblock`, eine Schleife oder ein `branchstatement` dar.[35]

Der Begriff des Blocks(`wordblock`) ist dabei grundlegend. Wie auf Seite 16 eingeführt, ist ein Block eine maximale, endliche Anzahl von aufeinanderfolgenden Worten, die durch `REP`- oder `PER` -`keytoken` eingegrenzt sind. Diese Blöcke werden entweder als Schleifenblöcke bezeichnet, diese fangen mit `REP` an, oder sind Nichtschleifenblöcke, wenn sie mit `PER` anfangen. Letztere dürfen auch die Länge 0 haben, sprich leer sein. Die in der Beschreibung angegebenen Blöcke sind in der Reihenfolge ihres Auftretens mit den natürlichen Zahlen durchnumeriert. Ferner wird für den Algorithmus ein nullter Block benötigt, der aber nicht in der Beschreibung mit angegeben wird. Dieser nullte Block besteht per Definition nur aus identischen Gewinnwerten, zweckmäßigerweise aus einem Gewinnwort der Länge $\max L$, und bildet im Sinn der Grammatiksematik den Schleifenrumpf einer endlosen Schleife vor der Haufenposition 0, die mit `ENDREP` enden würde. Dadurch kann die $s \leq n$ Bedingung in der Definition 7 entfallen, und es wird eine äquivalente Definition realisiert, bei der der SGW auf $\max L$ für negative Haufengrößen gesetzt wird. An dieser Stelle ließe sich der Algorithmus auch auf anders definierte nullte Blöcke verallgemeinern — diese können als Startfolge interpretiert werden —, was schon durch eine Änderung des Programmes in der Funktion `prepend_infinity_block` geleistet würde.[36]

Schleifen sind stets durch einen, eventuell leeren, Nichtschleifenblock voneinander getrennt. Äußere Schleifen werden dabei als aus inneren Blöcken bestehend aufgefaßt. In diesem Sinn besteht z.B. die Beschreibung in Beobachtung 18 aus 4 Blöcken, wobei der dritte Block, der die beiden Schleifen

[35] Für das weitere Verständnis der Beweisskizze ist es nicht nötig, sich mit dem `branchstatement` auseinanderzusetzen, zumal es für die in der Arbeit geführten Beweise nicht benötigt wird. Nur soviel: Es wird als aus $x + 1$ Anweisungen bestehend angesehen, wobei x die Anzahl seiner beinhalteten `THEN`-Anweisungen ist und eine die eventuell leere `ELSE`-Anweisung ist. Eine Anweisung umfaßt mindestens einen Block, kann aber auch in viele Blöcke zerfallen, die Schleifen und weitere `branchstatements` beinhalten.

[36] In der Beschreibung eines Subtraktions-Spieles könnte dies durch einen optionalen Block geschehen, der durch ein neues Schlüsselwort gekennzeichnet ist, und dann beispielsweise durch `FROM_INFINITY_DOWN REPEAT` … `ENDREP` beschrieben wird.

trennt, leer ist. Oder die Beschreibung in Satz 8 sieht vereinfacht wie folgt aus:

```
n >= 2 { 2n 4n-1 4n+1 4n+5 8n-2 }
1. Block  der Länge 0
REPEAT
   2. Block  der Länge 24n+2
   FOR i ....  REP
      3. Block  der Länge 0
      FOR j .... REP
         4. Block  der Länge 14n+1
      PER
         5. Block  der Länge 0
      FOR j .... REP
         6. Block  der Länge 14n+1
      PER
      7. Block  der Länge 24n+2
   PER
   8. Block  der Länge 0
ENDREP
```

Anmerkung: In meiner schriftlichen Ausführung zur Beschreibung in Satz 8 hatte ich für den Leser nur die nichtleeren Blöcke als solche bezeichnet, um ihn nicht zu verwirren.

Auch die von Althöfer und Bültermann [AB95] angegebene Struktur ihres Subtraktions-Spieles konnte mit meinem Algorithmus, sprich Programm, verifiziert werden. Eine leichte Transformation ihrer Beschreibung in meine Syntax ergibt diese grob vereinfachte Beschreibung:

```
n >= 2  { n 4n 12n+1 16n+1 }   # Zugmenge
1. Block Länge 0
REPEAT
   2. Block   Länge 40n+1  #  Block 1
   FOR  j FROM 2 UPTO n-1 REP
      3. Block  Länge 28n+1  #  Subblock 2(j)
   PER
   4. Block  Länge 0
   FOR  j FROM 0 UPTO n-2 REP
      5. Block  Länge 33n+2  #  Subblock 3a(j)
      FOR  i FROM 1 UPTO j  REP
         6. Block  Länge 28n+1  #  Subsubblock 3b(j,i)
```

```
    PER
    7. Block  Länge 20n  #  Subblock 3c(j)
    FOR  i FROM j+2 UPTO n-1  REP
       8. Block  Länge 28n+1  #  Subsubblock 3d(j,i)
    PER
    9. Block  Länge 0
 PER
 10. Block  Länge 33n+2  #  Block 4
 FOR  j FROM 1 UPTO n-2  REP
    11. Block  Länge 28n+1  #  Subblock 5(j)
 PER
 12. Block  Länge 73n+3  #  Block 6
 FOR  j FROM 1 UPTO n-2  REP
    13. Block  Länge 28n+1  #  Subblock 7(j)
 PER
 14. Block  Länge 32n  #  Block 8
 FOR  j FROM 1 UPTO n-2  REP
    15. Block  Länge 0
    FOR i FROM 0 UPTO j-2  REP
       16. Block  Länge 28n+2  #  Subsubblock 9a(j,i)
    PER
    17. Block  Länge 49n+3  #  Subblock 9b(j)
                             #  Subblock 9c(j,i=1)
    FOR  i FROM 2 UPTO n-j-1  REP
       18. Block  Länge 28n+1  #  Subsubblock 9c(j,i)
    PER
    19. Block  Länge 32n-j  #  Subblock 9d(j)
 PER
 20. Block  Länge 0
 FOR  j FROM 1 UPTO n-2  REP
    21. Block  Länge 28n+2  #  Subblock 10(j)
 PER
 22. Block  Länge 60n+4  #  Block 11
 FOR  j FROM 1 UPTO n-2 REP
    23. Block  Länge 28n+2  #  Subblock 12(j)
 PER
 24. Block  Länge 39n+3  #  Block 13
ENDREP
```

Die Althöfer/Bültermann'sche Subblock- und Blockbezeichnungen sind als Kommentar zur Orientierung mit aufgeführt. Die Fallunterscheidung in ihrem `Block 9c(j)` ist durch Extraktion ihrer Iteration `i=1` in den vorangehenden Block aufgelöst worden. Ihre Fallunterscheidung für `i=2` ist sogar völlig unnötig.[37] In dieser neuen Form der Beschreibung konnte mein Programm für alle $n \geq 2$ ihre Struktur verifizieren. Um sicherzustellen, daß keine Unterperioden auftreten, schaue man sich die längsten Folgen von $\mathcal{P}$-Worten an. Diese haben die Länge $4n + 1$ und treten nur im 2., 5. und 10. Block auf. Es folgt ihnen stets ein nichtleeres $\mathcal{N}$-Wort und ein weiteres $\mathcal{P}$-Wort. Aber nur im 2. Block hat dieses die Länge $4n$. Somit kann keine echte Unterperiode vorliegen, und die Periodenlänge beträgt wie vorausgesagt $56n^3+52n^2+9n+1$ für alle n. Der Beweis der behaupteten Vorperiodenlänge ist trivial.

Man muß jetzt „nur" für alle Worte eines jeden Blockes verfolgen, auf welche Position es jeder der gegebenen Züge aus der Zugmenge führt. Da die Züge lineare Funktionen in den freien Parametern der Zugmengen sind, wird die Ausgangsposition innerhalb der Beschreibung eines Wortes um die Größe des jeweiligen Zuges verringert. Dies führt für gewisse Worte unter Ausführung bestimmter Züge aus ihrem zugehörigen Block. Das erste Wort eines Blockes wird bei Betrachtung des größten Zuges natürlich am weitesten zurück geworfen. Daher müssen alle Blockübergänge zu unmittelbar angrenzenden Vorgängerblöcken betrachtet werden, sprich durch einen Algorithmus erzeugt werden, bis der gesamte Übergang mindestens die Länge des Ausgangsblockes plus Länge des größten Zuges hat. In meinem Programm leistet dies die rekursive Funktion `prepend_further_block`. Um dies logisch einfach zu handhaben, wurde in der 3. Voraussetzung des Beweises zu Satz 8 gefordert, daß die Länge jedes Blockes von allen freien Variablen des Subtraktions-Spieles abhängig sein muß. Dies wird nun generell für nichtleere Blöcke gefordert. Bei diesem Rückwärtsaneinanderfügen treten fünf Arten von Fällen auf:

a) Der aktuelle Block ist ein Schleifenblock und die Analyse wird mit dem Block fortgesetzt, der mit zu dem Schleifenblock-`REP` korrespondierendem `PER` endet.

b) Der aktuelle Schleifenblock wird durch den Schleifenkopf hinaus verlassen und der vorausgehende Block als nächster bearbeitet.

c) Nach dem aktuellen Nichtschleifenblock, der an eine vorherige Schleife grenzt, wird jene durch ihr Schleifenende hindurch betreten und die

[37] In diesem Zusammenhang kann man sich natürlich auch nach der kürzesten Beschreibung bei vorgegebener Grammatik fragen, aber dies führt weit über den Rahmen dieser Arbeit hinaus. Tatsächlich war es aber die Motivation für die Einführung des `branchstatements`.

Analyse mit derem letzten Block fortgesetzt.

d) Nach dem aktuellen Nichtschleifenblock wird die unmittelbar vorausgehende Schleife übersprungen. Dies bedeutet, die Analyse wird mit dem ihr vorausgehenden Block fortgesetzt.

e) Der aktuelle Nichtschleifenblock grenzt an den nullten Block.

Ausgenommen im letzten, einfacsten Fall(e), müssen bei den anderen vier möglichen Übergängen(a-d), die Werte der jeweiligen Schleifenvariablen berücksichtigt werden. Die sich daraus ergebenden logischen Bedingungen sind entweder:

a) Die aktuelle Schleifenvariable muß für alle weiteren Untersuchungen ihres gesamten Schleifenrumpfes, der eventuell viele Blöcke umfaßt, durch die um eins verminderte Variable ersetzt werden.

b) Die aktuelle Schleifenvariable wird **im nachhinein** durch die Startexpression der Schleife ersetzt werden (oder die Gleichheit dieser Schleifenvariable mit der Startexpression müßte erfüllt sein).

c) Die neue Schleifenvariable wird durch ihre Endexpression ersetzt. Ferner muß die Startexpression dieser Schleife nun $\leq$ ihrer Endexpression sein und die Startexpression soll ≥ 0 sein.

d) Die Startexpression der vorausgehenden Schleife muß $>$ ihrer Endexpression sein.

Insgesamt gesehen gibt es somit zu jedem Übergang ein System von linearen Ungleichungen — die Start- und Endexpressions sind ja affine Linearformen —, die erfüllt sein müssen, um den Übergang zu ermöglichen. Das Ausgangs-Ungleichungssystem wird dabei erzeugt, in dem jeder mögliche Block einer Beschreibung als Ausgangssituation betrachtet wird. Dazu werden für ihn für jede Schleifenverschachtelungsebene, der er angehört, drei Bedingungen generiert. Die Schleifenvariable der entsprechenden Verschachtelungsebene muß erstens $\geq$ der zugehörigen Startexpression sein, zweitens muß sie $\leq$ der zugehörigen Endexpression sein und drittens soll diese Startexpression ≥ 0 sein. Die letzte Bedingung ist eingeführt worden, da sie keine wesentliche Einschränkung darstellt, und das resultierende Ungleichungssystem mittels des Simplex-Algorithmusses [Kre69] auf Zulässigkeit hin überprüft werden kann, ohne vorzeichen-unbeschränkte Variablen einführen zu müssen. Dies war auch der Grund im Fall b) eine Variablensubstitution durchzuführen anstelle der Einführung von Gleichungen für den Simplex-Algorithmus. Im nachhinein ist debattierbar, ob man nicht doch eine Simplex-Algorithmus-Version mit Gleichungen zuläßt, was zwar zur Laufzeiterhöhung führen wird, aber eine elegantere und kürzere Implementation ohne Variablensubstitutionen zulassen würde. Z.B. lautet das Ausgangs-Ungleichungssystem des 6. Blockes in der Beschreibung zu Satz 8:

```
n >= 1            1-n    <= 0
1 >= 0
i >= 1            1-i    <= 0
i <= n-1          1-n+i  <= 0
0 >= 0
j >= 0            0-j    <= 0
j <= i-1          1+j-i  <= 0
```

Die erste Zeile trägt der globalen Ungleichung der freien Variablen `n` Rechnung, die nächsten 3 Ungleichungen stammen von der `i`-Schleife und die letzten 3 Ungleichungen von der den 6. Block steuernden `j`-Schleife. Hierbei ist für jede nichttriviale Bedingung, diese hängt von keiner Variablen ab, noch rechts daneben die normierte affine Darstellung angegeben, wie sie in der Implementation des Algorithmuses realisiert ist. Zu solchen Ausgangsungleichungen jedes „aktivierten“ Blockes kommen beim Eintreten der Fälle c) und d) natürlich noch weitere Ungleichungen hinzu. [38]

Hat man nun für alle Blöcke der Beschreibung genügend lange Folgen von Worten mit zugehörigem Ungleichungssystem erzeugt, müssen diese im Sinne der Zug-Rückverfolgung auf Konsistenz geprüft werden. Dazu werden alle Worte einer solchen Übergangsfolge mit Positionen relativ zum Anfang der gesamten Folge versehen. Dies entspricht genau der Zuordnung der Summe der vom Folgenanfang aufaddierten einzelnen Wortlängen bis zum aktuell zu untersuchenden Wort. Daher kann man diese Relativpositionen eines Wortes stets als affine Linearformen in allen Variablen auffassen. Beim Rückverfolgen eines möglichen Zuges, daß heißt beim Betrachten der Relativposition, auf die das betrachtete Wort abgebildet wird, muß unter den Relativpositionen innerhalb der Übergangsfolge gesucht werden. Logisch läuft dies einfach auf einen Größer-Kleiner-Vergleich von affinen Linearformen hinaus, wobei die Gültigkeit des globalen Ungleichungssystems unterstellt wird. Dabei kann es ohne weiteres geschehen, daß die Ordnungsrelation solcher Relativpositionen nicht eindeutig ist. Tritt dies ein, müssen die beiden möglichen Fälle

[38] Eine auftretende $Start > End$-Bedingung wird zu `-Start+End+1 <= 0` umgeformt.

logisch getrennt untersucht werden, was algorithmentechnisch eine „Verrenkung“ ist, nach dem Motto „das hätte man besser vorher gewußt“, und auf Programmebene einem Exception-Handling mittels `setjmp()` entspricht. Kennt man nun die Position, auf die ein Wort durch einen Zug geführt wird, sind meistens weitere Zerlegungen dieses Wortes nötig, da es oft auf Teile mehrerer Worte abgebildet wird. Die Teile dieser letzten notwendigen logischen Aufsplittung habe ich kurz als Splits eines Wortes bezeichnet. Die Mengen der Splits für ein Wort für jeden der möglichen Züge sind typischerweise nicht untereinander synchronisiert. Das heißt ihre Positionen relativ zum untersuchten Wort vermindert um die Größe des betrachteten Zuges ist nicht aufeinander abgestimmt. Um diese synchron auszurichten, müssen die Splits verfeinert werden. Dies bedeutet, daß ihre „gröbste gemeinsame Verfeinerung“ erzeugt wird. Nun endlich kann für jeden einzelnen Split zusammen mit den synchron korrespondierenden, daß heißt ausgerichteten Splits der anderen Züge, der „Sollwert“ des Wortes an der ursprünglichen Position überprüft werden.

Bei Verallgemeinerung von Worten mit konstantem SGW auf Worte mit verschiedenen SGW treten neue Probleme auf. Ich möchte nur kurz die beiden wesentlichen zusätzlichen Probleme bei der logischen und algorithmischen Beweisführung erwähnen. Als Wert eines Wortes kann nun nicht ein SGW verwendet werden, sondern es wird eine endliche Folge — kurz Zyklus genannt — als Wortwert bezeichnet. Es muß daher bei der Relativpositions-Berechnung auch jedesmal die Position innerhalb einer solchen endlichen Folge berücksichtigt werden. Glücklicherweise hatte ich dieses Problem bald in Form einer zyklischen Folge erfaßt. Diese Interpretation ist auch in der `<DE>` Struktur des nach Satz 22 aufgeführten Subtraktions-Spieles als grundlegend erkennbar.[39] Danach ist dann „nur noch“ ein weiterer Exception-Handler zu realisieren, der beim Auftreten der Situation aktiviert wird, daß die Teilbarkeit einer affinen Linearform durch eine feste gegebene Zykluslänge k nicht entscheidbar ist. Logisch bedeutet dies, daß eine Variable x durch $kx + r$ zu substituieren ist und für alle k möglichen Werte von r separate Fälle zu untersuchen sind, zumindest für das die Fallunterscheidung auslösende Wort.

Das Programm `verify` kann auch optional sämtliche untersuchten Fälle tabellarisch ausgeben. Wer dem Programm nicht traut, kann daher jederzeit das einzelne Subtraktions-Spiel an Hand der Liste aller generierten Übergänge, Wortrückverfolgungen und deren Splits diese Liste nochmals auf Vollständig-

[39] Jetzt war es auch nötig die Gomory-Variante [Kre69, p. 238–240] des Simplex-Algorithmuses zu implementieren, da zulässige Bereiche manchmal nur nichtganzzahlige Werte enthielten und auf Widersprüche in der Beschreibung führten.

keit und Korrektheit hin überprüfen. Dazu sind lediglich ca. 1500 Split-Wort-Verifizierungen nötig, die für die Beschreibung in Satz 8 erzeugt werden oder gar nur 3730 Einzeltests für das Subtraktions-Spiel von Althöfer und Bültermann (Sie finden sicher dabei auch eine effizientere Zusammenfassung). Ein moderner Computer schafft dies mit `verify` in weniger als 10 Sekunden. `:>`

Offene Fragen — Ausblick

Durch das Studieren der Bahnen von `NAND`-Schieberegistern, $v(i) = \underset{j \in L}{\mathrm{NAND}}\ v(i-j)$, kann man wahrscheinlich tiefergehende Resultate über die Struktur der $\mathcal{P}$-$\mathcal{N}$-Folgen von Subtraktions-Spielen L erhalten. Dabei sollte man entweder beliebige 0-kollisions-freie Startfolgen oder gleich den gesamten $\{0,1\}^{\max L}$ betrachten und die dann erreichbaren Periodenlängen untersuchen. Zum Beispiel könnte man Antworten auf Fragen erhalten wie:

- Wie häufig sind Bahnen, die keine spiegelsymmetrischen Teilfolgen der Länge $\max L$ oder $\max L+1$ für eine gegebene symmetrische Zugmenge L enthalten?
- Wann ist die Bahnlänge der Bahn, die eine Teilfolge von $\max L$ aufeinanderfolgenden $\mathcal{P}$-Positionen enthält, größer als $(1+\varepsilon)^{\max L}$ für irgendein $\varepsilon \in \mathbb{R}^+$?

Dies würde Subtraktions-Spiele mit exponentieller Periodenlänge charakterisieren. Natürlich wäre es schon interessant allein für die klassische $\mathcal{P}$-Startfolge,

- die Existenz von 5-Zug-Familien mit exponentieller Periodenlänge zu beweisen,
- das Wachstum der Periodenlängen bzw. Vorperiodenlängen von 4-Zug-Familien zumindest dem Polynomgrad nach zu erfassen,
- Kandidaten für Familien von superpolynominellem Wachstum in der Periodenlänge bei den 4-Zug-Subtraktions-Spielen explizit anzugeben.

Und für die 3-Zug-Subtraktions-Spiele steht

- sowohl die allgemeine Strukturanalyse aus,
- als auch eine explizite Formel für die Periodenlänge und Vorperiodenlänge.

Man sollte sich bewußt machen, daß alle Beweise über Periodenlängen bisher stets eine Strukturanalyse der $\mathcal{P}$-$\mathcal{N}$-Folge voraussetzen und keinerlei nichttriviale zahlentheoretische Sätze für allgemeine n-Zug-Subtraktions-Spiele mit $n > 3$ über die Periodenlänge greifbar scheinen.

Ein noch breiteres Untersuchungsgebiet wird eröffnet, wenn man Züge aus $\mathbb{R}^+$ zuläßt.[40] Dabei könnte man jetzt auch unendlich viele Züge zulassen, aber dennnoch $\max L$ beschränken.[41] Es stellen sich elementare Fragen wie:

- Für welche beschränkten Zugmengen ist die Rückwärtsberechnung ihrer $\mathcal{P}$-$\mathcal{N}$-Folge eindeutig?
- Wann treten Perioden für irrationale Zugwerte auf?

Der Beweis auf Grundlage von $\max L$ und dem ursprünglich diskreten Wert n der Haufengröße ist ja überhaupt nicht anwendbar, aber dennoch bleibt für irrationale Zugmengen die Aussage von Satz 4 vollständig korrekt. Auch stellen sich viele neuartige Fragen:

- Wie kann man für nichttriviale irrationale Zugmengen ihre $\mathcal{P}$-$\mathcal{N}$-Folge in $\mathbb{R}^+$ effektiv berechnen?
- Können bei endlichen Zugmengen beliebig kleine Verlustintervalle auftreten?
- Wann oder wie schnell kann die $\mathcal{P}$-$\mathcal{N}$-Folge zerstäuben?
- Was heißt „lange“ Periode, wenn die $\mathcal{P}$-$\mathcal{N}$-Folge einer beschränkten irrationalen Zugmenge periodisch wird?

Ein Verfahren der Intervall-Vereinigung zur Berechnung der $(v_L(r))_{r=0}^{\infty}$ bietet sich hier an, bei dem man zu jedem neu erkannten Verlustintervall, angefangen bei $[0, \inf L[$, die Vereinigung der zugehörigen Gewinnintervalle bildet, damit das nächste Verlustintervall gefunden hat und so iterativ die Menge $\mathbb{R}^+$ von unten ausschöpft. Folgender Algorithmus präzisiert dies:

für alle $r \in \mathbb{R}_0^+$ $\quad v(r) \leftarrow 0$

$b \leftarrow 0$

`REPEAT`

[40] Züge ≤ 0 zuzulassen, macht dagegen keinen Sinn, da ein solcher Zug stets möglich wäre und daher dann alle Haufenpositionen als unentschieden zu bewerten wären.

[41] Die Endlichkeit der Subtraktions-Spiele wird durch die zusätzliche Bedingung $\inf L > 0$ gesichert — diese ist hinreichend, aber nicht notwendig wie man im Fall $L = \mathbb{R}^+ \setminus \mathbb{Q}$ sieht.

$a \leftarrow \inf\{r \geq b \mid v(r) = 0\}$

$b \leftarrow \min\{\ a + \inf L\ ,\ \inf\{r \geq a \mid v(r) = 1\}\ \}$

für alle $s \in L$ und für alle $r \in [a+s, b+s[\qquad v(r) \leftarrow 1$

`ENDREP`

Ob die Intervallgrenzen der jeweiligen Gewinnintervalle im letzten Schritt mitberücksichtigt werden müssen, hängt davon ab, ob der Wert von a bzw. b nur ein Infimum oder auch Minimum ist, sprich angenommen wird. Ferner erkennt man, daß Gewinnintervalle verschmelzen, wenn für zwei verschiedene Züge $s_i, s_j \in L$ gilt $|s_i - s_j| \leq b - a$. Daher bereiten auch Zugmengen, die Teilfolgen dichter Züge enthalten, keine prinzipiellen Schwierigkeiten, ihre $\mathcal{P}$-$\mathcal{N}$-Folge zu berechnen.

Erst beim Überarbeiten dieses Kapitels „Offene Fragen — Ausblick“ wurde mir bewußt, welche Perspektiven dieser Ansatz nach sich zieht. Man hat jetzt ein analytisches Mittel zur Hand, um kleine Epsilon-Variationen an den Zugwerten durchzuführen und die Auswirkungen auf die Periodenlänge zu untersuchen. Somit könnte man nun lokal erkennen, ob eine Variation in einem Zugwert eine Unstetigkeit,[42] eine polynomielle Änderung oder eine andere funktionale Abhängigkeit in der Periodenlänge nach sich zieht.

Ferner wird nun klar, daß die Grenzen der fraktalen Gebiete der normierten 3-Zug-Subtraktions-Spiele, siehe z.B. Bild v, auf Teilmengen liegen, die durch Linearformen mit rationalen Koeffizienten beschrieben werden, und daß innerhalb von „größeren“ Gebieten desselben Typs sicherlich keine weiteren Grenzen auftreten. Für die 4-Zug-Subtraktions-Spiele gilt im Prinzip die gleiche Aussage, nur muß der Begriff des Gebietes flexibler formuliert werden, um gegen den „Einbrucheffekt“ robust zu sein. Als Illustration folgt daher ein Bild der Gebiete der Periodenlängen der 4-Zug-Subtraktions-Spiele im Unterraum $s_1 + s_3 = s_4$. Nach dem eben gesagten ist dies eigentlich nur ein zweidimensionaler Schnitt — sprich Ebene — , da man irgendeinen Zug zum Skalieren aller Elemente der Zugmenge benutzen könnte. Und in der Tat gleichen sich auch die Bilder für z. B. verschiedene Werte s_3, wenn man an den Achsen nur die Quotienten anderen Züge mit s_3 aufträgt.

Hier ist für eine feste Primzahl s_3, an der horizontalen Achse s_1 von rechts nach links linear anwachsend und an der vertikalen Achse s_2 von unten nach oben linear ansteigend, die Periodenlänge als Grauwert aufgetragen.

[42] In diesem Fall treten neue Gewinn- oder Verlustintervalle in der $\mathcal{P}$-$\mathcal{N}$-Folge auf oder vorhandene Intervalle verschwinden.

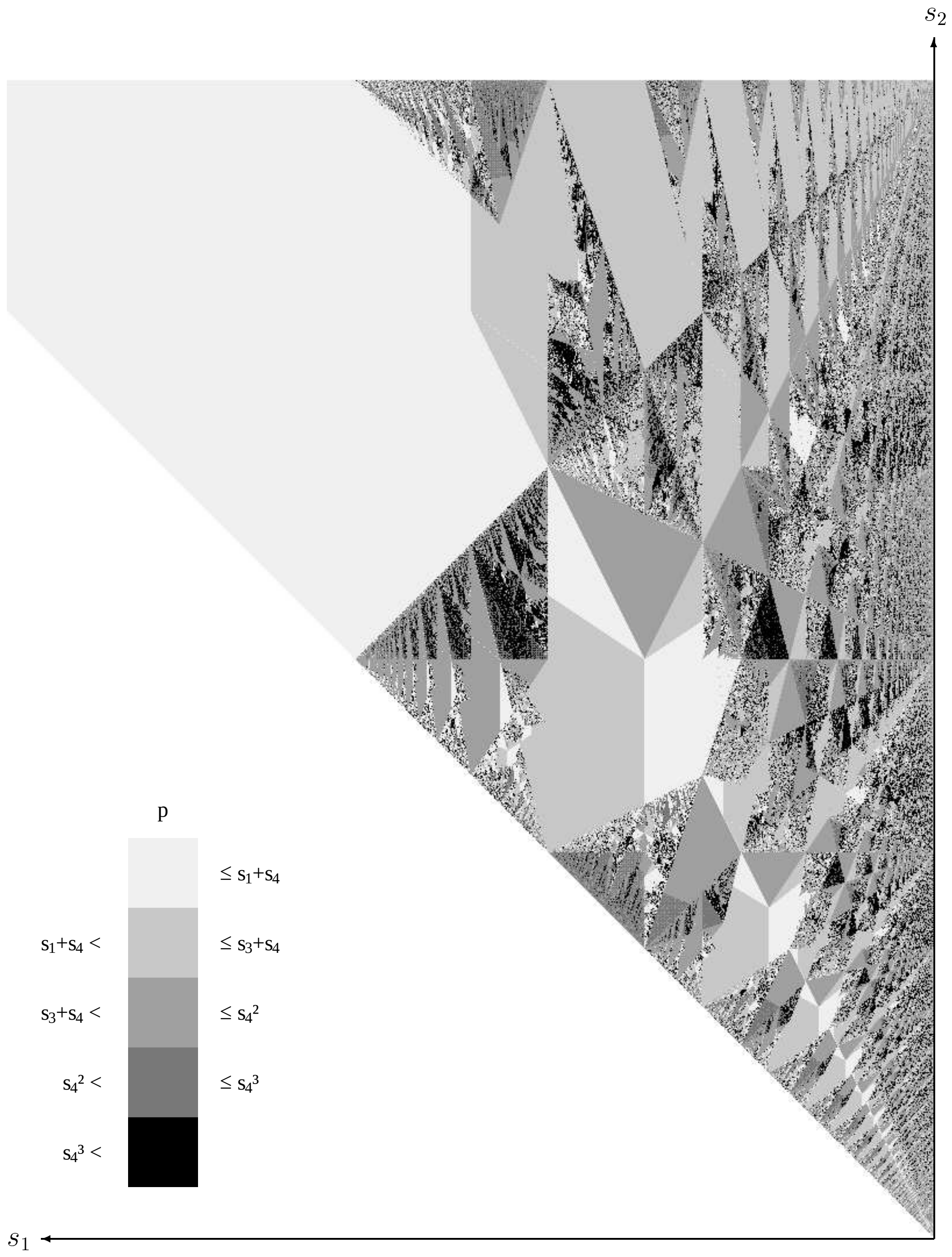


Bild xiii $s_3 = 997, \; s_1 + s_3 = s4$ Periodenlängen der 4-Zug-Subtraktions-Spiele in die Gebiete mit $p \leq s_1 + s_4$, $s_1 + s_4 < p \leq s_3 + s_4$, $s_3 + s_4 < p \leq s_4^2$, $s_4^2 < p \leq s_4^3$ und $p > s_4^3$ eingeteilt und durch fünf verschiedene, in der Helligkeit abfallende, Grauwerte dargestellt.

Für die mehr nicht notwendig anwendungsorientiert interessierten Leser kann man die `NAND`-Verknüpfung auch durch andere boolwertige Funktionen ersetzen und die sich dann ergebenden 0-1-Folgen untersuchen. Als nächstes[43] bietet sich die `NOR`-Verknüpfung an. Diese läßt sich sogar leicht spieltheoretisch interpretieren. Hier handelt es sich einfach um die misèré-Version des normalen Subtraktions-Spieles — sprich wer als erster nicht mehr ziehen kann, hat gewonnen — und die hat als Gewinn/Verlust-Folge einfach die komplementäre des normalen Subtraktions-Spieles. Welche elementare symmetrische Funktion kann man denn noch betrachten? Für die Elektrotechniker und Logiker ist dies sicherlich die Exklusiv-Oder-Verknüpfung, die auch als `XOR`-Funktion bezeichnet wird. Allerdings ist die Definition dieser Funktion für den Theoretiker — sprich Mathematiker — die einer „es gibt genau eine Variable die wahr ist"-Verknüpfung und für den Praktiker — sprich Ingenieur — die einer Paritätsfunktion. Im letzten Fall entspricht dies mathematisch einer Addition modulo 2. Für solche Schieberegister über endlichen Körpern gibt es bereits eine schön entwickelte Theorie. Insbesondere läßt sich der Gewinn- oder Verlustwert einer Haufenposition $v_L(n)$ in $\Theta(\max L^2 \ln n)$ Schritten [LN83, p. 402–411] bestimmen und man ist nicht auf eine Brute-Force-Methode mit Aufwand proportional zu n angewiesen. Im Falle der Entweder-Oder-Interpretation der `XOR`-Funktion scheinen die Schwierigkeiten der Analyse solcher Schieberegister, der der `NAND`-Funktion ebenbürtig. Eine etwas weitergehende Verallgemeinerung von monotonen, symmetrischen Schieberegisterfunktionen [Hen96] — die Anzahl der Verlustpositionen auf die man ziehen könnte wird bestimmt, und die aktuelle Position gilt genau dann als gewonnen, wenn diese Anzahl größer einer fest zum Spiel vorgegebenen Konstante ist — wird in einer von Althöfer vergebenen Diplomarbeit untersucht.

[43] `UND`- bzw. `ODER`-Verknüpfungen scheiden offenbar als interessante Funktionen aus.

Programm-Anhang

Der Anhang gliedert sich in drei Teile. Einer Programmdokumentation von `verify.c`, dem eigentlichen Quellcode und einem Beispieldatensatz, welcher die Beschreibung für das Subtraktions-Spiel $L = (n, 2n + 1, 4n + 2, 5n + 3, 6n + 3)$ (ab Seite 64) im Fall, daß n ungerade ist, enthält. Das compilierte Programm `verify` könnte dann z.B. mit `verify -v data_file` gestartet werden, falls die Datei `data_file` die unten aufgeführte Beschreibung enthält.

Innerhalb dieses Programm-Anhangs sind *kursiv* gesetzte Worte Fachworte aus der *C*-Sprachbeschreibung [HS91] oder meiner Grammatikdefinition. Die im `typewriter`-Font gesetzten Zeichen sind buchstabengetreu benutzte *C*-Sprachelemente. Und natürlich sind mathematische Formelteile *italic* vom normalen Text abgesetzt. Die Übersetzung von englischsprachigen Programmier-Fachausdrücken ins Deutsche habe ich nur so weit getrieben, wie sie mir persönlich angemessen erschien. Z.B. benutze ich den „Zeiger“ statt des “Pointers”, habe aber “String” nicht mit „Zeichenkette“ übersetzt oder versucht die Informatikbegriffe „parsen“ oder „Token“ einzudeutschen.

Programmdokumentation

Das Programm `verify` hat die Optionen `-r -t -v -X -D` und `-+` und benötigt genau einen Dateinamen als letzten Paramater. Ohne Dateinamen oder mit unzulässigen Optionen aufgerufen gibt es eine kurze *usage*-Meldung, seine Versionsnummer und seine aktuellen Tabellengrößen aus. Dies sieht z.B. so aus:[44]

```
usage: verify [-rtvX[no]D+] filename
Version 4.1        1996/10/12/18/15
MAX_WORDS=3200  MAX_SIZE=200  MAX_SPLITS=100  MAX_BLOCKS=80  HEAP_SIZE=6400
MAX_MOVES=5  MAX_VARS=4  MAX_COND=(3*4+3)  MAX_TOKEN_LEN=40  MAX_NEST=4
MAX_MOD_DEEP=4  MAX_PROTO=1000  MAX_LINE_LEN=255  SPACING=12  VBITS=3
```

Ansonsten erwartet `verify` eine lesbare Datei des angegebenen Dateinamens, in dem sich eine syntaktisch korrekte Beschreibung (*description*) ei-

[44] Aus den letzten drei Zeilen dieser Meldung geht für den eingeweihten Benutzer hervor, welche Parameter wie beschränkt sind. Z.B. bedeutet `MAX_MOVES=5`, daß nur Subtraktions-Spiel-Familien mit höchstens 5 Zügen bearbeitet werden können

nes Subtraktions-Spieles befindet.[45] Diese wird eingelesen, unter Beachtung eventuell gegebener Optionen analysiert und auf semantische Korrektheit hin überprüft. Dabei werden normalerweise nur Inkonsistenzen in der Beschreibung in der Form `Error at` ... ausgegeben.

Defaultmäßig wird als *value* ein Sprague-Grundy-Wert `_`, `A`, `B`, `C`, ... erwartet, der als Großbuchstabe kodiert ist — um nicht mit einer Anzahl verwechselt zu werden — und als $0, 1, 2, 3, \ldots$ interpretiert wird. Mit der `-+` Option kann alternativ dazu der Wertebereich `+,-` als Gewinn- oder Verlustkodierung eingestellt werden.

Mittels der `-X[no]` Option wird ein extern definierter Wertebereich inklusive einer Berechnungsfunktion aktiviert. Hierbei kann auch optional eine ganze Zahl `no` mit angegeben werden, die an diese benutzerdefinierbare Funktion `compute_value()` übergeben wird.

Die `-r` Option (raw) dient nur zur Analyse eines *wordblock*s, berücksichtigt daher keinerlei Schleifen- oder Verzweigungssyntax, und somit auch keine Bedingungen an Variablen. Bei Benutzung dieser Option ist die Verwendung des *letstament*s auch für freie Variablen erlaubt.

Die `-t` Option (transition) erzeugt aus der Beschreibung alle zu überprüfenden Übergänge, überprüft diese aber nicht auf Konsistenz.

Die `-v` Option (verbose) gibt zusätzlich zu Fehlermeldungen über Inkonsistenzen weitere ausführliche Informationen zur Beschreibung und ihrer Überprüfung.

Die `-D` Option (dump) gibt tabellarisch alle Fallunterscheidungen bis zur Ebene der Splits aus.

Es folgt abschließend für alle einstellbaren Tabellengrößen (siehe `#define`), die neu definierten, strukturierten Typen, die globalen Variablen und alle Funktionen eine kurze Charakterisierung. Die für die *defines* angegebenen Werte sind default Werte, die für die von mir mit `verify` überprüften Beschreibungen ausreichend waren. Prinizpiell hätte man diese natürlich auch dynamisch zur Laufzeit bestimmen oder anpassen können, statt wie von mir

[45] Ab Version 4.0 wird das *branchstatment* geparst, ab 4.1 in alle internen Tabellen eingetragen und ab 4.2 sollten seine Blöcke bei der Erzeugung der Übergänge ausgewertet werden.

geschehen bereits beim Übersetzen des Programmes festlegen zu müssen.

`#include <stdio.h>`
wird benötigt um die *IO*-Funktionen wie `printf`, `fopen`, `fgets`, ... sauber zu deklarieren.

`#define   MAX_WORDS       3200`
Maximale Anzahl Worte, die in einer Beschreibung sein dürfen.
`#define   MAX_SIZE         200`
Maximale Anzahl Worte, die einen Übergang bilden.
`#define   MAX_SPLITS       100`
Maximale Anzahl von Splits, die bei einer Wortrückverfolgung erzeugt werden.
`#define   MAX_MOVES          5`
Maximale Zuganzahl der Zugmenge.
`#define   MAX_VARS           4`
Maximale Anzahl der Variablen in einer Beschreibung.
`#define   MAX_BLOCKS        80`
Maximale Anzahl von Blöcken, aus der eine Beschreibung bestehen kann.
`#define   MAX_NEST           4`
Maximale Verschachtelungstiefe der Schleifen innerhalb einer Beschreibung.
`#define   MAX_COND      (3*MAX_NEST+3)`
Maximale Anzahl von Ungleichungen, die bei Überprüfung eines Überganges auftreten dürfen.
`#define   HEAP_SIZE  (MAX_WORDS*(1+1))`
Maximale Zeichenanzahl aller geparsten Worte.
`#define   MAX_TOKEN_LEN     40`
Maximale Zeichenanzahl eines Tokens beim Scannen der Beschreibung.
`#define   MAX_MOD_DEEP       4`
Maximale Verschachtelungstiefe des `mod`-Fängers während der Überprüfung.
`#define   MAX_PROTO       1000`
Maximale Protokollgröße pro Übergang im verbose-Modus.
`#define   MAX_LINE_LEN     255`
Maximale Zeilenlänge beim Einlesen.
`#define   SPACING           12`
TAB-spacing in der Ausgabe des dump-Modus.
`#define   VBITS              3`
Kleinste ganze Zahl die $\geq \log_2($`MAX_VARS`$)$ ist.

`typedef struct { int v[MAX_VARS+1]; }  affine;`
Realisierung einer affinen Linearform mit `MAX_VARS` Variablen und einer Kon-

stanten, deren Wert in `v[0]` steht.

```
typedef struct { affine addr; affine leng; char *value; }  word;
```
Realisierung eines Wortes aus Adresse, Länge und Wert. Adresse und Länge werden durch affine Linearformen bezeichnet und der Wert durch einen Zeiger, der auf einen *NUL*-terminierten String aus dem vom Benutzer gewählten Alphabet weist.

```
typedef struct { unsigned anz, first; unsigned ex, in;
                 affine length; char var; affine start, stop;
                 unsigned active; }  block;
```
Realisierung eines Blockes. Dieser wird charakterisert durch die Anzahl belegter Worte (`anz`) die Nummer des ersten Wortes (`first`) zwei Indizes (`ex` und `in`) die auf die beiden möglichen Vorgängerblöcke zeigen (den Index `0` hat per Definition der Block mit den Werten für negative Haufenpositionen), die Länge `length` dieses Blocks und eine Variable `var`. Falls diese `!= NOT EXIST` ist, wird sie als Kontrollvariable des Blockes angesehen, der dann eine echte Schleife darstellt. In diesem Fall sind die Werte für `start` und `stop` definiert und bezeichnen die *Startexpression* bzw. *Endexpression* der Schleife. Dann kann `active` die Bedingung an die Kontrollvariable in den `conditions[]` indizieren, die aus der Startbedingung der Schleife resultiert. Dies dient dazu diese Bedingung schnell zu finden und zu ändern, wenn der zugehörge Block verlassen oder abermals durchlaufen wird.

```
unsigned  moveCNT,
```
Anzahl der Züge.

```
          freevars,
```
Anzahl der freien Variablen.

```
          condCNT,
```
Anzahl der aktuell relevanten Bedingungen.

```
          trans,
```
Zähler der generierten Übergänge.

```
          flags,
```
Interne Options- und Markierungsleiste.

```
          nest,
```
Variable für die aktuelle Block-Verschachtelungstiefe der Grammatik beim Parsen.

```
          level[MAX_NEST],
```
Feld der Blocknummern der aktuellen Verschachtelung beim Parsen.

```
          period,
```
Flag zum Merken von Endlosschleifen in der geparsten Beschreibung.

```
          total,
```

Zähler für die Anzahl untersuchter Fälle.

```
          threshold;
```

Ein vom Benutzer per Option definierbarer Wert, der in der `compute_value` Funktion dann zur Verfügung steht.

```
static unsigned  perm[MAX_VARS+1],
```

Permutationsreihenfolge für die Ausgabe von affinen Linearformen.

```
                 inver[MAX_VARS+1];
```

Inverse Permutation von `perm`.

```
affine  move[MAX_MOVES],
```

Liste der zulässigen Züge.

```
        replace[MAX_VARS],
```

Liste der aktuellen Substitutionen der Variablen.

```
        condition[MAX_COND+1];
```

Liste der aktuellen (affin-linearen) `<= 0` Bedingungen.

```
block  group[MAX_BLOCKS];
```

Liste der gespeicherten Blöcke.

```
word  sub[MAX_WORDS+1],
```

Liste der gespeicherten Worte der geparsten Grammatik.

```
      list[MAX_SIZE+1];
```

Liste der gespeicherten Worte für einen bestimmten Übergang von Blöcken.

```
FILE  *fp;
```

Zeiger auf eine Datei in der eine Beschreibung eines Spieles stehen sollte. Während des parsens können von `fp` weitere Tokens gelesen werden.

```
char  valueheap[HEAP_SIZE],
```

Haufen/Liste aller Werte der geparsten Worte. Diese sind jeweils durch *NUL* eingeschlossen.

```
      *next_free,
```

Zeiger auf den nächsten freien Eintrag in `valueheap`.

```
      valuestring[MAX_MOVES+1],
```

Feld der möglichen einzelnen Wortwerte (Buchstaben des Alphabets).

```
      varnames[MAX_VARS+1],
```

Liste, welche die verschiedenen Variablennamen (einbuchstabig) enthält.

```
      protocol[MAX_PROTO];
```

Zeichenkette, die das dynamische (Kontroll-)Protokoll enthält.

```
extern  void * malloc( size_t ), * free( void * );
```

Routinen zum Anfordern und Wiederfreigeben von Speicher — auch in `<stdlib.h>` deklariert.

```
extern  unsigned  define_values( char values[] , unsigned no );
```
Benutzerdefinierbare Funktion für den Definitionsbereich der Wortwerte. Diese müssen zeichenweise dem Feld `values` zugewiesen werden. Als Rückgabewert wird die Anzahl dieser Werte geliefert. `no` gibt die Anzahl der Züge an.

```
extern  char  compute_value( char * nextvalue[] , unsigned no ,
                             char values[] , unsigned param );
```
Benutzerdefinierbare Funktion für die Berechnungsregel eines Wertes aufgrund der Nachfolgerwerte, die durch die möglichen Züge bestimmt werden. Dabei ist `*nextvalue` ein Feld von Zeigern auf diese möglichen Nachfolgerwerte, `no` gibt deren Anzahl an, `values` beinhaltet den Definitionsbereich der Werte und `param` den vom Benutzer angebbaren Wert der Option `X`.

```
#include <setjmp.h>
static jmp_buf  catcher_environment[2], *catcher[2];
```
Zwei Fängerumgebungen mit zugehörigen Zeigern auf diese.

```
unsigned  cmp_mark=4711, mod_mark;
```
Rückgabewerte für den `cmp`-Fänger und den `mod`-Fänger.

```
#define  update_gcd(ggt,other)
```
ist ein Makro, das in `ggt` den *ggt* von `ggt` und `other` (beide vom Typ `unsigned`) akkumuliert/berechnet. Dabei muß `ggt` außerdem vom Typ *lvalue* sein.

```
bool  strequ( char *str1, char *str2 )
```
prüft zwei Strings auf Gleichheit.

```
void  zero( affine *a )
```
initialsiert `*a` mit identisch `0`.

```
int  is_zero( affine a )
```
prüft ob `a` identisch `0` ist.

```
void  incr( affine *a, affine b )
```
erhöht `*a` um `b`.

```
void  decr( affine *a, affine b )
```
vermindert `*a` um `b`.

```
void  multiply( affine *a, int b )
```
multipliziert `b` zu `*a`.

```
unsigned  reduce( affine *a )
```
teilt `*a` durch seinen größten ganzzahligen Faktor.

`unsigned  mod( affine a, unsigned  b )`
liefert den Rest von `a` mod `b` und löst gegebenenfalls die `mod`-exception aus.

`unsigned  printaff( affine arg )`
gibt `arg` schematisch nach `stdout` aus.

`int  substitute( affine *arg, unsigned k, affine repl )`
ersetzt in `*arg` die `k`-te Variable durch `repl`.

`int  desubstitute( affine *arg, unsigned k, int fac, affine repl )`
macht eine Ersetzung der `k`-ten Variablen in `arg` durch `repl` rückgängig. `fac` gibt hierbei den Koeffizienten dieser Variablen an.

`unsigned  printaffine( affine arg )`
gibt `arg` schön aufbereitet nach `stdout` aus.

`unsigned  sprintaffine( char *str, affine arg )`
gibt `arg` schön aufbereitet in den String `*str` aus.

`void  init_permutation( void )`
initialisiert die Permutation und ihre Inverse für die Variablenreihenfolge in der Reihenfolge, wie die Variablen beim Scannen der Beschreibung auftreten.

`void  swap_permutation( unsigned i, unsigned j )`
tauscht die durch `i` und `j` bezeichneten Positionen in der Permutation aus. Dies entspricht einem Vertauschen in der Variablenreihenfolge. Natürlich wird auch die inversen Permutation entsprechend geändert.

`unsigned  show_conditions( void )`
gibt die aktuell vorhandenen Bedingungen nach `stdout` aus.

`char  * next( char *ptr )`
liefert die Endposition eines Strings zurück, der durch `*ptr` bezeichnet wird.

`char  * last( char *ptr )`
sucht in dem durch `*ptr` bezeichneten String das letzte aufgetretene *EOL* oder *TAB* und gibt dessen Position zurück.

`char  calculate_value( char *pointers_to_values[] )`
liefert den Haufenwert, der aus den `moveCNT` vielen Zeigern `*pointers_to_values` auf die möglichen Zielhaufenwerte berechnet wird.

`unsigned cyclelength( char * cycle )`
liefert die Länge des Strings, der durch den Zeiger `cycle` bestimmt ist. Der String muß von *NUL* eingeschlossen sein.

`char *  shift_cycle( char * cycle , int  delta )`
liefert einen um `delta`-Werte zyklisch verschobenen Zeiger des ursprünglichen `cycle` Zeigers. Negative `delta` vermindern den Zeiger im allg., positive erhöhen ihn im allg. .

`unsigned  copy_cyle( char  str[] , char * cycle )`
kopiert einen Zyklus, der durch *NUL* eingeschlossenen sein muß, ab `cycle` in den String `str[]`.

`unsigned  printword( word sgv )`
gibt das Wort `sgv` und seine Relativadresse schön formatiert nach `stdout` aus.

`void  dumpword( word arg )`
gibt das Wort `arg` mit seiner Relativadresse schematisch nach `stdout` aus.

```
bool  primal_simplex_with_objective_fkt(
                  int koeff[ROWS+1][MAX_VARS+1+ROWS],
                  unsigned k, m, unsigned basis[ROWS] )
```
prüft, ob das durch `koeff[ROWS+1][MAX_VARS+1+ROWS]` beschriebene und `<= 0` normierte, lineare Ungleichungssystem aus `k` Bedingungen (Zeilen) in `m` Variablen (Spalten) einen nichtleeren Definitionsbereich in $\mathbb{Q}^m$ hat — somit eine zulässige Belegung der Variablen existiert. Hierzu wird eine Hilfszielfunktion errechnet und der primale Simplex-Algorithmus auf eine zulässige Ausgangslösung angewendet. `basis[ROWS]` enthält die Indizes der Basisvariablen.

```
bool dual_simplex_without_objective_fkt(
                  int koeff[ROWS+1][MAX_VARS+1+ROWS],
                  unsigned k, m, unsigned basis[ROWS] )
```
prüft, ob das durch `koeff[ROWS+1][MAX_VARS+1+ROWS]` beschriebene und `<= 0` normierte, lineare Ungleichungssystem aus `k` Bedingungen (Zeilen) in `m` Variablen (Spalten) einen nichtleeren Definitionsbereich in $\mathbb{Q}^m$ hat — somit eine konsistente Belegung der Variablen existiert. Hierzu wird der duale Simplex-Algorithmus auf eine optimale, aber nicht zulässige Ausgangslösung angewendet. `basis[ROWS]` enthält die Indizes der Basisvariablen.

```
bool  gomory_algo( int koeff[ROWS+1][MAX_VARS+1+ROWS],
                   unsigned k, m,
                   unsigned basis[ROWS], unsigned *more )
```
prüft, ob das durch `koeff[ROWS+1][MAX_VARS+1+ROWS]` beschriebene und `<= 0` normierte, lineare Ungleichungssystem aus `k` Bedingungen (Zeilen) in `m` Variablen (Spalten) einen nichtleeren Definitionsbereich für rein ganzzahlige

Variablenwerte hat. Dabei wird die Lösbarkeit in $\mathbb{Q}^m$ schon unterstellt. Die Anzahl der zusätzlich generierten Gleichungen, die in `koeff` abgelegt werden, steht in `*more`. `basis[ROWS]` enthält die Indizes der Basisvariablen.

```
bool  form_simplex_tableau_from_conditions(
                    int koeff[ROWS+1][MAX_VARS+1+ROWS],
                    unsigned *equat, *vars, unsigned basis[ROWS] )
```
formt ein Ungleichungssystem, welches als Sammlung von affinen Linearformen in `conditions[]` steht, in ein Ausgangstableau (`koeff`) für den primalen Simplex-Algorithmus um. In `*equat` wird die Anzahl der Gleichungen und in `*vars` die Anzahl der benötigten Variablen zurückgegeben.

`bool  conditions_consistent( affine *further )`
überprüft, ob das im globalen `conditions[]`-Feld gespeicherte Ungleichungssystem erfüllbar ist, falls `further` ein *NULL*-Zeiger ist, ansonst wird `*further` als weitere zu erfüllende ≤ 0-Relation aufgefaßt und überprüft, ob diese mit den aktuell vorhandenen Bedingungen konsistent ist.

`int  cmp( affine a, affine b )`
vergleicht zwei affine Linearformen `a` und `b`, ob sie unter den in `conditions[]` abgespeicherten Bedingungen $<$, $>$, $=$ oder unvergleichbar sind. Im letzten Fall wird die `cmp`-exception ausgelöst. Dies bewirkt die Rückkehr an eine bestimmte übergeordnete Aufrufstelle der aktuellen Verschachtelung, den sogenannten *exceptioncatcher*. Algorithmisch simuliert dies eine Rückkehr (*Return)* über mehrere Funktionsebenen bis zu einer Fehlerfängerebene zurück.

`int  unequal_zero( affine arg )`
ist ein spezieller Vergleich mit der 0-wertmäßigen affinen Linearform. Es entspricht logisch `cmp( arg, (affine)0 )`.

`unsigned  scanword( char token[], char *string )`
parst das Token `token`, das als Wortwert aufgefaßt wird, und legt es an einer durch den Zeiger `string` bezeichneten Stelle ab.

`int  scanlinear( char token[], affine *help )`
parst das Token `token`, das als affine Linearform in kanonischer Grammatik vorliegen sollte, und speichert es in `*help`.

`bool  get_token_from_input( char token[MAX_TOKEN_LEN] )`
Liest zeilenweise aus der globalen Datei `*fp` in einen internen Buffer und liefert aus dem das nächste Token nach `token`.

`void  skip_upto_eol( void )`
überliest in der durch den globalen Zeiger `fp` bezeichneten Datei alles bis

zum Zeilenende.

```
unsigned  scanmoves( affine move[] )
```
parst eine eventuell leere Liste von *orderrelation*s, gefolgt von einer nichtleeren Zugliste deren Züge in affinen Linearformen dargestellt sind, und speichert sie in `move[]`.

```
void  close_present_block( unsigned no , unsigned m ,
                           affine counter )
```
schließt den aktuellen Block `no` ab. In `m` steht die Anzahl der insgesamt gescanten Worte, und `counter` gibt deren Gesamtlänge als affine Linearform an.

```
void  open_new_block( unsigned no , unsigned m )
```
eröffnet einen neuen Block `no` nach dem `m`-ten eingelesenen Wort.

```
void  parse_loophead( char token[] , unsigned m , unsigned no ,
                      unsigned j , affine counter )
```
parst einen Schleifenkopf in der Beschreibung (von *FOR* bis ausschließlich *REP*). `token` enthält das erste zu untersuchende Token, `m` gibt die Anzahl der gelesenen Worte an, `no` die Anzahl der gelesenen Blöcke, `j` eine Referenz auf den Token-Zähler und `counter` die Adresse in der momentanen Wortliste.

```
unsigned  parse_possible_loop(
                      char token[] , unsigned m , unsigned no ,
                      unsigned j , affine counter )
```
parst eine mögliche Schleife. `token` enthält das erste zu untersuchende Token, `m` gibt die Anzahl der gelesenen Worte an, `no` die Anzahl der gelesenen Blöcke, `j` eine Referenz auf den Token-Zähler und `counter` die Adresse in der momentanen Wortliste.

```
void  parse_upto_then( unsigned no )
```
parst eine *IF*-expression bis einschließlich des ersten *THEN*-Schlüsselwortes.

```
void  check_branch_nest_syntax( char type )
```
überprüft die *IF-ELIF-ELSE-FI*-Klammerung eines *branchstatement*s.

```
bool  parse_bossible_branch(
                      char token[] , unsigned m , unsigned no ,
                      unsigned j , affine counter )
```
parst eine mögliche Verzweigung. `token` enthält das erste zu untersuchende Token, `m` gibt die Anzahl der gelesenen Worte an, `no` die Anzahl der gelesenen Blöcke, `j`, eine Referenz auf den Token-Zähler und `counter` die Adresse in der momentanen Wortliste.

```
void  parse_let_statement( char token[] )
```
parst eine Zuweisungsanweisung. `token` enthält das erste zu untersuchende Token.

```
bool  parse_word( char token[] , unsigned m , affine *counter )
```
parst ein *word* der Beschreibung. `token` enthält das erste zu untersuchende Token. `m` gibt die Anzahl der gelesenen Worte an und `counter` die Adresse in der momentanen Wortliste.

```
unsigned  parse_grammar( void )
```
parst die zum Lesen geöffnete Datei `*fp`, führt die syntaktische Analyse durch und speichert die Beschreibung in den globalen Variablen `group[MAX_BLOCKS]` und `sub[MAX_WORDS+1]`.

```
unsigned  build_back_list( word *sgv, unsigned k,
                           word back[MAX_SPLITS] )
```
baut zum `k`-ten Wort (`*sgv`) die Liste der durch einen Zug erreichbaren Worte in `back[]` auf. Die Größe dieser Liste wird zurückgeliefert.

```
unsigned align_back_list( word back[MAX_SPLITS], unsigned n )
```
zerlegt die Liste der Worte in `back[]` in ihre Splits. `n` gibt die Anzahl der Splits an und wird als Rückgabewert zurückgeliefert.

```
unsigned  check_split( char *value_ptr[] , word orig ,
                       unsigned index , word back[] ,
                       unsigned i , unsigned err )
```
berechnet aus den `i`-ten Splits in der `back[]`-Liste, die durch die `*value_ptr[]`-Zeiger bezeichnet werden, den regelkonformen Wert und vergleicht ihn mit dem des Wortes `orig`. `index` gibt die Nummer des Wortes `orig` in dem untersuchten Übergang an, und in `err` wird der Fehlerzähler übergeben.

```
int  check_word( word *sgv, unsigned k , unsigned *err)
```
überprüft das `k`-te Wort (`*sgv`) auf Konsistenz und zählt in `*err` gegebenenfalls die Fehler.

```
int  check_transition( word *sgv, unsigned m )
```
überprüft die Liste `sgv` der Länge `m` auf Konsistenz. Im wesentlichen ist dies eine Schleife in der mittels `check_word()` die einzelnen Worte geprüft werden. Dieser Schleife ist der komplexe `mod`-exception-catcher vorangestellt.

```
unsigned  eliminate_empty_words( unsigned m , word *sgv , *safe )
```
entfernt Worte der Länge null aus dem zu prüfenden Übergang der Länge `m`, der in `sgv` steht. In `safe` steht gegebenfalls der ursprüngliche Übergang.

Die Anzahl der entfernten Worte wird zurückgeliefert.

`int  analyze_propagation( word *sgv, unsigned m )`
überprüft die Liste `sgv` der Länge `m` auf Konsistenz. Diese Funktion ruft im wesentlichen `check_transition()` mit vorgeschaltetem `cmp`-exception-catcher auf.

`unsigned  prepend_infinity_block( unsigned k, affine *counter )`
fügt den nullten Block vor den aktuellen Übergang ab Position `k`. `*counter` wird auf identisch 0 gesetzt.

```
unsigned  prepend_this_block( unsigned k, unsigned no,
                              affine *counter )
```
fügt den Block `no` vor den aktuellen Übergang und erweitert dabei die Wortliste `list[]` ab Position `k`. Hierbei wird `*counter` um die Länge des Blockes `no` vermindert.

```
unsigned  go_back_one_iteration( unsigned k, unsigned no,
                                 affine counter )
```
läuft die aktuelle Schleife abermals durch und dekrementiert dabei die zugehörige Schleifenvariable um 1.

```
unsigned  leave_loop_through_head( unsigned k, unsigned no,
                                   affine counter )
```
verläßt den aktuellen Block und substituiert die Schleifenvariable durch ihre *Startexpression*.

```
unsigned  skip_previuous_block( unsigned k, unsigned no,
                                affine counter )
```
überspringt die vorausgehende Schleife bis zum übernächsten Block. Die Bedingung dafür, das heißt *Endexpression* $+1 \leq$ *Startexpression*, wird zum Ungleichungssytem dazugefügt.

```
unsigned  enter_block_from_its_end( unsigned k, unsigned no,
                                    affine counter )
```
betritt die vorangehende Schleife durch ihr Ende. Die Schleifenvariable wird dabei durch ihre *Endexpression* ersetzt.

```
unsigned  prepend_further_block( unsigned k, unsigned no,
                                 affine *counter )
```
untersucht alle Möglichkeiten — dies sind pro Funktionsaufruf in der Regel zwei, bei *branchstatment*s eventuell mehr —, einen weiteren Block, der dem aktuellen Block `no` logisch vorangehen könnte, vor dem zu erzeugenden Übergang einzufügen. Hierbei gibt `*counter` die noch zu erbringende Min-

destlänge der „Verlängerung“ an. Diese Funktion ruft sich selbst rekursiv solange auf bis `*counter` kleiner oder gleich null geworden ist.

`void show_grammar( unsigned no )`
gibt die in `no` Blöcken geparste Beschreibung schön nach `stdout` aus.

`int generate_transitions( unsigned no )`
erzeugt den Übergang angefangen beim Block `no` durch Voreinanderhängen möglicher in `group[]` gespeicherter Blöcke.

`unsigned init_valuestring_and_infinity_block( void )`
initialisiert den Definitionsbereich der Wortwerte und legt den Wortwert des nullten Blockes fest.

`int main( unsigned argc, char *argv[] )`
ist die zentrale Startroutine, die Programm-Optionen auswertet, die Grammatik einscant, syntaktisch prüft, und diese dann mittels `generate_transitions()` auf Konsistenz hin überprüft.

Es folgt die Tabelle sämtlicher möglicher Fehlermeldungen:

```
Returnvalue       Error Message
case   1:    "too many variables  increase MAX_VARS\n"
case   2:    "too many words  increase MAX_WORDS\n"
case   3:    "too many moves  increase MAX_MOVES\n"
case   4:    "heap overflow  increase MAX_SPLITS\n"
case   5:    "list overflow  increase MAX_SIZE\n"
case   6:    "too many blocks  increase MAX_BLOCKS\n"
case   7:    "too deep loop nesting  increase MAX_NEST\n"
             "too deep branch nesting  increase MAX_NEST\n"
case   8:    "too many conditions  increase MAX_COND\n"
             "too many conditions in Gomory  increase MAX_COND\n"
case   9:    "too many wordvalues  increase HEAP_SIZE\n"
case  10:    "invalid variablename: %s\n",var
case  11:    "FROM expected\n"
case  12:    "UPTO expected\n"
case  13:    "REPEAT missing\n"
case  14:    "control variable %c already in use\n",var
             "free variable %c can't be used as control variable\n",var
case  15:    "variable %c not defined\n",var
             "variable %c not active\n",var
case  16:    "variable %c selfdefined\n",var
case  17:    "ENDREP without REPEAT\n"
             "FI without IF\n"
             "ELSE without IF\n"
```

```
                "ELIF without IF\n"
case  18:       "missing  ENDREP\n"
                "missing  FI\n"
case  19:       "missing ) in token:%s\n",token
case  20:       "= expected as second char in token:%s\n",token
                "> or < expected followed by = in token %s\n",token
                "> or < or = or ! expected followed by = in token %s\n",token
case  21:       "unknown letter:%s\n",token
case  22:       "pattern after ) in token:%s\n",token
                "pattern after } in token:%s\n",token
case  23:       "length of loopbody is independent of %c\n",var
case  24:       "words can have negative length\n"
case  25:       "variables in wrong lexicographical order\n"
case  26:       "unknown option %s\n",opt
case  27:       "can't open %s\n",filename
case  28:       "no infinity loop given\n"
                "several infinity loops given\n"
case  29:       "internal ERROR  different length\n"
                "internal ERROR  variable substitution\n"
                "internal ERROR  no admissible solution available\n"
                "internal ERROR  conditions\n"
                "internal ERROR  consistence\n"
                "internal ERROR  define values\n"
                "internal ERROR  need 2-complement representation\n"
case  30:       "malloc: not enough space\n"
case  31:       "int overflow\n"
                "unsigned overflow   VBITS too large\n"
case  32:       "too many congruences  increase MAX_MOD_DEEP\n"
case  33:       "syntax error  missing value of letter in %s\n",token
case  34:       "congruence MOD %u is decisive\n",rem
                "congruence MOD %u of variable %c is decisive\n",rem,var
case  35:       "THEN expected\n"
case  36:       "several ELSE\n"
case  37:       "line too long  increase MAX_LINE_LEN\n"
case  38:       "token too long  increase MAX_TOKEN_LEN\n"
```

Die folgenden drei Programmstellen sind wohl am anspruchvollsten:

- mod-catcher in `check_transition` — schon durch Kommentare kenntlich gemacht [HS91, Kap. 19.4].
- cmp-catcher in `analyze_propagation` — schon durch Kommentare kenntlich gemacht [HS91, Kap. 19.4].
- Alle `if`- und `?`-Anweisungen mit ihren `&&` und `||` Verknüpfungen in `primal_simplex_with_objective_fkt`. Hierbei liegt die „Schwierigkeit" darin, zu erkennen, daß die vorhandenen Vergleiche an diesen Stellen nötig und hinreichend, aber an anderen Stellen nicht nötig sind.

Die Auswahl der letzten „Stelle" ist mir schwer gefallen, da diese Gewichtung sicherlich stark subjektiv ist. Wie ein erfahrener Programmierer am Quellcode erkennt, habe ich das Programm in viele Funktionen zerlegt. Diese werden vielfach nur an einer Programmstelle aufgerufen — sprich existieren als eigenständige Funktion nur — , um gerade solche sonst komplizierten Stellen aufzulösen.

Quellcode

```
#define _POSIX_SOURCE   /* (C) Achim Flammenkamp, University of Jena, Germany */
#define  VERSION   "Version 4.1a       1996/10/12/18/15"
#include <stdio.h>
#define  MAX_WORDS    3200
#define  MAX_SIZE      200
#define  MAX_SPLITS    100
#define  MAX_MOVES       5
#define  MAX_VARS        4
#define  MAX_BLOCKS     80
#define  MAX_NEST        4
#define  MAX_COND   (3*MAX_NEST+3)
#define  MAX_TOKEN_LEN  40
#define  MAX_LINE_LEN  255
#define  LETTERS         1
#define  HEAP_SIZE  (MAX_WORDS*(1+LETTERS))

typedef struct { int  v[MAX_VARS+1]; } affine;  /* must be embedded in struct */
typedef struct { affine  addr;  affine  leng;  char *value; }  word;
typedef struct { unsigned  anz, first;  unsigned  ex, in;   affine  length;
                 char  var;  affine  start, stop;  unsigned  active; }  block;
#define  bool  unsigned
#define  MAX_PROTO    1000

#define  Identifier2String(para)    #para
#define  tot(arg)    #arg "=" Identifier2String(arg)
#define  NOT_EXIST  ' '
#define  WL      (1<<0)
#define  SPECIAL (1<<2)
#define  RAW     (1<<3)
#define  PARSE   (1<<4)
#define  VERBOSE (1<<5)
#define  DUMP    (1<<6)
#define  SPACING  12
#define  MOST    ~ (~(unsigned)0 >> 1)

unsigned  moveCNT, freevars, condCNT, trans, flags, total, threshold;
unsigned  nest, level[MAX_NEST], period, ind, lineno;  /* parsing globals */
static unsigned  perm[MAX_VARS+1], inver[MAX_VARS+1];
affine  move[MAX_MOVES], replace[MAX_VARS], condition[MAX_COND+1];
block  group[MAX_BLOCKS];
word  sub[MAX_WORDS+1], list[MAX_SIZE+1];
FILE  *fp;
char  valueheap[HEAP_SIZE], *next_free, valuestring[MAX_MOVES+1],
      varnames[MAX_VARS+1], protocol[MAX_PROTO], inputline[MAX_LINE_LEN];
extern  void * malloc( size_t ), * free( void * );   /* in <stdlib.h> */
extern  unsigned  define_values( char [] , unsigned );
extern  char  compute_value( char * [] , unsigned , char [] , unsigned );


#include <setjmp.h>
static jmp_buf /* is array type */  catcher_environment[2], *catcher[2];
unsigned const  cmp_mark=4711;
#define  VBITS  3
#if  1<<VBITS <= MAX_VARS
#error  "2^VBITS <= MAX_VARS"
#endif
#define  MAX_MOD_DEEP  MAX_VARS
```

```
#define  update_gcd(ggt,other)  do { unsigned register  r, q= (other);  \
       /* ggt must be lvalue */  while(q){ r= ggt%q; ggt=q; q=r; } } while (0)


int strequ( char *str1, char *str2 )
{  while (*str1 && *str1 == *str2)
      str1++,  str2++;
   return  *str1 == *str2;
}


void zero( affine *a )
{  unsigned  h;
   for (h=0;h<=MAX_VARS;h++)
      a->v[h] = 0;
   return ;
}


int is_zero( affine a )
{  unsigned  h;
   for (h=0;h<=MAX_VARS;h++)
      if (a.v[h])  return  0;
   return  1;
}


void incr( affine *a, affine b )
{  unsigned  h;
   for (h=0;h<=MAX_VARS;h++)
      a->v[h] += b.v[h];
   return ;
}


void decr( affine *a, affine b )
{  unsigned  h;
   for (h=0;h<=MAX_VARS;h++)
      a->v[h] -= b.v[h];
   return ;
}


void multiply( affine *a, int b )
{  unsigned  h;
   for (h=0;h<=MAX_VARS;h++)
      a->v[h] *= b;
   return ;
}


unsigned reduce( affine *a )
{  unsigned  h;
   int  ggt=0;
   for (h=0;ggt!=1 && h<=MAX_VARS;h++)
      update_gcd( ggt, (a->v[h]>=0 ? a->v[h] : -a->v[h]) );
   if (ggt > 1)
      for (h=0;h<=MAX_VARS;h++)
         a->v[h] /= ggt;
   return  ggt;
}
```

```
unsigned mod( affine a, unsigned  b )
{  unsigned  h;
   for (h=1;h<=MAX_VARS;h++)
      if (a.v[h]%(int)b)
      {  if (*catcher[1])
         {  unsigned  ggt = (a.v[h] ? a.v[h] : -a.v[h]);
            update_gcd(ggt,b);
            b /= ggt;
            if (flags&VERBOSE)
               fprintf(stdout,"congruence MOD %u of variable %c is decisive\n",
                       b,varnames[h-1]);
            if ((b<<VBITS)>>VBITS != b)
            {  fprintf(stderr,"unsigned overflow   VBITS too large\n");
               exit(31);
            }
            longjmp(*catcher[1], b<<VBITS | h);
         }
         else
         {  fprintf(stderr,"congruence MOD %u of variable %c is decisive\n",
                    b,varnames[h-1]);
            exit(34);
         }
      }
   h = (int)b + a.v[0] % (int)b;  if (h >= b)  h -= b;
   return  h;
}


unsigned printaff( affine arg )
{  unsigned  h, i, j=0;
   for (h=0;h<=MAX_VARS;h++)
      if (!(i=perm[h]))
         j+= fprintf(stdout,"%+2d",arg.v[i]);
      else if (varnames[i-1])
         j+= fprintf(stdout,"%+2d%c",arg.v[i],varnames[i-1]);
   return j;
}


int substitute( affine *arg, unsigned k, affine repl )
{  unsigned  i;
   int  fac;
   if (fac=arg->v[k])
   {  arg->v[k]=0;
      for (i=0;i<=MAX_VARS;i++)
         arg->v[i] += fac * repl.v[i];
   }
   return  fac;
}


int desubstitute( affine *arg, unsigned k, int fac, affine repl )
{  unsigned  j;
   if (fac)
   {  for(j=0;j<=MAX_VARS;j++)
         arg->v[j] -= fac * repl.v[j];
      arg->v[k] += fac;
   }
   return  arg->v[k];
}
```

```
unsigned printaffine( affine arg )
{  unsigned  h, i, j, k;
   k=j=0;
   for (h=0;h<=MAX_VARS;h++)
   {  i=perm[h];
      if (k  &&  arg.v[i] > 0)
         j += printf("+");
      if (!i && arg.v[i])
         j += printf("%d", arg.v[i]);
      else if (arg.v[i]==1)
         j += printf("%c",varnames[i-1]);
      else if (arg.v[i]== -1)
         j += printf("-%c",varnames[i-1]);
      else if (arg.v[i])
         j += printf("%d%c", arg.v[i],varnames[i-1]);
      if (arg.v[i])  k++;
   }
   if (!k)
      j += printf("0");
   return j;
}


unsigned sprintaffine( char *str, affine arg )
{  unsigned  h, i, j, k;
   k=j=0;
   for (h=0;h<=MAX_VARS;h++)
   {  i=perm[h];
      if (k  &&  arg.v[i] > 0)
         j += sprintf(str+j,"+");
      if (!i && arg.v[i])
         j += sprintf(str+j,"%d", arg.v[i]);
      else if (arg.v[i]==1)
         j += sprintf(str+j,"%c",varnames[i-1]);
      else if (arg.v[i]== -1)
         j += sprintf(str+j,"-%c",varnames[i-1]);
      else if (arg.v[i])
         j += sprintf(str+j,"%d%c", arg.v[i],varnames[i-1]);
      if (arg.v[i])  k++;
   }
   if (!k)
      j += sprintf(str+j,"0");
   return  j;
}


void init_permutation( void )
{  unsigned  h;
   for (h=0;h<MAX_VARS;h++)
   {  perm[h]=h+1;
      inver[h+1]=h;
   }
   perm[MAX_VARS]=0;
   inver[0]=MAX_VARS;
   return ;
}


void swap_permutation( unsigned i, unsigned j )
{  unsigned  h, temp;
   for (h=i;h+h<j+i;h++)
```

```
   {  temp = inver[h];
      inver[h] = inver[j-h+i];
      inver[j-h+i] = temp;
      perm[inver[h]]=h;
      perm[inver[j-h+i]]=j-h+i;
   }
   return ;
}


unsigned show_conditions( void )
{  unsigned  h;
   for (h=0;h<condCNT;h++)
   {  printaffine(condition[h]);
      printf(" <= 0\n");
   }
   return  condCNT;
}


char * next( char *ptr )
{  while (*ptr)  ptr++;
   return  ptr;
}


char * last( char *ptr )
{  while (*ptr!='\n' && *ptr!='\t')  ptr--;
   return  ptr;
}


char  calculate_value( char *pointers_to_values[] )
{  unsigned  h=0, i;
   if (flags&SPECIAL)
      return  compute_value( pointers_to_values, moveCNT, valuestring,
                             threshold );
   else if (flags&WL)
   {  for (h=0;h<moveCNT;h++)
         if (*(pointers_to_values)[h] == valuestring[0])
            break;
         return  valuestring[h<moveCNT];
   }
   else
   {  for (i=0;h<moveCNT;i++)
         for (h=0;h<moveCNT;h++)
            if (*(pointers_to_values)[h] == valuestring[i])
               break;
      return  valuestring[i-1];
   }
}


unsigned cyclelength( char * cycle )
{  unsigned  len;
   if (! *cycle)  return 0;
   for (len=0; *(cycle-len) ; len++);
   while ( * ++cycle ) len++;
   return  len;
}
```

```
char * shift_cycle( char * cycle, int  delta )
{
   if (! *cycle)
      ;
   else if (delta > 0)
   {  while (delta--)
         if (! *++cycle)
         {  while (*--cycle);
            cycle++;
         }
   }
   else if (delta < 0)
   {  while (delta++)
         if (! *--cycle)
         {  while (*++cycle);
            cycle--;
         }
   }
   return  cycle;
}


unsigned copy_cycle( char  str[], char * cycle )
{  char  *stop= cycle, *start= str;
   if (!*cycle)  return  str[0]='\0';
   for (; *str = *cycle ; str++,cycle++);
   while (*--cycle);
   cycle++;
   while (cycle != stop)
      *str++ = *cycle++;
   *str= '\0';
   return  str-start;
}


unsigned printword( word sgv )
{  unsigned  h=0, i;
   char  cyclevalue[MAX_TOKEN_LEN];
   for (i=MAX_VARS;i;i--)
      if (sgv.leng.v[i])
         break;
   copy_cycle(cyclevalue, sgv.value);
   if (!i  &&  sgv.leng.v[i] < MAX_TOKEN_LEN)
      cyclevalue[sgv.leng.v[i]]='\0';
   h+= printf("<%s>(",cyclevalue);
   h+= printaffine(sgv.leng);
   putchar( (int)')' );
   return  h;
}


void dumpword( word arg )
{  printaff(arg.addr);
   printf(":  <%s>(",arg.value);
   printaffine(arg.leng);
   printf(")");
   return ;
}


#define  ROWS  (MAX_COND+1)
#if  -1U>>1 != -1UL>>1
```

```
#define  preciser  long
#else
#define  preciser  double
#endif

bool primal_simplex_with_objective_fkt( int koeff[ROWS+1][MAX_VARS+1+ROWS],
                                 unsigned k, unsigned m, unsigned basis[ROWS] )
{  unsigned  h, i, j, ggt, jj;
   int  cc, pivot;
   preciser  compare;
   for (i=0;i<k;i++)
   {  ggt= -koeff[i][0];
      for (j=1;ggt != 1 && j<m;j++)
         update_gcd(ggt, (koeff[i][j]>=0 ? koeff[i][j] : -koeff[i][j]) );
      if (ggt > 1)
         for (j=0;j<m;j++)
            koeff[i][j] /= (int)ggt;
   }
   for (;;)
   {  j=0;
      for (h=1;h<m;h++)
         if (koeff[0][h]  &&  (!j || koeff[0][h] > koeff[0][j]))
            j=h;
      if (koeff[0][0]  &&  koeff[0][j] <= 0)
         return  0;
      if (!j)
         break;
      h=0;
      if (koeff[0][j] > 0)
      {  for (i=1;i<k;i++)
            if (koeff[i][j] > 0  &&  (!h ||
                koeff[i][0]*koeff[h][j] > koeff[h][0]*koeff[i][j]) )
               h=i;
      }
      else
      {  for (i=1;i<k;i++)
            if (koeff[i][j] < 0)
               h=i;
         for (jj=0;jj<m;jj++)
            koeff[h][jj]= -koeff[h][jj];
      }
      if (!h)
      {  fprintf(stderr,"internal ERROR  conditions\n");
         exit(29);  /* should never happen */
      }
      basis[h-1]=j;
      for (i=0;i<k;i++)
      {  if (i==h)  continue;
         ggt= koeff[h][j];
         cc = (koeff[i][j]>=0 ? koeff[i][j] : -koeff[i][j]);
         update_gcd(ggt,cc);
         cc= koeff[i][j]/(int)ggt;
         pivot= koeff[h][j]/ggt;
         for (jj=0;jj<m;jj++)
         {  compare = (preciser)pivot*koeff[i][jj] - (preciser)cc*koeff[h][jj];
            if (compare != (preciser)(koeff[i][jj] = compare))
            {  fprintf(stderr,"int overflow\n");
               exit(31);
            }
         }
      }
   }
```

```
   return  1;
}


bool dual_simplex_without_objective_fkt( int koeff[ROWS+1][MAX_VARS+1+ROWS],
                                 unsigned k, unsigned m, unsigned basis[ROWS] )
{  unsigned  h, i, j, ggt, jj;
   int  pivot, cc;
   preciser  compare;
   h=k;
   for (;koeff[h][0]>0;)
   {  ggt= koeff[h][0];
      for (j=1;ggt>1 && j<=m;j++)
      {  i=(koeff[h][j]>=0 ? koeff[h][j] : -koeff[h][j]);
         update_gcd(ggt,i);
      }
      if (ggt != 1)
         for (j=0;j<=m;j++)
            koeff[h][j] /= (int)ggt;
      j=0;
      for (jj=m;jj>=1;jj--)
         if (koeff[h][jj] < 0  &&  koeff[h][jj] < koeff[h][j])
            j=jj;
      if (!j)
         return  0;
      basis[h-1]=j;
      for (jj=0;jj<=m;jj++)
         koeff[h][jj] = -koeff[h][jj];
      for (i=0;i<=k;i++)
      {  if (i==h)  continue;
         ggt= koeff[h][j];
         cc = (koeff[i][j]>=0 ? koeff[i][j] : -koeff[i][j]);
         update_gcd(ggt,cc);
         cc= koeff[i][j]/(int)ggt;
         pivot= koeff[h][j]/ggt;
         for (jj=0;jj<=m;jj++)
         {  compare = (preciser)pivot*koeff[i][jj] - (preciser)cc*koeff[h][jj];
            if (compare != (preciser)(koeff[i][jj] = compare))
            {  fprintf(stderr,"int overflow\n");
               exit(31);
            }
         }
      }
      h=0;
      for (i=1;i<=k;i++)
         if (koeff[i][0] > koeff[h][0])  h=i;
   }
   return  1;
}


bool gomory_algo( int koeff[ROWS+1][MAX_VARS+1+ROWS], unsigned k, unsigned m,
                  unsigned basis[ROWS], unsigned *more )
{  unsigned  h, i, j, numer, denom, ggt, den, num;
   *more=0;
   for (;;)
   {  den=1;  num=0;  i=0;
      for (h=1;h<k;h++)
      {  j = basis[h-1];
         if (j > MAX_VARS)  continue;
         ggt=koeff[h][j];
         update_gcd(ggt,-koeff[h][0]);
```

```
         denom= koeff[h][j]/ggt;
         numer= -koeff[h][0]/ggt % denom;
#ifndef  NORMAL_CUT
         if (numer)  numer = denom-numer;
#endif
         if ((preciser)numer*den > (preciser)num*denom)
         {  den=denom;
            num=numer;
            i=h;
         }
      }
      if (!i)
         break;
      if (k > ROWS)
      {  fprintf(stderr,"too many conditions in Gomory  increase MAX_COND\n");
         exit(8);
      }
      h= basis[i-1];
      koeff[k][0]= -koeff[i][0] % koeff[i][h];
#ifndef  NORMAL_CUT
      if (koeff[k][0])
         koeff[k][0]= koeff[i][h] - koeff[k][0];
#endif
      for (j=1;j<m;j++)
      {  if (koeff[i][j] <= 0)
         {  koeff[k][j]=  -koeff[i][j] % koeff[i][h] ;
#ifdef  NORMAL_CUT
            if (koeff[k][j])
               koeff[k][j] -= koeff[i][h];
#else
            koeff[k][j]= -koeff[k][j];
#endif
         }
         else
         {  koeff[k][j]= koeff[i][j] % koeff[i][h];
#ifdef  NORMAL_CUT
            koeff[k][j]= -koeff[k][j];
#else
            if (koeff[k][j])
               koeff[k][j] -= koeff[i][h];
#endif
         }
      }
      koeff[k][m]= koeff[i][h];
      for (i=0;i<k;i++)
         koeff[i][m]= 0;
      basis[k-1]= m;
      for (i=1;i<=k;i++)
      {  ggt= (i==k ? koeff[i][0] : -koeff[i][0]);
         for (j=1;ggt != 1 && j<=m;j++)
            update_gcd(ggt, (koeff[i][j]>=0 ? koeff[i][j] : -koeff[i][j]) );
         if (ggt > 1)
            for (j=0;j<=m;j++)
               koeff[i][j] /= (int)ggt;
      }
      if (!dual_simplex_without_objective_fkt(koeff,k,m,basis))
         return  0;
      j=basis[k-1];
      k++;  m++;
      ++ *more;
   }
   return  1;
```

```
}


bool form_simplex_tableau_from_conditions( int koeff[ROWS+1][MAX_VARS+1+ROWS],
                       unsigned *equat, unsigned *vars, unsigned basis[ROWS] )
{  unsigned  h, i, j, k;
   for (h=k=0;k<condCNT;k++)
   {  for (j=1;j<=MAX_VARS;j++)
         if (condition[k].v[j])  break;
      if (j<=MAX_VARS)
      {  h++;
         for (j=0;j<=MAX_VARS;j++)
            koeff[h][j] = condition[k].v[j];
         for (;j<MAX_VARS+ROWS;j++)
            koeff[h][j] = 0;
         koeff[h][MAX_VARS+h] = 1;
         basis[h-1]=MAX_VARS+h;
         if (koeff[h][0] > 0)
            for (j=0;j<=MAX_VARS+h;j++)
               koeff[h][j] = -koeff[h][j];
      }
      else if (condition[k].v[0] > 0)
         return  0;
   }
   k= ++h;
   for (h=0;h<MAX_VARS+k;h++)
      koeff[0][h] = 0;
   i=0;
   for (h=1;h<k;h++)
   {  if (koeff[h][MAX_VARS+h] < 0)
         for (j=0;j<MAX_VARS+k;j++)
            koeff[0][j] += koeff[h][j];
   }
   *equat = k;
   *vars = k+MAX_VARS;
   return  1;
}


bool conditions_consistent( affine *further )
{  unsigned  h, i, j, ggt;
   int  cc;
   static int  pivot=0, coeff[ROWS+1][MAX_VARS+1+ROWS];
   static unsigned  k, m, base[ROWS];
   if (!further)
   {  pivot=0;
      if (!form_simplex_tableau_from_conditions(coeff,&k,&m,base))
         return  0;
      if (!primal_simplex_with_objective_fkt(coeff,k,m,base))
         return  0;
      if (!gomory_algo(coeff,k,m,base,&h))
         return  0;
      k += h;
      m += h;
      return  pivot=1;
   }
   else if (!pivot)
   {  fprintf(stderr,"internal ERROR  no admissible solution available\n");
      exit(29);
      return  0;
   }
   else
```

```
   {  /*  check system with additional condition  */
      int  koeff[ROWS+1][MAX_VARS+1+ROWS];
      unsigned  basis[ROWS];
      cc=1;
      for (j=1;j<k;j++)
      {  ggt= coeff[j][base[j-1]];
         h= ggt;
         update_gcd(ggt,cc);
         h /= ggt;
         if ((preciser)(h*cc) != (preciser)h * cc )
         {  fprintf(stderr,"int overflow\n");
            exit(31);
         }
         cc *= h;
      }
      for (h=0;h<m;h++)
      {  koeff[k][h] = (h<=MAX_VARS ? further->v[h] : 0);
         koeff[k][h] *= cc;
         for (j=1;j<k;j++)
         {  i = base[j-1];
            if (i<=MAX_VARS)
               koeff[k][h] -=  further->v[i] * coeff[j][h]*(cc/coeff[j][i]);
         }
      }
      for (h=0;h<k;h++)
         for (j=0;j<m;j++)
            koeff[h][j] = coeff[h][j];
      for (h=0;h<k;h++)
         koeff[h][m] = 0;
      koeff[k][m]= cc;
      for (j=1;j<k;j++)
         basis[j-1]=base[j-1];
      basis[k-1]=m;
      if (!dual_simplex_without_objective_fkt(koeff,k,m,basis))
         return  0;
      else
         return  gomory_algo(koeff,k+1,m+1,basis,&h);
   }
}


int cmp( affine a, affine b )
{  unsigned  h;
   if (flags&RAW)
   {  for (h=0;h<=MAX_VARS;h++)
         if (a.v[perm[h]] < b.v[perm[h]])
            return  -1;
         else if (a.v[perm[h]] > b.v[perm[h]])
            return  +1;
      return  0 ;
   }
   else
   {  condition[condCNT]=b;
      decr(&condition[condCNT],a);
      condition[condCNT].v[0] += 1;
      h= conditions_consistent(&condition[condCNT]);
      condition[condCNT]=a;
      decr(&condition[condCNT],b);
      condition[condCNT].v[0] += 1;
      if (!conditions_consistent(&condition[condCNT]))
         return  h;
      if (!h)
```

```
         return  -1;
      else
      {  if (flags&VERBOSE)
         {  fprintf(stdout,"expressions indifferent:  ");
            printaffine(b);  printf("  <&>  ");
            printaffine(a);  printf("\n");
         }
         condition[condCNT].v[0] -= 1;
         if (*catcher[1]  &&  *catcher[0])
            longjmp(*catcher[1], (1<<VBITS)-1);
         else if (*catcher[0])
            longjmp(*catcher[0], cmp_mark);
         else
            return  MOST;
      }
   }
}


int unequal_zero( affine arg )
{  affine  help;
   zero(&help);
   return  cmp(arg,help) ;
}


unsigned  scanword( char token[], char *string )
{  unsigned  h, i=0, j;
   for (h=j=0;token[h] && string+j < valueheap+HEAP_SIZE; h++,j++)
      if (token[h] >= '0' && token[h] <= '9')
         i *= 10,  i += token[h]-'0';
      else
      {  if ((flags&WL) || (flags&SPECIAL))
         {  for (i=0;token[h]!=valuestring[i];i++)
               if (!valuestring[i])
               {  fprintf(stderr,"line %u: ",lineno);
                  fprintf(stderr,"unknown letter:%s\n",token);
                  exit(21);
               }
            string[j] = token[h];
         }
         else
         {  if (token[h] == '_')
               string[j] = '_';
            else if (token[h] >= 'A' && token[h] <= 'A'+moveCNT-1)
               string[j] = 'a' + token[h]-'A';
            else
            {  fprintf(stderr,"line %u: ",lineno);
               fprintf(stderr,"unknown letter:%s\n",token);
               exit(21);
            }
         }
         i= (!h  ||  token[h-1] < '0'  ||  token[h-1] > '9' ? 1 : i);
         if (!i)
            j--;
         else
            for (;--i;j++)
               string[j+1] = string[j];
      }
   if (token[h] || string+j >= valueheap+HEAP_SIZE-1)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"too many wordvalues  increase HEAP_SIZE\n");
```

```
      exit(9);
   }
   if (i)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"syntax error  missing value of letter in %s\n",token);
      exit(33);
   }
   string[j] = '\0';
   return  j;
}


unsigned scanlinear( char token[], affine *help )
{  unsigned  h, i, j, number, sgn;
   zero(help);
   sgn = 0;
   number = 0;
   for (h=0;token[h];h++)
   {  if (token[h] >= '0' && token[h] <= '9')
         number *= 10,  number += token[h]-'0';
      else if (token[h]=='+')
      {  if (number)   help->v[0] += (sgn?-number:number),  number=0;
         sgn = 0;
      }
      else if (token[h]=='-')
      {  if (number)   help->v[0] += (sgn?-number:number),  number=0;
         sgn = 1;
      }
      else
      {  if (token[h]<'a' || token[h]>'z')
         {  fprintf(stderr,"line %u: ",lineno);
            fprintf(stderr,"invalid variablename: %c\n",token[h]);
            exit(10);
         }
         if (!(flags&RAW))
         {  for (i=nest;i;i--)
               if  (group[level[i]].var == token[h])
                  break;
         }
         for (j=0;varnames[j] != token[h];j++)
            if (j >= MAX_VARS)
            {  fprintf(stderr,"line %u: ",lineno);
               fprintf(stderr,"too many variables  increase MAX_VARS\n");
               exit(1);
            }
            else if (!varnames[j])
            {  if (!(flags&RAW))
               {  fprintf(stderr,"line %u: ",lineno);
                  fprintf(stderr,"variable %c not defined\n",token[h]);
                  exit(15);
               }
               varnames[j]=token[h];
               varnames[j+1]='\0';
               break;
            }
         if (!(flags&RAW)  &&  !i && j >= freevars)
         {  fprintf(stderr,"line %u: ",lineno);
            fprintf(stderr,"variable %c not active\n",token[h]);
            exit(15);
         }
         if (!h || token[h-1]=='-' || token[h-1]=='+')  number=1;
         help->v[j+1] += (sgn?-number:number);
```

```
        number=0,  sgn=0;
      }
   }
   if (!h || token[h-1]=='-' || token[h-1]=='+')  number=1;
   if (number)  help->v[0] += (sgn?-number:number);
   return  0;
}


bool  get_token_from_input( char token[MAX_TOKEN_LEN] )
{
#define  Tokenstring(len)    "%" Identifier2String(len) "s" "%n"
#define  TOKSTR    Tokenstring(MAX_TOKEN_LEN)
   unsigned  h;
   while (sscanf(inputline+ind,TOKSTR,token,&h)<=0)
   {  lineno++;
      if (!fgets(inputline,MAX_LINE_LEN,fp))
         return  0;
      else
         ind=0;
      for (h=0;inputline[h];h++)
         if (inputline[h]=='\n')  break;
      if (!inputline[h])
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"line too long  increase MAX_LINE_LEN\n");
         exit(37);
      }
   }
   ind+=h;
   if (h >= MAX_TOKEN_LEN)
   {  for (h=0;token[h];h++);
      if (h >= MAX_TOKEN_LEN-1)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"token too long  increase MAX_TOKEN_LEN\n");
         exit(38);
      }
   }
   return  1;
}


void  skip_upto_eol( void )
{  for (;inputline[ind];ind++)
   return;
}


unsigned scanmoves( affine move[] )
{  unsigned  h, i, j;
   char  token[MAX_TOKEN_LEN+2];
   freevars=0;
   for (h=0;h<=MAX_VARS;h++)
      varnames[h]='\0';
   for(condCNT=0; get_token_from_input(token) && token[0]!='{' ;condCNT++)
   {  if (token[0]=='#')
      {  skip_upto_eol();
         condCNT--;
         continue;
      }
      if (token[0]<'a' || token[0]>'z' || token[1])
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"invalid variablename: %s\n",token);
```

```
      exit(10);
   }
   for (h=0;varnames[h] != token[0];h++)
      if (h >= MAX_VARS)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"too many variables  increase MAX_VARS\n");
         exit(1);
      }
      else if (!varnames[h])
      {  varnames[h]=token[0];
         zero(&replace[h]);
         replace[h].v[h+1]=1;
         freevars++;
         break;
      }
   get_token_from_input(token);
   if (token[0] != '>' && token[0] != '<' || token[1] != '=')
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"> or < expected followed by = in token:%s\n", token);
      exit(20);
   }
   i= token[0]=='<';
   get_token_from_input(token);
   scanlinear(token,&condition[condCNT]);
   if (condition[condCNT].v[h+1])
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"variable %c selfdefined\n",varnames[h]);
      exit(16);
   }
   else
      condition[condCNT].v[h+1] = -1;
   if (i)
   {  for (h=0;h<=MAX_VARS;h++)
         condition[condCNT].v[h] = -condition[condCNT].v[h];
   }
}
if (token[1]=='\0')
   get_token_from_input(token);
else
   for (h=0;token[h]=token[h+1];h++);
for (j=i=0;!i;j++)
{  for (h=0;token[h];h++)
      if (token[h]=='}')
      {  token[h]='\0';
         i=h+1;
         break;
      }
   if (!token[0])  break;
   if (j >= MAX_MOVES)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"too many moves  increase MAX_MOVES\n");
      exit(3);
   }
   scanlinear(token,&move[j]);
   if (!i)
      get_token_from_input(token);
}
if (token[i])
{  token[i-1]='}';
   fprintf(stderr,"line %u: ",lineno);
   fprintf(stderr,"pattern after } in token:%s\n",token);
   exit(22);
```

```
   }
return  j;
}


void close_present_block( unsigned no, unsigned m, affine counter )
{  unsigned  h;
   h = group[no].first;
   if (group[no].anz= m - h)
   {  group[no].length= counter;
      decr(&group[no].length,sub[h].addr);
   }
   else
      zero(&group[no].length);
   return;
}


void open_new_block( unsigned no, unsigned m )
{  if (no >= MAX_BLOCKS)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"too many blocks  increase MAX_BLOCKS\n");
      exit(6);
   }
   group[no].first= m;
   group[no].anz= 0;
   group[no].ex= no-1;
   return;
}


void parse_loophead( char token[], unsigned m, unsigned no,
                                   unsigned j, affine counter )
{  unsigned  h;
   close_present_block(no,m,counter);
   open_new_block(++no,m);
   j += get_token_from_input(token);
   if (token[1] || token[0]<'a' || token[0]>'z')
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"invalid variablename: %c\n",token[0]);
      exit(10);
   }
   for (h=0;varnames[h] != token[0];h++)
      if (h >= MAX_VARS)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"too many variables  increase MAX_VARS\n");
         exit(1);
      }
      else if (!varnames[h])
      {  varnames[h]=token[0];
         varnames[h+1]='\0';
         break;
      }
   if (h < freevars)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"free variable %c can't be used as control variable\n",
                    token[0]);
      exit(14);
   }
   group[no].var=token[0];
   zero(&replace[h]);
   replace[h].v[h+1]=1;
```

```
   j += get_token_from_input(token);
   if (!strequ(token,"FROM"))
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"FROM expected\n");
      exit(11);
   }
   j += get_token_from_input(token);
   scanlinear(token,&group[no].start);
   if (group[no].start.v[h+1])
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"variable %c selfdefined\n",varnames[h]);
      exit(16);
   }
   j += get_token_from_input(token);
   if (!strequ(token,"UPTO"))
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"UPTO expected\n");
      exit(12);
   }
   j += get_token_from_input(token);
   scanlinear(token,&group[no].stop);
   if (group[no].stop.v[h+1])
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"variable %c selfdefined\n",varnames[h]);
      exit(16);
   }
}


#define  j   (*J)
bool parse_possible_loop( char token[], unsigned m, unsigned no,
                                        unsigned j, affine counter )
{  unsigned  h, head=0;
   if (strequ(token,"FOR"))
   {  parse_loophead(token,m,no,j,counter);
      ++no;
      j+=5;
      head=1;
      j += get_token_from_input(token);
   }
   if (strequ(token,"REP") || strequ(token,"REPEAT"))
   {  if (!head)
      {  close_present_block(no,m,counter);
         open_new_block(++no,m);
         group[no].var=NOT_EXIST;
         if (!period)
            period=no;
         else
         {  fprintf(stderr,"line %u: ",lineno);
            fprintf(stderr,"several infinity loops given\n");
            exit(28);
         }
      }
      else
         for (h=1;h<=nest;h++)
            if  (group[level[h]].var == group[no].var)
            {  fprintf(stderr,"line %u: ",lineno);
               fprintf(stderr,"control variable %c already in use\n",
                       group[no].var);
               exit(14);
            }
      nest++;
```

```
      if (nest >= MAX_NEST)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"too deep loop nesting  increase MAX_NEST\n");
         exit(7);
      }
      level[nest]=no;
      head=0;
      return  1;
   }
   if (head)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"REPEAT missing\n");
      exit(13);
   }
   if (strequ(token,"PER") || strequ(token,"ENDREP") || strequ(token,"ENDREPEAT"))
   {  close_present_block(no,m,counter);
      if (!nest)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"ENDREP without REPEAT\n");
         exit(17);
      }
      h=level[nest];
      if (group[h].var=='I' || group[h].var=='C' || group[h].var=='E')
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"missing  FI\n");
         exit(18);
      }
      group[ level[nest] ].in= no;
      open_new_block(++no,m);
      group[no].in= group[ level[nest] ].ex;
      nest--;
      group[no].var= group[ level[nest] ].var;
      group[no].stop= group[ level[nest] ].stop;
      group[no].start= group[ level[nest] ].start;
      return  1;
   }
   return  0;
}


void parse_upto_then( unsigned no )
{  unsigned  h;
   char  token[MAX_TOKEN_LEN+2];
   get_token_from_input(token);
   scanlinear(token,&group[no].start);
   get_token_from_input(token);
   if (token[1] != '=' || token[2])
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"= expected as second char in token:%s\n",token);
      exit(20);
   }
   switch (token[0])
   {  case '=':  h= 1;  break;
      case '!':  h= 2;  break;
      case '<':  h= 3;  break;
      case '>':  h= 4;  break;
      default:   fprintf(stderr,"line %u: ",lineno);
                 fprintf(stderr,"> or < or = or ! expected"
                               " followed by = in token %s\n",token);
                 exit(20);
   }
   get_token_from_input(token);
```

```
   scanlinear(token,&group[no].stop);
   if (h != 4)
      decr(&group[no].start,group[no].stop);
   else
   {  decr(&group[no].stop,group[no].start);
      group[no].start = group[no].stop;
      h= 3;
   }
   group[no].stop.v[0]= h;
   get_token_from_input(token);
   if (!strequ(token,"THEN"))
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"THEN expected\n");
      exit(35);
   }
   return;
}


void check_branch_nest_syntax( char type )
{  unsigned  h;
   if (!nest)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"%s without IF\n",
                     (type=='E' ? "ELSE": (type=='C' ? "ELIF":"FI") ) );
      exit(17);
   }
   h=level[nest];
   if (group[h].var=='E' && type!='F')
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"several ELSE\n");
      exit(36);
   }
   if (group[h].var!='I' && group[h].var!='C' && group[h].var!='E')
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"missing  ENDREP\n");
      exit(18);
   }
   return ;
}


bool parse_possible_branch( char token[], unsigned m, unsigned no,
                                          unsigned j, affine counter )
{  unsigned  h=0;
   if (strequ(token,"IF"))
   {  close_present_block(no,m,counter);
      open_new_block(++no,m);
      group[no].var='I';
      parse_upto_then(no);  j+= 4;
      nest++;
      if (nest >= MAX_NEST)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"too deep branch nesting  increase MAX_NEST\n");
         exit(7);
      }
      level[nest]=no;
      return  1;
   }
   if (strequ(token,"ELIF"))
   {  close_present_block(no,m,counter);
      check_branch_nest_syntax('C');
```

```
      open_new_block(++no,m);
      group[ level[nest] ].ex= no;
      group[no].in= level[nest];
      group[no].var='C';
      parse_upto_then(no);  j+= 4;
      level[nest]=no;
      return  1;
   }
   if (strequ(token,"ELSE"))
   {  close_present_block(no,m,counter);
      check_branch_nest_syntax('E');
      open_new_block(++no,m);
      group[ level[nest] ].ex= no-1;
      group[no].in= level[nest];
      group[no].var='E';
      zero(&group[no].start);
      group[no].stop.v[0]= 1;
      level[nest]=no;
      return  1;
   }
   if (strequ(token,"FI") || strequ(token,"ENDIF"))
   {  close_present_block(no,m,counter);
      check_branch_nest_syntax('F');
      open_new_block(++no,m);
      group[level[nest]].ex= no-1;
      group[no].in= level[nest];
      nest--;
      h=no;
      do {
         unsigned  hh= group[h].in;
         group[h].in= group[h].ex+1;
         h=hh;
      } while (group[h].var!='I');
      group[h].in= group[h].ex+1;
      group[no].ex= group[no].in= h;
      return  1;
   }
   return  0;
}
#undef  j


void parse_let_statment( char token[] )
{  unsigned  h;
   if (token[1] != '=')
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"= expected as second char in token:%s\n",token);
      exit(20);
   }
   if (token[0]<'a' || token[0]>'z')
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"invalid variablename: %c\n",token[0]);
      exit(10);
   }
   if (!(flags&RAW))
      for (h=1;h<=nest;h++)
         if  (group[level[h]].var == token[0])
         {  fprintf(stderr,"line %u: ",lineno);
            fprintf(stderr,"control variable %c already in use\n",token[0]);
            exit(14);
         }
   for (h=0;varnames[h] != token[0];h++)
```

```
      if (h >= MAX_VARS)
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"too many variables  increase MAX_VARS\n");
         exit(1);
      }
      else if (!varnames[h])
      {  varnames[h]=token[0];
         varnames[h+1]='\0';
         break;
      }
   if (!(flags&RAW)  &&  h < freevars)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"free variable %c can't be used as control variable\n",
                     token[0]);
      exit(14);
   }
   scanlinear(token+2,&replace[h]);
   return ;
}


bool  parse_word( char token[], unsigned m , affine *counter )
{  unsigned  h, k;
   affine  help;
   if (m == MAX_WORDS)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"too many words  increase MAX_WORDS\n");
      exit(2);
   }
   if (token[0] == '(' || token[0] == '<')
   {  for (k=1;token[k];k++)
         if (token[k]==')' || token[k]=='>')
            break;
      if (token[k]!=')' && token[k]!='>')
      {  fprintf(stderr,"line %u: ",lineno);
         fprintf(stderr,"missing > in token:%s\n",token);
         exit(19);
      }
      token[k]='\0';
      sub[m].value= next_free;
      h=scanword(token+1,sub[m].value);
      next_free += h+1;
      h=k+1;
   }
   else
   {  h = (unsigned) token[1];
      token[1]='\0';
      sub[m].value= next_free;
      scanword(token,sub[m].value);
      next_free += 1+1;
      token[1]= (char) h;
      h=1;
   }
   if (!token[h])
   {  zero(&help);
      help.v[0]= 1;
   }
   else
   {  if (token[h] == '(')
      {  h++;
         for (k=h;token[k];k++)
            if (token[k]==')')
```

```
            break;
         if (token[k]!=')')
         {  fprintf(stderr,"line %u: ",lineno);
            fprintf(stderr,"missing ) in token:%s\n",token);
            exit(19);
         }
         else if (token[k+1])
         {  fprintf(stderr,"line %u: ",lineno);
            fprintf(stderr,"pattern after ) in token:%s\n",token);
            exit(22);
         }
         else
            token[k]='\0';
      }
      scanlinear(token+h,&help);
   }
   for (k=1;k<=MAX_VARS;k++)
      substitute(&help,k,replace[k-1]);
   if (is_zero(help))
      return  0;
   else
   {  sub[m].addr = *counter;
      if (token[0]=='(')
      {  for(h=0; *(sub[m].value+h) ; h++);
         multiply(&help,h);
      }
      incr(counter, help);
      sub[m].leng = help;
      return  1;
   }
}


unsigned parse_grammar( void )
{  unsigned  h, j, k, m, no;
   char  token[MAX_TOKEN_LEN+2];
   affine  counter;
   j=0;
   if (!(moveCNT=scanmoves(move)))
   {  fprintf(stdout,"no moves given. no check\n");
      exit(0);
   }
   else if (moveCNT > MAX_MOVES)
   {  fprintf(stderr,"line %u: ",lineno);
      fprintf(stderr,"too many moves  increase MAX_MOVES\n");
      exit(3);
   }
   else if (flags&VERBOSE)
   {  if (flags&DUMP)
      {  printf("{ ");
         for (h=0;h<moveCNT;h++)
         {  if (h)  printf(" , ");
            printaffine(move[h]);
         }
         printf(" }\n");
      }
      printf("%u moves read\n",moveCNT);
   }
   if (!(flags&RAW))
   {  period=0;
      nest=0;
   }
```

```
open_new_block( no=1 , 0);
group[no].in= 0;
group[no].var= NOT_EXIST;
level[nest]=no;
zero(&counter);
if (flags&RAW)
   for (h=0;h<MAX_VARS;h++)
   {  zero(&replace[h]);
      replace[h].v[h+1]=1;
   }
for (m=0; get_token_from_input(token) ; )
{  j++;
   sub[m].addr= counter;
   if (token[0]=='#')
   {  skip_upto_eol();
      continue;
   }
   if (!(flags&RAW))
   {  if (parse_possible_loop(token,m,no,&j,counter))
      {  no++;
         continue;
      }
      if (parse_possible_branch(token,m,no,&j,counter))
      {  no++;
         continue;
      }
   }
   if (strequ(token,"LET"))
   {  j += get_token_from_input(token);
      parse_let_statment(token);
      continue;
   }
   if (parse_word(token,m,&counter))
      m++;
}
sub[m].addr = counter;
close_present_block(no,m,counter);
no++;
if (nest)
{  h=level[nest];
   fprintf(stderr,"at EOF: ",lineno);
   if (group[h].var=='I' || group[h].var=='C' || group[h].var=='E')
      fprintf(stderr,"missing  FI\n");
   else
      fprintf(stderr,"missing  ENDREP\n");
   exit(18);
}
if (!period)
{  fprintf(stderr,"at EOF: ",lineno);
   fprintf(stderr,"no infinity loop given\n");
   exit(28);
}
if (flags&VERBOSE)
{  fprintf(stdout,"%u lines with %u token scaned\n",lineno-1,j);
   if (!(flags&RAW))
   {  k=0;
      for (h=1;h<no;h++)
         k += !!group[h].anz;
      fprintf(stdout,"%u(%u) blocks and ",k,no-1);
   }
   fprintf(stdout,"%u words processed",m);
   if (flags&RAW)
```

```
      {  fprintf(stdout,"  length=");
         printaffine(counter);
      }
      fprintf(stdout,", heapallocation= %ld bytes\n",next_free-valueheap);
   }
   return  (flags&RAW ? m : no);
}


unsigned build_back_list( word *sgv, unsigned k, word back[MAX_SPLITS] )
{  unsigned  h, i, j, n=0;
   affine  aim, help;
   for (j=0;j<moveCNT;j++)
   {  aim = sgv[k].addr;
      decr(&aim,move[j]);
      for (i=k;;i--)
      {  if (cmp(sgv[i].addr,aim) <= 0)
         {  if (n >= MAX_SPLITS)
            {  fprintf(stderr,"heap overflow  increase MAX_SPLITS\n");
               exit(4);
            }
            zero(&back[n].addr);
            back[n].value=sgv[i].value;
            help = aim;
            decr(&help,sgv[i].addr);
            h = cyclelength(sgv[i].value);
            back[n].value= shift_cycle(back[n].value, mod(help,h));
            back[n].leng = sgv[k].leng;
            n++;
            help = aim;
            incr(&aim,sgv[k].leng);
            while (cmp(aim,sgv[++i].addr) > 0)
            {  if (n >= MAX_SPLITS)
               {  fprintf(stderr,"heap overflow  increase MAX_SPLITS\n");
                  exit(4);
               }
               back[n].leng= back[n-1].leng;
               back[n-1].leng= sgv[i].addr;
               decr(&back[n-1].leng,help);
               back[n].addr=back[n-1].addr;
               incr(&back[n].addr,back[n-1].leng);
               back[n].value=sgv[i].value;
               decr(&back[n].leng,back[n-1].leng);
               help = sgv[i].addr;
               n++;
            }
            break;
         }
         else if (!i)
         {  if (flags&DUMP)
               for (;k--;)
               {  printf("index=%u  addr=",k);
                  dumpword(sgv[k]);
                  printf("\n");
               }
            catcher[1] = (jmp_buf *) 0;
            return  0;
         }
      }
   }
   return  n;
}
```

```
unsigned align_back_list( word back[MAX_SPLITS], unsigned n )
{  unsigned  h, i, j, k;
   affine  help;
   for (i=0;i<n;i++)
   {  if (!unequal_zero(back[i].leng))
         back[i--]=back[--n];
   }
   for (i=0;i<n;i++)
   {  for (j=i+1;j<n;j++)
      {  if (cmp(back[i].addr,back[j].addr) > 0)
         {  word  temp=back[i];  back[i]=back[j];  back[j]=temp;  }
      }
   }
   for (i=0;i<n;i=j)
   {  for (j=i+1;j<n;j++)
         if (cmp(back[j].addr,back[i].addr) > 0)
            break;
      if (j < n)
      {  k=j;
         for (;i<j;i++)
         {  help = back[i].addr;
            incr(&help,back[i].leng);
            if (cmp(help,back[j].addr) > 0)
            {  if (n >= MAX_SPLITS)
               {  fprintf(stderr,"heap overflow  increase MAX_SPLITS\n");
                  exit(4);
               }
               for (h=n;h>k;h--)
                  back[h] = back[h-1];
               n++;
               back[i].leng = back[j].addr;
               decr(&back[i].leng,back[i].addr);
               back[k].value = back[i].value;
               h = cyclelength(back[i].value);
               back[k].value= shift_cycle(back[k].value, mod(back[i].leng,h));
               back[k].addr = back[j].addr;
               back[k].leng = help;
               decr(&back[k].leng,back[k].addr);
            }
            k++;  /* stabil order of moves */
         }
      }
   }
   return  n;
}


unsigned check_split( char *value_ptr[], word orig, unsigned index, word back[],
                      unsigned i, unsigned err )
{  unsigned  h, j, k;
   char  predicted;
   for(k=h=0;unequal_zero(back[i].leng);)
   {  predicted = calculate_value(value_ptr);
      if (predicted != *value_ptr[moveCNT])
      {  fprintf(stderr,"Error at index=%u   %u. split  ", index,(i+1)/moveCNT);
         dumpword(orig), printf("\n");
         for (j=0;j<moveCNT;j++)
            fprintf(stderr,"%c",*value_ptr[j]);
         fprintf(stderr," -> %c    but %c expected\n",
                 predicted,*value_ptr[moveCNT]);
```

```
         ++err;
      }
      for (j=0;j<=moveCNT;j++)
         if (! * ++value_ptr[j])
         {  while ( * --value_ptr[j] );
               value_ptr[j]++;
         }
      h++;
      back[i].leng.v[0]--;
      if (! --k)
         break;
      for (j=0;j<moveCNT;j++)
         if (value_ptr[j] != back[i-moveCNT+1+j].value)
            break;
      if (j==moveCNT && h >= cyclelength(orig.value))
      {  if (h % cyclelength(orig.value))
         {  fprintf(stderr,"congruence MOD %u is decisive\n",
                    cyclelength(orig.value));
            exit(34);
         }
         else if (!(k=mod(back[i].leng, h)))
            break;
      }
   }
   back[i].leng.v[0] += h;
   return  err;
}


#define  err  (*ERR)
bool check_word( word *sgv, unsigned k, unsigned err)
{  unsigned  h, i, j, n;
   word  back[MAX_SPLITS];
   char  * value_ptr[MAX_MOVES+1];
   if (!(n=build_back_list(sgv,k,back)))
      return  1;
   n= align_back_list(back,n);
   value_ptr[moveCNT] = sgv[k].value;
   for (j=i=0;i<n;i++)
   {  value_ptr[j] = back[i].value;
      j++;
      if (j==moveCNT)
      {  err= check_split(value_ptr,sgv[k],k,back,i,err);
         j=0;
      }
      else if (cmp(back[i].addr,back[i+1].addr))
      {  fprintf(stderr,"variables in wrong lexicographical order\n");
            exit(25);
      }
      else if (cmp(back[i].leng,back[i+1].leng))
      {  fprintf(stderr,"internal ERROR  different length\n");
         exit(29);
      }
   }
   if (flags&DUMP)
   {  for (h=j=0;j<n;j++)
      {  if (!(j%moveCNT))
         {  printaff(back[j].addr);
            printf(":  ");
         }
         h+= printword( back[j] );
         if ((j+1)%moveCNT)
```

```
         {  for (;h<SPACING;h++)
               putchar( (int)' ' );
            h -= SPACING;
         }
         else
         {  putchar( (int)'\n' );
            h=0;
         }
      }
   }
   return 0;
}
#undef  err


int check_transition( word *sgv, unsigned m )
{  unsigned  h, i, j, k, err, pending;
   unsigned  deep, multi[3][MAX_MOD_DEEP];
   int  fac;
   err=0;
   catcher[1] = &catcher_environment[1];
   for (k=m;k--;)
   {  if (flags&DUMP)
      {  printf("index=%u  addr=",k);
         dumpword(sgv[k]);
         printf("\n");
      }
/********** MOD catcher *********/
      pending=deep=0;
      do {
         while ((h=setjmp(catcher_environment[1])) || check_word(sgv,k,&err))
         {  if (!h)
               return  err;
            else if (!(h>>VBITS))
            {  pending = 1;
               break;
            }
            if (deep == MAX_MOD_DEEP)
            {  fprintf(stderr,"too many congruences  increase MAX_MOD_DEEP\n");
               exit(32);
            }
            multi[0][deep] = j= h & (1<<VBITS)-1;
            multi[1][deep] = fac= h>>VBITS;
            multi[2][deep] = 0;
            deep++;
            for (i=0;i<=k;i++)
            {   sgv[i].addr.v[j] *= fac;
               sgv[i].leng.v[j] *= fac;
            }
            for (i=0;i<condCNT;i++)
               condition[i].v[j] *= fac;
            if (!conditions_consistent( (affine*)0 ))
               break;
         }
         h=deep;
         while (h--)
         {  j=multi[0][h];
            fac= multi[1][h];
            multi[2][h]++;
            if (pending  ||  multi[2][h] == multi[1][h])
            {  multi[2][h]--;
               for (i=0;i<=k;i++)
```

```
            {  sgv[i].addr.v[j] /= fac;
               sgv[i].addr.v[0] -= (int)multi[2][h] * sgv[i].addr.v[j];
               sgv[i].leng.v[j] /= fac;
               sgv[i].leng.v[0] -= (int)multi[2][h] * sgv[i].leng.v[j];
            }
            for (i=0;i<condCNT+pending;i++)
            {  condition[i].v[j] /= fac;
               condition[i].v[0] -= (int)multi[2][h] * condition[i].v[j];
            }
            conditions_consistent( (affine*)0 );
            deep--;
         }
         else
         {  for (i=0;i<=k;i++)
            {  sgv[i].addr.v[0] += sgv[i].addr.v[j] / fac;
               sgv[i].leng.v[0] += sgv[i].leng.v[j] / fac;
            }
            for (i=0;i<condCNT;i++)
               condition[i].v[0] += condition[i].v[j] / fac;
            if (conditions_consistent( (affine*)0 ))
               break;
            else
               h++;
         }
      }
   } while (deep);
   if (pending)
      longjmp(*catcher[0], cmp_mark);
/********* END OF MOD catcher *********/
   }
   return  (int) ~(~(unsigned)0 >> 1);
}


#define  safe (*SAFE)
unsigned eliminate_empty_words( unsigned m, word *sgv, word *safe )
{  unsigned  h, i, j, k;
   if (!conditions_consistent((affine*)0))
   {  fprintf(stderr,"illegal call:  conditions inconsistent\n");
      return  0;
   }
   i=0;
   safe= (word*)0;
   for (h=k=m;k--;)
   {  zero(&condition[condCNT]);
      decr(&condition[condCNT],sgv[k].leng);
      condition[condCNT].v[0] += 1;
      if (!conditions_consistent(&condition[condCNT]))
      {  if (!safe)
         {  if (!(safe=(word*)malloc(m * sizeof(word))))
            {  fprintf(stderr,"malloc: not enough space\n");
               exit(30);
            }
            for (j=0;j<m;j++)
               *(safe+j) = sgv[j];
         }
      }
      else
      {  sgv[--h] = sgv[k];
         i++;
      }
   }
```

```
   return  m-i;
}
#undef  safe


unsigned analyze_propagation( word *sgv, unsigned m )
{  unsigned  h, i, j, k;
   int  ret;
   word  *safe;
   unsigned  condCNT_orig, cases;
   if (!(flags&RAW))
      if (i= eliminate_empty_words(m,sgv,&safe))
         sgv += i,  m -= i;
   if (flags&DUMP)
      for (k=m+1;k--;)
         decr(&sgv[k].addr, sgv[0].addr);
   condCNT_orig=condCNT;
   k=0;
   cases=0;
   if (flags&VERBOSE)
      printf("%u. transition:\n",trans);
/******** RELATION catcher *********/
   catcher[0] = &catcher_environment[0];
   do {
      while (setjmp(catcher_environment[0]) || (ret=check_transition(sgv,m))<0)
      {  if (condCNT == MAX_COND)
         {  fprintf(stderr,"too many conditions  increase MAX_COND\n");
            exit(8);
         }
         for (h=1;h<=MAX_VARS;h++)
            if (condition[condCNT].v[h])
               break;
         if (condition[condCNT].v[h] > 0)
         {  for (h=0;h<=MAX_VARS;h++)
               condition[condCNT].v[h]= -condition[condCNT].v[h];
         }
         condCNT++;
         if (!conditions_consistent((affine*)0))
         {  fprintf(stderr,"internal ERROR  consistence\n");
            exit(29);
         }
      }
      cases++;
      k += ret;
      while (condCNT > condCNT_orig)
      {  condCNT--;
         for (h=1;h<=MAX_VARS;h++)
            if (condition[condCNT].v[h])
               break;
         if (condition[condCNT].v[h] < 0)
         {  for (h=0;h<=MAX_VARS;h++)
               condition[condCNT].v[h]= -condition[condCNT].v[h];
            condCNT++;
            break;
         }
      }
      if (!conditions_consistent((affine*)0))
      {  fprintf(stderr,"internal ERROR  consistence\n");
         exit(29);
      }
   } while (condCNT > condCNT_orig);
   catcher[0] = (jmp_buf *) 0;
```

```
/******** END OF RELATION catcher *********/
   if (!k && (flags&VERBOSE))
      printf("consistent\n");
   if (cases>1 && (flags&VERBOSE))
      printf("%u subcases\n",cases);
   if (!(flags&RAW))
      total += cases;
   if (!(flags&RAW) && safe)
   {  sgv -= i;
      m += i;
      for (j=0;j<m;j++)
         sgv[j] = *(safe+j);
      free(safe);
   }
   return  k;
}


unsigned prepend_infinity_block( unsigned k, affine *counter )
{  if (!k--)
   {  fprintf(stderr,"list overflow  increase MAX_SIZE\n");
      exit(5);
   }
   list[k].addr = list[k+1].addr;
   list[k].leng = *counter;
   list[k].value= &valueheap[1];
   decr(&list[k].addr, *counter);
   zero(counter);
   return  k;
}


unsigned prepend_this_block( unsigned k, unsigned no, affine *counter )
{  unsigned  h, i;
   i = group[no].first;
   for (h = group[no].anz;h--;)
   {  if (!k--)
      {  fprintf(stderr,"list overflow  increase MAX_SIZE\n");
         exit(5);
      }
      list[k] = sub[group[no].first+h];
      for (i=1;i<=MAX_VARS;i++)
         substitute(&list[k].leng,i,replace[i-1]);
      if (!is_zero(list[k].leng))
      {  list[k].addr = list[k+1].addr;
         condition[condCNT]= list[k].leng;
         condition[condCNT].v[0] += 1;
         if (conditions_consistent(&condition[condCNT]))
         {  if (flags&VERBOSE)
               printf("%s\n",protocol);
            fprintf(stderr,"word can have negative length\n");
            dumpword(list[k]);
            exit(24);
         }
         decr(&list[k].addr, list[k].leng);
         decr(counter, list[k].leng);
         if (!(flags&DUMP))
            if (-unequal_zero(*counter) >= 0)
               break;
      }
      else
         k++;
```

```
   }
   return  k;
}


unsigned prepend_further_block(); /* Must be declared before used */

unsigned go_back_one_iteration( unsigned k, unsigned no, affine counter )
{  unsigned  h, m;
   if (group[no].in)
   {  if (group[no].var != NOT_EXIST)
      {  if (flags&VERBOSE)
            sprintf(next(protocol),"%c -> %c - 1\n",group[no].var,group[no].var);
         for (h=0;varnames[h] != group[no].var;h++);
         replace[h].v[0] -= 1;
         condition[group[no].active].v[0] += 1;
      }
      else if (flags&VERBOSE)
         sprintf(next(protocol)," \n");
      if (conditions_consistent((affine*)0))
         m = prepend_further_block(k, group[no].in, counter);
      if (group[no].var != NOT_EXIST)
      {  replace[h].v[0] += 1;
         condition[group[no].active].v[0] -= 1;
      }
   }
   else if (flags&VERBOSE)
   {  sprintf(next(protocol),"  \n");
      m=0;
   }
   return  m;
}


unsigned leave_loop_through_head( unsigned k, unsigned no, affine counter )
{  unsigned  h, i, m;
   affine  help;
   int  *factor;
   if (group[no].var != NOT_EXIST)
   {  for (h=1;varnames[h-1] != group[no].var;h++);
      if (flags&VERBOSE)
      {  sprintf(last(next(protocol)-2)+1,"retroactive:  ");
         sprintaffine(next(protocol),replace[h-1]);
         sprintf(next(protocol)," %s ", (replace[h-1].v[h] ? "->" : ">=") );
         sprintaffine(next(protocol),group[no].start);
         sprintf(next(protocol),"\n");
      }
      if (replace[h-1].v[h] == 1)
      {  help=group[no].start;
         decr(&help,replace[h-1]);
         help.v[h] = 0;
         if (!(factor=(int*)malloc((2*(MAX_SIZE-k)+condCNT)*sizeof(int))))
         {  fprintf(stderr,"malloc: not enough space\n");
            exit(30);
         }
         for (i=k;i<MAX_SIZE;i++)
         {  *factor++ = substitute(&list[i].addr,h,help);
            *factor++ = substitute(&list[i].leng,h,help);
         }
         for (i=0;i<condCNT;i++)
            *factor++ = substitute(&condition[i],h,help);
         factor -= 2*(MAX_SIZE-k)+condCNT;
```

```
      }
      else if (replace[h-1].v[h])
      {  if (flags&VERBOSE)
            printf("%s\n",protocol);
         fprintf(stderr,"internal ERROR  variable substitution\n");
         exit(29);
      }
      else
      {  h=0;
         if (condCNT == MAX_COND)
         {  fprintf(stderr,"too many conditions  increase MAX_COND\n");
            exit(8);
         }
         zero(&condition[condCNT]);
         decr(&condition[condCNT],condition[group[no].active]);
         condCNT++;
      }
   }
   else if (flags&VERBOSE)
      sprintf(last(next(protocol)-2)+1,"  \n");
   if (conditions_consistent((affine*)0))
      m = prepend_further_block(k, group[no].ex, counter);
   if (group[no].var != NOT_EXIST)
   {  if (h)
      {  for (i=k;i<MAX_SIZE;i++)
         {  desubstitute(&list[i].addr,h,*factor,help);
            factor++;
            desubstitute(&list[i].leng,h,*factor,help);
            factor++;
         }
         for (i=0;i<condCNT;i++)
         {  desubstitute(&condition[i],h,*factor,help);
            factor++;
         }
         factor -= 2*(MAX_SIZE-k)+condCNT;
         free(factor);
      }
      else
         condCNT--;
   }
   return  m;
}


unsigned skip_previous_block( unsigned k, unsigned no, affine counter )
{  unsigned  h, i, m;
   if (i=group[no].ex)
   {  if (condCNT == MAX_COND)
      {  fprintf(stderr,"too many conditions  increase MAX_COND\n");
         exit(8);
      }
      condition[condCNT] = group[i].stop;
      decr(&condition[condCNT], group[i].start);
      condition[condCNT].v[0] += 1;
      for (h=1;h<=MAX_VARS;h++)
         substitute(&condition[condCNT],h,replace[h-1]);
      condCNT++;
      if (flags&VERBOSE)
      {  sprintaffine(next(protocol),group[i].start);
         sprintf(next(protocol)," > ");
         sprintaffine(next(protocol),group[i].stop);
         sprintf(next(protocol),"\n");
```

```
      }
   }
   else if (flags&VERBOSE)
      sprintf(next(protocol),"\n");
   if (conditions_consistent((affine*)0))
      m = prepend_further_block(k, group[no].in, counter);
   else
      m = 0;
   if (i)
      condCNT--;
   return  m;
}


unsigned enter_block_from_its_end( unsigned k, unsigned no, affine counter )
{  unsigned  h, i, m;
   if (i=group[no].ex)
   {  if (condCNT == MAX_COND)
      {  fprintf(stderr,"too many conditions  increase MAX_COND\n");
         exit(8);
      }
      zero(&condition[condCNT]);
      decr(&condition[condCNT],group[i].start);
      for (h=1;h<=MAX_VARS;h++)
         substitute(&condition[condCNT],h,replace[h-1]);
      condCNT++;
      condition[condCNT]= group[i].start;
      decr(&condition[condCNT],group[i].stop);
      for (h=1;h<=MAX_VARS;h++)
         substitute(&condition[condCNT],h,replace[h-1]);
      group[i].active=condCNT;
      condCNT++;
      if (flags&VERBOSE)
      {  sprintf(last(next(protocol)-2)+1,"%c -> ",group[i].var);
         sprintaffine(next(protocol),group[i].stop);
         sprintf(next(protocol),"\n");
      }
      for (h=0;varnames[h] != group[i].var;h++);
      replace[h] = group[i].stop;
      if (conditions_consistent((affine*)0))
         m = prepend_further_block(k, group[no].ex, counter);
      else
         m = 0;
      condCNT-=2;
   }
   else
      m=0;
   return  m;
}


unsigned prepend_further_block( unsigned k, unsigned no, affine counter )
{  unsigned  m=0;
   if (flags&VERBOSE)
      sprintf(next(protocol),"block %u  \t",no);
   if (!no)
      k=prepend_infinity_block(k,&counter);
   else
      k=prepend_this_block(k,no,&counter);
   if (-unequal_zero(counter) >= 0)
   {  list[MAX_SIZE].addr = list[MAX_SIZE-1].addr;
      incr(&list[MAX_SIZE].addr, list[MAX_SIZE-1].leng);
```

```
      if (flags&VERBOSE)
         printf("%s\n",protocol);
      trans++;
      if (!(flags&PARSE))
         m += analyze_propagation(list+k,MAX_SIZE-k);
   }
   else
   {  if (group[no].in >= no)
      {  m+= go_back_one_iteration(k,no,counter);
         m+= leave_loop_through_head(k,no,counter);
      }
      else
      {  m+= skip_previous_block(k,no,counter);
         m+= enter_block_from_its_end(k,no,counter);
      }
      if (flags&VERBOSE)
         *(last(next(protocol)-2)+1) = '\0';
   }
   if (flags&VERBOSE)
      *(last(next(protocol)-2)+1) = '\0';
   return  m;
}


void show_grammar( unsigned no )
{  unsigned  h, j, k;
   char  line[MAX_LINE_LEN+1];
   for (k=0;k<MAX_LINE_LEN;k++)
      line[k]=' ';
   line[k]='\0';
   for (k=0,h=1;h<no;h++)
   {  if (group[h].ex < group[h].in)
      {  if (group[h].var == 'C' || group[h].var == 'E')
            k--;
         printf("%s",line+MAX_LINE_LEN-3*k);
         if (group[h].var == 'I' || group[h].var == 'C')
         {  if (group[h].var == 'C')
               printf("EL");
            printf("IF "), printaffine( group[h].start );
            if (group[h].stop.v[0] == 1)
               printf(" == 0 ");
            else if (group[h].stop.v[0] == 2)
               printf(" != 0 ");
            else
               printf(" <= 0 ");
            printf("THEN\n");
         }
         else if (group[h].var == 'E')
            printf("ELSE\n");
         else if (group[h].var != NOT_EXIST)
         {  printf("FOR %c",group[h].var);
            printf(" FROM "), printaffine( group[h].start );
            printf(" UPTO "), printaffine( group[h].stop );
            printf(" REP\n");
         }
         else
            printf("REPEAT\n");
         k++;
      }
      else if (group[h].in)
      {  k--;
         printf("%s",line+MAX_LINE_LEN-3*k);
```

```
         if (group[h].in == group[h].ex)
            printf("FI\n");
         else if (k)
            printf("PER\n");
         else
            printf("ENDREP\n");
      }
      if (group[h].anz)
      {  printf("%s",line+MAX_LINE_LEN-3*k);
         printf("%u. block:   %u words   length=", h, group[h].anz);
         printaffine( group[h].length );
         printf("\n");
         if (flags&DUMP)
            for (j=0;j<group[h].anz;j++)
            {  dumpword( sub[group[h].first + j] );
               printf("\n");
            }
      }
   }
}


int  generate_transitions( unsigned no )
{  unsigned  h, j, k;
   affine  counter;
   trans=0;
   j=0;
   for (k=1;k<no;k++)
   {  if (group[k].var=='I')
      {
      }
      else if (group[k].var=='C' || group[k].var=='E')
         continue;
      else if (group[k].ex < group[k].in)
      {  if (group[k].var != NOT_EXIST)
         {  zero(&condition[condCNT]);
            decr(&condition[condCNT],group[k].start);
            condCNT++;
            group[k].active = condCNT;
            for (h=freevars;varnames[h]!=group[k].var;h++);
            condition[condCNT] = group[k].start;
            condition[condCNT].v[h+1] = -1;
            condCNT++;
            zero(&condition[condCNT]);
            decr(&condition[condCNT],group[k].stop);
            condition[condCNT].v[h+1] = 1;
            condCNT++;
         }
      }
      else if (group[k].in)
         condCNT -= 3;
      if (group[k].anz)
      {  if (flags&VERBOSE)
            sprintf(protocol,"\n");
         for (h=1;h<=MAX_VARS;h++)
         {  zero(&replace[h-1]);
            replace[h-1].v[h]=1;
         }
         counter = group[k].length;
         incr(&counter,move[moveCNT-1]);
         if (conditions_consistent((affine*)0))
            j += prepend_further_block(MAX_SIZE, k, counter);
```

```
         else
            fprintf(stderr,"WARNING  block %u never active\n",k);
      }
   }
   if (flags&VERBOSE)
      printf("%u transitions with %u cases checked\n",trans,total);
   return  j;
}


unsigned  init_valuestring_and_infinity_block( void )
{  unsigned  i;
   next_free= valueheap;
   *next_free++ ='\0';
   if (flags&WL)
   {  for (i=0;i<2;i++)
         valuestring[i] = (i ? '+' : '-');
      *next_free++ = valuestring[1];
      *next_free++ = valuestring[2] = '\0';
   }
   else if (flags&SPECIAL)
   {  if (!(i=define_values(valuestring,MAX_MOVES)))
      {  fprintf(stderr,"internal ERROR  define_values\n");
         exit(29);
      }
      *next_free++ = valuestring[0];
      *next_free++ = valuestring[i] = '\0';
   }
   else
   {  for (i=0;i<=MAX_MOVES;i++)
         valuestring[i] = (i ? 'a'+i-1 : '_');
      *next_free++ = valuestring[MAX_MOVES];
      *next_free++ ='\0';
   }
   return  i;
}


int main( unsigned argc, char *argv[] )
{  unsigned  h, i, no;
   if (MOST != -MOST)
   {  fprintf(stderr,"internal ERROR  need 2-complement representation\n");
      exit(29);
   }
   flags=0;
   for (i=1; i < argc  &&  *argv[i]=='-' ;i++)
   {  if (*(argv[i]+1) == '-' && !*(argv[i]+2))
      {  i++;
         break;
      }
      else if (*(argv[i]+1) == '+' && !*(argv[i]+2))
         flags |= WL;
      else if (*(argv[i]+1) == 'r' && !*(argv[i]+2))
         flags |= RAW;
      else if (*(argv[i]+1) == 't' && !*(argv[i]+2))
         flags |= PARSE;
      else if (*(argv[i]+1) == 'v' && !*(argv[i]+2))
         flags |= VERBOSE;
      else if (*(argv[i]+1) == 'X')
      {  flags |= SPECIAL;
         if (!*(argv[i]+2))
            threshold=0;
```

```
      else if (1!=sscanf(argv[i]+2,"%u",&threshold))
         fprintf(stderr,"invalid number for option X\n");
   }
   else if (*(argv[i]+1) == 'D' && !*(argv[i]+2))
      flags |= DUMP;
   else
   {  fprintf(stderr,"unknown option %s\n",argv[i]);
      fprintf(stdout,"usage: %s [-rtvX[no]D+] filename\n",argv[0]);
      exit(26);
   }
}
if (i >= argc)
{  fprintf(stdout,"usage: %s [-rtvX[no]D+] filename\n%s\n",argv[0],VERSION);
   fprintf(stdout,tot(MAX_WORDS)"  "tot(MAX_SIZE)"  "tot(MAX_SPLITS)"  "
                  tot(MAX_BLOCKS)"  "tot(HEAP_SIZE)"\n");
   fprintf(stdout,tot(MAX_MOVES)"  "tot(MAX_VARS)"  "tot(MAX_COND)"  "
                  tot(MAX_TOKEN_LEN)"  "tot(MAX_NEST)"\n");
   fprintf(stdout,tot(MAX_MOD_DEEP)"  "tot(MAX_PROTO)"  "tot(MAX_LINE_LEN)"  "
                  tot(SPACING)"  "tot(VBITS)"\n");
   exit(0);
}
if (!(fp=fopen(argv[i],"r")))
{  fprintf(stderr,"can't open %s\n",argv[i]);
   exit(27);
}
else
   inputline[ind=0]= '\0';
init_permutation();
init_valuestring_and_infinity_block();
no=parse_grammar();
fclose(fp);
if (flags&VERBOSE)
{  varnames[MAX_VARS]=varnames[freevars];
   varnames[freevars]='\0';
   if (varnames[0])
      printf("free variables=%s   ",varnames);
   varnames[freevars]=varnames[MAX_VARS];
   varnames[MAX_VARS]='\0';
   if (varnames[freevars])
      printf("control variables=%s\n",varnames+freevars);
   else
      printf("no control variables\n");
}
if (!(flags&RAW))
{  if (flags&VERBOSE)
      show_grammar(no);
   for (h=1;h<no;h++)
   {  if (group[h].ex < group[h].in  &&  group[h].anz &&
            group[h].var!='I' && group[h].var!='C' && group[h].var!='E')
      {  for (i=0;i<freevars;i++)
            if (!group[h].length.v[i+1])
            {  fprintf(stderr,"length of loopbody is independent of %c\n",
                                 varnames[i]);
               exit(23);
            }
      }
   }
}
if (!(flags&RAW))
   return  -generate_transitions(no);
else if (!(flags&PARSE))
   return  -analyze_propagation(sub,no);
```

```
   else
      return  0;
}
```

Beispielbeschreibung

```
# L=(n,2n+1,4n+2,5n+3,6n+3)    p = n(10n+4)    n >= 1
#v = (10n+4)(n + (n+(n%2)-6)/2) + 3n + (n%2) - 1    n >= 5

#n >= 3  {n 2n+1 4n+2 5n+3 6n+3}
 m >= 3  {2m-1 4m-1 8m-2 10m-2 12m-3}

   # j=0
   _(2m-1)  A(2m-1)
   _1  B(2m-2) A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  D(2m-1)
   C1  E1 B1 E(2m-4) D1
   C(2m-1)  D1 B1
   # j=1
   _(2m-1)  A(2m-1)
   _1  E1 B(2m-3) A1
   _(2m-1)  A(2m-1)  B1 D1 B(2m-3)
   C(2m-1)  D2 B1 D1 D(2m-5)
   C1  E1 D1 B1 E(2m-5) D1
   C(2m-1)  D1 B1
FOR  j FROM 2 UPTO 2m-4  REP
   _(2m-1)  A(2m-1)
   _1  B1 <DE>(j-1) B(2m-2-j) A1
   _(2m-1)  A(2m-1)  B1 D1 B(2m-3)
   C(2m-1)  D2 B1 <DE>(j) D(2m-4-j)
   C1  E1 D1 E1 B(j-1) E(2m-4-j) D1
   C(2m-1)  D1 B1
PER
# j=2m-3
   _(2m-1)  A(2m-1)
   _1  B1 <DE>(2m-4) B(1) A1
   _(2m-1)  A(2m-1)  B1 D1 B(2m-3)
   C(2m-1)  D2 B1 <DE>(2m-4)
   C1  E1 D1 E1 B(2m-4)
   C(2m-1)  D1 B1
# j=2m-2
   _(2m-1)  A(2m-1)
   _1  B1 <DE>(2m-3) A1
   _(2m-1)  A(2m-1)  B1 D1 B(2m-3)
   C(2m-1)  B1 D1 B1 <DE>(2m-4)
   C1  B1 D1 E1 B(2m-4)
   C(2m-1)  D2
FOR  j FROM 2m-1 UPTO 3m-3  REP
   _(2m-1)  A(2m-1)
   _1  E(j-2m+1) B1 E(j-2m+1) <DE>(6m-5-2j) A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2j-4m+4) <DE>(6m-6-2j)
   C1  B(j-2m+3) E1 B(4m-5-j)
   C(2m-1)  D2
PER
FOR  j FROM 3m-2 UPTO 4m-6  REP
   _(2m-1)  A(2m-1)
   _1  E(j-2m+1) B1 E(4m-4-j) A1
```

```
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2m-2)
   C1  B(j-2m+3) E1 B(4m-5-j)
   C(2m-1)  D2
PER
#  j=4m-5
   _(2m-1)  A(2m-1)
   _1  E(2m-4) B1 E1 A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2m-2)
   C1  B(2m-2) D1
   C(2m-1)  D2
#  j=4m-4
   _(2m-1)  A(2m-1)
   _1  E(2m-3) B1 A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  D(2m-1)
   C1  E1 B(2m-2)
   C(2m-1)  D1 B1
#  j=4m-3
   _(2m-1)  A(2m-1)
   _1  E(2m-2) A1
   _(2m-1)  A(2m-1)  B1 D1 B(2m-3)
   C(2m-1)  B1 D1 B1 D(2m-4)
   C1  B1 D1 B(2m-3)
   C(2m-1)  D2
REPEAT
FOR  j FROM 0 UPTO m-3  REP
   _(2m-1)  A(2m-1)
   _1  E(j) B1 E(j) D1 E(2m-4-2j) A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2j+3) E1 D(2m-6-2j)
   C1  B(j+2) E1 B(2m-4-j)
   C(2m-1)  D2
PER
#  j=m-2
   _(2m-1)  A(2m-1)
   _1  E(m-2) B1 E(m-2) D1  A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2m-2)
   C1  B(m) E1 B(m-2)
   C(2m-1)  D2
FOR  j FROM m-1 UPTO 2m-5  REP
   _(2m-1)  A(2m-1)
   _1  E(j) B1 E(2m-3-j) A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2m-2)
   C1  B(j+2) E1 B(2m-4-j)
   C(2m-1)  D2
PER
#  j=2m-4
   _(2m-1)  A(2m-1)
   _1  E(2m-4) B1 E1 A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  B1 D(2m-2)
   C1  B(2m-2) D1
   C(2m-1)  D2
#  j=2m-3
   _(2m-1)  A(2m-1)
   _1  E(2m-3) B1 A1
   _(2m-1)  A(2m-1)  B(2m-1)
   C(2m-1)  D(2m-1)
```

```
   C1  E1 B(2m-2)
   C(2m-1)  D1 B1
#  j=2m-2
   _(2m-1)  A(2m-1)
   _1  E(2m-2) A1
   _(2m-1)  A(2m-1)  B1 D1 B(2m-3)
   C(2m-1)  B1 D1 B1 D(2m-4)
   C1  B1 D1 B(2m-3)
   C(2m-1)  D2
ENDREP
```

Literaturverzeichnis

[AB95] I. Althöfer and J. Bültermann, *Superlinear period lengths in some subtraction games*, Theoretical Computer Science **148** (1995) 111–119.

[Bac1612] C.-G. Bachet de Méziriac, *Problèmes plaisants et délectables*, Paris (1612), 1re édition.

[BC39] W. W. Rouse Ball and H. S. M. Coxeter, *Mathematical Recreations & Essays*, Mac Millan 1939, 11th Edition.

[BCG82]* E. R. Berlekamp, J. H. Conway, and R. K. Guy, *WINNING WAYS for Your Mathematical Plays*, **1** – **2** 1982.
Deutsche Ausgabe: *GEWINNEN — Strategien für mathematische Spiele*, **1** – **4** Vieweg, Braunschweig 1985 – 1986.

[Bou02] C. L. Bouton, *Nim, a game with a complete mathematical theory*, Annals of Mathematics, Princeton (2) **3** (1901 – 1902) 35–39.

[Bül93] J. Bültermann, *Periodenlängen in Nim-Spielen*, Diplomarbeit, Fakultät für Mathematik, Universität Bielefeld 1993.

[Fla91] A. Flammenkamp, *Drei Beiträge zur diskreten Mathematik: Additionsketten, No-Three-in-Line-Problem, Sociable Numbers*, Diplomarbeit, Fakultät für Mathematik, Universität Bielefeld 1991.

[GS56] R. K. Guy and C. A. B. Smith, *The G-values for various games*, Proc. Camb. Phil. Soc. 52 (1956) 514–526

[Kre69] B. Krekó, *Lehrbuch der Linearen Optimierung*, VEB Berlin 1969, 4te Auflage.

[Gru39] P. M. Grundy, *Mathematics and Games*, Eureka **2** (1939) 6–8; Reprint Eureka **27** (1964) 9–11.

[HS91] S. P. Harbison and G. L. Steele Jr., *C, a Reference Manual*, New Jersey 1991 3rd Edition.

[Hen97] A. Henning, *Einige nichtlineare Shiftregister mit langen Perioden*, Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena 1997.

*Anm.: Seiten- und Bandangaben beziehen sich hier stets auf die deutsche Ausgabe.

[Knu68] D. E. Knuth, *The Art of Computer Programming*, Fundamental Algorithms **1**, Addison-Wesley 1973, 2nd Edition.

[Knu69] D. E. Knuth, *The Art of Computer Programming*, Seminumerical Algorithms **2**, Addison-Wesley 1981, 2nd Edition.

[Kos93] K.-U. Koschnick, private Mitteilung über T. Sillke <1993>.

[LN83] R. Lidl and H. Niederreiter, *Finite fields*, Encyclopedia of Mathematics and its Applications **20**, Addison-Wesley 1983.

[Sch68] F. Schuh, *The Master Book of Mathematical Recreations*, Dover Publications 1968.

[Sil93] T. Sillke, private Mitteilung <1993>.

[Spr35] R. P. Sprague, Über mathematische Kampfspiele, Tôhoku Math. J. **41** (1935-36) 438–444; ZBl. **13**, 290.