

Netzwerkalgorithmen

May 2, 2016

1 Zusätzliches blabla

Makros in C/C++: `#define alias replace`, wobei `replace` auch Code sein kann.

2 Datentypen für Graphen und Netzwerke (LEDA)

Definition eines Datentyps

Definition der Objekte des Typs: *stack* $< T >$

Konstruktion: *stack* $< int >$ S(100) (max Größe)

Operationen: *s.push(Tx)*, *ts.pop()*

Bemerkung zu Implementierung

Graph-Datentyp in LEDA

Der Typ `graph` repräsentiert gerichtete Graphen.

Ein Graph `g` besteht aus zwei Typen von Objekten: `node` und `edge`

Mit jedem Knoten `v` sind zwei Listen von Kanten (*list* $< edge >$) verbunden (eingehend und ausgehend)

Mit jeder Kante `e` werden 2 Knoten `source` und `target` gespeichert.

Operationen auf G

Update:

`node G.new_node()`, erzeugt einen neuen Knoten in `G` und gibt ihn zurück. `edge G.new_edge(node v, node w)`

`void G.del_edge(edge)`

Access:

`list < edge > G.out_edges(node v);`

`int G.outdeg(node v);`

`node G.source(edge);`

`node G.target(edge);`

Iteration:

`forall_nodes(v,G)`

`forall_edges(e,G)`

`forall_out_edges(e,v)`

`forall_in_edges(e,v)`

1. Problem

Gegeben: Graph $G=(V,E)$

Frage: Ist G azyklisch?

Algorithmus siehe Topologisches Sortieren: Entferne jeweils einen Knoten v mit $\text{indeg}(v)=0$ bis der Graph leer ist. Falls wir keinen solchen Knoten finden dann ist der Graph zyklisch, falls G am Ende leer, ist er azyklisch. C++:

```
bool ACYCLIC(graph G){                               //Call by value damit G nicht zerstört
    list<node> zero;
    node v;
    forall_nodes(v,G){
        if (G.indeg(v)==0) zero.append(v);
    }
    while (!zero.empty()){
        node u = zero.pop();
        edge e;
        forall_out_edges(e,u){
            node w=G.target(e);
            G.del_edge(e);
            if (G.indeg(w) == 0){
                zero.append(w);
            }
        }
    }
    return G.empty();
}
```

Daten für Knoten und Kanten

1. Parametrisierte Graphen: $\text{GRAPH}<\text{node_type}, \text{edge_type}> G$
2. Temporäre Daten: $\text{besucht}[v] \leftarrow \text{true}$

Datentypen in LEDA

$\text{node_array}<T> A(G,x)$: Feld über die Knoten des Graphen G $\text{edge_array}<T> B(G,y)$ analog Verwendet für: Temporäre Daten, Eingabedaten, Resultate

Anwendung im topologischen Sortieren

injektive Abbildung: $\text{topnum}: V \rightarrow \{1, \dots, n\}$ mit $\forall (v, w) \in E: \text{topnum}[v] < \text{topnum}[w]$

```
bool TOPSORT(const graph& G, node_array<int>& topnum){
    int count = 0;
    list<node> zero;
    node_array<int> indeg(G);
    node v;
    forall_nodes(v,G){
        indeg[v] = G.indeg(v);
        if (indeg[v] == 0) zero.append(v);
    }
    while (!zero.empty()){
        node v = zero.pop();
        topnum[v] = ++count;
        edge e;
        forall_out_edges(e,v){
            node w = G.target(e);
            if (--indeg[w] == 0) zero.append(w);
        }
    }
}
```

```

    }
}
return count == G.number_of_nodes();
}

```

Tiefensuche

Hauptprogramm:

```

void DFS(const graph& G, node_array<int>& dfsnum, node_array<int>& compnum){
    int count1 = 0;
    int count2 = 0;
    node_array<bool> visited(G, false);
    node v;
    forall_nodes(v,G){
        if (!visited[v]) dfs(G,v,count1,count2,dfsnum,compnum)
    }
}

```

Rekursive Funktion dfs:

```

void dfs(const graph& g, node v, int& count1, int& count2, node_array<int>& dfsnum, -
    node_array<int>& compnum){
    dfsnum[v] = ++count1;
    visited[v] = true;
    edge e;
    forall_out_edges(e,v){
        edge w = G.target(e);
        if (!visited[w]) dfs(G,w,count1,count2,dfsnum,compnum)
    }
    compnum[v] = ++count2
}

```

Berechnung starker Zusammenhangskomponenten

Definition: Ein gerichteter Graph ist stark zusammenhängend, wenn $\forall v, w \in V : v \rightarrow^* w$ (es existiert ein Pfad von v nach w)

Die starken Zusammenhangskomponenten (SZK) von G sind die maximalen SZK Teilgraphen von G .

Idee für Algorithmus:

1. führe DFS mit $G' = (V', E')$ dem Teilgraphen aufgespannt von bereits besuchten Knoten
2. Verwalte SZK von G' während DFS ausgeführt wird.

Ablauf:

Sei (v,w) die nächste in dfs betrachtete Kante

1. Fall: $(v,w) \in T$ (Baumkante), w noch nicht besucht. $V' = V' \cup \{w\}, E' = E' \cup \{(v,w)\}, SZK = SZK \cup \{\{w\}\}$
2. Fall: $(v,w) \notin T$, d.h. w wurde schon besucht, ist also in V' enthalten. $E' = E' \cup \{(v,w)\}$. Nun kann (v,w) mehrere bereits bekannte SZK vereinigen (Rückwärtskante oder Cross-Kante). Bemerkung: Vorwärtskanten generieren keine neuen Pfade in G' , deswegen dabei keine Änderung der SZK.

Bezeichnungen:

1. Eine SKZ K heißt abgeschlossen, falls die Aufrufe von dfs für alle Knoten v in K beendet sind.
2. Die Wurzel einer SZK K ist der Knoten mit der kleinsten dfsnum in K .
3. "Unfertig" ist die Folge aller Knoten für die dfs aufgerufen wurde, aber deren SZK in der sie sich befinden noch nicht abgeschlossen ist.

4. "Wurzeln" ist die Folge aller Wurzeln der nicht abgeschlossenen SZK nach dfsnum sortiert.

Situation, wenn DFS beim Knoten g angekommen ist:

Unfertig: a, b, c, e, f, g

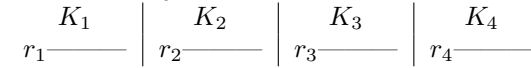
Wurzeln: $1, b, e, g$

Der Algorithmus betrachtet danach die Kanten aus g (g, d) $\in C$: es passiert nichts, da d in einer abgeschlossenen SZK ist; (g, c) $\in C$ Vereinigt die 3 SZK mit den Wurzeln b, e, g durch entfernen von e, g aus der Wurzelfolge.

Beobachtung: Hinzufügen und Streichen nur am Ende \rightarrow Stack eignet sich als Datenstruktur.

Allgemeine Situation für $(v, w) \in T$:

$K' = K_2 \cup K_3 \cup K_4$.



Ergänzungen von DFS für SZK:

1. Aktion: while $\text{dfsnum}[\text{wurzel.top}()] > \text{dfsnum}[w]$ do $\text{wurzel.pop}()$ od
2. Falls $(v, w) \in T$: $\text{Wurzeln.push}(w)$; $\text{Unfertig.push}(v)$
3. Abschluss eine SZK: SZK von v wird endgültig verlassen, sie ist nun abgeschlossen.

Am Ende von $\text{dfs}(v)$: if $v == \text{Wurzel.top}()$ then

$\text{Wurzeln.pop}()$;

repeat $w = \text{unfertig.pop}()$ until $w == v$

fi

Übung 2: Algo zu SZK.

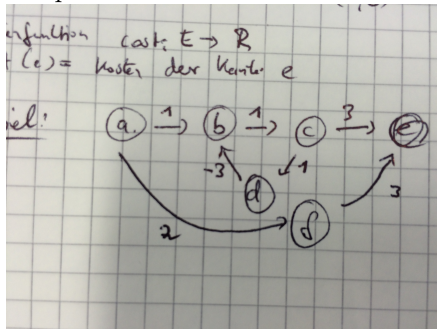
Darstellung der einzelnen SZKs: $\text{node_array} < \text{int} > \text{szknum}(G)$; $k = \# \text{komponenten}$ forall $v \in V$ $\text{szknum}[v] = i \Leftrightarrow v$ in der Komponente i

$\rightarrow \text{int SZK}(\text{const graph\& } G, \text{node_array} < \text{int} > \& \text{sznum})$

3 Kürzeste Wege / Billigste Wege

Allgemeines Problem: Sei gegeben ein gerichteter Graph $G = (V, E)$, eine Kostenfunktion $\text{cost} : E \rightarrow \mathbb{R}$ mit $\text{cost}(e) = \text{Kosten der Kante } e$.

Beispiel:



$k=3$

Kosten eines Pfades P : $\text{cost}(P) := \sum_{e \in P} \text{cost}(e)$

Billigste-Wege Problem: Finde einen Pfad P mit $\text{cost}(P)$ minimal

- a) Zwischen zwei gegebenen Knoten
- b) Von einem Knoten s zu allen anderen Knoten (single source shortest paths)
- c) Zwischen allen Paaren $(v, w) \in V \times V$

Im Beispiel: $\text{dist}(a, f) = 2$, $\text{dist}(a, e) = -\infty$ weil $a \rightarrow b \rightarrow (c \rightarrow d \rightarrow b \rightarrow c)^i \rightarrow e$

Definition: $\text{dist}(v, w) := \inf\{\text{cost}(P) \mid P \text{ ist Pfad von } v \text{ nach } w\}$ d.h. wenn es einen negativen Zyklus gibt $= -\infty$, wenn kein Pfad existiert: ∞ .

Wir betrachten hier die Single-Source Variante:

Eingabe: $G = (V, E)$, $s \in V$, $\text{cost} : E \rightarrow \mathbb{R}$

Ausgabe: $\forall v \in V : \text{dist}(s, v)$

Beobachtung: Die dist -Funktion erfüllt die Dreiecksungleichung: $\text{dist}(s, w) \leq \text{dist}(s, v) + \text{cost}(v, w)$

Idee für allgemeinen Algorithmus (Label Correcting):

Überschätze die dist-Werte ($DIST = \infty$). Solange eine Kante (u, v) die Dreiecksungleichung verletzt, korrigiere $DIST[v]$

```
for all v in V do
     $DIST[v] = \infty$ 
od
 $DIST[s] = 0$ 
while  $\exists (u, v) \in E$  mit  $DIST[v] > DIST[u] + cost(u, v)$ 
     $DIST[v] = DIST[u] + cost(u, v)$ 
od
```

Eigenschaften

- a) Es gilt immer $DIST[v] \geq dist(s, v)$
- b) Wenn $DIST[v] < \infty$, dann existiert ein Pfad von s nach v mit den Kosten $DIST[v]$
- c) Die kürzesten Pfade bilden einen Baum T mit Wurzel s . Begründung: In jeden Knoten mit $DIST[v] < \infty$ mündet genau ein kürzester Pfad. Bei korrigieren von $DIST$ -Wert wird die eingehende Kante von T definiert.
- d) Für jede Kante (v, w) auf einem Kürzesten Pfad gilt: $dist(s, w) = dist(s, v) + cost(v, w)$

Effizienter Algorithmus: Kandidatenmenge

Wir speichern alle Knoten aus denen Kanten ausgehen können, die die Dreiecksungleichung verletzen in einer Menge U .

Am Anfang gilt: $U = \{s\}$