TU Ilmenau, Fakultät für Informatik und Automatisierung
FG Komplexitätstheorie und Effiziente Algorithmen
Univ.-Prof. Dr. M. Dietzfelbinger, Philipp Schlag, Stefan Walzer, Alexander Jeuk
https://www.tu-ilmenau.de/iti/lehre/lehre-ss-2020/aud/

TECHNISCHE UNIVERSITÄT
ILMENAU

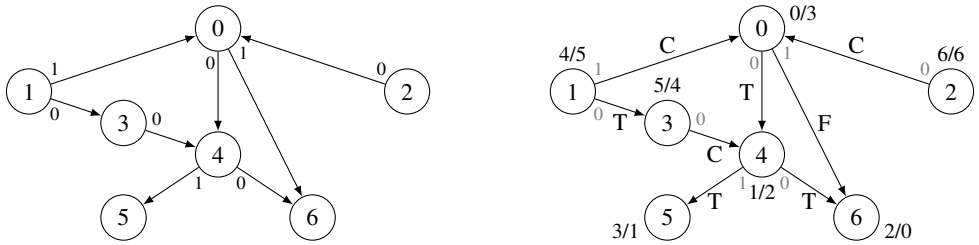
Algorithmen und Datenstrukturen SS 2020 – Praktikum 5

Zur Bearbeitung zwischen 6. Juli - 10. Juli (KW 28)

Ziel dieses Praktikums ist die Anwendung der vollen Tiefensuche zur Lösung verschiedener Probleme in Digraphen. Die Eingabedateien enthalten jeweils eine oder mehrere Zeilen. Jede Zeile beschreibt einen Digraphen G durch Angabe von positiven ganzen Zahlen n und m – der Anzahl von Knoten und Kanten – sowie von m Zahlenpaaren die jeweils eine Kanten beschreiben.
Beispielsweise enthält die Datei `example.in` eine einzige Zeile:

```
1 7 8 (0,4) (0,6) (1,3) (1,0) (2,0) (3,4) (4,6) (4,5)
```

Diese repräsentiert den links abgebildeten Digraphen. (Die Reihenfolgen der Kanten in den Adjazenzlisten sind als Kantenmarkierungen, beginnend bei 0, gegeben.) Rechts sehen Sie das zugehörige Ergebnis der vollen Tiefensuche.



Beachten Sie: Die **Reihenfolge** in der ausgehende Kante eines Knotens betrachtet werden entspricht der Reihenfolge in der Eingabe (z.B. wird Kante $(4,6)$ vor Kante $(4,5)$ betrachtet). Die Knotenindizes sowie die **dfs**- und **fin**-Nummern sind **nullbasiert**.

Aufgabenstruktur

Die Tiefensuche ist bereits implementiert. Alle Aufgaben können allein durch zusätzlichen Code in Callback Funktionen (`dfsVisit`, `finVisit`,...) gelöst werden.

Vorimplementierte Tiefensuche

Jede zu bearbeitende `.cpp`-Datei enthält bereits ein funktionstüchtiges Programm (inklusive `main`-Funktion).
Die Eingabedatei wird zeilenweise eingelesen und für jede Zeile wird der zugehöriger Digraph konstruiert. In die `int`-Variablen `n` und `m` wird die Anzahl an Knoten und Kanten eingetragen. Der Digraph selber wird als `vector<vector<int>> G` (Array von Adjazenzlisten) dargestellt: Im obigen Beispiel hat der Knoten 4 die beiden ausgehenden Kanten $(4,6)$ und $(4,5)$ (in dieser Reihenfolge). Diese werden durch `G[4]` vom Typ `vector<int>` beschrieben: Wir haben `G[4][0]==6` und `G[4][1]==5`.
Auf dem eingelesenen Digraphen wird dann die volle Tiefensuche durchgeführt. Danach enthalten die beiden Arrays `dfsNum` und `finNum` die den Knoten zugeordneten Nummern, z. B. ist `finNum[4]==2`. Die Kanten werden *in der Reihenfolge ihrer Klassifizierung* jeweils in eines der vier Arrays `T`, `B`, `F`, `C` eingefügt. Eine Kante von u nach v wird hierbei als `pair<int,int>` mit `e.first==u` und `e.second==v` gespeichert.

Verwendung der Callback Funktionen

Sie können alle Aufgaben lösen, indem Sie in die folgenden Callback Funktionen geeigneten Code einfügen.

- `preDFS` wird nach dem Einlesen des aktuellen Digraphen, aber noch vor dem Start (auch der zugehörigen Initialisierungen) der vollen Tiefensuche aufgerufen. Wenn Sie zusätzliche Variablen (im Beispiel `level`) verwenden, können Sie diese hier vor jedem Neustart (mit einem neuen Digraphen) wieder initialisieren.
- `dfsVisit` wird für Knoten `v` aufgerufen, sobald er seine **dfs**-Nummer erhalten hat (am Anfang von `dfs(v)`), aber bevor die ausgehenden Kanten untersucht werden.
- `finVisit` wird für Knoten `v` aufgerufen, sobald er seine **fin**-Nummer erhalten hat (am Ende von `dfs(v)`), insbesondere also nachdem alle ausgehenden Kanten untersucht wurden.
- `postDFS` wird nach Beenden der vollen Tiefensuche aufgerufen. Dies kann für abschließende Berechnungen und Ausgaben genutzt werden.

Das Beispielprogramm `material/_Beispiel/example.cpp` implementiert diese Methoden folgendermaßen:

```
22 int level;               /* additional data structures */
23 void preDFS() {          /* actions directly before DFS() */
24     cout << "n␣=␣" << n << ",␣m␣=␣" << m << endl;
25     level = -1;
26     cout << "(node,level):";
27 }
28 void dfsVisit(const int v) {    /* preorder actions at node v */
29     cout << "␣(" << v << "," << ++level << ")";
30 }
31 void finVisit(const int v) {    /* postorder actions at node v */
32     --level;
33 }
34 void postDFS() {         /* actions directly after DFS() */
35     cout << endl << "dfsNum:";
36     for(int i : dfsNum) cout << "␣" << i;
37     cout << endl << "T:";
38     for(auto& e : T) cout << "␣(" << e.first << "," << e.second << ")";
39     cout << endl;
40 }
```

Die Ausgabe für den Beispielgraphen (siehe oben) ist

```
1 n = 7, m = 8
2 (node,level): (0,0) (4,1) (6,2) (5,2) (1,0) (3,1) (2,0)
3 dfsNum: 0 4 6 5 1 3 2
4 T: (0,4) (4,6) (4,5) (1,3)
```

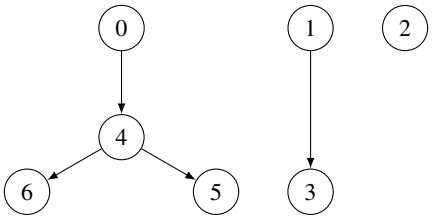
Machen Sie sich klar wie diese Ausgabe zustande kommt!

Aufgabe 1 (Tiefensuch-Wald, `dfsForest`)

Geben Sie die bei der vollen Tiefensuche implizit konstruierten Tiefensuchbäume als Zeichenketten aus. Unsere Beispiel-Instanz soll zur Ausgabe

```
0(4(6()5())) 1(3()) 2() : 3
```

führen, die dem Tiefensuch-Wald



entspricht. Die Zahl nach dem Doppelpunkt gibt dabei die Anzahl entstandener DFS-Bäume an.
Hinweis: Knoten mit demselben Vaterknoten sollen in der Reihenfolge ihrer Entdeckung in der Ausgabe vorkommen. In der Beispiel-Ausgabe steht Knoten 6 vor Knoten 5, da er zuerst entdeckt wurde (weil die zugehörige Kante zuerst untersucht wurde).
Tipp: Um für das Beispiel die Ausgabe „`0(4(6()5()))1(3())2()`“ zu bekommen genügen je eine Zeile in `dfsVisit` und `finVisit`. Um die Leerzeichen richtig zu platzieren und die Anzahl der Bäume zu bestimmen, können Sie ähnlich wie im Beispielprogramm das aktuelle `level` mitführen. Welchen Wert hat `level`, wenn gerade ein Baum beginnt bzw. endet?

Aufgabe 2 (Erreichbarkeit, `reachability`)

Bestimmen Sie alle vom Knoten 0 aus erreichbaren Knoten (in der Reihenfolge ihrer Entdeckung!) und geben Sie diese aus. Die Beispiel-Instanz soll

```
0 4 6 5
```

als Ausgabe liefern.
Hinweis: Nutzen Sie aus, dass Knoten 0 die Wurzel des ersten DFS-Baums ist.

Aufgabe 3 (Topologische Sortierung, `linearize`)

Ziel dieser Aufgabe ist es, die Knoten in topologisch sortierter Reihenfolge auszugeben (siehe Kapitel 8.3 der Vorlesung). Falls keine topologische Sortierung existiert, ist „-“ auszugeben. Unsere Beispiel-Instanz soll zur Ausgabe

```
2 1 3 0 4 5 6
```

führen.
Hinweis: Natürlich kann es für einen Digraphen mehr als eine topologische Sortierung geben. Die erwartete Ausgabe entspricht derjenigen topologischen Sortierung, die sich gemäß dem Verfahren aus der Vorlesung ergibt, weiterhin unter Beachtung der gegebenen Kantenreihenfolgen.

Aufgabe 4 (Einfache Wege, `simplePaths`)

In einem azyklischen Digraphen soll die Anzahl einfacher Wege von Knoten 0 zu Knoten $n-1$ bestimmt werden. Wiederholen Sie hierzu Aufgabe 2 von Übungsblatt 9. Falls der gegebene Digraph einen Kreis besitzt, ist „-“ auszugeben. Unsere Beispiel-Instanz soll die Ausgabe

```
2
```

ergeben, da genau die beiden einfachen Wege $(0,4,6)$ und $(0,6)$ von 0 zu $n-1=6$ führen.
Hinweis: Im Allgemeinen kann die Anzahl einfacher Wege zwischen zwei Knoten sehr groß werden. Für alle unsere Beispiele ist der Datentyp `int` dennoch ausreichend.

Hinweis: Folgende *Zusatzaufgabe* ist nicht verpflichtend.

Aufgabe 5 (Kritischer Pfad, `criticalPath`)

Ziel dieser Aufgabe ist es, die Länge eines längsten Weges (gemessen in der Anzahl an Knoten) sowie einen Weg dieser Länge in einem azyklischen Digraphen zu bestimmen (siehe Kapitel 8.3 der Vorlesung, Folien 55 – 57). Falls der gegebene Digraph einen Kreis besitzt, ist wieder „-“ auszugeben. Die Beispiel-Instanz soll zur Ausgabe

```
4: 1 3 4 6
```

führen, da die Knotenfolge $(1,3,4,6)$ der Länge 4 der erste vollständig klassifizierte kritische Pfad ist.
Hinweis: Falls es mehrere kritische Pfade gibt, ist derjenige auszugeben, dessen Kanten als erstes vollständig klassifiziert wurden. Das bedeutet:

- Gibt es mehrere kritische Pfade, so ist derjenige auszugeben, dessen Startknoten die kleinste **fin**-Nummer besitzt. In unserer Beispiel-Instanz beginnen vier kritische Pfade in Knoten 1 und zwei weitere in Knoten 2. Da Knoten 1 die kleinere **fin**-Nummer besitzt, darf keiner der beiden bei 2 beginnenden kritischen Pfade ausgegeben werden. (Ein kritischer Pfad beginnt immer bei einer Wurzel eines DFS-Baums, da der Startknoten keine eingehenden Kanten besitzen kann. Damit ist derjenige Pfad auszugeben, der mit der kleinsten **fin**-, **dfs**- bzw. Knotennummer beginnt.)
- Gibt es auch hiervon mehrere kritische Pfade (diese beginnen also alle mit demselben Startknoten), so ist derjenige auszugeben, der (beginnend beim Startknoten) jeweils diejenige ausgehende Kante verfolgt, die weiter vorne in der Eingabe steht. In unserer Beispiel-Instanz kommt die Kante $(1,3)$ vor der Kante $(1,0)$. Im weiteren Verlauf kommt die Kante $(4,6)$ vor der Kante $(4,5)$.