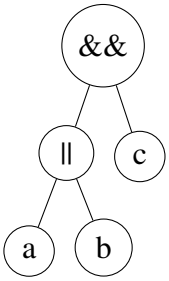
FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

12. Übung

Abgabe bis 26.07.2017, 10:00 Uhr

Einzelaufgabe 12.1: Kantorowitsch-Bäume — 9 EP

Sogenannte *Kantorowitsch-Bäume* eignen sich besonders gut zur symbolischen Darstellung von arithmetischen oder logischen Termen. Die inneren Knoten sind dabei Operatoren, deren Operanden jeweils in den Unterbäumen stehen (Reihenfolge ist bei nicht-kommutativen Operatoren *wichtig*!). Die Blätter stellen meist Konstanten oder Variablen dar. Beispiel:



Textuelle Darstellungen:

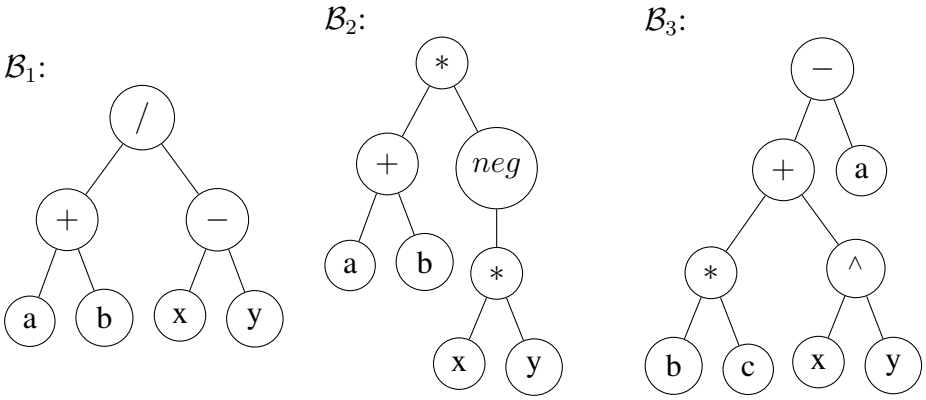
Prefix: $\&\&\ ||\ a\ b\ c$

Infix: $(a\ ||\ b)\ \&\&\ c$

Postfix: $a\ b\ ||\ c\ \&\&$

Traversiert man einen *Kantorowitsch-Baum* jeweils in *preorder*/*inorder*/*postorder*-Reihenfolge und gibt die besuchten Knoten entsprechend aus, dann ergibt sich die zugehörige textuelle Darstellung in *Prefix*/*Infix*/*Postfix*-Notation wie im obigen Beispiel. Anders als die übliche *Infix*-Schreibweise kommt die *Prefix*-Notation ohne Klammern aus.

a) Schreiben Sie folgende *Kantorowitsch-Bäume* jeweils in *Prefix*-, *Infix*- und *Postfix*-Notation. Anmerkung: $a\hat{}b$ steht für a^b, $neg\ a$ steht für die unäre Negation $-a$.



b) Schreiben Sie folgende Notationen in die jeweils anderen Notationen um:

Prefix: $+\ -\ *\ a\ b\ /\ c\ d\ e$

Infix: $(((a\ +\ b)\ +\ c)\ +\ d)\ +\ e$

Postfix: $x\ y\ z\ -\ i\ +\ -\ k\ -$

c) Erstellen Sie für die in Teilaufgabe b) gegebenen Ausdrücke jeweils *Kantorowitsch-Bäume*.

Einzelaufgabe 12.2: Kürzeste Wege **30 EP**

Gegeben sei der folgende ungerichtete aber gewichtete Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in Mengendarstellung mit Kantenmenge $\mathcal{E} = \{\ [C, 63, B], [G, 30, F], [A, 27, B], [G, 18, D], [C, 18, A], [Z, 15, H], [K, 12, H], [F, 12, E], [D, 12, E], [Z, 9, K], [C, 9, G], [B, 9, E], [K, 6, F], [H, 6, D]\ \}$, dabei stellt $[v_i, d_i^j, v_j]$ eine ungerichtete Kante zwischen den Knoten v_i und v_j mit Gewicht d_i^j dar.

a) Zeichnen Sie $\mathcal{G}$ *planar*, d.h. in einer Ebene und *ohne* Überschneidungen der Kanten.

b) Geben Sie die *Adjazenz-Listen*-Darstellung der Knoten von $\mathcal{G}$ einschließlich Entfernungen in der Form $A \rightarrow \boxed{U, 47} \rightarrow \boxed{D, 11}$ an. Zählen Sie die Zielknoten alphabetisch aufsteigend auf.

c) Bestimmen Sie mit Hilfe des Algorithmus von *Dijkstra* den kürzesten Weg von A bis Z (und zu allen anderen Knoten). Geben Sie Ihre Lösung in folgender Form mit alphabetisch aufsteigend sortierten Knoten in den Spalten an und $\boxed{\text{markieren}}$ Sie in jeder Zeile den jeweils als nächstes zu betrachtenden Ausgangsknoten. Setzen Sie für die noch zu bearbeitenden Knoten einen Prioritäts*stapel* (Stack: bei gleicher Entfernung wird der jüngere Knoten gewählt) ein.

	A	U	D
	$\boxed{0}$	∞	∞
	...	...	...
Ergebnis:	...	...	...

d) Wenden Sie den Algorithmus von Kruskal an und geben Sie die Wegabschnitte in der Reihenfolge an, in der Sie sie in die Ergebnisliste aufnehmen. Wie groß ist die Summe der Kanten im finalen Spannbaum?

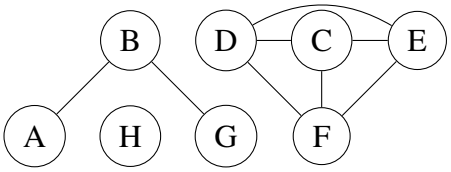
e) Betrachten Sie nun den *gerichteten* Teilgraphen $\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$, der aus $\mathcal{G}$ entsteht, indem die Knoten H, K und Z zusammen mit ihren eingehenden sowie ausgehenden Kanten entfernt und die verbleibenden Kanten als gerichtet angesehen werden, z.B. $[C, 63, B] \mapsto (C, 63, B)$. Ermitteln Sie mit Hilfe des Algorithmus von *Floyd* die kürzesten Wege zwischen allen Knotenpaaren in $\mathcal{H}$. Tragen Sie dazu in einer Tabelle folgender Form die jeweils betrachteten Kantenpaare $(u_j, v_i) \wedge (v_i, w_k)$, deren Länge $\gamma_{j,i,k} := |(u_j, v_i)| + |(v_i, w_k)|$ sowie die zuletzt bekannte kürzeste Entfernung $\gamma_{alt} := |(u_j, w_k)|$ ein. Dabei können Sie die Zeilen weglassen, die keinen gültigen Pfad im Graph darstellen (z.B. $A \rightarrow C \rightarrow B$). Bearbeiten Sie die Knoten in alphabetischer Reihenfolge: sortiert zuerst nach v_i, dann nach u_j und schließlich nach w_k! Zeichnen Sie den Graphen $\mathcal{H}$, der sich nach Anwendung des Algorithmus von Floyd ergibt. Jedes eingetragene Kantengewicht soll die kürzeste Entfernung zwischen den jeweils verbundenen Knoten darstellen.

Vorgänger u_j	*Knoten* v_i	*Nachfolger* w_k	$\lvert(u_j, v_i)\rvert + \lvert(v_i, w_k)\rvert$ $\gamma_{j,i,k}$	*„alte Länge“* γ_{alt}
...	A	...	...	...
...	...	...	...	...

Geben Sie Ihre Lösung als `Graphentheorie.pdf` über EST ab.

Einzelaufgabe 12.3: Zusammenhangskomponenten — 7 EP

Einen maximalen zusammenhängenden Teilgraphen eines beliebigen ungerichteten Graphen nennt man *Zusammenhangskomponente*. In dieser Aufgabe sollen Sie alle Zusammenhangskomponenten eines ungerichteten Graphen identifizieren. Erstellen Sie dazu eine Klasse `GraphOperations` mit einer Klassenmethode `getComponents` so, dass der öffentliche Testfall damit übersetzt und erfolgreich läuft. Der im Testfall verwendete Beispiel-Graph und die erwartete Ergebnisstruktur sind:



Zusammenhangskomponenten:
`[[D, C, E, F], [B, A, G], [H]]`
(Die Reihenfolge der Knoten in den Teillisten ist irrelevant!)

Gruppenaufgabe 12.4: (Speicher-)Darstellungen von Graphen — 24 GP

In der Vorlesung wurden verschiedene Datenstrukturen vorgestellt, mit denen man Graphen im Speicher darstellen kann. In dieser Aufgabe sollen Sie verschiedene Repräsentationen selbst implementieren und so dargestellte Graphen verarbeiten.
WICHTIG: Betrachten Sie die öffentlichen Testfälle als festen Bestandteil der Aufgabenstellung (Spezifikation) – beachten Sie *insbesondere* die Einschränkungen bzgl. der Schnittstelle: Schlägt einer der deutlich erkennbaren Testfälle fehl, werden Sie für diese Aufgabe *keine Punkte* bekommen!
Die vorgegebene Klasse `AbstractGraph` beschreibt einen Graphen mit fester Anzahl an Knoten. Die Knoten haben keinen expliziten Namen, sondern werden durch einfache Nummerierung ab 0 indiziert. Sie können davon ausgehen, dass ein Graph immer mindestens einen Knoten enthält.

a) Erstellen Sie eine abstrakte Klasse `AbstractGraphWithOps`, die `AbstractGraph` erweitert. Diese neue Klasse kennt nicht die konkrete Repräsentation des Graphen und belässt daher `addEdge` und `hasEdge` weiterhin unimplementiert. Sie kann aber trotzdem bereits folgende Methoden bereitstellen, die Sie entsprechend implementieren sollen:

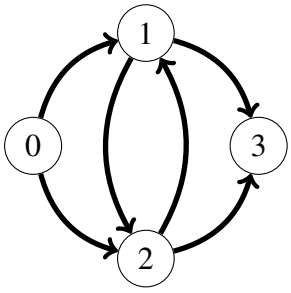
- `isUndirected()` soll prüfen, ob der Graph ungerichtet (genauer: doppelt-gerichtet) ist, also ob es zu jeder Kante (n_i, n_k) auch eine Gegenkante (n_k, n_i) gibt. Beachten Sie dabei, dass ein Graph mit *Schlingen per definitionem nicht ungerichtet* sein soll.
- `isTree()` soll bestimmen, ob der Graph ein Baum ist. Der Graph darf also keine Zyklen enthalten und es muss genau einen Pfad vom Wurzelknoten zu jedem anderen Knoten geben. Wenn der Graph diese Eigenschaften erfüllt, wird der Wurzelknoten zurückgegeben, andernfalls -1. Sollten mehrere Knoten als Wurzelknoten in Frage kommen, soll der mit der kleinsten Nummer zurückgegeben werden.
  Tipp: In ungerichteten Graphen ist entweder *kein* Knoten ein Wurzelknoten oder alle Knoten erfüllen diese Eigenschaft.
- `dfsPreOrder()` sowie `bfsPreOrder()` sollen den Graphen, ausgehend vom Startknoten, in Tiefen- bzw. Breitensuche traversieren und die besuchten Knoten in der Reihenfolge des ersten Besuchs zurückgeben. Innerhalb eines Knotens sollen die Folgeknoten in aufsteigender Reihenfolge der Knotennummern besucht werden.

b) Erstellen Sie eine Klasse `AdjacencyMatrixGraph`, die von `AbstractGraphWithOps` erbt. Diese Klasse darf nur ein 2D-`boolean`-Feld $\mathcal{M}$, einen Konstruktor sowie die beiden Methoden `addEdge` bzw. `hasEdge` bereitstellen. Sie verwaltet die *gerichteten* Kanten in der Adjazenz*matrix* $\mathcal{M}$.

c) Erstellen Sie eine Klasse `AdjacencyListGraph`, die von `AbstractGraphWithOps` erbt. Diese Klasse darf nur ein `List<List<Integer>>`-Attribut $\mathcal{L}$, einen Konstruktor sowie die beiden Methoden `addEdge` bzw. `hasEdge` bereitstellen. Sie verwaltet die *gerichteten* Kanten in der Adjazenz*liste* $\mathcal{L}$.

d) Erstellen Sie eine Klasse `AdjacencyArrayGraph`, die von `AbstractGraphWithOps` erbt. Diese Klasse darf nur ein `List<Integer>`-Attribut $\mathcal{F}$, einen Konstruktor sowie die beiden Methoden `addEdge` bzw. `hasEdge` bereitstellen. Sie verwaltet die *gerichteten* Kanten im Adjazenz*feld* $\mathcal{F}$. Hierbei handelt es sich um eine „Array-Einbettung“ eines Graphen, die bis vor einigen Semestern noch in der AuD-Vorlesung gelehrt wurde und wie folgt funktioniert:
Für einen Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ mit $n := |\mathcal{V}|$ Knoten und $e := |\mathcal{E}|$ Kanten wird ein Feld $\mathcal{F}$ der Länge $n+1+e$ angelegt. Die ersten n Einträge (bei Positionen 0 bis $n-1$) repräsentieren die Knoten: Es sind Verweise auf den Bereich ab $n+1$, in dem die Kantenziele verwaltet werden (der Eintrag bei Position n ist ein „Dummy“-Eintrag der auf $n+1+e$, also „hinter“ das Feld verweist, um das Ende des Knotenbereichs zu markieren). Der Ausgangsgrad $outdegree(\nu)$ eines Knotens ν errechnet sich damit direkt aus dem Feld: $outdegree(\nu) = \mathcal{F}[\nu+1] - \mathcal{F}[\nu]$.

Beispiel (Knoten 3 ist eine Senke, daher verweist der Eintrag ebenfalls auf 11.):



0	1	2	3	4	5	6	7	8	9	10	11
5	7	9	11	11	1	2	2	3	1	3	

Das Feld $\mathcal{F}$ ist dann z.B. so zu lesen:
„Die Nachfolger des Knotens 1 befinden sich ab Position 7 aber vor 9 in $\mathcal{F}$ (sind also 2 und 3).“

46 EP + 24 GP = 70 Punkte