Dokumentation der ZEPPELIN-Bibliothek

1. Juli 2020

1 Graphenobjekte

Der Graph wird beschrieben durch Objekte, welche durch gerichtete Abhängigkeiten miteinander verkettet sind. Hierbei gibt es keine Einschränkungen an die Anzahl und Zielobjekte der Verknüpfung. Daher ist der originale Graph zwar gerichtet, kann aber per se nicht näher charakterisiert werden (als Baum, Liste o.ä.).

DependencyObject:

Ein verknüpfungsfähiges Graphenobjekt. Die Klasse hält Zeiger auf andere solche Objekte, welche als Abhängigkeiten gedeutet werden, sowie umgekehrt gerichtete Pointer auf die Adressaten. Mittels der Funktion **dependsOn()** wird eine neue Verknüpfung gesetzt, und zwar vom aktuellen Objekt zu einem anderen. Hierbei werden als Richtung des Graphenkanten nicht der Datenfluss, sondern die Datenanfragen vom abhängigen zum unabhängigen Objekt verstanden.

In Folge der beliebigen Verknüpfungsmöglichkeiten muss nicht nur eine Umstrukturierung sondern auch eine Umgruppierung des Graphen vorgenommen werden, um eine auswertbare Ordnung erzeugen zu können. Zu diesem Zweck können Graphenobjekte gekapselt werden, die entweder eineindeutige (sequentielle) Verknüpfungen untereinander oder zyklische Zusammenhänge aufweisen. Der Graph kann also alternativ als zyklenfreier Graph von gruppierten Modellobjekten begriffen werden.

DependencyGroup:

Kapselt mehrere Modellobjekte, fasst ihre Abhängigkeiten und Adressaten zusammen. Folglich kann die **DependencyGroup** ebenso als Objekt eines gerichteten Graphen gedeutet werden und leitet von **DependencyObject** ab. Durch die Funktion **insert()** kann ein neues Gruppenobjekt hinzugefügt werden. Dabei werden alle Gruppenmitglieder als ungruppierte Objekte abgelegt. Dies bedeutet, das Hinzufügen einer DependencyGroup als Objekt einer anderen Gruppe würde zur Auflösung der hinzugefügten Gruppe führen, eine mehr als einstufige Kapselung ist nicht vorgesehen. Außerdem impliziert diese Einstufigkeit, dass innerhalb von Zyklen stets die größte Menge an zyklisch verknüpften Elementen abgelegt wird (also alle Elemente, die durch die Verknüpfungen untereineinander von sich selbst aus erreichbar sind), analog gilt dies auch für Sequenzen. Zyklische oder sequentielle Gruppen innerhalb von Gruppen müssten anderenfalls zugelassen werden, um einen azyklischen Graphen zu garantieren. Mit **erase()** kann ein Gruppenmitglied wieder entfernt werden. Des Weiteren müssen bei der Erstellung von Gruppen unter Umständen Mengenoperationen wie Zusammenfassen, Schneiden und Komplementbildung ausgeführt werden, die im Fall sequentieller oder zyklischer Gruppen speziellen Regeln genügen.

Wie bereits implizit aufgeführt, ist letztendlich lediglich der geornete Graph von Interesse. Um diesen zu erstellen, muss allerdings zunächst ein zyklenfreier Graphen erstellt werden, also Graphen aus diesen DependencyGroups.

DependencyGraph:

Graph in mehreren Erscheinungsformen. Neben der azyklische Graphenstruktur aus gekapselten Gruppenobjekten (vom Typ **DependencyGroup**/**DependencyObject**) ist der Graph in seiner geordneten Form

(als sortierte Liste von Elementen des Types **DependencyObject** und **ParallelObjects**) abglegt. Der DatenTyp **ParallelObjects** repräsentiert hierbei einen Container parallel liegender Graphenknoten in Form von Objekten oder Gruppen, die später als parallel auswertbare Modellobjekte gedeutet werden können. Es werden stets alle drei Erscheinungsformen in der Speicherstruktur gehalten. Dies bedeutet, direkt bei der Initialisierung des Graphens durch die Graphenobjekte werden mehrere Datencontainer erzeugt: der gruppierte azyklische Graph („*m_ objects*“), der geordnete Graph („*m_ orderedObjects*“) und der geordnete Graph mit parallelen Gruppen („*m_ orderedParallelObjects*“). Hierfür werden nacheinander Funktionen für die Objektgruppierung **clusterGraph()** und die Graphensortierung **orderGraph()** aufgerufen, welche die zugehörigen Algorithmen enthalten.

2 Cluster- und Sortieralgorithmus

Die Ordnung des Graphen erfolgt hierarchisch in drei Schritten: die Gruppierung für zyklische und sequentielle Zusammenhänge, die Bestimmung paralleler Gruppen als Übereinheit, und die Bestimmung einer Auswertungsreihenfolge von parallelen Gruppen durch Einbettung in eine geordnete Liste.

Für die Clusterung werden zunächst alle sequentiellen Zusammenhänge bestimmt, welche durch eineindeutige Verknüpfungen zueinander charakterisiert und sofort durch Verfolgung der Abhängigkeiten durch den ungeordneten Graphen auffindbar sind. Konkret werden für jeden ungeprüften Knoten die in einer Sequenz verbundenen Adressaten und Abhängigkeiten bestimmt und in Gruppen zusammengefasst. Einzelne, nicht in Sequenzen zusammengefasste Objekte bilden hierbei einelementige Gruppen. Nach diesem ersten Schritt können alle Graphknoten durch disjunkte ein- oder mehrelementige Objektgruppen ersetzt werden.

Zyklische Zusammenhänge sind gegenüber sequentiellen Zusammenhängen übergeordnet. Dies bedeutet, Zyklen können Sequenzen enthalten, Sequenzen allerdings keine Zyklen. Daher ist die Zyklensuche algorithmisch nach der Sequenzensuche durchzuführen. Bei der Suche werden zu jedem ungeprüften Knoten alle vorhandenen Abhängigkeiten rekursiv Ast für Ast durchlaufen und die zugehörigen Knoten markiert. Ein Zyklus bildet einen Schluss der markierten Knotenkette und kann einfach erkannt werden. Allerdings müssen nicht alle Zyklen disjunkt sein, beispielsweise können größere Zyklen kleinere enthalten. Deshalb muss nach Abschluss der rekursiven Suche von jedem Startknoten eine Zusammenfassung aller gefundenen Gruppen erfolgen. Alle Graphenknoten, die in Zyklen enthalten sind, sind entsprechend zu ersetzen, so dass der Graph am Ende wiederum nur disjunkte Gruppen enthält.

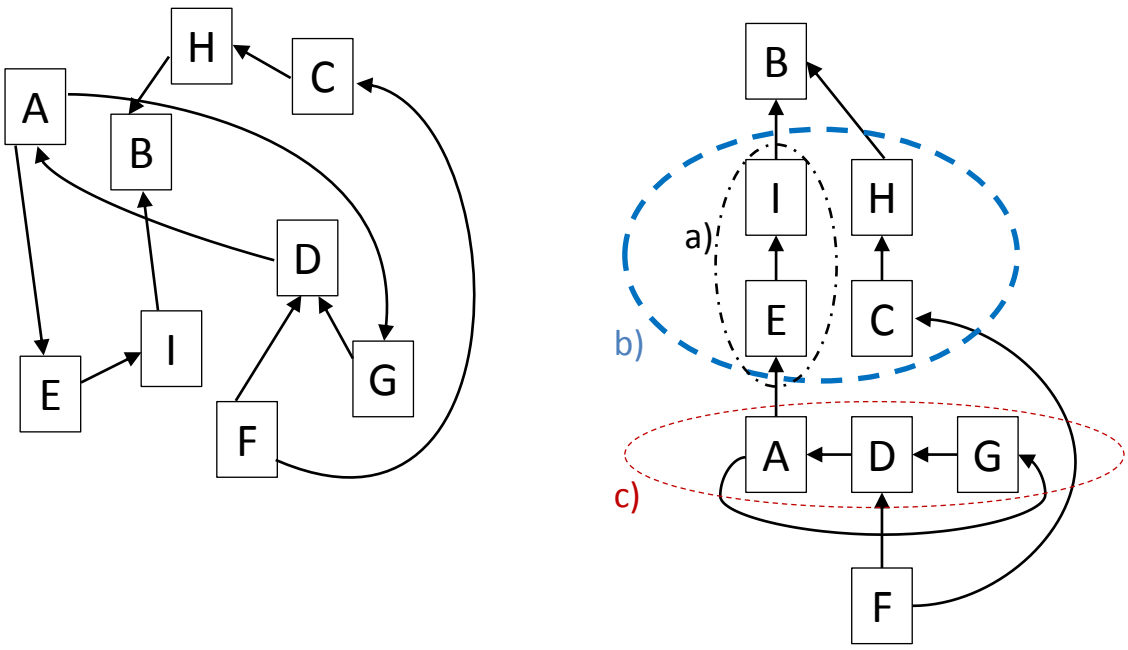

Abbildung 1: Graphenerscheidnungsformen für Beispiel 1 in ZeppelinTest: ungeordneter Graph (links), gruppierter und geordneter Graph (rechts)

Parallel zur Ersetzung der Graphenobjekte durch übergeordnete Gruppenstrukturen werden alle neu entstandenen Graphenkanten zwischen verschiedenen Objektgruppen additiv den Objektabhängigkeiten hinzugefügt.

Die Auswertung paralleler Gruppen und die Ordnung des Graphen erfolgt gleichzeitig. Beginnend bei den Knotenenden, also Graphenobjekten ohne Abhängigkeiten zu anderen Knoten, kann der gruppierte Graph schrittweise abgetragen werden. Parallele Gruppen sind hierbei Endknoten desselben Teilgraphen, die folglich nicht voneinander abhängen und parallel ausgewertet werden können. Bei Abtragung der Knotenenden eines jeden Teilgraphen werden die parallelen Gruppen in einen Container gestapelt und bilden nach Abschluss der Funktion den geordneten Graphen.

Das Beispiel 1, sowie ein weiteres Beispiel 2 sind Bestandteil des Testprojektes *ZeppelinTest*, dokumentiert unter */doc/example1* und */doc/example2*. In beiden Testvarianten wird ein ungeordneter Graph eingelesen, gruppiert und geordnet ausgegeben.

3 Erweiterung für die Modellberechnungen am Beispiel der Gebäudesimulationsplattform

Um die Graphenalgorithmen für eine Modellberechnung nutzen zu können, müssen die zunächst nur als Knoten interpretierten Objekte um eine Modellberechnungsfunktionalität erweitert werden. Mit Hilfe der Graphenalgorithmen kann dann eine Modellordnung erzeugt werden, welche die richtige Reihenfolge bei Auswertung der Modelle für geänderte Randbedingungen garantiert. Prinzipiell ist für Modellobjekte, deren Ergebnis von den Berechnungen anderer Modellgrößen abhängt, die Vererbung aus der *DependencyObject*-Klasse sinnvoll.

Die Entscheidung, welche Graphen erzeugt und mit welchen konkreten Objekten befüllt werden, birgt allerlings zusätzliche Freiheit. Am Beispiel der Gebäudesimulation wird deutlich, dass es sowohl Größen gibt, welche sich nur in Abhängigkeit der Zeit ändern und für alle davon abhängigen Berechnungsgrößen eine Aktualisierung anfordern. Als rein zeitabhängig sind Größen wie das Außenklima oder ein Nutzerverhalten zu deuten, welches durch einen Zeitplan beschrieben ist. Eine Aktualisierung der abhängigen Größen wie die Wärmeströme innerhalb der Außenwände müßte hier bei Änderung eines jeden Zeitschrittes erfolgen. Dasselbe gilt für rein zustandsabhängige Größen, welche sich ändern, sobald ein übergeordnetes Lösungsverfahren einen neuen Lösungsvorschlag erstellt. Im Fall der Gebäudesimulation sind dies Raum und Wandtemperaturen, und eine Änderung des Lösungsvorschlages würde eine Vielzahl an Modellberechnungsgrößen verändern wie zum Beispiel die Empfindungstemperatur, Heizregelgrößen und Wärmeströme.

Diese Überlegungen führen dazu, dass die Modelle mit zwei verschiedenen Berechnungsfunktionalitäten ausgestattet sein müssen: eine Aktualisierung aller Modellgrößen zu Änderung des Zeitschrittes und eine Aktualisierung aller Modellgrößen bei Änderung des globalen Lösungsvorschlages durch das umliegende Zeitintegrationsverfahren oder nichtlineare Lösungverfahren. Demzufolge ist es sinnvoll, zwei unterschiedliche Graphen aufzubauen, je einen für jede Berechungsfunktionalität.

Des Weiteren kann es vorkommen, dass die Ordnung von Modellobjekten in der Berechnungsreihenfolge bekannt ist. Dies trifft für von anderen Ergebnissen unabhängige Modellberechnungen zu, sowie implizit bekannte Modelle. Bei der Gebäudesimulationsplattform ist dieser Aspekt wesentlich, da hier eine Kombination vordefinierter und nutzerdefinierter Modelle vorliegt. Dies bedeutet, Teilkomponenten innerhalb des Graphen und ihre Verknüpfungen untereinander sind bereits bekannt wie zum Beispiel die Wärmestromberechnung durch Wände und die Auswertung der Innen- sowie Außenrandbedingungen. Für diese Teilgraphen ist die Berechnungsreihenfolge ersichtlich und kann ohne Bemühung der Graphenalgorithmen angegeben werden. Es kann also ein Graph allein aus nutzerdefinierten Modellobjekten erstellt werden, auch wenn diese Verknüpgungen zu vordefinierten Modellergebnissen aufweisen. Die Erzeugung von Teilgraphen mit Verknüpfungen zu Objekten außerhalb des Graphen ist explizit erlaubt.

Die Initialisierung des Graphen erzeugt aus den Modellobjekten eine Liste von Gruppen (*DependencyGroup*), die als Argument zurückgegeben werden. DesWeiteren ist der gruppierte Graph in geordneter Form als *orderedObjects()* oder *parallelOrderedObjects()* abrufbar. Zu beachten ist dabei allerdings, dass die Berechnungsvorschrift der einzelnen Modellobjekte unzureichend ist, um den gruppierten Graph auszuwerten. Insbesondere für Modellobjekte, die in einer zyklischen Abhängigkeit stehen, ist die Implementierung der Auswertungsfunktionalität der zugeordneten Gruppe unumgänglich. Zu jeder DependencyGroup muss daher ein eigenes Modellobjekt (abgeleitet aus *DependencyObject*) erstellt werden, welches die Modellberechnungen der ent-

haltenen Modellobjekte koordiniert, wie es beispielsweise bei der Gebäudesimulation der Fall ist. In diesem Anwendungsfall hat sich für zyklische Modellgruppen eine Auswertung durch das Newton-Verfahren bewährt.

Zur globalen Auswertung der Modellgruppen kann der geordnete Graphen genutzt werden (Zugriff über *orderedObjects()* oder *parallelOrderedObjects()*), wobei die Modellberechnung für geordnete Ebenen nacheinander, für parallele Objekte gleichzeitig durchgeführt werden kann. Dabei entspricht die Listenreihenfolge einem auswertbaren Stapel, das erste Element ist also stets das zuerst auswertbare, das letzte Element muss zu Ende der Modellberechnung aktualisiert werden.