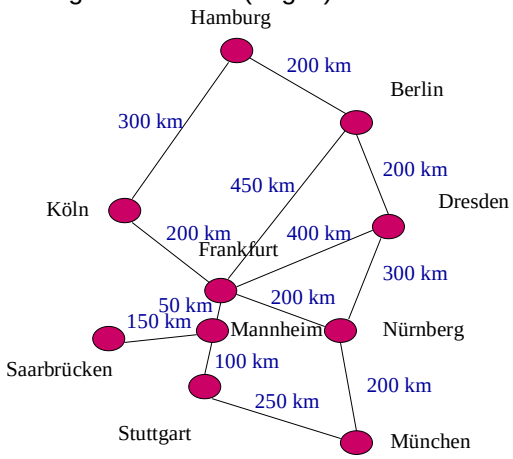
Graph-Algorithmen

1

Ein Graph G=(V,E) besteht aus Knoten und Kanten:

- V ist die Menge der Knoten (nodes or vertices) und
- E die Menge der Kanten (edges).



Knoten repräsentieren bestimmte Objekte. Eine Kante verbindet zwei Knoten, falls die beiden Knoten eine bestimmte Beziehung haben.

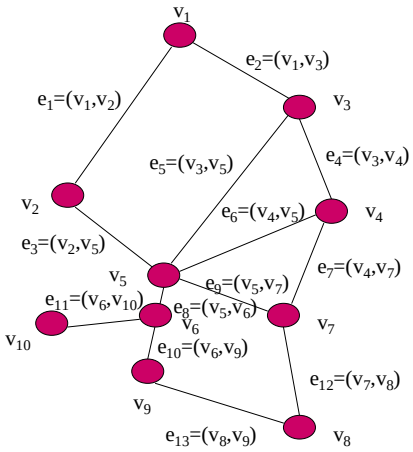
Graph-Algorithmen

2

Der i-te Knoten wird in der Regel mit v_i bezeichnet.

Die j-te Kante wird meist mit e_j bezeichnet.

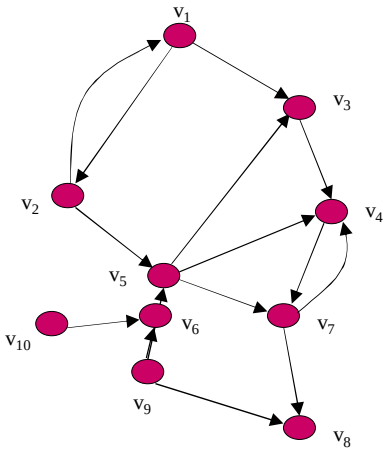
Der Grad eines Knoten ist gleich der Zahl der Kanten, die im Knoten enden.



Manchmal bezeichnen wir die Kanten auch als e_{ij} , wobei die Indizes i und j die Nummern der Knoten sind, die durch e_{ij} verbunden werden.

Graph-Algorithmen

Kanten können auch eine Richtung besitzen, wenn zum Beispiel die Beziehung, die sie repräsentieren, eine „Richtung“ besitzt (einseitig und nicht symmetrisch ist).

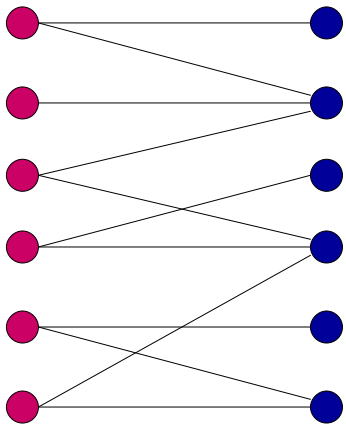


Die Kanten in einem Graphen repräsentieren oft auch Bewegungsmöglichkeiten in dem Graphen (Beispiel: Routenplaner). Falls die Kanten gerichtet sind darf man sich nur in Richtung des Pfeils „über die Kante bewegen“.

Graphen mit gerichteten Kanten werden als **gerichtete Graphen** bezeichnet, andernfalls nennt man die Graphen **ungerichtet**.

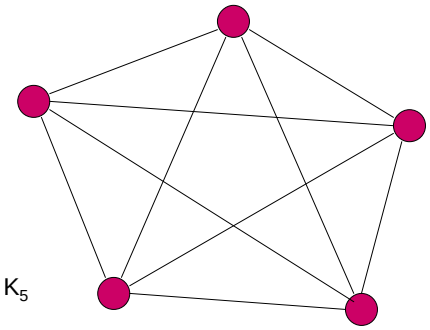
Graph-Algorithmen

Ein **bipartiter Graph** besteht aus zwei disjunkten Mengen von Knoten (rot, blau), wobei es nur Kanten zwischen Knotenpaaren mit verschiedenen Farben (ein roter und ein blauer Knoten) gibt.



Mitarbeiter Aufgaben

Existiert zwischen jedem Knotenpaar eine Kante, so spricht man von einem **vollständigen Graphen**.

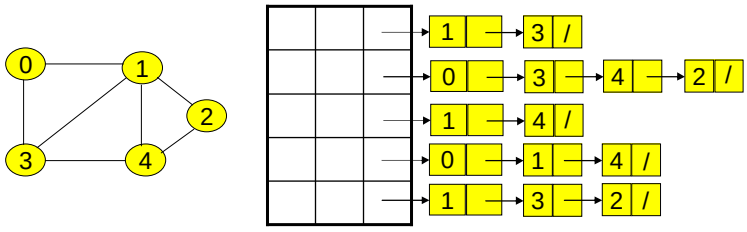


K_5

Ein vollständiger Graph mit n Knoten hat $n(n-1)/2 = O(n^2)$ Kanten.

- **Wie kann man Graphen repräsentieren?**

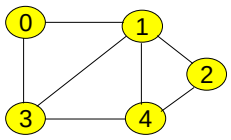
Wir speichern die Knoten in einem Feld der Größe n.



Die Kanten werden mittels einer Liste gespeichert (im obigen Bild eine einfach verkettete Liste).

- **Wie kann man Graphen repräsentieren?**

Man kann einen Graphen auch mittels einer Adjazenz-Matrix A = a_{ij} speichern. Hierbei speichert man die Knoten in einem Feld und man hat eine zusätzliche Adjazenz-Matrix, die angibt, welche Knoten über Kanten miteinander verbunden sind.



	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	1	1
2	0	1	0	0	1
3	1	1	0	0	1
4	0	1	1	1	0

Die Darstellung mittels Adjazenz-Matrix ist dann sinnvoll, wenn man oft überprüfen muss, ob zwei Knoten über eine Kante miteinander verbunden sind (Laufzeit: O(1)).

a_{ij} = 1 wenn Knoten i und j über eine Kante verbunden sind
a_{ij} = 0 sonst

Wie kann man Graphen repräsentieren?

Wir verwenden im Folgenden die Graphklasse von LEDA (Library of Efficient Data Types and Algorithms). LEDA ist eine C++-Bibliothek, die von Kurt Mehlhorn und Stefan Näher am MPII entwickelt wurde.

```
graph  G;        // Deklaration eines Graphen G
node v, w;       // Deklaration von Knoten
edge e, f ;      // Deklaration von Kanten
```

Wir nehmen zunächst für die Implementierung an, dass wir es nur mit gerichteten Graphen zu tun haben. Ungerichtete Graphen können wir zum Beispiel realisieren, indem wir jede ungerichtete Kante durch zwei gerichtete Kanten repräsentieren.

```
source(e)        // Jede Kante e hat einen Quelle (Startknoten)
target(e)        // und ein Ziel (Zielknoten, Endknoten).
```

In LEDA werden Knoten oder Kanten wie folgt mit definierten Werten initialisiert:

```
node v = nil;
```

Graph-Algorithmen


```
forall_nodes(v, G){ }
```

Iterator für Knoten: Die Anweisungen in den runden Klammern werden für jeden Knoten v von G ausgeführt (der Name v kann natürlich dort verwendet werden).

```
forall_edges(e, G){ }
```

Iterator für Kanten: Die Anweisungen in den runden Klammern werden für jede Kante e von G ausgeführt (der Name e kann dort verwendet werden).

```
forall_out_edges(e,v) { }        // Für alle Kanten, deren Quelle v ist.
forall_in_edges(e,v){ }          // Für alle Kanten, deren Ziel v ist.
forall_inout_edges(e,v){ }       // Für alle Kanten von v.
```

Iteratoren für die Kanten eines Knotens v.

`G.number_of_edges( );` liefert die Zahl der Kanten des Graphen G

Es ist oft sinnvoll zu den Knoten und Kanten zusätzliche Informationen zu speichern. LEDA stellt hier unter anderem die folgende Möglichkeit zur Verfügung:

```
node_array<bool>  marked(G, false);   // Initialisierung mit false.
edge_array<float>  length(G, 1.0);    // Initialisierung mit Länge 1.0
```

Graph-Algorithmen

Wie generiert man einen Graphen?

```
graph G;
node v0 = G.new_node();
node v1 = G.new_node();
G.new_edge(v0,v1);                // fügt eine Kante von v0 nach v1 ein
```

LEDA verfügt auch über parametrisierte Graphen:

```
GRAPH<string, int> H;

// Sei v ein Knoten von H und e eine Kante von H, dann können wir
// mittels des [ ]-Operators wie folgt Zuweisungen tätigen:
H[v] = “Saarbrücken“;
H[e] = 5;
```

Die obige Deklaration von H ordnet jedem Knoten v von H einen String und jeder Kante e von H eine ganze Zahl als zusätzliches Datenelement zu.

Als Beispielanwendung von LEDA-Graphen betrachten wir im Folgenden eine Kopierfunktion, die eine Kopie H eines Graphen G erzeugt. Jeder kopierte Knoten speichert zusätzlich den entsprechenden Originalknoten und jede kopierte Kante speichert die entsprechende Originalkante.

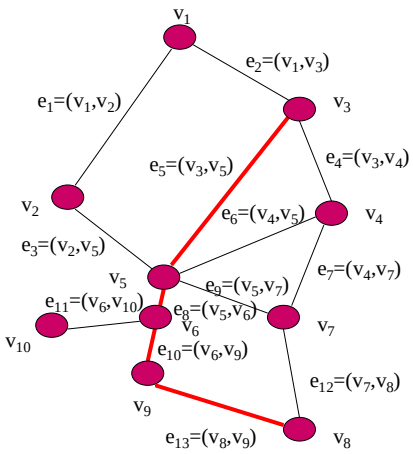
```
void CopyGraph(GRAPH<node,edge>& H, const graph& G) {
        H.clear();                    // H ist ein leerer Graph
        node_array<node>  copy_in_H(G);
        node v;
        forall_nodes(v, G)  copy_in_H[v] = H.new_node(v);
        edge e;
        forall_edges(e, G)
                H.new_edge(copy_in_H[source(e)], copy_in_H[target(e)],e);
}
```

Es gilt:

H[copy_in_H[v]] = v für alle Knoten v aus G

copy_in_H[H[w]] = w für alle Knoten w aus H

Ein **Pfad** von v_i nach v_j ist eine Folge von Knoten v_i , $v_?$, ..., v_j ,wobei jedes Paar von aufeinander folgenden Knoten v_k, v_l durch eine Kante (v_k, v_l) verbunden ist.

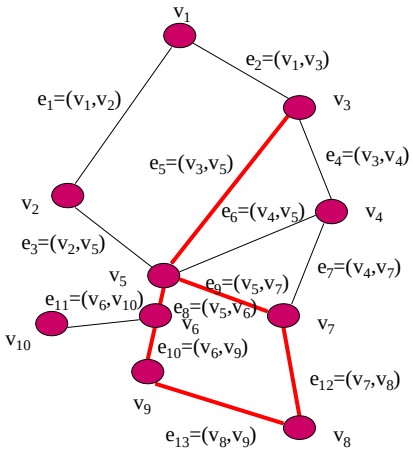


Ein Pfad wird als **einfach** bezeichnet, wenn jeder Knoten in der Pfad-beschreibung genau einmal vorkommt.

Graph-Algorithmen

Ein Pfad wird als **Circuit** (Rundgang) bezeichnet, wenn der erste und der letzte Knoten des Pfades identisch sind:

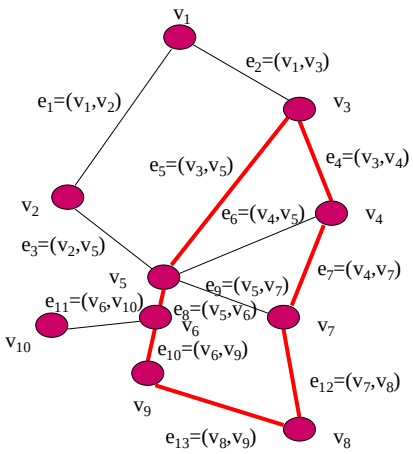
$v_3\ v_5\ v_6\ v_9\ v_8\ v_7\ v_5\ v_3$



Graph-Algorithmen

Ein **Circuit** (Rundgang) wird als einfach (oder als Kreis) bezeichnet, falls alle Knoten außer dem ersten und dem letzten nur einmal auftreten:

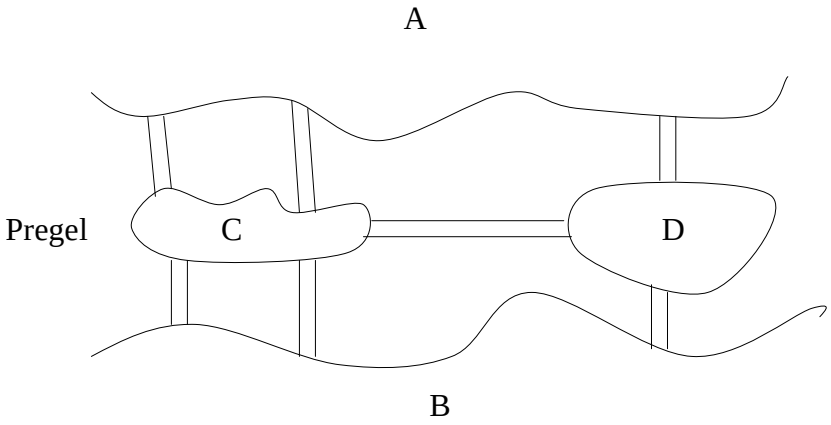
$v_3\ v_5\ v_6\ v_9\ v_8\ v_7\ v_4\ v_3$



Euler-Graphen

15

Der Schweitzer Mathematiker Leonard Euler hat sich 1736 mit der folgenden Fragestellung, dem sogenannten Königsberger-Brückenproblem, beschäftigt:

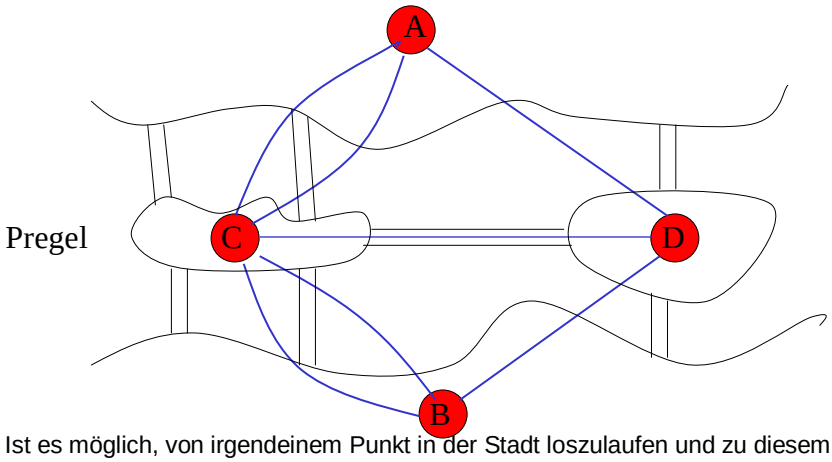


Ist es möglich, von irgendeinem Punkt in der Stadt loszulaufen und zu diesem Punkt wieder zurückzukehren und dabei genau einmal über jede der sieben Brücken zu wandern.

Euler-Graphen

16

Der Schweitzer Mathematiker Leonard Euler hat sich 1736 mit der folgenden Fragestellung, dem sogenannten Königsberger-Brückenproblem, beschäftigt:

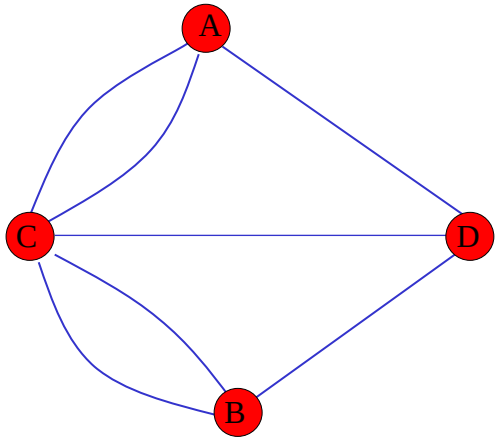


Ist es möglich, von irgendeinem Punkt in der Stadt loszulaufen und zu diesem Punkt wieder zurückzukehren und dabei genau einmal über jede der sieben Brücken zu wandern.

Euler-Graphen

17

Multi-Graph: Mehrere Kanten zwischen zwei Knoten sind erlaubt.



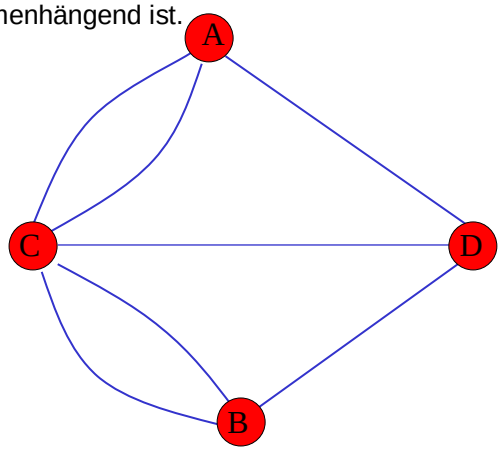
= Gibt es einen Rundgang in diesem Graphen, der jede Kante nur einmal „begeht“ (verwendet)?

Nein! Warum kann es keinen solchen Circuit geben?

Euler-Graphen

18

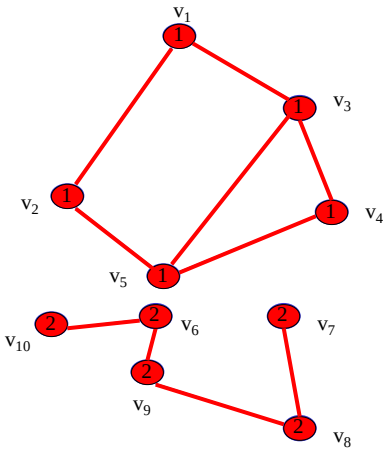
Antwort: Es gibt Knoten mit ungeradem Grad (3 und 5). Ein solcher Circuit kann aber nur existieren, wenn alle Knoten geraden Grad haben und der Graph zusammenhängend ist.



Ein Graph heißt **zusammenhängend** (connected), wenn man von jedem Knoten aus jeden anderen Knoten über einen Pfad erreichen kann.

Euler-Graphen

Zusammenhängende Teilgraphen eines Graphen bezeichnet man als Zusammenhangskomponenten:

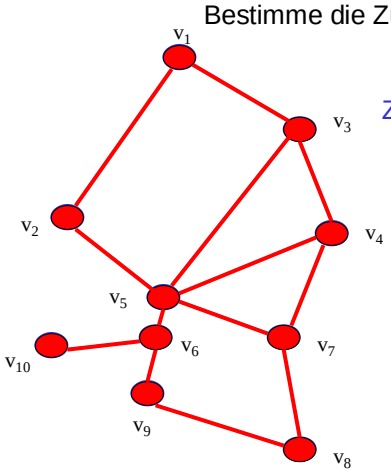


Ein Graph heißt zusammenhängend (connected), wenn man von jedem Knoten aus jeden anderen Knoten über einen Pfad erreichen kann.

Euler-Graphen

Zusammenhangskomponenten:

Gegeben ein ungerichteter Graph G=(V,E) und ein Knoten v.
Bestimme die Zusammenhangskomponente, zu der v gehört.



Zusammenhangskomponente(G,v):

1. Markiere v;
2. Für alle Kanten (v,w) führe die folgenden Operationen aus:
   a) Falls w nicht markiert ist, dann starte Zusammenhangskomponente(G,w);

Zusammenhangskomponenten:

Gegeben ein ungerichteter Graph G=(V,E) und ein Knoten v.
Bestimme die Zusammenhangskomponente, zu der v gehört.

```
// Datenstruktur zur Speicherung der Nummer der ZHK:
// node_array<int> marked(G, 0);
// Aktuelle Nummer ist „no_of_CC ≠ 0“
void  connectedComponent(      node v,
                               const graph& G,
                               node_array<int>& marked,
                               int  no_of_CC) {

        marked[v] = no_of_CC;
        edge e;
        forall_out_edges(e,v) {
            if (marked[target(e)] == 0)
                connectedComponent(target(e), G, marked, no_of_CC);
        }
}
```

Die Laufzeit des obigen Programms ist O(Zahl der Kanten der ZHK)

Zusammenhangskomponenten:

Gegeben ein ungerichteter Graph G=(V,E) und ein Knoten v.
Bestimme alle Zusammenhangskomponenten

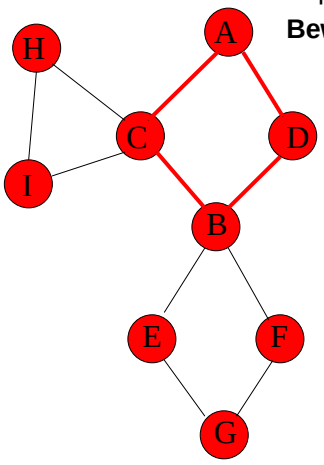
```
// Datenstruktur zur Speicherung der Nummer der ZHK:
// node_array<int> marked(G, 0);

int  connectedComponents(      const graph& G,
                               node_array<int>& marked) {
    int no_of_CC = 0;
    node v;
    forall_nodes( v , G) {
        if (marked[v] == 0)
            connectedComponent( v, G, marked, ++no_of_CC);
    }
   return no_of_CC;
}
```

Satz [1]: Die Zusammenhangskomponenten eines ungerichteten Graphen mit n Knoten und m Kanten können in Zeit O(n+m) bestimmt werden.

Euler-Graphen

Definition [1]: Ein Graph heißt **Euler-Graph**, wenn jeder Knoten geraden Grad hat und der Graph zusammenhängend ist.

Satz [2]: Ein ungerichteter Graph besitzt genau dann einen Euler-Circuit, wenn der Graph ein Euler-Graph ist.



Beweis: Wir beweisen die schwierigere Richtung (Euler-Graph => Euler Circuit) mittels vollständiger Induktion über die Zahl m der Kanten:

Induktionsstart: m = 3 (trivial).

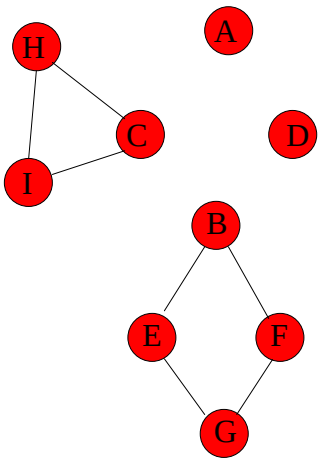
Induktionsannahme: Jeder zusammenhängende Graph mit weniger als m Kanten, dessen Knoten geraden Grad haben, besitzt einen Euler-Circuit.

Induktionsschluss:
Sei G=(V,E) ein Euler-Graph mit m Kanten.

1. Bestimme in G einen Kreis K (siehe Übung).
   Zum Beispiel K=ACBDA.
2. Lösche die Kanten von K.

Euler-Graphen

Den neuen Graphen nennen wir G‘. Alle Knoten von G‘ haben geraden Grad. Aber der Graph G‘ kann aus mehreren Zusammenhangskomponenten (ZHKs) mit weniger als m Kanten bestehen:



ZHK(1) = {A}, ZHK(2) = {D}, ZHK(3) = {H,I,C}, ZHK(4) = {B,E,G,F}

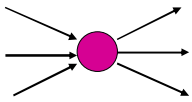
3. Bestimme die ZHKs von G‘
4. Starte für alle ZHKs von G‘ diesen (rekursiven) Algorithmus zur Berechnung eines Euler-Circuits:
   Circ(Z(3)) = CIHC Circ(Z(4)) = BEGFB
   Aufgrund der Induktionsannahme existieren für diese ZHKs Euler-Circuits.
5. Mische den Kreis K und die Euler-Circuits der ZHKs:

   Circuit = ACIHCBEGFBDA

Misch-Prozedur: Übungaufgabe. ■

Euler-Graphen

Definition [2]: Ein Knoten v eines gerichteten Graphen ist **balanciert**, wenn die Zahl der in v endenden Kanten gleich der Zahl der in v startenden Kanten ist.



Definition [3]: Ein gerichteter Graph ist **balanciert**, wenn jeder Knoten des Graphen balanciert ist.

Definition [4]: Ein gerichteter Graph heißt **Euler-Graph**, wenn der Graph zusammenhängend und balanciert ist.

Satz [3]: Ein Graph besitzt genau dann einen Euler-Circuit, wenn der Graph ein Euler-Graph ist.

Beweis: Analog zum konstruktiven Beweis im Falle des ungerichteten Graphen.

Euler-Graphen

Definition [5]: Ein Graph besitzt einen **Euler-Pfad**, wenn es einen Pfad im Graphen gibt, der jede Kante genau einmal verwendet bzw. genau einmal über diese Kante geht.

Definition [6]: Ein Knoten v eines gerichteten Graphen ist **semi-balanciert**, falls

$$| \text{indegree}(v) - \text{outdegree}(v) | = 1.$$

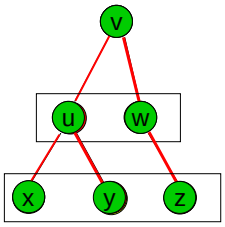
Satz [4]: Ein zusammenhängender Graph besitzt genau dann einen **Euler-Pfad**, wenn er höchstens zwei semi-balancierte Knoten besitzt und alle anderen balanciert sind.

Beweis: Analog zum Beweis der Existenz eine Euler-Circuit (mit komplexerer Fallunterscheidung).

Wie durchsucht man einen Graphen?

Breadth-first search: BFS

BFS ist einer der einfachsten Algorithmen für die Suche in Graphen:



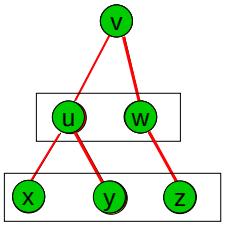
gelb : noch nicht entdeckt
rot : entdeckt, aber noch nicht abgearbeitet
grün : abgearbeitet (fertig)

Ausgehend von einem ausgezeichneten Knoten v (der Wurzel) entdeckt man alle Knoten in der Zusammenhangskomponente von v. Zunächst besucht man die direkten Nachbarn von v und fügt sie in eine Liste ein. Dann besucht man alle Nachbarn der Knoten in der Liste und fügt die noch nicht entdeckten Nachbarn dieser Knoten in die Liste ein. Man wiederholt diesen Prozess solange, bis die Liste leer ist. BFS entdeckt also die Knoten in der Reihenfolge ihres Abstands zu v. Hierbei ist der Abstand gleich der kleinsten Zahl von Kanten zwischen v und dem Knoten.

Wie durchsucht man einen Graphen?

Breadth-first search: BFS

BFS baut einen „breadth-first tree“ auf.



gelb : noch nicht entdeckt
rot : entdeckt, aber noch nicht abgearbeitet
grün : abgearbeitet (fertig)

Der Name BFS kommt daher, dass BFS die Grenze zwischen besuchten und nicht besuchten Knoten gleichförmig über die gesamte Breite der Grenze Schritt für Schritt erweitert, d.h., der Algorithmus entdeckt alle Knoten mit Abstand k von v bevor er die Knoten mit Abstand k+1 entdeckt.

BFS

29

Wie durchsucht man einen Graphen?

Wir diskutieren im Folgenden eine BFS-Implementierung, die den Abstand von jedem Knoten w in der ZHK(v) zu v mitbestimmt.

```
// Die Entfernungen der Knoten von v werden in „node_array<int> dist(G,∞)“
void  bfs(const graph& G, node v, node_array<int>& dist) {
        std::list<node> L;
        node  w, u ;
        dist[v] = 0;
        L.push_back(v);
        while (!L.empty()) {
                w = L.front();  L.pop_front();
                forall_adj_nodes(u,w) {
                        if (dist[u] == INFINITY)  {      // INFINITY wird als
                                dist[u] = dist[w] + 1;  //  größte ganze Zahl
                                L.push_back(u);       //  definiert.
                        }
                }
        }
}
```

BFS

30

Satz [4]:

Die Laufzeit von BFS für einen Graphen mit n Knoten und m Kanten ist O(n + m).

Beweis:

Jeder Knoten w wird genau einmal in die Liste der entdeckten Knoten aufgenommen und genau einmal aus dieser Liste entfernt. Dies wird durch die Abfrage der gespeicherten Entfernung dist[w]==INFINITY garantiert. Beim Abarbeiten eines Knoten schaut man sich jede Kante des Knotens an. Die Laufzeit der forall_adj_nodes-Schleife ist proportional zur Zahl der Kanten des Knotens. Da man jede Kante höchstens zweimal betrachtet (von jedem Knoten aus), ist die Gesamtlaufzeit proportional zur Zahl der Knoten und der Kanten, die betrachtet werden. ■

BFS berechnet die Distanz d(v,u) = dist[u] von v zu jedem Knoten u, der von v aus erreichbar ist, d.h., der in der ZHK von v liegt.

Definition [7]:

Hierbei verstehen wir unter Distanz d(v,u), die Länge des kürzesten Pfades von v zu u. Der kürzeste Pfad ist der Pfad mit der minimalen Zahl von Kanten.

BFS

Knoten u, die nicht von v aus erreichbar sind, haben die Distanz unendlich: $d(v,u) = \infty$.

Lemma [1]:

Sei G = (V,E) ein gerichteter oder ungerichteter Graph und sei $v \in V$ ein beliebiger Knoten. Dann gilt für jede Kante $(w, u) \in E$:

$$d(v,u) \leq d(v,w) + 1$$

Beweis:

Falls w von v aus erreichbar ist, dann ist es auch u. Dann kann der kürzeste Pfad von v nach u nicht länger sein als der kürzeste Pfad von v nach w gefolgt von der Kante (w,u). ■

Wir werden im folgenden zeigen, dass BFS die Distanzen d(v,u) als dist[u] korrekt berechnet. Wir zeigen zunächst, dass dist[u] eine obere Schranke von d(v,s) ist.

Lemma [2]:

Für jeden Knoten u in V gilt: $dist[u] \geq d(v,u)$

Beweis:

nächste Seite

BFS

Vollständige Induktion über die Zahl der „push_back“ Operationen:

Induktionsstart: i = 1: Der Knoten v wurde zuerst in die Liste eingefügt:

$$dist[v] = 0 = d(v,v) \quad \text{und} \quad dist[u] = \infty \geq d(v,u)$$

Induktionsschritt: Wir betrachten einen Knoten u, der während der Suche von einem Knoten w aus entdeckt wurde.

Aus der Induktionsannahme folgt für w: $dist[w] \geq d(v,w)$

In dieser Situation gilt:

$$\begin{aligned} dist[u] &= dist[w] + 1 \\ &\geq d(v,w) + 1 \\ &\geq d(v,u) \quad \text{Lemma [1]} \end{aligned}$$

Da sich der Wert von dist[u] im Laufe der Prozedur nicht mehr ändert, ist der Induktionsschluss vollzogen.

■

BFS

Lemma [3]:

Enthält die Knotenliste während der Ausführung von BFS die Knoten $(u_1,u_2, \dots , u_r)$, wobei u_1 der Kopf und u_r das Ende der Liste ist, dann gilt:

$$dist[u_r] \leq dist[u_1] + 1 \text{ und } dist[u_i] \leq dist[u_{i+1}] \text{ für } i = 1, 2, \dots, r-1$$

Beweis: Vollständige Induktion über die Zahl der push_back- und pop_front-Operationen:

Induktionsstart: Enthält die Liste nur v, so ist die obige Aussage trivial.

Wir müssen zeigen, dass die obige Aussage sowohl nach allen push_back-fügungen als auch nach den pop_front-Operationen gilt:

Entfernt man den Kopf u_1, so folgt die Korrektheit der Aussage direkt aus der Induktionsannahme:

$$dist[u_r] \leq dist[u_1] + 1 \leq dist[u_2] + 1$$

Fügt man einen neuen Knoten u_{r+1} bei der Suche mit Knoten w in die Liste ein, so wurde w gerade aus der Liste entfernt und es folgt aus der Induktionsannahme: $dist[w] \leq dist[u_1]$

Daher gilt:

$$dist[u_{r+1}] = dist[w] + 1 \leq dist[u_1] + 1 \text{ und } dist[u_r] \leq dist[w] + 1 = dist[u_{r+1}]$$ ■

BFS

Korollar [1]:

Wird der Knoten u_i vor dem Knoten u_j von BFS in die Knotenliste eingeführt, so gilt:

$$dist[u_i] \leq dist[u_j]$$

Satz [5]:

Sei G = (V,E) ein Graph, den man ausgehend von Knoten v mit BFS durchsucht. Dann findet BFS alle Knoten u, die von v aus erreicht werden können, und die berechnete Distanz „dist[u]“ ist gleich der Distanz d(v,u) des kürzesten Pfades von v nach u im Graphen G.

Beweis:

Nehmen wir an, dass ein Knoten eine Distanz erhält, die nicht gleich der Länge des kürzesten Pfades ist. Nehmen wir ferner an, dass s der Knoten ist, mit dem kleinsten falschen Wert d(v,s). Sicherlich ist s nicht gleich v.

Nach Lemma [2] gilt:

$dist[s] \geq d(v,s)$ Dann muss folglich $dist[s] > d(v,s)$ gelten.

Knoten s muss von v aus erreichbar sein, denn sonst ist $d(v,s) = \infty = dist[s]$.

BFS

35

Sei u der Knoten, der auf dem kürzesten Pfad von v nach s liegt und von dem aus man direkt nach s gelangt, d.h., die letzte Kante des kürzesten Pfades ist (u,s):

$$d(v,u) + 1 = d(v,s)$$

Wegen der Art und Weise, wie wir s definiert haben, muss folgendes gelten:

$$dist[u] = d(v, u)$$

Fassen wir die letzten beiden Aussagen zusammen, so erhalten wir:

$$dist[s] > d(v,s) = d(v,u) + 1 = dist[u] + 1 \qquad ***$$

Nun betrachten wir den Zeitpunkt, zu dem u aus der Liste entfernt wurde. Wir betrachten nun drei Fälle und zeigen, dass wir in jedem Fall einen Widerspruch zur Annahme erhalten:

(1) s wurde von u aus entdeckt:

Dann wird von der Prozedur dist[s] = dist[u] + 1 gesetzt, was der vorhergehenden Ungleichung (***) widerspricht.

(2) s wurde bereits vor u aus der Liste entfernt:

Dann wurde s bereits aus der Liste entfernt und aufgrund von Korollar [1] muss $dist[s] \leq dist[u]$ gelten, was der obigen Ungleichung (***) widerspricht.

BFS

36

(3) u wird vor s aus der Liste entfernt, aber s ist bereits entdeckt worden:

Dann wurde s entdeckt, als ein u vorhergehender Knoten w aus der Liste entfernt wurde, für den gilt:

$$dist[s] = dist[w] + 1$$

Nach Korollar [1] gilt aber:

$$dist[w] \leq dist[u]$$

Hieraus folgt:

$$dist[s] \leq dist[u] + 1$$

Dies liefert wiederum einen Widerspruch zur Ungleichung (***).

Somit haben wir gezeigt, dass für alle $s \in V$ gilt: $dist[s] = d(v,s)$.

■

BFS

37

Wir ändern nun die Prozedur BFS so ab, dass wir auch den BFS-Baum problemlos bestimmen können.

```
// Die Eltern im BFS-Baum speichern wir in „node_array<node> parent(G,nil)“
void  bfs(        const graph& G, node v, node_array<int>& dist
                  node_array<node>& parent) {
        std::list<node> L;
        node  w, u ;
        dist[v] = 0;
        L.push_back(v);
        while (!L.empty()) {
                w = L.front();  L.pop_front();
                forall_adj_nodes(u,w) {
                        if (dist[u] == INFINITY)  {
                                dist[u] = dist[w] + 1;
                                L.push_back(u);
                                parent[u] = w;          //  w ist der Vater von u
                        }
                }
        }
}
```

BFS

38

Definition [8]:

Sei G = (V,E) ein Graph mit Quelle v. Wir definieren den Predecessor-Graph von G als $G_p = (V_p , E_p)$, wobei

$$V_p = \{s \in V \mid parent[s] \neq nil\} \cup \{v\}$$

und

$$E_p = \{(parent[s], s) \mid s \in V_p \setminus \{v\}\}$$

Lemma [4]:

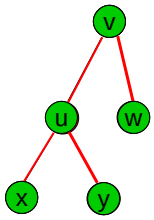
Wendet man BFS auf einen gerichteten oder ungerichteten Graphen G = (V,E) an, so ist der berechnete Predecessor-Graph ein Baum, der sogenannte BF-Tree(Breadth-First Tree).

Der Pfad von der Wurzel v zu einem Knoten s entspricht gerade einem kürzesten Pfad von v nach s. Die Kanten des Baumes geben an, von welchem Knoten (parent[s]) aus jeder erreichbare Knoten s entdeckt wurde.

Wie durchsucht man einen Graphen?

Depth-first search: DFS

Man taucht man bei DFS immer so „tief“ wie möglich in den Graphen ein.



gelb : noch nicht entdeckt
rot : entdeckt, aber noch nicht abgearbeitet
grün : abgearbeitet (fertig)

Ist die Suche im Knoten v angekommen, so testen wir alle Kanten von v und bestimmen den ersten Nachbarknoten u, der noch nicht entdeckt wurde. Dann bewegen wir uns nach u und fahren analog fort, d.h., wir testen alle Kanten und suchen den ersten Nachbarknoten x, der noch nicht entdeckt wurde. Falls ein Knoten keine nicht entdeckten Kinder mehr besitzt, dann bewegen wir uns zurück zum „vorhergehenden“ Knoten und testen, ob er abgearbeitet ist oder nicht. Falls nicht, suchen wir den nächsten nicht besuchten Nachbarn usw.

Wie durchsucht man einen Graphen?

Wir diskutieren im Folgenden eine DFS-Implementierung, die die Reihenfolge, in der die Knoten der ZHK von v erreicht werden, mitberechnet und die verwendeten Kanten des DFS-Baums in einer Liste speichert:

```
// Die Reihenfolge wird in „node_array<int> no(G,-1)“
// gespeichert. Die Kanten speichern wir in der Liste „list<edge> T“.
static int  counter = 0;

void  dfs(const graph& G, node v, node_array<int>& no, list<edge>& T) {
          no[v] = counter++;
          edge e;
          forall_out_edges(e,v) {
              node w = target(e);
              if (no[w] == -1) {
                    T.push_back(e);  // falls erforderlich parent[w] = v;
                    dfs(G, w, no, T);
              }
          }
}
```

DFS

Wir setzen nun voraus, dass wir die obige DFS-Prozedur so abgeändert haben, dass wir die Bäume von allen ZHKs von G berechnet haben (triviale Übung).

Eigenschaften von DFS

Satz [6]:

Die Laufzeit von DFS für einen Graphen G = (V,E) mit n Knoten und m Kanten ist O(n + m) = O(|V| + |E|).

Definition [9]:

Sei G = (V,E) ein Graph. Wir definieren den DFS-Predecessor-Graph von G als $G_p = (V , E_p)$, wobei

$$E_p = \{(parent[s], s) \mid s \in V \wedge parent[s] \neq nil\}$$

Der Predecessor-Graph von DFS bildet einen „Depth-First Forest“, der sich aus mehreren „Depth-First Trees“ zusammensetzen kann. Die Kanten in E_p nennt man die Baumkanten.

DFS

Wir führen nun eine Zeitmessung in die DFS-Prozedur ein:

```
static int  counter = 0;
static int  dfs_time = 0;              //  Unsere Uhr

void  dfs(const graph& G, node v, node_array<int>& no, list<edge>& T,
          node_array<int>& detection_time,          // Zeit der Entdeckung
          node_array<int>& finishing_time ) {       // Knoten ist abgearbeitet
          no[v] = counter++;
          edge e;
          forall_out_edges(e,v) {
              node w = target(e);
              if (no[w] == -1) {
                    detection_time[w] = ++dfs_time; // neuer Knoten entdeckt
                    T.push_back(e);
                    dfs(G, w, no, T, detection_time, finishing_time);
                    finishing_time[w] = ++dfs_time;  // Knoten abgearbeitet
              }
          }
}
```

Eigenschaften von DFS

Wir verwenden im Folgenden die Abkürzungen DT und FT für „detection_time“ und „finishing_time“:

Satz [7]:

Nach Anwendung von DFS auf einen Graphen G gilt für zwei beliebige Knoten u und v des Graphen eine der drei folgenden Bedingungen:

[1] Die Zeitintervalle [DT[u], FT[u]] und [DT[v], DT[v]] sind disjunkt und weder v noch u sind Nachfahren des anderen Knotens im DF Forest.

[2] Das Intervall [DT[u], FT[u]] ist vollständig im Intervall [DT[v], FT[v]] enthalten und u ist ein Nachfahre von v in einem Baum des DF Forest.

[3] Das Intervall [DT[v], FT[v]] ist vollständig im Intervall [DT[u] , FT[u]] enthalten und v ist ein Nachfahre von u in einem Baum des DF Forest.

Beweis: Wir betrachten den Fall DT[u] < DT[v].

Im symmetrischen Fall vertausche man einfach die Rollen von v und u.

Zunächst betrachten wir den Teilfall, dass DT[v] < FT[u], d.h., v wurde entdeckt, bevor u abgearbeitet ist.
Dies impliziert, dass v ein Nachfahre von u ist, und dass aufgrund der rekursiven Implementierung die Kanten von v bereits abgearbeitet worden sind, bevor die Prozedur zum Knoten u zurückkehrt. In diesem Fall ist [DT[v], FT[v]] vollständig in dem Intervall [DT[u],FT[u]] enthalten und v ist ein Nachfahre von u [Fall 3].

Wir betrachten nun den zweiten Teilfall, dass FT[u] < DT[v]. In diesem Fall müssen die Zeitintervalle disjunkt sein und daher ist weder v noch u ein Nachfahre des anderen Knotens. ■

Aus dem obigen Satz folgt direkt:

Korollar [2]:

Knoten v ist genau dann eine Nachfahre von Knoten u im DF Forest, wenn gilt:

$$DT[u] < DT[v] < FT[v] < FT[u]$$

DFS

Satz [8]:

Im DF Forest eines gerichteten oder ungerichteten Graphen G = (V,E) ist ein Knoten v genau dann ein Nachfahre von Knoten u, falls zu der Zeit, wenn u entdeckt wird (DT[u]), ein Pfad von u nach v existiert, der nur gelbe (noch nicht entdeckte) Knoten enthält.

Beweis:

=>: Wir nehmen an, dass v ein Nachfahre von u in einem Baum des DF Forest ist. Sei w irgend ein Knoten auf dem Pfad zwischen u und v im entsprechenden DF Tree.

Dann gilt nach Korollar [2] DT[u] < DT[w] und daher sind alle Knoten w mit der obigen Eigenschaft gelb (noch nicht endeckt).

<=: Wir nehmen an, dass zum Zeitpunkt DT[u] der Knoten v von u aus über einen Pfad mit gelben Knoten erreicht werden kann, dass aber v nicht zu einem Nachkommen von u im DF Forest wird.

O.B.d.A. nehmen wir an, dass jeder Knoten auf dem Pfad vor v eine Nachfahre von Knoten u im DF Forest wird (andernfalls betrachten wir den Knoten, der u am nächsten ist und diese Eigenschaft besitzt).

Sei w der Vorgänger von v auf dem „gelben“ Pfad. Nach Korollar [2] gilt:

FT[w] < FT[u]

DFS

Man beachte, dass v nach u entdeckt wird, aber bevor w abgearbeitet ist.

Daher muss gelten:

DT[u] < DT[v] < FT[v] < FT[u]

Aus Satz [7] folgt dann, dass das Intervall [DT[v], FT[v]] in dem Intervall [DT[u],FT[u]] vollständig enthalten sein muss und dass v ein Nachfahre von u im DF Forest sein muss.

Klassifikation von Kanten des Graphen G = (V,E)

DFS kann zur Klassifikation der Kanten des Graphen G verwendet werden:

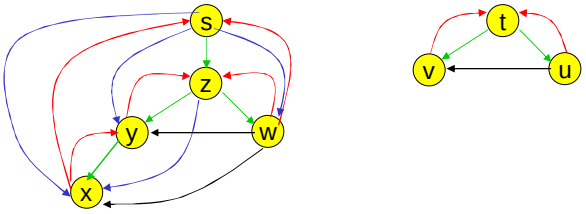
[1] **Tree edges** sind Kanten des DF Forest G_p. Die Kante (u,v) ist eine Baumkante, falls v über die Kante (u,v) entdeckt wurde.

[2] **Back edges** sind Kanten (u,v), die einen Knoten u mit einem Vorfahren v in einem DF Tree verbinden.

[3] **Forward edges** sind Kanten (u,v), die nicht zum DF Forest gehören und einen Knoten u mit einem Nachfahren v verbinden.

[4] **Cross edges** sind die anderen Kanten.

DFS



Klassifikation von Kanten des Graphen G = (V,E)

DFS kann zur Klassifikation der Kanten des Graphen G verwendet werden:

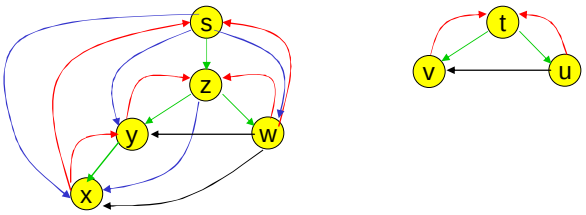
[1] **Tree edges** sind Kanten des DF Forest G_p. Die Kante (u,v) ist eine Baumkante, falls v über die Kante (u,v) entdeckt wurde.

[2] **Back edges** sind Kanten (u,v), die einen Knoten u mit einem Vorfahren v in einem DF Tree verbinden.

[3] **Forward edges** sind Kanten (u,v), die nicht zum DF Forest gehören und einen Knoten u mit einem Nachfahren v verbinden.

[4] **Cross edges** sind die anderen Kanten.

DFS



Man kann DFS so modifizieren, dass man die Kanten (u,v) klassifizieren kann: Hierzu betrachten wir den Knoten v, der von u aus erreicht wird, wenn die Kante (u,v) zum ersten Mal untersucht wird.

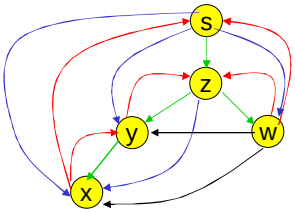
Ist v gelb (noch nicht endeckt), dann ist (u,v) eine Baumkante.

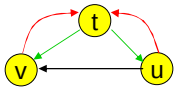
Ist v rot (entdeckt, aber nicht abgearbeitet), dann ist (u,v) eine Rückwärts-Kante.

Ist v grün (abgearbeitet), dann ist (u,v) eine Vorwärts-Kante oder eine Quer-Kante .

Da in einem ungerichteten Graphen (u,v) und (v,u) die gleiche Kante repräsentieren, ordnen wir die Kante in die erste zutreffende Klasse (in der vorgegebenen Reihenfolge) ein.

DFS





Satz [9]:

Die DFS-Klassifizierung eines ungerichteten Graphen G = (V,E) ordnet jede Kante von G entweder in die Klasse Baumkante oder in die Klasse Rückwärts-Kante ein.

Beweis: Sei (u,v) eine beliebige Kante von G und sei o.B.d.A

DT[u] < DT[v]

Dann wird v entdeckt und abgearbeitet, bevor u fertigt wird, denn v ist ein Nachbar von u, d.h., es gibt die Kante (u,v). Der Knoten v muss also ein Nachfahre von u sein. Wurde v entdeckt, als die Kante (u,v) von u aus betrachtet wurde, so ist (u,v) eine Baumkante. Betrachtete man die Kante (u,v) zum ersten Mal von v aus, dann ist (u,v) eine Rückwärts-Kante. Der Fall DT[u] > DT[v] geht analog. ■

Topologisches Sortieren

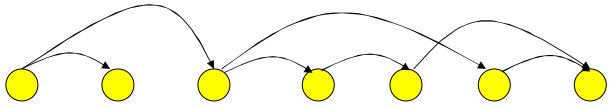
Topologisches Sortieren mittels DFS

Definition [10]

Ein DAG (directed acyclic graph) ist ein gerichteter azyklischer Graph, d.h., ein gerichteter Graph, der keine Rundgänge besitzt.

Definition [11]

Eine topologische Sortierung eines DAG ist eine (lineare) Ordnung aller Knoten, so dass für alle Kanten (u,v) des Graphen gilt: Der Knoten u erscheint in der Ordnung vor v.



Topological-Sort(G):

(1) Starte DFS und berechne die „finishing-time“ (FT) für alle Knoten.

(2) Wenn ein Knoten abgearbeitet ist, dann füge ihn am Anfang der (Ordnungs-)Liste ein.

(3) Gebe die Liste zurück.

Topologisches Sortieren mittels DFS

Lemma [5]

Ein gerichteter Graph ist genau dann azyklisch, wenn DFS keine Rückwärts-Kanten produziert (findet).

Beweis: =>: Wir nehmen an, dass es eine Rückwärts-Kante (u,v) gibt. Dann ist u ein Nachfahre von v im DF Forest und es gibt folglich einen Pfad von v nach u. Fügen wir die Kante (u,v) an diesen Pfad, so erhalten wir einen Kreis. Folglich ergibt sich ein Widerspruch zur Annahme, dass wir es mit einem azyklischen Graphen zu tun haben.

<=: Wir nehmen an, dass G einen Zyklus C enthält, und zeigen, dass unter diesen Voraussetzungen DFS eine Rückwärts-Kante findet. Sei v der erste Knoten des Zyklus C, den DFS entdeckt, und sei (u,v) die Kante im Zyklus, die zu v führt. Zur Zeit DT[v] gilt: Der Teilpfad des Zyklus C von v nach u besteht nur aus gelben Knoten. Daher folgt aus Satz [8], dass u ein Nachfahre von v im DF Forest ist. Also ist (u,v) eine Rückwärts-Kante. ■

Topologisches Sortieren mittels DFS

Satz [10]

Topological-Sort produziert eine topologische Sortierung auf einem DAG.

Beweis: Es genügt zu zeigen, dass für die Knoten jeder Kante (u,v) im DAG gilt: $FT[v] < FT[u]$

Sei (u,v) eine Kante, die von DFS bearbeitet wird. Zum Zeitpunkt der Bearbeitung der Kante (u,v) kann v nicht rot (entdeckt, aber noch nicht abgearbeitet) sein, denn dann hätte DFS eine Rückwärts-Kante gefunden (nach vorhergehendem Lemma ist das ein Widerspruch zur Vorrausetzung, dass der Graph ein DAG ist). Folglich ist v entweder gelb (noch nicht entdeckt) oder grün (abgearbeitet). Falls v gelb ist, wird v ein Nachfahre von u und deshalb gilt:

$$FT[v] < FT[u]$$

Ist v grün, so ist der Knoten bereits abgearbeitet und FT[v] ist schon gesetzt worden. Da wir aber den Knoten u noch bearbeiten, wird FT[u] noch gesetzt und es gilt deshalb:

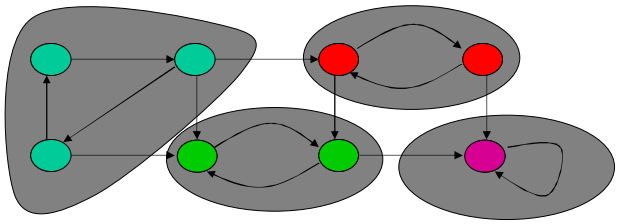
$$FT[v] < FT[u]$$ ■

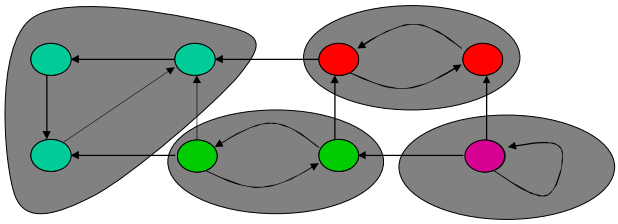
Strongly connected components (Starke Zusammenhangskomponenten)

Definition [12]:

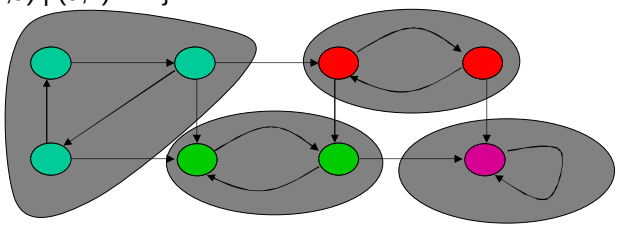
[a] Eine starke Zusammenhangskomponente eines gerichteten Graphen G=(V,E) ist ein maximale Knotenmenge $C \subseteq V$, so dass für jedes Paar $u,v \in C$ gilt: Der Knoten u kann von v aus über einen Pfad, der vollständig zu C gehört, erreicht werden und der Knoten v kann von u aus über einen Pfad, der vollständig zu C gehört, erreicht werden.

[b] Ein gerichteter Graph wird als stark zusammenhängend bezeichnet, wenn er aus einer einzigen starken Zusammenhangskomponente besteht.





Um die starken Zusammenhangskomponenten zu berechnen, betrachten wir neben G auch den transponierten Graphen $G^T = (V,E^T)$ von G, wobei $E^T = \{ (v,u) \mid (u,v) \in E \}$.



Beobachtung: G und G^T haben die gleichen starken Zusammenhangskomponenten.

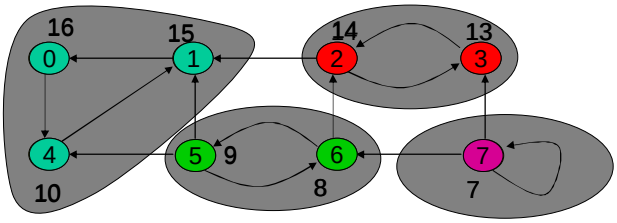
Strongly-Connected-Components(G): SCC(G):

(1) Berechne die „finishing_time“ (FT) für jeden Knoten mit DFS(G).

(2) Berechne den transponierten Graphen G^T

(3) Berechne DFS(G^T), wobei die Knoten in der Reihenfolge ihrer „finishing_time“s aus der DFS(G) Berechnung in Schritt (1) (fallend) in der Hauptschleife von DFS(G^T) abgearbeitet werden.

(4) Gib die Knoten eines jeden Baumes des DF Forest, der in Zeile (3) berechnet wurde, als eine starke Zusammenhangskomponente aus.

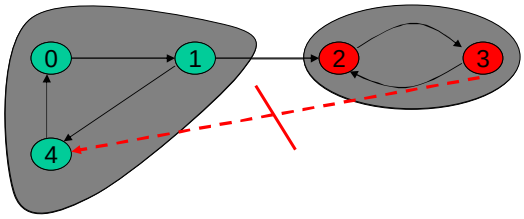



```
static int  counter = 0;
static int  dfs_time = 0;              //  Unsere Uhr

void  DFS(const graph& G, node_array<int>& no, list<edge>& T,
          node_array<int>& detection_time,        // Zeit der Entdeckung
          node_array<int>& finishing_time ) {     // Knoten ist abgearbeitet
          node v;
          forall_nodes(v,G) {
              if (no[v] == -1) dfs(G, v, no, T,detection_time, finishing_time);
          }
}
```

(3) Berechne DFS(G^T), wobei die Knoten in der Reihenfolge ihrer „finishing_time“s aus der DFS(G) Berechnung in Schritt (1) (fallend) in der Hauptschleife von DFS abgearbeitet werden.

Starke Zusammenhangskomponenten

57



Lemma [6]:

Seien C und C‘ zwei verschiedenen starke Zusammenhangskomponenten in einem gerichteten Graph G = (V,E) . Seien ferner u, v ∈ C und u‘, v‘ ∈ C‘. Gibt es einen Pfad u →u‘ von u nach u‘ in G, dann kann es keinen Pfad v‘ → v von v‘ nach v geben.

Beweis: Gäbe es einen Pfad von v nach v‘, dann wären weder C noch C‘ starke Zusammenhangskomponenten, da dann jeder Knoten von C von jedem Knoten von C‘ aus erreichbar wäre und umgekehrt.

■

Starke Zusammenhangskomponenten

58

Wenn wir im folgenden „detection_time“ und „finishing_time“ von Knoten diskutieren, so beziehen wir uns auf den ersten Aufruf der DFS-Prozedur.

Bezeichnungen:

Sei U ⊆ V eine Teilmenge der Knoten:

d(U) = min { DT[u] | u∈U }

f(U) = max { FT[u] | u∈U }

Lemma [7]:

Seien C und C‘ zwei verschiedene starke Zusammenhangskomponenten des gerichteten Graphen G=(V,E). Gibt eine Kante (u,v) mit u ∈ C und v ∈ C‘, dann gilt:

f(C) > f(C‘)

Beweis: Fall 1: d(C) < d(C‘):
Sei x der erste Knoten in C, der entdeckt wird : d(C) = DT[x]
Zu diesem Zeitpunkt sind alle Knoten von C und C‘ außer x gelb.
Zu jedem Knoten von C und C‘ existiert von x aus ein gelber Pfad.
Satz [8] impliziert, dass alle Knoten in C und C‘ Nachfahren von x sind. Aus Korollar [2] folgt, dass FT[x] = f(C) > f(C‘).

Starke Zusammenhangskomponenten

Fall 2: d(C) > d(C‘):

Sei y der erste Knoten in C‘, der entdeckt wird : d(C‘) = DT[y]

Zu diesem Zeitpunkt sind alle Knoten von C‘ \ y gelb.

Zu jedem Knoten von C‘ existiert von y aus ein gelber Pfad.

Satz [8] impliziert, dass alle Knoten in C‘ Nachfahren von y im DFF sind.

Aus Korollar [2] folgt, dass FT[y] = f(C‘).

Da es eine Kante (u,v) von C nach C‘ gibt, kann nach Lemma 6 kein Pfad von C‘ nach C existieren.

Deshalb ist kein Knoten von C von C‘ aus erreichbar und alle Knoten in C sind immer noch gelb, wenn alle Knoten von C‘ abgearbeitet sind.

Daher gilt für jeden Knoten w von C:

$$FT[w] > f(C')$$

Hieraus folgt:

$$f(C) > f(C')$$

■

Starke Zusammenhangskomponenten

Korollar [3]:

Seien C und C‘ zwei verschiedene starke Zusammenhangskomponenten in einem gerichteten Graphen G = (V,E). Gibt es eine Kante (u, v) $\in E^T$ im transponierten Graphen G, wobei u $\in$ C und v $\in$ C‘ ist, dann ist f(C) < f(C‘).

Beweis:

Da (u,v) $\in E^T$, ist (v,u) $\in$ E. Da G und G^T die gleichen starken Zusammenhangskomponenten enthalten, folgt aus dem vorhergehenden Lemma, dass f(C) < f(C‘) ist. ■

Das obige Korollar liefert den Schlüssel zum Verständnis von SCC(G): Betrachten wir hierzu den zweiten Aufruf der Prozedur DFS(G^T) auf dem transponierten Graphen G^T = (V,E^T): Die Knoten werden beginnend mit dem Knoten mit der größten „finishing_time“ in absteigender Reihenfolge abgearbeitet:

Aus Korollar [3] folgt:

Bearbeiten wir gerade eine starke Zusammenhangskomponente C‘, so gibt es keine Kanten aus Knoten von C‘, die zu nicht entdeckten Knoten in anderen starken Zusammenhangskomponenten führen, d.h., ⇨

Starke Zusammenhangskomponenten

61

Betrachten wir eine Kante (u,v) mit u ∈ C‘, so ist v

- entweder ein bereits entdeckter und abgearbeiteter Knoten einer sZHK C, die bereits abgearbeitet wurde, d.h., f(C) > f(C‘).
- oder ein entdeckter oder noch nicht entdeckter Knoten von C‘.

Die obige Aussage besagt, dass die neuentdeckten Knoten alle zur starken Zusammenhangskomponente C‘ gehören.

Frage: Findet man auch alle Knoten von C‘?

Da C‘ eine sZHK ist, gibt es von jedem Knoten x, der als erster entdeckt wird, einen (gelben) Pfad zu jedem anderen Knoten der starken Zusammenhangskomponente C‘.

Satz [11]:

Die Prozedur SCC(G) berechnet die sZHKn eines gerichteten Graphen G korrekt.

Beweis:

Vollständige Induktion über die Zahl k der Bäume, die in Zeile (3) der Prozedur SCC(G) beim Aufruf von DFS(G^T) berechnet und konstruiert wurden.

Starke Zusammenhangskomponenten

62

Die Induktionsannahme ist, dass die ersten k Bäume, die in Zeile (3) von SCC berechnet wurden, sZHK von G sind.

Wir betrachten den k+1-ten DF Baum, der in Zeile (3) berechnet wird.

Sei u die Wurzel dieses Baumes und u gehöre zur sZHK C. Es gilt:

FT[u] = f(C) > f(C‘)

für alle noch nicht bearbeiteten (noch nicht entdeckten) sZHKn C‘.

Da u der erste entdeckte Knoten von C ist, gilt zum Zeitpunkt DT[u], dass alle anderen Knoten von C zu diesem Zeitpunkt gelb sind. Dann folgt aus Satz [8], dass alle anderen Knoten von C Nachfahren von u im k+1-ten DF Baum sind.

Aus Korollar [3] und der Induktionsannahme folgt ferner, dass alle Kanten in E^T, die C verlassen, auf Knoten in bereits entdeckten und abgearbeiteten sZHKn zeigen.

Deshalb wird kein Knoten einer anderen sZHK ein Nachfahre von u im k+1-ten DF Baum sein.

Daher bilden die Knoten des k+1-ten DF Baumes eine starke Zusammenhangskomponente (die k+1-te starke Zusammenhangskomponente). ■

Der Komponenten-Graph $G^{SCC} = (V^{SCC}, E^{SCC})$:

Für jede starke Zusammenhangskomponente C_i addiert man einen Knoten v_i zur Knotenmenge V_{scc}. Es wird genau dann eine gerichtete Kante (v_i, v_j) zur Kantenmenge E^{SCC} hinzugefügt, die die starken Zusammenhangskomponenten C_i und C_j verbindet, wenn es mindestens eine gerichtete Kante in E^T von einem Knoten $x \in C_i$ zu einem Knoten $y \in C_j$ gibt.