Seminar „Algorithmen auf Graphen“, WS 1998/99

Generierung zufälliger, regulärer Graphen

von Thomas Dreibholz
Aus: Journal of Algorithms, 1984

1. Vorbemerkungen

1.1. Quellen dieser Seminarunterlagen

WWW: http://ibm.rhrz.uni-bonn.de/~uzs6pq
Lokal: ~dreibhol/randgraph
E-Mail: Dreibholz@bigfoot.com (Thomas Dreibholz)

1.2. Einige Definitionen zur Wiederholung

Definition: Grad eines Knotens v
Der Grad eines Knotens v ist die Anzahl der vom diesem Knoten ausgehenden oder abgehenden Kanten.
Definition: adjazent/inzident
Zwei Knoten heißen adjazent, falls sie durch eine Kante verbunden sind. Zwei Kanten heißen adjazent, falls sie einen gemeinsamen Endpunkt besitzen. Ein Knoten x und die Kante (x,y) heißen inzident.
Definition: r-regulärer Graph
Ein Graph G ist r-regulär, falls alle Knoten des Graphen den gleichen Grad r besitzen. Er ist kubisch, falls r = 3.
Definition: bipartiter Graph
Ein Graph G ist bipartit, wenn sich seine Knotenmenge V in zwei disjunkte Mengen *Rot* und *Blau* zerlegen läßt, wobei keine zwei Knoten aus der gleichen Menge adjazent, also miteinander verbunden sind.
Ein bipartiter Graphen G ist (r,s)-biregulär, falls der Grad eines jeden Knotens der Menge *Rot* r ist und der Grad eines jeden Knotens der Menge *Blau* s ist.

Für weitere Definitionen siehe auch Kapitel 1.2 im Informatik III/IV-Skript unter ftp://theory.cs.uni-bonn.de/pub/reports/scripts/EidI.ps.gz

1.3. Motivation

Es gibt viele noch unbekannte Eigenschaften von r-regulären Graphen. Zum Beispiel ist noch nicht bekannt, ob fast alle r-regulären Graphen hamiltonisch sind? Um r-reguläre Graphen auf bestimmte mögliche Eigenschaften zu testen, ist es deshalb sinnvoll, zufällige r-reguläre Graphen zu erzeugen und mit diesen die gewünschten Tests durchzuführen. Diese zufällige Erzeugung solcher Graphen ist das Thema dieses Vortrages.

2. Einleitung

Es werden nun die folgenden drei Prozeduren vorgestellt:

- RandCubic() zum Erzeugen eines zufälligen, gelabelten, kubischen Graphen mit gegebener Anzahl von Knoten.
- Bipart() zur Erzeugung eines zufälligen, gelabelten, bipartiten Graphen mit Knoten eines gegebenen Grades, was äquivalent zur Erzeugung einer zufälligen 0-1-Matrix mit gegebenen Zeilen- und Spaltensummen ist.
- RandGraph() zur Erzeugung eines zufälligen, gelabelten Graphen mit gegebener Knotenzahl.

Hierbei durchlaufen Bipart() und RandGraph() wiederholte Erzeugungsversuche, die eventuell fehlschlagen können, jedoch bei Erfolg den gewünschten Graphen liefern. RandCubic() ist ein normaler Algorithmus.
Anschließend wird gezeigt werden, daß Zeit- und Speicherplatzkomplexität polynomiell sind, wobei natürlich nur RandCubic() eine endliche Worst-Case-Zeitkomplexität besitzen kann. Des weiteren wird gezeigt werden, daß es hierfür nicht unbedingt notwendig ist, daß die Zufallsfunktion exakt mit der gleichen Wahrscheinlichkeit 0 oder 1 liefert, sondern daß es ausreichend ist, wenn das Verhältnis der beiden Wahrscheinlichkeiten sehr nahe bei 1 liegt.

3. RandCubic()

Ein kubischer Graph besitzt eine Reihe von Untergraphen verschiedener Typen, die mit einer bekannten Wahrscheinlichkeit auftreten können. RandCubic() erzeugt mittels Operationen, die Graphen des einen in den anderen Typ überführen gemäß der bekannten Wahrscheinlichkeiten einen kubischen Graph.

3.1. Graphentypen:

A-Graph: Der Graph besitzt einen Knoten mit dem Grad 2, der Rest ist vom Grad 3.
B-Graph: Der Graph besitzt zwei Knoten vom Grad 2 mit den Labels „+“ und „-“, die als positiver und negativer Pol bezeichnet werden. Der Rest der Knoten besitzt den Grad 3.
- Typ 1: Die Pole sind nicht adjazent; dies entspricht einem C-Graph (siehe unten).
- Typ 2: Die Pole sind 2 Knoten in einem 3er-Kreis.
- Typ 3: Alle anderen Fälle, d.h. Pole sind adjazent zueinander und zu zwei anderen Knoten.

C-Graph: Wie B, jedoch sind die Pole nicht adjazent. Seien u und v die Knoten adjazent zum positiven Pol.
- Typ 1: u und v sind nicht adjazent.
- Typ 2: u und v sind adjazent, (+) ist der einzige Knoten, der zu u und v adjazent ist.

Für alle weiteren Typen: u und v adjazent und es existiert ein Knoten w, w nicht gleich (+), der zu u und v adjazent ist.
- Typ 3: w ist (-).
- Typ 4: w ist adjazent zu (-).
- Typ 5: sonst.

D-Graph: Der Graph ist kubisch, d.h. alle Knoten besitzen den Grad 3.

3.2. Graphenanzahlen:

Die Anzahl der Graphen von Typ A,B,C oder D mit n Knoten wird mit a[n], b[n], c[n] bzw. d[n] bezeichnet. Nach Konvention sind leere Graphen kubisch, besitzen jedoch keine Knoten mit Grad 2. Also: a[0] = b[0] = c[0] = 0 und d[0] = 1.
Im weiteren Verlauf werden nun insgesamt neun Operationen eingeführt, die Graphen des einen Typs in einen anderen Typ überführen, indem Knoten und Kanten hinzugefügt bzw. entfernt

werden, was zu der Möglichkeit führt, a[n], b[n], c[n] und d[n] rekursiv zu berechnen und schließlich Graphen der Typen A,B,C und D zufällig zu erzeugen.

3.3. Definition von Operationen

1. Entfernung einer Menge S mit k Knoten aus einem Graphen mit n Knoten:
Löschen aller Knoten in S aus dem Graphen G und Löschung aller Kanten, die mit einem Knoten aus S inzident sind. Danach Relabeling der Knoten von G mit den Labels 1,...,n-k, wobei jedoch die numerische Ordnung der Knoten nicht verändert wird!
Beispiel: Labels: 1,2,3,4; Löschung von 2 und 3 => 4 wird in 2 umbenannt

2. Hinzufügen von k neuen Knoten mit Labels u[1],...,u[k] zu einem Graphen mit n-k Knoten:
Die Knoten werden hinzugefügt und ein lineares Update der alten Labels durchgeführt, d.h. für die Knoten von V werden Labels aus {1,...,n} \ {u[1],...,u[k]} so gewählt, daß die lineare numerische Ordnung der Labels nicht verändert wird. Wenn also die neuen Labels schon gewählt sind, ist das Ergebnis des linearen Updates der Labels eindeutig festgelegt.
Beispiel: Alt: 1,2,3; Neu: 2,3 => Für alte Labels: 2->4 und 3->5

3. Eine Operation, die aus einer Menge von Graphen T1 eine Menge von Graphen T2 erzeugt, ist exakt, wenn:
- Die Anwendung der Operation auf unterschiedliche Elemente der Menge T1 liefert unterschiedliche Elemente der Menge T2.
- Jedes Element von T2 kann durch die Anwendung der Operation aus T1 erzeugt werden.
- Die Anzahl der verschiedenen Möglichkeiten, die Operation auf einen Graphen G aus T1 anzuwenden, ist gleich für alle G.

Exakte Operationen sind nützlich für Aufzählungen, da sie Viele-zu-Eins-Anzahlzusammenhänge von der Graphenmenge T2 zur Graphenmenge T1 aufweisen.

3.4. Umwandlungs-Operationen

Operation 1: B-Graph mit n-1 Knoten -> A-Graph mit n Knoten

Beschreibung:
=> Eingabe: Gegeben ist ein B-Graph G mit n-1 Knoten.
- Füge einen neuen Knoten v hinzu.
- Kennzeichne v mit Label als {1,...,n} (dafür gibt es n Möglichkeiten).
- Verbinde v mit (+) und (-).
- Lösche „+“ und „-“-Labels und unterziehe Labels der alten Knoten einem linearen Update.

<= Ausgabe: A-Graph mit n Knoten.

Lemma 1:
(a) Jeder A-Graph mit n Knoten kann durch Anwendung von Operation 1 aus genau zwei verschiedenen B-Graphen mit n-1 Knoten erzeugt werden.
(b) Es können genau n unterschiedliche A-Graphen durch unterschiedliche Anwendung von Operation 1 aus einen B-Graphen mit n-1 Knoten erzeugt werden.

Beweis:
(a) Wird v aus dem A-Graphen entfernt, gibt es genau zwei Möglichkeiten, die beiden Knoten vom Grad 2 zu labeln: „+“ und „-“ bzw. „-“ und „+“.
(b) Es gibt genau n Möglichkeiten , den neuen Knoten v zu labeln.

Korollar 1: a[n] = (n/2)*b[n].

Operation 2: A-Graph mit n-3 Knoten -> B-Graph Typ 2 mit n Knoten

Beschreibung:

=> Eingabe: Gegeben ist ein A-Graph G mit n-3 Knoten. Der Knoten mit Grad 2 sei mit x bezeichnet.

- Füge drei neue Knoten hinzu: (+), (-) und w.
- Füge vier neue Kanten hinzu: wx, (+)(-), (+)w und (-)w. (Also: w, (+) und (-) sind 3er-Kreis)
- Wähle drei neue unterschiedliche Labels aus {1,...,n} gewählt (Binomial(n,3) Möglichkeiten).
- Kennzeichne die neuen Knoten mit diesen Labels (3! = 6 Möglichkeiten).
- Unterziehe die alten Labels einem linearen Update.

<= Ausgabe: B-Graph Typ 2 mit n Knoten.

Lemma 2:

Operation 2 ist exakt und erzeugt aus einem A-Graph mit n-3 Knoten einen B-Graph Typ 2 mit n Knoten.

Beweis:

Gegeben ist ein B-Graph vom Typ 2. Die Knoten (+), (-) und der dritte Knoten im 3er-Kreis sind bekannt. Werden diese drei Knoten entfernt, liegt ein eindeutig bestimmter A-Graph vor. Dies ist die Umkehrung von Operation 2 und das Lemma folgt direkt aus der Definition von Operation 2.

Korollar 2: Die Anzahl der B-Graphen Typ 2 mit n Knoten beträgt 6*Binomial(n,3)*a[n-3].

Operation 3: B-Graph mit n-2 Knoten -> B-Graph Typ 3 mit n Knoten

Beschreibung:

=> Eingabe: Gegeben ist ein B-Graph mit n-2 Knoten. w und x bezeichnen den positiven und den negativen Pol.

- Entferne die Labels „+“ und „-“.
- Füge zwei neue Knoten (+) und (-) hinzu.
- Füge drei neue Kanten (+)(-), (+)w und (-)x hinzu.
- Wähle zwei neue Labels aus {1,..,n} (dafür gibt es n*(n-1) Möglichkeiten)).
- Kennzeichne die neuen Knoten (+) und (-) mit diesen Labels
- Unterziehe die alten Labels einem linearen Update.

=> Ausgabe: B-Graph Typ 3 mit n Knoten.

Lemma 3:

Operation 3 ist exakt und erzeugt aus einem B-Graph mit n-2 Knoten einen B-Graph Typ 3 mit n Knoten.

Beweis:

Gegeben ist ein B-Graph Typ 3 H mit n Knoten. Die Knoten (+) und (-) sind festgelegt, denn wenn diese vertauscht werden, ändert sich auch der Graph. Durch Entfernen von (+) und (-) entsteht ein eindeutig bestimmter B-Graph G mit n-2 Knoten, wobei festgelegt ist, daß der Knoten mit dem positiven Pol von G in H adjazent zum dortigen positiven Pol ist.

Korollar 3: Die Anzahl der B-Graphen Typ 3 mit n Knoten beträgt n*(n-1)*b[n-2].

Operation 4: A-Graph mit n-1 Knoten -> C-Graph Typ 1 mit n Knoten

Beschreibung:

=> Eingabe: Gegeben ist ein A-Graph G mit n-1 Knoten (damit (3*n-4)/2 Kanten).

- Wähle eine Kante e, die nicht mit dem Knoten vom Grad 2 inzident ist (dazu gibt es (3*n-8)/2 Möglichkeiten). Seien v und w die Knoten an den Enden von e.
- Lösche Kante e aus dem Graphen G.
- Füge eine neuen Knoten (+) hinzu. Dies wird der positive Pol.
- Füge die Kanten (+)v und (+)w hinzu.
- Wähle ein Label für (+) aus {1,...,n} (dafür gibt es n Möglichkeiten).
- Unterziehe die alten Labels einem linearen Update.
- Der alte Knoten mit Grad 2 ist jetzt der negative Pol.

<= Ausgabe: C-Graph Typ 1 mit n Knoten.

Lemma 4:

Operation 4 ist exakt und erzeugt aus einem A-Graph mit n-1 Knoten einen C-Graph Typ 1 mit n Knoten.

Beweis:

Es ist klar, daß Operation 4 aus einem A-Graph einen C-Graph erzeugt.

Umgekehrt ist ein C-Graph Typ 1 H gegeben bei dem (+) und die beiden zu (+) adjazenten Knoten bekannt sind. Daher erzeugt die Entfernung von (+) und Hinzufügung einer Kante e zwischen diesen zwei Knoten einen eindeutig bestimmten A-Graph. Der Knoten mit Grad 2 ist nicht inzident mit e, da (+) und (-) im Graph H nicht adjazent sind.

Korollar 4: Die Anzahl der C-Graphen Typ 1 mit n Knoten beträgt ((3*n-8)/2)*n*a[n-1].

Operation 5: B-Graph mit n-2 Knoten -> C-Graph Typ 2 mit n Knoten

Beschreibung:

=> Eingabe: Gegeben ist ein B-Graph G mit n-2 Knoten. Seien w, x die Knoten adjazent zu (+).

- Lösche Kanten (+)w und (+)x.
- Füge zwei neue Knoten u und v hinzu.
- Füge fünf neue Kanten (+)u, (+)v, uv, uw und vx hinzu.
- Wähle zwei neue Labels aus {1,..,n} (dafür gibt es n*(n-1) Möglichkeiten)).
- Kennzeichne die Knoten u und v mit diesen Labels
- Unterziehe die alten Labels einem linearen Update.

<= Ausgabe: C-Graph Typ 2 mit n Knoten

Lemma 5:

Operation 5 ist exakt und erzeugt aus einem B-Graph mit n-2 Knoten einen C-Graph Typ 2 mit n Knoten.

Beweis:

Analog zu Lemma 1 bis 4.

Korollar 5: Die Anzahl der C-Graphen Typ 2 mit n Knoten beträgt n*(n-1)*b[n-2].

Operation 6: D-Graph mit n-4 Knoten -> C-Graph Typ 3 mit n Knoten

Beschreibung:

=> Eingabe: Gegeben ist ein D-Graph G mit n-4 Knoten, also ein kubischer Graph. Im Fall n=4 ist dies der leere Graph.

- Füge vier neue Knoten hinzu: Die Pole (+), (-) sowie u und v.
- Füge fünf neue Kanten hinzu: (+)u, (+)v, uv, u(-) und v(-).
- Wähle ein geordnetes Paar von verschiedenen Labels l1, l2 aus {1,...,n} (dafür gibt es n*(n-1) Möglichkeiten).
- Kennzeichne (+) und (-) mit diesen Labels.
- Wähle ein ungeordnetes Paar von verschiedenen Labels aus {1,...,n}\{l1,l2} (dafür gibt es (n-2)*(n-3)/2 Möglichkeiten).
- Kennzeichne u mit dem kleineren Wert und v mit dem größeren.
- Unterziehe die alten Labels einem linearen Update (falls überhaupt welche vorhanden).

<= Ausgabe: C-Graph Typ 3 mit n Knoten.

Lemma 6:

Operation 6 ist exakt und erzeugt aus einem D-Graph mit n-4 Knoten einen C-Graph Typ 3 mit n Knoten.

Beweis:

Analog zu Lemma 1 bis 4.

Korollar 6: Die Anzahl der C-Graphen Typ 3 mit n Knoten beträgt 12*Binomial(n,4)*d[n-4].

Operation 7: A-Graph mit n-5 Knoten -> C-Graph Typ 4 mit n Knoten

Beschreibung:
=> Eingabe: Gegeben ist ein A-Graph G mit n-5 Knoten.
- Füge fünf neue Knoten hinzu: Die Pole (+), (-) und u,v und w.
- Füge sieben neue Kanten hinzu: (+)u, (+)v, uv, uw, vw, w(-) sowie eine Kante zwischen (-) und dem Knoten vom Grad 2.
- Verschiedene Labels l1, l2 und l3 aus {1,...,n} werden für (+), (-) und w sowie ein ungeordnetes Paar von Labels für u und v aus {1,...,n}\{l1,l2,l3} gewählt. Für die Auswahl gibt es 60*Binomial(n,5) Möglichkeiten.
- Kennzeichne die Knoten mit den gewählten Labels.
- Unterziehe die alten Labels einem linearen Update.

<= Ausgabe: C-Graph Typ 4 mit n Knoten.
Lemma 7:
Operation 7 ist exakt und erzeugt aus einem A-Graph mit n-5 Knoten einen C-Graph Typ 4 mit n Knoten.
Beweis:
Analog zu Lemma 1 bis 4.
Korollar 7: Die Anzahl der C-Graphen Typ 4 mit n Knoten beträgt 60*Binomial(n,4)*a[n-5].

Operation 8: B-Graph mit n-4 Knoten -> C-Graph Typ 5 mit n Knoten
Beschreibung:
=> Eingabe: Gegeben ist ein B-Graph G mit n-4 Knoten. Der positive Pol sei mit x bezeichnet.
- Entferne das „+“-Label von x.
- Füge vier neue Knoten hinzu: Den Pol (+) sowie u, v und w.
- Füge sechs neue Kanten hinzu: (+)u, (+)v, uv, uw, vw und wx.
- Wähle ein geordnetes Paar von Labels l1, l2 aus {1,...,n} für (+) und w und ein ungeordnetes Paar von Labels aus {1,...,n}\{l1,l2} für u und v. Für die Auswahl gibt es 12*Binomial(n,4) Möglichkeiten.
- Kennzeichne die Knoten mit den gewählten Labels.
- Unterziehe die alten Labels einem linearen Update.

<= Ausgabe: C-Graph Typ 5 mit n Knoten.
Lemma 8:
Operation 8 ist exakt und erzeugt aus einem B-Graph mit n-5 Knoten einen C-Graph Typ 5 mit n Knoten.
Beweis:
Analog zu Lemma 1 bis 4.
Korollar 8: Die Anzahl der C-Graphen Typ 5 mit n Knoten beträgt 12*Binomial(n,4)*b[n-4].

Operation 9: C-Graph mit n Knoten -> D-Graph mit n Knoten
Beschreibung:
=> Eingabe: Gegeben ist ein C-Graph G mit n Knoten.
- Füge eine neue Kante (+)(-) hinzu.
- Lösche die „+“ und „-“-Labels

<= Ausgabe: D-Graph (kubischer Graph) mit n Knoten.
Lemma 9:
Jeder D-Graph (kubischer Graph) mit n>=4 Knoten kann mit Operation 9 aus genau 3*n verschiedenen C-Graphen mit n>=4 Knoten erzeugt werden.
Beweis:
Jeder C-Graph mit n>0 Knoten kann aus einem kubischen Graphen mit n Knoten erzeugt werden, indem eine Kante von insgesamt (3*n)/2 entfernt wird und die beiden Knoten vom Grad 2 mit „+“ und „-“ gekennzeichnet werden (2 Möglichkeiten). Dies ist die Umkehrung von Operation 9, woraus das Lemma folgt.

Korollar 9: c[n] = 3*n*d[n]

3.5. Rekursive Berechnung der Graphen-Anzahlen

Aus den Startwerten a[0], ..., d[0] sowie den Korollaren 1 bis 9 folgt nun für n >= 4 die rekursive Berechnung der a[n],...,d[n]:
(1)a[n] = (n/2)*b[n-1]
(2)c[n] = ((3*n-8)/2) * n * a[n-1] + n*(n-1)*b[n-2] + 12*Binomial(n,4)*d[n-4] +
60*Binomial(n,5)*a[n-5] + 12*Binomial(n 4)*b[n-4]
(3)b[n] = c[n] + 6*Binomial(n,3)*a[n-3] + n*(n-1)*b[n-2]
(4)d[n] = c[n] / 3*n
Dabei sind Werte von negativem Index immer Null. Beginnend von n = 0 können nun die Werte für beliebige n berechnet werden. Dabei wird für ungerade n Gleichung (1) (in diesem Fall gibt es nur A-Graphen) berechnet und für ungerade n die Gleichungen (2) bis (4) (in diesem Fall gibt es keine A-Graphen).

n	A-Graphen	B-Graphen	C-Graphen	D-Graphen
0	0	0	0	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	12	12	1
5	30	0	0	0
6	0	1620	1260	70
7	5670	0	0	0
8	0	565320	464520	19355
9	2543940	0	0	0
10	0	390385800	335424600	11180820
11	2147121900	0	0	0
12	0	470878739100	415989812700	11555272575
13	3060711804150	0	0	0
14	0	909667780961940	819278536216140	19506631814670
15	6822508357214550	0	0	0
16	0	2641226277354865200	2412622018262055600	50262958713792825
17	4003679283806802584	0	0	0
18	0	4188763974121917680	11120749577250104016	205939806986113037

3.6. Erzeugung der Graphen

Speicherung der Graphen

Die Labels werden in einem globalen Array labels[] mit n (+1 wg. Index 0) Einträgen gespeichert. Die Kanten werden in einem globalen Array edges[] mit 3*n (+1 wg. Index 0) Einträgen gespeichert, wobei die Kanten des Graphen durch (edges[1],edges[2]), (edges[3],edges[4]), ... dargestellt sind. Dabei enthalten die Einträge edges[n] die Labels der Knoten und nicht die Knotennummern!
Zusätzlich sollen folgende Konventionen gelten:

- In einem A-Graph mit n Knoten ist der Label des Knotens vom Grad 2 in edges[3*n-1] gespeichert. In einem solchen Graph ist (edges[3*n-2],edges[3*n-1]) die letzte Kante.
- In einem B- oder C-Graph mit n Knoten ist der Label des positiven Pols in edges[3*n-2], die beiden inzidenten Kanten in (edges[3*n-5],edges[3*n-4]) und (edges[3*n-3],edges[3*n-2]) gespeichert. Der Label des negativen Pols befindet sich in der globalen Variable neg.

Ein auf diese Weise gespeicherter Graph soll „richtig dargestellt“ (properly represented) heißen.

Eine Erzeugungsprozedur heißt „effektiv“, wenn ein zufälliger und richtig dargestellter Graph des gewünschten Typs erzeugt wird.

Im folgenden wird die Existenz einer Funktion Random(n) angenommen, die eine Zufallszahl im Bereich von 1 bis n liefert, wobei n eine positive Zahl ist. Des weiteren sollten sich die Graphenanzahlen aus Abschnitt 3.5 in globalen Arrays A[], B[], C[] und D[] befinden.
Im Folgenden werden die Datentypen Integer und Cardinal (aus Modula-2) verwendet, um Zahlen mit und ohne Vorzeichen zu kennzeichnen ohne dabei aber auf die Bitbreite bestimmter Systeme einzugehen

Choose()-Prozedur:
Für die folgenden Graph-Erzeugungsprozeduren wird zudem eine Prozedur benötigt, die aus den Labels labels[1],...,labels[N] eine zufällige, geordnete Auswahl trifft:

```
// Choose-Funktion:
// Labels L[n-k+1] bis L[n] sollen eine zufällige geordnete Auswahl
// der Labels L[1] bis L[n] sein.
void Choose(const cardinal n, const cardinal k)
{
   cardinal x,i;
   integer  t;

   if(k >= n) return;
   for(i = n;i >= 1 + n - k;i--) {
      x = Random(i);
      t = labels[x];
      labels[x] = labels[i];
      labels[i] = t;
   }
}
```

Man beachte: Die vorher in labels[n-k+1] bis labels[n] gespeicherten Werte werden nicht gelöscht, sondern mit dem gewählten Label getauscht! Es fehlen am Ende also keine Labels und es kommen auch keine Labels mehrfach vor.

GraphA()-Prozedur:
Mit der oben angegebenen Choose()-Prozedur kann nun die Erzeugung eines A-Graph mit n Knoten implementiert werden: Ein Label wird ausgewählt, mit den restlichen n-1 Labels wird ein B-Graph erzeugt. Danach wird zu diesem B-Graph der ausgewählte n-te Knoten gemäß Operation 1 hinzugefügt:

```
// GraphA-Funktion:
// Erzeugung eines A-Graphen mit n Knoten basierend auf Lemma 1
// Voraussetzungen: n >=4 muß ungerade sein!
void GraphA(const cardinal n)
{
   cardinal neg;
   integer  t;

   Choose(n,1);
   GraphB(n - 1,neg);

   t = 3 * n - 5;
   pos = edges[t];
   // pos und neg sind die Labels des positiven und negativen Pols des
   // oben generieren B-Graphen.
   // Setzen der neuen Kanten:
   edges[t + 1] = neg;   edges[t + 2] = labels[n];
   edges[t + 3] = pos;   edges[t + 4] = labels[n];
}
```

Lemma A:
Für ungerade n >= 5 ist GraphA(n) effektiv, wenn GraphB(n-1,neg) effektiv ist.
Beweis:
GraphA() ruft Choose(n,1) auf und stellt somit sicher, daß labels[n] ein Element von labels[1] bis labels[n] ist. GraphB(n-1,neg) erzeugt mit den Labels labels[1] bis labels[n-1] einen B-Graphen mit n-1 Knoten. Der Rest der Prozedur fügt gemäß Operation 1 zwei Kanten hinzu.

Die hat die gleiche Wirkung wie das zufällige Erzeugen eines B-Graph mit n-1 Knoten, dem Hinzufügen eines neuen Knoten v mit Label labels[n] und dem linearen Update der alten Labels.

GraphB()-Prozedur:

Die GraphB(n,neg)-Prozedur erzeugt einen B-Graph mit n Knoten und der Label des negativen Pols wird in neg abgelegt. Dazu wird zufällig und gemäß der Wahrscheinlichkeiten aus der Anzahltabelle ein B-Graph vom Typ 1 (dies entspricht einem C-Graph), Typ 2 oder Typ 3 erzeugt:

```
// GraphB-Funktion:
// Erzeugung eines B-Graphen mit n Knoten basierend auf Lemma 2 bis 8
// Voraussetzungen: n >=4 muß gerade sein!
void GraphB(const cardinal n, cardinal& neg)
{
   cardinal x;

   x = Random(B[n]);
   if(x <= C[n]) {
      GraphC(n,neg);
   }
   else {
      if(x <= C[n] + n * (n - 1) * (n - 2) * A[n - 3]) {
         GraphB2(n,neg);
      }
      else {
         GraphB3(n,neg);
      }
   }
}

// GraphB2-Funktion:
// Erzeugung eines B-Graphen Typ 2 mit n Knoten basierend auf Lemma 2
void GraphB2(const cardinal n, cardinal& neg)
{
   cardinal w,x;
   integer  t;

   Choose(n,3);
   GraphA(n - 3);
   w   = labels[n - 2];
   neg = labels[n - 1];
   pos = labels[n];
   t   = 3 * n - 10;
   x   = edges[t];
   // Die Kanten des oben generierten A-Graphen liegen in edges[1] bis
   // edges[t]. x ist der Label des Knotens mit Grad 2.
   // Setzen der neuen Kanten:
   edges[t + 1] = x;     edges[t + 2] = w;
   edges[t + 3] = w;     edges[t + 4] = neg;
   edges[t + 5] = w;     edges[t + 6] = pos;
   edges[t + 7] = neg;   edges[t + 8] = pos;
}

// GraphB3-Funktion:
// Erzeugung eines B-Graphen Typ 3 mit n Knoten basierend auf Lemma 3
void GraphB3(const cardinal n, cardinal& neg)
{
   cardinal w,x;
   integer  t;

   Choose(n,2);
   GraphB(n - 2,neg);

   t   = 3 * n - 8;
   w   = edges[t];
   x   = neg;
   neg = labels[n - 1];
   pos = labels[n];
   // Im oben generierten Graph hat der positive Pol den Label w und
   // der negative Pol den Label x.
   // Setzen der neuen Kanten:
   edges[t + 1] = x;   edges[t + 2] = neg;
   edges[t + 3] = w;   edges[t + 4] = pos;
   edges[t + 5] = neg; edges[t + 6] = pos;
}
```

Lemma B:

Sei n >= 4 gerade. GraphB(n,neg) ist effektiv, wenn GraphC(n,neg) und GraphA(k) für k ungerade und 4 <= k < n effektiv sind und GraphB(k,neg) für k gerade und 4 <= k < n effektiv ist.

Beweis:

Es darf angenommen werden, daß GraphA(n-3) effektiv ist, da für n-3 kleiner 4 gilt: A[n-3] = 0 und die Wahrscheinlichkeit, daß GraphB2(n,neg) von GraphB(n,neg) aufgerufen wird, null ist. Ebenso darf angenommen werden, daß GraphB(n-2,neg) effektiv ist.

Die Effektivität von GraphB2(n,neg) und GraphB3(n,neg) folgt analog zu Lemma A mit den Operationen 2 und 3. Da die Operationen 2 und 3 exakt sind, folgt daraus, daß die einzige Voraussetzung für ein einheitliches zufälliges Ergebnis der Start mit einem zufälligen Graph des entsprechenden Typs ist, auf den dann die Operation randomisiert angewendet wird.

Die Überprüfung auf richtige Darstellung ist klar.

Die Anzahl der B-Graphen Typ 1 ist C[n], die des Typs 2 ist n*(n-1)*(n-2)*A[n-3] (nach Korollar 2). In GraphB() wird x zufällig von 1 bis B[n] gewählt. Daraus folgt für die Wahrscheinlichkeiten für den Aufruf von GraphC() und GraphB2(): C[n]/B[n] und n*(n-1)* (n-2)*A[n]/B[n]. Andernfalls wird GraphB3() aufgerufen.

Hieraus folgt nun das Lemma.

GraphC()-Prozedur:

Die GraphC(n,neg)-Prozedur erzeugt einen C-Graphen mit n Knoten. Der Label des negativen Pols wird in neg abgelegt. Dazu wird zufällig und gemäß der Wahrscheinlichkeiten aus der Anzahltabelle ein C-Graph vom Typ 1 bis Typ 5 erzeugt:

```
// GraphC-Funktion:
// Erzeugung eines C-Graphen mit n Knoten basierend auf Lemma 4 bis 8
// Voraussetzungen: n >=4 muß gerade sein!
void GraphC(const cardinal n, cardinal& neg)
{
   cardinal r1,r2,r3,r4;
   cardinal x;

   x = Random(C[n]);

   r1 = ((3 * n - 8) / 2) * n * A[n - 1];
   r2 = n * (n - 1) * B[n - 2] + r1;
   if(n == 4)
      r3 = x;
   else
      r3 = r2 + n * (n - 1) * (n - 2) * (n - 3) * D[n - 4] / 2;
   if(n <= 5)
      r4 = r3;
   else
      r4 = r3 + n * (n - 1) * (n - 2) * (n - 3) * (n - 4) * A[n - 5] / 2;

   // r[j] ist die Anzahl der C-Graphen mit n Knoten des Typs j.
   if(x <= r1)               GraphC1(n,neg);
   if((r1 < x) && (x <= r2)) GraphC2(n,neg);
   if((r2 < x) && (x <= r3)) GraphC3(n,neg);
   if((r3 < x) && (x <= r4)) GraphC4(n,neg);
   if(r4 < x)                GraphC5(n,neg);
}

// GraphC1-Funktion:
// Erzeugung eines C-Graphen Typ 1 mit n Knoten basierend auf Lemma 4
void GraphC1(const cardinal n, cardinal& neg)
{
   cardinal v,w,pos,x;
   integer  t;

   Choose(n,1);
   GraphA(n - 1);

   neg = edges[3 * n - 4];
   // neg ist der Label des Knotens mit Grad 2 im oben erzeugten A-Graphen

   x   = Random((3 * n - 8) / 2);
   v   = edges[2 * x - 1];
   w   = edges[2 * x];
   // v und w sind zwei zufällig gewählte Kanten, die nicht mit dem Knoten
   // vom Grad 2 inzident sind.
```

```
   t = 3 * n - 6;
   edges[2 * x - 1] = edges[t + 1];
   edges[2 * x]     = edges[t + 2];
   // edges[1] bis edges[t] beinhalten jetzt die Kanten des A-Graphen ohne
   // die Kante v-w.
   // Setzen der neuen Kanten:
   pos = labels[n];
   edges[t + 1] = v;
   edges[t + 2] = pos;
   edges[t + 3] = w;
   edges[t + 4] = pos;
}

// GraphC2-Funktion:
// Erzeugung eines C-Graphen Typ 2 mit n Knoten basierend auf Lemma 5
void GraphC2(const cardinal n, cardinal& neg)
{
   cardinal u,v,w,pos,x;
   integer t;

   Choose(n,2);
   GraphB(n - 2,neg);

   t   = 3 * n - 12;
   w   = edges[t + 1];
   x   = edges[t + 3];
   pos = edges[t + 4];
   u   = labels[n - 1];
   v   = labels[n];
   // pos ist der Label des positiven Pols des oben erzeugen B-Graphen.
   // w und x sind dessen Nachbarknoten. edges[1] bis edges[t] beinhalten
   // des B-Graphen.
   // Setzen der neuen Kanten:
   edges[t +  1] = w;
   edges[t +  2] = u;
   edges[t +  3] = x;
   edges[t +  4] = v;
   edges[t +  5] = u;
   edges[t +  6] = v;
   edges[t +  7] = u;
   edges[t +  8] = pos;
   edges[t +  9] = v;
   edges[t + 10] = pos;
}

// GraphC3-Funktion:
// Erzeugung eines C-Graphen Typ 3 mit n Knoten basierend auf Lemma 6
void GraphC3(const cardinal n, cardinal& neg)
{
   cardinal u,v,pos;
   integer  t;

   Choose(n,4);
   if(n > 4)
      GraphD(n - 4);

   neg = labels[n - 3];
   u   = labels[n - 2];
   v   = labels[n - 1];
   pos = labels[n];
   t = 3 * n - 12;

   // Setzen der neuen Kanten:
   edges[t +  1] = neg;   edges[t +  2] = u;
   edges[t +  3] = neg;   edges[t +  4] = v;
   edges[t +  5] = u;     edges[t +  6] = v;
   edges[t +  7] = u;     edges[t +  8] = pos;
   edges[t +  9] = v;     edges[t + 10] = pos;
}

// GraphC4-Funktion:
// Erzeugung eines C-Graphen Typ 4 mit n Knoten basierend auf Lemma 7
void GraphC4(const cardinal n, cardinal& neg)
{
   cardinal u,v,w,pos;
   integer  t;

   Choose(n,5);
   GraphA(n - 5);

   neg = labels[n - 4];
```

```
   w   = labels[n - 3];
   u   = labels[n - 2];
   v   = labels[n - 1];
   pos = labels[n];
   t = 3 * n - 16;

   // Die Kanten des A-Graphen liegen in edges[1] bis edges[t]. edges[t] ist
   // der Label des Knotens vom Grad 2.
   // Setzen der neuen Kanten:
   edges[t +  1] = edges[t]; edges[t +  2] = neg;
   edges[t +  3] = neg;      edges[t +  4] = w;
   edges[t +  5] = w;        edges[t +  6] = u;
   edges[t +  7] = w;        edges[t +  8] = v;
   edges[t +  9] = u;        edges[t + 10] = v;
   edges[t + 11] = u;        edges[t + 12] = pos;
   edges[t + 13] = v;        edges[t + 14] = pos;
}

// GraphC5-Funktion:
// Erzeugung eines C-Graphen Typ 5 mit n Knoten basierend auf Lemma 8
void GraphC5(const cardinal n, cardinal& neg)
{
   cardinal u,v,w,pos,x;
   integer  t;

   Choose(n,4);
   GraphB(n - 4,neg);

   t = 3 * n - 14;
   x   = edges[t];
   w   = labels[n - 3];
   u   = labels[n - 2];
   v   = labels[n - 1];
   pos = labels[n];

   edges[t +  1] = x;   edges[t +  2] = w;
   edges[t +  3] = w;   edges[t +  4] = u;
   edges[t +  5] = w;   edges[t +  6] = v;
   edges[t +  7] = u;   edges[t +  8] = v;
   edges[t +  9] = u;   edges[t + 10] = pos;
   edges[t + 11] = v;   edges[t + 12] = pos;
}
```

Lemma C:

Sei n >= 4 gerade. GraphC(n,neg) ist effektiv, wenn GraphA(k) für k ungerade sowie GraphB(k,neg) und GraphD(k) für k gerade effektiv sind.

Beweis:

Analog zu Lemma B unter Benutzung der Lemmata und Korollare 4 bis 8.

Die Forderung nach einem ungeordneten Paar von Labels für u und v in den Operationen 6 bis 8 wird von GraphC3() bis GraphC5() durch ein zufälliges, geordnetes Label-Paar erfüllt, was natürlich ausreichend ist.

GraphD()-Prozedur:

GraphD(n) erzeugt einen D-Graphen mit n Knoten, indem ein C-Graph mit n Knoten erzeugt wird und gemäß Operation 9 zwei Kanten hinzugefügt werden:

```
// GraphD-Funktion:
// Erzeugung eines D-Graphen mit n Knoten basierend auf Lemma 9
// Voraussetzungen: n >=4 muß gerade sein!
void GraphD(const cardinal n)
{
   cardinal neg;
   integer  t;

   GraphC(n,neg);
   t = 3 * n - 2;
   edges[t + 1] = neg;
   edges[t + 2] = edges[t];
}
```

Theorem:

Sei n >= 4 gerade. Der Algorithmus GraphD(n) generiert einen richtig dargestellten, kubischen Graph mit n Knoten.

Beweis:

Beobachtung: Wenn GraphC(n,neg) effektiv ist, dann ist GraphD(n) effektiv.
Beweis durch Induktion über n (n >= 4):
Induktionsannahme: Für 4 <= k <= n-1 und k gerade sind GraphB(k,neg), GraphC(k,neg) und GraphD(k) effektiv und für 5 <= k <= n-1 und k ungerade ist GraphA(k) effektiv.
Für n = 4: Aus Lemma C folgt, daß GraphC(4,neg) effektiv ist. Daher ist auch GraphD(4) effektiv. GraphB(4,neg) ist nach Lemma B ebenfalls effektiv.
Für n >= 5: Für ungerade n folgt aus der Induktionsannahme die Behauptung von Lemma A (da n-1 gerade) und somit: GraphA(n) ist effektiv.
Für gerade n folgt aus Lemma C und der Induktionsannahme: GraphC(n,neg) ist effektiv. Dies impliziert, daß GraphB(n,neg) und GraphD(n,neg) effektiv sind (nach der Beobachtung am Anfang und Lemma B).
Daher: GraphD(n) ist effektiv für alle geraden n >= 4.

RandCubic()-Prozedur:

RandCubic(n) erzeugt einen zufälligen kubischen Graphen mit n Knoten:

```
// RandCubic-Funktion:
void RandCubic(cardinal n)
{
   for(cardinal i = 1;i <= n;i++) labels[i] = i;
   GraphD(n);
}
```

4. Bipart()

Es soll nun ein zufälliger, bipartiter Graph mit einem gegebenen Grad für jeden Knoten erzeugt werden. Dies ist äquivalent zur zufälligen Erzeugung einer (0,1)-Matrix mit den Zeilensummen r[1], ..., r[n] (dies sind die Gradzahlen der Knoten in der roten Menge) und den Spaltensummen c[1], ..., c[m] (dies sind die Gradzahlen der Knoten in der blauen Menge):

- Sei s die Summe der r[j] und daher auch die der c[i]. Man nimmt nun s Bälle, von denen c[i] mit 1<=i<=m gelabelt sind und ordnet diese zufällig. (Also z.B. c[] = {2,3,1} => s=6 und Bälle mit folgenden Labels: 1,1,2,2,2,3).
- Die ersten r[1] Bälle werden nun in eine Schale mit dem Label 1 gelegt, die nächsten r[2] in eine Schale mit Label 2 usw.
- Sie M nun die (0,1)-Matrix, wobei die Eintrag (i,j) die Anzahl der Bälle mit Label j in der Schale i ist. Dann ist die i-te Zeilensumme wie gefordert r[i] und die j-te Spaltensumme c[j]. Ist M dann eine (0,1)-Matrix, d.h. keine Schale beinhaltet zwei Bälle des gleichen Labels, war der Versuch erfolgreich. Ansonsten kann ein neuer Versuch erfolgen.

Lemma:
War der oben angegebene Versuch erfolgreich, dann ist M eine zufällige (0,1)-Matrix mit der i-ten Zeilensumme r[i] und der j-ten Spaltensumme c[j].
Beweis:
Gegeben ist eine (0,1)-Matrix M mit bekannter Zeilen- und Spaltensumme. Wir betrachten die Anzahl der verschiedenen Anordnungen der Bälle, die zu M führen:
Die r[i] verschiedenen Schalen mit den Bällen mit Label i sind durch M festgelegt. Es gibt jedoch r[i]! verschiedene Möglichkeiten, die Bälle in die einzelnen Schalen zu legen. Nachdem Entschieden ist, in welche Schale ein Ball gelegt wird, gibt es c[j]! Möglichkeiten, die unterschiedlich gelabelten Bälle in der Schale zu ordnen.
Daher gibt es (Πr[i]! * Πc[j]!) unterschiedliche Möglichkeiten, die zu M führen. Da diese Zahl nur von r[] und c[] und nicht von M abhängig ist, folgt daraus das Lemma.
Implementation:

M wird als Adjazenzmatrix eines bipartiten Graphen interpretiert, wobei die Knoten einen gegebenen Grad besitzen:

```
// Initialisierung des Kantenarrays
// grad[] ist ein Array der Länge knoten mit den Gradzahlen der Knoten
// In menge[] wird eine Liste von grad[i] Labels des i-ten Knoten für
// i in aufsteigender Ordnung erstellt.
// Also z.B. 1,1,1,2,3,3 für Gradzahlen 3 1 2
void Initialisiere(cardinal knoten, cardinal* grad, cardinal* menge)
{
   cardinal i,j,k;

   k = 0;
   for(i = 1;i <= knoten;i++) {
      for(j = 0;j < grad[i];j++) {
         k++;
         menge[k] = i;
      }
   }
}

// Erzeugung eines bipartiten Graphen
// knotenRot und knotenBlau enthalten die Anzahlen der roten bzw. blauen Knoten
// kanten ist die Anzahl der Kanten
// gradRot[i] und gradBlau[i] enthalten die Gradzahl der i-ten blauen
//    bzw. roten Knotens
// Nach der Ausführung sind die Kanten des erzeugten Graphen in folgender
//    Weise gespeichert: (rot[1],blau[1]), ..., (rot[kanten],blau[kanten])
void Bipart(cardinal knotenRot, cardinal knotenBlau, cardinal kanten,
            cardinal* gradRot, cardinal* gradBlau, cardinal* rot, cardinal* blau)
{
   bool   good;
   cardinal i,t,x;
   cardinal zaehler = 0;

   Initialisiere(knotenRot,gradRot,rot);
   Initialisiere(knotenBlau,gradBlau,blau);

   good = false;
   while(!good) {
      // Neuen Versuch starten, einen bipartiten Graphen zu erzeugen
      Neu();
      good = true;
      i = kanten;
      while((i >= 1) && good) {
         // x wird zufällig aus {1,...,i} gewählt.
         x = Random(i);
         // Auf multi-edges überprüfen
         if(Verbunden(rot[i],blau[x])) {
            good = false;
         }
         else {
            t = blau[x];
            blau[x] = blau[i];
            blau[i] = t;
            Verbinde(rot[i],blau[i]);
            i--;
         }
      }
   }
}
```

Theorem:

Die Prozedur Bipart() erzeugt einen zufälligen bipartiten Graphen mit knotenRot roten Knoten, wobei der i-te Knoten (1<=i<=knotenRot) den Grad knotenRot[i] besitzt und knotenBlau blauen Knoten, wobei der j-te Knoten (1<=j<=knotenBlau) den Grad knotenBlau[j] besitzt.

Beweis:

Der Beweis folgt aus dem Lemma, wobei gradRot[i] = r[i] und gradBlau[j] = c[j].

Die Einträge von blau[] sind die Labels der Bälle, von denen gradBlau[i] mit i gelabelt sind. Die fortlaufende zufällige Auswahl von x aus {1,...,i} und der folgende Austausch von blau[x] und blau[i] erzeugen eine zufällige Permutation von blau[1], ..., blau[kanten], wobei i von kanten bis 1 rückwärts läuft. Die Ablage des i-ten permutierten Balls in die i-te Schale ist äquivalent zum Hinzufügen des geordneten Paares (rot[i],blau[i]) zur Kantenmenge des Graphen.

Ein mehrfaches Vorkommen eines Elementes (äquivalent zu mehreren Bällen mit gleichem Label in der gleichen Schale) in der Kantenmenge entspricht dabei einer mehrfach vorhandenen Kante. In diesem Fall wird der Versuch abgebrochen und muß neu gestartet werden.
Nach dem Lemma hat Bipart() bei Beendigung eine zufällige (0,1)-Matrix mit Zeilensummen gradRot[i] und Spaltensummen gradBlau[i] erzeugt. Dabei geben die Elemente der Kantenmenge die Koordinanten der 1-en an. Dies entspricht einem zufälligen bipartiten Graphen mit Knoten von gegebenem Grad und Kanten (rot[i],blau[i]) für 1<=i<=kanten.

5. RandGraph()

Ein Graph G kann als bipartiter Graph B aufgefaßt werden, wobei jeder blaue Knoten den Grad 2 besitzt. Um Schleifen und mehrfache Kanten in G zu vermeiden, ist es notwendig und ausreichend, vorauszusetzen, daß in B keine mehrfachen Kanten vorkommen und daß keine zwei blauen Knoten von B das gleiche Paar von Nachbarknoten besitzen. Die Knoten von G sind dann die roten Knoten von B. Zwei dieser Knoten sind dann adjazent in G, wenn sie zum gleichen blauen Knoten in B adjazent sind. Da die blauen Knoten von B gelabelt sind, führt dies zu einem Labeling der Kanten von G. Wenn der Graph m Kanten besitzt, können diese mit m! Möglichkeiten gelabelt werden, da die Labels der Knoten jede Kante eindeutig bestimmen. Daher kann ein zufälliger Graph G mit Knoten der Grade r[1], ..., r[n] generiert werden, indem zuerst mit Bipart() ein bipartiter Graph B mit 2*m Kanten und gradBlau[j]=2 für alle j erzeugt wird und das Ergebnis verworfen wird, wenn zwei blaue Knoten die gleichen zwei Nachbarknoten besitzen.
Dieser Prozeß kann jedoch mit einigen kleinen Änderungen vereinfacht werden:

```
// Erzeugung eines Graphen
// knoten enthält die Anzahl der Knoten
// kanten enthält die Anzahl der Kanten
// grad[i] ist der Grad des i-ten Knotens
// Nach der Ausführung sind die Kanten des Graphen in folgender Weise
//   gespeichert:
//   (rot[1],rot[2]), ..., (rot[2 * kanten - 1],rot[2 * kanten])
void RandGraph(cardinal knoten, cardinal kanten, cardinal* grad,  cardinal* rot)
{
   bool   good;
   cardinal i,t,x;
   cardinal zaehler = 0;

   Initialisiere(knoten,grad,rot);
   good = false;

   while(!good) {
      good = false;
      while(!good) {
         // Neuen Versuch starten, einen Graphen zu erzeugen
         Neu();
         good = true;
         i = kanten;
         while((i >= 1) && good) {
            x = Random(2 * i - 1);
            // Auf Schleifen überprüfen
            if(rot[x] == rot[2 * i]) {
               good = false;
            }
            else {
               // Auf multi-edges überprüfen
               if(Verbunden(rot[x],rot[2 * i])) {
                  good = false;
               }
               else {
                  t = rot[x];
                  rot[x] = rot[2 * i - 1];
                  rot[2 * i - 1] = t;
                  Verbinde(rot[2 * i - 1],rot[2 * i]);
                  i--;
               }
            }
         }
      }
   }
```

```
}
```

Theorem:
Die Prozedur RandGraph() erzeugt einen zufälligen Graph mit n Knoten der Grade grad[1], ..., grad[n].

Beweis:
Die Einträge von rot[] seien die Labels von 2*kanten unterscheidbaren Bällen, die geordnet sind. Wenn zwei Elemente von rot[] getauscht werden, werden die beiden entsprechenden Bälle getauscht. Die fortlaufende zufällige Auswahl von x aus {1, ..., 2*i-1} und der folgende Austausch von rot[2*i-1] und rot[x] erzeugt eine zufällige Paarung dieser Bälle für absteigende i von kanten bis 1. Die Label-Paare auf den Ball-Paaren sind (rot[1],rot[2]), ..., (rot[2*kanten-1],rot[2*kanten]). Die Ordnung dieser Paare ist nicht von Bedeutung; jedoch kann jede mögliche Menge von Paaren mit gleicher Wahrscheinlichkeit vorkommen.
Betrachtet man das Paar (rot[2*kanten-1],rot[2*kanten]) als Kante eines Graphen, ist der Erzeugungsversuch nur dann erfolgreich (d.h. RandGraph() liefert ein Ergebnis), wenn der Graph keine Schleifen (d.h. rot[2*i-1] != rot[2*i] für alle i) und keine mehrfachen Kanten besitzt.
Betrachte die Paarungen als eine Plazierung der 2*kanten unterscheidbaren Bälle in kanten ununterscheidbare Schalen: Wenn ein gegebener Graph erzeugt wurde, kann für jede Schale entschieden werden, welche Labels die Bälle in dieser Schale besitzen. Da kein Paar zwei Bälle mit gleichem Label besitzt und grad[i] Einträge von rot[] gleich i sind, gibt es grad[i] Möglichkeiten, die Bälle mit Label i in die einzelnen Schalen zu legen. Also gibt es insgesamt T= Πgrad[i]! Möglichkeiten für alle Bälle zusammen.
Da keine zwei Schalen Bälle mit dem gleichen Paar von Labels beinhalten, führt jede dieser T Möglichkeiten zu einer anderen Menge von Ball-Paaren, also zu einer unterschiedlichen Paarung von Bällen. Daher ist jeder Graph das Ergebnis von T verschiedenen Paarungen und jeder Graph hat die gleiche Wahrscheinlichkeit.