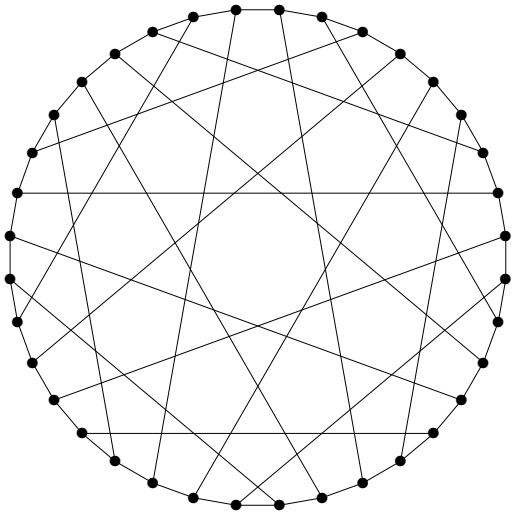
Erzeugung regulärer Graphen

Markus Meringer



Bayreuth, Januar 1996

Diplomarbeit bei Prof. Dr. Laue
Lehrstuhl II für Mathematik der Universität Bayreuth

Vorwort

Die Erzeugung vollständiger Listen nichtisomorpher regulärer Graphen gehört zu den ältesten Problemen der konstruktiven Kombinatorik. Bereits gegen Ende des letzten Jahrhunderts wurden Versuche unternommen, vollständige Listen 3-regulärer Graphen zu gegebener Knotenzahl aufzustellen. So gelang es Jan de Vries im Jahre 1889 alle 3-regulären Graphen mit 10 Knoten anzugeben. Mit der Entwicklung leistungsfähiger Rechner wurden auch auf diesem Gebiet entscheidende Fortschritte erzielt.

Die vorliegende Arbeit beschreibt einen Algorithmus zur Erzeugung der k-regulären Graphen mit n Knoten zu gegebenem k und n. Ausgangspunkt dafür war ein Verfahren zur Konstruktion 3-regulärer Graphen, das Gunnar Brinkmann in [Bri] vorstellt. Um auch für größeren Grad eine hohe Effizienz zu erreichen, verwende ich zudem Techniken, die sich schon in dem bestehenden MOLGEN - Generator bewährt haben.

Auf Grundlage dieser Methoden wurde ein Generator entwickelt, mit dem auch große Probleme in vertretbarer Zeit abgearbeitet werden können, so etwa die Konstruktion der 4-regulären Graphen mit 18 Knoten und der 5-regulären Graphen mit 16 Knoten. Es besteht zudem die Möglichkeit eine untere Grenze für die Taillenweite der erzeugten Graphen vorzuschreiben (auf der Titelseite ist ein 3-regulärer Graph mit 36 Knoten und Taillenweite 8 abgebildet). Auch mit dieser Restriktion werden bemerkenswerte Resultate erzielt. So wurden beispielsweise alle vier (5,5)-Cages berechnet. Damit ist dieses Problem vollständig gelöst.

Die eigentliche Intension dieser Arbeit war jedoch nicht die „Entdeckung neuer Graphen“. Vielmehr ist sie in folgenden größeren Rahmen eingebunden: Reguläre Graphen bilden zusammen mit ihren Automorphismengruppen die Eingabe für einen Algorithmus, der daraus via Homomorphieprinzip schlichte Graphen zu gegebener Gradpartition konstruiert. Dieses Verfahren wurde von Thomas Grüner im Rahmen einer Diplomarbeit [Grü] implementiert. In Hinsicht auf die Reichweite ist GRADPART bereits in seiner derzeitigen Form vergleichbaren Generatoren deutlich überlegen. Die Zukunft wird zeigen, ob es gelingt, das Programm dahingehend zu erweitern, daß vorgegebene Restriktionen berücksichtigt werden können, und es — ähnlich wie MOLGEN — eine wichtige Rolle bei interdisziplinären Forschungsprojekten spielen kann.

Mein Dank gilt vor allem Herrn Prof. Laue dafür, daß er mich mit einem Thema betraute, welches mein Interesse an der konstruktiven Kombinatorik [Lau] erweckte, und mir zudem die Möglichkeit eröffnete, meine Fertigkeiten am Rechner auszubauen. Desweiteren möchte ich mich bei Thomas Grüner für die Zusammenarbeit bei der Einbindung meines Programmteils in GRADPART und für das Korrekturlesen bedanken.

Markus Meringer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Grundlagen

Dieses Kapitel umfaßt einige grundlegende Definitionen und Sätze über Gruppen und Operationen (mehr dazu siehe [Ker]), sowie elementare Begriffe aus der Graphentheorie und bildet somit das Fundament für unsere Betrachtungen. Außerdem wird die, der ordnungstreuen Erzeugung zugrundeliegende lexikographische Ordnung vorgestellt.

1.1 Gruppen

1.1.1 Definition:
Sei G eine nichtleere Menge und $\circ : G \times G \longrightarrow G$ eine Abbildung. Das Paar $(G, \circ)$ heißt *Gruppe*, falls folgende Bedingungen erfüllt sind:

- $\forall a, b, c \in G : \circ(a, \circ(b, c)) = \circ(\circ(a, b), c)$.
- $\exists e \in G : \circ(a, e) = a \ \forall \ a \in G$.
- $\forall a \in G \ \exists a' \in G : \circ(a, a') = e$.

Das Element e heißt *Identität* der Gruppe und wird im folgenden mit id bezeichnet. Obiges a' heißt *Inverses* von a und wird mit a^{-1} bezeichnet. Die Abbildung $\circ$ wird als Verknüpfung zweier Gruppenelemente geschrieben: $ab := a \circ b := \circ(a, b)$.
Sei $(G, \circ)$ eine Gruppe und $U \subseteq G$. Das Paar $(U, \circ|_U)$ heißt *Untergruppe* von G, wenn gilt: $\circ(u, v^{-1}) \in U \ \forall u, v \in U$. Man schreibt $U \leq G$.

1.1.2 Beispiel:
Sei $\Omega := \{\omega_1, \omega_2, \ldots, \omega_n\}$ eine endliche Menge mit n Elementen. Die Menge aller bijektiven Abbildungen von Ω nach Ω bildet zusammen mit der Hintereinanderausführung eine Gruppe S_Ω, die *symmetrische Gruppe* vom Grad n auf Ω.
Für $\underline{n} := \{1, 2, \ldots, n\} \subseteq \mathbb{N}$ wird diese Gruppe mit $S_n := S_{\underline{n}}$ bezeichnet. Die Untergruppen von S_n heißen *Permutationsgruppen* vom Grad n, und deren Elemente werden als Permutationen bezeichnet. Für Permutationen $\pi, \tau \in S_n$ und $i \in \underline{n}$ gilt also $\pi(\tau(i)) = (\pi\tau)(i)$.

1.1.3 Definition:
Zu $n \in \mathbb{N}, n > 0$ heißt jede endliche Folge natürlicher Zahlen

$$\lambda := (\lambda_1, \lambda_2, ..., \lambda_m) \text{ mit } \sum_i \lambda_i = n$$

Partition von n (kurz $\lambda \models n$). Gilt darüber hinaus, daß $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m$, so nennen wir die Partition λ *kanonisch* (kurz $\lambda \vdash n$).

1.1.4 Bemerkung:
Durch eine Partition $\lambda = (\lambda_1, \lambda_2, ..., \lambda_m) \models n$, wird eine Einteilung der Menge $\underline{n}$ in disjunkte Teilmengen

$$\underline{n}_i^\lambda := \left\{ x \in \underline{n} \;\middle|\; \sum_{\nu=1}^{i-1} \lambda_\nu < x \leq \sum_{\nu=1}^{i} \lambda_\nu \right\}, \quad i = 1, ..., m$$

vorgenommen, wobei $\underline{n}_i^\lambda$ genau λ_i Elemente besitzt. Es gilt:

$$\underline{n} = \bigcup_{i=1}^{m} \underline{n}_i^\lambda.$$

Die zu λ gehörige *Younguntergruppe* von S_n definieren wir als

$$S_\lambda := \prod_{i=1}^{m} S_{\underline{n}_i^\lambda}.$$

1.2 Operationen

1.2.1 Definition:
Sei G eine endliche Gruppe und Ω eine nichtleere endliche Menge. Eine Abbildung

$$\varphi : \Omega \times G \longrightarrow \Omega, \qquad (\omega, g) \longmapsto \omega^g := \varphi(\omega, g)$$

heißt *Operation* von G auf Ω, falls φ mit der Verknüpfung in G verträglich ist und falls id die Elemente von Ω fest läßt. Es müssen also folgende Bedingungen gelten:

- $\forall \omega \in \Omega \; \forall g, g' \in G : \omega^{gg'} = (\omega^g)^{g'}$.
- $\forall \omega \in \Omega : \omega^{id} = \omega$.

Operiert G auf Ω, so wird diese Operation mit ${}_G\Omega$ bezeichnet. Die Operation heißt *transitiv*, falls gilt:

$$\forall \omega, \omega' \in \Omega \; \exists g \in G : \omega^g = \omega'.$$

1.2.2 Beispiel:
Ein sehr einfaches Beispiel liefert die Operation von S_n auf $\underline{n}$. Definieren wir $x^\pi := \pi^{-1}(x)$, dann sind für $\pi, \tau \in S_n$, $x \in \underline{n}$ die obigen Bedingungen erfüllt, denn $id^{-1}(x) = x$ und

$$x^{\pi\tau} = (\pi\tau)^{-1}(x) = (\tau^{-1}\pi^{-1})(x) = \tau^{-1}(\pi^{-1}(x)) = \tau^{-1}(x^\pi) = (x^\pi)^\tau.$$

1.2.3 Definition:
Sei ${}_G\Omega$ eine Operation, $U \subseteq G$, $\omega \in \Omega$ und $\Delta \subseteq \Omega$.

i) $C_G(\Delta) := \{g \in G \mid \delta^g = \delta \ \forall \delta \in \Delta\}$ heißt *Zentralisator* oder *punktweiser Stabilisator* von Δ in G bzgl. ${}_G\Omega$. Ist $\Delta = \{\delta\}$ so schreibt man auch $C_G(\delta)$.

ii) $N_G(\Delta) := \{g \in G \mid \delta^g \in \Delta \ \forall \delta \in \Delta\}$ heißt *Normalisator* oder *mengenweiser Stabilisator* von Δ in G bzgl. ${}_G\Omega$.

iii) $Fix_\Omega(U) := \{\omega \in \Omega \mid \omega^u = \omega \ \forall u \in U\}$ heißt *Fixpunktmenge* von $U \subseteq G$. Ein Element von $Fix_\Omega(U)$ heißt *Fixpunkt* von U.

iv) $\omega^G := \{\omega^g \in \Omega \mid g \in G\}$ heißt *Bahn* von ω unter ${}_G\Omega$.

v) $G\backslash\backslash\Omega := \{\omega^G \mid \omega \in \Omega\}$ ist die Menge der Bahnen der Operation ${}_G\Omega$.

1.2.4 Definition:
Sei G eine Gruppe, Ω eine Menge und ${}_G\Omega$ eine Operation. Eine Menge $T \subseteq \Omega$ heißt *vollständiges Repräsentantensystem* oder *Transversale* der Bahnen dieser Operation, falls gilt:

$$|B \cap T| = 1 \qquad \forall B \in G\backslash\backslash\Omega$$

Eine Transversale T enthält also aus jeder Bahn genau ein Element. Die Menge aller Transversalen wird mit $\mathcal{T}(G\backslash\backslash\Omega)$ bezeichnet.

Der Begriff der Bahn aus Definition 1.2.3 läßt sich auch über eine Äquivalenzrelation definieren. Dies hat den Vorteil, daß man elementare Eigenschaften von Bahnen ohne Rechnung erkennen kann.

1.2.5 Lemma:
Sei G eine Gruppe, Ω eine Menge und ${}_G\Omega$ eine Operation. Auf Ω werde folgende Relation definiert:

$$\omega \sim \omega' \quad :\Longleftrightarrow \quad \exists g \in G : \omega^g = \omega'$$

Dann ist $\sim$ eine Äquivalenzrelation, und die Bahnen der Operation ${}_G\Omega$ entsprechen den Äquivalenzklassen von $\sim$.

Beweis:
Die Eigenschaften einer Äquivalenzrelation sind einfach nachzurechnen:
$\omega \sim \omega$, da $\omega^{id} = \omega$; $\omega \sim \omega' \Rightarrow \exists g \in G : \omega^g = \omega' \Rightarrow \omega'^{g^{-1}} = (\omega^g)^{g^{-1}} = \omega^{gg^{-1}} = \omega \Rightarrow \omega' \sim \omega$;
$\omega \sim \omega', \omega' \sim \omega'' \Rightarrow \exists g, g' \in G : \omega^g = \omega', \omega'^{g'} = \omega'' \Rightarrow \omega^{gg'} = (\omega^g)^{g'} = \omega'^{g'} = \omega'' \Rightarrow \omega \sim \omega''$
für alle $\omega, \omega', \omega'' \in \Omega$. Die Beziehung zwischen Bahnen und Äquivalenzklassen folgt direkt aus der Definition von $\sim$. $\square$

1.2.6 Satz:
Sei G eine Gruppe, Ω eine Menge und ${}_G\Omega$ eine Operation. Dann gilt:

- Je zwei Bahnen von ${}_G\Omega$ sind entweder gleich oder disjunkt:

$$\forall \omega, \omega' \in \Omega : \quad \omega^G \cap \omega'^G \neq \emptyset \Longleftrightarrow \omega' \in \omega^G$$

- Ω ist die disjunkte Vereinigung der Bahnen von ${}_G\Omega$:

$$\forall T \in \mathcal{T}(G\backslash\backslash\Omega) : \quad \Omega = \overset{\circ}{\bigcup_{\omega \in T}} \omega^G$$

Beweis: Die Aussagen folgen aus Lemma 1.2.5 und aus den Eigenschaften von Äquivalenzklassen. □

1.2.7 Beispiel und Definition:
Sei G eine Gruppe, $U \leq G$ eine Untergruppe. Dann ist durch die Abbildung

$$G \times U \longrightarrow G, \qquad (g, u) \longmapsto g^u := gu$$

eine Operation von U auf G erklärt. Für $g \in G$ heißt die Bahn

$$gU := g^U = \{g^u \mid u \in U\} = \{gu \mid u \in U\}$$

Linksnebenklasse von U in G. Die Menge aller Linksnebenklassen von U in G wird mit G/U bezeichnet. Nach Satz 1.2.6 sind je zwei Linksnebenklassen gleich oder disjunkt, und G ist die disjunkte Vereinigung der Linksnebenklassen von U in G.

1.2.8 Definition:
Sei G eine Gruppe, Ω eine Menge und ${}_G\Omega$ eine Operation. Weiter sei $T \in \mathcal{T}(G\backslash\backslash\Omega)$ eine Transversale der Bahnen dieser Operation. Die Abbildung $\rho : \Omega \longrightarrow G$ heißt *fusionierende Abbildung* zu T, falls

$$\forall \omega \in \Omega : \quad \omega^{\rho(\omega)} \in T.$$

Das *fusionierende Element* $\rho(\omega)$ bildet somit ω auf den zugehörigen Bahnrepräsentanten ab.

1.2.9 Bemerkung:
Eine fusionierende Abbildung ρ wird insbesondere dann wichtig, wenn zwei Bahnen ω^G und ω'^G verglichen werden sollen. Anstatt zu testen, ob ω' in ω^G liegt, braucht man nur die beiden Elemente $\omega^{\rho(\omega)}$ und $\omega'^{\rho(\omega')}$ zu bestimmen. Dann ist

$$\omega^G = \omega'^G \Longleftrightarrow \omega^{\rho(\omega)} = \omega'^{\rho(\omega')}.$$

1.2.10 Bemerkung:
Sei G eine Gruppe, X eine Menge und ${}_GX$ eine Operation. Dann wollen wir folgende induzierte Operationen betrachten:

- G operiert auf der Menge $\binom{X}{p}$ der p-elementigen Teilmengen von X, $1 < p < |X|$ folgendermaßen:

$$\binom{X}{p} \times G \longrightarrow \binom{X}{p}, \qquad \{x_1, ..., x_p\}^g := \{{x_1}^g, ..., {x_p}^g\}.$$

- Sei Y eine Menge. Dann läßt sich auf folgende Weise eine Operation von G auf Y^X, der Menge der Abbildungen γ von X nach Y erklären:

$$Y^X \times G \longrightarrow Y^X, \qquad (\gamma, g) \longmapsto \gamma^g \text{ mit } (\gamma^g)(x) := \gamma(x^{g^{-1}}) \ \forall x \in X.$$

Wir wollen für den Fall ${}_G(Y^X)$ die Eigenschaften einer Operation nachrechnen:
$(\gamma^{gg'})(x) = \gamma(x^{(gg')^{-1}}) = \gamma(x^{g'^{-1}g^{-1}}) = \gamma((x^{g'^{-1}})^{g^{-1}}) = (\gamma^g)(x^{g'^{-1}}) = ((\gamma^g)^{g'})(x)$ und
$(\gamma^{id})(x) = \gamma(x^{id}) = \gamma(x) \ \forall x \in X, \ \forall \gamma \in Y^X, \forall g, g' \in G.$

1.2.11 Definition:
Sei $n \in \mathbb{N}$, $n > 1$. Wir definieren

$$X := \binom{\underline{n}}{2}$$

als die Menge der 2-elementigen Teilmengen von $\underline{n}$. Dann heißen die Elemente von

$$\mathcal{G}_n := \{0, 1\}^X$$

numerierte schlichte Graphen mit n Knoten. Zu $\Gamma \in \mathcal{G}_n$ nennen wir die Menge

$$E := \{e \in X \mid \Gamma(e) = 1\}$$

Kantenmenge von Γ. Zwei Knoten $i, j \in \underline{n}$, $i \neq j$ sind *benachbart*, falls $\Gamma(\{i, j\}) = 1$. Unter dem Grad von Knoten i verstehen wir die Anzahl der zu i benachbarten Knoten:

$$grad(i) := grad_\Gamma(i) := \sum_{j \in \underline{n}, j \neq i} \Gamma(\{i, j\}).$$

Haben alle Knoten gleichen Grad k, sprechen wir von einem *numerierten k-regulären Graphen* mit n Knoten. Die Menge dieser Graphen bezeichnen wir mit

$$\mathcal{R}_{n,k} := \{\Gamma \in \mathcal{G}_n \mid grad_\Gamma(i) = k \ \forall i \in \underline{n}\} \subseteq \mathcal{G}_n.$$

1.2.12 Definition:
Nach Bemerkung 1.2.10 erklären wir folgende Operation von S_n auf $\mathcal{G}_n$:

$$\mathcal{G}_n \times S_n \longrightarrow \mathcal{G}_n, \qquad (\Gamma, \pi) \longmapsto \Gamma^\pi \quad \text{mit}$$
$$\Gamma^\pi(\{i,j\}) := \Gamma(\{\pi(i), \pi(j)\}) \quad \forall \{i,j\} \in X.$$

Zwei numerierte schlichte Graphen $\Gamma, \Gamma' \in \mathcal{G}_n$ sind zueinander *isomorph*, wenn sie in der gleichen Bahn unter dieser Operation liegen. Die Elemente von $S_n \backslash\backslash \mathcal{G}_n$ heißen *schlichte Graphen* mit n Knoten.
Bei der beschriebenen Operation bleibt Regularität erhalten. Mit der gleichen Abbildungsvorschrift können wir auch eine Operation von S_n auf $\mathcal{R}_{n,k}$ definieren. Die Elemente von $S_n \backslash\backslash \mathcal{R}_{n,k}$ nennen wir *k-reguläre Graphen* mit n Knoten.

1.2.13 Bemerkung:
Einem numerierten schlichten Graphen $\Gamma \in \mathcal{G}_n$ entspricht seine *Adjazenzmatrix* $A = (a_{i,j})_{1 \le i,j \le n}$ mit

$$a_{i,j} := \begin{cases} \Gamma(\{i,j\}), & \text{falls } i \neq j, \\ 0, & \text{sonst.} \end{cases}$$

Die Adjazenzmatrix A ist also eine symmetrische $n \times n$-Matrix mit Einträgen Null oder Eins und Nullen auf der Diagonale. Umgekehrt gibt es zu jeder solchen Matrix A einen numerierten schlichten Graphen Γ, so daß A die Adjazenzmatrix von Γ ist. Wir können die Menge dieser Matrizen mit $\mathcal{G}_n$ identifizieren.
Die Adjazenzmatrix wird bei den folgenden Betrachtungen eine zentrale Rolle als Datenstruktur für numerierte Graphen spielen. Insbesondere werden wir dabei soweit wie möglich die Symmetrie dieser Matrizen ausnutzen, ohne jedesmal erneut darauf hinzuweisen. So müssen wir beispielsweise beim Vergleich zweier Adjazenzmatrizen nur die oberen rechten Hälften betrachten. Insbesondere werden wir zu $\pi \in S_n$ Vergleiche von A und A^π durchführen. Dabei ist A^π die Adjazenzmatrix von Γ^π:

$$A^\pi = (a_{\pi(i),\pi(j)})_{1 \le i,j \le n}.$$

Wir erhalten A^π durch simultanes Vertauschen der Zeilen und Spalten von A.

1.2.14 Bezeichnungen:
Da diese Arbeit ausschließlich schlichte Graphen behandelt, können wir im folgenden auf das Adverb „schlicht“ verzichten. Es ist also $S_n \backslash\backslash \mathcal{G}_n$ die Menge der Graphen und $\mathcal{G}_n$ die Menge der numerierten Graphen mit n Punkten.
Ein numerierter Graph $\Gamma \in \mathcal{G}_n$ ist durch die Menge seiner Kanten $E = \{e_1, ..., e_t\}$ eindeutig bestimmt. Wir werden diese Eigenschaft nutzen, um Γ durch seine Kantenmenge zu beschreiben: $\Gamma = \{e_1, ..., e_t\}$. Zu $\pi \in S_n$ ist dann $\Gamma^\pi = \{{e_1}^\pi, ..., {e_t}^\pi\}$.
Für zwei numerierte Graphen Γ und Γ' bedeutet $\Gamma \subseteq \Gamma'$, daß $E \subseteq E'$, wobei E, E' die zugehörigen Kantenmengen bezeichnen.
Verwenden wir für eine Kante $e = \{v, w\} \in E$ die Schreibweise $e = (v, w)$ (mit „runden Klammern“), dann setzen wir voraus, daß $v < w$ gilt.

1.3 Graphentheorie

1.3.1 Definitionen:
Sei $\Gamma \in \mathcal{G}_n$ ein numerierter Graph. Die Kanten $e_1, e_2, ..., e_q$ von Γ bilden eine *Kantenfolge*, wenn für $i = 2, 3, ..., q-1$ die Kante e_i den einen ihrer Knoten mit e_{i-1} und den anderen mit e_{i+1} gemeinsam hat. Ein *Kantenzug* ist eine Kantenfolge, die keine Kante mehrfach enthält. Ein *Weg* ist ein Kantenzug, der keinen Knoten mehrfach enthält. Einen die Knoten v und w verbindenden Weg beschreiben wir durch die von ihm durchlaufenen Knoten: $W = (v, v_1, v_2, ..., v_{q-1}, w)$. Die beiden Knoten v und w nennen wir *Endpunkte* von W. Die Anzahl q der Kanten des Weges W wird als seine Länge $l(W)$ bezeichnet: $l(W) = q$. Die Länge des kürzesten Weges, der v und w verbindet, heißt *Abstand* von v und w:

$$dist_\Gamma(v, w) := \min\{l(W) \mid W \text{ Weg, der } v \text{ und } w \text{ verbindet}\},$$

falls ein solcher Weg existiert. Anderenfalls setzen wir $dist_\Gamma(v, w) = \infty$ und speziell für $v = w$: $dist_\Gamma(v, v) := 0$. Weiterhin definieren wir für eine Kante $\{u, u'\}$ den Abstand von v und $\{u, u'\}$ als

$$dist_\Gamma(v, \{u, u'\}) := \min\{dist_\Gamma(v, u), dist_\Gamma(v, u')\}.$$

Wir nennen Γ *zusammenhängend*, wenn je zwei Knoten durch einen Weg miteinander verbunden sind.

1.3.2 Bemerkung:
Die für Knotenpaare erklärte Relation, durch einen Weg verbunden zu sein, ist eine Äquivalenzrelation. Die Äquivalenzklassen hierzu heißen *Zusammenhangskomponenten*. Γ ist also genau dann zusammenhängend, wenn nur eine Zusammenhangskomponente existiert. Einen Knoten vom Grad 0 nennen wir *triviale Zusammenhangskomponente*.

1.3.3 Definition:
Die Menge der zusammenhängenden k-regulären numerierten Graphen mit n Knoten nennen wir

$$\mathcal{R}^*_{n,k} := \{\Gamma \in \mathcal{R}_{n,k} \mid \Gamma \text{ zusammenhängend}\}.$$

Mit $\mathcal{G}^*_n$ wollen wir die numerierten Graphen $\Gamma \in \mathcal{G}_n$ bezeichnen, die genau eine nichttriviale Zusammenhangskomponente besitzen.

1.3.4 Definition:
Sind in $\Gamma \in \mathcal{G}_n$ die beiden Endpunkte v und w eines Weges $W = (v, v_1, v_2, ..., v_{q-1}, w)$ durch eine Kante $e = \{v, w\}$ verbunden, so bildet $K := W \cup \{e\}$ einen *Kreis* der Länge $l(K) = q + 1$. Gibt es in Γ keinen Kreis kleinerer Länge, so spricht man von einem *Taillenkreis*. Die Länge eines Taillenkreises bezeichnen wir als *Taillenweite* oder *Girth* des Graphen:

$$girth(\Gamma) := \min\{l(K) \mid K \text{ Kreis in } \Gamma\}.$$

Besitzt Γ mindestens einen Kreis, so ist $girth(\Gamma)$ als Minimum einer endlichen Menge wohldefiniert. Anderenfalls setzen wir $girth(\Gamma) := \infty$.

1.3.5 Definition:
Ein zusammenhängender numerierter Graph, der keinen Kreis besitzt, ist eine *Baum*. Die Knoten vom Grad 1 heißen *Blätter*, alle anderen nennen wir *innere Knoten*. Ist ein Knoten als *Wurzel* ausgezeichnet, sprechen wir auch von einem *Wurzelbaum*. Wir wollen diese Begriffe auch für kreisfreie $\Gamma \in \mathcal{G}_n^*$ verwenden, wobei Wurzel immer ein Knoten vom Grad > 0 ist.

1.3.6 Bemerkung:
Sei $\Gamma \in \mathcal{G}_n^*$ ein Wurzelbaum mit Wurzel 1 und v_0 ein Knoten mit $dist_\Gamma(v_0, 1) = d$. Ist $0 < d < \infty$, dann gibt es genau einen Nachbarn v von v_0 mit $dist_\Gamma(v, 1) = d - 1$. Dieser Knoten heißt Vater von v_0. Für alle übrigen Nachbarn w von v_0 gilt $dist_\Gamma(w, 1) = d + 1$. Diese Knoten sind die Söhne von v_0. Die Menge der Söhne von v_0 bezeichnen wir mit $succ_1(v_0)$. Weiter definieren wir rekursiv für $i = 1, ..., n-1$:

$$succ_{i+1}(v_0) := \bigcup_{u \in succ_i(v_0)} succ_1(u).$$

Dann nennen wir die Elemente von $succ(v_0) := \bigcup_{i=1}^{n} succ_i(v_0)$ Nachfolger von v_0.

1.3.7 Definition:
Seien $n, k, t \in \mathbb{N}$, $n > 1$, $k < n$, $t \geq 3$. Wir definieren:

$$\begin{aligned}
\mathcal{G}_{n,k} &:= \{\Gamma \in \mathcal{G}_n \mid grad_\Gamma(i) \leq k \ \forall i \in \underline{n}\},\\
\mathcal{G}^*_{n,k} &:= \{\Gamma \in \mathcal{G}^*_n \mid grad_\Gamma(i) \leq k \ \forall i \in \underline{n}\},\\
\mathcal{R}_{n,k,t} &:= \{\Gamma \in \mathcal{R}_{n,k} \mid girth(\Gamma) \geq t\},\\
\mathcal{R}^*_{n,k,t} &:= \{\Gamma \in \mathcal{R}^*_{n,k} \mid girth(\Gamma) \geq t\}.
\end{aligned}$$

1.3.8 Bemerkung:
Taillenweite sowie Anzahl von Zusammenhangskomponenten sind Invariante bzgl. der Operation aus Definition 1.2.12. Ziel dieser Arbeit wird es sein, folgende Mengen zu gegebenen n, k, t zu bestimmen:

- $S_n \backslash\backslash \mathcal{R}_{n,k}$, die k-regulären Graphen mit n Knoten,
- $S_n \backslash\backslash \mathcal{R}^*_{n,k}$, die zusammenhängenden k-regulären Graphen mit n Knoten und
- $S_n \backslash\backslash \mathcal{R}^*_{n,k,t}$, die zusammenhängenden k-regulären Graphen mit n Knoten und Taillenweite mindestens t.

Wir wollen jeweils ein vollständiges Repräsentantensystem angeben. Im folgenden wird beschrieben, wie die Repräsentanten ausgewählt werden.

1.4 Lexikographische Ordnung

1.4.1 Definition:
Die Menge $X = \binom{n}{2}$ läßt sich wie folgt ordnen: Seien $e_1 = (v_1, w_1)$, $e_2 = (v_2, w_2) \in X$, dann ist

$$e_1 < e_2 \quad :\Longleftrightarrow \quad v_1 < v_2 \ \vee \ (v_1 = v_2 \ \wedge \ w_1 < w_2).$$

Dann ist auch $\mathcal{G}_n$ lexikographisch geordnet. Für zwei numerierte Graphen $\Gamma, \Gamma' \in \mathcal{G}_n$ mit $\Gamma = \{e_1, ..., e_t\}$ und $\Gamma' = \{e'_1, ..., e'_{t'}\}$ gelte $e_1 < ... < e_t$, $e'_1 < ... < e'_{t'}$. Dann ist

$$\begin{aligned} \Gamma < \Gamma' \quad :\Longleftrightarrow \quad & (\exists i \leq \min\{t, t'\} : e_j = e'_j \forall j < i \wedge e_i < e'_i) \\ & \vee (t < t' \wedge e_j = e'_j \forall j \leq t). \end{aligned}$$

1.4.2 Definition:
Auf 0/1-Vektoren der Länge l definieren wir die Ordnung „$\succ$“:

$$(a_1, ..., a_l) \succ (a'_1, ..., a'_l) \quad :\Longleftrightarrow \quad \exists i \leq l : a_j = a'_j \forall j < i \wedge a_i > a'_i.$$

Zwei numerierte Graphen $\Gamma, \Gamma' \in \mathcal{G}_n$ kann man auch vergleichen, indem man die Einträge ihrer Adjazenzmatrizen A und A' zeilenweise hintereinanderschreibt und sie als 0/1-Vektoren der Länge n^2 betrachtet:

$$A \succ A' \quad :\Longleftrightarrow \quad (a_{11}, ..., a_{1n}, ..., a_{n1}, ..., a_{nn}) \succ (a'_{11}, ..., a'_{1n}, ..., a'_{n1}, ..., a'_{nn}).$$

1.4.3 Bemerkung:
Auf den ersten Blick mag Definition 1.4.2 überflüssig erscheinen, da es sich ebenfalls um eine lexikographische Ordnung handelt. Tatsache ist, daß für die Anwendung am Rechner in erster Linie die Adjazenzmatrix als Datenstruktur für numerierte Graphen Verwendung findet. Deshalb liegt bei der Entwicklung von Algorithmen die Ordnung „$\succ$“ zugrunde. Für theoretische Belange, wie die Formulierung von Sätzen und Beweisführung hat sich Definition 1.4.1 als eleganter erwiesen. Eine Aussage über die Äquivalenz der beiden Ordnungen liefert folgendes

1.4.4 Lemma:
Haben zwei numerierte Graphen $\Gamma, \Gamma' \in \mathcal{G}_n$ gleiche Anzahl von Kanten, dann gilt für die Adjazenzmatizen A von Γ und A' von Γ':

$$A \succ A' \quad \Longleftrightarrow \quad \Gamma < \Gamma'.$$

Beweis:

„$\Rightarrow$“: Sei (i_0, j_0) die erste Stelle, an der sich A und A' unterscheiden, d.h. $a_{i_0 j_0} = 1 > 0 = a'_{i_0 j_0}$. Dann ist $i_0 < j_0$. Alle Kanten $e_j, e'_j < (i_0, j_0)$ in Γ bzw. Γ' rühren von Einsen vor Stelle (i_0, j_0) in A bzw. A'. Sei m die Anzahl solcher Kanten, dann ist $e_j = e'_j \ \forall j \leq m$. Für $e_{m+1} = (i_0, j_0)$ gilt $e_{m+1} \in \Gamma$, aber $e_{m+1} \notin \Gamma'$. Da gleiche Anzahl von Kanten besteht, muß Γ' noch eine Kante $e'_{m+1} > e_{m+1}$ besitzen. Wir haben für Γ die Kanten $e_1 < ... < e_m < e_{m+1} < ...$ und für Γ' die Kanten $e_1 < ... < e_m < e'_{m+1} < ...$, also $\Gamma < \Gamma'$.

„$\Leftarrow$“: Da Γ und Γ' gleiche Anzahl von Kanten haben $\exists m : e_i = e'_i \ \forall i < m, e_m < e'_m$. Sei $e_m = (i_0, j_0)$. Die Kanten e_j, bzw. e'_j mit $j \geq m$ bewirken keine Einsen vor Stelle (i_0, j_0) in A und A'. Vor Stelle (i_0, j_0) sind A und A' gleich. Weiter gilt $a_{i_0 j_0} = 1 > 0 = a'_{i_0 j_0}$, somit $A \succ A'$.

□

1.4.5 Bemerkung:
Die Aussage von Lemma 1.4.4 wird falsch, wenn verschiedene Anzahlen von Kanten vorliegen. Ein Beispiel liefern die beiden numerierten Graphen mit 3 Knoten

$$\Gamma = \{(1,2)\}, \ A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

und

$$\Gamma' = \{(1,2),(1,3)\}, \ A' \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Es ist $\Gamma < \Gamma'$ und $A \prec A'$. In allen Fällen, in denen wir Graphen lexikographisch vergleichen müssen, werden diese jeweils gleiche Anzahl von Kanten besitzen.

1.4.6 Definition:
Sei $\Gamma \in \mathcal{G}_n$. Wir nennen Γ *kanonisch* (bzgl. der lexikographischen Ordnung „$<$“), falls Γ minimal in seiner Bahn unter S_n ist:

$$\Gamma \text{ kanonisch} \quad :\Longleftrightarrow \quad \forall \pi \in S_n \ : \ \Gamma \leq \Gamma^\pi.$$

Nach Lemma 1.4.4 ist dies auch gleichbedeutend mit $A \succeq A^\pi \ \forall \pi \in S_n$, wobei A die Adjazenzmatrix von Γ ist. Die Menge der kanonischen Bahnrepräsentanten

$$rep_<(S_n \backslash\backslash \mathcal{G}_n) := \{\Gamma \in \mathcal{G}_n \mid \forall \pi \in S_n : \Gamma \leq \Gamma^\pi\}$$

ist ein kanonisches Repräsentantensystem von $S_n \backslash\backslash \mathcal{G}_n$. Wir werden die Mengen aus Bemerkung 1.3.8 bestimmen, indem wir jeweils das kanonische Repräsentantensystem (bzgl. der lexikographischen Ordnung „$<$“) angeben:

$$\begin{aligned} rep_<(S_n \backslash\backslash \mathcal{R}_{n,k}) &:= \{\Gamma \in \mathcal{R}_{n,k} \mid \forall \pi \in S_n : \Gamma \leq \Gamma^\pi\}, \\ rep_<(S_n \backslash\backslash \mathcal{R}^*_{n,k}) &:= \{\Gamma \in \mathcal{R}^*_{n,k} \mid \forall \pi \in S_n : \Gamma \leq \Gamma^\pi\}, \\ rep_<(S_n \backslash\backslash \mathcal{R}^*_{n,k,t}) &:= \{\Gamma \in \mathcal{R}^*_{n,k,t} \mid \forall \pi \in S_n : \Gamma \leq \Gamma^\pi\}. \end{aligned}$$

Kapitel 2

Existenzkriterien für reguläre Graphen

In diesem Kapitel werden eine Aussage über die Existenz k-regulärer Graphen mit n Knoten gemacht und kurz die trivialen Fälle vorgestellt. In Abschnitt 2.2 wird dann eine notwendige Bedingung für die Existenz eines k-regulären Graphen mit n Knoten und vorgegebener Taillenweite erarbeitet. Die dabei gewonnenen Erkenntnisse werden später auch in Kapitel 3 angewendet. Der Begriff des (k,t)-Cage wird eingeführt und eine Übersicht über bekannte Cages gegeben.

2.1 Reguläre Graphen

2.1.1 Bemerkung:
Seien $n, k \in \mathbb{N}$, $n \geq 2$, $\Gamma \in \mathcal{R}_{n,k}$. Die Anzahl der Kanten von Γ ist $|\Gamma| = \frac{nk}{2} \in \mathbb{N}$. Sind n und k ungerade, dann ist $\mathcal{R}_{n,k} = \emptyset$.
Ist $k = 0$, so sprechen wir vom *leeren Graphen* mit n Knoten. Für $k = 1$ existiert $\Gamma \in \mathcal{R}_{n,1}$, falls n gerade. Γ besitzt dann keine Kreise und ist nicht zusammenhängend, wenn $n > 2$. Für gerades n ist $|S_n \backslash\backslash \mathcal{R}_{n,1}| = 1$. Für $k = 2$, $n > 2$ ist $\Gamma \in \mathcal{R}^*_{n,2}$ ein Kreis der Länge n. Es ist $|S_n \backslash\backslash \mathcal{R}^*_{1,n}| = 1$. Für $k = 3$ bezeichnen wir 3-reguläre Graphen auch als *kubische* Graphen.

2.1.2 Definition:
Sei $k \geq 1$, $n = k+1$. Dann heißt der k-reguläre numerierte Graph, bei dem je zwei Knoten durch eine Kante verbunden sind *vollständiger Graph* $K_n := \binom{n}{n-1}$ (siehe Abb. 2.1). Es gibt zu jedem $n \geq 2$ genau einen vollständigen Graphen: $|\mathcal{R}_{n,n-1}| = |S_n \backslash\backslash \mathcal{R}_{n,n-1}| = 1$.

2.2 Reguläre Graphen gegebener Taillenweite

2.2.1 Definition:
Seien $k, t \in \mathbb{N}$, $k \geq 2$, $t \geq 3$. Wir definieren

$$f(k,t) := \min\{n \in \mathbb{N} \mid \mathcal{R}_{n,k,t} \neq \emptyset\}.$$

2.2.2 Bemerkung:
Es ist $f(2,t) = t$ für $t \geq 3$. Für $t = 3$ ist $f(k,3) = k+1$, da $K_{k+1} \in \mathcal{R}_{k+1,k,3} \neq \emptyset$. Für $t = 4$ ist $f(k,4) = 2k$, denn der bipartite numerierte Graph B_{2k} mit $2k$ Knoten hat Taillenweite 4: $B_{2k} := \{\{i,j\} \mid 1 \leq i \leq k \wedge k < j \leq 2k\} \in \mathcal{R}_{2k,k,4} \neq \emptyset$ (vgl. Abb 2.2). Begründung dafür, daß die Knotenzahl in beiden Fällen jeweils minimal ist, liefert Satz 2.2.4. Für $t > 4$ sind Funktionswerte $f(k,t)$ nicht so einfach zu bestimmen und in fast allen Fällen noch unbekannt (siehe Tabelle 2.2 und Bemerkung 2.2.8).

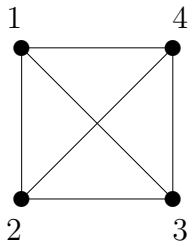


Abbildung 2.1: Der K_4.

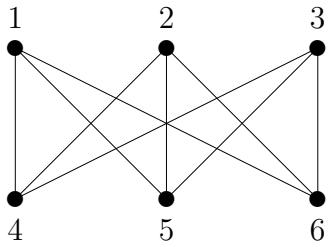


Abbildung 2.2: Der B_6.

2.2.3 Definition:
Sei $k \geq 2$, $t \geq 3$, $n_0 = f(k,t)$. Dann heißen die Elemente von $S_{n_0}\backslash\backslash\mathcal{R}_{n_0,k,t}$ (k,t)-*Cages*. Allgemeiner nennen wir für $n \geq n_0$ die Elemente von $S_n\backslash\backslash\mathcal{R}_{n,k,t}$ (k,t)-*Graphen*.

Eine untere Grenze für $f(k,t)$ liefert

2.2.4 Satz:
Für $k, t \in \mathbb{N}$, $k \geq 2$, $t \geq 3$ sei

$$f_0(k,t) := \begin{cases} 1 + k\sum_{i=1}^{\frac{t-1}{2}}(k-1)^{i-1}, & \text{falls } t \text{ ungerade,} \\ 2\sum_{i=1}^{\frac{t}{2}}(k-1)^{i-1}, & \text{sonst.} \end{cases}$$

Dann ist $f(k,t) \geq f_0(k,t)$.

Beweis: Sei $\Gamma \in \mathcal{R}_{n,k,t}$.

i) $t = 2d+1$ ungerade. Wir definieren für $i = 0, ..., d$ disjunkte Mengen

$$niv(i) := niv_{\Gamma,1}(i) := \{v \in \underline{n} \mid dist(1,v) = i\}.$$

Für $1 \leq i \leq d$ gilt: Jeder Knoten $w \in niv(i)$ besitzt genau einen Nachbarn in $niv(i-1)$ $(*)$. Denn wäre w zu zwei Knoten $v_1, v_2 \in niv(i-1)$ benachbart, dann hätte Γ einen Kreis der Länge

$$dist(1, v_1) + dist(v_1, w) + dist(w, v_2) + dist(v_2, 1) = 2i \leq 2d < t,$$

was einen Widerspruch bedeutet.

Weiterhin gilt für $1 \leq i < d$: Jeder Knoten $w \in niv(i)$ besitzt genau $k-1$ Nachbarn in $niv(i+1)$ $(**)$. Wäre w zu einem Knoten $w' \in niv(i)$ benachbart, dann hätte Γ einen Kreis der Länge

$$dist(1, w) + dist(w, w') + dist(w', 1) = 2i + 1 < 2d + 1 = t,$$

ein Widerspruch. Die beiden Eigenschaften $(*)$ und $(**)$ bezeichnen wir im folgenden mit „Baumstruktur bis $niv(d)$“ (siehe auch Abbildung 2.3).

Wir zeigen durch Induktion: $|niv(i)| = k(k-1)^{i-1}$, $1 \leq i \leq d$. Für $i = 1$ ist die Behauptung klar. Sei $i > 1$. Für die Anzahl der Kanten, die Knoten aus $niv(i-1)$ mit Knoten aus $niv(i)$ verbinden gilt:

$$|niv(i-1)|(k-1) = |niv(i)|.$$

Nach Induktionsvoraussetzung ist dann

$$|niv(i)| = k(k-1)^{i-2}(k-1) = k(k-1)^{i-1}.$$

Γ besitzt also mindestens

$$\sum_{i=0}^{d} |niv(i)| = 1 + \sum_{i=1}^{d} k(k-1)^{i-1}$$

Knoten.

ii) Sei nun $t = 2d + 2$ gerade. Wir wählen einen Nachbarn z von Knoten 1 aus und definieren für $i = 0, ..., d$ die disjunkten Mengen

$$niv(i) := niv_{\Gamma,\{1,z\}}(i) := \{v \in \underline{n} \mid dist(v, \{1, z\}) = i\}.$$

Für $1 \leq i \leq d$ gilt: Jeder Knoten $w \in niv(i)$ besitzt genau einen Nachbarn in $niv(i-1)$. Denn wäre w zu zwei Knoten $v_1, v_2 \in niv(i-1)$ benachbart, dann hätte Γ einen Kreis der Länge $\leq$

$$\begin{aligned} &dist(1, z) + dist(v_1, \{1, z\}) + dist(v_1, w) + dist(w, v_2) + dist(v_2, \{1, z\}) \\ &= 2i + 1 \leq 2d + 1 < t, \end{aligned}$$

ein Widerspruch.

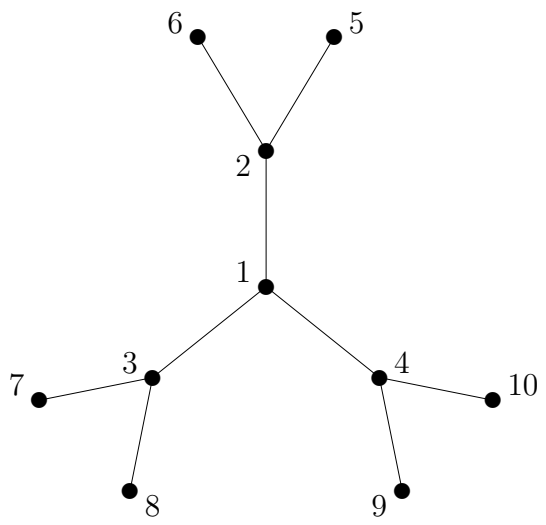


Abbildung 2.3: Der Baum $T_{3,5}$.

Weiterhin gilt für $1 \leq i < d$: Jeder Knoten $w \in niv(i)$ besitzt genau $k-1$ Nachbarn in $niv(i+1)$. Denn wäre w zu einem Knoten $w' \in niv(i)$ benachbart, dann hätte Γ einen Kreis der Länge $\leq$

$$dist(1,z) + dist(w,\{1,z\}) + dist(w,w') + dist(w',\{1,z\}) = 2i + 2 < 2d + 2 = t,$$

ein Widerspruch. Wir haben Baumstruktur bis $niv(d)$ (siehe auch Abbildung 2.4).

Behauptung: $|niv(i)| = 2(k-1)^i$, $0 \leq i \leq d$. Für $i = 0$ ist die Behauptung klar. Sei $i > 0$. Für die Anzahl der Kanten, die Knoten aus $niv(i-1)$ mit Knoten aus $niv(i)$ verbinden gilt:

$$|niv(i-1)|(k-1) = |niv(i)|.$$

Nach Induktionsvoraussetzung ist

$$|niv(i)| = 2(k-1)^{i-1}(k-1) = 2(k-1)^i.$$

Γ besitzt also mindestens

$$\sum_{i=0}^{d} |niv(i)| = \sum_{i=0}^{d} 2(k-1)^i$$

Knoten.

□

In dem Beweis betrachten wir Bäume $T_{k,t}$, wie sie in Abbildung 2.3 und 2.4 gezeigt werden. Wir wollen $T_{k,t}$ genau definieren:

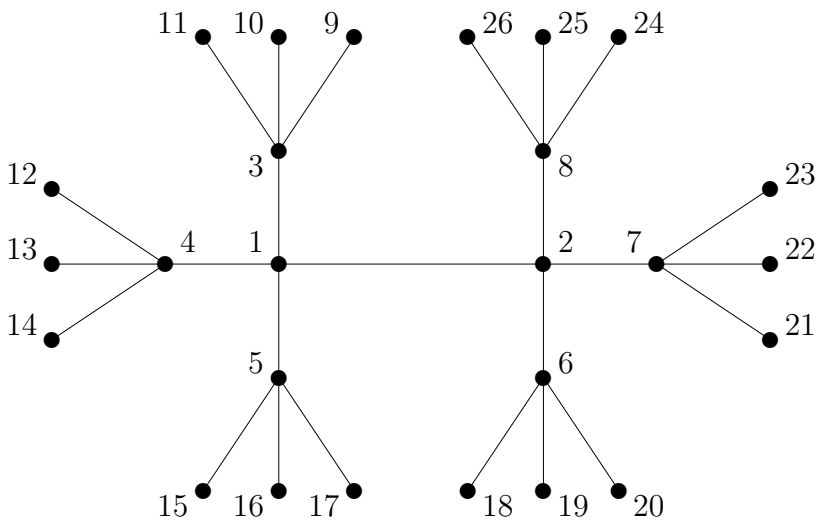


Abbildung 2.4: Der Baum $T_{4,6}$.

2.2.5 Definition:
Sei $k \geq 2$, $t \geq 3$. Für ungerades $t = 2d+1$ bezeichne $T_{k,t} \in \mathcal{G}_{f_0(k,t)}$ den Baum mit Wurzel 1 und folgenden Eigenschaften:

a) Jedes Blatt v hat Abstand $dist(v, 1) = d$ zu Knoten 1.

b) Jeder innere Knoten w hat $grad(w) = k$.

c) $T_{k,t} \in \mathcal{G}_{f_0(k,t)}$ ist kanonisch im Sinne von Definition 1.4.6.

Für gerades $t = 2d + 2$ bezeichne $T_{k,t} \in \mathcal{G}_{f_0(k,t)}$ den Baum mit Wurzel 1 und folgenden Eigenschaften:

a) Knoten 1 und 2 sind benachbart und für jedes Blatt v gilt $dist(v, \{1, 2\}) = d$.

b) Jeder innere Knoten w hat $grad(w) = k$.

c) $T_{k,t} \in \mathcal{G}_{f_0(k,t)}$ ist kanonisch im Sinne von Definition 1.4.6.

Allgemeiner wollen wir mit $\bar{T}_{n,k,t}$ den numerierten Graphen mit $n \geq f_0(k, t)$ Knoten bezeichnen, den man aus $T_{k,t}$ durch Hinzunahme von $n - f_0(k, t)$ Knoten vom Grad 0 erhält.

2.2.6 Bemerkung:
Sei $n = f_0(k, t)$. Zu $T_{k,t}$ betrachten wir die disjunkten Mengen

$$niv(i) := \begin{cases} niv_{T_{k,t},1}(i) & = \{v \in \underline{n} \mid dist_{T_{k,t}}(v, 1) = i\}, \quad \text{falls } t = 2d + 1 \text{ ungerade,} \\ niv_{T_{k,t},\{1,2\}}(i) & = \{v \in \underline{n} \mid dist_{T_{k,t}}(v, \{1, 2\}) = i\}, \quad \text{falls } t = 2d + 2 \text{ gerade,} \end{cases}$$

wobei $0 \leq i \leq d$.

Die folgenden Aussagen über $T_{k,t}$ sind leicht einzusehen:

i) $v \in niv(i),\ w \in niv(j),\ 0 \leq i < j \leq d \Longrightarrow v < w$;

ii) $v, v' \in niv(i),\ w, w' \in niv(i+1),\ 0 \leq i < d,\ w$ ein Sohn von v, w' ein Sohn von v', $v < v' \Longrightarrow w < w'$,

iii) $v, v' \in niv(i),\ w, w' \in niv(j),\ 0 \leq i < j \leq d,\ w$ Nachfolger von v, w' Nachfolger von v', $v < v' \Longrightarrow w < w'$,

iv) Die Knoten mit Grad k sind $1, ..., f_0(k, t-2)$, wobei $f_0(k,2) := 2$ und $f_0(k,1) := 1$.

v) Die Blätter sind $f_0(k, t-2)+1, ..., f_0(k,t)$.

vi) Für ungerades $t > 3$ sind $f_0(k,t-2)+1, ..., f_0(k,t-1)$ die Blätter, die Nachfolger von Knoten 2 sind. $f_0(k,t-1)+1$ ist das kleinste Blatt, das Nachfolger von Knoten 3 ist.

vii) Für gerades $t > 4$ sind $f_0(k,t-1)+1, ..., f_0(k,t)$ die Blätter, die Nachfolger von Knoten 2 sind. $f_0(k,t-2)+1$ ist das kleinste Blatt, das Nachfolger von Knoten 3 ist.

2.2.7 Lemma:
Sei $\Gamma \in rep(S_n \backslash\backslash \mathcal{R}_{n,k,t})$. Dann ist $\bar{T}_{n,k,t} \subset \Gamma$. Desweiteren gilt: Für jedes $w \in \underline{n}$ gibt es ein $\pi \in S_n$, so daß $w^\pi = 1$ und $\bar{T}_{n,k,t} \subset \Gamma^\pi$.

Beweis:
Sei d wie in Bemerkung 2.2.6. Wie im Beweis von 2.2.4 zeigt man, daß Γ Baumstruktur bis $niv(d)$ besitzt. $\Gamma|_{f_0(k,t-2)} := \{(x,y) \in \Gamma \mid x \leq f_0(k,t-2)\} \in \mathcal{G}_{f_0(n,k)}$ ist somit ein Baum, der Bedingungen a) und b) aus Definition 2.2.5 entspricht. Da Γ kanonisch und $\Gamma|_{f_0(k,t-2)} \subset \Gamma$, $\Gamma|_{f_0(k,t-2)} < \Gamma$ folgt $\Gamma|_{f_0(k,t-2)}$ kanonisch (man zeigt dies wie Satz 3.1.2). Damit ist auch c) erfüllt, d.h. $\bar{T}_{n,k,t} = \Gamma|_{f_0(k,t-2)}$ und der erste Teil der Aussage gezeigt. Sei $w \in \underline{n}$ gegeben, $t = 2d+1$ ungerade. Wir betrachten die disjunkten Mengen $niv_{\Gamma,w}(i) = \{v \in \underline{n} \mid dist_\Gamma(v,w) = i\}$, $0 \leq i \leq d$. Wie im Beweis von 2.2.4 zeigen wir, daß Baumstruktur bis $niv_{\Gamma,w}(d)$ vorliegt. Die Nachbarn von w seien $w_1, ..., w_k$, die Knoten in $niv_{\Gamma,w}(2)$ seien $w_{1,1}, ..., w_{1,k-1}, w_{2,1}, ..., w_{k,k-1}$. Allgemein bezeichnen wir zu Knoten $w_{j_1,...,j_i} \in niv_{\Gamma,w}(i)$ die $k-1$ Nachbarn in $niv_{\Gamma,w}(i+1)$ mit $w_{j_1,...,j_i,1}, ..., w_{j_1,...,j_i,k-1}$. Wir definieren $\pi \in S_n$ so, daß $w^\pi = 1$, ${w_1}^\pi = 2, ..., {w_k}^\pi = k+1 (= f_0(k,3))$ und für $w_{j_1,...,j_i} \in niv_{\Gamma,w}(i)$ gilt: ${w_{j_1,...,j_i}}^\pi = f_0(k,2i-1) + 1 + \sum_{l=1}^{i} (k-1)^{i-l}(j_l - 1)$. Dann ist $\bar{T}_{n,k,t} = \Gamma^\pi|_{f_0(k,t-2)}$ und es folgt die Behauptung.

$f_0(k,t)$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$t=3$	4	5	6	7	8
$t=4$	6	8	10	12	14
$t=5$	10	17	26	37	50
$t=6$	14	26	42	62	86
$t=7$	22	53	106	187	302
$t=8$	30	80	170	312	518
$t=9$	46	161	426	938	1824
$t=10$	62	242	682	1563	3120

Tabelle 2.1: Funktionswerte für f_0.

$f(k,t)$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$t=5$	10 (1)	19 (1)	30 (4)	40 (1)	50 (1)
$t=6$	14 (1)	26 (1)	42 (1)	62 (1)	90 (1)
$t=7$	24 (1)	$f(k,t)$ oder Anzahl von (k,t)-Cages nicht bekannt			
$t=8$	30 (1)				
$t=9$	58 (18)				
$t=10$	70 (3)				

Tabelle 2.2: Funktionswerte für f und Anzahl von (k,t)-Cages.

Der Fall, daß $t = 2d+2$ gerade ist, läßt sich analog behandeln: Wir wählen einen Nachbarn z von w und betrachten $niv_{\Gamma,\{w,z\}}(i) = \{v \in \underline{n} \mid dist_\Gamma(v,\{w,z\}) = i\}$. Γ hat Baumstruktur bis $niv(d)$. In $niv_{\Gamma,\{w,z\}}(1)$ seien $w_1,...,w_{k-1}$ die Nachbarn von w und $w_k,...,w_{2k-2}$ die Nachbarn von z. Allgemein bezeichnen wir zu Knoten $w_{j_1,...,j_i} \in niv_{\Gamma,\{w,z\}}(i)$ die $k-1$ Nachbarn in $niv_{\Gamma,\{w,z\}}(i+1)$ mit $w_{j_1,...,j_i,1},...,w_{j_1,...,j_i,k-1}$. Wir definieren $\pi \in S_n$ so, daß $w^\pi = 1$, $z^\pi = 2$, ${w_1}^\pi = 3,...,{w_{2k-2}}^\pi = 2k(= f_0(k,4))$ und für $w_{j_1,...,j_i} \in niv_{\Gamma,w}(i)$ gilt: ${w_{j_1,...,j_i}}^\pi = f_0(k,2i)+1+\sum_{l=1}^{i}(k-1)^{i-l}(j_l-1)$. Dann ist $\bar{T}_{n,k,t} = \Gamma^\pi|_{f_0(k,t-2)}$ und es folgt die Behauptung. □

2.2.8 Bemerkung:
Wie man Tabelle 2.1 entnehmen kann, wachsen die Funktionswerte $f_0(k,t)$ schon für kleines k und t sehr stark (sogar exponentiell in Abhängigkeit von t). Deshalb sind Funktionswerte für f erst in wenigen Fällen bekannt. Tabelle 2.2 enthält einige dieser Funktionswerte und in Klammern die Anzahlen $|S_{f(k,t)}\backslash\backslash\mathcal{R}_{f(k,t),k,t}|$ von (k,t)-Cages. Die Zahlen beruhen auf den Angaben in [Won] und [BriMcK] sowie dem im Rahmen dieser Diplomarbeit angefertigten Programm. In einigen weiteren Fällen kennt man bislang nur $f(k,t)$, nicht aber die Anzahl von (k,t)-Cages. So berechneten kürzlich Brendan McKay und Wendy Myrvold, daß $f(3,11) = 112$.

Kapitel 3

Ordnungstreue Erzeugung

Wir wollen uns nun konkret den in 1.3.8 formulierten Konstruktionsaufgaben zuwenden. Dabei bedienen wir uns des Prinzips der ordnungstreuen Erzeugung, welches erstmals von Read in [Re] beschrieben wurde. Zugrunde liegt dabei die in Definition 1.4.1 erklärte totale Ordnung auf der Menge der numerierten Graphen. Es wird ein Algorithmus erarbeitet, der durch sukzessives Einsetzen einzelner Kanten alle numerierten k-regulären Graphen in lexikographisch aufsteigender Reihenfolge konstruiert, die für das kanonische Repräsentantensystem $rep_<(S_n\backslash\backslash\mathcal{R}_{n,k})$ in Frage kommen. Dabei können auch die Eigenschaften „zusammenhängend“ und „Taillenweite $\geq t$“ berücksichtigt werden. Ein Kanonizitätstest muß entscheiden, in welchen Fällen es sich um kanonische Bahnrepräsentanten handelt. Da ein solcher Test mit großem Aufwand verbunden ist (siehe Kapitel 4), spielt es eine wichtige Rolle, die Menge der Testkandidaten zu begrenzen. Dazu werden notwendige Kriterien für die Kanonizität eines k-regulären numerierten Graphen vorgestellt, mit deren Hilfe schon beim Einsetzen der Kanten eine Vielzahl nichtkanonischer Kandidaten vermieden werden kann.

3.1 Die Methode von Read

Zunächst wollen wir Reads Methode allgemein formulieren: Sei $(Z, \leq)$ eine geordnete Menge und G eine Gruppe, die auf Z operiert. Dann ist

$$rep_<(G\backslash\backslash Z) = \{z \in Z \mid \forall g \in G : z \leq z^g\}$$

eine kanonische Transversale der Bahnen von G auf Z. Die ordnungstreue Erzeugung nach Read beschreibt der folgende

3.1.1 Satz:
Sei $Z = \bigcup_{i=1}^{n} Z_i$, ${Z_i}^G \subseteq Z_i$, $Z_i \cap Z_j = \emptyset$ $(i \neq j)$, und P ein Algorithmus, der $\forall z \in rep_<(G\backslash\backslash Z_i)$ eine Menge $P(z) \subseteq Z$ erzeugt, so daß

- $\forall i : rep_<(G\backslash\backslash Z_{i+1}) \subseteq \bigcup_{z \in rep_<(G\backslash\backslash Z_i)} P(z)$,
- $\forall i : \forall z \in rep_<(G\backslash\backslash Z_{i+1}) : \exists!\ z' \in rep_<(G\backslash\backslash Z_i) : z \in P(z')$.

Dann erhält man die gesuchte Transversale durch:

i) Erzeuge $rep_<(G\backslash\backslash Z_1)$ und setze $rep_<(G\backslash\backslash Z) := rep_<(G\backslash\backslash Z_1)$.

ii) Für $i = 1, ..., n$ durchlaufe $rep_<(G\backslash\backslash Z_i)$ mit z, und erzeuge dabei $P(z)$. Durchlaufe weiter $P(z)$ mit w und prüfe, ob w minimal in seiner Bahn ist. Falls ja, so setze $rep_<(G\backslash\backslash Z) := rep_<(G\backslash\backslash Z) \cup w$.

Wir können diesen Satz auf die Menge $\mathcal{G}_n$ anwenden. Dazu benutzen wir die lexikographische Ordnung auf $\mathcal{G}_n$ aus Definition 1.4.1. Der folgende Satz ist auch in [HKLMW] zu finden.

3.1.2 Satz:
Falls $\Gamma \in rep_<(S_n\backslash\backslash\mathcal{G}_n)$ und $\Gamma_1 \subset \Gamma$ mit $\Gamma_1 < \Gamma$, so folgt $\Gamma_1 \in rep_<(S_n\backslash\backslash\mathcal{G}_n)$.

Beweis:
Sei $\Gamma = \Gamma_1 \cup \Gamma_2$ und $\Gamma_1 \notin rep_<(S_n\backslash\backslash\mathcal{G}_n)$ mit $\Gamma_1 < \Gamma$. Dann ist $\Gamma_1^\pi < \Gamma_1$ für ein $\pi \in S_n$. Falls $\Gamma_1^\pi = \{e_1, ..., e_t\}$ mit $e_1 < e_2 < ... < e_t$, dann existiert ein i mit $\Gamma_1 = \{e_1, ..., e_{i-1}, e_i', ..., e_t'\}$, $e_{i-1} < e_i' < ... < e_t'$ und $e_i < e_i'$. Es ist $\Gamma^\pi = \Gamma_1^\pi \cup \Gamma_2^\pi \supseteq \{e_1, ..., e_i\}$, so daß $\Gamma^\pi < \Gamma_1 < \Gamma$, ein Widerspruch zu $\Gamma \in rep_<(S_n\backslash\backslash\mathcal{G}_n)$. □

3.1.3 Bemerkung:
Um Satz 3.1.1 anwenden zu können, müssen wir für einen Repräsentanten die Menge $P(\Gamma)$ definieren als

$$P(\Gamma) := \{\Gamma \cup \{e\} \mid e > \max\{e' \in \Gamma\}\}.$$

Obwohl es keine Komplexitätsanalyse für 3.1.1 gibt, ist dieser Ansatz doch recht effizient. Wir werden im folgenden sehen, wie man diese Methode zur Erzeugung regulärer Graphen verwenden kann.

3.2 Anwendung auf reguläre Graphen

3.2.1 Definition:
Für $\Gamma \in \mathcal{G}_n$ sei $P^{(1)}(\Gamma) := P(\Gamma)$ und

$$P^{(i+1)}(\Gamma) := \bigcup_{\Gamma' \in P^{(i)}(\Gamma)} P(\Gamma'),\ i \geq 1.$$

Ist $\max\{e' \in \Gamma'\} = (n-1, n)$, dann sei $P(\Gamma') := \emptyset$. Mit $Q(\Gamma)$ bezeichnen wir die Menge

$$Q(\Gamma) := \bigcup_{i \geq 1} P^{(i)}(\Gamma).$$

Offenbar gilt für diese Menge: $Q(\Gamma) = \{\Gamma' \in \mathcal{G}_n \mid \Gamma \subset \Gamma' \wedge \Gamma < \Gamma'\}$. Die folgende Aussage ist beinahe trivial:

3.2.2 Lemma:
Es gilt $rep_<(S_n \backslash\backslash \mathcal{R}_{n,k}) \subseteq Q(\{(1,2)\})$.

Beweis:
Sei $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$, $\Gamma = \{e_1, ..., e_m\}$, $e_1 < ... < e_m$, $m = \frac{nk}{2}$. Dann ist $e_1 = (1,2)$. Seien für $i = 2, ..., m$ $\Gamma^{(i)} := \{e_1, ..., e_i\} \subseteq \Gamma$.
Wir zeigen per Induktion $\Gamma^{(i)} \in P^{(i-1)}(\{(1,2)\})$: $\Gamma^{(2)} = \{e_1, e_2\} \in P(\{e_1\}) = P^{(1)}\{e_1\}$, $\Gamma^{(i+1)} = (\Gamma^{(i)} \cup \{e_{i+1}\}) \in P(\Gamma^{(i)}) \subseteq P^{(i)}(\Gamma)$, da nach Voraussetzung $\Gamma^{(i)} \in P^{(i-1)}$. □

3.2.3 Algorithmus:
Basierend auf Satz 3.1.2 können wir folgende Funktion beschreiben, die nach Aufruf von ordrek({(1,2)}) die Menge $rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$ ausgibt.

ordrek(Γ)

1. `Prüfe, ob` Γ `durch Einsetzen weiterer Kanten ein regulärer Graph mit n Knoten vom Grad k werden kann, wenn nein: return;`
2. `Prüfe, ob` $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{G}_n)$ `, wenn nein: return;`
3. `Falls` $\Gamma \in \mathcal{R}_{n,k}$`:` Γ `ausgeben; return;`
4. `Durchlaufe` $P(\Gamma)$ `aufsteigend mit` Γ' `und führe ordrek(`Γ'`) aus;`

Der Algorithmus liefert tatsächlich alle Elemente von $rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$: Sei Γ ein gesuchter Repräsentant, $\Gamma^{(i)} \subseteq \Gamma$ wie im Beweis von Lemma 3.2.2. Dann besteht $\Gamma^{(i)}$ jeweils den Test unter 1. Nach Satz 3.1.2 gilt für jedes i: $\Gamma^{(i)} \in rep_<(S_n \backslash\backslash \mathcal{G}_n)$.
Es ist leicht einzusehen, daß der Algorithmus die Graphen in lexikographisch aufsteigender Reihenfolge durchläuft. So kann es auch nicht vorkommen, daß ein Repräsentant mehrfach ausgegeben wird.
Um unter 2. zu entscheiden, ob $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{G}_n)$, muß ein Kanonizitätstest (siehe Kapitel 4) durchgeführt werden. Unter 1. muß offenbar gelten $\Gamma \in \mathcal{G}_{n,k}$. Darüberhinaus werden hier noch weitere Kriterien überprüft:

3.2.4 Lemma:
Sei $(x,y) = \max\{e \in \Gamma\}$, $y > n - k$ und $grad(x) < k$. Ist $n - y < k - grad(x)$, dann folgt $Q(\Gamma) \cap \mathcal{R}_{n,k} = \emptyset$.

Beweis:
Es werden noch $k - grad(x)$ Nachbarn für x gebraucht. Dafür kommen nur die Knoten $y + 1, ..., n$ in Frage. Es gibt also noch $n - y$ mögliche Nachbarn. Deshalb muß gelten $n - y \geq k - grad(x)$. □

3.2.5 Lemma:
Sei $(x, y) = \max\{e \in \Gamma\}$, $x \geq n - k$ und $grad(x) = k$. Wenn für ein i mit $y < i \leq n$ gilt $n - x - 1 < k - grad(i)$, dann folgt $Q(\Gamma) \cap \mathcal{R}_{n,k} = \emptyset$.

Beweis:
Ist $grad(i) = k$, dann ist die Bedingung in keinem Fall erfüllt (die linke Seite der Ungleichung ist immer ≥ 0). Sei also $grad(i) < k$. Dann braucht Knoten i noch $k - grad(i)$ weitere Nachbarn. Dafür stehen Knoten $x + 1, ..., i - 1, i + 1, ..., n$ zur Verfügung, also $n - x - 1$ mögliche Nachbarn. Deshalb muß gelten $n - x - 1 \geq grad(i) - k$. □

3.2.6 Bemerkung:
Man kann noch stärkere Bedingungen wie in Lemma 3.2.4 und 3.2.5 formulieren. Erfahrungsgemäß wird dann jedoch der Aufwand, der bei der Überprüfung dieser Bedingungen anfällt, größer als die Einsparung, die damit erzielt werden kann. Weiterhin haben solche Kriterien den gemeinsamen Nachteil, daß sie erst beim Einsetzen der letzten Kanten verstärkt wirksam werden und somit bei größerem n eine zunehmend unwichtige Rolle spielen. Trotzdem kann man auf dieser Grundlage für kleine n einen recht effizienten Algorithmus entwerfen. Wir wollen den folgenden Pseudocode der Funktion ORDREK(x, y, v) genauer betrachten:

ORDREK(x, y, v)**:**

```
(1 )   if y > n − k ∧ deg[x] < k ∧ n − y < k − deg[x] then return
(2 )   if x ≥ n − k ∧ deg[x] = k
(3 )         for i = y + 1 to n do
(4 )               if n − x − 1 < k − deg[i] then return
(5 )         end
(6 )   end
(7 )   while x < n ∧ deg[x] = k do x := x + 1
(8 )   if v ≤ y then v := y + 1
(9 )   if x = v then return
(10 )  if KATEST() = 0 then return
(11 )  if x = n ∧ deg[x] = k then OUTPUT()
(12 )  y := x
(13 )  while y < n do
(14 )        y := y + 1
(15 )        if deg[y] < k
(16 )              INSERT(x, y)
(17 )              ORDREK(x, y, v)
(18 )              DELETE(x, y)
(19 )        end
(20 )  end
(21 )  return
```

Folgende Datenstrukturen werden benötigt:

- Eine Datenstruktur zur Speicherung des Graphen. In Frage kommt dafür z.B. ein zweidimensionales Feld der Größe $n \times n$ für die Adjazenzmatrix, oder ein Feld der Länge nk, wo für jeden Knoten i ab Stelle $(i-1)k+1$ die Nachbarn von i eingetragen werden, eine sogenannte Nachbarschaftsliste. In der vorliegenden Implementierung werden sogar mehrere Datenstrukturen gleichzeitig geführt, damit in verschiedenen Situationen jeweils ein optimaler Zugriff gewährleistet ist. Die Funktionen INSERT(x,y) und DELETE(x,y) erledigen dann abgestimmt auf die verwendete Datenstruktur das Einfügen und Löschen von Kante (x,y).

- Ein Vektor deg der Länge n, welcher die Grade der Knoten enthält. Natürlich könnte man diese auch jeweils anhand der Adjazenzmatrix bzw. Nachbarschaftsliste bei jeder Anfrage berechnen. Man spart aber Zeit, wenn man die betreffenden Einträge in deg jeweils nur beim Einfügen oder Löschen einer Kante durch INSERT(x,y) bzw. DELETE(x,y) aktualisieren muß.

- Variablen x, y und v, die beim Funktionsaufruf übergeben werden. Dabei ist (x,y) die zuletzt eingesetzte Kante und v der kleinste Knoten mit Grad 0 vor dem Einfügen von Kante (x,y).

- Globale Variablen n und k sowie die lokale Variable i.

Vor dem ersten Aufruf wird die Datenstruktur für den Graphen so initialisiert, daß sie $\bar{T}_{n,k,3}$ aus Definition 2.2.5 enthält (Wir benutzen dabei Lemma 2.2.7). Entsprechend werden auch die Knotengrade angepaßt: $deg[1]=k$, $deg[2]=...=deg[k+1]=1$, $deg[i]=0$ für $i=k+2,...,n$. Beim Aufruf von ORDREK$(1,k+1,k+1)$ wird das Repräsentantensystem $rep_<(S_n\backslash\backslash\mathcal{R}^*_{n,k})$ der zusammenhängenden k-regulären Graphen mit n Knoten ausgegeben.
In Zeilen 1 und 2-6 wird geprüft, ob Voraussetzungen für Lemma 3.2.4 bzw. 3.2.5 zutreffen. Falls ja, kann in der Rekursion ein Schritt zurückgegangen werden. In Zeile 7 bekommt x den kleinsten Knoten mit Grad $< k$ zugewiesen. Existiert kein solcher Knoten, dann ist nun $x = n$. Zeile 8 bewirkt, daß v den kleinsten Knoten vom Grad 0 enthält (falls es keinen Knoten mit dieser Eigenschaft gibt, ist danach $v = n+1$). Stimmt dieser Knoten mit x überein, dann kann keine weitere Kante eingesetzt werden, so daß der Graph zusammenhängend bleibt im Sinne von Definition 1.3.3 (dabei wird Lemma 3.2.7 verwendet). Falls nur zusammenhängende k-reguläre Graphen konstruiert werden sollen, muß also kein abschließender Zusammenhangstest durchgeführt werden. Durch Weglassen von Zeile 9 werden auch reguläre Graphen mit mehreren Zusammenhangskomponenten erzeugt. In Zeile 10 wird auf Kanonizität getestet. Die Funktion KATEST() ist in 4.2.5 erklärt. Zeile 11 stellt fest, ob ein k-regulärer Graph mit n Knoten gefunden ist. Trifft dies zu, so wird er ausgegeben.

Zeilen 12-20 steuern das Einsetzen einer neuen Kante. Dabei werden alle in Frage kommenden Kanten (unter Verwendung von Lemma 3.2.8) in aufsteigender Reihenfolge durchlaufen. Zeile 15 stellt sicher, daß y noch nicht Grad k hat. Hier könnte man auch noch Lemma 3.2.9 anwenden. Zeile 16 nimmt das Einsetzen der neuen Kante (x, y) vor. In Zeile 17 erfolgt der rekursive Aufruf von ORDREK(x, y, v), und wenn dieser abgearbeitet ist, wird (x, y) wieder gelöscht.
Wir wollen nicht versäumen die verwendeten Lemmata nachzureichen:

3.2.7 Lemma:
Sei $x = \min\{i \in \underline{n} \mid grad_\Gamma(i) < k\}$, $grad(x) = 0$
$\Longrightarrow P(\Gamma) \cap \mathcal{G}^*_{n,k} = \emptyset$, $Q(\Gamma) \cap \mathcal{G}^*_{n,k} = \emptyset$, $Q(\Gamma) \cap \mathcal{R}^*_{n,k} = \emptyset$.

Beweis:
klar. □

3.2.8 Lemma:
Sei $x = \min\{i \in \underline{n} \mid grad_\Gamma(i) < k\}$. Dann gilt für $x' = x + 1, ..., n - 1$, $y = x' + 1, ..., n$:
$Q(\Gamma \cup \{(x', y)\}) \cap \mathcal{R}_{n,k} = \emptyset$.

Beweis:
$\forall \Gamma' \in Q(\Gamma \cup \{(x', y)\})$: $grad_\Gamma(x) < k$. □

3.2.9 Lemma:
Besitzt Γ Knoten vom Grad 0 , dann sei $v = \min\{i \in \underline{n} \mid grad_\Gamma(i) = 0\}$. Weiter sei $x = \min\{i \in \underline{n} \mid grad_\Gamma(i) < k\}$ und $x \neq v$. Dann gilt für $v < y \leq n$:
$(\Gamma \cup \{(x, y)\}) \notin rep_<(S_n \backslash\backslash \mathcal{G}_n)$

Beweis:
$(\Gamma \cup \{(x, y)\})^{(v,y)} = (\Gamma \cup \{(x, v)\}) < (\Gamma \cup \{(x, y)\})$. Es handelt sich dabei um einen Spezialfall vom Satz 3.3.2. □

3.2.10 Bemerkung:
Algorithmus 3.2.3 führt nahezu nach jeder Hinzunahme einer neuen Kante einen Kanonizitätstest durch. Ein solcher Test ist sehr aufwendig, da grundsätzlich alle Elemente π von S_n durchlaufen werden müssen, um zu prüfen, ob $\Gamma \leq \Gamma^\pi$ gilt (siehe Kapitel 4). Dies bedeutet in zweifacher Hinsicht unnötigen Zeitaufwand: Zum einen müssen Kandidaten, die letztlich bestehen, insgesamt mehrmals getestet werden. Zum anderen kann es vorkommen, daß Graphen zwar mehrfach den Kanonizitätstest bestehen, später aber festgestellt wird, daß sie nicht zu regulären Graphen komplettiert werden können. In beiden Fällen war der Aufwand zum Testen der Kanonizität dann umsonst. Es hat sich als wesentlich effektiver erwiesen, lediglich bei Graphen aus $\mathcal{R}_{n,k}$ den Kanonizitätstest durchzuführen. Ansonsten werden nur notwendige Kanonizitätsbedingungen überprüft. Dies ist wichtig, damit nicht zu viele Kandidaten für den abschließenden Kanonizitätstest produziert werden. Solche Bedingungen werden in den nächsten Abschnitten erarbeitet.

3.3 Das Zeilenkriterium

3.3.1 Definition:
Sei $\Gamma \in \mathcal{G}_n$, $\Gamma_i := \{(i,v) \in \Gamma\}$, $1 \leq i < n$. Es gilt also $\Gamma = \bigcup_{i=1}^{n-1} \Gamma_i$, $\Gamma_i \cap \Gamma_j = \emptyset$ $(i \neq j)$ und mit $1 \leq i < j < n$: $e \in \Gamma_i$, $e' \in \Gamma_j \Longrightarrow e < e'$.
Wir definieren $C_1 := C_{S_n}(\{1\})$ und für $1 \leq i < n$:

$$\begin{aligned} N_i &:= N_{C_i}(\Gamma_i) = \{\pi \in C_i \mid {\Gamma_i}^{\pi} = \Gamma_i\}, \\ C_{i+1} &:= C_{N_i}(\{1,...,i+1\}) = \{\pi \in N_i \mid (i+1)^{\pi} = i+1\}. \end{aligned}$$

3.3.2 Satz:
Sei $\Gamma \in rep_{<}(S_n \backslash\backslash \mathcal{G}_n)$. Dann gilt

$$\forall i < n: \ \forall \pi \in C_i: \quad \Gamma_i \leq {\Gamma_i}^{\pi} \qquad (*).$$

Beweis:
Indirekt: Sei i_0 das kleinste i für das die Behauptung nicht erfüllt ist, d.h. $\exists \tau \in C_{i_0}$: ${\Gamma_{i_0}}^{\tau} < \Gamma_{i_0}$. Wegen $C_{i_0} \leq N_j \ \forall j < i_0$ ist ${\Gamma_j}^{\tau} = \Gamma_j \ \forall j < i_0$. Es ist somit $\Gamma^{\tau} < \Gamma$, ein Widerspruch zur Kanonizität von Γ. □

3.3.3 Definition:
Besitzt $\Gamma \in \mathcal{G}_n$ die Eigenschaft $(*)$ aus Satz 3.3.2, dann nennen wir Γ *semikanonisch*.

3.3.4 Bemerkung:
Auf gleiche Weise wird der Begriff „semikanonisch“ auch in [Gru2] erklärt. Unser Ziel ist es, schon beim Einsetzen der Kanten darauf zu achten, daß nur semikanonische Kandidaten konstruiert werden. Dies erfordert ein zeilenweises Vorgehen: Die Kanten aus Γ_i entsprechen den Einsen rechts der Diagonalen in der i-ten Zeile a_i der Adjazenzmatrix A von Γ. Die Kanten in Γ_i müssen also so gesetzt werden, daß $\Gamma_i \leq {\Gamma_i}^{\pi} \forall \pi \in C_i$. Dies ist gleichbedeutend mit $a_i \succeq a_i^{\pi} \forall \pi \in C_i$, wobei a_i^{π} die i-te Zeile von A^{π}. Wir werden erarbeiten, was dies für die Zeile a_i bedeutet.
Ein großer Vorteil dieser Methode liegt darin, daß man den benötigten Normalisator N_i leicht anhand von a_i und C_i bestimmen kann. Sowohl bei den C_i, als auch bei den N_i handelt es sich nämlich um Younguntergruppen von S_n. Besonders leicht lassen sich die C_i ermitteln:

3.3.5 Bemerkung:
Den Zentralisator $C_{S_n}(\{1,...,i\})$ der Ziffern $1,...,n$ kann man auch als Younguntergruppe von S_n auffassen: Sei $\mu = (\mu_1,...,\mu_{i+1}) \models n$ mit $\mu_1 = ... = \mu_i = 1$ und $\mu_{i+1} = n - i$. Dann ist $S_\mu = C_{S_n}(\{1,...,i\})$.
Gilt für $\lambda = (\lambda_1,...,\lambda_m) \models n$: $S_\lambda \leq C_{S_n}(\{1,...,i\})$, dann hat λ die Form

$$\lambda = (\underbrace{1,...,1}_{i}, \lambda_{i+1}, ...\lambda_m).$$

Dann kann man $C_{S_\lambda}(\{1,...,i+1\})$ ebenfalls als Younguntergruppe schreiben: Für

$$\zeta := \begin{cases} (\underbrace{1,...,1}_{i+1}, \lambda_{i+1} - 1, \lambda_{i+2}, ..., \lambda_m) & \text{falls } \lambda_{i+1} > 1, \\ \lambda & \text{sonst,} \end{cases}$$

gilt $S_\zeta = C_{S_\lambda}(\{1,...,i+1\})$.

Auch bei den N_i handelt es sich um Younguntergruppen. Um diese zu bestimmen, bedarf es einiger kombinatorischer Vorüberlegungen:

3.3.6 Definition:
Es sei $\mu = (\mu_1,...,\mu_s) \models n$ eine Partition von n. Wir betrachten die Operation von S_μ auf der Menge der 0/1-Vektoren der Länge n:

$$\{0,1\}^{\underline{n}} \times S_\mu \longrightarrow \{0,1\}^{\underline{n}}, \qquad (x_1,...,x_n)^\pi := (x_{\pi(1)},...,x_{\pi(n)}).$$

Sei $x \in \{0,1\}^{\underline{n}}$. Wir nennen x *kanonisch* bzgl. μ, wenn

$$\forall \pi \in S_\mu : \quad x \succeq x^\pi.$$

Eine Permutation $\sigma \in S_\mu$ heißt *kanonisierende Permutation* von x bzgl. μ, wenn x^σ kanonisch ist bzgl. μ.

3.3.7 Lemma:
Mit den Bezeichnungen aus Definition 3.3.6 gilt:

$$x \text{ kanonisch bzgl. } \mu \quad \Longleftrightarrow \quad \forall i = 1,...,s \; \forall j, j' \in \underline{n}_i^\mu \text{ mit } j < j' : \; x_j \geq x_{j'}.$$

Beweis:
$\pi \in S_\mu$ permutiert nur Einträge von x innerhalb der durch μ gegebenen Blöcke. Damit sind beiden Implikationen klar. □

3.3.8 Bemerkung:
Ist $x \in \{0,1\}^{\underline{n}}$ kanonisch bzgl. μ, dann stehen also innerhalb der durch μ gegebenen Blöcke die Einsen von x links und die Nullen rechts. Andererseits kann man eine kanonisierende Permutation von x ermitteln, indem man innerhalb der μ-Blöcke die Einsen nach links und die Nullen nach rechts sortiert. Wir können auf diese Weise leicht eine fusionierende Abbildung definieren.

Unser Interesse soll nun dem Stabilisator zu kanonischem $x \in \{0,1\}^{\underline{n}}$ gelten, d.h. der Menge $\{\pi \in S_\mu \mid x^\pi = x\}$. Dies sind gerade diejenigen Permutationen, welche die Einträge innerhalb der μ-Blöcke so vertauschen, daß die Einsen jeweils links und die Nullen rechts bleiben. Dies beschreibt folgende

3.3.9 Definition:
Sei $\mu = (\mu_1, ..., \mu_s) \models n$ eine Partition und $x \in \{0,1\}^{\underline{n}}$ kanonisch bzgl. μ. Weiter sei für $i = 1, ..., s$ $\mu_i^1 := |\{j \in \underline{n}_i^\mu : x_j = 1\}|$ die Anzahl der Einsen von x im i-ten Block bzgl. μ und $\mu_i^0 := |\{j \in \underline{n}_i^\mu : x_j = 0\}|$ die Anzahl der Nullen von x im i-ten Block bzgl. μ. Aus der Partition

$$(\mu_1^1, \mu_1^0, \mu_2^1, \mu_2^0, ..., \mu_s^1, \mu_s^0) \models n$$

entfernen wir alle Komponenten μ_i^1 und μ_i^0, die den Wert 0 haben. Die resultierende Partition $\eta \models n$ nennen wir *kanonische Verfeinerung* von μ bzgl. x.

3.3.10 Lemma:
Mit den Bezeichnungen aus Definition 3.3.9 gilt:

$$S_\eta = \{\pi \in S_\mu \mid x^\pi = x\}.$$

Beweis:
Klar nach Konstruktion von η. □

Bevor wir diese Hilfsmittel auf die Zeilen der Adjazenzmatrix A anwenden, sei noch ein fiktives Beispiel angegeben:

3.3.11 Beispiel:
Es sei $n = 10$, $\mu = (3,1,2,4)$ und $x = (0,1,0,1,0,0,1,0,1,0)$. Zur besseren Übersicht markieren wir die durch μ gegebenen Blöcke von x mit Längsstrichen:

$$x = (0,1,0,\mid 1,\mid 0,0,\mid 1,0,1,0).$$

Eine kanonisierende Permutation ist $\pi = (1,2)(8,9) \in S_\mu$, wobei

$$x^\pi = (1,0,0,\mid 1,\mid 0,0,\mid 1,1,0,0).$$

Aus den Anzahlen der Einsen und Nullen in den μ-Blöcken von x erhalten wir

$$(\mu_1^1, \mu_1^0, \mu_2^1, \mu_2^0, \mu_3^1, \mu_3^0, \mu_4^1, \mu_4^0) = (1,2,1,0,0,2,2,2),$$

und durch Weglassen der Komponenten mit Wert 0 ergibt sich die kanonische Verfeinerung η von μ bzgl. x:

$$\eta = (1,2,1,2,2,2).$$

3.3.12 Bemerkung und Definition:
Wir können die Zeile $a_i = (a_1, ..., a_n)$ von A auch als 0/1-Vektor auffassen. Für eine Permutation $\pi \in S_n$ ist ${a_i}^\pi$ die i-te Zeile von A^π:

$$ {a_i}^\pi := (a_{\pi(i),\pi(1)}, ..., a_{\pi(i),\pi(n)}). $$

Sei $\zeta = (\zeta_1, ..., \zeta_i, \zeta_{i+1}, ...) \models n$ eine Partition mit $\zeta_j = 1$, $j = 1, ..., i$. Dann gilt $S_\zeta \leq C_{S_n}(\{1, ..., i\})$. Für $\tau \in S_\zeta$ ist

$$ {a_i}^\tau = (a_{\tau(i),\tau(1)}, ..., a_{\tau(i),\tau(n)}) = (a_{i,1}, ..., a_{i,i}, a_{i,\tau(i+1)}, ..., a_{i,\tau(n)}), $$

d.h. τ permutiert nur die Einträge von Zeile a_i ab Stelle $i+1$ innerhalb der ζ-Blöcke. Wir nennen Zeile a_i *kanonisch* bzgl. ζ, wenn

$$ \forall \tau \in S_\zeta : \quad a_i \succeq {a_i}^\tau. $$

Eine Permutation $\sigma \in S_\zeta$ heißt *kanonisierende Permutation* von Zeile a_i bzgl. ζ, wenn ${a_i}^\sigma$ kanonisch ist bzgl. ζ. Dieser Begriff wird später im Kanonizitätstest eine zentrale Rolle spielen. Wir kommen nun zur

3.3.13 Anwendung:
Für unseren Algorithmus wenden wir die gewonnenen Erkenntnisse folgendermaßen an: Sei $\Gamma_i = \emptyset$ und $C_i = S_{\zeta^{(i)}}$ bekannt (am Anfang setzen wir nach 3.3.5 $\zeta^{(1)} = (1, n-1)$). Das Einsetzen von Kanten in Γ_i ist so vorzunehmen, daß a_i kanonisch (im Sinne von 3.3.12) bzgl. $\zeta^{(i)}$. Dies erreichen wir, indem wir die Einsen so setzen, daß sie innerhalb der $\zeta^{(i)}$-Blöcke immer links der Nullen stehen. Es gilt dann nach 3.3.7

$$ \forall \pi \in S_{\zeta^{(i)}} : \quad a_i \succeq {a_i}^\pi. $$

Dies ist gleichbedeutend mit

$$ \forall \pi \in C_i : \quad \Gamma_i \leq {\Gamma_i}^\pi. $$

Dann berechnen wir die kanonische Verfeinerung $\eta^{(i)}$ von $\zeta^{(i)}$ bzgl. a_i. Für die induzierte Younguntergruppe gilt nach 3.3.10:

$$ S_{\eta^{(i)}} = \{\pi \in S_{\zeta^{(i)}} \mid {a_i}^\pi = a_i\} = \{\pi \in C_i \mid {\Gamma_i}^\pi = \Gamma_i\} = N_i. $$

Schließlich ermitteln wir $C_{i+1} = C_{S_{\eta^{(i)}}}(\{1, ..., i+1\})$ wie in 3.3.5. Wir erhalten $\zeta^{(i+1)}$ mit

$$ S_{\zeta^{(i+1)}} = C_{i+1} $$

und wir können mit Γ_{i+1} fortfahren.

3.3.14 Beispiel:
Für den Fall $n = 8$ und $k = 3$ gibt es 10 Möglichkeiten die ersten drei Zeilen der Adjazenzmatrix so zu füllen, daß sie dem Zeilenkriterium genügen. Die $\zeta^{(i)}$- Blöcke sind dabei jeweils durch Striche gekennzeichnet:

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

$$
\begin{array}{c|c|c|c|c|c|c|c}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

3.4 Das Taillenkriterium

Wir wollen uns nun einer Bedingung zuwenden, die insbesondere Regularität voraussetzt. Die in diesem Abschnitt vorgestellte Methode wurde von Gunnar Brinkmann entwickelt. Lemma 3.4.3 ist in ähnlicher Form auch in [Bri] zu finden.

3.4.1 Satz:
Sei $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$. Dann liegt Knoten 1 auf einem Taillenkreis von Γ. Knoten 2 und 3 gehören zu demselben Taillenkreis.

Beweis:
Liefert Lemma 3.4.3. □

3.4.2 Bemerkung:
Wenn wir die Bedingung, daß Γ regulär ist fallen lassen, wird Satz 3.4.1 falsch. Abbildung 3.1 zeigt ein Beispiel aus $rep_<(S_n \backslash\backslash \mathcal{G}_n)$, bei dem Knoten 1 nicht auf einem Taillenkreis liegt.

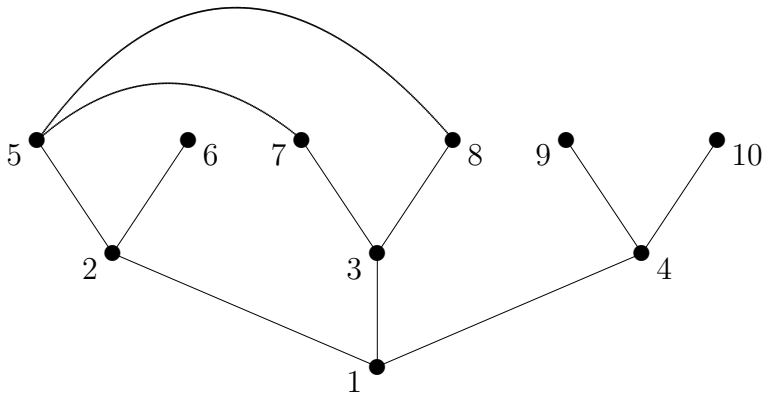


Abbildung 3.1: Ein Beispiel für das Taillenkriterium.

3.4.3 Lemma:
Wenn $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$, dann beschreibt die Funktion girthcheck(Γ) einen Taillenkreis in Γ, auf dem auch Knoten 3 liegt.

girthcheck(Γ)

1. `Starte bei Knoten 1 und gehe zu Knoten 2;`
2. `Angelangt bei Knoten i gehe zum kleinsten Nachbarn, ausgenommen dem Vorgänger auf dem bisherigen Weg;`
3. `Wiederhole 2. bis Knoten 1 erreicht ist;`

Beweis:
Sei $t = girth(\Gamma)$.

i) $t = 2d + 1$ ungerade. Daß Knoten $u = f_0(k, t-2) + 1$ erreicht wird ist klar nach Bemerkung 2.2.6 und Lemma 2.2.7. u ist der kleinste Nachfolger von Knoten 2 in $niv_{\Gamma,1}(d) = \{v \in \underline{n} \mid dist_\Gamma(v, 1) = d\}$, bzw. $u = 2$, falls $t = 3$. Wir betrachten die Nachbarn von u. u hat einen Nachbarn in $niv_{\Gamma,1}(d-1)$, den Vorgänger auf dem bislang beschriebenen Weg. Der nächstgrößere Nachbar ist in $niv_{\Gamma,1}(d)$ zu suchen. Wegen der Taillenweite kann u nicht benachbart sein zu einem Nachfolger von Knoten 2 in $niv_{\Gamma,1}(d)$. Der kleinste Nachfolger $w = f_0(k, t-1) + 1$ von Knoten 3 in $niv_{\Gamma,1}(d)$ kann Nachbar von u sein.
Behauptung: $(u, w) \in \Gamma$. Angenommen $(u, w) \notin \Gamma$. Wir wählen einen Taillenkreis $v_0, v_1, ..., v_d, v_{d+1}, ..., v_{t-1}, v_0$. Lemma 2.2.7 liefert ein $\pi \in S_n$ mit:

$$v_0{}^\pi = 1,\ v_d{}^\pi = f_0(k, t-2) + 1,\ v_{d+1}{}^\pi = f_0(k, t-1) + 1.$$

Dann ist $(f_0(k, t-2) + 1, f_0(k, t-1) + 1) \in \Gamma^\pi < \Gamma$, da Γ^π und Γ in den kleineren Kanten übereinstimmen (dies sind gerade die Kanten von $\bar{T}_{n,k,t}$). Wir haben also einen Widerspruch zur Kanonizität von Γ, und die Behauptung ist gezeigt.
Angelangt bei Knoten w kommt man über 3 zurück zu 1.

ii) $t = 2d + 2$ gerade. Nach Bemerkung 2.2.6 und Lemma 2.2.7 wird Knoten $w = f_0(k, t-1)+1$ erreicht. w ist der kleinste Nachfolger von Knoten 2 in $niv_{\Gamma,\{1,2\}}(d) = \{v \in \underline{n} \mid dist_\Gamma(v, \{1, 2\}) = d\}$. Der kleinste Nachbar von w befindet sich in $niv_{\Gamma,\{1,2\}}(d-1)$. Als nächstgrößerer Nachbar von w kommt der kleinste Knoten aus $niv_{\Gamma,\{1,2\}}(d)$ in Frage. Dieser ist $u = f_0(k, t-2) + 1$. Wie unter i) zeigt man, daß $(u, w) \in \Gamma$. Der Rest ist klar, weil u ein Nachfolger von Knoten 3 ist, bzw. $u = 3$, falls $t = 4$.

□

3.4.4 Folgerung:
Wenn $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$, $\Gamma' \subset \Gamma$ mit $\Gamma' < \Gamma$ und Γ' genau einen Kreis enthält, dann wird dieser von girthcheck(Γ') beschrieben, und er stimmt mit dem Kreis, den girthcheck(Γ) konstruiert, überein.

Beweis:
Sei $\Gamma = \{e_1, ..., e_m\}$, $e_1 < ... < e_m$, $t = girth(\Gamma)$, $\Gamma' = \{e_1, ..., e_{m'}\}$, $m' < m$. Es ist $\bar{T}_{k,t} = \{e_1, ..., e_s\}$, $s = f_0(k, t) - 1$. Nach dem Beweis von Lemma 3.4.3 ist $e_{s+1} = (u, w)$. Da Γ' einen Kreis enthält, ist $m' \geq s + 1$ und es folgt die Behauptung. □

3.4.5 Anwendung:
Die neu gewonnenen Erkenntnisse können wir zur Reduzierung der Kandidatenmenge heranziehen:
Falls Γ noch keinen Kreis besitzt und durch Einsetzen von Kante e ein erster Kreis geschlossen wird, prüft man, ob girthcheck($\Gamma \cup \{e\}$) diesen Kreis beschreibt und ob Knoten 3 darauf liegt. Trifft beides zu, merkt man sich die Länge dieses Kreises und fährt mit dem Einsetzen weiterer Kanten fort. Anderenfalls kann $\Gamma \cup \{e\}$ verworfen werden.
Falls Γ bereits einen Kreis besitzt und $girth(\Gamma) > 3$ gemerkt wurde, muß beim Einsetzen von Kante e überprüft werden, ob $girth(\Gamma \cup \{e\}) < girth(\Gamma)$. Trifft dies zu, so kann $\Gamma \cup \{e\}$ verworfen werden. Abbildung 3.1 zeigt eine Situation, die bei unserem Algorithmus aus Abschnitt 3.2 eintritt: Die neu eingesetzte Kante ist (5,8). Die Taillenweite war zuvor 5 und ist nun 4. In der Rekursion kann ein Schritt zurückgegangen werden.
Die Tatsache, daß ein neuer Kreis geschlossen wurde, läßt sich leicht feststellen. In dem Algorithmus aus Abschnitt 3.2 ist dies genau dann der Fall, wenn bei Aufruf von ORDREK(x, y, v) gilt $y < v$. Verläuft beim ersten Kreis girthcheck(Γ) erfolgreich, dann ist die Länge dieses Kreises gleich der Anzahl der durchlaufenen Knoten (wobei Knoten 1 natürlich nur einmal gezählt wird). In den meisten Fällen existieren bereits Kreise. Dann wird nach „breadth-first“-Methode, ausgehend von den beiden Knoten der neu eingesetzten Kante (x, y), gesucht, ob ein neuer Kreis mit Länge $< girth(\Gamma)$ entstanden ist. Dies ist natürlich nur relevant, wenn $girth(\Gamma) > 3$ war.

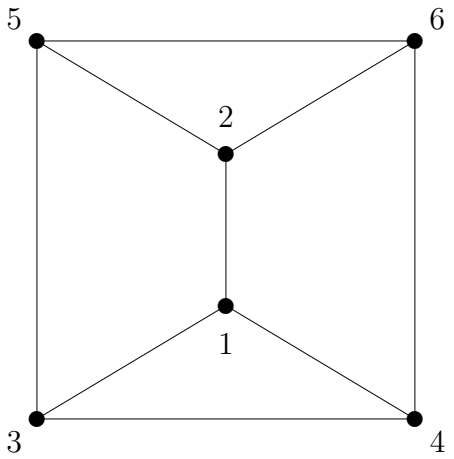


Abbildung 3.2: Ein vermeidbarer Kandidat.

3.4.6 Beispiel:
Abbildung 3.2 zeigt ein Element $\Gamma \in \mathcal{R}_{6,3}$, bei dem das Taillenkriterium angewendet werden kann. Γ ist semikanonisch, würde nach unserem bisherigen Verfahren konstruiert und müßte nun auf Kanonizität getestet werden. Allerdings stellt man mit 3.4.4 schon nach Einsetzen von Kante $(3, 4)$ fest, daß kein Repräsentant entstehen kann.

			ohne Taillenkriterium			mit Taillenkriterium		
n	k	Graphen	Kandidaten	Faktor	Zeit	Kandidaten	Faktor	Zeit
10	3	19	250	13,16	0,0 s	61	3,21	0,0 s
12	3	85	2999	35,28	0,6 s	513	6,04	0,2 s
14	3	509	41856	82,23	8,2 s	5825	11,44	1,9 s
16	3	4060	665765	163,98	127 s	75873	18,69	22,5 s
18	3	41301	11887987	287,83	2258 s	1147792	27,79	319 s
9	4	16	268	16,75	0,0 s	159	9,94	0,0 s
10	4	59	2224	37,69	0,4 s	1066	18,07	0,2 s
11	4	265	20045	75,64	3,3 s	7940	29,96	1,7 s
12	4	1544	194437	125,93	31,2 s	64745	41,93	12,7 s
13	4	10778	2028053	188,17	319 s	577369	53,57	111 s

Tabelle 3.1: Wirkung des Taillenkriteriums.

Man spart nicht nur einen Kanonizitätstest, sondern auch das Einfügen weiterer Kanten (siehe dazu auch Abbildung 3.3).

3.4.7 Bemerkung:

Mit dem Taillenkriterium läßt sich eine beträchtliche Reduzierung der Kandidatenmenge bewirken, wie Tabelle 3.1 belegt. Dabei steht in Spalte

Graphen: die Anzahl der zusammenhängenden k-regulären Graphen mit n Knoten;

Kandidaten: jeweils die Anzahl der numerierten zusammenhängenden k-regulären Graphen, die auf Kanonizität getestet werden müssen;

Faktor: jeweils das Verhältnis Kandidaten/Graphen;

Zeit: jeweils die benötigte CPU-Zeit auf einem PC 486 DX2/66.

Es wäre zu untersuchen, ob stärkere Aussagen als Satz 3.4.1 gültig sind und wie sich solche auf die Kandidatenmenge auswirken würden.

3.4.8 Bemerkung:

Wir können nun auch $rep_<(S_n \backslash\backslash \mathcal{R}_{n,k,t})$ mit $t > 3$ konstruieren. Dazu müssen wir die Datenstruktur für den Graphen am Anfang so initialiseren, daß sie $\bar{T}_{n,k,t}$ enthält. Schließt Kante e dann einen ersten Kreis, so daß girthcheck$(\Gamma \cup \{e\})$ korrekt abläuft, dann hat dieser Kreis Länge $\geq t$. Somit haben auch alle später geschlossenen Kreise Länge $\geq t$ und der reguläre Graph am Ende die gewünschte Taillenweite. Es soll aber nicht verschwiegen werden, daß die derzeitige Implementierung im Hinblick auf die Konstruktion regulärer Graphen mit großer Taillenweite noch wesentlich verbessert werden kann.

3.5 Der Lerneffekt

Mit Hilfe der Ergebnisse eines negativ verlaufenen Kanonizitätstests können wir die Kandidatenmenge weiter reduzieren oder mit den Worten von [Gru1]:

Man kann etwas lernen !

Der folgende Satz beschreibt, wie wir im Falle eines nichtkanonischen Testkandidaten ein notwendiges Kriterium für die Kanonizität seiner lexikographischen Nachfolger erhalten.

3.5.1 Satz:
Sei $\Gamma \in \mathcal{G}_n$ nicht kanonisch, $\Gamma = \{e_1, ..., e_t\}$ mit $e_1 < e_2 < ... < e_t$
$\Longrightarrow \exists \pi \in S_n : \Gamma^\pi < \Gamma$
$\Longrightarrow \exists i < t : \Gamma^\pi = \{e_1, ..., e_i, e'_{i+1}, ..., e'_t\}$ mit $e'_{i+1} < e_{i+1}$.
Es ist $\{e_1^{\pi^{-1}}, ..., e_i^{\pi^{-1}}, e'^{\pi^{-1}}_{i+1}\} = \{e_{j_1}, ..., e_{j_{i+1}}\}$ mit $1 \leq j_l \leq t \ \forall l = 1, ..., i+1$.
Sei $r := \max\{j_1, ..., j_{i+1}, i+1\} \Longrightarrow$ alle $\tilde{\Gamma} \in \mathcal{G}_n$ mit $\tilde{\Gamma} = \{e_1, ..., e_r, \tilde{e}_{r+1}, ..., \tilde{e}_s\}$,
$e_1 < ... < e_r < \tilde{e}_{r+1} < ... < \tilde{e}_s$ sind ebenfalls nicht kanonisch.

Beweis: Da $r \geq i+1$ ist $\tilde{\Gamma} = \{e_1, ..., e_i, e_{i+1}, ..., e_r, \tilde{e}_{r+1}, ..., \tilde{e}_s\}$. Außerdem ist $\tilde{\Gamma}^\pi = \{e_1^\pi, ..., e_r^\pi, \tilde{e}_{r+1}^\pi, ..., \tilde{e}_s^\pi\} \supseteq \{e_{j_1}^\pi, ..., e_{j_{i+1}}^\pi\} = \{e_1, ..., e_i, e'_{i+1}\} \Longrightarrow \tilde{\Gamma}^\pi < \tilde{\Gamma}$ wegen $e'_{i+1} < e_{i+1}$. □

3.5.2 Anwendung:
Für die praktische Anwendung bedeutet dies: Wurde ein nicht-kanonischer Kandidat getestet, ermittle r wie im obigen Lemma und gehe in der Rekursion ORDREK(x, y, v) soweit zurück, bis $e_r \notin \Gamma$. Je kleiner e_r ist, desto mehr Kandidaten können vernachlässigt werden.

3.5.3 Bemerkung:
Allerdings liefert ein negativ verlaufener Kanonizitätstest nicht immer diejenige Permutation, die den größten Lerneffekt ermöglicht. Dies liegt daran, daß der aufwendige Kanonizitätstest beim ersten gefundenen $\pi \in S_n$ mit der Eigenschaft $\Gamma^\pi < \Gamma$ abbricht (siehe Kapitel 4). Insgesamt wäre es aber nicht rentabel, immer nach dem Element weiterzusuchen, welches den größtmöglichen Lerneffekt erzielt.
Wir wollen noch präzisieren, wie wir e_r berechnen (dazu werden die Bezeichnungen von Satz 3.5.1 beibehalten): Im Falle eines negativ verlaufenen Kanonizitätstests erhalten wir π mit $\Gamma^\pi < \Gamma$ und die Kante $(v, w) = e_{i+1}{}'$. Dann ist
$\{e_1^{\pi^{-1}}, ..., e_i^{\pi^{-1}}, e'^{\pi^{-1}}_{i+1}\} = \{(x, y)^{\pi^{-1}} \mid (x, y) \in \Gamma, \ (x, y) < (v, w)\} \cup \{(v, w)^{\pi^{-1}}\}$ und
$e_r = \max\Big(\{(x, y)^{\pi^{-1}} \mid (x, y) \in \Gamma, \ (x, y) < (v, w)\} \cup \{(v, w)^{\pi^{-1}}\} \cup \{(v, w)\}\Big)$.
Die Kante e_r läst sich also mit sehr geringem Aufwand ermitteln. Angesichts dessen ist es überraschend, welche Wirkung man mit diesem Lerneffekt erzielt. Zur Demonstration sei folgendes Beispiel angeführt:

3.5.4 Beispiel:
Wir betrachten $\Gamma \in \mathcal{R}_{9,4}$ gegeben durch

$$\begin{aligned}\Gamma \;=\; &\{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),\\ &(4,5),(4,8),(5,9),(6,7),(6,8),(6,9),(7,8),(7,9),(8,9)\}\end{aligned}$$

Γ ist nicht kanonisch. Mit $\pi = (3,2)$ ist

$$\Gamma^\pi = \{(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),...\} < \Gamma.$$

Mit den Bezeichnungen aus Bemerkung 3.5.3 ist $(v,w) = (2,4)$. Wir erhalten

$$e_r = \max(\{(1,2),(1,3),(1,4),(1,5),(2,3)\} \cup \{(3,4)\} \cup \{(2,4)\}) = (3,4).$$

In diesem Fall können 37 Kandidaten übersprungen werden:

{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),(4,6),(4,7),(5,8),(5,9),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),(4,6),(4,8),(5,7),(5,9),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),(4,6),(4,8),(5,8),(5,9),(6,7),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),(4,8),(4,9),(5,6),(5,7),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),(4,8),(4,9),(5,6),(5,8),(6,7),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,5),(4,8),(4,9),(5,8),(5,9),(6,7),(6,8),(6,9),(7,8),(7,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,5),(4,7),(5,8),(5,9),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,5),(4,8),(5,7),(5,9),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,5),(4,8),(5,8),(5,9),(6,7),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,6),(4,8),(5,7),(5,8),(5,9),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,7),(4,8),(5,6),(5,8),(5,9),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,7),(4,8),(5,7),(5,8),(5,9),(6,8),(6,9),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,8),(4,9),(5,6),(5,7),(5,8),(5,9),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,8),(4,9),(5,6),(5,8),(5,9),(6,7),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,8),(4,9),(5,7),(5,8),(5,9),(6,7),(6,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,6),(4,8),(4,9),(5,7),(5,8),(5,9),(6,8),(6,9),(7,8),(7,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,6),(5,7),(5,9),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,6),(5,8),(5,9),(6,7),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,8),(5,6),(5,9),(6,7),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,9),(5,6),(5,7),(6,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,9),(5,6),(5,8),(6,7),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,9),(5,6),(5,9),(6,7),(6,8),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,5),(4,9),(5,8),(5,9),(6,7),(6,8),(6,9),(7,8),(7,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,7),(5,6),(5,8),(5,9),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,8),(5,6),(5,7),(5,9),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,8),(5,7),(5,8),(5,9),(6,7),(6,9),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,9),(5,6),(5,7),(5,8),(6,9),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,9),(5,6),(5,7),(5,9),(6,8),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,9),(5,6),(5,8),(5,9),(6,7),(7,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,9),(5,7),(5,8),(5,9),(6,7),(6,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,9),(5,7),(5,8),(5,9),(6,7),(6,9),(7,8),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,6),(4,9),(5,7),(5,8),(5,9),(6,8),(6,9),(7,8),(7,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,8),(4,9),(5,6),(5,7),(5,8),(6,7),(6,9),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,8),(4,9),(5,6),(5,7),(5,9),(6,7),(6,8),(7,9),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,8),(4,9),(5,6),(5,7),(5,9),(6,7),(6,9),(7,8),(8,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,8),(4,9),(5,6),(5,7),(5,9),(6,8),(6,9),(7,8),(7,9)}
{(1,2),(1,3),(1,4),(1,5),(2,3),(2,6),(2,7),(3,4),(3,8),(4,8),(4,9),(5,6),(5,8),(5,9),(6,7),(6,9),(7,8),(7,9)}

			ohne Lerneffekt			mit Lerneffekt		
n	k	Graphen	Kandidaten	Faktor	Zeit	Kandidaten	Faktor	Zeit
10	3	19	61	3,21	0,0 s	37	1,95	0,0 s
12	3	85	513	6,04	0,2 s	214	2,52	0,1 s
14	3	509	5825	11,44	1,9 s	1406	2,76	1,0 s
16	3	4060	75873	18,69	22,5 s	10432	2,57	8,9 s
18	3	41301	1147792	27,79	319 s	96279	2,33	95,7 s
9	4	16	159	9,94	0,0 s	57	3,56	0,0 s
10	4	59	1066	18,07	0,2 s	219	3,71	0,1 s
11	4	265	7940	29,96	1,7 s	997	3,76	0,6 s
12	4	1544	64745	41,93	12,7 s	5194	3,36	3,2 s
13	4	10778	577369	53,57	111 s	33139	3,07	22,5 s

Tabelle 3.2: Wirkung des Lerneffekts.

3.5.5 Bemerkung:
Bei Beispiel 3.5.4 handelt es sich keineswegs um einen sorgfältig ausgesuchten Spezialfall. Wie man Tabelle 3.2 entnehmen kann, wird in allen Fällen eine deutliche Reduzierung der Kandidatenmenge erreicht.
Besonders bemerkenswert ist dabei die Tatsache, daß bei Verwendung des Lerneffekts das Verhältnis von Kandidaten zu Repräsentanten bei größerwerdendem n kleiner wird (abgesehen von einigen Ausnahmen für „kleine“ n), was auch Tabelle 5.3 bestätigt. Es besteht insbesondere die Vermutung, daß sich bei wachsendem n die Anzahl der Kandidaten asymptotisch an die Anzahl der gesuchten Repräsentanten annähert.

3.6 Erzeugungsbäume

Wir können uns den Erzeugungsalgorithmus als Baum vorstellen: Wurzel ist $\bar{T}_{n,k,t}$ und jeder, durch Einsetzen einer neuen Kante entstandene Graph, ist ein weiterer Knoten. In der tiefsten Ebene befinden sich nur Elemente aus $\mathcal{R}^*_{n,k,t}$ Auf den nächsten Seiten sind einige solcher Erzeugungsbäume zu verschiedenen n,k und t dargestellt. Der interessierte Leser kann anhand dieser Abbildungen das sukzessive Einsetzen der Kanten und die Anwendung der verschiedenen Kriterien nachvollziehen. In den überschaubaren Beispielen ist jeweils vermerkt, welche Kante eingesetzt wird. Die Kennzeichnung der Knoten hat folgende Bedeutung:

- ◦ : Taillenkriterium nicht erfüllt.
- ⦿ : Kanonizitätstest durchgeführt mit negativem Ergebnis.
- ● : Kanonizitätstest durchgeführt mit positivem Ergebnis.

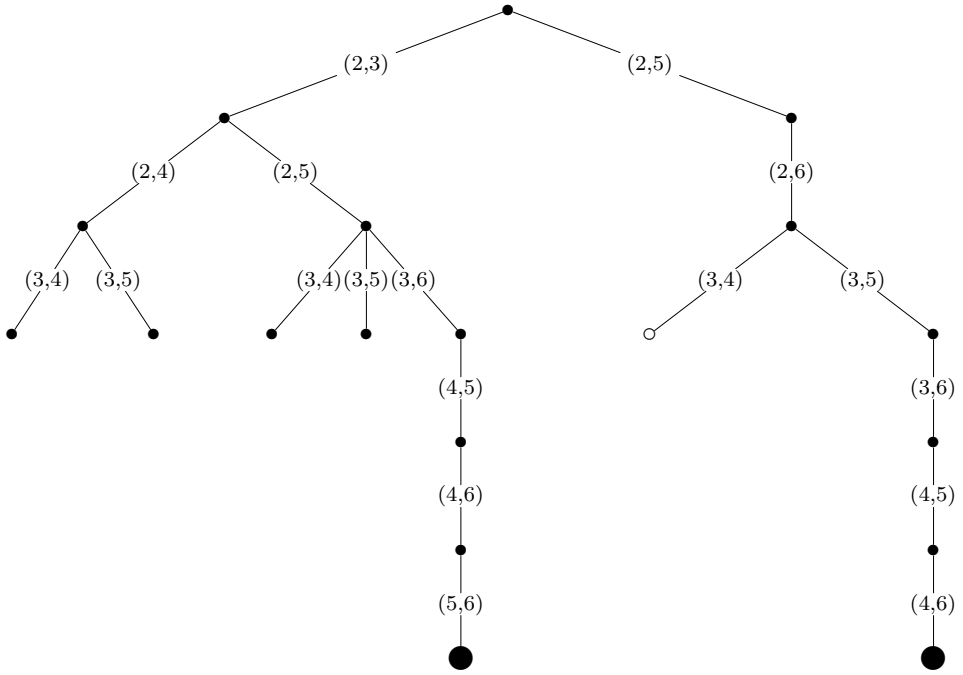


Abbildung 3.3: Der Erzeugungsbaum für $n = 6$, $k = 3$.

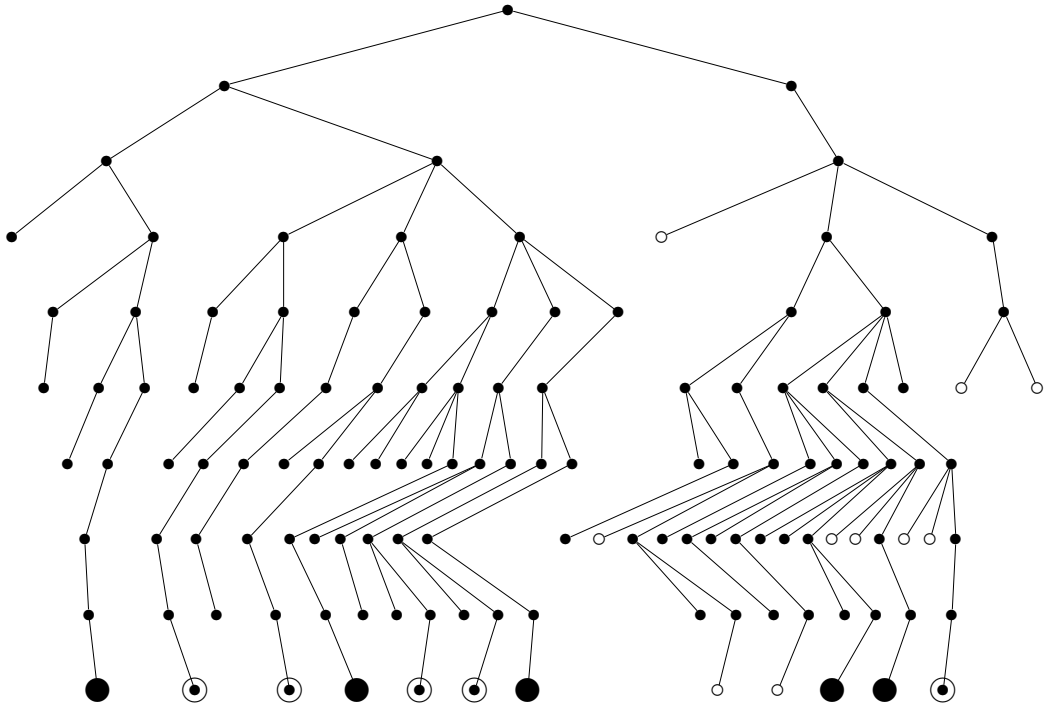


Abbildung 3.4: Der Erzeugungsbaum für $n = 8$, $k = 3$.

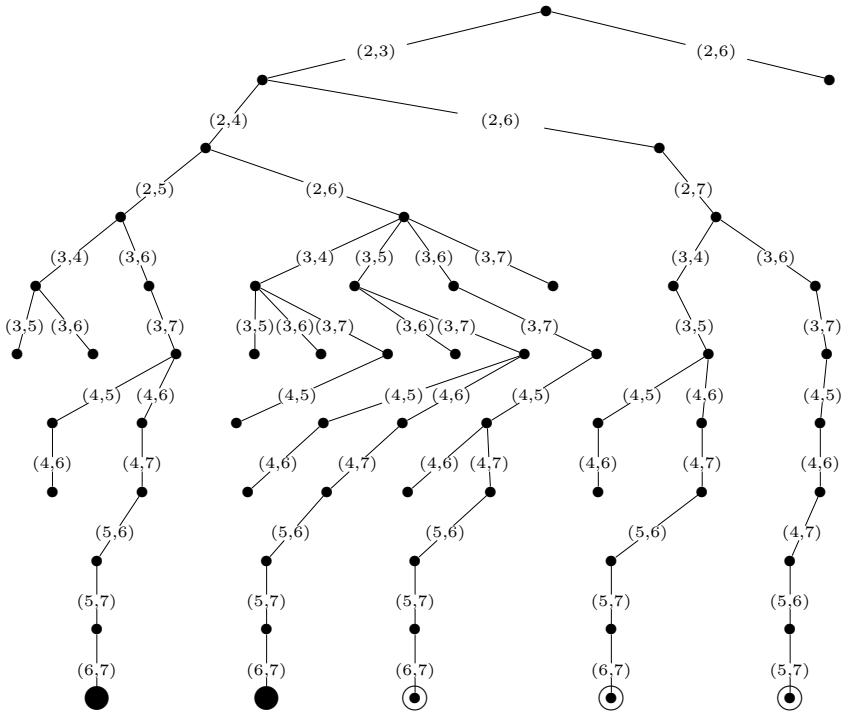


Abbildung 3.5: Der Erzeugungsbaum für $n = 7$, $k = 4$.

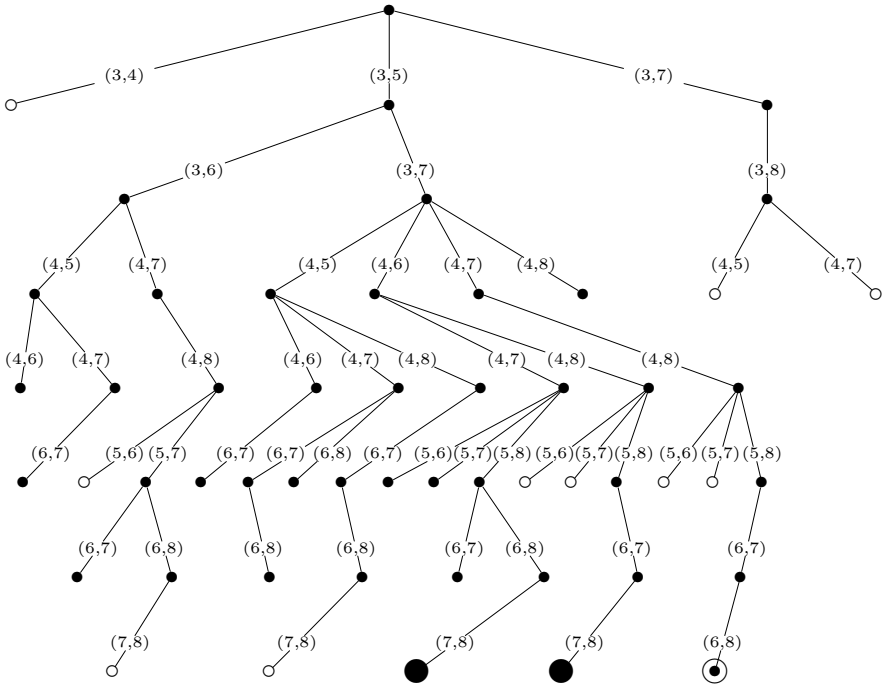


Abbildung 3.6: Der Erzeugungsbaum für $n = 8$, $k = 3$, $t = 4$.

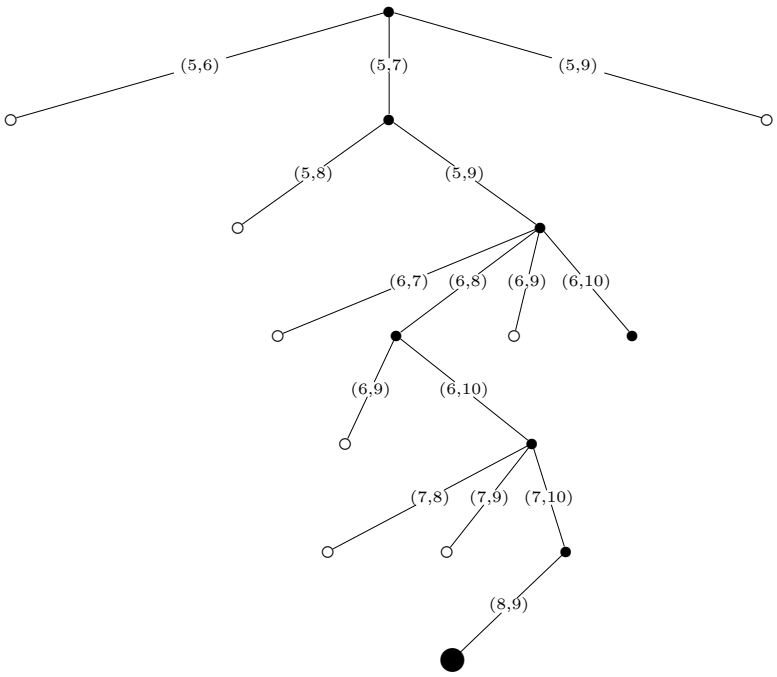


Abbildung 3.7: Der Erzeugungsbaum für $n = 10$, $k = 3$, $t = 5$.

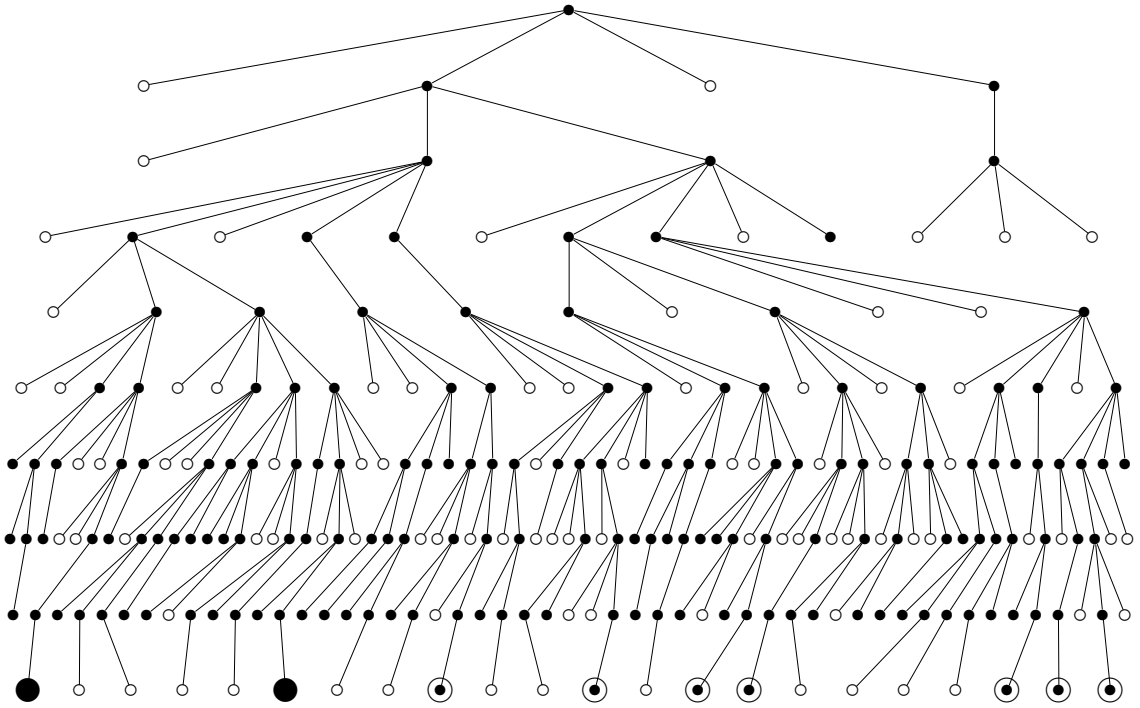


Abbildung 3.8: Der Erzeugungsbaum für $n = 12$, $k = 3$, $t = 5$.

Kapitel 4

Der Kanonizitätstest

Dieser Test ist der schwierigste Teil des ganzen Algorithmus. Am Ende wird die Effizienz der Implementierung wesentlich davon abhängen, ob es gelingt, den Aufwand für diesen Test zu beschränken. Es muß entschieden werden, ob für einen gegebenen Kandidaten $\Gamma \in \mathcal{R}_{n,k}$ gilt

$$\Gamma \leq \Gamma^\pi \qquad \forall \pi \in S_n.$$

Zuerst wird eine Durchlaufstrategie für Permutationsgruppen dargestellt, die auf Verwendung einer Reihe spezieller Untergruppen, sogenannter Simsketten, basiert. Weiter wird beschrieben, wie ein System von Erzeugern der Automorphismengruppe gefunden werden kann und sich damit eine erste Verkürzung des Tests erzielen läßt. Schließlich werden wir von der Tatsache Gebrauch machen, daß nur semikanonische Kandidaten zu testen sind. Wir benutzen die in Abschnitt 3.3 definierten Normalisatoren und das Konzept der kanonisierenden Permutation, um den Kanonizitätstest derart weiter zu verfeinern, daß von vorneherein nur noch wenige Permutationen betrachtet werden müssen. Beim zeilenweisen Vergleich der Adjazenzmatrizen von Γ und Γ^π kann dann nach jeder Zeile entschieden werden, ob Teile des Tests vernachlässigt werden dürfen. Dieses Verfahren wird an zwei ausführlichen Beispielen demonstriert.

4.1 Simsketten

Um die oben gestellte Frage beantworten zu können, müssen wir zunächst einige Techniken zum Umgang mit Permutationsgruppen erarbeiten. Wir brauchen insbesondere:

- Eine Strategie, um alle Elemente von S_n zu durchlaufen.
- Eine Datenstruktur, um die Automorphismengruppe von $\Gamma \in rep_<(S_n \backslash\backslash \mathcal{R}_{n,k})$ als Untergruppe von S_n im Speicher eines Rechners zu verwalten.

Beide Aufgaben lassen sich mit dem Prinzip der Simskette (siehe [Sim]) lösen.

4.1.1 Definition:
Sei $U \leq S_{\underline{n}}$ und für $i = 1, ..., n-1$

$$U_i := C_U(\{1, ..., i\})$$

der Zentralisator der Ziffern $1, ..., i$. Die Untergruppenkette

$$U =: U_0 \geq U_1 \geq U_2 \geq ... \geq U_{n-1} = \{id\}$$

heißt Simskette von U.

4.1.2 Satz:
Sei $1 \leq i < n$, $l(i) := \mid U_{i-1}/U_i \mid$ und $\{u_{i,1}, ..., u_{i,l(i)}\}$ ein Repräsentantensystem von U_{i-1}/U_i, wobei $u_{i,1} = id$. Dann gibt es folgende disjunkte Zerlegung von U_{i-1} in Linksnebenklassen von U_i in U_{i-1} :

$$U_{i-1} = \bigcup_{j=1}^{l(i)} u_{i,j}U_i.$$

Beweis:
Die Aussage folgt aus Satz 1.2.6 und Beispiel 1.2.7. □

4.1.3 Folgerung:
Jedes Gruppenelement $u \in U$ ist somit eindeutig darstellbar als

$$u = \prod_{i=1}^{n-1} u_{i,j_i}, \quad 1 \leq j_i \leq l(i),\ 1 \leq i < n.$$

Beweis:
Wir zeigen die Behauptung für $u \in U_i$, $n-1 \geq i \geq 0$ per Induktion nach i.
Für $i = n-1$ ist $u = id$ und die Behauptung klar.
Sei $u \in U_{i-1}$. Nach Satz 4.1.2 $\exists!\ j_i \leq l(i) : u \in u_{i,j_i}U_i$. Dann $\exists!\ u' \in U_i : u = u_{i,j_i}u'$.
Nach Induktionsvoraussetzung ist u' eindeutig darstellbar als $u' = u_{i+1,j_{i+1}}...u_{n-1,j_{n-1}}$.
Somit $u = u_{i,j_i}u_{i+1,j_{i+1}}...u_{n-1,j_{n-1}} = \prod_{\nu=1}^{n-1} u_{\nu,j_\nu}$, wobei $j_\nu = 1$ für $\nu < i$. □

4.1.4 Beispiel:
Für $U = S_n$ können wir die Zerlegung der Zentralisatoren in Linksnebenklassen explizit angeben:

$$U_{i-1} = \bigcup_{j=i}^{n} (i, j)U_i, \quad i = 1, ..., n-1.$$

Die Nebenklassenrepräsentanten sind Transpositionen. Jede Permutation $\pi \in S_n$ ist eindeutig darstellbar in der Form:

$$\pi = \prod_{i=1}^{n-1} (i, j_i), \quad i \leq j_i \ \leq n, 1 \leq i < n.$$

(i, i) bezeichnet dabei jeweils die Identität.

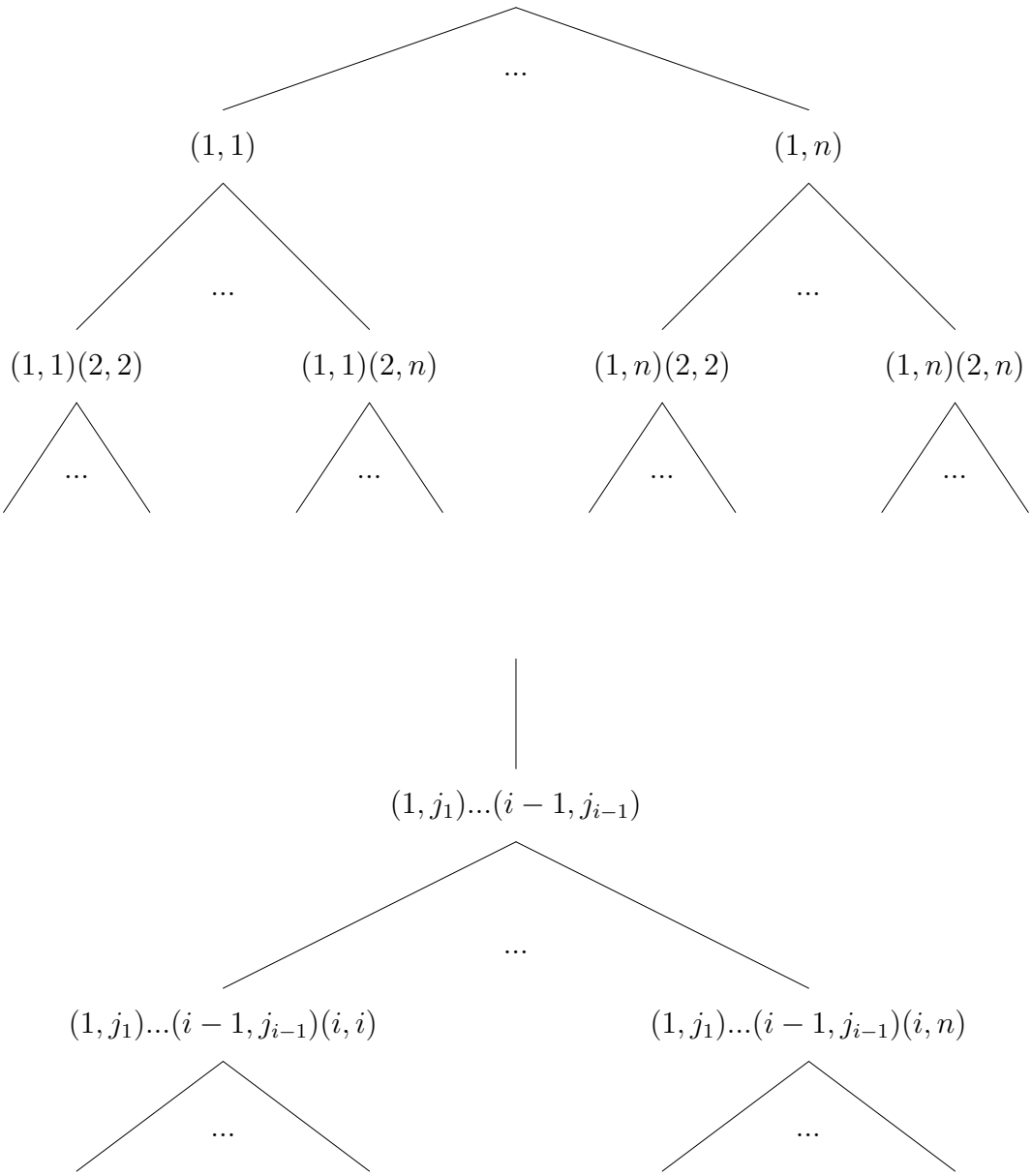


Abbildung 4.1: Der Gruppenbaum von S_n.

4.1.5 Bemerkung:
Beispiel 4.1.4 folgend, können wir uns S_n als Baum der Tiefe $n-1$ vorstellen (siehe Abbildung 4.1). Die Blätter des Gruppenbaumes sind die Elemente von S_n. Die Knoten in Tiefe 1 entsprechen den Nebenklassenrepräsentanten von U_0/U_1 und die Blätter des Zweigs von Knoten $(1,j)$ sind die Elemente von $(1,j)U_1$. Allgemein sind zu Knoten π in Tiefe i die Blätter des Zweigs von π die Elemente von πU_i. Am linken Rand haben wir in jeder Tiefe jeweils id. Damit sind für $j = 1, ..., n$ die ersten $j!$ Blätter von links gerade die Elemente von U_{n-j}.

4.1.6 Anwendung:
Indem wir den Gruppenbaum nach „depth-first " Methode durchlaufen, erreichen wir nacheinander (von links nach rechts) jedes Blatt (=Gruppenelement). Dieses Verfahren liefert einen einfachen Kanonizitätstest:

naivtest(Γ)
`Durchlaufe` S_n `nach der depth-first Methode und prüfe, ob gilt`

$$\Gamma \leq \Gamma^{(1,j_1)(2,j_2)\ldots(n-1,j_{n-1})}, \quad i \leq j_i \leq n, \quad 1 \leq i < n \ ,$$

```
wenn nein: return (nicht kanonisch);
```

Offensichtlich testet dieser Algorithmus alle $n!$ Permutationen und ist somit noch sehr uneffektiv. Die folgenden Betrachtungen haben zum Ziel, die Anzahl der Permutationen, die tatsächlich angewendet werden müssen, zu verringern.

4.2 Berechnung der Automorphismengruppe

Für viele theoretische und praktische Anwendungen ist es wichtig, nicht nur ein Repräsentantensystem von $S_n \backslash\backslash \mathcal{R}_{n,k}$, sondern auch die Automorphismengruppen der berechneten Strukturen zu kennen (siehe [Grü],[GLM] oder auch Abschnitt 5.2). Wir werden nun sehen, wie wir zu kanonischem $\Gamma \in \mathcal{R}_{n,k}$ ein Erzeugendensystem der Automorphismengruppe $Aut(\Gamma)$ finden. Dazu wollen wir die Durchlaufstrategie von 4.1.6 genauer betrachten. Wir erkennen:

- Es wird mit dem kleinsten Zentralisator $U_{n-1} = \{id\}$ begonnen und dann zum nächstgrößeren übergegangen:

$$U_{n-1} \longrightarrow U_{n-2} \longrightarrow ... \longrightarrow U_0.$$

- Dies geschieht, indem nach U_i die Nebenklassen U_{i-1}/U_i durchlaufen werden:

$$U_i \longrightarrow (i,i+1)U_i \longrightarrow ... \longrightarrow (i,n)U_i.$$

  Danach hat man alle Elemente von U_{i-1} aufgesucht.

Beim Auffinden des jeweils ersten Automorphismus in einer Nebenklasse können wir folgenden Satz anwenden:

4.2.1 Satz:
Sei $j > i$ und $\pi \in U_i$ die erste beim Durchlauf von $(i,j)U_i$ gefundene Permutation mit $\Gamma^{(i,j)\pi} = \Gamma$. Dann folgt $\Gamma \leq \Gamma^{(i,j)\sigma}\ \forall \sigma \in U_i$.

Beweis:
Es ist $\Gamma^{\tau} \geq \Gamma\ \forall \tau \in U_i$ (nach Definition von naivtest). Sei $\sigma \in U_i$, dann ist $\Gamma^{(i,j)\sigma} = \Gamma^{\pi^{-1}\sigma} \geq \Gamma$. □

4.2.2 Bemerkung:
Die übrigen Elemente von $(i,j)U_i$ dürfen vernachlässigt werden. Wir können sofort mit der nächsten Nebenklasse $(i,j+1)U_i$ fortfahren. Die beim Durchlauf der Nebenklassen von U_i gefundenen Automorphismen liefern uns ein Repräsentantensystem von $C_{Aut(\Gamma)}(\{1,...,i-1\})/C_{Aut(\Gamma)}(\{1,...,i\})$. Wir wollen noch exakt formulieren, wie man eine Simskette von $Aut(\Gamma)$ erhalten kann:

4.2.3 Definitionen und Bemerkungen:
Sei für $1 \leq i < n$

$$V_i := C_{Aut(\Gamma)}(\{1,...,i\}), \quad 1 \leq i < n$$

der Zentralisator der Ziffern $1,...,i$ von $Aut(\Gamma) =: V_0 =: V$. Dann ist

$$V_i \leq U_i, \quad 0 \leq i < n.$$

Aus unserem Kanonizitätstest kennen wir die Menge

$$J(i) := \{j \mid \exists \alpha \in (i,j)U_i \text{ mit } \Gamma^{\alpha} = \Gamma\}.$$

Wir wählen zu jedem $j \in J(i)$ ein $\alpha_{i,j} \in (i,j)U_i$ mit $\Gamma^{\alpha_{i,j}} = \Gamma$. (Nach dem oben Gesagten nehmen wir natürlich immer das erste, das wir finden.) Dann gilt der folgende

4.2.4 Satz:
Für $1 \leq i < n$ bilden die Elemente von

$$\{\alpha_{i,j} \mid j \in J(i)\}$$

ein Repräsentantensystem der Linksnebenklassen V_{i-1}/V_i.

Beweis:
Es ist zu zeigen, daß

$$V_{i-1} = \overset{\circ}{\bigcup_{j \in J(i)}} \alpha_{i,j} V_i.$$

Die Vollständigkeit der Zerlegung folgt aus der Definition vom $J(i)$. Bleibt die Disjunktheit zu zeigen: Seien $j, j' \in J(i)$, $j \neq j'$. Es ist $(i,j)U_i = \alpha_{i,j}U_i \supseteq \alpha_{i,j}V_i$ („=“ gilt wegen $\alpha_{i,j} \in (i,j)U_i$ und „$\supseteq$“ wegen $U_i \geq V_i$). Ebenso $(i,j')U_i \supseteq \alpha_{i,j}V_i$. Da $(i,j)U_i \cap (i,j')U_i = \emptyset$ folgt $\alpha_{i,j}V_i \cap \alpha_{i,j'}V_i = \emptyset$. □

4.2.5 Algorithmus:

Wir wollen diese Methode als Algorithmus formulieren. Wir brauchen dazu drei Funktionen: KATEST() steuert, welche Untergruppen und Nebenklassen durchlaufen werden müssen und ruft dazu KAREK(i) auf. KAREK(i) arbeitet rekursiv eine Nebenklasse von U_{i-1} ab. Jedesmal, wenn ein Blatt π im Gruppenbaum erreicht ist, wird ADJVGL() aufgerufen, wo die Adjazenzmatrix A des Testkandidaten $\Gamma \in \mathcal{G}_n$ mit A^π verglichen wird. Man könnte diesen Test auch bei anderen Objekten als Graphen einsetzen. Man müßte dann nur die Funktion ADJVGL() ersetzen.

KATEST():

```
(1 )    erz = 0
(2 )    for  i = n  downto  1  do
(3 )            for  j = i + 1  to  n  do
(4 )                    kmn[i] := j
(5 )                    kmn[j] := i
(6 )                    erg = KAREK(i + 1)
(7 )                    kmn[i] := i
(8 )                    kmn[j] := j
(9 )                    if erg = −1  then  return  0
(10 )           end
(11 )   end
(12 )   return  1
```

KAREK(tz):

```
(1 )    if  tz = n
(2 )            erg =ADJVGL()
(3 )            if  erg = 0
(4 )                    erz = erz + 1
(5 )                    STORE(kmn,aut[erz])
(6 )            end
(7 )            return  erg
(8 )    end
(9 )    for  i = tz  to  n  do
(10 )           CHANGE(kmn[i],kmn[tz])
(11 )           erg = KAREK(tz + 1)
(12 )           CHANGE(kmn[i],kmn[j])
(13 )           if erg ≠ 1  then  return  erg
(14 )   end
(15 )   return  1
```

ADJVGL():

```
(1 )    for  i = 1  to  n  do
(2 )            for  j = i + 1  to  n  do
(3 )                    erg = A[i][j] − A[kmn[i]][kmn[j]]
(4 )                    if erg ≠ 0  then  return  erg
(5 )            end
(6 )    end
(7 )    return  0
```

Folgende Datenstrukturen werden benötigt:

- Ein Feld kmn der Länge n, welches die jeweils aktuelle Permutation beim Durchlaufen des Gruppenbaums enthält. Vor dem ersten Aufruf von KATEST() ist kmn als Identität zu initialisieren, d.h. $kmn[i] = i \ \forall i \in \underline{n}$.

- Ein zweidimesionales Feld aut der Größe $maxerz \times n$, wo die gefundenen Automorphismen gespeichert werden. $maxerz$ ist dabei so zu wählen, daß alle v_{ij} aufgenommen werden können. Wir sind großzügig und setzen $maxerz := \frac{n(n-1)}{2}$.

- Ein zweidimensionales Feld der Größe $n \times n$ für die Adjazenzmatrix A. Natürlich kann an dieser Stelle auch eine andere Datenstruktur stehen, je nachdem welche Art von Objekten betrachtet wird.

- Eine globale Variable erz für die die Anzahl der gefundenen Automorphismen, sowie die globale Variable n und lokale Variablen i, j und erg.

In Zeile 1 von KATEST() wird die Anzahl der gespeicherten Automorphismen auf 0 gesetzt. Zeile 2 bedeutet, daß U_{i-1} zu bearbeiten ist, und Zeile 3 legt die Nebenklasse $(i,j)U_i$ fest. In Zeile 4 und 5 wird die Transposition (i,j) angewendet. Man könnte stattdessen ebenso CHANGE($kmn[i]$,$kmn[j]$) schreiben, was bedeutet, daß Inhalt von $kmn[i]$ und $kmn[j]$ vertauscht werden. In Zeile 6 erfolgt der Aufruf von KAREK($i+1$), womit Nebenklasse $(i,j)U_i$ durchlaufen wird. Als Ergebnis erg erhält man

-1, falls in $(i,j)U_i$ ein Element π gefunden wurde, so daß $A \prec A^\pi$ ist, d.h. Γ nicht kanonisch;

0, falls in $(i,j)U_i$ ein Automorphismus von Γ gefunden wurde;

1, sonst.

In Zeile 7 und 8 wird nochmals (i,j) angewendet. kmn enthält jetzt wieder id. War das Ergebnis von KAREK($i+1$) negativ, steht fest, daß Γ nicht kanonisch und es kann abgebrochen werden. Rückgabewert ist dann 0. Falls Γ kanonisch ist, wird am Ende 1 zurückgegeben.

Zur Funktion KAREK(tz):

- Ist die Bedingung in Zeile 1 erfüllt, dann wurde ein Blatt π im Gruppenbaum erreicht. Das Ergebnis erg von ADJVGL() ist

  -1, falls $A \prec A^\pi$, d.h. Γ nicht kanonisch;

  0, falls $A = A^\pi$, d.h. π ist Automorphismus;

  1, sonst.

  Wird festgestellt, daß es sich um einen Automorphismus handelt, erhöhen wir die Anzahl erz der gefundenen Automorphismen um 1 und führen die Funktion STORE(kmn,$aut[erz]$) aus. Dabei werden lediglich die Inhalte von kmn nach $aut[erz]$ kopiert. Zeile 7 bedeutet, daß wir im Gruppenbaum wieder einen Schritt zurück (nach oben) gehen.

- Ist die Bedingung in Zeile 1 nicht erfüllt, dann befinden wir uns an einem inneren Knoten im Gruppenbaum, genauer gesagt in Tiefe $tz - 1$. Dann bewirkt die in Zeile 9 beginnende for-Anweisung, daß nacheinander alle Söhne im Gruppenbaum aufgesucht werden. Zeile 13 prüft, ob inzwischen ein Automorphismus gefunden oder nicht-Kanonizität festgestellt wurde und bricht ggf. ab.

ADJVGL() führt den lexikographischen Vergleich von A und A^π durch.

4.2.6 Beispiel:

Wir wollen noch ein konkretes Beispiel betrachten. Sei $\Gamma \in \mathcal{R}_{6,3}$ durch seine Adjazenzmatrix A gegeben:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}.$$

In der folgenden Tabelle werden wir für jede, beim Kanonizitätstest durchlaufene Nebenklasse $(i, j)U_i$ den ersten gefundenen Automorphismus, die Anzahl der tatsächlich getesteten Permutationen und die Gesamtzahl ihrer Elemente aufführen (id wird dabei nicht berücksichtigt).

Nebenklasse	Automorphismus	getestet	gesamt
$(5,6)U_5$	(5,6)	1	1
$(4,5)U_4$	—	2	2
$(4,6)U_4$	—	2	2
$(3,4)U_3$	(3,4)	1	6
$(3,5)U_3$	—	6	6
$(3,6)U_3$	—	6	6
$(2,3)U_2$	(2,3)	1	24
$(2,4)U_2$	(2,4)	1	24
$(2,5)U_2$	—	24	24
$(2,6)U_2$	—	24	24
$(1,2)U_1$	(1,2)(2,5)(3,6)	91	120
$(1,3)U_1$	(1,3)(2,5)(3,6)	91	120
$(1,4)U_1$	(1,4)(2,5)(3,6)	91	120
$(1,5)U_1$	(1,5)	1	120
$(1,6)U_1$	(1,6)	1	120
		343	719

Wir können in diesem Beispiel über die Hälfte der Permutationen von S_6 vernachlässigen. Besonders wichtig sind dabei die Einsparungen bei den Nebenklassen von U_1. Leider werden die Automorphismen in $(1,2)U_1$, $(1,3)U_1$ und $(1,4)U_1$ erst sehr spät gefunden, so daß jeweils nur noch 29 Elemente übersprungen werden können. Unser Kanonizitätstest weißt also noch einige Unzulänglichkeiten auf:

4.2.7 Bemerkung:
Es gibt Situationen, in denen man frühzeitig entscheiden kann, daß der aktuelle Zweig des Gruppenbaums nicht durchlaufen werden muß:
Sind wir im Gruppenbaum in Tiefe i bei Knoten $(1,j_1)...(i,j_i) =: \pi$ angelangt, dann sind alle Blätter im Teilbaum von π Elemente aus πU_i. Die linke obere Teilmatrix von A^π mit i Zeilen und Spalten bezeichnen wir mit $A^\pi|_i$. Diese Teilmatrix bleibt unverändert bei Anwendung von $\tau \in U_i$ auf A^π: $A^\pi|_i = A^{\pi\tau}|_i$. Oft kann man schon sehr bald an $A^\pi|_i$ entscheiden, daß πU_i vernachlässigt werden darf. Einfachstes Beispiel: $i = 2$ und $A^{(1,j_1)(2,j_2)}(1,2) = 0$. Wir werden im folgenden Abschnitt erarbeiten, wie sich solche Situationen erkennen und von vorneherein vermeiden lassen. Beim Durchlauf des Gruppenbaums in Tiefe i muß nämlich in den meisten Fällen nicht mit allen Nebenklassenrepräsentanten von U_{i-1}/U_i multipliziert werden.
Wir kommen bei diesen Betrachtungen wieder auf die bereits in 3.3 definierten Normalisatoren N_i zurück und wenden kanonisierende Permutationen an. Wir nehmen eine „kanonische Lichtung des Gruppenbaums“ (siehe [Gru1]) vor:

4.3 Kanonische Lichtung des Gruppenbaums

Wir können den Kanonizitätstest noch wesentlich effizienter durchführen, indem wir die spezielle Struktur der Testkandidaten berücksichtigen. Im folgenden werden wir voraussetzen, daß $\Gamma \in \mathcal{R}_{n,k}$ semikanonisch ist. A sei die Adjazenzmatrix von Γ, a_i die i-te Zeile von A. Wir verwenden die Bezeichnungen aus Abschnitt 3.3:

$$\begin{aligned} N_0 &:= S_n = S_{\eta^{(0)}}, \\ C_1 &= C_{N_0}(\{1\}) = S_{\zeta^{(1)}}, \\ N_i &= N_{C_i}(\Gamma_i) = \{\pi \in C_i \mid {a_i}^\pi = a_i\} = S_{\eta^{(i)}}, \\ C_{i+1} &= C_{N_i}(\{1,...,i+1\}) = \{\pi \in N_i \mid (i+1)^\pi = i+1\} = S_{\zeta^{(i+1)}}, \end{aligned}$$

wobei $1 \leq i < n$ und $\eta^{(0)} = (n)$.

4.3.1 Bemerkung:
Es gibt folgende disjunkte Zerlegung der Normalisatoren N_{i-1} in Linksnebenklassen von C_i in N_{i-1}:

$$N_{i-1} = \bigcup_{j=i}^{\eta_i^{(i-1)}+i-1} (i,j)C_i, \quad i = 1,...,n.$$

Wir werden im folgenden zeigen, daß beim Durchlauf des Gruppenbaums in Tiefe i nicht mit allen Nebenklassenrepräsentanten von U_{i-1}/U_i multipliziert werden muß, sondern nur solche von C_{i-1}/N_i betrachtet werden brauchen.

4.3.2 Definition:
Wir definieren folgende Mengen:

$$\begin{aligned} M_1^{\succ} &:= \{\pi \in S_n \mid {a_1}^\pi \succ a_1\}, \\ M_1^{\prec} &:= \{\pi \in S_n \mid {a_1}^\pi \prec a_1\}, \\ M_1^{=} &:= \{\pi \in S_n \mid {a_1}^\pi = a_1\}. \end{aligned}$$

Es ist $S_n = M_1^{\succ} \cup M_1^{\prec} \cup M_1^{=}$ und $\mathrm{Aut}(\Gamma) \subseteq M_{=}^{(1)}$. Da mit $\succ$ die Matrizen zeilenweise verglichen werden, gilt offensichtlich:

$$\begin{aligned} \pi \in M_1^{\succ} &\implies A^\pi \succ A \quad (\Gamma \text{ nicht kanonisch}), \\ \pi \in M_1^{\prec} &\implies A^\pi \prec A, \\ \pi \in M_1^{=} &\implies A^\pi \succ A \ \vee \ A^\pi \prec A \ \vee \ A^\pi = A. \end{aligned}$$

Ziel ist es, die Mengen $M_1^{\succ}$, $M_1^{\prec}$ und $M_1^{=}$ möglichst einfach mit Hilfe des Gruppenbaums zu beschreiben. Dazu formulieren wir das folgende

4.3.3 Lemma:
Es sei die Nebenklassenzerlegung wie in 4.1.2 gegeben:

$$N_0 = \bigcup_{j=1}^{n} (1,j)C_1.$$

Für jedes j sei $\sigma_{1,j} \in C_1$ eine kanonisierende Permutation von Zeile $a_1^{(1,j)}$. Dann gilt:

$$(1,j)\sigma_{1,j}N_1 \subseteq M_1^{=} \text{ und } (1,j)\sigma_{1,j}(C_1 - N_1) \subseteq M_1^{\prec}.$$

Beweis:
Da Γ regulär ist, gilt ${a_1}^{(1,j)\sigma_{1,j}} = a_1$. Nach Definition von N_1 ist $\forall\sigma \in N_1$: ${a_1}^{\sigma} = a_1$, und somit ${a_1}^{(1,j)\sigma_{1,j}\sigma} = a_1 \forall\sigma \in N_1$, also $(1,j)\sigma_{1,j}N_1 \subseteq M_1^{=}$.
Da Γ semikanonisch ist, gilt $\forall\sigma' \in C_1$: ${a_1}^{\sigma'} \preceq a_1$ und mit der Definition von N_1 folgt $\forall\sigma \in C_1 - N_1$: ${a_1}^{\sigma} \prec a_1$. Wir haben dann ${a_1}^{(1,j)\sigma_{1,j}\sigma} = {a_1}^{\sigma} \prec a_1 \ \forall\sigma \in C_1 - N_1$, also $(1,j)\sigma_{1,j}(C_1 - N_1) \subseteq M_1^{\prec}$. □

4.3.4 Bemerkung:
Wir bekommen mit Hilfe der kanonisierenden Permutation $\sigma_{1,j}$ von Zeile $a_1^{(1,j)}$ eine Zerlegung der Nebenklasse $(1,j)C_1$ in einen $M_1^{\prec}$- und einen $M_1^{=}$-Anteil. Ab Zeile 2 interessieren uns nur noch Elemente aus

$$M_1^{=} = \bigcup_{j \in \underline{n}} (1,j)\sigma_{1,j}N_1.$$

Da diese Elemente Zeile 1 festlassen, brauchen wir Zeile 1 nicht mehr zu betrachten. Wir können diese Vorgehensweise dahingehend verallgemeinern, daß wir beim Durchlauf des Gruppenbaums in Tiefe i, $1 \leq i \leq n-2$ jeweils nur Zeile i untersuchen müssen.

4.3.5 Definition:
Dazu definieren wir für $1 \leq i \leq n-2$ folgende Mengen:

$$\begin{aligned}
M_i^{\succ} &:= \{\pi \in S_n \mid {a_j}^{\pi} = a_j,\ j = 1,...,i-1,\ {a_i}^{\pi} \succ a_i\},\\
M_i^{\prec} &:= \{\pi \in S_n \mid {a_j}^{\pi} = a_j,\ j = 1,...,i-1,\ {a_i}^{\pi} \prec a_i\},\\
M_i^{=} &:= \{\pi \in S_n \mid {a_j}^{\pi} = a_j,\ j = 1,...,i-1,\ {a_i}^{\pi} = a_i\}.
\end{aligned}$$

Zur Bestimmung dieser Mengen benutzen wir den folgenden

4.3.6 Satz:
Sei $\pi \in M_{i-1}^{=}$, $(i,j) \in N_{i-1}$ und $\sigma_{i,j} \in C_i$ eine kanonisierende Permutation von ${a_i}^{\pi(i,j)}$. Dann gilt:

$$\begin{array}{lrcl}
\text{(i)} & {a_i}^{\pi(i,j)\sigma_{i,j}} \succ a_i & \Longrightarrow & \pi(i,j)\sigma_{i,j} \in M_i^{\succ} \quad (\Gamma \text{ nicht kanonisch}), \\
\text{(ii)} & {a_i}^{\pi(i,j)\sigma_{i,j}} \prec a_i & \Longrightarrow & \pi(i,j)\sigma_{i,j}C_i \subseteq M_i^{\prec}, \\
\text{(iii)} & {a_i}^{\pi(i,j)\sigma_{i,j}} = a_i & \Longrightarrow & \pi(i,j)\sigma_{i,j}N_i \subseteq M_i^{=} \ \wedge \\
 & & & \pi(i,j)\sigma_{i,j}(C_i - N_i) \subseteq M_i^{\prec}.
\end{array}$$

Beweis:

i) Da $\pi \in M_{i-1}^{=}$, $(i,j) \in N_{i-1}$ und $\sigma_{i,j} \in C_i$ gilt ${a_l}^{\pi(i,j)\sigma_{i,j}} = a_l$ für $1 \leq l < i$. Wir haben $A^{\pi(i,j)\sigma_{i,j}} \succ A$.

ii) Nach Definition der kanonisierenden Permutation ist $\forall \sigma \in C_i$: ${a_i}^{\pi(i,j)\sigma_{i,j}} \succeq {a_i}^{\pi(i,j)\sigma}$. Es gilt $a_i \succ {a_i}^{\pi(i,j)\sigma_{i,j}} \succeq {a_i}^{\pi(i,j)\sigma} \forall \sigma \in C_i$, also folgt $\pi(i,j)C_i = \pi(i,j)\sigma_{i,j}C_i \subseteq M_i^{\prec}$.

iii) Nach Definition von N_i ist $\forall \sigma \in N_i$: ${a_i}^{\sigma} = a_i$ und somit ${a_i}^{\pi(i,j)\sigma_{i,j}\sigma} = {a_i}^{\sigma} = a_i \forall \sigma \in N_i$, also $\pi(i,j)\sigma_{i,j}N_1 \subseteq M_i^{=}$.
Da Γ semikanonisch ist, gilt $\forall \sigma' \in C_i$: $a_i \succeq {a_i}^{\sigma'}$ und mit der Definition von N_i folgt $\forall \sigma \in C_i - N_i$: $a_i \succ {a_i}^{\sigma}$. Wir haben dann $a_i \succ {a_i}^{\sigma} = {a_i}^{\pi(i,j)\sigma_{i,j}\sigma} \forall \sigma \in C_i - N_i$, also $\pi(i,j)\sigma_{i,j}(C_i - N_i) \subseteq M_i^{\prec}$.

□

4.3.7 Bemerkung:
Wir wenden diesen Satz zunächst auf die Elemente $\pi = (1,j_1)\sigma_{1,j_1}$ und $(2,j_2) \in N_1$ an. Es kann durchaus vorkommen, daß man schon beim Untersuchen von Zeile 2 feststellt, daß A nicht kanonisch ist (siehe Beispiel 4.3.10). Besonders effektiv ist auch Fall (ii), bei dem die ganze Nebenklasse $\pi(2,j_2)C_2$ ausscheidet. Im schlechtesten Fall erhalten wir immerhin wieder eine Aufspaltung in $M_2^{\prec}$- und $M_2^{=}$- Anteile. Wiederholt angewendet erhalten wir so die Menge

$$M_i^{=} = \bigcup_{\substack{(j_1,...,j_i) : \forall l = 1,...,i : \\ {a_l}^{(1,j_1)\sigma_{1,j_1}...(i,j_l)\sigma_{i,j_l}} = a_l,}} (1,j_1)\sigma_{1,j_1}...(i,j_i)\sigma_{i,j_i}N_i$$

Dabei ist für $l = 1,...,i$ $\sigma_{l,j_l} \in C_l$ jeweils eine kanonisierende Permutation der Zeile ${a_l}^{(1,j_1)\sigma_{1,j_1}...(l-1,j_{l-1})\sigma_{l-1,j_{l-1}}(l,j_l)}$.

4.3.8 Anwendung:
Wir durchlaufen den Gruppenbaum wie gehabt nach der „defth-first“ Methode. Anders als zuvor nehmen wir nun in Tiefe i eine Lichtung nach Satz 4.3.6 vor. Dabei ist jeweils folgender Zeilentest durchzuführen: Zu gegebenem $\tau \in M^{=}_{(i-1)}$ muß man die Zeile ${a_i}^{\tau}$ und eine kanonisierende Permutation σ von Zeile ${a_i}^{\tau}$ berechnen, sowie einen lexikographischen Vergleich von ${a_i}^{\tau\sigma}$ mit a_i auswerten. Dies kann in einem Schritt durchgeführt werden, indem man innerhalb der $\zeta^{(i)}$-Blöcke die Einsen von Zeile ${a_i}^{\tau\sigma}$ nach links sortiert und dabei überprüft, ob deren Anzahl mit der Anzahl der Einsen von a_i in dem entsprechenden Block übereinstimmt.
Tritt Situation (i) bzw. (ii) ein, so kann man den Test abbrechen, bzw. muß den aktuellen Zweig im Gruppenbaum nicht mehr verfolgen. Im schlechtesten Fall (iii) ist $\tau\sigma N_i$ weiter zu untersuchen. Dabei wenden wir die Zerlegung 4.3.1 von N_i an. Es sind dann also $\eta^{(i)}_{i+1}$ Söhne im Gruppenbaum zu betrachten. Wenn wir im ungünstigsten Fall bei Satz 4.3.6 immer Gleichheit feststellen, so müssen insgesamt $\sum_{i=1}^{n-1} \prod_{j=1}^{i} \eta_i^{(i-1)}$ zeilenweise Vergleiche durchgeführt werden.
Dann kann natürlich verstärkt eine Lichtung des Gruppenbaums durch das Auffinden von Automorphismen wie in Satz 4.2.1 vorgenommen werden. Die beiden Lichtungskriterien lassen sich ohne weiteres kombinieren und „ergänzen“ sich sogar gewissermaßen. Die Wirkung des verbesserten Kanonizitätstests wollen wir anhand des Beispiels aus Abschnitt 4.2 nachvollziehen:

4.3.9 Beispiel:
Wir betrachten $\Gamma \in \mathcal{R}_{6,3}$ mit der Adjazenzmatrix

$$A = \left(\begin{array}{c|c|c|c|c|c} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{array}\right).$$

Dabei sind jeweils die $\zeta^{(i)}$-Blöcke eingezeichnet. Es ist

$$\begin{aligned} N_0 &= S_{(6)} = C_1 \cup (1,2)C_1 \cup (1,3)C_1 \cup (1,4)C_1 \cup (1,5)C_1 \cup (1,6)C_1, \\ N_1 &= S_{(1,3,2)} = C_2 \cup (2,3)C_2 \cup (2,4)C_2, \\ N_2 &= S_{(1,1,2,2)} = C_3 \cup (3,4)C_3, \\ N_3 &= N_4 = S_{(1,1,1,1,2)} = C_5 \cup (5,6)C_5, \\ N_5 &= S_{(1,1,1,1,1,1)} = \{id\}. \end{aligned}$$

Von vorneherein sind somit höchstens $6+6*3+6*3*2+6*3*2*1+6*3*2*1*2 = 168$ Zeilentests durchzuführen. Die folgende Tabelle enthält eine detaillierte Analyse des verfeinerten Kanonizitätstests.

Nebenklasse	Tiefe	Baumdurchlauf	kan. Perm.	lex. Vergleich	
(5,6)C_5	5	(5,6)	id	$({a_5}^{(5,6)} = a_5)$	(A)
(3,4)C_3	3	(3,4)	id	${a_3}^{(3,4)} = a_3$	
	4	(3,4)	id	${a_4}^{(3,4)} = a_4$	(A)
(2,3)C_2	2	(2,3)	id	${a_2}^{(2,3)} = a_2$	
	3	(2,3)	id	${a_3}^{(2,3)} = a_3$	
	4	(2,3)	id	${a_4}^{(2,4)} = a_4$	(A)
(2,4)C_2	2	(2,4)	id	${a_2}^{(2,4)} = a_2$	
	3	(2,4)	id	${a_3}^{(2,4)} = a_3$	
	4	(2,4)	id	${a_4}^{(2,4)} = a_4$	(A)
(1,2)C_1	1	(1,2)	(3,6)(4,5)	${a_1}^{(1,2)(3,6)(4,5)} = a_1$	
	2	(1,2)(3,6)(4,5)	id	${a_2}^{(1,2)(3,6)(4,5)} = a_2$	
	3	(1,2)(3,6)(4,5)	id	${a_3}^{(1,2)(3,6)(4,5)} = a_3$	
	4	(1,2)(3,6)(4,5)	id	${a_4}^{(1,2)(3,6)(4,5)} = a_4$	(A)
(1,3)C_1	1	(1,3)	(2,6)(4,5)	${a_1}^{(1,3)(2,6)(4,5)} = a_1$	
	2	(1,3)(2,6)(4,5)	id	${a_2}^{(1,3)(2,6)(4,5)} = a_2$	
	3	(1,3)(2,6)(4,5)	id	${a_3}^{(1,3)(2,6)(4,5)} = a_3$	
	4	(1,3)(2,6)(4,5)	id	${a_4}^{(1,3)(2,6)(4,5)} = a_4$	(A)
(1,4)C_1	1	(1,4)	(2,6)(3,5)	${a_1}^{(1,4)(2,6)(3,5)} = a_1$	
	2	(1,4)(2,6)(3,5)	id	${a_2}^{(1,4)(2,6)(3,5)} = a_2$	
	3	(1,4)(2,6)(3,5)	id	${a_3}^{(1,4)(2,6)(3,5)} = a_3$	
	4	(1,4)(2,6)(3,5)	id	${a_4}^{(1,4)(2,6)(3,5)} = a_4$	(A)
(1,5)C_1	1	(1,5)	id	${a_1}^{(1,5)} = a_1$	
	2	(1,5)	id	${a_2}^{(1,5)} = a_2$	
	3	(1,5)	id	${a_3}^{(1,5)} = a_3$	
	4	(1,5)	id	${a_4}^{(1,5)} = a_4$	(A)
(1,6)C_1	1	(1,6)	id	${a_1}^{(1,6)} = a_1$	
	2	(1,6)	id	${a_2}^{(1,6)} = a_2$	
	3	(1,6)	id	${a_3}^{(1,6)} = a_3$	
	4	(1,6)	id	${a_4}^{(1,6)} = a_4$	(A)

Es ist aufgeführt, in welcher Tiefe beim Baumdurchlauf welches Gruppenelement betrachtet wird. Weiter ist jeweils die gewählte kanonisierende Permutation und das Ergebnis des lexikographischen Vergleichs angegeben. Wird festgestellt, daß man einen Automorphismus gefunden hat, so ist dies mit einem (A) vermerkt.
Insgesamt werden also 29 Zeilentests durchgeführt. Ein Blick auf 4.2.6 macht die Verbesserung deutlich, zumal es sich bei der alten Version jeweils um Vergleiche der gesamten (halben) Adjazenzmatrix handelt.
Zum besseren Verständnis sei noch bemerkt: Solange wir beim Baumdurchlauf nur id betrachten, werden natürlich keine Zeilentests durchgeführt. Stimmen die zu vergleichenden Matrizen bis Zeile 4 überein, dann brauchen wir die beiden letzten Zeilen nicht mehr zu vergleichen. Es ist ein Automorphismus gefunden. In Tiefe 1 ist eigentlich nur die kanonisierende Permutation zu bestimmen. Gleichheit ist dann klar wegen der Re-

gularität.
Abbildung 4.2 zeigt den Gruppenbaum für Γ. Die durch die Zerlegung der N_i in Nebenklassen von C_{i+1} gegebene Lichtung wurde dabei schon vorgenommen. Der Baum hat somit $1+6+6*3+6*3*2+6*3*2*1+6*3*2*1*2 = 169$ Knoten (davon $6*3*2*1*2 = 72$ Blätter). Knoten, bei denen tatsächlich ein Zeilentest durchgeführt werden muß sind mit einem „ ● “ markiert.

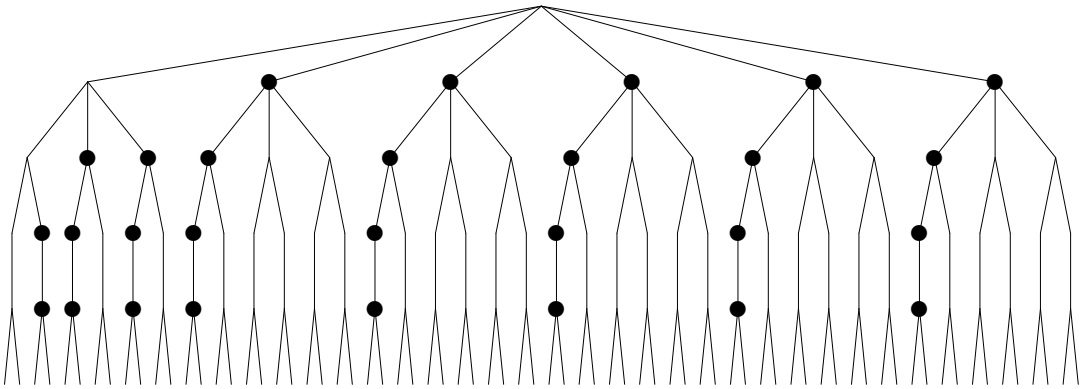


Abbildung 4.2: Ein gelichteter Gruppenbaum.

4.3.10 Beispiel:
Wir wollen noch unser Beispiel aus 3.5.4 durchrechnen:

$$A = \left(\begin{array}{c|c|c|c|c|c|c|c|c} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}\right).$$

Es ist

$$\begin{aligned} N_0 &= C_1 \cup (1,2)C_1 \cup (1,3)C_1 \cup ... \cup (1,9)C_1, \\ N_1 &= C_2 \cup (2,3)C_2 \cup (2,4)C_2 \cup (2,5)C_2, \\ N_2 &= N_3 = C_4 \cup (4,5)C_4, \\ N_4 &= N_5 = C_6 \cup (6,7)C_6, \\ N_7 &= N_8 = \{id\}. \end{aligned}$$

Eine obere Grenze für die Anzahl der Zeilentests ist 9+36+36+72+72+144+144+144 = 657. Die genaue Analyse liefert folgende Tabelle:

Nebenklasse	Tiefe	Baumdurchlauf	kan. Perm.	lex. Vergleich	
(6,7)C_6	6	(6,7)	id	${a_6}^{(6,7)} = a_6$	
	7	(6,7)	id	${a_7}^{(6,7)} = a_7$	(A)
(4,5)C_4	4	(4,5)	(8,9)	${a_4}^{(4,5)(8,9)} = a_4$	
	5	(4,5)(8,9)	id	${a_5}^{(4,5)(8,9)} = a_5$	
	6	(4,5)(8,9)	id	${a_6}^{(4,5)(8,9)} = a_6$	
	7	(4,5)(8,9)	id	${a_7}^{(4,5)(8,9)} = a_7$	(A)
(2,3)C_2	2	(2,3)	id	${a_2}^{(2,3)} \succ a_2$	

Es ist ${a_2}^{(2,3)} = 101110000$. Wir erhalten $\pi = (2,3)$ und Kante $(2,4)$ als Eingabe für unseren Lerneffekt aus Abschnitt 3.5.

4.3.11 Bemerkung:
Der soweit beschriebene Kanonizitätstest kann für beliebige $\Gamma \in \mathcal{G}_n$ benutzt werden. Regularität wurde nur im Beweis von Lemma 4.3.3 verwendet. Eine allgemeinere Formulierung von 4.3.3, die Regularität nicht voraussetzt, finden wir in Satz 4.3.6.
Die folgende Aussage gilt nur für reguläre Kandidaten und erlaubt uns in manchen Fällen ganze Nebenklassen $(1,i)C_1$ beim Kanonizitätstest zu überspringen.

4.3.12 Lemma:
Sei $\Gamma \in \mathcal{R}_{n,k}$. Liegt Knoten i nicht auf einem Taillenkreis, dann gilt

$$\forall \pi \in (1,i)C_1 : \quad \Gamma^\pi > \Gamma.$$

Beweis:
Die Aussage folgt aus Satz 3.4.1. □

4.3.13 Bemerkung:
Die Nebenklasse $(1,i)C_1$ muß also beim Kanonizitätstest nur dann untersucht werden, wenn Knoten i auf einem Taillenkreis liegt. Für $k > 3$ liegen allerdings die meisten Knoten auf einem Taillenkreis, so daß nur im Falle kubischer Graphen eine wirksame Einsparung erzielt wird. Es müßte eine stärkere Aussage als Satz 3.4.1 gefunden werden, damit auf diese Weise die Effizienz des Kanonizitätstests gesteigert werden kann.

Kapitel 5

Resultate

Die vorgestellten Methoden wurden in dem C-Programm GENREG realisiert. GENREG wurde in erster Linie für UNIX Rechner konzipiert, ist aber auch unter anderen Betriebssystemen (DOS, OS/2) lauffähig. Es ist zu bemerken, daß kein nennenswerter Bedarf an Arbeitsspeicher besteht und das Programm somit auch für kleine Rechner geeignet ist.

Weiterhin wurde eine (nahezu identische) C++ Version implementiert, die als Unterprogramm von GRADPART reguläre Graphen mit ihren Automorphismengruppen erzeugt. Während für diese Anwendung auch Graphen mit mehreren Zusammenhangskomponenten berücksichtigt werden müssen, konstruiert GENREG nur die graphentheoretisch interessanteren zusammenhängenden regulären Graphen. GENREG bietet einige hilfreiche Optionen, die dem Benutzer den Umgang mit den erzeugten Strukturen erleichtern sollen. Ein kurzes Manual für GENREG findet man im Anhang A.

Außerdem wurde im Rahmen dieser Diplomarbeit eine Visualisierung von GENREG implementiert, welche den Erzeugungsalgorithmus graphisch in Form eines Baumes (vgl. Abschnitt 3.6) darstellt. Durch Verwendung farbiger Knotenpunkte kann der Benutzer feststellen, an welcher Stelle des Algorithmus welches Kriterium angewendet wird. Ferner können mehrere Knoten per Mausklick ausgewählt und die zugehörigen Graphen angezeigt werden. Auf diese Weise kann die Konstruktion eines regulären Graphen durch sukzessives Einsetzen einzelner Kanten optisch nachvollzogen werden.

Es sei noch erwähnt, daß Routinen existieren, die den von GENREG produzierten Code einlesen, die Graphen dekodieren und in EWS-Format überführen. Somit können EWS-Funktionen wie etwa zweidimensionale Plazierung oder Test auf Planarität angewendet werden. Ebenso gibt es eine Routine, die codierte Graphen in das MOLGEN-Format mbf transformiert und somit einen Zugriff auf den MOLGEN 3-D-Plazierer ermöglicht.

Die Korrektheit von GENREG wird durch Übereinstimmung mit den Resultaten aus [Bri] belegt (auch die produzierten Codes sind identisch). Weiterhin konnte Gunnar Brinkmann die Anzahl $|S_{18} \backslash\backslash \mathcal{R}^*_{18,4}|$ bestätigen. Die Tatsache, daß GADPART fehlerfrei arbeitet, ist insbesondere als Indiz für die Korrektheit der Automorphismengruppen zu werten.

Knoten	Grad 2	Grad 3	Grad 4	Grad 5	Grad 6	Grad 7
3	1	0	0	0	0	0
4	1	1	0	0	0	0
5	1	0	1	0	0	0
6	1	2	1	1	0	0
7	1	0	2	0	1	0
8	1	5	6	3	1	1
9	1	0	16	0	4	0
10	1	19	59	60	21	5
11	1	0	265	0	266	0
12	1	85	1544	7848	7849	1547
13	1	0	10778	0	367860	0
14	1	509	88168	3459383	21609300	21609301
15	1	0	805491	0	1470293675	0
16	1	4060	8037418	2585136675		
17	1	0	86221634	0		0
18	1	41301	985870522			
20	1	510489				
22	1	7319447				
24	1	117940535				

Tabelle 5.1: Anzahlen zusammenhängender regulärer Graphen.

5.1 Tabellen

Tabelle 5.1 gibt einen Überblick der bislang berechneten zusammenhängenden regulären Graphen. Bei den leeren Feldern handelt es sich um Fälle, die außerhalb der Reichweite von GENREG liegen. Dazu sei bemerkt, daß die Rate der Repräsentanten, die pro Zeiteiheit gefunden werden, immer noch relativ hoch liegt, aber die Anzahl der existierenden Graphen, welche bekanntlich mit der Knotenzahl exponentiell wächst, zu groß ist, um vollständige Listen zu konstruieren.

Genauere Angaben zu den einzelnen Programmläufen findet man in Tabelle 5.3. Dort ist insbesondere die Anzahl der Kandidaten für den Kanonizitätstest und das Verhältnis von Kandidaten zu Repräsentanten angegeben. Die letzte Spalte enthält jeweils die benötigte CPU-Zeit auf einer DEC-Alpha UNIX-Workstation.

GENREG bietet die Möglichkeit, eine untere Schranke für die Taillenweite vorzuschreiben. Tabellen 5.4 bis 5.8 enthalten Anzahlen zusammenhängender regulärer Graphen, die die jeweilige Forderung an die Taillenweite erfüllen. Diese Berechnungen wurden auf einer DEC-Station 5000/200 durchgeführt.

Mit einer kleinen Veränderung kann man das Programm zur Konstruktion bipartiter Graphen verwenden. Tabelle 5.9 enthält Anzahlen zusammenhängender bipartiter regulärer Graphen. Diese Option ist derzeit noch nicht für GENREG verfügbar.

Knoten	Grad 2	Grad 3	Grad 4	Grad 5	Grad 6	Grad 7
3	1	0	0	0	0	0
4	1	1	0	0	0	0
5	1	0	1	0	0	0
6	2	2	1	1	0	0
7	2	0	2	0	1	0
8	3	6	6	3	1	1
9	4	0	16	0	4	0
10	5	21	60	60	21	5
11	7	0	266	0	266	0
12	9	94	1547	7849	7849	1547
13	10	0	10786	0	367860	0
14	13	540	88193	3459386	21609301	21609301
15	17	0	805579	0	1470293676	0
16	21	4207	8037796	2585136741		
17	25	0	86223660	0		0
18	33	42110	985883873			
20	39	516344				
22	49	7373924				
24	110	118573592				

Tabelle 5.2: Anzahlen (nicht notwendig zusammenhängender) regulärer Graphen.

Zum Schluß wollen wir noch etwas elementare Kombinatorik betreiben, um zu sehen, wie man die Anzahl $|S_n \backslash\backslash \mathcal{R}_{n,k}|$ (nicht notwendig zusammenhängender) regulärer Graphen aus den Anzahlen zusammenhängender regulärer Graphen berechnen kann:
Zu einer kanonischen Partition $\lambda \vdash n$ mit $\lambda = (\lambda_1, ..., \lambda_l)$ bezeichnen $\mu_1, ..., \mu_m$ die Anzahlen gleicher Summanden in λ:

$$\underbrace{\lambda_1 = ... = \lambda_{\nu_2 - 1}}_{\mu_1 \text{ mal}} > \underbrace{\lambda_{\nu_2} = ... = \lambda_{\nu_3 - 1}}_{\mu_2 \text{ mal}} > \lambda_{\nu_3} ..... \lambda_{\nu_m - 1} > \underbrace{\lambda_{\nu_m} = ... = \lambda_l}_{\mu_m \text{ mal}}$$

Setzen wir $\nu_1 := 1$, dann gilt für $1 \leq i \leq m$: Summand λ_{ν_i} kommt μ_i mal in λ vor. Für die Anzahl regulärer Graphen ergibt sich die Formel:

$$|S_n \backslash\backslash \mathcal{R}_{n,k}| = \sum_{\lambda \vdash n} \prod_{i=1}^{m} \binom{\mu_i + |S_{\lambda_{\nu_i}} \backslash\backslash \mathcal{R}_{\lambda_{\nu_i},k}| - 1}{\mu_i}$$

Auf diese Weise kann Tabelle 5.2 zusammengestellt werden. Durch Verwendung der bekannten Bijektion zwischen k-regulären Graphen und $(n-k-1)$-regulären Graphen (ersetze Kanten durch Nichtkanten und umgekehrt) kann man die Tabelle für größere Grade ergänzen.

Knoten	Grad	Graphen	Kandidaten	Kand./Graph	Prozessorzeit
4	3	1	1	1,00	0,0 s
6	3	2	2	1,00	0,0 s
8	3	5	10	2,00	0,0 s
10	3	19	37	1,95	0,0 s
12	3	85	214	2,52	0,0 s
14	3	509	1406	2,76	0,4 s
16	3	4060	10432	2,57	3,2 s
18	3	41301	96279	2,33	33,3 s
20	3	510489	1079585	2,11	6 min 58 s
22	3	7319447	14341762	1,96	1 h 47 min
24	3	117940535	217873241	1,85	1 d 6 h 3 min
5	4	1	1	1,00	0,0 s
6	4	1	1	1,00	0,0 s
7	4	2	5	2,50	0,0 s
8	4	6	14	2,33	0,0 s
9	4	16	57	3,56	0,0 s
10	4	59	219	3,71	0,0 s
11	4	265	997	3,76	0,2 s
12	4	1544	5194	3,36	1,1 s
13	4	10778	33139	3,07	8,0 s
14	4	88168	251546	2,85	1 min 5 s
15	4	805491	2177590	2,70	9 min 42 s
16	4	8037418	20656320	2,57	1 h 41 min
17	4	86221634	212449363	2,46	19 h 14 min
18	4	985870522	2354685107	2,39	8 d 21 h 36 min
6	5	1	1	1,00	0,0 s
8	5	3	10	3,33	0,0 s
10	5	60	291	4,85	0,1 s
12	5	7848	24306	3,10	6,6 s
14	5	3459383	9503164	2,75	55 min 5 s
16	5	2585136675	6834826727	2,64	27 d 10 h 5 min
7	6	1	1	1,00	0,0 s
8	6	1	1	1,00	0,0 s
9	6	4	18	4,50	0,0 s
10	6	21	159	7,57	0,0 s
11	6	266	1407	5,29	0,3 s
12	6	7849	26416	3,37	8,9 s
13	6	367860	1018030	2,77	6 min 8 s
14	6	21609300	55550457	2,57	5 h 53 min
15	6	1470293675	3668827079	2,50	16 d 23 h 10 min

Tabelle 5.3: Zusammenhängende reguläre Graphen.

Knoten	Grad	Graphen	Prozessorzeit
6	3	1	0,0 s
8	3	2	0,0 s
10	3	6	0,0 s
12	3	22	0,1 s
14	3	110	0,6 s
16	3	792	4,6 s
18	3	7805	47,5 s
20	3	97546	10 min 3 s
22	3	1435720	2h 31 min
8	4	1	0,0 s
9	4	0	0,0 s
10	4	2	0,0 s
11	4	2	0,1 s
12	4	12	0,2 s
13	4	31	0,9 s
14	4	220	4,6 s
15	4	1606	30,8 s
16	4	16828	4 min 30 s
17	4	193900	46 min 51 s
18	4	2452818	8 h 56 min
10	5	1	0,0 s
12	5	1	0,1 s
14	5	7	1,9 s
16	5	388	3 min 56 s
18	5	406824	12 h 45 min
12	6	1	0,0 s
13	6	0	0,0 s
14	6	1	0,5 s
15	6	1	6,3 s
16	6	9	54,9 s
17	6	6	7 min 54 s
18	6	267	10 h 36 min
14	7	1	0,0 s
16	7	1	3,6 s
18	7	8	1 h 55 min

Tabelle 5.4: Zusammenhängende reguläre Graphen mit Taillenweite ≥ 4.

Knoten	Grad	Graphen	Prozessorzeit
10	3	1	0,0 s
12	3	2	0,0 s
14	3	9	0,1 s
16	3	49	1,0 s
18	3	455	11,5 s
20	3	5783	2 min 30 s
22	3	90938	40 min 2 s
24	3	1620479	12 h 41 min
17	4	0	0,0 s
18	4	0	0,0 s
19	4	1	0,4 s
20	4	2	2,3 s
21	4	8	29,1 s
22	4	131	6 min 16 s
23	4	3917	1 h 23 min
24	4	123859	21 h 12 min
26	5	0	0,0 s
28	5	0	16,1 s
30	5	4	8 h 13 min
50	7	1	5 min 17 s

Tabelle 5.5: Zusammenhängende reguläre Graphen mit Taillenweite ≥ 5.

Knoten	Grad	Graphen	Prozessorzeit
14	3	1	0,0 s
16	3	1	0,0 s
18	3	5	0,4 s
20	3	32	3,2 s
22	3	385	48,1 s
24	3	7574	15 min 11 s
26	3	181227	5 h 33 min
26	4	1	0,1 s
27	4	0	2,6 s
28	4	1	3 min 17 s
29	4	0	5 h 8 min
30	4	4	5 d 9 h 51 min
42	5	1	9,2 s
62	6	1	nicht gemessen

Tabelle 5.6: Zusammenhängende reguläre Graphen mit Taillenweite ≥ 6.

Knoten	Grad	Graphen	Prozessorzeit
22	3	0	0,0 s
24	3	1	0,5 s
26	3	3	4,8 s
28	3	21	1 min 7 s
30	3	546	23 min 49 s
32	3	30368	9 h 48 min

Tabelle 5.7: Zusammenhängende reguläre Graphen mit Taillenweite ≥ 7.

Knoten	Grad	Graphen	Prozessorzeit
30	3	1	0,1 s
32	3	0	9,6 s
34	3	1	7 min 40 s
36	3	3	1 h 20 min
38	3	13	10 h 54 min

Tabelle 5.8: Zusammenhängende reguläre Graphen mit Taillenweite ≥ 8.

Knoten	Grad 2	Grad 3	Grad 4	Grad 5
4	1	0	0	0
6	1	1	0	0
8	1	1	1	0
10	1	2	1	1
12	1	5	4	1
14	1	13	14	4
16	1	38	129	41
18	1	149	1980	1981
20	1	703	62611	304495
22	1	4132	2806490	
24	1	29579		
26	1	245627		
28	1	2291589		

Tabelle 5.9: Anzahlen zusammenhängender bipartiter regulärer Graphen.

5.2 Abbildungen

Auf den folgenden Seiten befinden sich einige Abbildungen interessnter regulärer Graphen, die mit GENREG erzeugt wurden. Es wurde darauf geachtet, die Plazierungen möglichst symmetrisch vorzunehmen. Die Vorgehensweise sei kurz skizziert:
Grundsätlich suchen wir nach Hamiltonkreisen. Bei kleinen Graphen (bis $n = 10$ Knoten) erhält man schon recht ansehnliche Plazierungen, wenn man die Knoten in der Reihenfolge, wie sie in einem Hamiltonkreis des Graphen durchlaufen werden, auf einem regelmäßigen n-Eck anordnet.
Bei größeren Graphen findet man allein mit dieser Methode keine brauchbaren Plazierungen mehr. Operiert die Automorphismengruppe auf der Menge der Knoten nicht transitiv, so suchen wir nach Hamiltonkreisen, in denen Knoten der gleichen Bahn mit konstantem Abstand folgen. Knoten der gleichen Bahn sollen somit symmetrisch über das regelmäßige n-Eck verteilt werden.
Bei transitiver Automorphismengruppe können wir nicht auf eine solche Aufteilung der Knotenmenge zurückgreifen. Allerdings kommt uns in diesen Fällen zugute, daß die Automorphismengruppe i.a. große Ordnung besitzt und sich der Graph „leichter“ symmetrisch plazieren läßt. In den betrachteten Fällen benügte es, nach Hamitonkreisen zu suchen, in denen gegenüberliegende Knoten nicht benachbart sein dürfen. Bei der Plazierung auf dem regelmäßigen n-Eck verlaufen dann keine Kanten durch die Mitte.

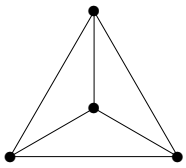


Abbildung 5.1: Der kubische Graph mit 4 Knoten.

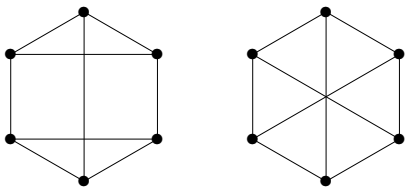


Abbildung 5.2: Die kubischen Graphen mit 6 Knoten.

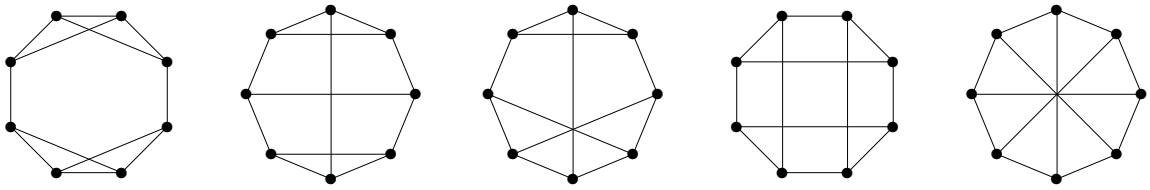


Abbildung 5.3: Die zusammenhängenden kubischen Graphen mit 8 Knoten.

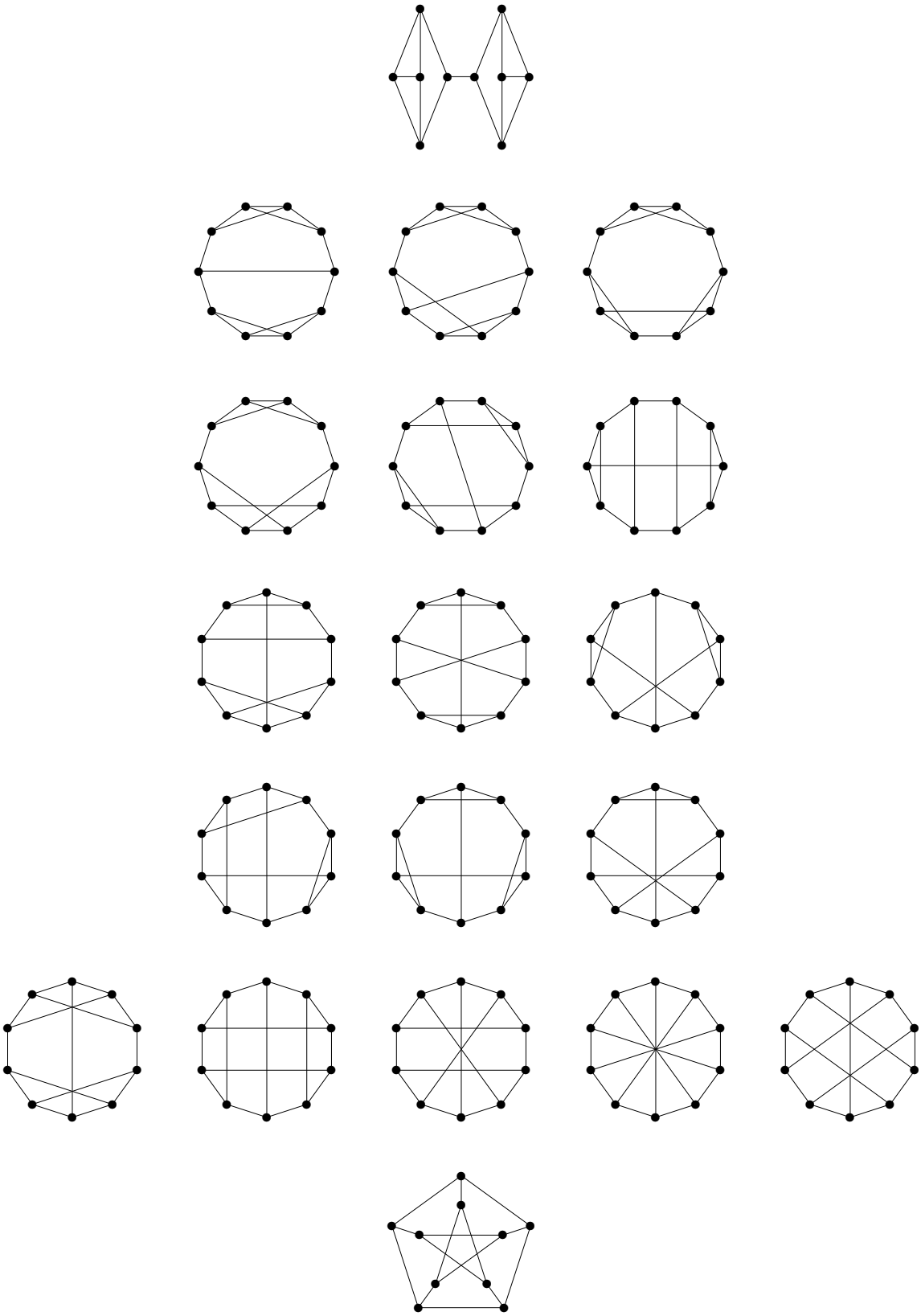


Abbildung 5.4: Die zusammenhängenden kubischen Graphen mit 10 Knoten.

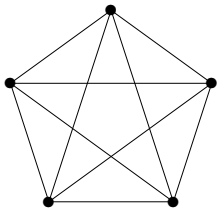


Abbildung 5.5: Der 4-reguläre Graph mit 5 Knoten.

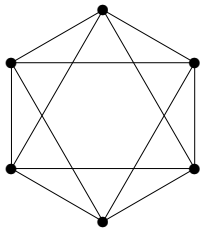


Abbildung 5.6: Der 4-reguläre Graph mit 6 Knoten.

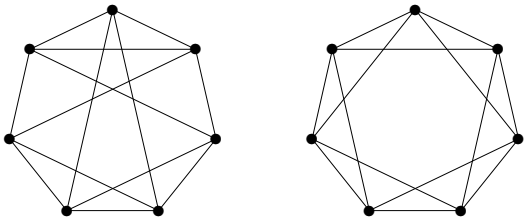


Abbildung 5.7: Die 4-regulären Graphen mit 7 Knoten.

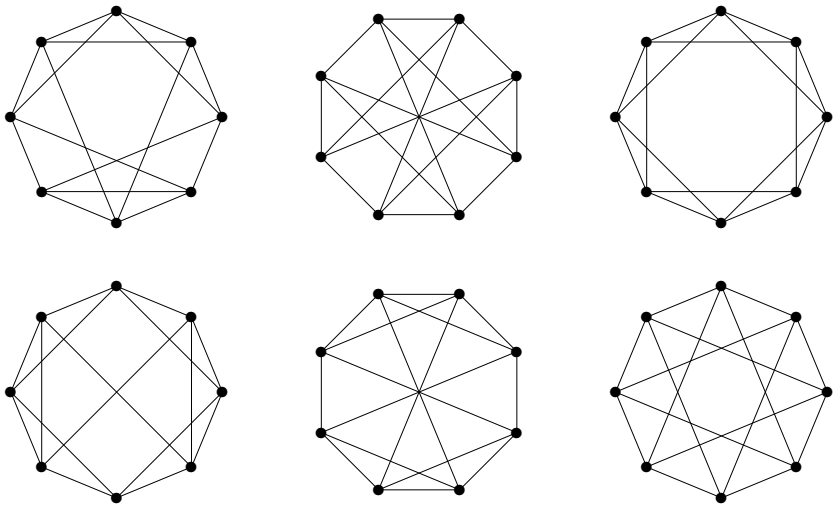


Abbildung 5.8: Die 4-regulären Graphen mit 8 Knoten.

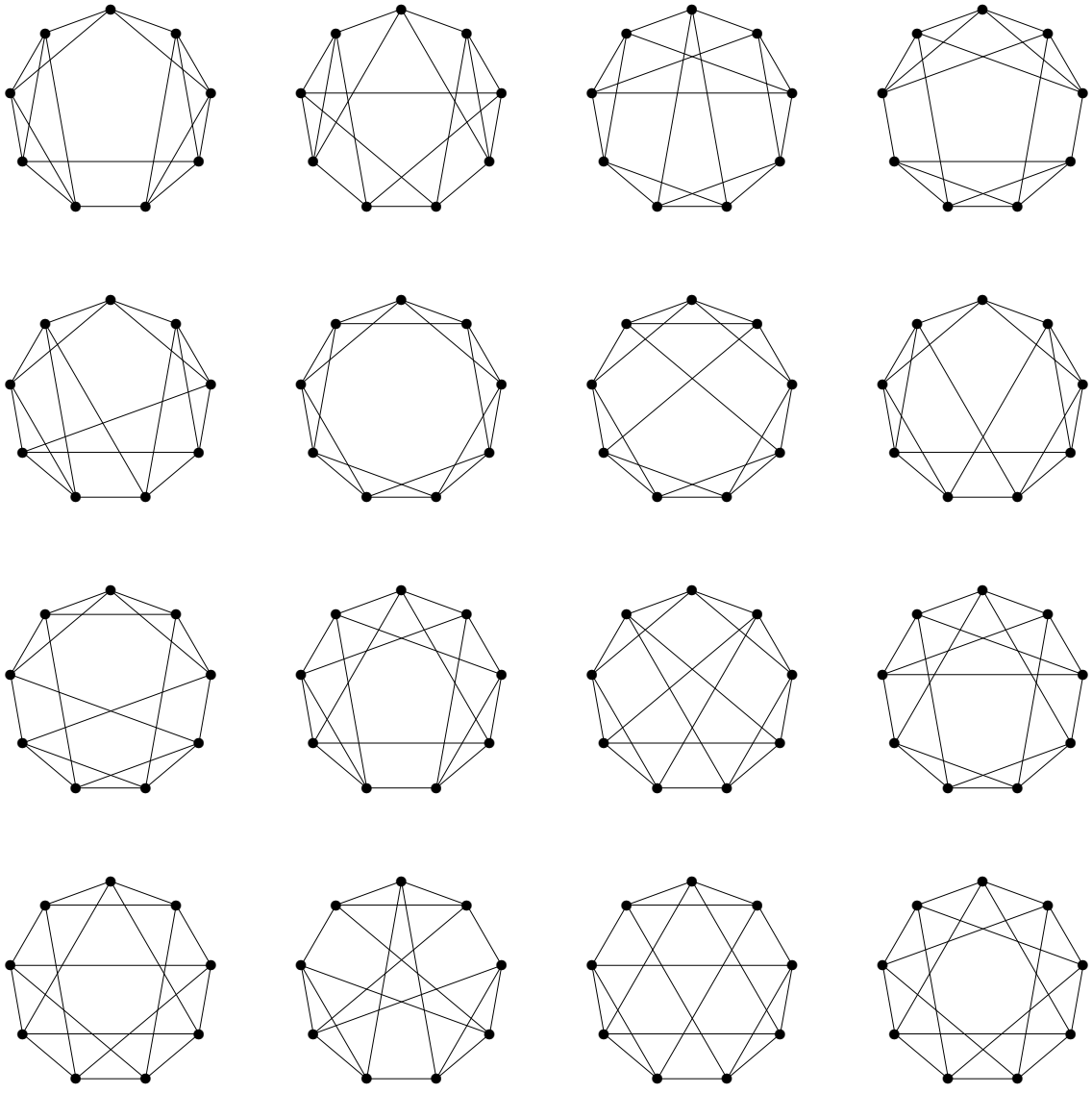


Abbildung 5.9: Die 4-regulären Graphen mit 9 Knoten.

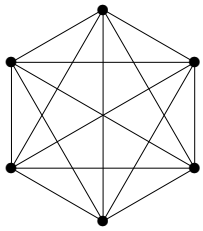


Abbildung 5.10: Der 5-reguläre Graph mit 6 Knoten.

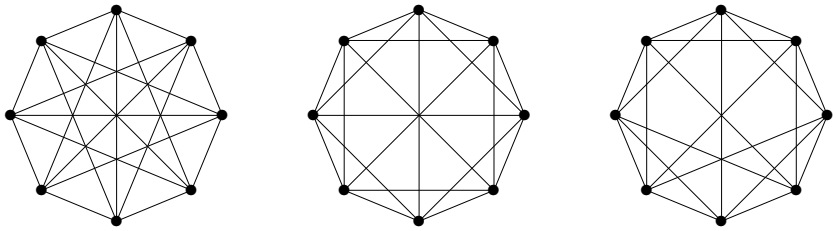


Abbildung 5.11: Die 5-regulären Graphen mit 8 Knoten.

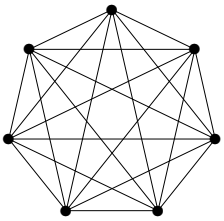


Abbildung 5.12: Der 6-reguläre Graph mit 7 Knoten.

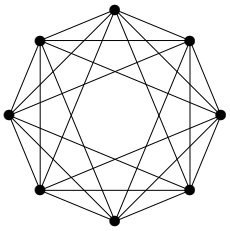


Abbildung 5.13: Der 6-reguläre Graph mit 8 Knoten.

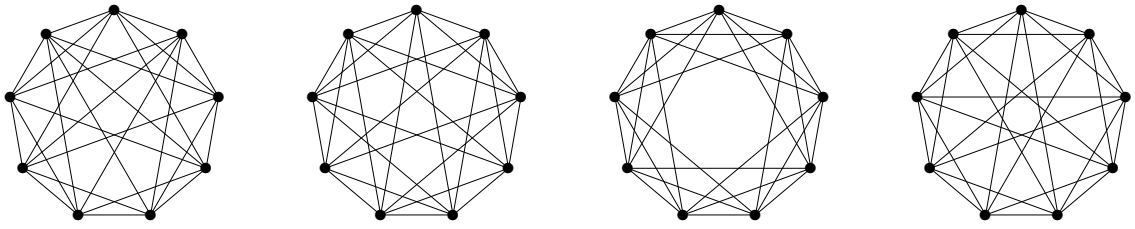


Abbildung 5.14: Die 6-regulären Graphen mit 9 Knoten.

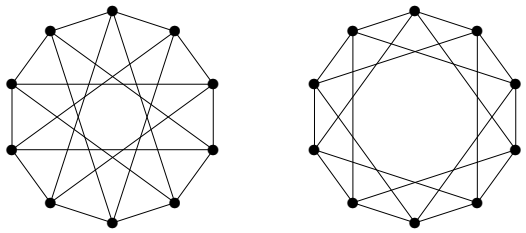


Abbildung 5.15: Die (4,4)-Graphen mit 10 Knoten.

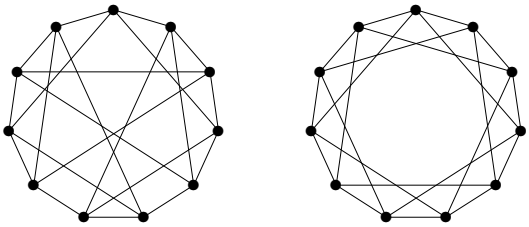


Abbildung 5.16: Die (4,4)-Graphen mit 11 Knoten.

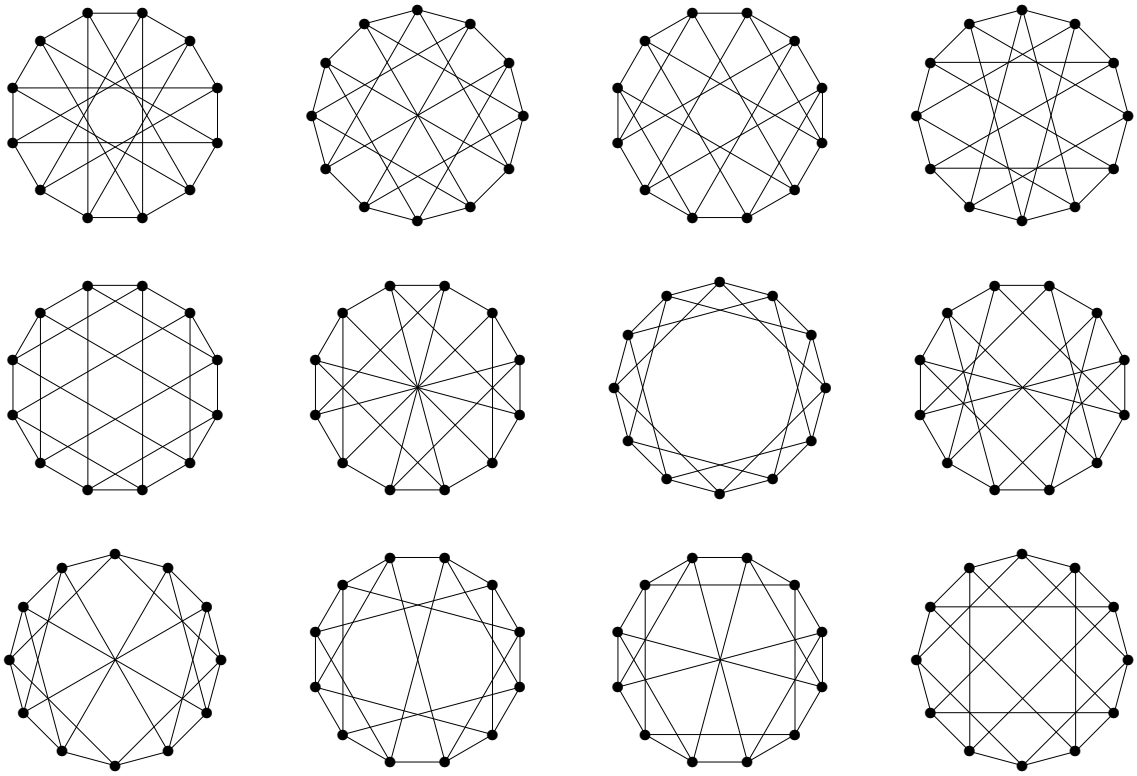


Abbildung 5.17: Die (4,4)-Graphen mit 12 Knoten.

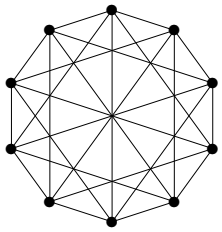


Abbildung 5.18: Der (5,4)-Graph mit 10 Knoten.

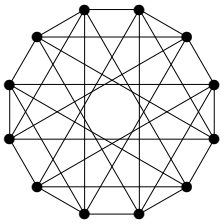


Abbildung 5.19: Der (5,4)-Graph mit 12 Knoten.

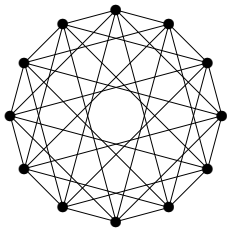


Abbildung 5.20: Der (6,4)-Graph mit 12 Knoten.

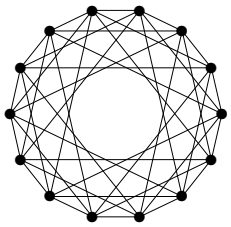


Abbildung 5.21: Der (6,4)-Graph mit 14 Knoten.

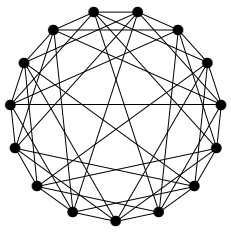


Abbildung 5.22: Der (6,4)-Graph mit 15 Knoten.

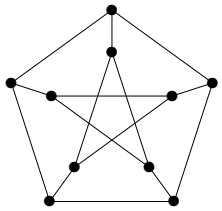


Abbildung 5.23: Der (3,5)-Cage hat 10 Knoten.

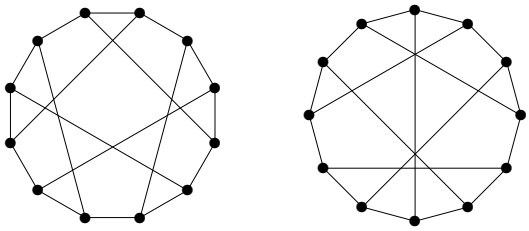


Abbildung 5.24: Die (3,5)-Graphen mit 12 Knoten.

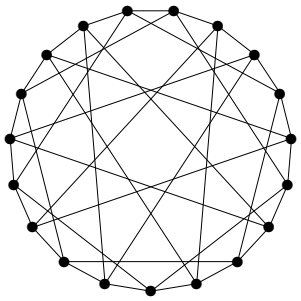


Abbildung 5.25: Der (4,5)-Cage hat 19 Knoten.

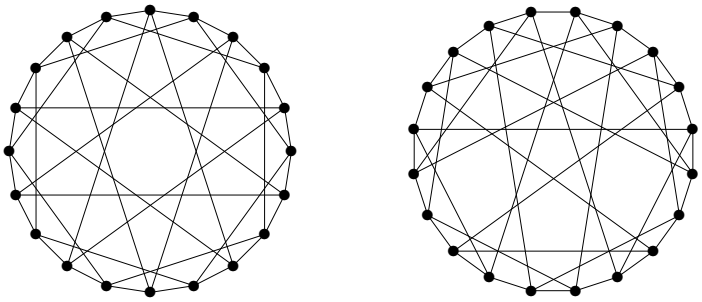


Abbildung 5.26: Die (4,5)-Graphen mit 20 Knoten.

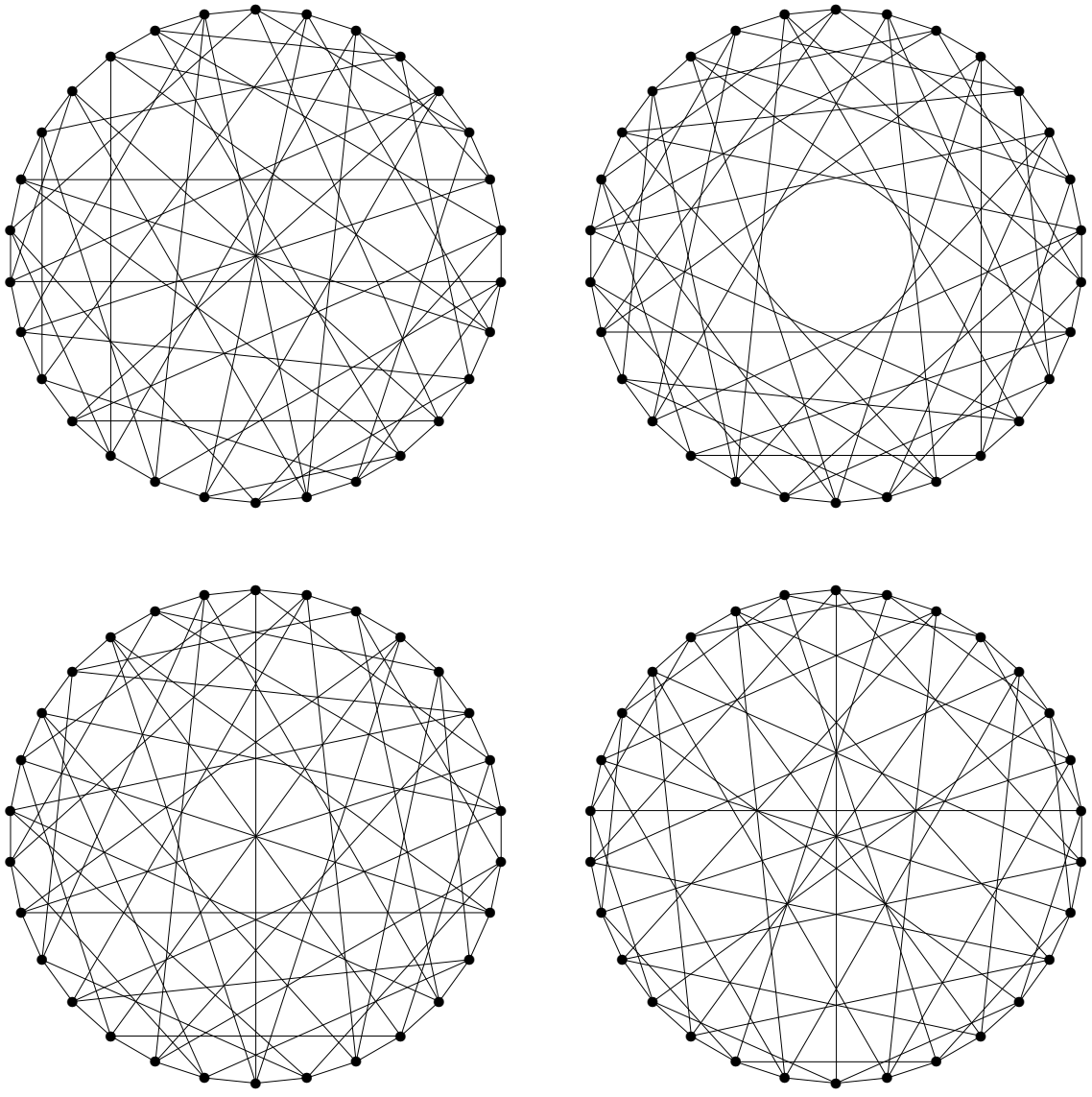


Abbildung 5.27: Die (5,5)-Cages haben 30 Knoten.

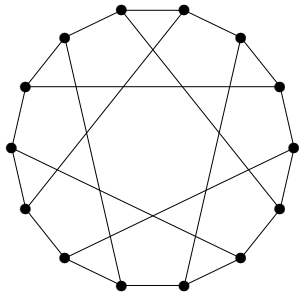


Abbildung 5.28: Der (3,6)-Cage hat 14 Knoten.

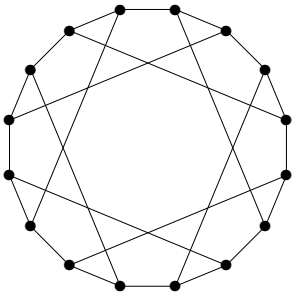


Abbildung 5.29: Der (3,6)-Graph mit 16 Knoten.

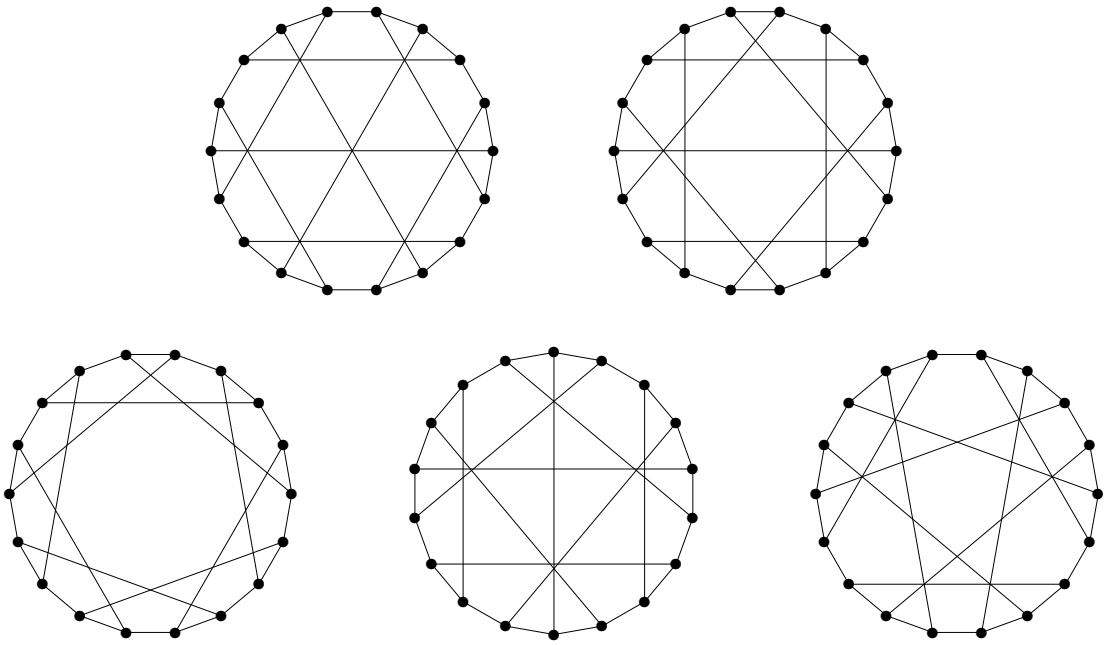


Abbildung 5.30: Die (3,6)-Graphen mit 18 Knoten.

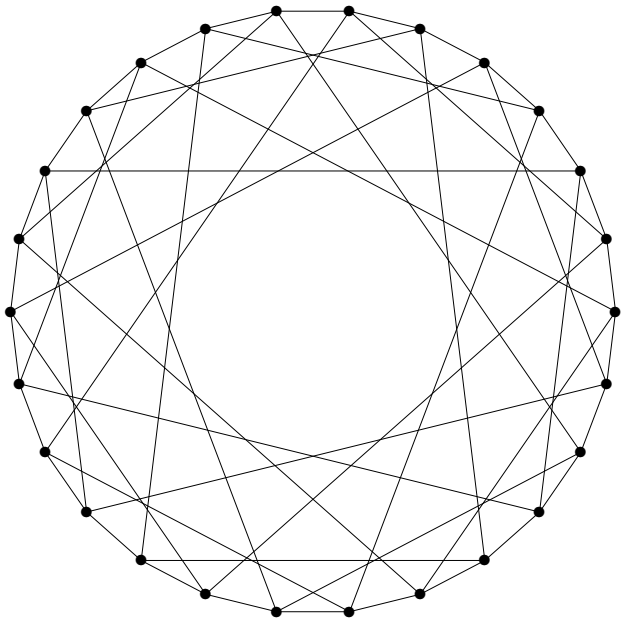


Abbildung 5.31: Der (4,6)-Cage hat 26 Knoten.

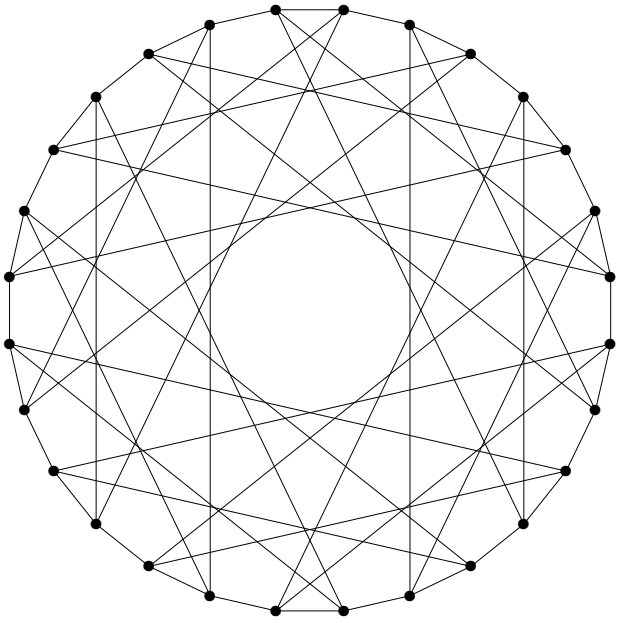


Abbildung 5.32: Der (4,6)-Graph mit 28 Knoten.

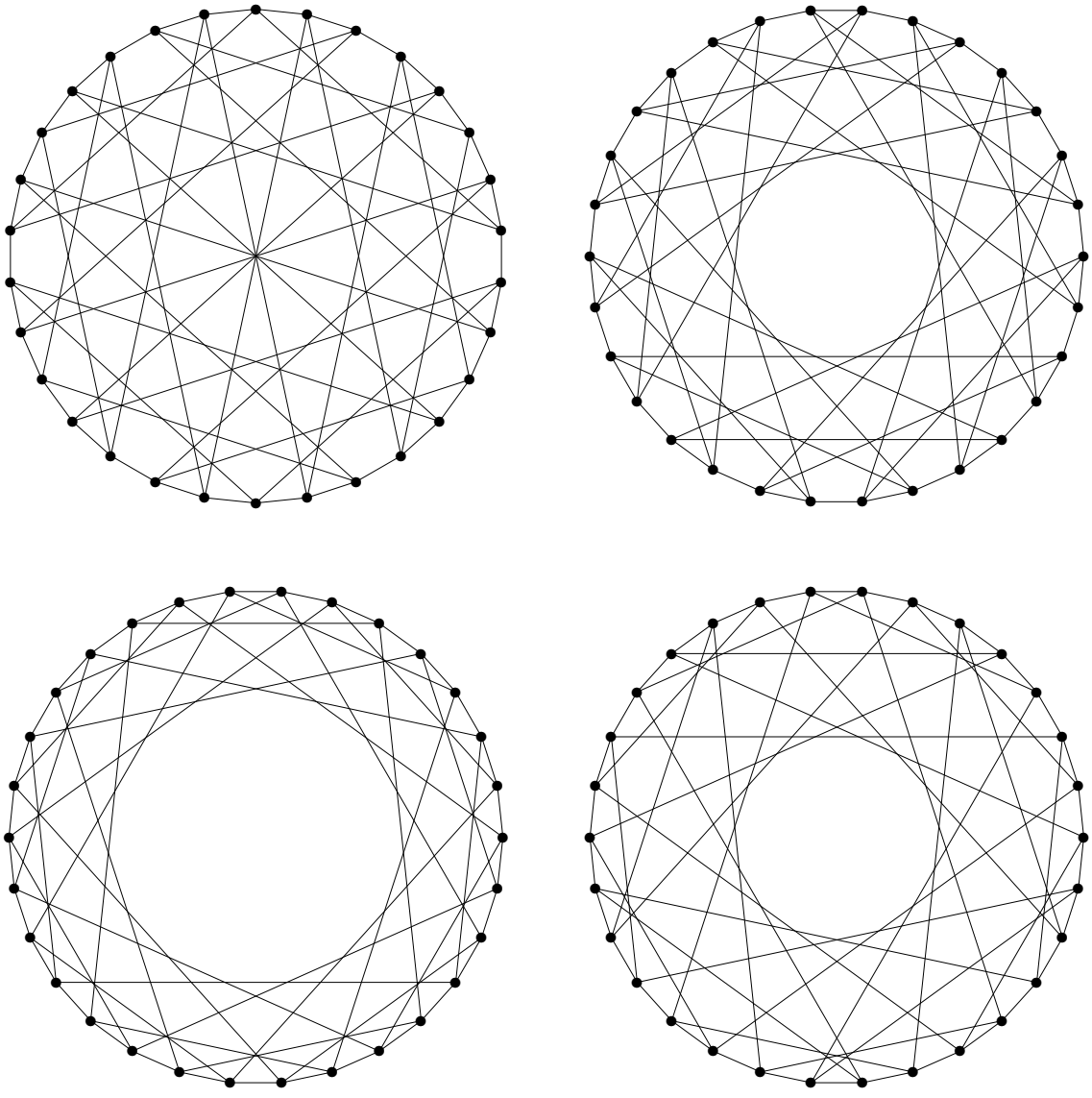


Abbildung 5.33: Die (4,6)-Graphen mit 30 Knoten.

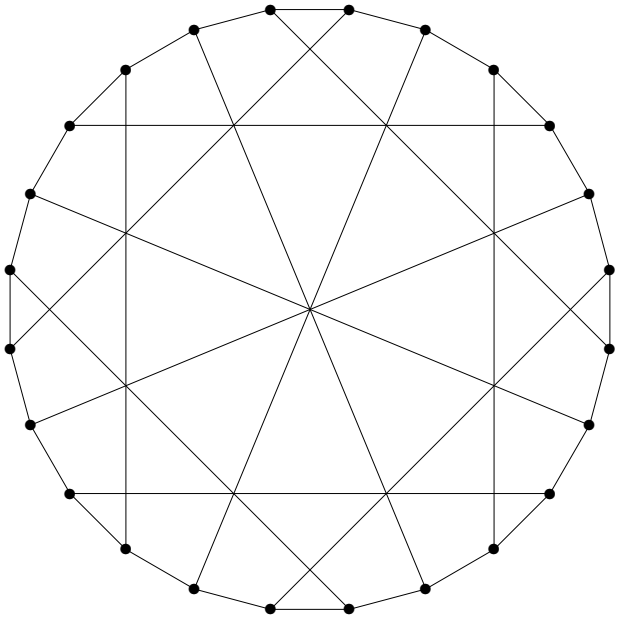


Abbildung 5.34: Der (3,7)-Cage hat 24 Knoten.

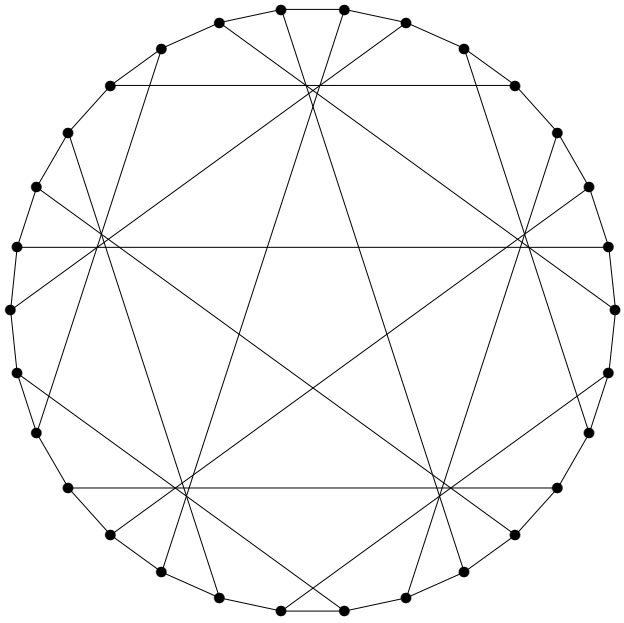


Abbildung 5.35: Der (3,8)-Cage hat 30 Knoten.

Anhang A

GENREG-Manual

Um die zusammenhängenden k-regulären Graphen mit n Knoten zu erzeugen, muß **genreg n k** aufgerufen werden. Um nur solche Graphen mit Taillenweite $\geq t$ zu erhalten, kann t als weiterer Parameter angefügt werden, also **genreg n k t**. Für die Steuerung der Ausgabe stehen folgende Optionen zur Verfügung:

-a Die Nachbarschaftslisten der erzeugten Graphen werden in ein ASCII-File mit Endung `.asc` geschrieben. Weiter wird die Taillenweite, ein Erzeugendensystem der Automorphismengruppe und deren Ordnung angegben. Beispielsweise wird bei Aufruf von **genreg 4 3 -a** das File `04_3_3.asc` mit folgendem Inhalt erzeugt:

```
Graph 1:

1 : 2 3 4
2 : 1 3 4
3 : 1 2 4
4 : 1 2 3
Taillenweite: 3

3 : 1 2 4 3
2 : 1 3 2 4
2 : 1 4 3 2
1 : 2 1 3 4
1 : 3 2 1 4
1 : 4 2 3 1
Ordnung: 24
```

Zuerst wird die Nachbarschaftsliste ausgegeben. In jeder Zeile stehen nach dem Doppelpunkt in aufsteigender Reihenfolge die Nachbarn des Knoten vor dem Doppelpunkt.

Der Nachbarschaftsliste folgt die Taillenweite und danach in jeder Zeile ein Automorphismus π. Vor dem Doppelpunkt steht jeweils $\min\{j \in \underline{n} \mid \pi(j) \neq j\}$ und nach dem Doppelpunkt steht an i-ter Position $\pi(i)$. Es handelt sich dabei um Transversalenelemente der Linksnebenklassen einer Zentralisatorkette der Automorphismengruppe (Simskette).

Gibt man zusätzlich eine Zahl x an, so werden nur die ersten x Graphen gespeichert, und dann das File geschlossen. Da solche Files u.U. nicht alle Graphen zu gegebenen n,k und t enthalten, werden deren Filenamen mit einem `-U` (für „unvollständig“) gekennzeichnet. **genreg 6 3 -a 1** liefert also das File `06_3_3-U.asc`.

Wird die Option gefolgt von **stdout**, dann wird kein File erzeugt, sondern auf die Standardausgabe geschrieben.

-s Die erzeugten Graphen werden in ein binäres File mit Endung `.scd` geschrieben. Dabei wird folgende Kodierung (Shortcode) verwendet:

Es werden nacheinander Knoten 1 bis n betrachtet. Nur benachbarte Knoten mit größerer Nummer werden aufgeschrieben. Somit wird für jede Kante genau ein Eintrag vorgenommen. Im obigen Beispiel ($n = 4$, $k = 3$) erhält man den Code

```
2 3 4 3 4 4.
```

Um bei einer großen Anzahl von Graphen die anfallende Datenmenge weiter zu komprimieren, vergleicht man den Code des nächsten erzeugten Graphen mit dem vorhergehenden und stellt fest, bis zu welcher Stelle die beiden Codes identisch sind. Anstatt nun dieses gemeinsame Anfangsstück wiederholt abzuspeichern, wird lediglich dessen Länge und anschließend das sich unterscheidende Endstück auf File geschrieben. Dem Code des ersten Graphen wird somit eine Null vorangestellt. Die 4-regulären Graphen mit 7 Knoten haben die Codes

```
2 3 4 5 3 4 5 6 7 6 7 6 7 7 und
```

```
2 3 4 5 3 4 6 5 7 6 7 6 7 7
```

File `07_4_3.scd` hat den Inhalt

```
0 2 3 4 5 3 4 5 6 7 6 7 6 7 7 6 6 5 7 6 7 6 7 7
```

Diese Art der Komprimierung gewinnt vor allem bei großem n oder $t > 3$ an Effektivität. Das C-Programm **readscd.c** enthält Routinen, die Shortcode-Files lesen und die Nachbarschaftslisten der gespeicherten Graphen ausgeben.

Auch diese Option kann von einer Zahl bzw. **stdout** gefolgt werden, falls nur eine bestimmte Anzahl von Graphen gespeichert, bzw. die Standardausgabe benutzt werden soll.

-e Die Parameter n,k und t, die Anzahl der erzeugten Graphen und die benötigte CPU-Zeit werden in ein File mit Endung `.erg` geschrieben. **genreg 21 4 5 -e** liefert das File `21_4_5.erg` mit dem Inhalt

```
GENREG - Generator fuer regulaere Graphen
21 Knoten, Grad 4, Taillenweite mind. 5
Erzeugung gestartet...
8 Graphen erzeugt.
Laufzeit:20.9s  0.4 Repr./s
```

Solange die Erzeugung noch nicht abgeschlossen ist, trägt dieses File den Namen `21_4_5-U.erg` (und natürlich fehlen die beiden letzten Zeilen).

Wird die Option nicht verwendet, dann schreibt GENREG diese Angaben auf die Standardfehlerausgabe.

-c Während des Programmlaufs wird die Anzahl der bereits erzeugten Graphen ausgegeben. Je nachdem, ob Option **-e** benutzt wird, schreibt GENREG diese Angabe in das File mit Endung `.erg` oder auf die Standardfehlerausgabe.

Für den Fall, daß sehr große Probleme bearbeitet werden sollen und mehrere Rechner zur Verfügung stehen, kann folgende „modulo“-Option verwendet werden:

-m Diese Option muß von zwei Zahlen i und j gefolgt werden, und dient dazu die Erzeugung in j Teile aufzuspalten. Aufruf von **genreg 20 3 -s -m 1 2** bewirkt, daß der erste von zwei Teilen abgearbeitet wird. Die Codes der Graphen werden in ein File mit Namen `20_3_3-1.scd` geschrieben. Sind alle Graphen gespeichert, wird es umbenannt in `20_3_3#1.scd`. Die übrigen Graphen erhält man mit **genreg 20 3 -s -m 2 2**.

Anmerkungen:

- Damit GENREG auch unter verschiedenen Betriebssystemen lauffähig bleibt, wurden alle Filenamen so gewählt, daß ihre Länge acht Zeichen nicht überschreitet. **genreg 20 3 -s -m 1 2** und **genreg 20 3 -s -m 1 3** produzieren Files mit gleichen Namen, aber verschiedenem Inhalt.
- Solange die Anzahl j der Teile nicht zu groß wird, sollten die einzelnen Teile etwa gleich viel Rechnerzeit benötigen und gleich viele Graphen erzeugen. Bei sehr großen Problemen, insbesondere mit Taillenweite > 3, kann es vorkommen, daß dies nicht mehr gewährleistet ist. In solchen Fällen sollte man sich mit dem Autor in Verbindung setzen (markus@btm2x2mat.uni-bayreuth.de).

Anhang B

Quelltext

```
/* Diese Version: 2.November'95 */


#include "gendef.h"


static SCHAR n,k,t,mid_max,splitlevel;
static ULONG tostore,toprint,count;
static FILE  *ergfile,*lstfile,*autfile;
static UINT  jobs,jobnr;
static SCHAR storeall,printall;

static SCHAR springen,i1,j1,f0,f1,girth_exact;
static SCHAR **g,**l,**part,**einsen,**transp,**zbk;
static SCHAR *kmn,*grad,*lgrad,*lastcode;
static ULONG calls,dez,anz;

static long fpos;

/*
 mit Maxtest bei mid_max
 ACHTUNG: vm geaendert : vm=n+1 falls grad[n]>=1

 letzte Aenderung:
 statt if(vm<=n&&...)vm++ jetzt if(vm<=my)vm=my+1

 Aufteilung der Erzeugung in mehrere
 jetzt Schritte moeglich
```

```
 Shortcode implementiert
*/

#ifdef STAB
static SCHAR **aut;
static int  erz,merz;

void stabprint()
{
 int i,j;
 long ord=1;
 SCHAR next,last=0,mult=1;

 for(i=1;i<=erz;i++)
  {
   next=aut[i][0];
   if(last==next)
    mult++;
   else
    {
     last=next;
     ord*=mult;
     mult=2;
    }
   fprintf(autfile,"\n%d :",next);
   for(j=1;j<=n;j++)
    fprintf(autfile," %d",aut[i][j]);
  }
 fprintf(autfile,"\nOrdnung: %ld\n",ord*mult);
}
#endif

void err()
{
 fprintf(stderr,"Abbruchfehler: Speicher voll");
 exit(0);
}

void nachblist()
{
 SCHAR i,j;
 for(i=1;i<=n;i++)
  {
```

```
   fprintf(autfile,"\n%d : ",i);
   for(j=1;j<=grad[i];j++)
    fprintf(autfile,"%d ",l[i][j]);
  }
 fprintf(autfile,"\n");
}

/*
 (f0)f1 ist kleinster Knoten fuer
 den aufgrund der Taillenweite
 und des Grades (k)eine Kante
 fest vorgegeben ist

 Bsp.:
 k=3,t=3 -> f1=2,f0=5
 k=3,t=5 -> f1=5,f0=11
*/

#ifndef SHORTCODE

void komprtofile()
{
 SCHAR i,j,c;
 SCHAR *store;
 store=g[0];

 for(i=f1;i<f0;i++)
  store[i]=2;
 for(i=f0;i<=n;i++)
  store[i]=1;

 for(i=f1;i<=n;i++)
  for(j=store[i];j<=k;j++)
   {
    c=l[i][j];
    store[c]++;
    putc(c,lstfile);
   }
}

#else

void komprtofile()
```

```
{
 SCHAR i,j,c,equ=1;
 UINT  h=0;

 for(i=1;i<=n;i++)
  for(j=1;j<=k;j++)
   {
    c=l[i][j];
    if(c>i)
     {
      if(equ&&c!=lastcode[++h])
       {
        equ=0;
        putc(h-1,lstfile);  /*Anzahl der uebereinstimmenden Kanten*/
        lastcode[h]=c;
        putc(c,lstfile);
       }
      else
      if(!equ)
       {
        lastcode[++h]=c;
        putc(c,lstfile);
       }
     }
   }
}

#endif

/*
 girthstart ermittelt Taillenweite
 und gibt diese zurueck bzw 0,
 falls Knoten 3 nicht auf dem
 (ersten Taillenkreis liegt
 wird nur aufgerufen, wenn
 1.Kreis soeben geschlossen wurde
*/

SCHAR girthstart()
{
 SCHAR tw=2,last=1,now=2,next;

 while(now!=3)
```

```
  {
   next=l[now][1];
   if(next==last)
    next=l[now][2];
   if(next==0||next==2);
    return(0);
    last=now;now=next;tw++;
  }
 return(tw);
}

/*
 girthcheck2 testet,ob sich die
 Taillenweite nach Einfuegen
 von Kante(mx,my) verkleinert,
 wenn ja wird weue tw zurueckgegeben
 (d.h. G nicht max), sonst 127
*/

SCHAR girthcheck2(mx,my,vm,tw)
SCHAR mx,my,vm,tw;
{
 SCHAR *status,*xnachb,*ynachb;
 SCHAR welle=1,a,i,j,x,y;
 xnachb=l[mx];ynachb=l[my];

 if(tw==4)
  {
   for(i=1;i<grad[mx];i++)
    {
     x=xnachb[i];
     for(j=1;j<grad[my];j++)
      if(x==ynachb[j])
       return(3);
    }
   return(127);
  }

 if(tw==5)
  {
   for(i=1;i<grad[mx];i++)
    {
     x=xnachb[i];
```

```
    for(j=1;j<grad[my];j++)
     {
      y=ynachb[j];
      if(x==y)
       return(3);
      if(g[x][y])
       return(4);
     }
   }
  return(127);
 }

status=zbk[0];
for(i=1;i<=n;i++)status[i]=0;
status[mx]=1;status[my]=2;
zbk[2][1]=my;zbk[2][0]=1;
for(i=1;i<grad[mx];i++)
 {
  x=xnachb[i];
  zbk[3][i]=x;
  status[x]=3;
 }
zbk[3][0]=grad[mx]-1;
while(++welle<tw-1)
 {
  a=0;
  for(i=1;i<=zbk[welle][0];i++)
   {
    x=zbk[welle][i];
    for(j=1;j<=grad[x];j++)
     {
      y=l[x][j];
      if(status[y]==0)
       {
        status[y]=welle+2;
        zbk[welle+2][++a]=y;
       }
      else
      if(status[y]==welle+1)
       return(welle+1);
     }
   }
  zbk[welle+2][0]=a;
```

```
  }
 return(127);
}

/*
 ongirth0 prueft fuer gerade Taillenweite tw,
 ob Knoten v auf einem Taillenkreis liegt.
 Wenn ja, wird 1 zurueckgegben, sonst 0.
*/

SCHAR ongirth0(v,tw)
SCHAR v,tw;
{
 SCHAR *status,next,first,last,a,h,i,j=0,x,y;
 status=zbk[0];
 for(i=1;i<=n;i++)status[i]=0;
 status[v]=-1;
 for(h=1;h<=k;h++)
  {
   last=tw/2+h-2;
   x=l[v][h];status[x]=h;
   zbk[last][1]=x;zbk[last][0]=1;
   while(last>h)
    {
     a=0;
     for(i=1;i<=zbk[last][0];i++)
      {
       x=zbk[last][i];
       for(j=1;j<=k;j++)
        {
         y=l[x][j];
         if(status[y]==0)
          {
           status[y]=h;
           zbk[last-1][++a]=y;
          }
        }
      }
     zbk[--last][0]=a;
    }
  }
 for(h=1;h<k;h++)
  {
```

```
   for(i=1;i<=zbk[h][0];i++)
    {
     x=zbk[h][i];
     for(j=1;j<=k;j++)
      {
       y=l[x][j];
       if(status[y]>=0)
        status[y]=h;
      }
    }
   for(last=h+1;last<=k;last++)
    {
     for(i=1;i<=zbk[last][0];i++)
      {
       x=zbk[last][i];j=0;
       for(j=1;j<=k;j++)
        {
         y=l[x][j];
         if(status[y]==h)
          return(1);
        }
      }
    }
  }
 return(0);
}

/*
 ongirth1 prueft fuer ungerade Taillenweite tw,
 ob Knoten v auf einem Taillenkreis liegt.
 Wenn ja, wird 1 zurueckgegben, sonst 0.
*/

SCHAR ongirth1(v,tw)
SCHAR v,tw;
{
 SCHAR *status,next,first,last,a,h,i,j=0,x,y;
 status=zbk[0];
 for(i=1;i<=n;i++)status[i]=0;
 status[v]=-1;
 for(h=1;h<=k;h++)
  {
   last=tw/2+h-1;
```

```
   x=l[v][h];status[x]=h;
   zbk[last][1]=x;zbk[last][0]=1;
   while(last>h)
    {
     a=0;
     for(i=1;i<=zbk[last][0];i++)
      {
       x=zbk[last][i];
       for(j=1;j<=k;j++)
        {
         y=l[x][j];
         if(status[y]==0)
          {
           status[y]=h;
           zbk[last-1][++a]=y;
          }
        }
      }
     zbk[--last][0]=a;
    }
   if(h>1)
    {
     for(i=1;i<=zbk[h][0];i++)
      {
       x=zbk[h][i];
       for(j=1;j<=k;j++)
        {
         y=l[x][j];
         if(status[y]>0)
          return(1);
        }
      }
    }
  }
 return(0);
}

SCHAR knoten_als_eins(v,tw)
SCHAR v,tw;
{
 SCHAR i,j,x,*vnachb,*zeile;
 if(tw==3)
  {
```

```
   for(i=1;i<=k;i++)
    {
     x=l[v][i];
     for(j=i+1;j<=k;j++)
      if(g[x][l[v][j]])
       return(1);
    }
   return(0);
  }
 else
 if(tw%2==0)
  return(ongirth0(v,tw));
 else
  return(ongirth1(v,tw));
}

void sprungindex(i,j)
SCHAR i,j;
{
 SCHAR r,s,x,y,z,e;
 i1=1;j1=j;
 for(r=1;r<=i;r++)
  {
   e=lgrad[r];
   for(s=r+1;e<k&&(r<i||s<j);s++)
    {
     z=g[r][s];e+=z;
     if(z)
      {
       x=kmn[r];y=kmn[s];
       if(x>y)
        {z=x;x=y;y=z;}
       if(x>i1||(x==i1&&y>j1))
        {i1=x;j1=y;}
      }
    }
  }
 x=kmn[i];y=kmn[j];
 if(x>y)
  {z=x;x=y;y=z;}
 if(x>i1||(x==i1&&y>j1))
  {i1=x;j1=y;}
}
```

```
void transpinv(mperm)
SCHAR *mperm;
{
 SCHAR i,x,re,li;
 i=2*mperm[0];
 while(i>0)
  {
   re=mperm[i--];
   li=mperm[i--];
   x=kmn[re];
   kmn[re]=kmn[li];
   kmn[li]=x;
  }
}

SCHAR maxinblock(zeile,mperm,e,li,re)
SCHAR *zeile,*mperm;
SCHAR e,li,re;
{
 SCHAR i,x;
 i=2*mperm[0];
 while(e>=0&&li<=re)
  {
   while(zeile[kmn[li]])
    {
     e--;
     if(++li>re)
      return(e);
    }
   if(li==re)
    return(e);
   while(zeile[kmn[re]]==0)
    if(--re==li)
     return(e);
   mperm[++i]=li;
   mperm[++i]=re;
   mperm[0]++;
   x=kmn[li];
   kmn[li]=kmn[re];
   kmn[re]=x;
   li++;re--;e--;
  }
```

```
 return(e);
}

SCHAR maxinzeile(tz)
SCHAR tz;
{
 SCHAR b,*zeile,*block,*eintr,*mperm;
 SCHAR e,li,re,erg=0;
 zeile=g[kmn[tz]];
 block=part[tz];
 eintr=einsen[tz];
 mperm=transp[tz];
 mperm[0]=0;

 for(b=1;b<=block[0]&&erg==0;b++)
  {
   e=eintr[b];
   li=block[b];re=block[b+1]-1;
   erg=maxinblock(zeile,mperm,e,li,re);
  }
 if(erg==0)
  return(0);
 else
 if(erg>0)
  return(1);   /*urspr. Nummerierung groesser*/
 sprungindex(tz,li+e);
 return(-1);
}

SCHAR maxrekneu(tz)
SCHAR tz;
{
 SCHAR i,j,x,e;
 SCHAR erg;
#ifdef STAB
 SCHAR *aut1;
 if(tz>=n-1)
  {
   aut1=aut[++erz];
   for(i=1;i<=n;i++)
    aut1[i]=kmn[i];
   return(0);
  }
```

```
#else
 if(tz>=n-1)
  return(0);
#endif
 erg=maxinzeile(tz);
 if(erg==0)
  erg=maxrekneu(tz+1);
 transpinv(transp[tz]);
 e=part[0][tz];
 for(i=tz+1;i<=e&&erg==1;i++)
  {
   x=kmn[tz];
   kmn[tz]=kmn[i];
   kmn[i]=x;
   erg=maxinzeile(tz);
   if(erg==0)
    erg=maxrekneu(tz+1);
   transpinv(transp[tz]);
   x=kmn[tz];
   kmn[tz]=kmn[i];
   kmn[i]=x;
  }
 return(erg);
}

SCHAR maxstartneu(tw)
SCHAR tw;
{
 SCHAR i,j,x,e;
 SCHAR erg;
#ifdef STAB
 erz=0;
#endif
 for(i=n-1;i>1;i--)       /*durchlauft die punktw. Stabilisatoren*/
  {                       /*Stab(n-2),...,Stab(0)=S(n)*/
   e=part[0][i];
   for(j=i+1;j<=e;j++)
    {
     kmn[i]=j;
     kmn[j]=i;            /*Transposition (i,j)*/
     erg=maxinzeile(i);
     if(erg==0)
      erg=maxrekneu(i+1); /*durchlaeuft Nebenklasse (i,j)Stab{0,...,i}*/
```

```
#ifdef STAB
     if(erg==0)
      aut[erz][0]=i;
#endif
     transpinv(transp[i]);
     kmn[i]=i;
     kmn[j]=j;
     if(erg==-1)
      return(0);
    }
  }
 for(j=2;j<=n;j++)          /*durchlauft die Nebenklassen von Stab(1)*/
  if(knoten_als_eins(j,tw))
   {
    kmn[1]=j;
    kmn[j]=1;               /*Transposition (i,j)*/
    erg=maxinzeile(1);
    if(erg==0)
     erg=maxrekneu(2);      /*durchlaeuft Nebenklasse (i,j)Stab{0,...,i}*/
#ifdef STAB
    if(erg==0)
     aut[erz][0]=i;
#endif
    transpinv(transp[1]);
    kmn[1]=1;
    kmn[j]=j;
    if(erg==-1)
     return(0);
   }
 return(1);
}

SCHAR maxinzeile1(tz)
SCHAR tz;
{
 SCHAR e,li,re,erg=0;
 SCHAR b,*zeile,*block,*eintr,*mperm;
 zeile=g[kmn[tz]];
 block=part[tz];
 eintr=einsen[tz];
 mperm=transp[tz];
 mperm[0]=0;
 for(b=1;b<=block[0]&&erg==0;b++)
```

```
  {
   e=eintr[b];
   li=block[b];re=block[b+1]-1;
   erg=maxinblock(zeile,mperm,e,li,re);
  }
 if(erg==0)
  return(0);
 else
 if(erg>0)
  return(1);
 return(-1);
}

SCHAR maxrekneu1(tz)
SCHAR tz;
{
 SCHAR i,j,x,e;
 SCHAR erg;
 if(tz==n-1)
  return(0);
 erg=maxinzeile1(tz);
 if(erg==0)
  erg=maxrekneu1(tz+1);
 transpinv(transp[tz]);
 e=part[0][tz];
 for(i=tz+1;i<=e&&erg==1;i++)
  {
   x=kmn[tz];
   kmn[tz]=kmn[i];
   kmn[i]=x;
   erg=maxinzeile1(tz);
   if(erg==0)
    erg=maxrekneu1(tz+1);
   transpinv(transp[tz]);
   x=kmn[tz];
   kmn[tz]=kmn[i];
   kmn[i]=x;
  }
 return(erg);
}

SCHAR maxstartneu1(vm)
SCHAR vm;
```

```
{
 SCHAR i,j,x,e;
 SCHAR erg;
 for(i=vm;i>1;i--)
  {
   e=part[0][i];
   for(j=i+1;j<=e;j++)
    {
     kmn[i]=j;
     kmn[j]=i;
     erg=maxinzeile1(i);
     if(erg==0&&i<n-1)
      erg=maxrekneu1(i+1);
     transpinv(transp[i]);
     kmn[i]=i;
     kmn[j]=j;
     if(erg==-1)
      return(0);
    }
  }
 for(j=2;j<=vm;j++)
  if(grad[j]==k)
   {
    kmn[1]=j;
    kmn[j]=1;
    erg=maxinzeile1(1);
    if(erg==0)
     erg=maxrekneu1(2);
    transpinv(transp[1]);
    kmn[1]=1;
    kmn[j]=j;
    if(erg==-1)
     return(0);
   }
 return(1);
}

/*
 semiverf erstellt part[x+1],
 die Verfeinerung von part[x]
 aufgrund der Eintraege in Zeile x.
 semiverf  wird erst aufgerufen,
 wenn Zeile x gefuellt ist.
```

```
*/

void semiverf(x)
SCHAR x;
{
 SCHAR *nextpart,*block,blanz,blockgr,einsanz,i;
 block=part[x];
 blanz=block[0];
 nextpart=part[x+1];
 einsanz=einsen[x][1];
 blockgr=block[2]-block[1];

 if(blockgr==1)
  {
   nextpart[0]=0;
   part[0][x+1]=x+1;
  }
 else
 if(einsanz==1)
  {
   nextpart[1]=x+2;
   nextpart[0]=1;
   part[0][x+1]=x+1;
  }
 else
 if(einsanz==blockgr||einsanz==0)
  {
   nextpart[1]=x+2;
   nextpart[0]=1;
   part[0][x+1]=x+blockgr;
  }
 else
  {
   nextpart[1]=x+2;
   nextpart[2]=x+einsanz+1;
   nextpart[0]=2;
   part[0][x+1]=x+einsanz;
  }

 for(i=2;i<=blanz;i++)
  {
   einsanz=einsen[x][i];
   blockgr=block[i+1]-block[i];
```

```
   if(einsanz==blockgr||einsanz==0)
    nextpart[++nextpart[0]]=block[i];
   else
    {
     nextpart[++nextpart[0]]=block[i];
     nextpart[++nextpart[0]]=block[i]+einsanz;
    }
  }
 nextpart[nextpart[0]+1]=n+1;
}

/*
 ordrek erledigt das Einsetzen
 der Kanten, ruft ggf Girth-
 und Maxtest auf, steuert die
 Ausgabe. Uebergabeparameter:
 (mx,my) zuletzt einges. Kante
 vm min. Knoten mit grad=0
 vor Einsetzen von (mx,my)
 tw Taillenweite vor Einsetzen
 von (mx,my). tw=0, falls noch
 kein Kreis existiert
 lblock Block von part[mx], wo
 eingesetzt wurde
*/

void ordrek(mx,my,vm,tw,lblock)
SCHAR mx,my,vm,tw,lblock;
{
 SCHAR i;

 if(my>n-k&&grad[mx]<k&&n-my<k-grad[mx]) /*noch genug Platz in Zeile mx,*/
  return;                                /*damit grad(mx)=k werden kann */

 if(mx>=n-k&&grad[mx]==k)
  for(i=my+1;i<=n;i++)                   /*testet,ob in Spalte i noch   */
   if(n-mx-1<k-grad[i])                  /*genug Platz, damit grad(i)=k */
    return;                              /*werden kann                  */

 if(my<vm)                               /*my<vm notw. fuer neue Kreise */
  {
   if(tw>3)                              /*falls Taillenweite>3, testen,*/
    {                                    /*ob sie nach Einfuegen von    */
```

```
     if(girthcheck2(mx,my,vm,tw)<tw)     /*(mx,my) gleich geblieben ist,*/
      return;                            /*wenn nein abbrechen          */
    }
   else
   if(tw==0)                             /*dies ist der 1.Kreis         */
    {                                    /*mit girthstart seine Laenge  */
     tw=girthstart();                    /*(=Taillenweite) ermittelt    */
     if(tw==0)
      return;
    }
  }

 while(mx<n&&grad[mx]==k)                /*wenn Zeile mx voll:          */
  {                                      /*verfeinerte                  */
   semiverf(mx++);                       /*Partition berechnen          */
   lblock=1;                             /*Einfuegen wieder von links   */
#ifdef JOBS
   if(mx==mid_max||mx==splitlevel)
    {
     if(maxstartneu1(vm-1)==0)
      return;
     if(mx==splitlevel)                  /*hier wird die Aufteilung auf */
     if(++calls%jobs!=jobnr)             /*mehrere Jobs organisiert     */
      return;
    }
#else
   if(mx==mid_max)
   if(maxstartneu1(vm-1)==0)
    return;
#endif
  }                                      /*mx nun min. Knoten mit grad<k */

 if(vm<=my)vm=my+1;                      /*vm nun min. Knoten mit grad=0 */

 if(mx==n&&grad[n]==k)                   /*Bedingung fuer Regularitaet   */
  {
   if(maxstartneu(tw))
    {
     anz++;
#ifdef LIST                              /*Ausgabe                       */
     if(count)
      {
       if(anz%dez==0)
```

```
        {
         fprintf(ergfile,"          %ld Graphen erzeugt\r",anz);
         fflush(ergfile);
         fseek(ergfile,fpos,0);
         if(anz%(dez*count)==0)
          dez*=10;
        }
       }
      if(storeall)
       komprtofile();
      else
      if(tostore>0)
       {
        komprtofile();
        if(--tostore==0)
         {
          fclose(lstfile);
          fprintf(ergfile,"          %ld Graphen erzeugt\r",anz);
          fflush(ergfile);
          fseek(ergfile,fpos,0);
         }
       }
#endif
#ifdef STAB
      if(printall)
       {
        fprintf(autfile,"\nGraph %d:\n",anz);
        nachblist();
        fprintf(autfile,"Taillenweite: %d\n",tw);
        stabprint();
       }
      else
      if(toprint>0)
       {
        fprintf(autfile,"\nGraph %d:\n",anz);
        nachblist();
        fprintf(autfile,"Taillenweite: %d\n",tw);
        stabprint();
        if(--toprint==0)
         {
          fclose(autfile);
          fprintf(ergfile,"          %ld Graphen erzeugt\r",anz);
          fflush(ergfile);
```

```
         fseek(ergfile,fpos,0);
        }
       }
#endif
     return;
    }
   springen=1;                              /*Lerneffekt aktivieren         */
   return;
  }

 for(i=lblock;i<=part[mx][0];i++)
  if((my=part[mx][i]+einsen[mx][i])<part[mx][i+1])
   {                                        /*moegliche Kante (mx,my)       */
    if(grad[my]<k&&my<=vm)                  /*Bedingung my<=vm notw. fuer   */
     {                                      /*zusammenhaengende Graphen     */
      g[mx][my]=g[my][mx]=1;                /*Einsetzen der neuen Kante     */
      l[mx][++grad[mx]]=my;
      l[my][++grad[my]]=mx;
      einsen[mx][i]++;
      lgrad[my]++;

      ordrek(mx,my,vm,tw,i);

      g[mx][my]=g[my][mx]=0;
      l[mx][grad[mx]]=0;
      l[my][grad[my]]=0;
      grad[mx]--;grad[my]--;
      lgrad[my]--;einsen[mx][i]--;

      if(mx>=n-k&&n-mx-1<k-grad[my])
       return;
      if(springen)
       {
        if(g[i1][j1]==0)
         springen=0;
        else
         return;
       }
     }
   }

 return;
}
```

```
/*
 ordstart initialisiert die benoetigten Datenstrukturen:

 g enthaelt die Adjazenzmatrix, also
 g[i][j]=1,falls i,j verbunden, 0 sonst

 l enthaelt die Nachbarschaftsliste, d.h.
 l[i][1],...,l[i][k] sind die Nachbarn von i

 part[i] enthaelt die Blockzerlegung aufgrund
 der Eintraege in  Zeile i-1 und part[i-1].
 Zeile i muss so gefuellt werden, dass innerhalb
 der Bloecke von part[i] die Einsen links und
 die Nullen rechts stehen.
 part[i][0] : Anzahl der Bloecke;
 part[i][j] : Beginn des j-ten Blocks
      Ausnahme: j=part[i][0], dann n+1;
 part[0][i] : Ende des 1.Blocks von Zeile i vor
      evtl. Abspalten eines 1er Blocks

 einsen[i] enthaelt Anzahl der Einsen in Zeile i
 bzgl. der Blockzerlegung von part[i].
 einsen[i][j] : Anzahl der Einsen in Block j

 lgrad[i] enthaelt Anzahl der Einsen
 in Zeile i links der Diagonalen
*/

void ordstart(_n,_k,_t,_mid_max,
      _splitlevel,_jobs,_jobnr,
      _lstfile,_autfile,_ergfile,
      _tostore,_toprint,_count,
      _storeall,_printall,_anz)
SCHAR _n,_k,_t,_mid_max,_splitlevel;
UINT  _jobs,_jobnr;
FILE  *_lstfile,*_autfile,*_ergfile;
ULONG _toprint,_tostore,_count;
SCHAR _storeall,_printall;
ULONG *_anz;
{
 SCHAR m,in=0,zu=1;
 int  h,i,j;
```

```
 n=_n;k=_k;t=_t;mid_max=_mid_max;
 splitlevel=_splitlevel;jobs=_jobs;jobnr=_jobnr;
 lstfile=_lstfile;autfile=_autfile;ergfile=_ergfile;
 toprint=_toprint;tostore=_tostore;count=_count;
 storeall=_storeall;printall=_printall;
 springen=girth_exact=calls=0;
 dez=1;
 anz=*_anz;
 fpos=ftell(ergfile);

 if(k>=n)
  return;
 if(k==0||k==1)
  return;
 if(n%2==1&&k%2==1)
  return;

 if(!(g     =(SCHAR**)calloc(n+1,sizeof(SCHAR*))))err();
 if(!(l     =(SCHAR**)calloc(n+1,sizeof(SCHAR*))))err();
 if(!(zbk   =(SCHAR**)calloc(n+1,sizeof(SCHAR*))))err();
 if(!(transp=(SCHAR**)calloc(n+1,sizeof(SCHAR*))))err();
 if(!(einsen=(SCHAR**)calloc(n+1,sizeof(SCHAR*))))err();
 if(!(part  =(SCHAR**)calloc(n+1,sizeof(SCHAR*))))err();
 if(!(kmn   =(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
 if(!(grad  =(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
 if(!(lgrad =(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();

 for(i=0;i<=n;i++)
  {
   if(!(g[i]     =(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
   if(!(l[i]     =(SCHAR*)calloc(k+1,sizeof(SCHAR))))err();
   if(!(transp[i]=(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
   if(!(einsen[i]=(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
   if(!(part[i]  =(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
   if(!(zbk[i]   =(SCHAR*)calloc(n+1+k*2,sizeof(SCHAR))))err();
  }

#ifdef STAB
 merz=n*(n-1)/2+1; /*n+(n-2)*k wuerde genuegen*/
 if(!(aut=(SCHAR**)calloc(merz+1,sizeof(SCHAR*))))err();
 for(i=1;i<=merz;i++)
  if(!(aut[i]=(SCHAR*)calloc(n+1,sizeof(SCHAR))))err();
```

```
#endif

#ifdef SHORTCODE
 if(!(lastcode=(SCHAR*)calloc(n*k/2+1,sizeof(SCHAR*))))err();
#endif

 for(i=1;i<=n;i++)
  kmn[i]=i;

 part[0][1]=n;
 part[1][0]=1;
 part[1][1]=2;
 part[1][2]=n+1;
 einsen[1][1]=k;

 for(i=3;i<=t;i++)
  {
   in+=zu;
   if(i%2==0)
    zu*=(k-1);
  }

 if(in*(k-1)+2>n)
  return;

 for(j=2;j<=k+1;j++)
  {
   l[1][++grad[1]]=j;
   l[j][++grad[j]]=1;
   g[1][j]=g[j][1]=1;
   lgrad[j]++;
  }

 for(i=2;i<=in;i++)
  {
   semiverf(i-1);
   for(h=1;h<k;h++)
    {
     l[i][++grad[i]]=j;
     l[j][++grad[j]]=i;
     g[i][j]=g[j][i]=1;
     lgrad[j++]++;
    }
```

```
   einsen[i][part[i][0]]=k-1;
  }

 f1=i;f0=j;
 m=--j;--i;

 if(girth_exact)
  {
   semiverf(i);
   i=in+1;j=in+zu+1;h=1;
   l[i][++grad[i]]=j;
   l[j][++grad[j]]=i;
   g[i][j]=g[j][i]=1;
   lgrad[j]++;
   while(part[i][h]!=j)h++;
   einsen[i][h]++;
  }

 ordrek(i,j,m,0,1);

#ifdef STAB
 for(i=1;i<=merz;i++)
  free(aut[i]);
 free(aut);
#endif

#ifdef SHORTCODE
 free(lastcode);
#endif

 for(i=0;i<=n;i++)
  {
   free(g[i]);
   free(l[i]);
   free(zbk[i]);
   free(transp[i]);
   free(einsen[i]);
   free(part[i]);
  }

 free(g);
 free(l);
 free(kmn);
```

```
 free(zbk);
 free(part);
 free(grad);
 free(lgrad);
 free(einsen);
 free(transp);

 *_anz=anz;
 return;
}
```

Literaturverzeichnis

[Bri] Gunnar Brinkmann: *Generating cubic graphs faster than isomorphism checking.* preprint.

[BriMcK] Gunnar Brinkmann, Brendan D. McKay and Carsten Saager: *The smallest cubic graphs of girth nine.* preprint.

[Gru1] Roland Grund: *Konstruktion schlichter Graphen mit gegebener Gradpartition.* Bayreuther Mathematische Schriften 44 (1993), 73-104.

[Gru2] Roland Grund: *Konstruktion molekularer Graphen mit gegebenen Hybridisierungen und überlappungsfreien Fragmenten.* Bayreuther Mathematische Schriften 49 (1995), 1-113.

[Grü] Thomas Grüner: *Ein neuer Ansatz zur rekursiven Erzeugung von schlichten Graphen.* Diplomarbeit Bayreuth, 1995.

[GLM] T. Grüner, R. Laue, M. Meringer: *Algorithms for group actions: homomorphism principle and orderly generation applied to graphs.* preprint.

[HKLMW] R. Hager, A. Kerber, R. Laue, D. Moser, W. Weber: *Construction of orbit representatives.* Bayreuther Mathematische Schriften 35 (1991), 157-169.

[Ker] Adalbert Kerber: *Algebraic Combinatorics Via Finite Group Actions.* Wissenschaftsverlag, Mannheim, Wien, Zürich, 1991.

[Lau] Reinhard Laue: *Construction of Combinatorial Objects - A Tutorial.* Bayreuther Mathematische Schriften 43 (1993), 53-96.

[Re] R.C. Read: *Everyone a winner.* Annals of Discrete Mathematics 2 (1978), 107-120.

[Sim] C.C. Sims: *Computation with Permutation Groups.* Proc. of the Second Symposium on Symbolic and Algebraic Manipulation (ed. Petrick,S.R.), New York, 1971.

[Won] Pak-Ken Wong: *Cages—A Survey.* Journal of Graph Theory, Vol. 6 (1982), 1-22.

Urhebervermerk

Ich versichere, daß ich diese Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel und Quellen verwendet habe.

Bayreuth,
8. Januar 1996 _____