Algorithmen auf periodischen (Di-)Graphen

Diplomarbeit in Mathematik

von

Daniel Hons

Matr.-Nr. 267729

vorgelegt der

Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen

angefertigt am

Lehrstuhl C für Mathematik

bei

Prof. Dr. Yubao Guo

im Oktober 2012

Vorwort

Periodische Graphen sind Graphen, deren Komponenten jeweils entweder endlich sind, oder sich periodisch wiederholen. Dadurch haben diese Graphen nur einen endlichen Informationsgehalt und können entsprechend auch mit endlichem Speicher implementiert werden. Man kann periodische Graphen unter anderem nutzen um Zeitpläne zu modellieren, die sich periodisch wiederholen (vgl. [7]).

Betrachtet man in der Chemie Kristallnetze, so bietet sich eine Darstellung als periodischer Graph ebenfalls aus Speichergründen an. Die hier untersuchten Fragestellungen ergeben sich in diesem Bereich und sind Teil der Entwicklung von *GTECS 3D* (GTECS, **G**raph **T**heoretical **E**valuation of extended **C**rystallographic **S**tructures,[1])

Dabei handelt es sich um ein Anfang 2011 begonnenes Gemeinschaftsprojekt des Instituts für anorganische Chemie, der Virtual Reality Group und des Lehrstuhls C für Mathematik der RWTH Aachen. Ziel des Projektes ist die Untersuchung und Darstellung kristallographischer Netze mit algorithmischen Methoden. Insbesondere auf die Vereinfachung und Charakterisierung verschiedener Strukturen soll in dieser Arbeit eingegangen werden.

Die aktuelle Version (1.2) ist auf der Seite http://www.gtecs.rwth-aachen.de quelloffen verfügbar. GTECS liest Dateien im CIF-Format (Crystallographic Information File) ein, einem Standard zum Austausch kristallographischer Informationen.

Ich möchte mich an dieser Stelle bei all denen bedanken, die mich bei der Anfertigung meiner Diplomarbeit so tatkräftig unterstützt haben. Ganz besonders danke ich Herrn Prof. Dr. Guo für die Möglichkeit zum Schreiben dieser Arbeit, für die interessante Themenstellung und die Vermittlung der Graphentheorie.

Steffen Grüter danke ich für die stets sehr hilfreichen Ratschläge und die kriti-

sche Durchsicht der Arbeit.

Ich danke allen Personen, die an GTECS mitarbeiten, für die gute Zusammenarbeit, ganz besonders Sven Porsche für seine Hilfe bei der Implementierung der Algorithmen.

Schließlich danke ich Sandra und meinen Eltern für die Unterstützung in der Studienzeit und allen Lebenslagen.

Inhaltsverzeichnis

1 Definitionen und bekannte Resultate

1.1 Definitionen

Wir übernehmen die Definition von periodischen Graphen wie sie bei [7] angegeben ist, erweitern diese aber sofort auf mehrdimensionale periodische Graphen.

Definition 1.1:
Es sei $D = (V, A)$ ein Digraph und $t : A \to \mathbb{Z}^k$ eine Abbildung, die wir *Shift-Funktion* nennen. Das Tripel $G = (V, A, t)$ nennen wir einen *Orbitgraph*. Der *periodischen Digraph* G^∞ hat die Eckenmenge $V(G^\infty) = \{v^p : v \in V, p \in \mathbb{Z}^k\}$ und die Bogenmenge $A(G^\infty)$ erfüllt die folgende periodische Eigenschaft:

$$v^p w^q \in A(G^\infty) \Leftrightarrow vw \in A(G) \text{ mit } t(vw) = q - p$$

Wir sagen, dass der Orbitgraph G den periodischen Graphen G^∞ *induziert*.
Ferner bezeichnen wir k als *Dimension* des Graphen G^∞.

Für Kantenzüge K schreiben wir verkürzend $t(K) = \sum_{k \in K} t(k)$.

In dieser Form sind periodische Graphen stets gerichtet.

Bemerkung 1.2:
Falls G^∞ ungerichtet sein soll, treten alle Kanten paarweise auf:
$uv \in A(G), t(uv) = z \Leftrightarrow vu \in A(G), t(vu) = -z$

Für die Verwendung in GTECS brauchen wir lediglich ungerichtete periodische Graphen. Daher werden wir uns im folgenden auf diese Variante beschränken.

Die Aussage aus Bemerkung (1.2) sei also im folgenden immer erfüllt. Daher können wir für diese Zwecke auch Zusammenhang und Komponenten analog zu gerichteten Graphen definieren.

Definition 1.3:
Wir definieren für $u, v \in V(G^\infty)$ die Relation:
$u \sim v :\Leftrightarrow$ es existiert ein ungerichteter Weg von u nach v.

$\sim$ ist offenbar eine Äquivalenzrelation auf $V(G^\infty)$.

Die Äquivalenzklassen von $\sim$ heißen *(Zusammenhangs-) Komponenten* von G^∞.

Ist $V(G^\infty)$ selbst die einzige Äquivalenzklasse von $\sim$, so nennen wir G^∞ *zusammenhängend*.

Weiter oben wurde bereits erwähnt, dass ein periodischer Graph aus endlichen und/oder unendlichen Komponenten bestehen kann.

Definition 1.4:
Ein (ungerichtete) *Kantenfolge* K in einem Graphen $G = (V, E)$ ist eine Folge
$K = v_1 e_1 v_2 e_2 \dots v_{n-1} e_{n-1} v_n$, mit
$v_i \in V$ für $1 \leq i \leq n$ und
$e_i = \{v_i, v_{i+1}\}$ für $1 \leq i \leq n-1$

Sind alle e_i paarweise verschieden, so nennen wir K einen *Kantenzug*

Sind alle v_i paarweise verschieden, so nennen wir K einen *Pfad*.

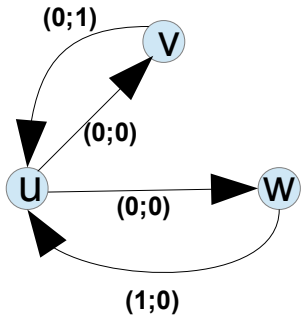


Abbildung 1.1: Orbitgraph...

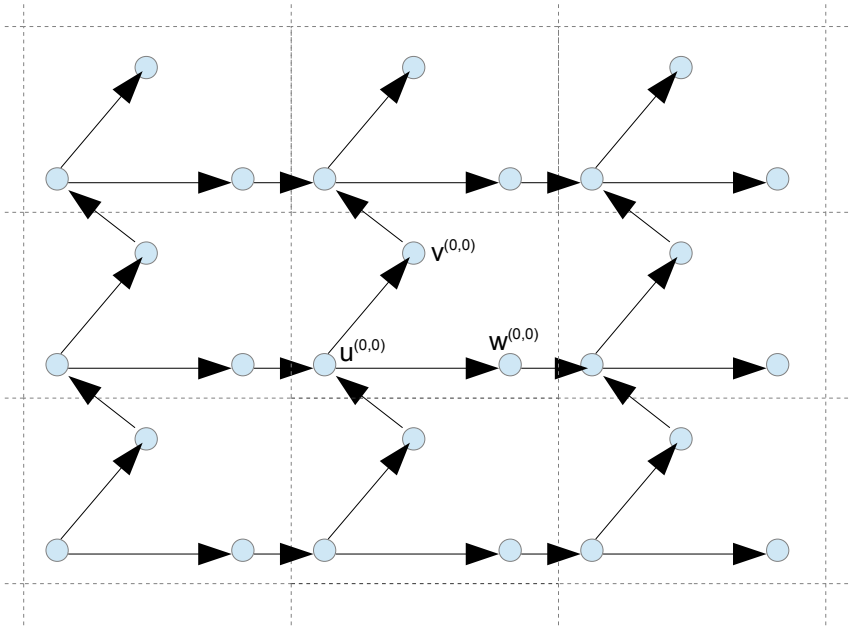


Abbildung 1.2: ...und induzierter periodischer Graph (Ausschnitt)

Falls zwischen zwei Ecken lediglich eine Kante verlaufen kann (G also keine Mehrfachkanten besitzt) so schreiben wir vereinfacht $K = v_1 \dots v_n$. Im folgenden betrachten wir nur Graphen ohne Mehrfachkanten, sofern es sich nicht um Orbitgraphen handelt.

Die *Länge einer Kantenfolge* K ist $|E(K)|$.

Definition 1.5:
Sei $G = (V, E)$ ein Graph und $v \in V$ eine Ecke. Dann ist die *Nachbarschaft* von v definiert durch $N(v) := \{w \in V; vw \in E\}$

Definition 1.6:
Seien $G = (V, A, t)$, G^∞ zusammenhängend mit Dimension k und $p \in \mathbb{Z}^k$. Dann ist die *Dimensionalität* von G^∞ definiert als die Anzahl der linear unabhängigen ganzzahligen Vektoren $\bar{t} \in \mathbb{Z}^k$, sodass es für eine Ecke $a \in V$ einen Pfad von a^p zu $a^{p+\bar{t}}$ in G^∞ gibt.

Bemerkung 1.7:
Die Begriffe „Dimension“ und „Dimensionalität“ sind zu unterscheiden. Offensichtlich ist die Dimensionalität eines periodischen Graphen durch die Dimension nach oben beschränkt, und damit wohldefiniert.

Definition 1.8:
Seien $G = (V, E)$ und $H = (V', E')$ Graphen, dann ist
$\phi : V \to V'$ ein *(Graph-) Isomorphismus* zwischen G und H, falls gilt:

$$\{v_1, v_2\} \in E \Leftrightarrow \{\phi(v_1), \phi(v_2)\} \in E' \text{ für alle } v_1, v_2 \in V$$

Falls ein solcher Isomorphismus existiert, so nennen wir G und H *isomorph*.

1.2 Bekannte Resultate

Periodische Graphen wurden bisher insbesondere für Lösungen von Zeitplanungs-Probleme (engl. Scheduling Problems) herangezogen. So bereits 1968 in [8] für Zeiplanung bei Luftlinien, aber auch 1992 in [11]. Eine weitere Anwendung von 2-dimensionalen periodischen Graphen ist die Darstellung und Optimierung von Schaltkreisen. Diese Anwendung spielt zum Beispiel in [9] und [4] eine besondere Rolle.

In [7] wird insbesondere der Zusammenhang von 1-dimensionalen periodischen Graphen untersucht. In [2] werden einige dieser Ergebnisse auf höherdimensionale Strukturen verallgemeinert. Dort werden auch Algorithmen angegeben, um Bipartitheit zu testen und einen Spannbaum minimaler Kosten zu finden. Planarität von periodischen Graphen wird in [9] behandelt.

Steiglitz und Iwano präsentieren 1987 eine Methode, um Kreise in periodischen Graphen zu finden, bzw. auf Kreisfreiheit zu testen.

Egon Wanke untersucht 1993 in [12] Wege und Kreise in endlichen periodischen Graphen. Er beschreibt Kriterien um zu garantieren, dass Pfad-Probleme mit polynomiellem Aufwand lösbar sind. Es wird außerdem gezeigt, dass solche Probleme im Allgemeinen für periodische Graphen sogar PSPACE-vollständig sind.

Ein wichtiges Resultat für das Verständnis von periodischen Graphen ist das folgende:

Lemma 1.9:
Sei $G = (V, A, t)$ ein Orbitgraph zu einem periodischen Graphen der Dimension k. Seien $u, v \in V$ und $r, s \in \mathbb{Z}^k$.

Dann existiert eine 1:1-Abbildung zwischen den endlichen Kantenzügen von u^r nach v^s in G^∞ und Kantenzügen K in G von u nach v mit $t(K) = s - r$.

Beweise finden sich in [6] für $k = 1$ und in [2] für den allgemeinen Fall.

Im folgenden werden einige der wichtigsten bekannten Ergebnisse zu periodischen Graphen angegeben.

Ein interessanter struktureller Zusammenhang befasst sich mit speziellen Manipulationen der Shiftfunktion t. Solche Manipulationen werden in [7] behandelt.

Definition 1.10:
Eine *Eckenzuweisung* (engl. vertex assignment) ist eine Abbildung $g : V \to \mathbb{Z}^d$.

Eine Eckenzuweisung führt direkt zu einer neuen Shiftfunktion t_g, die sich aus der ursprünglichen Shiftfunktion t folgendermaßen ergibt: Für einen Bogen uv gilt $t_g(uv) = t(uv) + g(u) - g(v)$.

Nun gilt:

Satz 1.11 (Orlin, [7])**:**
Ist $G = (V, A, t)$ ein Orbitgraph und g eine Eckenzuweisung von G. Dann ist auch $\bar{G} = (V, A, t_g)$ ein Orbitgraph und die periodischen Graphen G^∞ und $\bar{G}^\infty$ sind isomorph.

Ebenfalls dort wird ein hinreichendes Kriterium angegeben damit ein 1-dimensionaler periodischer Graph stark zusammenhängend ist. Fü $d = 1$ gilt der folgende Satz:

Satz 1.12 (Orlin, [7])**:**
Sei $G = (V, A, t)$ ein stark zusammenhängender Orbitgraph, und G^∞ zusammenhängend. Wenn gerichtete Kreise C^+ und C^- existieren, sodass
$t(C^-) < 0 < t(C^+)$,
dann ist G^∞ stark zusammenhängend.

Wie bereits erwähnt, untersucht Wanke in [12] endliche periodische Graphen. Zu einem gegebenen ganzzahligen Vektor m mit nicht-negativen Einträgen ist der endliche periodische Graph G^m ein induzierter Teilgraph von G^∞, bestehend aus den Ecken, die in einer Elementarzelle $\leq m$ liegen, wobei komponentenweise verglichen wird. Solche endlichen periodischen Graphen spielen zum Beispiel bei Architekturen von Schaltkreisen eine Rolle.

Wanke untersucht nun die Schwierigkeit von Kreis- und Pfadproblemen auf endlichen periodischen Graphen der Dimension d.

Wichtige Ergebnisse aus dieser Arbeit sind die folgenden:
Pfad- und Kreisprobleme sind grundsätzlich in PSPACE enthalten. Sie sind sogar PSPACE-vollständig in den folgenden Fällen:

- Der Graph hat Dimension 1, $|V| = 1$ und t und m enthalten nur Werte aus $\{-1, 0, 1\}$
- Der Graph hat Dimension 2
- Der Graph hat Dimension 4 und $|V| = 1$.

Hat der Graph die Dimension 1 und der Orbitgraph besteht aus nur einer Ecke, so sind beide Probleme immer noch NP-vollständig.

Für 1-dimensionale Graphen ist das Pfadproblem in P lösbar, falls die Shiftfunktion nur auf Vektoren mit Einträgen aus $\{-1, 0, 1\}$ abbildet.

In [4] werden unendliche periodische Graphen auf Kreisfreiheit untersucht, das heißt es wird das Entscheidungsproblem gelöst, ob G^∞ einen Kreis enthält oder nicht. Der dort angegebene Algorithmus hat für 1-dimensionale periodische Graphen eine Laufzeit von $\mathcal{O}(|V|^3)$ und im Falle der Dimension 2 eine Laufzeit von $\mathcal{O})|A| \cdot log(|A|))$.

2 Elementare Algorithmen

In diesem Kapitel stellen wir elementare Algorithmen zur Vereinfachung und Analyse von periodischen Graphen vor.

2.1 Ermittlung kürzester Pfade

Oft interessieren wir uns für die kürzesten Pfade zwischen zwei Ecken eines periodischen Graphen. Insbesondere wird sich in Kapitel 4 die Frage nach dem Abstand zweier Ecken stellen.

Hier soll zunächst erwähnt werden dass diese Probleme im Kontext periodischer Graphen wesentlich schwerer zu lösen sind als bei endlichen Graphen. Wir präsentieren dazu zunächst ein Ergebnis von Orlin ([7]).

Problem 2.1 (Gerichtete Pfade in G^∞)**:**
EINGABE: Ein Orbitgraph $G = (V, E, t)$ und zwei Ecken $u, v \in V(G^\infty)$
FRAGE: Gibt es einen gerichteten Weg von u nach v in G^∞?

Orlin erhält nun folgendes Resultat(Theorem 3 in [7]):

Lemma 2.2 (Orlin, [7])**:**
Das Problem gerichteter Pfade in G^∞ ist NP-vollständig.

Für den Beweis benötigen wir ein weiteres Problem, welches bekanntermaßen NP-vollständig ist.

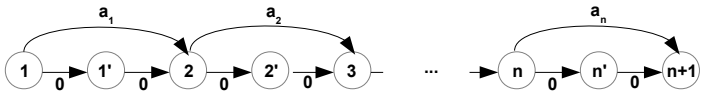
Problem 2.3 (Untermengensumme)**:**
EINGABE: Eine Menge von ganzen Zahlen $I = \{a_1, \dots a_n\}$, eine ganze Zahl c
FRAGE: Existiert $S \subset I$ mit $\sum_{x \in S} x = c$?

Beweis. [zu 2.2] Wir reduzieren das Problem Untermengensumme auf das Problem Gerichtete Pfade in G^∞. Dazu konstruieren wir also zu gegebener Menge $I = \{a_1, \ldots, a_n\} \subset \mathbb{Z}$ und Ganzzahl c einen periodischen Graphen (genauer: seinen Orbitgraphen), in dem es einen Pfad von einer Ecke a zu einer Ecke b gibt genau dann wenn eine Teilmenge $S \subset I$ gibt mit $\sum_{x \in S} x = c$.

Der Orbitgraph $G = (V, A, t)$ besitze die Eckenmenge
$V = \{1, 1', 2, 2', \ldots n, n', n+1\}$ und die Bogenmenge $A = A_1 \cup A_2$ mit
$A_1 = \{(i, i'), (i', i+1) \mid i \in \{1, \ldots, n\}\}$ und
$A_2 = \{(i, i+1) \mid i \in \{1, \ldots, n\}\}$

Es sei $t(a) = 0$ für $a \in A_1$ und $t(a) = t(i, i+1) = a_i$ für $a \in A_2$.

Ein gerichteter Pfad in G^∞ von 1^p nach $(n+1)^{(p+c)}$ existiert nach (1.9) genau dann, wenn es in G einen Pfad P von 1 nach $n+1$ gibt mit $t(P) = c$, denn alle Kantenzüge in G sind Pfade. Ein solcher Pfad entspricht aber genau einer Teilmenge $S \subset I$ mit $\sum_{x \in S} x = c$.



□

Für ungerichtete periodische Graphen, wie wir sie in GTECS betrachten, haben wir einen Algorithmus entworfen, um alle kürzesten Wege zwischen zwei Ecken zu bestimmen. Die Laufzeit ist exponentiell in der Größe des Abstands, aber polynomiell in der Anzahl der Ecken des Orbitgraphen.

Algorithmus 2.4:
Folgender Algorithmus berechnet alle kürzesten Wege zwischen zwei Ecken $a, b \in V(G^\infty)$. Dabei ist $S \subset V(G^\infty)$ die Menge der bereits besuchten Ecken und $dist : S \to \mathbb{N}$ ordnet diesen Ecken den Abstand von a zu.
Für $v \in V(G^\infty)$ ist Pre_v eine Liste von Vorgängern mit minimalem Abstand von a.
Ist der Abstand größer als max, so wird kein Weg gefunden.

Eingabe: Ein periodischer Graph G^∞, $a, b \in V(G^\infty)$ und $max \in \mathbb{N}$
Ausgabe: Alle kürzesten Wege von b nach a in G^∞, die kürzer als max sind.

$S \leftarrow \{a\}$
$dist(a) \leftarrow 0$
$finished \leftarrow false$
for $k = 0 \rightarrow max - 1$ **do**
  if finished **then return** $\{Pre_v | v \in S\}$
  end if
  for all $s \in S, dist(s) = k$ **do**
    for all $w \in N(S)$ **do**
      if $w \notin S$ **then**
        if $w = b$ **then**
          $fnished \leftarrow true$
        end if
        $S \leftarrow S \cup \{w\}$
        $dist(w) \leftarrow k + 1$
        $Pre_w \leftarrow \{v\}$ ▷ *
      else
        if $dist(w) = k + 1$ **then** ▷ $dist(w) \leq k + 1$ trivial
          $Pre_w \leftarrow Pre_w \cup \{v\}$ ▷ *
        end if
      end if
    end for
  end for
end for
return $dist(a, b) \geq max$

Einen ungerichteten Weg von b nach a erhält man jeweils rekursiv durch folgendes Vorgehen:

1. Setze $v_0 = b$
2. Setze $v_i \in Pre_{v_{i-1}}$ beliebig für $i = 1 \ldots$ bis $v_i = a$

Welchen Weg man erhält, hängt von den jeweiligen Wahlen in 2. ab.

Lemma 2.5:
Nach Abschluss des Algorithmus gilt

$$S = \{v \in G^\infty | dist(a, v) \leq max\}$$

Beweis. Das ist klar, da der Algorithmus eine Breitensuche durch G^∞ darstellt, beginnend bei a. Alle besuchten Ecken werden zu S hinzugefügt. □

Lemma 2.6:
Für eine Ecke $s \in S$ enthält Pre_s genau diejenigen Ecken, die auf einem kürzesten Weg von a nach b direkt vor s auftauchen. Insbesondere gilt $Pre_s = \{a\}$ für $s \in N(a)$.

Beweis. Ist $dist(a, b) = k$ und $P = av_1v_2 \dots v_{k-1}b$ ein kürzester Weg von a nach b, so gilt $Pre_{v_1} = \{a\}$, denn v_1 erhält den Abstand 1 von a und keine andere Ecke als a selbst hat einen kleineren Abstand. Im Algorithmus gilt nämlich $dist(v) = 0 \Leftrightarrow v = a$.
Offensichtlich gilt $Pre_s \subset N(s)$, da nur Nachbarn in die Liste der Vorgänger aufgenommen werden. Wegen den mit * markierten Zeilen gilt aber auch: $Pre_s \subset \{v \in S | dist(a, v) = dist(a, s) - 1\}$. Daraus folgt

$$Pre_s \subset \{v \in S | dist(a, v) = dist(a, s) - 1\} \cap N(s). \qquad (**)$$

Ist andererseits $v \in N(s)$ mit $dist(a, v) = dist(a, s) - 1$, so wird s beim Untersuchen von $N(v)$ in einer der beiden mit * markierten Zeilen zu Pre_s hinzugefügt. Damit gilt in (**) die Gleichheit und daraus folgt die Behauptung. □

Die Korrektheit von Algorithmus 2.4 folgt sofort aus 2.5 und 2.6. Für die Komplexität gilt nun:

Satz 2.7:
Sind $a, b \in V(G^\infty)$ mit $dist(a, b) = l$ und ist $max \geq l$, so findet Algorithmus 2.4 alle kürzesten Wege zwischen a und b in Laufzeit $\mathcal{O}(\Delta_G^l)$.

Beweis. Jedes Element aus S wird in konstanter Zeit abgearbeitet. Für jedes k werden die Elemente aus S mit $dist(s) = k$ betrachtet, also hat der Algorithmus eine Laufzeit von $\mathcal{O}(|S|)$.

Wegen Lemma (2.5) ist das aber identisch zu $\mathcal{O}(\Delta_G^l)$. □

Bemerkung 2.8:
Wählt man einen beliebigen kürzesten Weg, kann man Algorithmus (2.4) offensichtlich benutzen um den Abstand zweier Ecken zu ermitteln. Dabei gibt man ebenso eine maximale Suchtiefe an. Bis zu diesem Abstand arbeitet der Algorithmus korrekt. Ist der Abstand größer, wird kein Weg gefunden.

2.2 Vereinfachungen

Eine wichtige Aufgabe von GTECS ist es, dem Benutzer zur besseren Übersicht topologisch gleiche bzw. ähnliche aber einfachere Strukturen anzuzeigen. Die leichtesten Vereinfachungen sind dabei das Entfernen von terminalen Ecken (also Ecken vom Grad eins) sowie das Entfernen von Ecken mit Grad zwei, wobei deren Nachbarn verbunden werden. Führt man beide Schritte wiederholt aus (bis das Verfahren terminiert) erhält man eine wesentlich kleinere Struktur von insbesondere gleicher Dimensionalität und ähnlicher Topologie. So bleiben zum Beispiel Kreise erhalten, wobei sich ihre Länge ändern kann.

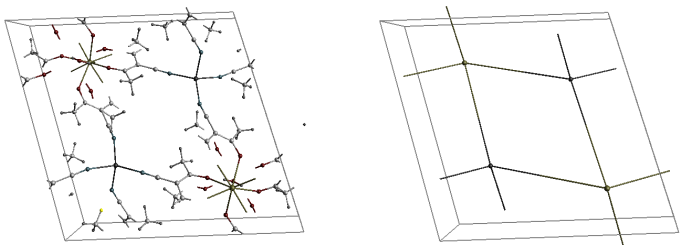


Abbildung 2.1: Kristall vor und nach elementarer Vereinfachung, Quelle: GTECS[1]

Beide Algorithmen sind nicht sehr kompliziert, sollten aber wegen der Vollständigkeit hier aufgeführt werden. Wie beginnen mit dem Algorithmus zum Entfernen terminaler Ecken.

Algorithmus 2.9:
Folgender Algorithmus entfernt Pfade am „Rand“ einer Struktur.
Eingabe: Ein ungerichteter Orbitgraph $G = (V, E)$
Ausgabe: Ein ungerichteter Orbitgraph $G' = (V', E')$ ohne terminale Ecken, d.h. $\delta(G') = 2$.

$G' \leftarrow G$
$finished \leftarrow false$
while $finished = false$ **do**
  $finished \leftarrow true$
  for all $v \in V$ **do**
    if $d_{G'}(v) = 1$ **then**
      $finished \leftarrow false$
      $G' \leftarrow G' - v$
    end if
  end for
end while

Die Korrektheit dieses Algorithmus ist trivial. Die Komplexität ist quadratisch:

Lemma 2.10:
Algorithmus (2.9) hat eine Worst-Case-Laufzeit von $\mathcal{O}(|V|^2)$.

Beweis. Zunächst sieht man, dass die Worst-Case-Laufzeit angenommen wird falls in jedem Durchlauf der ForAll-Schleife genau eine Ecke entfernt wird. Wird nämlich keine Ecke entfernt so terminiert der Algorithmus.

Wird aber in jedem Schritt genau eine Ecke entfernt, so werden im k-ten Durchlauf der While-Schleife genau $|V|-k$ Ecken betrachtet. Die letzte Ecke vom Grad null wird nicht mehr entfernt. Insgesamt werden also $\sum_{k=0}^{|V|-1} |V|-k = \sum_{k=1}^{|V|} k = \mathcal{O}(|V^2|)$ Schritte benötigt. □

Als nächstes betrachten wir einen Algorithmus zum Entfernen von Ecken mit Grad zwei.

Algorithmus 2.11:
Folgender Algorithmus entfernt Ecken mit Eckengrad zwei.
Eingabe: Ein ungerichteter Orbitgraph $G = (V, E)$
Ausgabe: Ein ungerichteter Orbitgraph $G' = (V', E')$ ohne Ecken vom Grad zwei.

$G' \leftarrow G$
$finished \leftarrow false$
while $finished = false$ **do**
  $finished \leftarrow true$
  for all $v \in V$ **do**
    if $d_{G'}(v) = 2$ **then** ▷ $N(v) = \{a, b\}$
      $finished \leftarrow false$
      $G' \leftarrow G' - v$
      $E' \leftarrow E' \cup (a, b)$
    end if
  end for
end while

Wieder ist die Korrektheit trivial, außerdem ist die Komplexität offensichtlich analog zum vorigen Algorithmus durch $\mathcal{O}(|V|^2)$ beschränkt.

2.3 Kreise

Zu gegebenen Graphen G und H möchte man oftmals wissen, ob G einen zu H isomorphen Teilgraphen enthält. Diese Frage ist bereits für endliche Graphen NP-vollständig (z.B. ist das Hamiltonkreisproblem auf diese Weise darstellbar). Es ist dennoch möglich, effiziente Algorithmen für gewisse Graphen H anzugeben.

Im Folgenden geben wir einen Algorithmus an, der Kreise einer gegebenen Länge k findet. Dies spielt in GTECS eine besondere Rolle, da solche Kreise im Zuge der Vereinfachung von Strukturen durch eine einzelne Ecke ersetzt werden. Außerdem stellt diese Möglichkeit ein Werkzeug dar, um nach weiteren Strukturen zu suchen. Ist H beispielsweise ein Tetraeder, so sucht man zunächst alle Kreise

der Länge $k = 3$. Haben alle Ecken eines solchen Kreises einen gemeinsamen Nachbarn, so bilden die Ecken des 3-Kreises zusammen mit diesem Nachbarn einen Tetraeder.

Sei nun $G = (V, E)$ ein Orbitgraph. Wir beschränken uns hier auf einen Algorithmus, der zu gegebenem $v \in V(G)$ und $k \in \mathbb{N}$ alle Kreise der Länge k in G^∞ findet, die die Ecke v beinhalten. Falls es solche Kreise gibt, so gibt es wegen der Periodizität unendlich viele. Es reicht jedoch aus die endliche Anzahl an Kreisen als Repräsentanten anzugeben, die sich in einer bestimmten Elementarzelle befinden, also beispielsweise die Ecke v^p mit $p = \begin{pmatrix}0\\0\\0\end{pmatrix}$ beinhalten.

Dazu laufen wir alle Pfade beginnend bei v^p mittels Tiefensuche auf G^∞ ab.

Algorithmus 2.12:
Folgender Algorithmus findet Kreise der Länge k durch eine Ecke v.
Eingabe: Ein Orbitgraph $G = (V, A, t)$, $k \in \mathbb{N}$, $v \in V$
Ausgabe: Alle Kreise der Länge k durch eine Kopie von v in G^∞

$p \leftarrow (0, 0, 0)^T$
$path \leftarrow v^p$
$cycleSearchRec(G, path, k)$
function CYCLESEARCHREC(G,$P = v_0^{p_0} v_1^{p_1} \dots v_t^{p_t}$,$k$)
  if $k = 1$ **then**
    if $v_t^{p_t} v_0^{p_0} \in A(G^\infty)$ **then** ▷ Kreis gefunden
      Füge $v_0^{p_0} v_1^{p_1} \dots v_t^{p_t} v_0^{p_0}$ zur Ausgabe hinzu.
    end if
  else
    for $w^q \in N(v_t^{p_t})$ **do**
      for $i = 0 \dots t$ **do**
        if $v_i = w$ und $p_i = q$ **then**
          verwerfe dieses w ▷ *
        end if
        $cycleSearchRec(G, v_0^{p_0} v_1^{p_1} \dots v_t^{p_t} w^q, k-1)$
      end for
    end for
  end if
end function

Satz 2.13:
Algorithmus (2.12) arbeitet korrekt.

Beweis. Augenscheinlich arbeiten wir hier auf der unendlichen Struktur G^∞, allerdings sind alle Abfragen bereits durch den Orbitgraphen G durchführbar. Ein Bogen zwischen zwei Ecken $v^p, w^q \in V(G^\infty)$ existiert genau dann, wenn es einen Bogen a von v nach w mit $t(a) = q - p$ in G gibt. Daher können wir den Algorithmus in der anschaulicheren Variante unter Verwendung von G^∞ formulieren.

Dass jeder Kantenzug beginnend in v^p untersucht wird, und damit jeder geschlossene Kantenzug gefunden wird, ist klar da es sich um eine gewöhnliche Tiefensuche handelt. Da die Rekursion k mit jeder hinzugefügten Kante um eins reduziert wird ist die Länge eines ausgegebenen Kantenzugs immer k. Es bleibt zu zeigen, dass alle gefundenen Kantenzüge auch wirklich Kreise sind.

Wird ein Kantenzug in die Ausgabe hinzugefügt, ist er offensichtlich geschlossen. Das garantiert die Fallunterscheidung im Fall $k = 1$.

Weiterhin kann aber keine Ecke von G^∞ doppelt auf einem gefundenen Kreis vorkommen (mit Ausnahme von $v_k^{p_k} = v_0^{p_0}$), denn diese Ecke wäre in Zeile (*) beim zweiten Auftreten verworfen worden. □

Da die Rekursionstiefe genau k ist und es maximal $\Delta_G^\infty = \Delta_G$ Verzweigungen in jedem Rekursionsschritt gibt, folgt sofort eine Abschätzung für die Komplexität.

Lemma 2.14:
Algorithmus (2.12) hat eine Laufzeit von maximal $\mathcal{O}(\Delta_G^k)$.

3 Der Traversal-Algorithmus

3.1 Motivation

Sobald in GTECS eine CIF-Datei geöffnet und geparst wurde (also als periodischer Graph vorliegt), ist es sinnvoll Informationen über die Dimensionalität und die Anzahl der Komponenten zu erhalten. Zusätzlich kann auch gleich eine Basis der periodischen Ausdehnung angegeben werden. Um die graphische Darstellung sinnvoll anzupassen, ist es außerdem notwendig zu erkennen, wie viele Elementarzellen angezeigt werden müssen, um einen vollständigen Überblick über die Struktur zu erhalten, ohne dass die Ausgabe unübersichtlich erscheint. Es ist also die Frage zu beantworten, ab wann sich die Struktur wiederholt. Um dies zu erreichen, nutzen wir den Traversal-Algorithmus, der im folgenden behandelt wird. Wir präsentieren den Algorithmus und zeigen, dass er die erwähnten Anforderungen erfüllt.

Falls der Orbit-Graph mehrere Komponenten besitzt, so werden diese angegeben. Jede Komponente des Orbit-Graphen wird seperat auf Periodizität untersucht und im Falle einer 3-dimensionalen Komponente in G wird die Anzahl der sich durchdringenden identischen Komponenten von G^{∞} angegeben.

3.2 Algorithmus

Wir geben zunächst den Algorithmus an, und wenden uns später der Analyse zu.

Algorithmus 3.1:
Traversal-Algorithmus.
Anfangs haben alle Ecken von G die Komponenten-Nummer (componentID) 0.

queue ist eine Warteschlange aus Elementen aus $V \times \mathbb{Z}^3$, funktioniert also nach dem FIFO-Prinzip (First In - First Out), das heißt Elemente werden in der gleichen Reihenfolge aus der *queue* entnommen und bearbeitet, in der sie hinzugefügt wurden.
In $ComponentUC$ wird die Elementarzelle gespeichert, in welcher eine Ecke zum ersten Mal gefunden wird.
Eingabe: Ein Orbit-Graph $G = (V, E)$

$component \leftarrow 1$ ▷ die erste untersuchte Komponente hat die Nummer 1
$minX \leftarrow 0$
$maxX \leftarrow 0$
$minY \leftarrow 0$
$maxY \leftarrow 0$
$minZ \leftarrow 0$
$maxZ \leftarrow 0$
$queue \leftarrow$ Leere Liste
for all $v \in V$ **do**
  if $componentID(v) = 0$ **then**
    $basis \leftarrow \emptyset$
    $compSizeX \leftarrow 0$
    $compSizeY \leftarrow 0$
    $compSizeZ \leftarrow 0$
    Füge $(v, 0, 0, 0)$ zu *queue* hinzu
    while *queue* nicht leer **do**
      Entnimm (u, x_u, y_u, z_u) von *queue*
      if $componentID(u) \neq 0$ **then** ▷ u markiert
        $(x'_u, y'_u, z'_u) \leftarrow componentUC(u)$
        if $(x'_u, y'_u, z'_u) \neq (x_u, y_u, z_u)$ **then** ▷ periodisch
          if $basis \cup \{(x'_u, y'_u, z'_u) - (x_u, y_u, z_u)\}$ linear unabh. **then**
            $basis \leftarrow basis \cup \{(x'_u, y'_u, z'_u) - (x_u, y_u, z_u)\}$
          else falls $k \cdot ((x'_u, y'_u, z'_u) - (x_u, y_u, z_u)) \in basis, k \in \mathbb{Z}_{\geq 1}$
            ersetze diesen durch $((x'_u, y'_u, z'_u) - (x_u, y_u, z_u))$
          end if
          $minX \leftarrow \min\{minX, u_x\}$
          $maxX \leftarrow \max\{maxX, u_x\}$
          $minY \leftarrow \min\{minY, u_y\}$
          $maxY \leftarrow \max\{maxY, u_y\}$

```
                minZ ← min{minZ, u_z}
                maxZ ← max{maxZ, u_z}
            end if
        else                                          ▷ u unmarkiert
            componentID(u) ← component
            componentUC(u) ← (x_u, y_u, z_u)
            for all w ∈ N(u), also für uw ∈ E do
                compSizeX ← max{compSizeX, |x_u|}
                compSizeY ← max{compSizeY, |y_u|}
                compSizeZ ← max{compSizeZ, |z_u|}
                v̄ ← (x_u, y_u, z_u) + t(uw)
                Füge (w, v̄) zur queue hinzu
            end for
        end if
        if compSizeX > maxX − minX then
            maxX ← minX + compSizeX
        end if
        if compSizeY > maxY − minY then
            maxY ← minY + compSizeY
        end if
        if compSizeZ > maxZ − minZ then
            maxZ ← minZ + compSizeZ
        end if
    end while
    component ← component + 1
  end if
end for
```

3.3 Dimension

Eine Anforderung an Algorithmus (3.1) war es, die Dimensionalität jeder Komponente zu bestimmen. Wir zeigen nun, dass der Algorithmus dem gerecht wird.

Satz 3.2:
Nach Durchlauf der WHILE-Schleife enthält *basis* eine Basis des Untervektorraums, in dem die entsprechende Komponente unendlich ausgedehnt, also periodisch ist.

Beweis. Wir betrachten zunächst die Vektoren, die zusammen mit den Ecken des Orbitgraphen in die Queue aufgenommen werden. Sie entsprechen der Elementarzelle, in welcher die Ecke gefunden wurde, relativ betrachtet zu der ersten Ecke, die zu einer Komponente in die Queue aufgenommen wird (Wir führen hier eine Breitensuche beginnend in Elementarzelle $(0, 0, 0)$ durch). Ihr Zustandekommen ist offensichtlich: Wird ein Eintrag $(u, \bar{u})$ ($\bar{u} \in \mathbb{Z}^3$) aus der Queue abgearbeitet, und ist $w \in N(u)$, so wird der Eintrag $(w, \bar{u} + t(uw))$ in die Queue eingefügt.

Eine Ecke u des Orbitgraphen wird genau dann mit verschiedenen Vektoren $\bar{u}, \bar{v}$ in die Queue eingetragen, wenn es einen Pfad in G^∞ zwischen u in Elementarzelle $\bar{u}$ und u in Elementarzelle $\bar{v}$ gibt. Genau dann ist aber G^∞ offensichtlich unendlich (periodisch) in Richtung $\bar{r} = \bar{u} - \bar{v}$. In diesem Fall wird die Basis auch im Algorithmus um diesen Vektor ergänzt, falls er nicht bereits zu linearer Abhängigkeit führt und damit schon (nicht zwingend explizit) enthalten ist.

Die Vektoren haben weiterhin minimale Größe, das heißt: Wird der Vektor $\bar{r}$ als Basisvektor benutzt, so gibt es keinen kürzeren ganzzahligen Vektor $c \cdot \bar{r}, c < 1$, sodass ein Pfad zwischen u in Elementarzelle $(0, 0, 0)$ und u in Elementarzelle $c \cdot \bar{r}$ existiert, denn sonst wäre $\bar{r}$ durch diesen Vektor ersetzt worden. □

3.4 Komponenten

Wir zeigen nun, dass Algorithmus (3.1) die Ecken in Komponenten des Orbitgraphen einteilt. Weiterhin kann mit diesem Algorithmus bestimmt werden, aus wie vielen Komponenten eine 3-dimensionale Struktur besteht. Für k-dimensionale Strukturen, $k \leq 2$, macht diese Bestimmung keinen Sinn, da es sich um unendlich viele Komponenten handelt, und die Anzahl der angezeigten Komponenten lediglich durch die Anzahl der angezeigten Elementarzellen beschränkt ist.

Satz 3.3:
Nach Durchlauf von Algorithmus (3.1) gilt für zwei Ecken u, v des Orbitgraphen:

$componentID(u) = componentID(v) \Leftrightarrow u$ und v liegen in einer Komponente.

Beweis. „⇒“ Falls $componentID(u) = componentID(v)$, so wurden beide Ecken im gleichen Durchlauf der WHILE-Schleife erreicht, denn zwischen verschiedenen Durchläufen wird *component* erhöht. Da die WHILE-Schleife durchlaufen wird bis *queue* leer ist, und nur Nachbarn von Ecken aus *queue* dort aufgenommen werden, gibt es einen Pfad von u nach v. Also liegen u und v in einer Komponente des Orbitgraphen.

„⇐“ Da u und v in der gleichen Komponente des Orbitgraphen liegen, gibt es einen Pfad $P = p_0 p_1 \dots p_k$ mit $p_0 = u$ und $p_k = v$. Angenommen nach Durchlauf des Algorithmus gilt $componentID(u) \neq componentID(v)$. Dann existiert auch ein $0 \leq l \leq k-1$ mit $componentID(p_l) \neq componentID(p_{l+1})$. O.B.d.A. sei $componentID(p_l) < componentID(p_{l+1})$. Daraus folgt, dass p_l als erster in der Queue aufgenommen und bearbeitet wird. Da jedoch $p_{l+1} \in N(p_l)$, wird p_l in die Queue aufgenommen und daher im gleichen Durchlauf der WHILE-Schleife bearbeitet. In diesem Durchlauf wird *component* aber nicht erhöht, sodass beide Ecken die gleiche $componentID$ bekommen. Das ist eine Widerspruch und beweist die Behauptung. □

Satz 3.4:
Falls eine Komponente des Orbitgraphen 3-dimensional periodisch ist, so ergibt sich die Anzahl der Komponenten in G^∞ als die Determinante der Matrix, deren Spalten die Basisvektoren sind, die in der WHILE-Schleife gesammelt wurden.

Beweis. Nach dem Beweis von (3.2) sind die Basisvektoren in ihrer Länge minimal für die Periodizität. Die Anzahl der Komponenten von G^∞ ergibt sich aus dem Volumen des ausgespannten Spats, dividiert durch die Größe einer Elementarzelle. Das ist aber genau die Determinante. □

3.5 Darstellungsgröße

Algorithmus (3.1) bestimmt eine sinnvolle Größe für die Darstellung. Das heißt es werden so viele Elementarzellen in jeder Richtung angezeigt wie notwendig sind, um die komplette Struktur zu erfassen. Dazu werden zwei Eigenschaften sichergestellt, für endliche und unendliche Komponenten.

Definition 3.5:
Nach Durchlauf von Algorithmus (3.1) ergibt sich der *Darstellungsvektor* als

$$v_{disp} = \begin{pmatrix} x_{disp} \\ y_{disp} \\ z_{disp} \end{pmatrix} = \begin{pmatrix} maxX - minX + 1 \\ maxY - minY + 1 \\ maxZ - minZ + 1 \end{pmatrix}$$

Jede Komponente des Darstellungsvektors entspricht dabei der Anzahl der Elementarzellen die in dieser Richtung angezeigt werden.

Bemerkung 3.6:
Die Addition der Konstante 1 in jeder Komponente ist notwendig. Sie garantiert insbesondere, dass Zusammenhangskomponenten, die in nur einer Elementarzelle liegen, überhaupt angezeigt werden, da in diesem Fall zum Beispiel $maxX - minX = 0$ gilt.

Zunächst stellen wir fest, dass endliche Zusammenhangskomponenten wie gewünscht dargestellt werden.

Satz 3.7:
Besitzt G^∞ eine endliche Komponente, so wird nach Wahl des Darstellungsvektors mindestens eine Kopie der endlichen Komponente vollständig angezeigt.

Beweis. Eine endliche, also nicht periodische Komponente von G^∞ ist bereits im Orbitgraphen G eine einzelne Komponente. Da jede Komponente von v_{disp} im Verlauf des Algorithmus höchstens vergrößert wird, braucht nur gezeigt zu werden, dass im Schleifendurchlauf zu $componentID = k$ der Vektor v_{disp} so groß gewählt wird, dass die endliche Zusammenhangskomponente k einmal vollständig angezeigt wird. Aus Symmetriegründen können wir uns ohne Einschränkung auf die x-Richtung beziehen.

Es muss also lediglich gezeigt werden, dass für zwei Ecken (v_1, x_1, y_1, z_1) und

(v_2, x_2, y_2, z_2), die in einer Zusammenhangskomponente von G^∞ liegen, gilt: $|x_1 - x_2| \leq maxX - minX$, wobei für $minX$ und $maxX$ der Wert nach Durchlauf der zu dieser Zusammenhangskomponente gehörenden Schleife angenommen wird.

Dies ist aber klar: Der Algorithmus ist eine Breitensuche über alle Ecken der Komponente. Dabei wird die erste Ecke, die zu dieser Komponente gehört, in Elementarzelle $(0, 0, 0)$ angenommen. $minX$ und $maxX$ speichern beim Ablaufen der Komponente die minimale bzw. maximale x-Koordinate der auftretenden Elementarzellen. □

Für unendliche, periodische Komponente haben wir ein vergleichbares Resultat.

Satz 3.8:
Für jede periodische Zusammenhangskomponente von G^∞ existiert in jede Richtung, in welche die Komponente periodisch ist, eine Ecke $u \in V(G)$ des Orbitgraphen, die in verschiedenen Elementarzellen angezeigt wird.

Beweis. Wie zuvor wird die gesamte Komponente des Orbitgraphen mittels Breitensuche abgearbeitet. Im Unterschied zu einer endlichen Komponente wird jedoch hier eine Ecke des Orbitgraphen in verschiedenen Elementarzellen erreicht, das heißt dass die IF-Abfrage, die mit „periodisch“ kommentiert ist, mindestens einmal erfüllt wird.

In diesem Fall werden keine Nachbarn zur *queue* hinzugefügt, denn die Ecke wurde bereits in einer anderen Elementarzelle behandelt. Es werden aber weiterhin die Werte $minX, maxX, \ldots$ angepasst, so dass diese Ecke gemeinsam mit der zuerst abgearbeiteten Kopie angezeigt wird. □

3.6 Komplexität

Nun untersuchen wir abschließend die Komplexität des vorgestellten Algorithmus.

Satz 3.9:
Algorithmus (3.1) hat eine Laufzeit von höchstens $\mathcal{O}(|V| \cdot (\Delta_G \cdot \kappa + 1))$.
κ bezeichnet dabei die Anzahl der Komponenten von G.

Beweis. Die äußere Schleife läuft über alle Ecken aus V, allerdings wird nur einmal pro Komponente der innere Teil ausgeführt. In diesem inneren Teil wird die *queue* abgearbeitet. Dort kann aber jede Ecke höchstens so oft auftreten, wie sie Nachbarn hat. Es ergibt sich also eine Worst-Case-Komplexität von
$\underbrace{(|V| - \kappa)}_{\mathcal{O}(|V|)} \cdot \mathcal{O}(1) + \kappa \cdot \mathcal{O}(\Delta_G \cdot |V|) = \mathcal{O}(|V| \cdot (\Delta_G \cdot \kappa + 1))$ □

4 Charakteristische Symbole

Zur Klassifizierung topologisch verschiedener Strukturen existiert eine Reihe unterschiedlicher charakteristischer Symbole. Die Namensgebung ist in der Literatur nicht eindeutig, allerdings schlagen Blatov, O'Keeffe und Proserpio in [10] eine Konvention vor, die hier verwendet wird.

4.1 Schläfli-Symbol

Obwohl das Schläfli-Symbol lediglich auf regulären Strukturen verwendet werden kann, stellt es einen gewissen Standard dar und sollte der Vollständigkeit halber auch hier behandelt werden. In GTECS wird das Schläfli-Symbol allerdings nicht berechnet, daher kann es hier bei einer kurzen Zusammenfassung bleiben, genauere Informationen sind in [10] und [3] zu finden.

Definition 4.1:
Ein Schläfli-Symbol hat die Form $\{p, q, r, \dots\}$. Ein Schläfli-Symbol $\{p\}$ mit $p \in \mathbb{N}$ entspricht einem regelmäßigen p-Eck. Treffen in jeder Ecke einer regulären Struktur q regelmäßige p-Ecke zusammen, so hat der entstehende Polyeder bzw. die entstehende Parkettierung das Schläfli-Symbol $\{p, q\}$. Treffen an jeder Kante einer regulären Struktur r Gebilde mit Schläfli-Symbol $\{p, q\}$ zusammen, so hat das entstehende Gebilde das Schläfli-Symbol $\{p, q, r\}$. Dieses Schema kann in höhere Dimensionen rekursiv fortgesetzt werden.

Bemerkung 4.2:
Üblicherweise wird zugelassen, dass p ein (nicht notwendig gekürzter) Bruch ist. Dann erhält man Sterne, wie zum Beispiel das Pentagramm mit dem Schläfli-Symbol $\frac{5}{2}$. Für unsere Betrachtungen sind Sterne jedoch nicht von Interesse.

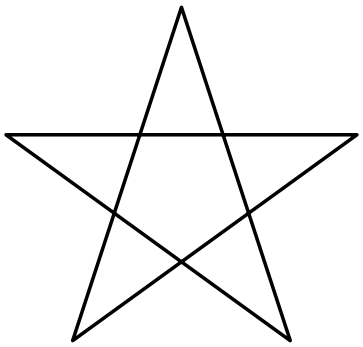


Abbildung 4.1: Pentagramm

Bemerkung 4.3:
Kehrt man die Reihenfolge der Zahlen eines Schläfli-Symbols um, so erhält man die duale Struktur.

Beispiel 4.4:
Polygone:

- $\{n\}$ entspricht einem regelmäßigen n-Eck.

Platonische Körper:

- $\{3, 3\}$ entspricht dem Tetraeder.
- $\{3, 4\}$ entspricht dem Oktaeder.
- $\{4, 3\}$ entspricht dem Hexaeder.
- $\{3, 5\}$ entspricht dem Ikosaeder.
- $\{5, 3\}$ entspricht dem Dodekaeder.

Parkettierungen:

- $\{3, 6\}$ entspricht einer Parkettierung durch Dreiecke.
- $\{6, 3\}$ entspricht einer Parkettierung durch Sechsecke.
- $\{4, 4\}$ entspricht einer Parkettierung durch Quadrate.

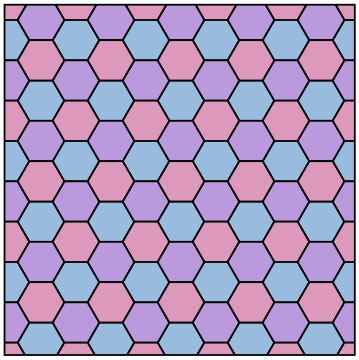


Abbildung 4.2: Sechseck-Parkettierung
[5]

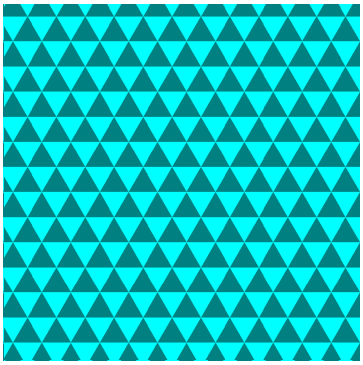


Abbildung 4.3: Dual: Dreieck-Parkettierung

4.2 Point- und Vertexsymbol

In GTECS werden sowohl das Vertex- als auch das Pointsymbol umgesetzt, deren Berechnung ähnlich ist, wobei das Vertexsymbol zum Teil wesentlich mehr Rechenzeit beansprucht. Dies liegt daran, dass meist nach viel Längeren Kreisen (bzw. Ringen) gesucht werden muss, was sich nachteilig auf die Laufzeit auswirkt.

Kreise, also geschlossene Kantenzüge, wurden bereits in (1.4) definiert. Um nun zusätzlich Ringe definieren zu können, brauchen wir folgenden Operator.

Definition 4.5:
Sei G ein Graph und C_1, C_2 Kreise in G. Dann ist die *Summe* $C_1 \oplus C_2$ definiert als diejenigen Kanten, die entweder in C_1 oder in C_2 auftreten.

Fasst man Kreise als Kantenmengen auf, so entspricht die Summe genau der Symmetrischen Differenz. Man beachte dass $\oplus$ assoziativ ist, und eine Kante genau dann in der Summe mehrerer Kreise enthalten ist wenn sie in einer ungeraden Anzahl Summanden enthalten ist.

Definition 4.6:
Ein Kreis C heißt *Ring* wenn er nicht die Summe zweier kleinerer Kreise ist. C heißt *starker Ring* wenn C nicht die Summe irgendeiner Anzahl kleinerer Kreise ist.

Sowohl beim Vertex- als auch beim Pointsymbol wird für jede Ecke v des Graphen die lokale Topologie in deren Nähe betrachtet. Für jedes Paar (a, b) von Nachbarn von v, diese Paare werden als *Winkel* bezeichnet, bestimmen wir die Länge eines kürzesten Kreises (Pointsymbol) bzw. Ringes (Vertexsymbol) zu dem sich der Teilpfad avb erweitern lässt, sowie die Anzahl dieser kürzesten Kreise bzw. Ringe. Hier ist zu beachten, dass die Anzahl der kürzesten Kreise in [10] nicht in das Pointsymbol übernommen wird, während wir bei GTECS auf diese zusätzliche Information nicht verzichten wollten.

Gibt es nun zu einem Winkel von v genau q kürzeste Kreise der Länge p, so ist p_q eine Komponente des Pointsymbols von v, wir bezeichnen p_q als *Symbolteil*. Dies gilt analog für Ringe beim Vertexsymbol. Hat v den Grad d, so besitzt v genau $\frac{d(d-1)}{2}$ Winkel, das heißt beide Symbole haben genau diese Anzahl an Komponenten. Für den Fall, dass es zu einem Winkel keinen Kreis oder Ring gibt, so wird anstelle einer Zahl das Symbol * benutzt.

Definition 4.7:
Sei $G = (V, E)$ ein Graph. Sei $v \in V$ eine Ecke mit Eckengrad d. Das *Vertexsymbol* und das *Pointsymbol* von v sind $\frac{d(d-1)}{2}$-Tupel von Symbolteilen

$$\left(s_1, s_2, \ldots s_{\frac{d(d-1)}{2}}\right)$$

wobei s_i dem Symbolteil p_{iq_i} oder * entspricht. Für jeden Winkel an v wird genau ein Symbolteil wie oben angegeben berechnet. Die Sortierung erfolgt so, dass $p_i \leq p_j$ für $i < j$ und $q_i \geq q_j$ für $i < j$ mit $p_i = p_j$.

Benutzt man zur Berechnung der Symbolteile kürzeste Ringe, nennt man das Tupel Vertexsymbol. Verwendet man Kreise erhält man das Pointsymbol.

Bemerkung 4.8:
In [10] wird für Ecken vom Eckengrad vier eine alternative Sortierung empfohlen. Für diesen Fall ($d = 4$) werden Paare von Symbolteilen gegenüberliegendenr Winkeln aufeinanderfolgend angegeben. Dies macht für den Algorithmus aber keinen Unterschied, und betrifft Ecken mit anderen Graden nicht.

Die Berechnung verläuft in zwei Teilen. Ein Algorithmus berechnet den Symbolteil p_q zu einem gegebenen Winkel. Sind alle Komponenten berechnet, so werden sie sortiert. Zunächst soll erörtert werden wie die einzelnen Komponenten berechnet werden. Dazu dient eine rekursive Funktion.

Algorithmus 4.9:
Folgender Algorithmus gibt p_q für einen gegebenen Winkel $v_0v_1v_2$ bei einer maximalen Suchtiefe $maxdepth$ aus.

$p_q \leftarrow CalcSingleVSPartRec(G = (V, E), P = v_0v_1v_2, maxdepth)$
function CalcSingleVSPartRec($G = (V, E)$,$P = v_0v_1 \dots v_t$,k,$maxdepth$)
  $q \leftarrow 0$
  if k=$maxdepth$ **then return** $*_0$
  end if
  if $v_0v_t \in E(G)$ **then**
    if P ist kein Ring **then return** $0_{maxdepth}$ (*)
    else
      return $(t+1)_1$
    end if
  else
    for $v \in N(v_t) \setminus V(P)$ **do**
      $p'_{q'} \leftarrow$ CalcSingleVSPartRec(G,$P' = Pv$,$k+1$,$maxdepth$)
      if $p' = maxdepth$ **then**
        $q \leftarrow q + q'$
      else
        $maxdepth \leftarrow p'$
        $q \leftarrow q'$
      end if
    end for
    $p \leftarrow maxdepth$ **return** p_q
  end if

end function

Man beachte, dass die mit (*) markierte Zeile bei der Berechnung des Pointsymbols entfällt.

Es ist offenbar wichtig Ringe von Kreisen unterscheiden zu können.

Lemma 4.10:
Ein Kreis $C = v_0 v_1 \dots v_{n-1} a_0$ ist ein Ring genau dann wenn es zwischen v_i und v_j keinen Pfad P gibt mit $L(P) < \min\{|j-i|, n-|j-i|\}$.

Beweis. $\Rightarrow$ Angenommen es gibt einen Pfad P zwischen zwei Ecken v_i, v_j von C mit $L(P) < |j-i|$ und $L(P) < n-|j-i|$. Wir wählen P, v_i, v_j für $0 \leq i < j < n$ o.B.d.A. so, dass $L(P)$ minimal wird (denn dann gilt $A(P) \cap A(C) = \emptyset$). Wir definieren $C_1 := C[v_i, v_j] \cup P$ und $C_2 := C[v_j, v_i] \cup P$. Dann ist $|A(C_1)| = |j-i| + L(P) < |j-i| + n - |j-i| = n$ und $A(C_2) = n - |j-i| + L(P) < n$. Außerdem gilt $C = C_1 \oplus C_2$, C ist also die Summe aus zwei kleineren Kreisen.

$\Leftarrow$ Angenommen C ist kein Ring. Dann gilt $C = C_1 \oplus C_2$ für zwei kleinere Kreise, das heißt $|A(C_1)|, |A(C_2)| < n$. Dann ist $P = A(C_1) \cap A(C_2)$ ein Pfad (denn die Summe der Kreise ist wieder genau ein Kreis) von $v_i \in A(C)$ nach $v_j \in A(C)$. Ohne Einschränkung sei $0 \leq i < j < n$. Damit sind also die Kreise o.B.d.A. $C_1 = C[v_i, v_j] \cup P$ und $C_2 = C[v_j, v_i] \cup P$. Wegen $L(C_1) = |A(C_1)| = |j-i| + L(P) < n$ gilt $L(P) < n - |j-i|$. Analog gilt wegen $L(C_2) = n - |j-i| + L(P) < n$ auch $L(P) < |j-i|$. Da P ein Pfad zwischen v_i und v_j ist, folgt daraus die Behauptung. □

Daraus lässt sich sofort ein Algorithmus ableiten, der von einem gegebenen Kreis entscheidet ob dieser ein Ring ist, indem man für jedes Paar von Kreisecken deren Abstand bestimmt.

Algorithmus 4.11:
Eingabe: Ein Kreis $C = v_0v_1 \dots v_{n-1}v_0$ in G
Ausgabe: *true* falls C ein Ring ist, sonst *false*

for $i = 0 \to (n-3)$ **do**
  for $j = (i+2) \to (n-1)$ **do**
    if $d_G(v_i, v_j) + n - j + 1 < n$ **then return** false
    end if
    if $d_G(v_i, v_j) + i - j < n$ **then return** false
    end if
  end for
end for
return true

Bemerkung 4.12:
Die Korrektheit dieses Algorithmus folgt unmittelbar aus (4.10). Bei der Berechnung der Abstände kann abgebrochen werden, wenn der Abstand größer als $\frac{n}{2}$ ist, da dann mindestens eine der Bedingungen im IF-Block auf jeden Fall erfüllt ist.

Satz 4.13:
Algorithmus (4.11) hat eine Worst-Case-Laufzeit von $\mathcal{O}(n^2 \cdot \Delta_G^{\lceil \frac{n}{2} \rceil})$.

Beweis. Offensichtlich wird die höchste Laufzeit erreicht falls C ein Ring ist. In diesem Fall wird für jedes Paar von Ecken auf dem Kreis C der Abstand bestimmt. Es gibt $\mathcal{O}(n^2)$ solche Paare. Die Bestimmung des Abstands hat nach (2.8) und (4.12) eine Laufzeit von $\mathcal{O}(\Delta_G^{\lceil \frac{n}{2} \rceil})$. Damit ergibt sich insgesamt eine Laufzeit von maximal $\mathcal{O}(n^2 \cdot \Delta_G^{\lceil \frac{n}{2} \rceil})$. □

Satz 4.14:
Der Algorithmus (4.9) berechnet zu einem Winkel $v_0v_1v_2$ die korrekte Komponente des Vertex- bzw. Pointsymbols, falls *maxdepth* ausreichend groß gewählt wird (das heißt $maxdepth \geq q$).

Beweis. Wir betrachten zu Beginn das Pointsymbol. Wir untersuchen zunächst die Korrektheit von p. Sei l die Länge eines kürzesten Kreises $C = v_0v_1v_2 \dots v_{l-1}v_0$ zu dem gegebenen Winkel. Es gelte $maxdepth \geq l$. Der Algorithmus läuft mit einer Tiefensuche durch die Umgebung von v_1. Für $2 \leq i < l - 1$

ist $v_i v_0$ keine Kante des Graphen, also wird die Schleife ausgeführt. Dadurch erreicht der Durchlauf nacheinander die Rekursion zu den Pfaden $v_0 v_1 \dots v_{i+1}$, also schließlich den Pfad $v_0 v_1 v_2 \dots v_{l-1}$. Dort wird l_1 zurückgegeben. In den höheren Iterationen bekommt *maxdepth* nun den Wert l. Da C ein kürzester Kreis ist und in der Schleife stets das kleinste p unter allen Nachbarn an die aufrufende Funktion zurückgeliefert wird, gilt am Ende $p = l$, das heißt p ist korrekt.

Aber auch q ist korrekt: Da *maxdepth* im Verlauf den Wert l annimmt werden sämtliche q' der Teiliterationen mit Ergebnis l'_q aufsummiert. Dabei trägt jeder einzelne Kreis aber nur 1 zu der Summe bei. Also ist q am Ende genau die Anzahl der Kreise mit Länge l.

Beim Vertexsymbol ändert sich hier nichts, es werden lediglich Kreise, die keine Ringe sind, ignoriert, also behandelt wie Pfade, die keine Kreise sind, im Fall der Berechnung des Pointssymbols. □

4.3 Koordinationssequenz

Ein weiteres charakteristisches Symbol einer Ecke stellt die Koordinationssequenz dar. Hier wird untersucht wie viele zusätzliche Ecken mit wachsender Pfadlänge erreichbar sind. Die Nachbarschaft einer Ecke wurde bereits definiert. Hier ist es nützlich diesen Begriff zu verallgemeinern.

Definition 4.15:
Sei $G = (V, E)$ ein Graph, $v \in V$ eine Ecke und $k \in \mathbb{N}_0$. Dann ist $N_k(v) := \{w \in V; dist_G(v, w) \leq k\}$ die *k-te Nachbarschaft* von v.
Insbesondere gilt $N_0(v) = \{v\}$ und zusätzlich sei $N_{-1}(v) = \emptyset$.

Definition 4.16:
Sei $G = (V, E)$ ein Graph und $v \in V$ eine Ecke. Ein Vektor $(n_0, n_1 \dots, n_{t-1}) \in \mathbb{N}^t$ heißt *t-te Koordinationssequenz* von v falls gilt: $n_i = |N_i(v) \setminus N_{i-1}(v)|$ für alle $0 \leq i \leq t - 1$.

Offensichtlich bietet sich eine Breitensuche zur Berechnung der Koordinationssequenz an. Allerdings kann bei periodischen Graphen eine Ecke des Orbitgraphen in mehreren neuen Nachbarschaften auftauchen. Daher ist es nicht möglich diese

Ecken zu steichen falls sie bereits einmal Element einer k-ten Nachbarschaft waren. Stattdessen wird eine Liste von bereits besuchten Ecken des periodischen Graphen geführt. Es sei noch einmal bemerkt, dass eine Ecke von G^∞ genau ein Paar (v, z) ist, wobei v eine Ecke des Orbitgraphen und z ein ganzzahliger Vektor ist.

Der Algorithmus zur Berechnung der Koordinationssequenz ist eine Variante von Algorithmus 2.4, hier kann aber auf das Mitführen der Liste von Vorgängern verzichtet werden. Wegen der Vollständigkeit geben wir diese Variante getrennt an:

Algorithmus 4.17:
Folgender Algorithmus berechnet die k-te Koordinationssequenz einer Ecke v in einem periodischen Graphen G^∞. Dabei ist $S \subset V(G^\infty)$ die Menge der bereits besuchten Ecken und $d : S \to \mathbb{N}$ ordnet diesen Ecken den Abstand von v zu.

Eingabe: Ein Graph G, $v \in V(G)$ und $t \in \mathbb{N}$
Ausgabe: t-te Koordinationssequenz von v in G

$S \leftarrow \{v\}$
$d(v) \leftarrow 0$
for $k = 0 \to t - 2$ **do**
  for all $s \in S, d(s) = k$ **do**
    for all $w \in N(S)$ **do**
      if $w \notin S$ **then**
        $S \leftarrow S \cup \{w\}$
        $d(w) \leftarrow k + 1$
      end if
    end for
  end for
end for
for $k = 0 \to t - 1$ **do**
  $n_k \leftarrow |d^{-1}(k)|$
end for
return $(n_0, \ldots, n_{t-1})$

Satz 4.18:
Algorithmus 4.17 berechnet die Koordinationssequenz korrekt, das heißt $n_i = |N_i(v) \setminus N_{i-1}(v)|$ für alle $0 \leq i \leq t-1$.

Beweis. Sei S bereits wie nach Abschluss des Algorithmus.
Wir zeigen zunächst per Induktion nach k, dass $S_k := \{s \in S \mid d(s) \leq k\} = \bigcup_{i=0}^{k} N_i(v)$.

Für $k = 0$ ist die Aussage klar, denn es gilt $d(v) = 0$ und $d(w) > 0 \; \forall \; w \neq v$.
Sei $S_k = \bigcup_{i=0}^{k} N_i(v)$ für ein beliebiges aber festes $k \in \mathbb{N}$.
Dann ist $S_{k+1} = \bigcup_{i=0}^{k+1} N_i(v)$, denn:
$s \in S_{k+1} \Leftrightarrow s \in S_k \vee s \in N(w)$ für ein $w \in S_k$
$\Leftrightarrow s \in \bigcup_{i=0}^{k} N_i(v) \vee s \in N_{k+1}(v) \Leftrightarrow s \in \bigcup_{i=0}^{k+1} N_i(v)$.

Daraus folgt aber wegen $d^{-1}(k) = S_k \setminus S_{k-1}$ bereits die Behauptung. □

Satz 4.19:
Algorithmus (4.17) hat eine Worst-Case-Laufzeit von $\mathcal{O}(|V|^2 \cdot \Delta_G \cdot t)$.

Beweis. Die äußere Schleife wird $(t-1)$ mal durchlaufen. Die zweite Schleife wird im schlimmsten Fall für ganz $S \subset V$ ausgeführt. Die innere Schleife wird für jeden Nachbarn einer Ecke durchlaufen, also maximal Δ_G mal. Im Schleifenkörper muss mit Aufwand höchstens $|S| \leq |V|$ geprüft werden, ob eine Ecke bereits in S enthalten ist.
Am Ende muss zur Bestimmung der n_i noch einmal S durchlaufen werden. Daher ergibt sich eine Gesamtlaufzeit von höchstens $\mathcal{O}(|V|^2 \cdot \Delta_G \cdot t + |S|) = \mathcal{O}(|V|^2 \cdot \Delta_G \cdot t)$.

□

Bemerkung 4.20:
Da t in der Praxis (so auch bei GTECS) meist eine feste Größe hat, ergibt sich eine Laufzeit von $\mathcal{O}(|V|^2 \cdot \Delta_G)$. Diese Laufzeit kann weiter verbessert werden, indem man geschickte Datenstrukturen verwenden. Verwendet man zum Beispiel einen Baum (anstatt einer Liste) zur Speicherung von S, so kann das Absuchen von S in logarithmischer Zeit erfolgen.

Literaturverzeichnis

[1] K. Lamberts C. Merkens R. Wang U. Englert D. Hons S. Grüter Y. Guo S. Porsche A. Hamacher D. Bündgens and T. Kuhlen. Gtecs3d: A new program for graphtheoretical evaluation of extended network structures. *Zeitschrift für Kristallographie Supplement Issue*, 2012.

[2] Edith Cohen and Nimrod Megiddo. Recognizing properties of periodic graphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1991.

[3] H.S.M Coxeter. *Regular Polytopes*. Dover Publications, 3rd edition, 1973.

[4] K. Iwano and K. Steiglitz. Testing for cycles in infinite graphs with periodic structure. *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 46–55, 1987.

[5] R.A. Nonenmacher. Tiling regular 6-3 hexagonal, 2008.

[6] James B. Orlin. Maximum throughput dynamic network flows. *Mathematical Programming*, 1982.

[7] James B. Orlin. Some problems on dynamic/periodic graphs. *Sloan W.P.*, 1983.

[8] R.W. Simpson. Scheduling and routing models for airline systems. *Report FTL-R68-3, Department of Aeronautics and Astronatics, MIT*, 1968.

[9] K. Iwano und K. Steiglitz. Planarity testing of doubly connected periodic infinite graphs. *Networks 18*, 1988.

[10] M.O'Keeffe V. A. Blatov and D. M. Proserpio. Vertex-, face-, point-, schläfli-, and delaney-symbols in nets, polyhedra and tilings: recommended terminology. *CrystEngComm*, 2010.

[11] L. Thiele W. Backes, U. Schwiegelshohn. Analysis of free schedule in periodic graphs. *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 333–342, 1992.

[12] Egon Wanke. Paths and cycles in finite periodic graphs. *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, 1993.

Versicherung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen verfasst habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in dieser oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht.

Aachen, 22.10.2012
Ort, Datum

Daniel Hons