Die zyklomatische Komplexität

Tobias Konsek

Technische Universität München
Fakultät für Informatik
Boltzmannstraße 3
85748 Garching
tobias.konsek@mytum.de

Abstract: In der folgenden Arbeit wird die zyklomatische Komplexität, eine Softwaremetrik, untersucht, welche Hinweise auf die Komplexität eines Programms liefert. Dabei wird sowohl auf den eigentlichen Ursprung der Metrik, als auch auf die heutige Verwendung und auf weitergehende Verwendungszwecke dieser Softwaremetrik eingegangen.

1 Die Wichtigkeit eines Komplexitätsmaßes

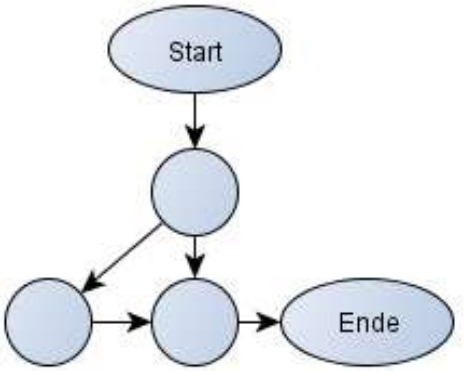
Softwaremetriken sind heute fester Bestandteil einer jeden gewinnorientierten Softwareentwicklung. Der Nutzen, der aus den Ergebnissen dieser Metriken gezogen werden kann, ist in vielen Fällen eine große Hilfestellung und ein Hinweis auf noch zu verbessernde Module oder Prozesse.

Weniger komplexe Programme, die einfach zu warten und zu testen sind, sparen Zeit und damit auch Geld. Die Wiederverwendbarkeit steigt und die Wahrscheinlichkeit, dass das Programm korrekt funktioniert, wird erhöht. Dies liegt darin begründet dass weniger komplexe Programme einfacher zu verstehen sind und so Fehler leichter erkannt beziehungsweise Erweiterungen leichter eingebettet werden können. Die zyklomatische Komplexität kann zu diesen wichtigen Punkten ihren Beitrag leisten. Dies wird im Folgenden gezeigt.

In den folgenden Abschnitten wird zunächst die Bedeutung und Funktionsweise der zyklomatischen Komplexität erläutert. Im Zuge dessen werden verschiedene Möglichkeiten der Berechnung erklärt und der Zusammenhang zur Strukturiertheit eines Programmes erörtert. Am Ende wird noch eine weiterführende Möglichkeit zur Verwendung der Komplexität, nämlich eine durch sie gestützte Testmethode, vorgestellt.

2 Die zyklomatische Komplexität

Zu Beginn des Kapitels werden zunächst noch Grundlagen, die zum weiterführenden Verständnis beitragen, besprochen.




```
private int doSth(int i){
    if(i < 10){
        i = 5;
    }

    i *= 10;

    return i;
}
```

Abbildung 1: Kontrollflussgraph und abgebildeter Code

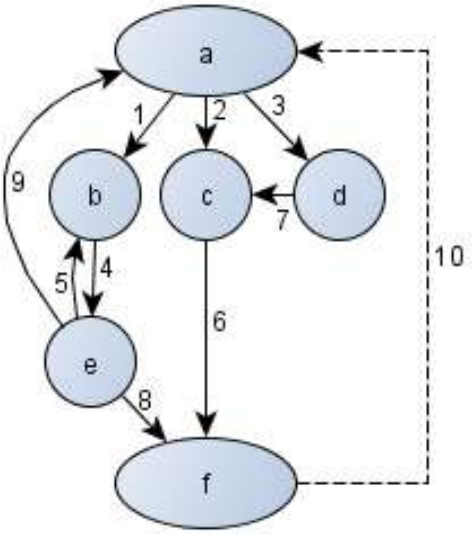
2.1 Grundlagen

Um die zyklomatische Komplexität sinnvoll nachvollziehen zu können, sind einige Grundlagen von Nöten, die im Folgenden erläutert werden.

In Abbildung 1 ist ein Kontrollflussgraph erkennbar. Sein Zweck ist die Abbildung von Programmcode. Bei einem einfachen Graphen wie diesem, der nur aus einer Komponente besteht, handelt es sich um eine einzelne Methode. Wichtig sind der Startknoten, der den Funktionsaufruf und der Endknoten, der die Rückkehr der Funktion abbildet. Des Weiteren sind alle Knoten im Graphen erreichbar und von jedem Knoten aus kann das Ende erreicht werden. Allgemein lässt dich die Überführung von Code zum Kontrollflussgraphen also durchführen, in dem man für jede Anweisung einen Knoten einführt und die logische Abfolge des Codes (z.B. if-Statements) durch Kanten modelliert.

2.2 Die zyklomatische Zahl

Die zyklomatische Zahl berechnet die Anzahl linear unabhängiger Pfade in einem Kontrollflussgraphen. Ausgehend von einer Menge M, die alle möglichen Pfade durch unser Programm enthält[1], ist also genau die Teilmenge T gesucht, die alle Pfade enthält, die nötig sind, um alle anderen Programmabläufe durch Aneinanderreihung und Linearkombination darzustellen.



–	1	2	3	4	5	6	7	8	9	10
$(abefa)$	1	0	0	1	0	0	0	1	0	1
(beb)	0	0	0	1	1	0	0	0	0	0
$(abea)$	1	0	0	1	0	0	0	0	1	0
$(acfa)$	0	1	0	0	0	1	0	0	0	1
$(adcfa)$	0	0	1	0	0	1	1	0	0	1

Abbildung 2: Kontrollflussgraph mit Basismatrix

Die Mächtigkeit dieser Teilmenge T ist gleich der zyklomatischen Zahl dieses Kontrollflussgraphen. Die Mächtigkeit der Basis beschreibt also die Anzahl der Basispfade durch den Kontrollflussgraphen, die nötig ist, um alle anderen Abläufe darzustellen. Abbildung 2 zeigt ein Beispiel eines Kontrollflussgraphen inklusive seiner Basispfade. Die ebenfalls dargestellte Matrix zeigt in den Zeilen die verschiedenen Basispfade und in den Spalte die durchnummerierten Kanten. Eine eins in der Matrix bedeutet also dass der Pfad dieser Zeile die Kante dieser Spalte nutzt.

Die zyklomatische Zahl ist wie folgt definiert:

$$v(G) = e - n + 2p$$

Die Komplexität v eines Graphen G ist definiert durch die Anzahl der Kanten e minus der Anzahl der Knoten n plus der Anzahl der getesteten Komponenten p.

[1] Gemeint sind hier alle Teilpfade, nicht nur Pfade vom Start zum Ende

Die Variable p in der Formel ermöglicht das Testen mehrerer Methoden oder ganzer Programme. Werden also zum Beispiel mehrere Methoden im Programm aufgerufen, kann die Anzahl der Knoten und Kanten zusammengerechnet werden und die Berechnung über die Anpassung der Variablen p modifiziert werden. Die Variable wird dabei gleich der Anzahl unabhängiger Komponenten gesetzt deren Komplexität berechnet wird.

Es lässt sich leicht überprüfen, dass die zyklomatische Komplexität des Graphen aus Abbildung 2, berechnet anhand der Formel, fünf ergibt. Die Mächtigkeit der Basis, ebenfalls in Abbildung 2 dargestellt, ist also gleich der anhand der Formel berechneten Zahl.

2.3 Fundamentale Eigenschaften

Einige grundlegende Eigenschaften der zyklomatischen Komplexität werden im Folgenden erläutert:

i. Die zyklomatische Zahl hat mindestens den Wert eins. Dies ist am einfachsten Graphen mit Start und Endknoten sowie einer Kante leicht nachzuvollziehen.
ii. Einfache Statements ändern die zyklomatische Komplexität nicht. Ein weiterer Knoten im Graphen aus i) bringt ebenfalls eine neue Kante mit sich. Bei Betrachtung der Formel wird klar, dass sich n+1 und e+1 kürzen lassen und sich die zyklomatische Zahl somit nicht ändert.
iii. Die zyklomatische Komplexität ist nur von der Verzweigungstruktur abhängig. Genau genommen führt das Einfügen einer neuen Entscheidung zur Erhöhung der Komplexität um genau eins. Fügt man anstatt des einfachen Knoten in ii) eine Entscheidungsstruktur ein, kann man mit Hilfe der Formel errechnen, dass die zyklomatische Zahl ansteigt. Intuitiv wird dies auch klar, da drei neue Knoten aber vier Kanten eingefügt werden. Anhand der Formel lässt sich dann leicht erkennen, dass dies die zyklomatische Komplexität erhöht.

3. Erfahrung und Anwendung in der Realität

Zu Zeiten als John McCabe die zyklomatische Komplexität veröffentlichte, waren Tools wie „Flow“ (programmiert in APL) für spezielle Plattformen wie den PDP – 10 im Einsatz. Heute haben Analyse- und Metriktools Einzug in viele Entwicklungsumgebungen gefunden. In Eclipse erwirbt man die Funktionalität über das Installieren eines Plugins, während in Visual Studio 2010 diese Werkzeuge bereits fest integriert sind.

Als Grenzwert, an dem ein Entwickler sich im Bezug auf die Komplexität orientieren sollte, hat sich über die Jahre der Wert zehn etabliert, obwohl die Meinungen hier auseinandergehen, da viele Entwickler der Ansicht sind, dass eine Komplexität von zehn, die zum Beispiel neun geschachtelten while-Schleifen entspricht, viel zu hoch sei. Andererseits sind drei Schleifen für eine Matrizenberechnung nichts außergewöhnliches. Werden dann noch in den Schleifen sowie außerhalb einige Entscheidungen (if-Statments) getroffen, kann man auch schnell an den Grenzwert zehn herankommen, ohne eine unlesbare Funktion geschrieben zu haben. Eine deutliche Ausnahme im Bezug auf die Aussagekraft der zyklomatischen Komplexität existiert aber in beinahe jedem Programm. Switch – Statements, die im Auftreten sehr kompakt und meist verständlich sind, treiben die Metrik aufgrund der hier verstecken Entscheidungen schnell nach oben. Ein Switch Case Statement mit sieben Cases beispielsweise hat bereits eine Komplexität von acht, ist aber deswegen in vielen Fällen keineswegs komplex.

4. Vereinfachung der Berechnung

Die in Punkt 2 vorgestellte Formel zur Berechnung der zyklomatischen Komplexität kann auf mehrere Arten vereinfacht werden. Zwei dieser Möglichkeiten werden im Folgenden besprochen.

4.1 Vereinfachung Anhand der Formel

Zum einen kann die Vereinfachung anhand der Formel selbst durchgeführt werden. Hierfür ist zunächst eine Klassifikation der Knoten *n* in 3 Klassen nötig. Es existieren Sammelknoten, die in diesem Fall den Start- und Endknoten repräsentieren. Desweiteren existieren Entscheidungsknoten, die eine Verzweigung darstellen und zu letzt sind natürlich auch einfache Statements in Form von Funktionsknoten vorhanden. Basierend auf dieser Gruppierung lässt sich die Knotenzahl *n* wie folgt berechnen:

$$n = Funktionsknoten + 2 * Entscheidungsknoten + 2$$

Die zwei am Ende repräsentiert den Start- und Endknoten.

Ausgehend von der obigen Definition lässt sich auch die Kantenzahl *e* wie folgt errechnen.

$$e = 1 + Funktionsknoten + 3 * Entscheidungsknoten$$

Setzt man diese beiden Formeln nun unter der Annahme, dass nur eine Komponente getestet wird, also $p = 1$ gilt, in die Formel der zyklomatischen Komplexität ein, ergibt sich nach Vereinfachung die Formel:

$$v(G) = Entscheidungsknoten + 1.$$

Dies stützt auch die Intuition, dass die Komplexität von der Verzweigungsstruktur abhängig ist, da nur die Entscheidungsknoten für die Berechnung mit der neuen Formel betrachtet sind. Der Vorteil an dieser Berechnung ist, dass man direkt vom Programmcode durch Abzählen der Verzweigungen das Ergebnis ablesen kann

4.2 Vereinfachung Anhand des Kontrollflussgraphen

Eine weitere Möglichkeit, die Berechnung zu vereinfachen, wird am Kontrollflussgraphen selbst durchgeführt. Hierfür ist der Eulersche Polyedersatz aus der Graphentheorie von großer Bedeutung. Der Eulersche Polyedersatz besagt, dass die Anzahl der Regionen r in einem planaren Graphen wie folgt berechnet werden kann:

$$r = e - n + 2$$

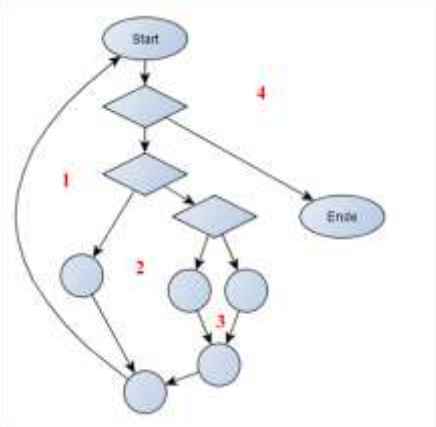


Abbildung 3: Kontrollflussgraph mit Regionen

Diese Berechnung entspricht der Basisformel zur Berechnung der zyklomatischen Komplexität. Abbildung 3 zeigt, dass auch diese Möglichkeiten der Berechnung äquivalent sind. Der Nachteil an dieser Methode liegt allerdings darin, dass der Kontrollflussgraph korrekt umgeformt[2] vorliegen muss

[2] Es ist möglich. dass ein planarer Graph zunächst nicht als solcher erkennbar ist und zuerst umgeformt werden muss.

5. Strukturlose Programmierung

Das Zusammenspiel von Struktur und Komplexität eines Programms wird durch die zyklomatische Komplexität sehr deutlich dargestellt. Ausgehend vom Kontrollflussgraphen lassen sich einige Strukturen zeigen, die die Komplexität eines Programms anheben und die Lesbarkeit beim Testen oder Warten somit erschweren. Wichtig ist hierbei aber das „goto"-Konstrukt[3] das der entscheidende Faktor vieler dieser Muster ist. Dieses Konstrukt findet in modernen Programmiersprachen keine Anwendung mehr. In vielen modernen Programmiersprachen wird es nicht mehr unterstützt, oder falls es doch noch unterstützt wird, wie zum Beispiel in C# findet es keine Verwendung mehr, da es die Lesbarkeit des Codes extrem verschlechtert. Das „goto"- Konstrukt beruht auf der Definition eines Labels an einer beliebigen Stelle im Code. Dieses Label kann dann von einer beliebigen Stelle im Programmcode per *jump Labelname* angesprungen werden. Dadurch ist es zum Beispiel möglich, an eine Stelle in einer Schleife zu springen, auch wenn man sich in einem komplett unabhängigen Codeteil befindet. Folgende Muster, die für eine schlechte Struktur eines Programms stehen, verwenden dieses Konstrukt.

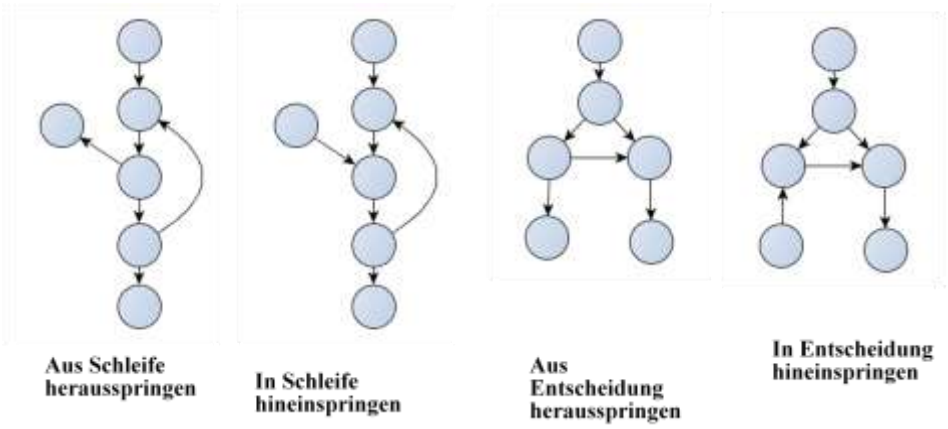


Abbildung 4: Mustergraphen für schlechte Struktur

Die umgangssprachliche Beschreibung der Graphen gibt eine intuitive Vorstellung, welche Konstrukte in Programmcode vermieden werden sollten.

Betrachtet man die obigen Graphen einzeln fällt die Komplexität noch nicht negativ auf. Allerdings kann keiner der in Abbildung 4 gezeigten Mustergraphen alleine auftreten, was in Abbildung 5 gezeigt wird. In allen drei Fällen sind weitere Mustergraphen enthalten.

[3] Manchmal auch jump oder call Konstrukt genannt

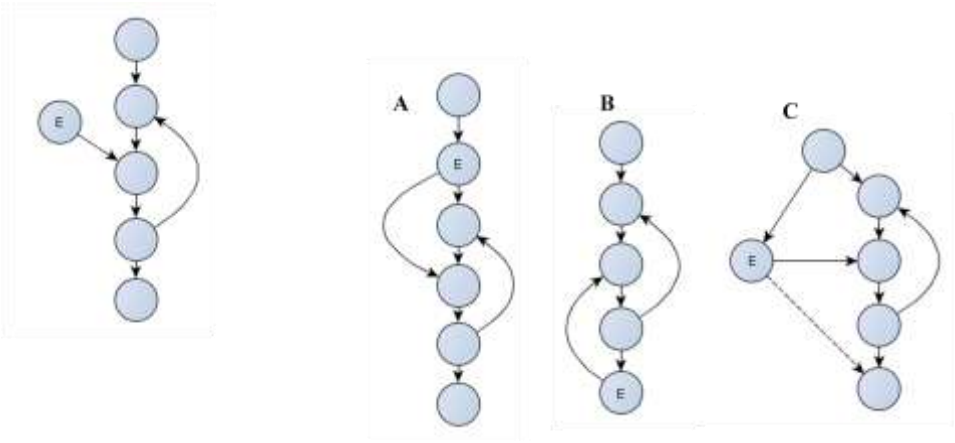


Abbildung 5: Mustergraph und Möglichkeiten des Auftretens

Eine weitere Möglichkeit, unstrukturierten Code zu erkennen, liegt darin, dass ein Kontrollflussgraph, der ein unstrukturiertes Programm abbildet, nicht vereinfacht werden kann. Ein strukturierter Graph hingegen kann soweit vereinfacht werden, bis die zyklomatische Komplexität gleich eins ist. Der Unterschied liegt darin, dass in strukturierten Programmen einheitliche Start- und Endknoten des Graphen beziehungsweise des Subgraphen vorhanden sind. Bei unstrukturierten Graphen ist dies nicht der Fall, weswegen auch, in Programmcode gesprochen, keine Modularisierung möglich ist. Abbildung 6 zeigt einen Schritt der Vereinfachung an einem strukturierten Kontrollflussgraphen.

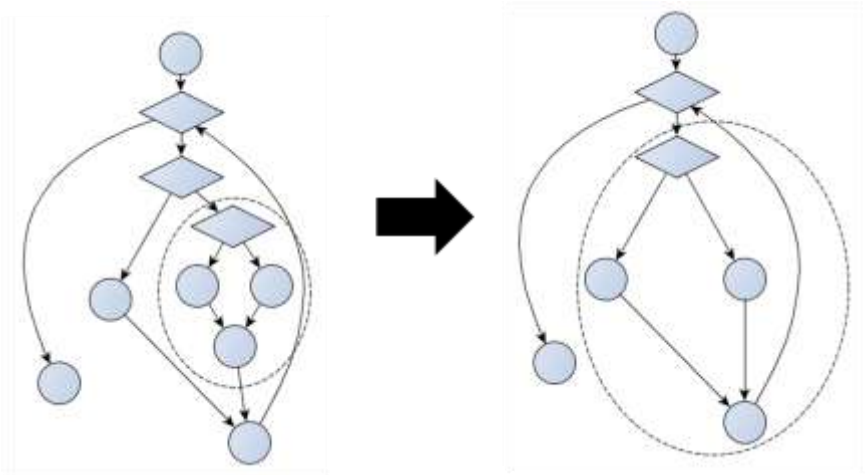


Abbildung 6: Vereinfachung an strukturiertem Graphen

6. Testmethodik

Die zyklomatische Komplexität kann nicht nur zum Errechnen des eigentlichen Metrikwertes verwendet werden. Vielmehr kann sie zum Beispiel genutzt werden, um Testmethoden zu definieren, die sich an ihrem Wert orientieren.

Im Folgenden wird eine solche Methode eingeführt:

Es liegt ein Programm p mit Komplexität v vor. Außerdem liegt die Zahl ac vor, die die Anzahl bisher getesteter Pfade repräsentiert. Solange $ac < v$ gilt, muss eine der drei folgenden Schritte durchgeführt werden, um ein zuverlässiges Testergebnis zu erhalten:

i. Es müssen mehr Pfade getestet werden.
ii. Der Kontrollflussgraph kann vereinfacht werden.
iii. Abhängige Verzweigungen können zusammengefasst werden.

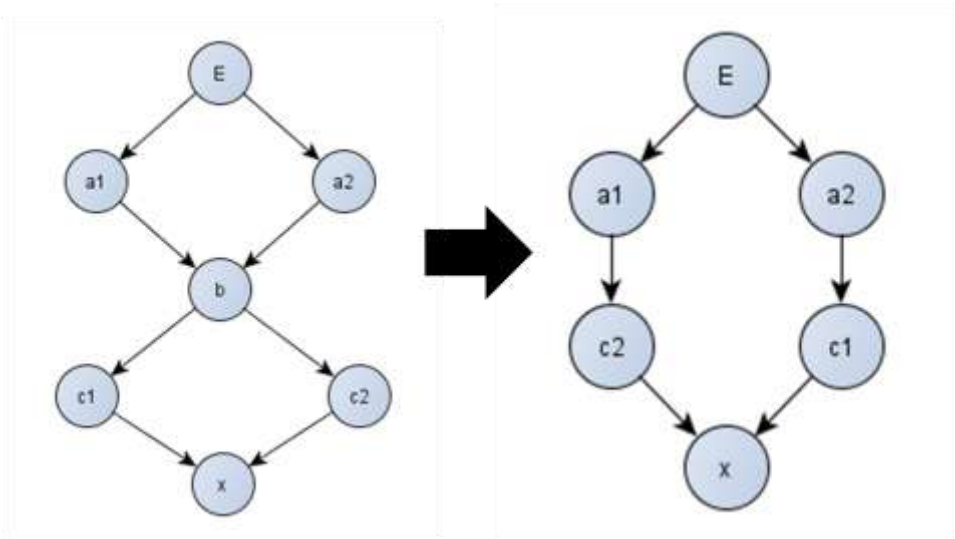


Abbildung 7: Kontrollflussgraph einer zu testenden Methode

Abbildung 7 zeigt einen Graphen mit $ac = 2$ und den getesteten Pfaden [E, a1, b, c2, x] und [E, a2, b, c1, x]. Nun trete der Fall ein, dass [E, a1, b, c1] und [E, a2, b, c2] nicht ausgeführt werden können. Offensichtlich können die Entscheidungen b und E also zusammengefasst werden, wie es in Abbildung 7 auf der rechten Seite zu sehen ist.

Literaturverzeichnis

[Mc76] Thomas J. McCabe: A Complexity Measure: December 1976