Netzwerkalgorithmen

May 9, 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 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: GRAPH<node_type,edge_type> G
2. Temporäre Daten: besucht[v] $\leftarrow$ true

Datentypen in LEDA

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

Anwendung im topologischen Sortieren

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

```
bool TOPSORT(const graph& G, node_array<int>& topnum){
        int count = 0;
        list<node> zerol
        node_array<int> indeg(G);
        node v;
        forall_nodes(v,G){
                indeg[v] = G.indeg(v);
                if(indev[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 $\forall v, w\ inV : v \rightarrow^* w$ (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) meherere 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 dei 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 → Stack eignet sich als Datenstruktur.
Allgemeine Situation für $(v,w) \in T$:
$K' = K_2 \cup K_3 \cup K_4$.

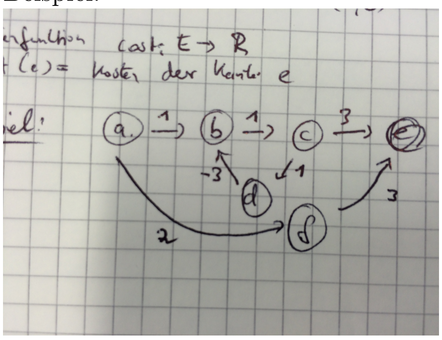
K_1	K_2	K_3	K_4
r_1———	r_2———	r_3———	r_4———

Ergänzungen von DFS für SZK:
1. Aktion: while dfsum[wurzel.top()] > dfsnum[w] do wurzeln.pop() od
2. Falls $(v,w) \in T$: Wurzeln.push(w); Unfertig.push(v)
3. Abschluss eine SZK: SZK von v wird endgültig verlassen, sie ist nun abgeschlossen.
    Am Ende von dfs(v): if v == Wurzel.top() then
        Wurzeln.pop();
        repeat w = unfertig.pop() until w==v
    fi

Übung 2: Algo zu SZK.
Darstellung der einzelnen SZKs: node_array< int > szknum(G); k=#kompnum $forall v \in V szknum[v] = i \Leftrightarrow$v in der Komponente i
→ int SZK(const graph& G, node_array< int >& sznum)

3 Kürzeste Wege / Billigste Wege

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



k=3

Kosten eines Pfades P: $cost(P) := \sum_{e \in P} cost(e)$
Billigste-Wege Problem: Finde einen Pfad P mit 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: $dist(a,f) = 2$, $dist(a,e) = -\infty$ weil $a \to b \to (c \to d \to b \to c)^i \to e$
Definition: $dist(v,w) := inf\{cost(P)|P$ist Pfad von v nach w$\}$ d.h. wenn es einen negativen Zyklus gibt $= -\infty$, wenn kein Pfad existiert: ∞.
Wir betrachten hier die Single-Source Variante:
Eingabe: $G = (V,E), s \in V, cost : E \Rightarrow \mathbb{R}$
Ausgabe: $\forall v \in V : dist(s,v)$
Beobachtung: Die dist-Funktion erfüllt die Dreiecksungleichung: $dist(s,w) \leq dist(s,v) + cost(v,w)$

Idee für allgemeinen Algorithmus (Label Correcting):
Überschätze die dist-Werte (DIST=∞). Solange eine Kante (u,v) die Dreiecksungleichung verletzt, korrigiere DIST[v]

```
forall v in V do
        DIST[v] = ∞
od
DIST[s] = 0
while ∃(u,v) ∈ 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: $U = \{s\}$. Immer wenn DIST[v] vermindert wird, nimm v nach U auf. Verfeinerter Algorithmus:

```
1.  forall v ∈ V do
        DIST[v] = ∞
    od
2.  DIST[s] = 0
3.  U = {s}
4.  while U ≠ ∅
5.       wähle und entferne ein u ∈ U
6        forall v ∈ V mit (u,v) ∈ E do
7.         c = DIST[u]+cost(u,v)
8.         if c < DIST[v] then
9.              DIST[v]=c
10.             U = U∪ {v}
11.        fi
12.      od
13.od
```

Eigenschaften des Algorithmus während Laufzeit (Lemma):
Falls der Graph G keine negativen Zyklen enthält, dann gilt:
a) Falls $v \notin U$, dann gilt für alle ausgehenden Kanten (v,w): DIST[w]$\leq$ DIST[v]+c(v,w).
b) Sei $v_0, ..., v_k$ ein billigster Pfad von s nach v. Falls nun DIST[v]$>$ dist(s,v), dann existiert ein i $0 \leq i \leq k-1$ mit DIST[v_i] $=$ dist(s,v_i) und $v_i \in U$
c) Es existiert immer ein $u \in U$ mit DIST[u]=dist(s,u) und wenn in Zeile 5 des Algorithmus ein solches u gewählt wird (perfekte Wahl), dann wird die while-Schleife für jeden Knoten höchstes einmal ausgeführt.
Beweis:
a) Induktion über Schleifendurchläufe i (while)
i=0: Vor dem ersten Durchlauf gilt die Behauptung, da DIST[s]=0 und DIST[v]=$\infty \forall v \in V$, U=$\{s\}$
i=i+1: Betrachte ein beliebiges $v \notin U$ nach der i+1-ten Ausführung
$v \notin U$ vor der (i+1)-ten Ausführung -¿ Nach IA: DIST[v]+c(v,w)$\geq$ DIST[w] für alle ausgehenden Kanten (v,w) und DIST[v] wurde in diesem Durchlauf nicht verändert. Die DIST-Werte der Nachbarn w von v wurden eventuell verändert $\Rightarrow DIST[v] + c(v,w) \geq DIST[w]$ für alle ausgehenden Kanten

$v \in U$ vor der (i+1)-ten Iteration: v wurde in Zeile 5 ausgewählt, dann stellt die innere Schleife die Dreiecksungleichung für alle ausgehenden Kanten wieder her.
b) Sei $s = v_0, ..., v_k = v$ ein billigster Pfad von s nach v mit $DIST[v_k] > dist(s, v_k)$
Sei i maximal mit $DIST[v_i] = dist(s, v_i) \Leftarrow 0 \leq i < k$ (v_0 hat korrekten Wert)
Wir zeigen $v_i \in U$:
Annahme $v_i \notin U$: $DIST[v_i] + c(v_i, v_{i+1}) \geq DIST[v_{i+1}]$ Außerdem gilt: dist(s,v_{i+1}) $= dist(s, v_i) + c(v_i, v_{i+1})$.
Daraus folgt: $dist(s, v_{i+1}) = dist(s, v_i) + c(v_i, v_{i+1}) = DIST[v_i] + c(v_i, v_{i+1}) \geq DIST[v_{i+1}] \Rightarrow dist(s, v_{i+1}) = DIST[v_{i+1}]$ Widerspruch zur Wahl von i.
c) Sei $v \in U$ beliebig
Falls DIST[u] = dist(s,u) ok.
Falls DIST[u] > dist(s,u): Betrachte einen billigsten Pfad von s nach u, auf dem nach b) ein $v_i \in U$ existiert, mit DIST[v_i] = dist(s,v_i)
Perfekte Wahl: Falls in Zeile 5 immer ein solcher Knoten $u \in U$ gewählt wird, dann kann u kein zweites Mal nach U angefügt werden.
Perkefte Wahl $\Rightarrow$ Gesamtaufwand: $\mathcal{O}(\sum_{v \in V}(1 + outdeg(v) + Verwaltung\ der\ Menge\ U) = n + m + ?$