Berechnung der Anzahl von DAGs unter allen Graphen mit n Knoten

File: daganz.rev in: /home/wiwi/pwolf/R/div

Peter Wolf

November 13, 2008

Inhalt

1 Das Problem

Es ist die Frage zu klären, wie viele DAGs (directed acyclic graphs) es unter den gerichteten Graphen mit n Knoten gibt. Ein spezieller Graph kann durch eine Adjazeten-Matrix beschrieben werden. Diese besitzt in Zeile i und Spalte j eine 1, falls es eine direkte Verbindung zwischen den Knoten i und j gibt. Eine 0 an der Stelle (i, j) zeigt an, dass keine direkte Verbindung von i nach j existiert. Da für n Knoten die Matrix $n \times n$ Elemente hat, gibt es $2^{n \times n} = 2^{n^2}$ verschiedene Graphen. Diese Zahlen wächst extrem schnell mit wachsendem n:

1 ⟨* 1⟩ ≡
```
options(digits=18)
n<-1:8
a<-cbind("#Graphen"=2^(n^2),
         "Stellen"=ceiling(log((2^(n^2)),base=10)))
rownames(a)<-paste("n =",n)
a
```

```
                   #Graphen Stellen
n = 1                     2       1
n = 2                    16       2
n = 3                   512       3
n = 4                 65536       5
n = 5              33554432       8
n = 6           68719476736      11
n = 7      562949953421312      15
n = 8 18446744073709551616      20
```

Wir setzen schnell die Ziffernanzahl für Ausdrucke zurück.

2 ⟨* 1⟩+ ≡
```
options(digits=7)
```
Die Anzahl der zyklenfreien Graphen unter diesen wird ebenfalls rapide anwachsen. Doch wie groß ist sie genau? Diese Frage wollen wir in diesem Papier beantworten für bis zu $n = 6$. Gleichzeitig soll dieses Papier eine Demonstration dafür sein, wie eine Lösung Schritt für Schritt entwickelt wird und wie wir den Entwicklungsprozess dokumentieren können.

2 Ein einfacher Ansatz für sehr kleine n-Werte

Eine einfache Lösung erhalten wir, indem wir für ein festes n ...

3 ⟨*start* 3⟩ ≡
```
n<-3
```
... alle

1. Adjazeten-Matrizen erstellen,
2. für jede Matrix prüfen, ob ein Zyklus vorliegt, und
3. auf einem Zähler die gefundenen DAGs zählen.

Die möglichen 2^{n^2} Matrizen erhalten wir, indem wir die Zahlen von 0 bis $2^{n^2}-1$ im Zweiersystem darstellen und als Matrix anordnen. Hierzu hilft uns die Funktion `encode()`.

4 ⟨*start* 3⟩+ ≡
```
encode <- function(number, base) {
   # simple version of APL-encode / APL-representation "T", pw 10/02
   # "encode" converts the numbers "number" using the radix vector "base"
   n.base <- length(base); result <- matrix(0, length(base), length(number))
   for(i in n.base:1){
     result[i,] <- if(base[i]>0) number %% base[i] else number
     number     <- ifelse(rep(base[i]>0,length(number)),
                          floor(number/base[i]), 0)
   }
   return( if(length(number)==1) result[,1] else result )
}
```
Hiermit können wir zu einer beliebigen Zahl `zahl` eine Adjazeten-Matrix erstellen:

5 ⟨*konstruiere zu Zahl* `zahl` *Adjazetenmatrix* `A` 5⟩ ≡
```
A<-matrix(encode(zahl,rep(2,n^2)),n,n)
```

Zur Demonstration berechnen wir zu den Zahlen 0, 7, 128, 511 die zugehörigen Matrizen.

6 ⟨*demonstriere Generierung von Adjazeten-Matrizen* 6⟩ ≡
```
for(zahl in c(0,7,128,511)) {
  cat("n =",n,"Zahl =",zahl,"\n")
  ⟨konstruiere zu Zahl zahl Adjazetenmatrix A 5⟩
  print(A)
}
NULL
```

```
n = 3 Zahl = 0
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
n = 3 Zahl = 7
     [,1] [,2] [,3]
[1,]    0    0    1
[2,]    0    0    1
[3,]    0    0    1
n = 3 Zahl = 128
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    1    0    0
[3,]    0    0    0
n = 3 Zahl = 511
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Es ist noch zu klären, wie man feststellen kann, ob ein Zyklus vorliegt. Wenn sich auf der Hauptdiagonalen der Adjazeten-Matrix, die wir im Folgenden mit A bezeichnen wollen, eine 1 befindet, haben wir einen Zyklus der Länge 1

gefunden. Befindet sich eine 1 auf der Diagonalen von A^2, existiert ein Zyklus der Länge 2 usw. Bei n Punkten brauchen wir also nur die Matrizen A^j mit $j = 1, \ldots, n$ untersuchen.

7 ⟨*checke* A *durch Multiplikation und setze* `is.dag` 7⟩ ≡
```
Aj<-diag(n)
is.dag<-TRUE
for(j in 1:n){
  Aj<-Aj%*%A
  if(any(0!=diag(Aj))) is.dag<-FALSE
}
```
Wir setzen einen Zähler `count`, der die DAGS zählt.

8 ⟨*initialisiere Zähler* `count` 8⟩ ≡
```
count<-0
```
Die Einzelteile können wir schnell zu einer Lösung zusammensetzen:

9 ⟨*erster Ansatz* 9⟩ ≡
```
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
for(zahl in 0:(2^(n^2)-1)){
  ⟨konstruiere zu Zahl zahl Adjazetenmatrix A 5⟩
  ⟨checke A durch Multiplikation und setze is.dag 7⟩
  count<-count+is.dag
}
})
cat("erster Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

Hier folgt das Ergebnis der Anweisungen:

```
erster Ansatz: n = 3 count = 25
[1] 1.696 0.000 1.814 0.000 0.000
```

Für $n = 3$ benötigen wir schon über 1 Sekunde Rechenzeit, was größere n-Werte sofort verbietet. Schon für $n = 4$ war der Entwickler zu ungeduldig, um das Ergebnis abzuwarten. Es müssen also Verbesserungen her.

3 Tuning und alternative Zyklussuche

Wir können die erste Lösung ein wenig tunen, indem wir die Schleife frühzeitig abbrechen. Denn wenn ein Zyklus entdeckt ist, steht die Klassifikation als *zyklisch* fest.

10 ⟨*checke* A *durch Multiplikation und setze* `is.dag` / *B* 10⟩ ≡
```
Aj<-diag(n)
is.dag<-TRUE
for(j in 1:n){
  Aj<-Aj%*%A
  if(any(0!=diag(Aj))){is.dag<-FALSE; break}
}
```

Diese Verbesserung ist schnell eingebaut.

11 ⟨*zweiter Ansatz* 11⟩ ≡
```
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
for(zahl in 0:(2^(n^2)-1)){
```

```
  ⟨konstruiere zu Zahl zahl Adjazetenmatrix A 5⟩
  ⟨checke A durch Multiplikation und setze is.dag / B 10⟩
  count<-count+is.dag
}
})
cat("zweiter Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

```
zweiter Ansatz: n = 3 count = 25
[1] 1.579 0.002 1.700 0.000 0.000
```

Die Verbesserung ist bescheiden. Vielleicht sollten wir zur Zykluserkennung auf eine andere Idee zurückgreifen. Falls ein Zyklus länger 1 vorliegt, ist entweder einer der Eigenwerte von A negativ oder komplex. Deshalb können wir schreiben:

12 ⟨*checke* A *durch Eigenwert-Check und setze* is.dag 12⟩ ≡

```
if(any(0!=diag(A))){ next }
ev<-eigen(A)$values
is.dag<-all(Re(ev)>=0 & 0==Im(ev))
#is.dag<-!any(Re(ev)<0 | Im(ev)!=0)
```

13 ⟨*dritter Ansatz* 13⟩ ≡

```
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
for(zahl in 0:(2^(n^2)-1)){
  ⟨konstruiere zu Zahl zahl Adjazetenmatrix A 5⟩
  ⟨checke A durch Eigenwert-Check und setze is.dag 12⟩
  count<-count+is.dag
}
})
cat("dritter Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

```
dritter Ansatz: n = 3 count = 25
[1] 1.590 0.000 1.701 0.000 0.000
```

Auch dieser Ansatz führt nicht zu einer Verbesserung. Vielleicht ändert sich die Situation für $n = 4$. Mal sehen.

14 ⟨* 1⟩+ ≡

```
n<-4
⟨erster Ansatz 9⟩
⟨zweiter Ansatz 11⟩
⟨dritter Ansatz 13⟩
```

Hier das Ergebnis:

```
erster Ansatz: n = 4 count = 543
[1] 341.627   0.622 386.139   0.000   0.000
zweiter Ansatz: n = 4 count = 543
```

```
[1] 334.540   0.373 366.629   0.000   0.000
dritter Ansatz: n = 4 count = 543
[1] 328.655   0.265 357.852   0.000   0.000
```

Und die Hoffnungen, für $n = 5$ schnell zu einem Ergebnis zu kommen, müssen begraben werden. Für $n = 5$ überschlagen wir zur Feststellung der Größenordnung: 500×5 Minuten ≈ 2400 Minuten $= 40$ Stunden. Der Faktor 500 ergibt sich durch die Division: $2^{5^2}/2^{4^2} = 33554432/65536 = 512$. Da die Einzeluntersuchungen für $n = 5$ schwieriger als für $n = 4$ sind, wird der reale Zeitverbrauch (weit?) über 40 Stunden liegen.

4 Ermittlung der Hauptursache für den Zeitverbrauch

Als nächstes wollen herausfinden, was eigentlich Zeit kostet. Ist die Prüfung der Adjazeten-Matrizen oder ist vielleicht die Erzeugung der Matrizen der kritische Schritt? Berechnen wir einmal nur die Eigenwerte.

15 ⟨* 1⟩+ ≡
```
n<-3
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
for(zahl in 0:(2^(n^2)-1)){
  A<-cbind(c(1,0,1),c(0,0,1),c(1,0,0))
  ⟨checke A durch Eigenwert-Check und setze is.dag 12⟩
  count<-count+is.dag
}
})
cat("Zeit Eigenwert-Berechnungen, dritter Ansatz: n =",n,"\n")
print(zeit)
```

Wir erhalten:

```
Zeit Eigenwert-Berechnungen, dritter Ansatz: n = 3
[1] 0.024 0.000 0.024 0.000 0.000
```

Und nun die Zeit für die Matrixerstelleung.

16 ⟨* 1⟩+ ≡
```
n<-3
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
for(zahl in 0:(2^(n^2)-1)){
  ⟨konstruiere zu Zahl zahl Adjazetenmatrix A 5⟩
  #checke [[A]] durch Eigenwert-Check und setze [[is.dag]]
  count<-count+3 # dummy-Addition
}
})
cat("Zeit Matrix-Konstruktion, dritter Ansatz: n =",n,"\n")
print(zeit)
```

```
Zeit Matrix-Konstruktion, dritter Ansatz: n = 3
[1] 1.379 0.000 1.476 0.000 0.000
```

Aha, dort steckt die Wurm drin! Die Zeit wird verschwendet während der Ermittlung der 0-1-Folgen .

5 Eine beschleunigte 0-1-Folgenkonstruktion

Wir sollten die 0-1-Folgen der Matrizen also anders ermitteln. Dazu überlegen wir: Die Hälfte aller 0-1-Folgen der Länge n erhalten wir, indem wir an die Menge aller Folge der Länge $n-1$ eine 1 hängen, die andere durch Anhängen einer 0. Tauschen wir 0 und 1 zur Einsparung von Speicherplatz gegen `FALSE` bzw. `TRUE` aus, erhalten wir die Menge aller 0-1-Folgen der Länge m durch:

17 ⟨*ermittle die Menge aller 0-1-Folgen der Länge m* 17⟩ ≡
```
Folgen.set<-rbind(TRUE,FALSE)
folgen.laenge<-m
i<-1
while(i<folgen.laenge){
  i<-i+1
  Folgen.set<-rbind(cbind(Folgen.set,TRUE),
                    cbind(Folgen.set,FALSE))
}
Folgen.set<-t(Folgen.set)
```
Mit diesem Ansatz bekommen wir die Adjazeten-Matrix durch Zugriff auf die Zeilen von `Folgen.set`. Günstiger ist es, auf Spalten zuzugreifen. Denn R-Kenner wissen, dass die Elemente einer Spalte im Speicher nebeneinander stehen. Deshalb transponieren wir `Folgen.set`. Der Einbau dieser neuen Teillösung folgt und wird gleich probiert:

18 ⟨*vierter Ansatz* 18⟩ ≡
```
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-n^2
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
for(zahl in 1:ncol(Folgen.set)){
  A<-matrix(Folgen.set[,zahl]+0,n,n)
  ⟨checke A durch Eigenwert-Check und setze is.dag 12⟩
  count<-count+is.dag
}
})
cat("vierter Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

19 ⟨* 1⟩+ ≡
```
n<-3
⟨vierter Ansatz 18⟩
```

```
vierter Ansatz: n = 3 count = 25
[1] 0.143 0.000 0.160 0.000 0.000
```

Das ist für $n = 3$ so schnell, dass wir uns gleich an $n = 4$ wagen können.

20 ⟨* 1⟩+ ≡
```
n<-4
⟨vierter Ansatz 18⟩
```

```
vierter Ansatz: n = 4 count = 543
[1] 12.057  0.035 13.004  0.000  0.000
```

Der vierte Ansatz ist um den Faktor 25 schneller als der dritte und sein Einsatz kann für $n = 5$ überlegt werden. $n = 5$ erfordert eine Größenordnung von mindestens 500×0.25 Minuten $= 125$ Minuten = 2 Stunden 5 Minuten.

Für eine einmalige Berechnung wäre dieses schon akzeptabel, doch wollen wir auch noch $n = 6$ in den Griff bekommen und würden nach diesem Ansatz mindestens 2 Stunde$\times$2000 = 4000 Stunden, also um die 160 Tage benötigen. Dabei ist noch nicht bedacht, dass die Eigenwertberechnung zeitaufwändiger wird – Faktor 2 (?) – und dass wir wahrscheinlich ein Speicherplatzproblem bekommen. Denn unsere Matrix wird $2^{n \cdot n} = 2^{36}$ = 64 Giga Spalten der Länge 6 haben.

6 Tuning der vierten Lösung

Wie lässt sich die vierte Lösung verbessern? Betrachten wir dazu die Adjazeten-Matrizen, die DAGs beschreiben. Diese werden auf jeden Fall keine 1 auf der Diagonalen haben. Deshalb können wir uns sparen, solche zu generieren. Weiter können wir uns jede Matrix zusammengesetzt aus zwei Dreiecksmatrizen vorstellen. Also können wir 0-1-Folgen der Länge $(n^2 - n)/2$ konstruieren und damit Dreiecksmatrizen bilden. Mit jeweils zwei Folgen aus der Menge `Folgen.set` können wir die Matrix `A` zusammensetzen.

21 ⟨*fünfer Ansatz* 21⟩ ≡

```
n<-4
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-(n^2-n)/2
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
A<-matrix(0,n,n)
upper.index<-upper.tri(A)
lower.index<-lower.tri(A)
for(lower.zahl in 1:ncol(Folgen.set)){
  for(upper.zahl in 1:ncol(Folgen.set)){
    A[upper.index]<-Folgen.set[,upper.zahl]+0
    A[lower.index]<-Folgen.set[,lower.zahl]+0
    ⟨checke A durch Eigenwert-Check und setze is.dag 12⟩
    count<-count+is.dag
  }
}
})
cat("5. Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

```
5. Ansatz: n = 4 count = 543
[1] 7.856 0.019 8.846 0.000 0.000
```

Auch dieser Ansatz erarbeitet das richtige Ergebnis und spart mehr als ein Drittel der Zeit ein. Doch ist das noch nicht genug.

7 Zur Eigenwertberechnung ein Schritt in die Tiefe

Zur Eigenwertberechung werden in `eigen()` immer wieder unnötige Falluntersuchungen getätigt. Diese lassen sich vermeiden, indem wir auf den Kernaufruf von `eigen()` zurückgreifen:

22 ⟨*checke* A *durch verbessertem Eigenwert-Check* 22⟩ ≡
```
only.values<-TRUE
ev<-.Call("La_rg", A, only.values, PACKAGE = "base")$values
is.dag<-all(Re(ev)>=0 & 0==Im(ev))
#is.dag<-!any(Re(ev)<0 | Im(ev)!=0)
```

Damit erhalten wir:

23 ⟨*sechster Ansatz* 23⟩ ≡
```
n<-4
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-(n^2-n)/2
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
A<-matrix(0,n,n)
upper.index<-upper.tri(A)
lower.index<-lower.tri(A)
for(lower.zahl in 1:ncol(Folgen.set)){
  for(upper.zahl in 1:ncol(Folgen.set)){
    A[upper.index]<-Folgen.set[,upper.zahl]+0
    A[lower.index]<-Folgen.set[,lower.zahl]+0
    ⟨checke A durch verbessertem Eigenwert-Check 22⟩
    count<-count+is.dag
  }
}
})
cat("6. Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

Wir erhalten:

```
6. Ansatz: n = 4 count = 543
[1] 0.326 0.000 0.349 0.000 0.000
```

Das hat es gebracht!! Wir realisieren eine Verbesserung um einen Faktor von etwa 20. Damit könnten wir auch schon fast zufrieden sein, denn die Berechnung für $n = 6$ wird erschwinglich.

8 DAG-Schnellerkennung und Abbruch

Schnellerkennung. Mit etwas Grübeln kommen wir zu dem Ergebnis, dass ein Graph zyklisch sein muss, wenn die Adjazeten-Matrix mehr als $(n^2 - n)/2$ Nicht-0-Einträge besitzt. Mit einer schnellen Abfrage können wir also verschiedene Eigenwertberechnungen einsparen.

24 ⟨*siebter Ansatz* 24⟩ ≡
```
n<-4
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-(n^2-n)/2
```

```
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
A<-matrix(0,n,n)
upper.index<-upper.tri(A)
lower.index<-lower.tri(A)
for(lower.zahl in 1:ncol(Folgen.set)){
  for(upper.zahl in 1:ncol(Folgen.set)){
    A[upper.index]<-Folgen.set[,upper.zahl]+0
    A[lower.index]<-Folgen.set[,lower.zahl]+0
    if(sum(A) > m) next
    ⟨checke A durch verbessertem Eigenwert-Check 22⟩
    count<-count+is.dag
  }
}
})
cat("7. Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

```
7. Ansatz: n = 4 count = 543
[1] 0.284 0.008 0.324 0.000 0.000
```

Vorzeitiger Schleifenabbruck. Statt `next` könnten wir aus der inneren Schleife ausbrechen, wenn die Liste der 0-1-Folgen nach der Anzahl der 1en sortiert ist. Dieses wird einen weiteren kleinen Geschwindigkeitsvorteil bringen. Gesagt, getan:

25 ⟨*achter Ansatz* 25⟩ ≡

```
n<-4
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-(n^2-n)/2
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
A<-matrix(0,n,n)
upper.index<-upper.tri(A)
lower.index<-lower.tri(A)
ind<-order(colSums(Folgen.set))
Folgen.set<-Folgen.set[,ind]
for(lower.zahl in 1:ncol(Folgen.set)){
  A[lower.index]<-Folgen.set[,lower.zahl]+0
  for(upper.zahl in 1:ncol(Folgen.set)){
    A[upper.index]<-Folgen.set[,upper.zahl]+0
    if(sum(A) > m) break
    ⟨checke A durch verbessertem Eigenwert-Check 22⟩
    count<-count+is.dag
  }
}
})
cat("8. Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

Wir erhalten:

```
8. Ansatz: n = 4 count = 543
[1] 0.178 0.000 0.198 0.000 0.000
```

Immerhin, der Mensch freut sich. So, jetzt ist die Kunst schon fast ausgereizt.

Doch noch haben wir eine Idee im Petto: Ausnutzung von Symmetrie.

9 Symmetrieüberlegungen

Eine Adjazeten-Matrix und ihre transponierte müssen dieselbe Zykluseigenschaft besitzen. Deshalb brauchen wir nur einen dieser Fälle untersuchen. Wir können auch formulieren: Für die erste untere Dreiecksmatrix müssen wir alle möglichen Dreiecksmatrizen oben durchspielen. Für die zweite untere Dreiecksmatrix können wir uns eine Dreiecksmatrix oben verzichten. Genauer können wir im zweiten Durchlauf auf diejenige Füllung oben verzichten, die sich durch Spiegelung der ersten unteren Füllung ergibt.

Wie können wir dieses umsetzen? Ganz einfach: Zuerst füllen wir die untere Dreiecksmatrix, indem wir eine 0-1-Folge (`Folgen.set[,lower.zahl]`) oben rechts ablegen und dann die Matrix transponieren. In einem zweiten Schritt füllen wir in die transformierte Matrix wieder oben rechts die zweite 0-1-Folge (`Folgen.set[,upper.zahl]`) ein.

Es ist zu beachten, dass wir bei einem gefundenen DAG den Zähler um 2 hochsetzen müssen, wenn der DAG einen nicht untersuchten Spiegelungspartner besitzt.

26 ⟨*neunter Ansatz* 26⟩ ≡

```
n<-5
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-(n^2-n)/2
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
A<-matrix(0,n,n)
upper.index<-upper.tri(A)
Folgen.set<-Folgen.set[,order(colSums(Folgen.set))]
for(lower.zahl in 1:ncol(Folgen.set)){
  A[upper.index]<-Folgen.set[,lower.zahl]+0; A<-t(A)
  for(upper.zahl in lower.zahl:ncol(Folgen.set)){
    A[upper.index]<-Folgen.set[,upper.zahl]+0
    if(sum(A) > m) break
    ⟨checke A durch verbessertem Eigenwert-Check 22⟩
    if(is.dag) count<-count+2-(lower.zahl==upper.zahl)
  }
}
})
cat("9. Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

```
9. Ansatz: n = 4 count = 543
[1] 0.094 0.000 0.098 0.000 0.000
```

10 Weitere Verbesserungen

Es wird weitere Verbesserungen geben. Eine leider wirkungslose sei erwähnt. Wenn die Matrix A ein Element auf der Diagonalen hat, existiert mindestens ein

Zyklus der Länge 1. Dieses wird schon durch unsere Konstruktion abgefangen. Wenn die Bedingung `any(A & t(A))` erfüllt ist, liegt mindestens ein Zyklus der Länge zwei vor. Diese führt zu der Alternative 10, die jedoch als wieder etwas langsamer herausstellte:

27 ⟨*neunter Ansatz* 26⟩+ ≡

```
n<-4
zeit<-system.time({
⟨initialisiere Zähler count 8⟩
m<-(n^2-n)/2
⟨ermittle die Menge aller 0-1-Folgen der Länge m 17⟩
A<-matrix(0,n,n)
upper.index<-upper.tri(A)
Folgen.set<-Folgen.set[,order(colSums(Folgen.set))]
for(lower.zahl in 1:ncol(Folgen.set)){
  A[upper.index]<-Folgen.set[,lower.zahl]+0; A<-t(A)
  for(upper.zahl in lower.zahl:ncol(Folgen.set)){
    A[upper.index]<-Folgen.set[,upper.zahl]+0
    if(sum(A) > m) break
    if(any(A & t(A))) next
    ⟨checke A durch verbessertem Eigenwert-Check 22⟩
    if(is.dag) count<-count+2-(lower.zahl==upper.zahl)
  }
}
})
cat("10. Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

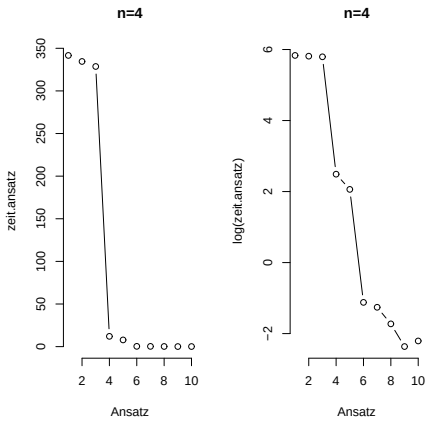
```
10. Ansatz: n = 4 count = 543
[1] 0.110 0.000 0.115 0.000 0.000
```

11 Darstellung der Rechenzeiten der Ansätze

Zum Abschluss wollen wir einen Blick darauf werfen, welche Verbesserungen mit dem Entwicklungsprozess einhergegangen sind.

28 ⟨* 1⟩+ ≡

```
zeit.ansatz<-c(
"1"=341.627,"2"=334.540,"3"=328.655,"4"=12.057,"5"=7.856,
"6"=0.326,"7"=0.284,"8"=0.178,"9"=0.094,"10"=0.110)
par(mfrow=1:2)
plot(zeit.ansatz,type="b",xlab="Ansatz",main="n=4")
plot(log(zeit.ansatz),type="b",xlab="Ansatz",main="n=4")
par(mfrow=c(1,1))
```



12 Realisierte Läufe

12.1 Eine Realisation auf dem wiwi21

Noch vor dem Ende der *Optimierung* wurde einem schnellen Rechner folgende Zwischenlösung anvertraut.

29 ⟨*wiwi21* 29⟩ ≡

```
# setze Knotenanzahl
print(date())
n<-4 ## Die Ausgabe unten ergibt sich mit: n<-6
# Off-Diag-Element
ODE<-(n-1)*n; ODEH<-1:(ODE/2); ODER<-(1:ODE)[-ODEH]
# Schleife über Matrizen
count<-0
mat<-matrix(0,n,n)
ind.lower<-lower.tri(mat)
ind.upper<-upper.tri(mat)
result<-rep(0.0,ODE)
file.remove("wieviel-n-6")
j<-0
while(j<2^ODE){
  if((j%%1000000)==0) base::cat(date(),j,"\n",append=T,file="wieviel-n-6")
  number<-j; j<-j+1
  for(i in ODE:1){
    result[i] <- number %% 2
    number    <- floor(number/2)
  }
  mat[ind.lower]<-result[ODEH]
  mat[ind.upper]<-result[ODER]
  if(any(mat & t(mat))) next
#  ev<-eigen(mat)$values
  storage.mode(mat) <- "double"
  only.values<-TRUE
 ev <- .Call("La_rg", mat, only.values, PACKAGE = "base")$values
 #   count<-count + all((as.real(ev)>=0) & (as.real(ev)==ev))
    count<-count + all((Re(ev)>=0) & (0==Im(ev)))
  #print("found");   print(ev) ;   print(mat);   print(count)
} #;  print(mat)
print(date()); print(count)
```

Diese hat innerhalb von über 4 Tagen das richtige Ergebnis hervorgebracht:

```
[1] "Fri Oct 27 15:01:03 2006"
[1] "Tue Oct 31 23:00:11 2006"
[1] 3781503
```

12.2 Eine Realisation auf einem Laptop

Alle der angesprochenen Verbesserungen wurden auf einem Laptop umgesetzt und führten zum selben Ergebnis, jedoch in wesentlich kürzerer Zeit.

30 ⟨* 1⟩+ ≡

```
n<-4
zeit<-system.time({
# 0-1-Folgen für Dreiecksmatrix
a<-rbind(TRUE,FALSE)
length.tri.mats<-(n^2-n)/2
i<-1
while(i<length.tri.mats){
  i<-i+1; a<-rbind(cbind(a,TRUE),cbind(a,FALSE))
}
a<-t(a)+0
# Anzahl der Folgen berechnen
anz.tri.mats<-ncol(a)
# Sortierung der Folgen nach Anzahl von Verbindungen
a<-a[,order(colSums(a)),drop=FALSE]
# Erstellung einer Rohmatrix
aa<-matrix(0,n,n)
# Zugriffsvektor auf untere Dreiecksmatirx
iu<-upper.tri(aa)
count<-0
only.values <- TRUE
for(i in 1:anz.tri.mats){
  aa[iu]<-h<-a[,i]; aa<-t(aa)
  j<-i-1; sumi<-sum(h)
  while(j<anz.tri.mats){
    j<-j+1
    aa[iu]<-h<-a[,j]
    if((sumi+sum(h))>length.tri.mats) break
    if(any(aa & t(aa))) next
 #   storage.mode(aa) <- "double"
    ev<-.Call("La_rg", aa, only.values, PACKAGE = "base")$values
    if( all(0<=Re(ev)) & all(Im(ev)==0)) count<-count+2-(i==j)
  }
}
count
})
cat("bester (?) Ansatz: n =",n,"count =",count,"\n")
print(zeit)
```

12.3 Darstellung des Ablaufs *schnellen* Berechnung

In diesem Abschnitt wollen wir darstellen, welche Effekt die entwickelten Abkürzungsstrategien haben. Dazu wurden Zwischenzeiten in einer LOG-Datei festgehalten. Für die Laptop-Lösung hat die Datei folgenden Inhalt:

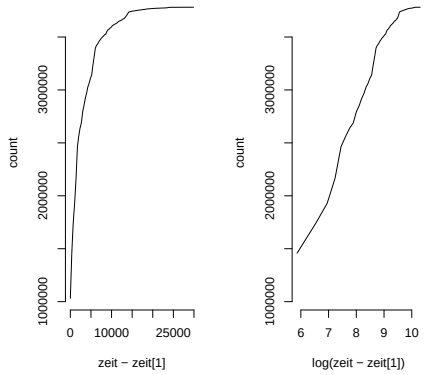
```
n = 6 count = 1031179 i = 100 Sat Oct 28 08:54:16 2006
n = 6 count = 1457505 i = 200 Sat Oct 28 09:00:08 2006
n = 6 count = 1743317 i = 300 Sat Oct 28 09:05:52 2006
...
n = 6 count = 3781503 i = 32600 Sat Oct 28 17:12:23 2006
n = 6 count = 3781503 i = 32700 Sat Oct 28 17:12:23 2006
n = 6 count = 3781503 i = 32768 j = 32768 Sat Oct 28 17:12:23 2006
```

Zur Darstellung des Ablaufs der Laptop-Lösung lesen wir zunächst die LOG-Datei ein und extrahieren die wichtigen Größen.

31 ⟨* 1⟩+ ≡
```
a<-scan("daganz.LOG","",sep="\n")
a<-strsplit(a," ")
stop<-a[length(a)]; a<-a[-length(a)]
i<-as.numeric(unlist(lapply(a,function(x) x[9])))
count<-as.numeric(unlist(lapply(a,function(x) x[6])))
zeit<-unlist(lapply(a,function(x) x[13]))
zeit<-unlist(lapply(strsplit(zeit,":"),
                    function(x) sum(as.numeric(x)*c(3600,60,1))))
```

Wie nähert sich der Zähler im Zeitablauf an das Ergebnis an?

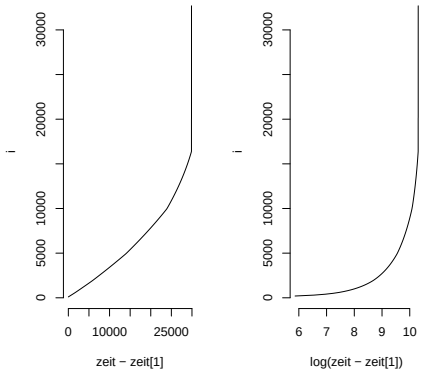
32 ⟨* 1⟩+ ≡
```
par(mfrow=1:2)
plot(zeit-zeit[1], count,type="l")
plot(log(zeit-zeit[1]), count,type="l")
par(mfrow=c(1,1))
```



Wie werden die vielen 0-1-Folgen durchgearbeitet? Dazu schauen wir uns an, wie sich der äußere Zähler gegenüber der Zeit entwickelt.

33 ⟨* 1⟩+ ≡
```
par(mfrow=1:2)
plot(zeit-zeit[1], i,type="l")
```

```
plot(log(zeit-zeit[1]), i,type="l")
par(mfrow=c(1,1))
```



13 Literatur

Die Auseinandersetzung mit DAGs ist nicht neu und auch die hier untersuchte Frage war (natürlich) bereits gelöst. So finden wir in:

> G. Melancon, I. Dutour, M. Bousquet-Melou (2000): Random Generation of Dags for Graph Drawing, Information Systems (INS), INS-R0005 February 29,
> `http://ftp.cwi.nl/CWIreports/INS/INS-R0005.pdf`

folgende Tabelle:

n	1	2	3	4	5	6	7	8	9
n_{DAGs}	1	3	25	543	29281	3781503	1138779265	783702329343	1213442454842881

Die Autoren weisen auf die folgenden Arbeiten hin:

> Bender E. A., Richmond. L. B., and Robinson R. W. (1988): The asymptotic number of acyclic digraphs, ii. Journal of Combinatorial Theory, Series B, 44:363–369.
> Bender E. A., Richmond. L. B., Robinson R. W., and Wormald N. C. (1986): The asymptotic number of acyclic digraphs, I. Combinatorica, 6(1):15–22.

Alle die, die etwas über Eigenwerte nachlesen wollen, seien zunächst verwiesen auf:

> J. H. Wilkinson (1967): The algebraic eigenvalue problem, Clarendon, Oxford, QB360 W686.