Entwurf einfacher Graphalgorithmen

1 Motivation

Die konstruktive Induktion ist eine Methode zur systematischen Entwicklung von Algorithmen. Die Kernidee ist, eine Lösung für Probleminstanzen der Grösse n auf Instanzen der Grösse n+1 zu erweitern. Dadurch ergeben sich in natürlicher Weise rekursive Algorithmen.

In dieser Unterrichtssequenz entwickeln wir Algorithmen, bei denen die konstruktive Induktion zu Lösungen führen kann, die nicht für alle Probleminstanzen korrekt sind. Dies ist vor allem dann der Fall, wenn die Induktion nicht wohlfundiert ist. Wir nutzen diese Erkenntnis, um die entworfenen Algorithmen so zu erweitern, dass sie für alle Probleminstanzen funktionieren.

Wir studieren zu diesem Zweck zwei einfache Graphalgorithmen. Bei diesen funktionieren naive rekursive Implementierungen nur für azyklische Graphen, terminieren aber möglicherweise nicht für zyklische Graphen. Um eine Anbindung an frühere Unterrichtseinheiten zum Programmieren mit Python zu schaffen, motivieren wir die ausgewählten Graphalgorithmen über die Speicherbereinigung (Garbage Collection) moderner Programmiersprachen.

2 Lernziele

Das Ziel dieser Unterrichtssequenz ist es, die Schülerinnen und Schüler (SuS) mit der ihnen bereits bekannten Methode der konstruktiven Induktion noch vertrauter zu machen und somit ihre Kenntnisse zu vertiefen. Die SuS sollen verstehen, dass die konstruktive Induktion voraussetzt, dass man die Grösse einer Probleminstanz klar definiert hat. Anhand von konkreten Beispielen lernen sie Problemstellungen kennen, bei denen eine naive Anwendung der Methode fehlschlägt, und sehen, wie man diese korrekt nutzen kann.

3 Kontext

Die Unterrichtssequenz richtet sich an Schülerinnen und Schüler im Ergänzungsfach Informatik im 11. und 12. Schuljahr. Sie steht am Ende einer Unterrichtsreihe zum Thema *Entwurf von Algorithmen mittels konstruktiver Induktion*.

Als Vorwissen aus der Informatik und Mathematik wird Folgendes vorausgesetzt:

- Die SuS kennen die mathematische Induktion als Beweisprinzip und wissen, dass die Induktion aus einem Induktionsstart und einem Induktionsschritt besteht.
- Sie kennen die Methode der konstruktiven Induktion und haben diese anhand einfacher Beispiele eingeübt.
- Die SuS sind vertraut mit Graphen und kennen verschiedene Arten gerichteter Graphen, insbesondere Bäume, azyklische Graphen und zyklische Graphen.
- Sie haben Programmiererfahrung in Python; insbesondere kennen sie:
  - Objekte, die durch Referenzen miteinander verbunden sind,
  - globale Variablen,
  - rekursive Methoden und
  - gängige Datenstrukturen wie Mengen und Dictionaries (assoziative Listen).

4 Umsetzung

Die Unterrichtssequenz ist konzipiert für eine Doppellektion und besteht aus Unterrichtsgesprächen im Plenum sowie Aufgaben, die von den SuS alleine oder in kleinen Gruppen bearbeitet werden können. Sie ist in zwei Phasen gegliedert:

1. Phase I entwickelt einen Algorithmus zur Bestimmung der erreichbaren Knoten in einem (gerichteten) Graphen. Diese Phase wendet die konstruktive Induktion an, um zunächst einen Algorithmus für baumartige Graphen zu entwickeln, und erweitert diesen dann auf allgemeine Graphen.
2. Phase II entwickelt einen Algorithmus zum Kopieren der erreichbaren Knoten in einem (gerichteten) Graphen. Der Ansatz aus Phase I reicht aus, um die Terminierung des Kopier-Algorithmus sicherzustellen, muss aber erweitert werden, damit der Algorithmus korrekt ist.

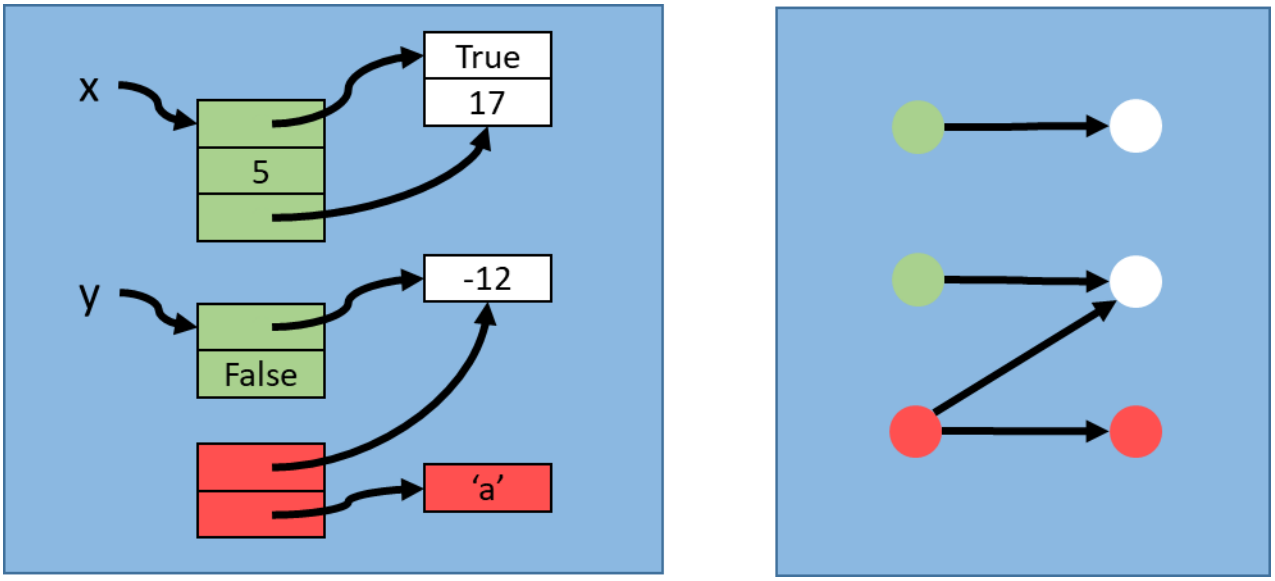
Phase I ist in sich abgeschlossen und kann somit ohne Anbindung von Phase II unterrichtet werden.

5 Unterrichtssequenz

5.1 Einführung

Viele moderne Programmiersprachen wie z.B. Python und Java bieten eine automatische Speicherbereinigung (Garbage Collection). Diese durchläuft periodisch den Heap-Speicher des Programms und entfernt automatisch alle Objekte, die vom Programm nicht mehr benötigt werden. Somit müssen sich Programmiererinnen und Programmierer nicht darum kümmern, Objekte zu löschen und ihren Speicher freizugeben (wie z.B. in C und C++), was die Programmierung erleichtert und Fehler vermeidet.

Um festzustellen, welche Objekte vom Programm noch benötigt werden, betrachtet die Speicherbereinigung den Heap-Speicher als Graphen, dessen Knoten die Objekte im Speicher sind und dessen gerichtete Kanten die Referenzen zwischen diesen Objekten. Eine *Wurzel* in diesem Graphen ist ein Objekt, zu dem eine Referenz in einer lokalen Variable des Programms gespeichert ist. Ein Objekt wird weiterhin benötigt, wenn es im Graph von einer Wurzel aus erreichbar ist, d.h., wenn es einen (möglichweise leeren) Pfad von einer Wurzel zu diesem Objekt gibt. Alle anderen Objekte sind Müll und können entfernt werden. Das folgende Diagramm veranschaulicht diese Konzepte:



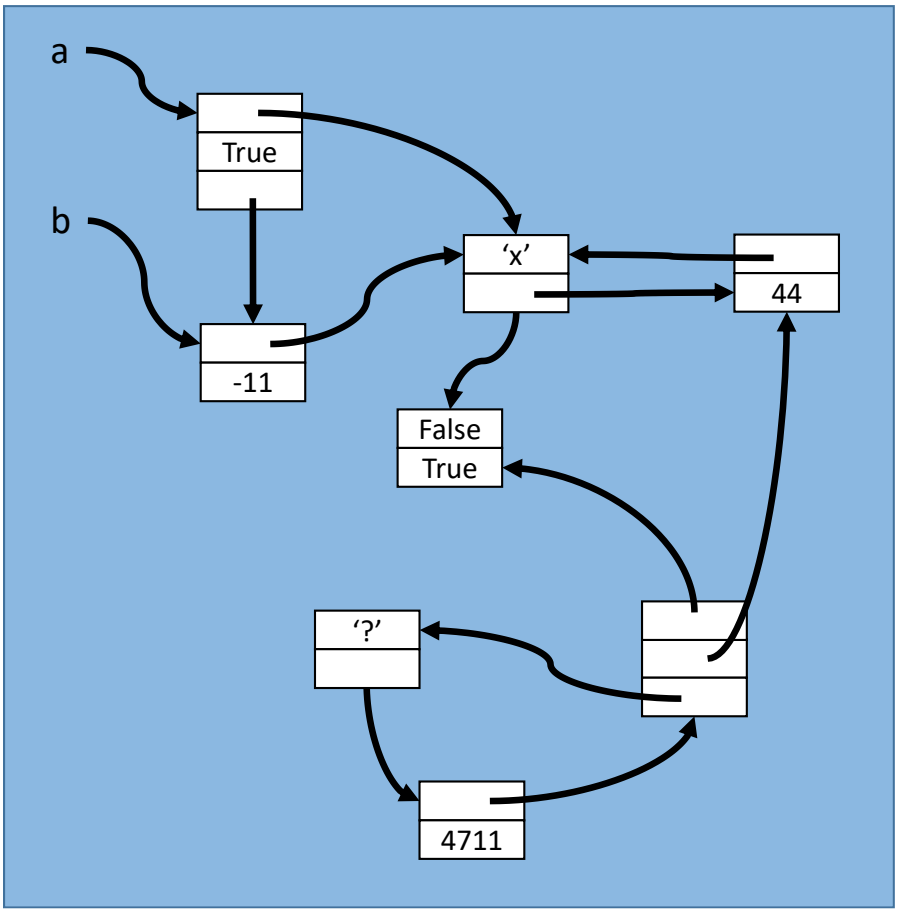
Auf der linken Seite sehen wir einen Programmspeicher mit den lokalen Variablen x und y sowie sechs Objekten. Auf der rechten Seite befindet sich der zugehörige Graph. Wir verwenden grün für Wurzeln und weiss für Objekte, die von Wurzeln aus erreichbar sind, aber selbst keine Wurzeln sind. Rot steht für Objekte, die nicht erreichbar sind und daher gelöscht werden können. Objektfelder, die keine Referenzen enthalten (sondern boolesche Werte, Zahlen, Zeichen, etc.), spielen für die Speicherbereinigung keine Rolle.

In dieser Unterrichtssequenz entwickeln wir die Graphalgorithmen, die von der Speicherbereinigung benutzt werden, um zu ermitteln, welche Objekte gelöscht

werden können. Dafür gehen wir davon aus, dass Graphen wie folgt dargestellt werden. Ein Graph G=(N,E) besteht aus einer Menge N von Knoten sowie einer Menge E von Kanten. Für eine Kante (n,m)∈E nennen wir m einen *Nachfolger* von n.

Aufgabe 1:

Gegeben ist der nachfolgende Programmspeicher. Konstruieren Sie den zugehörigen Graphen. Welche Knoten sind Wurzeln, welche sind erreichbar und welche sind nicht erreichbar? Färben Sie die Knoten entsprechend nach dem obigen Schema.



Um Knoten eindeutig identifizieren zu können, weisen wir jedem Knoten eine eindeutige Bezeichnung in Form einer Zahl zu. Das können wir z.B. dadurch erreichen, dass wir für N eine Teilmenge der natürlichen Zahlen verwenden (wie schon in der Lösung zu Aufgabe 1). Für die Anwendung in der Speicherbereinigung können wir uns vorstellen, dass diese Zahl die Adresse des Objektes im Heap-Speicher ist.

Aufgabe 2:

Beschreiben Sie den Graphen aus der Lösung zu Aufgabe 1 formal als Menge von Knoten und Menge von Kanten.

5.2 Phase I: Erreichbarkeit in Graphen

Eine einfache Methode der Speicherbereinigung ist, periodisch zu bestimmen, welche Knoten im Graphen von den Wurzeln aus erreichbar sind, und alle nicht-erreichbaren zu löschen. In den einfachen Beispielgraphen, die wir bisher betrachtet haben, kann man oft auf Anhieb sehen, welche Knoten erreichbar sind. Die Speicherbereinigung verfügt aber nicht über eine derartige «globale» Sicht auf den Speicher; sie muss sich von den Wuzeln aus Schritt für Schritt vorarbeiten, um die erreichbaren Knoten in einem Graphen zu finden, der oft zehntausende Knoten enthält. Unser Ziel ist es, einen entsprechenden Algorithmus für die Erreichbarkeit in Graphen zu entwickeln. Dafür verwenden wir die Methode der konstruktiven Induktion, die wir in den vorherigen Unterrichtseinheiten kennengelernt haben.

Der Einfachheit halber gehen wir davon aus, dass wir nur eine Wurzel haben; eine Verallgemeinerung auf mehrere Wurzeln ist einfach. Wir suchen nun folgenden Algorithmus und gehen davon aus, dass der Graph G=(N,E) in einer globalen Variable gespeichert ist, sodass wir ihn nicht als Parameter übergeben müssen:

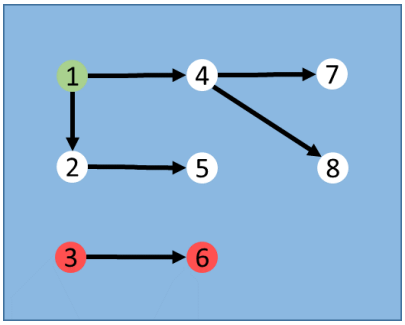
Algorithmus REACH(n)

Eingabe: Graph G=(N,E), Knoten $n \in N$ (die Wurzel)

Ausgabe: Die Menge $R \subseteq N$ der Knoten, die von n aus erreichbar sind

In einer Klassendiskussion erarbeiten wir einen Algorithmus anhand des (baumartigen) Graphen rechts. Für einen solchen Graphen liegt es nahe, die Problemgrösse über die Höhe eines Knotens im Graphen zu definieren:

$$hoehe(n) = \begin{cases} 0, falls\ (n,m) \notin E\ für\ alle\ m \\ \max(\{hoehe(m) | (n,m) \in E\}) + 1, sonst \end{cases}$$



Aufgabe 3:

Berechnen Sie für jeden Knoten in unserem Beispielgraphen dessen Höhe. Wie ändert sich die Höhe der einzelnen Knoten, wenn man Knoten 5 aus dem Graph entfernt?

Mit dieser Definition der Problemgrösse können wir die konstruktive Induktion verwenden und erhalten folgende erste Version des Algorithms:

Algorithmus REACH(n)

R := { n }
Für jeden Knoten m, sodass (n.m) $\in$ E:
    R := R $\cup$ REACH(m)
Gib R als Ergebnis zurück.

Hier besprechen wir, dass der Fall, dass n keine Nachfolger hat (Induktionsstart) implizit dadurch abgedeckt ist, dass die Schleife über die leere Menge iteriert, also nichts tut (und insbesondere keinen rekursiven Aufruf durchführt).

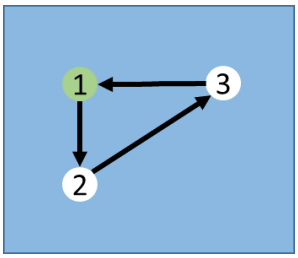
Aufgabe 4:

Führen Sie den Algorithmus REACH auf unserem Beispielgraphen aus. Geben Sie an, welches Ergebnis REACH(1) liefert, und beschreiben Sie die gesamte Abfolge rekursiver Aufrufe und ihrer Ergebnisse.

Im Sinne des Prinzips „Learning from Failures" betrachten wir nun ein Beispiel, für das unser Algorithmus nicht funktioniert.

Aufgabe 5:

Führen Sie den Algorithmus REACH auf dem Graphen rechts für n=1 aus. Welches Ergebnis erhalten Sie?



Wir stellen fest, dass REACH auf zyklischen Graphen nicht terminiert. Generell ist es wünschenswert, dass der Algorithmus auf allen Eingaben terminiert; insbesondere können in der Speicherbereinigung durchaus zyklische Graphen vorkommen, z.B. wenn der Heap-Speicher eine doppelt verkettete Liste enthält.

In einem Unterrichtsgespräch ergründen wir, warum die Methode der konstruktiven Induktion uns in diesem Beispiel zu einem Algorithmus geführt hat, der nicht terminiert. Dabei erkennen wir, dass unsere Problemgrösse hoehe(n) für zyklische Graphen nicht wohldefiniert ist. In unserem Beispiel gibt es keinen Knoten n ohne Nachfolger, sodass die rekursive Definition keinen Startpunkt hat (d.h. nicht wohlfundiert ist).

Aufgabe 6:

Wie würden Sie vorgehen, um im obigen zyklischen Graphen von Hand die erreichbaren Knoten zu bestimmen? In diesem kleinen Beispiel sieht man die Lösung sofort. Wie würden Sie das Problem in einem grossen Graphen lösen?

Wir stellen fest, dass der Algorithmus einen Knoten nur dann besuchen muss, wenn er nicht bereits in der Menge R der erreichbaren Knoten enthalten ist. Somit vergrössert jeder rekursive Aufruf des Algorithmus die Menge R; umgekehrt wird die Anzahl |N \ R| der noch nicht besuchten Knoten mit jedem Aufruf echt kleiner. Wir können diese Idee verwenden, um als Problemgrösse |N \ R| zu wählen, was uns zu folgendem Algorithmus führt. In diesem ist R eine globale Variable, die zu Beginn die leere Menge enthält.

Algorithmus REACH2(n)

R := R ∪ { n }
Für jeden Knoten m, sodass (n,m) ∈ E und m ∉ R:
    REACH2(m)

Hier sehen wir, dass die Schleife nur für Nachfolger von n durchlaufen wird, die nicht bereits besucht wurden. Daher wird mit jedem rekursiven Aufruf die Problemgrösse | N \ R | echt kleiner. Da dies nicht unendlich oft passieren kann, muss der Algorithmus terminieren.

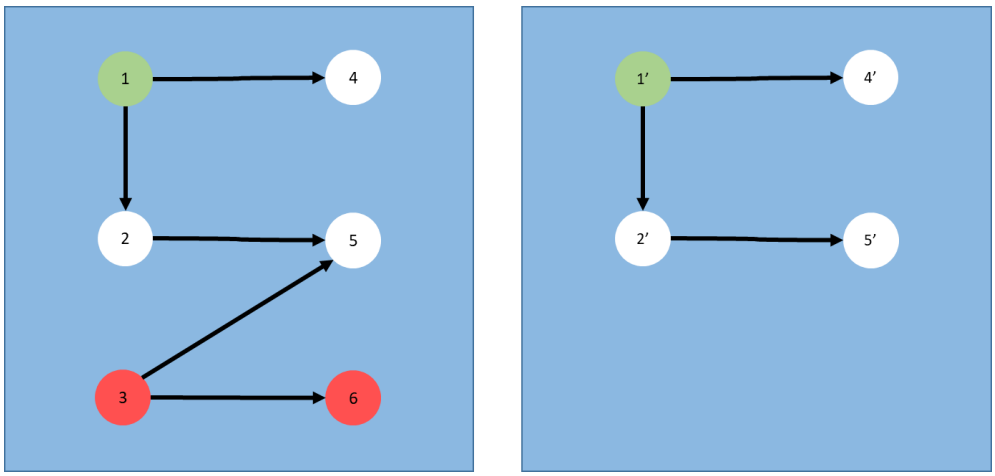
Aufgabe 7:

Führen Sie REACH2 auf unserem zyklischen Beispielgraphen aus. Geben Sie dabei für jeden Knoten n∈{1, 2, 3} an, welches Ergebnis REACH2(n) liefert.

5.3 Phase II: Kopieren von erreichbaren Teilgraphen

Eine andere Art der Speicherbereinigung (Generational Garbage Collection) kopiert regelmässig alle erreichbaren Objekte aus einem Speicherbereich für junge Objekte in einen Speicherbereich für ältere Objekte. Im Folgenden entwickeln wir einen Algorithmus für das Kopieren von Graphen.

Das Ziel unseres Algorithmus ist es, eine Kopie der erreichbaren Objekte eines gegebenen Graphen zu erstellen, d.h. einen Graphen mit der gleichen Form, aber neuen Knoten. Das folgende Diagramm zeigt die Eingabe (links) und Ausgabe (rechts) des Algorithmus. Die roten Knoten wurden nicht kopiert, weil sie nicht von der Wurzel aus erreichbar sind. Die Knoten im Ergebnis-Graphen haben andere Bezeichnungen. Wir schreiben k‘ für die Kopie von Knoten k; die konkreten Zahlwerte sind irrelevant, solange sie eindeutig sind.



Wir suchen somit folgenden Algorithmus:

Algorithmus COPY(n)

Eingabe: Graph G=(N,E), Knoten n∈N (die Wurzel)
Ausgabe: Eine Kopie G‘=(N‘,E‘) des Teilgraphs von G, der von n aus erreichbar ist, und die Kopie n‘ der Wurzel n

Wir gehen davon aus, dass G und G‘ in globalen Variablen gespeichert werden, sodass wir auf sie zugreifen können, ohne sie als Parameter und Ergebnis herumzureichen.

Aufgabe 8:

Entwickeln Sie auf der Basis von REACH2 einen Algorithmus COPY und führen Sie Ihren Algorithmus auf dem obigen Beispielgraphen aus.

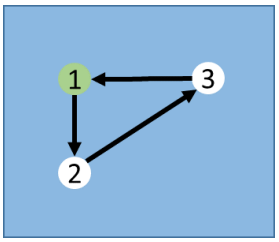
Basierend auf den Vorschlägen der SuS zu Aufgabe 8 entwickeln wir im Unterrichtsgespräch diesen Algorithmus. Die folgende Version enthält die Menge R der erreichbaren Knoten um sicherzustellen, dass der Algorithmus auch für zyklische Graphen terminiert. Sollten die SuS Algorithmen vorschlagen, die nicht immer terminieren, so können wir dies hier analog zu Phase I thematisieren und korrigieren. N‘ und E‘ sind zu Beginn leer.

Algorithmus COPY(n)

R := R ∪ { n }
Wähle n‘, sodass n‘ ∉ N ∪ N‘.
N‘ := N‘ ∪ { n‘ }
Für jeden Knoten m, sodass (n,m) ∈ E und m ∉ R:
    m‘ := COPY(m)
    E‘ := E‘ ∪ { (n‘,m‘) }
Gib n‘ als Ergebnis zurück.

Aufgabe 9:

Führen Sie den Algorithmus COPY auf dem Beispielgraphen rechts für n=1 aus. Welches Ergebnis erhalten Sie?



Wir beobachten, dass der Algorithmus für zyklische Graphen zwar terminiert, aber ein inkorrektes Ergebnis liefert: Die Kante, welche den Zyklus schliesst (in unserem Beispiel von Knoten 3‘ zu 1‘) fehlt! In einem Unterrichtsgespräch erarbeiten wir, dass es weder korrekt ist, Nachfolger von n, die bereits in R sind, einfach zu überspringen (wie im obigen COPY-Algorithmus) noch das Kopieren fortzusetzen (was zur Nicht-Terminierung führen würde). Stattdessen muss sich der Algorithmus „merken“, welcher Knoten die Kopie von Knoten 1 ist und diesen als Nachfolger der Kopie von Knoten 3 verwenden. Um das zu erreichen, ersetzen wir in unserem Algorithmus die Menge R der besuchten Knoten durch ein Dictionary (eine assoziative Liste), die Knoten in G auf ihre Kopie in G‘ abbildet. Die Schlüsselmenge dieses Dictionary ist genau die Menge R. D ist zu Beginn leer.

Algorithmus COPY2(n)

```
Wähle n‘, sodass n‘ ∉ N ∪ N‘
N‘ := N‘ ∪ { n‘ }
D := D ∪ { (n, n‘) }
Für jeden Knoten m, sodass (n,m) ∈ E:
    Falls m ∉ keys(D):
        m‘ := COPY2(m)
    sonst:
        m‘ := D[m]
    E‘ := E‘ ∪ { (n‘,m‘) }

Gib n‘ als Ergebnis zurück.
```

Aufgabe 10:

Führen Sie COPY2 auf unserem zyklischen Beispielgraphen aus. Welches Ergebnis erhalten Sie?

Aufgabe 11:

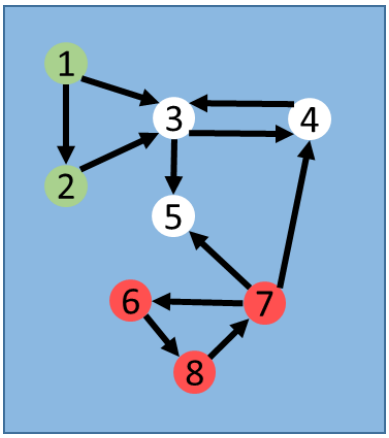
Finden Sie einen Graphen, der nicht zyklisch ist, für den COPY aber trotzdem das falsche Ergebnis liefert. Arbeitet COPY2 für diesen Graphen korrekt?

6 Zusammenfassung und Ausblick

In früheren Unterrichtseinheiten haben wir die konstruktive Induktion als Methode zur Entwicklung von Algorithmen kennengelernt. Die Beispiele in der vorliegenden Unterrichtssequenz haben gezeigt, dass man bei der Anwendung der konstruktiven Induktion darauf achten muss, eine sinnvolle Definition der Problemgrösse zu haben. Genau wie die mathematische Induktion verlangt die konstruktive Induktion, dass die Problemgrösse schrittweise kleiner wird, ohne aber unendlich oft kleiner werden zu können. Aus diesem Grund ist z.B. der Abstand eines Knotens von den Blättern eines Graphen keine geeignete Problemgrösse, weil dieser Abstand in zyklischen Graphen unendlich gross ist. Wir haben gezeigt, dass eine andere Definition der Problemgrösse (hier, die Anzahl der noch nicht besuchten Knoten) die Anwendung der konstruktiven Induktion ermöglicht und zu terminierenden Algorithmen führt.

7 Lösungen

Aufgabe 1: Der Graph enthält für jedes Objekt im Speicher einen Knoten und für jede Referenz zwischen Objekten eine Kante. Die beiden Objekte, die in den lokalen Variablen a und b gespeichert sind, sind Wurzeln. Drei weitere Objekte sind ebenfalls erreichbar. Drei der Objekte sind nicht von a oder b erreichbar:

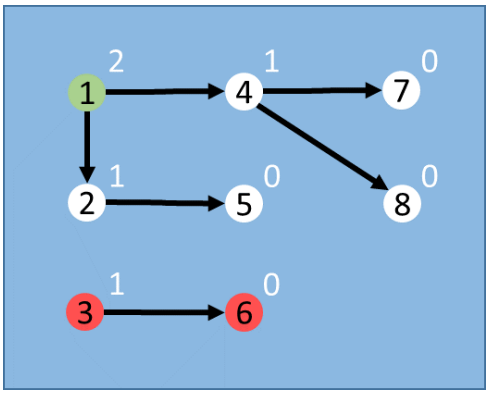


Aufgabe 2: Wir können obigen Graphen G = (N,E) wie folgt beschreiben:

N = {1, 2, 3, 4, 5, 6, 7, 8}

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

Aufgabe 3: Die weissen Zahlen geben *hoehe*(n) für jeden Knoten n an:



Wenn man den Knoten 5 entfernt, ändert sich *hoehe*(2) zu 0, da der Knoten 2 dann keine Nachfolger mehr hat. Die Höhe von Knoten 1 bleibt jedoch unverändert, da das Maximum der Höhen seiner Nachfolger immer noch 1 ist.

Aufgabe 4: Wir erhalten folgenden Ablauf:

REACH(1)
↳ REACH(2)
↳ REACH(5)
↳liefert {5}
↳liefert {2, 5}
↳ REACH(4)
↳ REACH(7)
↳liefert {7}
↳ REACH(8)
↳liefert {8}
↳liefert {4, 7, 8}
↳liefert {1, 2, 4, 5, 7, 8}

Aufgabe 5: Der Algorithmus terminiert nicht.

Aufgabe 6: Eine naheliegende Idee ist, einen Knoten nicht zu besuchen, wenn er bereits besucht wurde, also bereits in R enthalten ist.

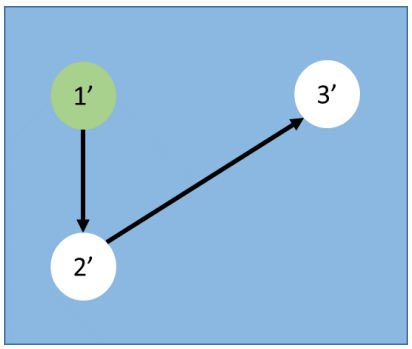
Aufgabe 7: Wir erhalten die erwünschten Resultate:

REACH2(1) = {1, 2, 3}
REACH2(2) = {1, 2, 3}
REACH2(3) = {1, 2, 3}

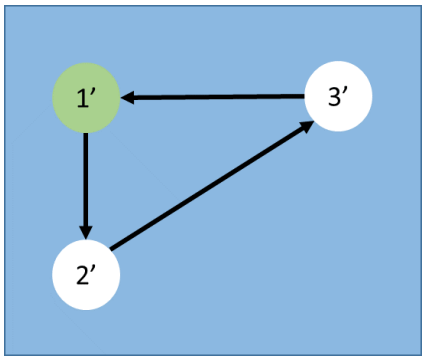
Aufgabe 8: Der folgende Algorithmus terminiert auch für zyklische Graphen:

Algorithmus COPY(n)
R := R ∪ { n }
Wähle n‘, sodass n‘ ∉ N ∪ N‘
N‘ := N‘ ∪ { n‘ }
Für jeden Knoten m, sodass (n,m) ∈ E und m ∉ R:
m‘ := COPY(m)
E‘ := E‘ ∪ { (n‘,m‘) }
Gib n‘ als Ergebnis zurück.

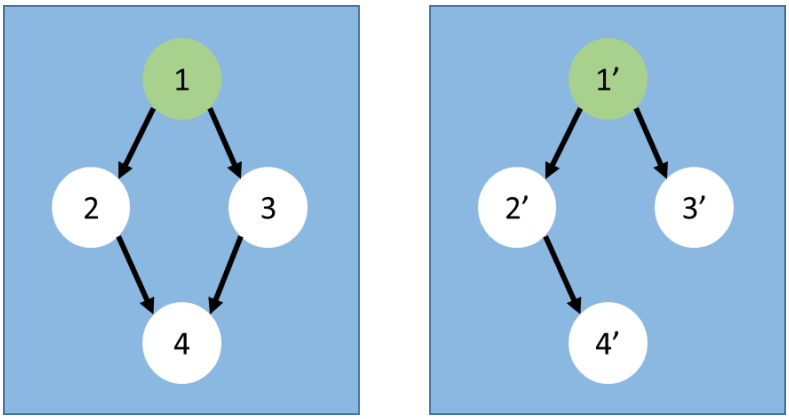
Aufgabe 9: Wir erhalten den folgenden Graphen:



Aufgabe 10: Nun erhalten wir das korrekte Ergebnis:



Aufgabe 11: Für azyklische Graphen, in denen ein erreichbarer Knoten mehr als einen erreichbaren Vorgänger hat, terminiert der Algorithmus COPY zwar, liefert aber ein inkorrektes Ergebnis. Wie schon für zyklische Graphen fehlen Kanten zu den Knoten, die über verschiedene Pfade erreichbar sind, wie folgendes Beispiel zeigt:



COPY2 liefert das korrekte Ergebnis.