Inhalt:

1. Tiefensuche

1.1. Vorgehensweise

Gegeben sei ein Graph *G*, die Menge seiner Knoten *V* und die Menge der Kanten zwischen den Knoten *E*. Weiterhin sei die Listendarstellung des Graphen *G* gegeben.

Ausgehend von einem beliebigen Knoten *s* des Graphen *G* werden immer zuerst die Kanten des zuletzt entdeckten Knoten *v* durchsucht. Sind alle Kanten von *v* durchsucht, springt man zu dem Knoten *u* zurück, von dem aus *v* entdeckt wurde usw... Dieses Verfahren endet erst, wenn alle Kanten von *s* durchsucht wurden, d.h. die Adjazenzliste von *s* komplett durchlaufen ist, und man wieder zu *s* „zurückgesprungen" ist. Sollten noch unentdeckte Knoten übrig bleiben, wählt man aus der Menge der übrigen Knoten einen als neuen Startknoten und beginnt dort von neuem.
Wie bei der Breitensuche auch wird bei der Tiefensuche das Vorgänger-Feld $\pi(v)$ eines Knotens *v* auf *u* gesetzt, wenn *v* von *u* aus zuerst entdeckt wurde, d.h. wenn $\pi(v)$ zum Zeitpunkt der Entdeckung noch ungesetzt war.
Wiederum analog zur Breitensuche werden auch bei der Tiefensuche die Knoten „eingefärbt", um ihren aktuellen Status anzuzeigen. Darüberhinaus versieht man bei der Tiefensuche jeden Knoten mit zwei Zeitmarkierungen, dem Zeitpunkt seiner Entdeckung *d[v]* und dem Zeitpunkt, an dem die Erforschung seiner Kanten abgeschlossen wurde *f[v]*.
Jeder Knoten beginnt weiß. Wird er entdeckt, so wird er grau eingefärbt und die aktuelle Zeitvariable in *d[v]* gespeichert. Wurden alle von *v* ausgehenden Kanten vollständig erforscht, wird der Knoten schwarz eingefärbt und die aktuelle Zeitvariable in *f[v]* gespeichert.

1.2. Algorithmus

```
DFS(G)
1    for each vertex u in V
2         do   color[u] = WHITE
3              π[u] = NIL
4    time = 0
5    for each vertex u in V
6         do if color[u] = WHITE
7              then DFS-VISIT(u)

DFS-VISIT(u)
1    color[u] = GRAY
2    time = time + 1
3    d[u] = time
4    for each vertex v in Adj[u]
5         do if color[u] = WHITE
6              then π[v] = u
7                   DFS-VISIT(v)
8    color[u] = BLACK
9    time = time + 1
10   f[u] = time
```

1.3. Erklärung des Algorithmus

Das Feld *color[u]* jedes Knotens *u* enthält dessen Farbe, das Feld $\pi[u]$ seinen Vorgänger, *d[u]* enthält den Zeitpunkt der Entdeckung des Knotens und *f[u]* den Zeitpunkt, zu dem die Durchsuchung aller Knoten in der Adjazenzliste *Adj[u]* abgeschlossen wurde. *time* ist eine globale Variable und enthält die verstrichene Zeit seit Beginn der Tiefensuche.

Die Methode DFS:

Die for-Schleife in den Zeilen 1 bis 3 bereitet den gegebenen Graphen *G* auf die Tiefensuche vor, indem die Farbe jedes Knotens *u* der Menge *V* auf weiß gesetzt wird und das Vorgängerfeld $\pi[u]$ gelöscht wird.
In Zeile 4 wird die Zeitvariable *time* mit 0 initialisiert. Danach wird jedem Knoten *u*, der weiß ist, mit der Methode DFS-VISIT „ein Besuch abgestattet", d.h. der Knoten *u* wird durchsucht, falls er weiß ist.

Die Methode DFS-VISIT:

In Zeile 1 wird die Farbe des übergebenen Knotens *u* auf grau gesetzt, also als entdeckt markiert. (vgl. Zeile 6 in DFS und Zeile 5 in DFS-VISIT) Zeile 2 inkrementiert die Zeitvariable, die in Zeile 3 in die Entdeckungs-Zeitmarkierung von *u*, *d[u]* gespeichert wird.
Nun durchläuft die For-Schleife in den Zeilen 4 bis 7 alle Knoten *v* in der Adjazenzliste von *u* (*Adj[u]*) und überprüft deren Farbe *color[v]*. Ist *v* weiß, so wird sein Vorgängerfeld $\pi(v)$ mit *u* belegt und *v* wird durchsucht.
Wurden alle weißen Knoten in der Adjazenzliste von *u* durchsucht, wird *u* in Zeile 8 schwarz markiert, die Zeitvariable wiederum inkrementiert (Zeile 9) und in der Zeitmarkierung für das Ende der Durchsuchung *f[u]* gespeichert (Zeile 10).

1.4. Durchführung an einem Beispielgraphen

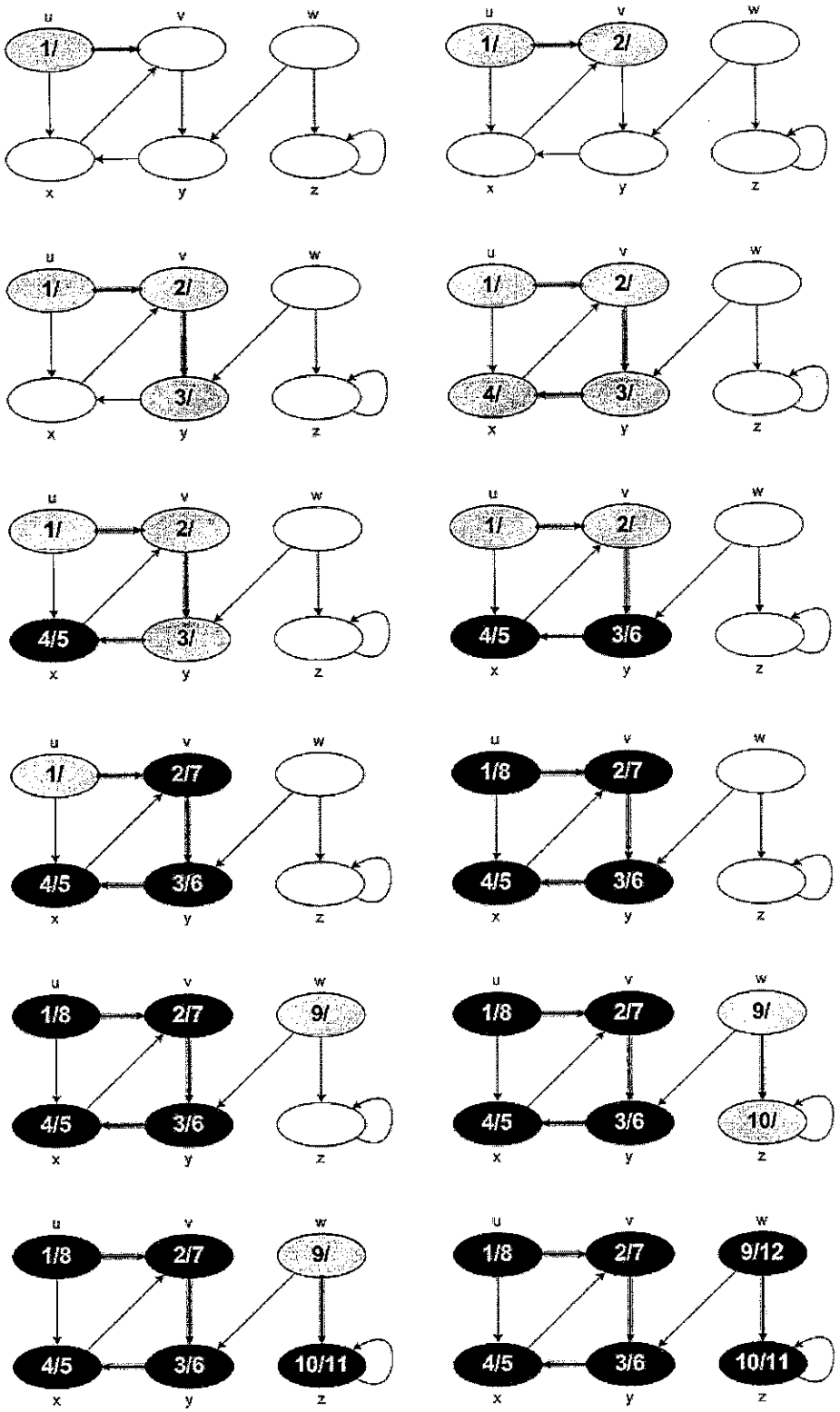


Bild 1: Tiefensuche am Beispiel eines gerichteten Graphen. Kanten, die im Verlauf der Tiefensuche verfolgt werden, werden schattiert dargestellt. Die Zeitmarkierungen für den Beginn und das Ende der Durchsuchung stehen in den Knoten.

1.5. Terminierung und Aufwandsberechnung

Jeder Knoten *u* wird höchstens einmal durchsucht, denn DFS-VISIT setzt in der ersten Zeile die Farbe auf grau und nur weiße Knoten werden durchsucht, sowohl in DFS und DFS-VISIT. Damit ist bereits gezeigt, dass der Algorithmus terminiert, denn beide Methoden enden, wenn die Liste aller Knoten einmal durchlaufen wurde.
Die For-Schleifen in den Zeilen 1 bis 3 bzw 5 bis 7 in der Methode DFS benötigen jeweils $O(V)$ Zeit ohne den Aufruf von DFS-VISIT.
DFS-VISIT wird für jeden Knoten *v* aus *V* genau einmal aufgerufen, da es nur für weiße Knoten aufgerufen wird und in Zeile 1 die Farbe des übergebenen Knotens auf grau setzt. Innerhalb einer Ausführung von DFS-VISIT(v) wird die For-Schleife in den Zeilen 4 bis 7 genau $|Adj[v]|$ -mal durchgeführt.

Aus $\sum_{v \in V} |Adj[v]| = O(E)$ folgt damit $O(Tiefensuche) = O(V+E)$.

1.6. Folgerungen und Ergebnisse der Tiefensuche

Bildet man einen Graphen G_π aus den Knoten der Menge *V* des ursprünglichen Graphen *G* und Kanten zwischen allen Knoten *u* und *v* mit $\pi[v] = u$, so bildet dieser Graph einen Wald. Dieser Wald wird **Tiefensuche-Wald** genannt und spiegelt die Reihenfolge der rekursiven Aufrufe von DFS-VISIT wider. In diesem Wald ist ein Knoten *v* genau dann ein Nachfolgerknoten von *u*, wenn *v* entdeckt wurde, während *u* grau eingefärbt war.
Eine weitere wichtige Eigenschaft der Tiefensuche ist die, dass die Zeitmarkierungen für Durchsuchungsbeginn und -ende Klammerstruktur aufweisen. Das heißt:
Schreibt man die Zeitmarkierungen entlang einer Zeitachse auf, indem man z.B. für *d[u]* '(u' und für *f[u]* 'u)' notiert, so erhält man einen korrekt geklammerten Ausdruck. Dieser Sachverhalt wird auch durch das Klammer-Theorem ausgedrückt:

Klammer-Theorem: *(Theorem 1)*

Bei jeder Tiefensuche eines (gerichteten oder ungerichteten) Graphen *G = (V, E)* gilt für zwei Knoten *u* und *v genau* eine der folgenden Aussagen:
- Die Intervalle *[d[u], f[u]]* und *[d[v], f[v]]* sind vollständig disjunkt und weder *u* noch *v* ist ein Nachfolgerknoten des jeweils anderen.
- Das Interval *[d[u], f[u]]* ist vollständig in *[d[v],f[v]]* enthalten und *u* ist ein Nachfolgerknoten von *v*.
- Das Interval *[d[v], f[v]]* ist vollständig in *[d[u],f[u]]* enthalten und *v* ist ein Nachfolgerknoten von *u*.

Beweis:

Sei *d[u] < d[v]*. Für diesen Fall ergeben sich zwei weitere Fälle, *d[v] < f[u]* oder *d[v] > f[u]*. *d[v] = f[u]* kann nach dem Algorithmus nicht eintreten, denn die Zeitvariable wird vor der Speicherung der Zeitmarkierungen iteriert. (sh DFS-VISIT, Zeile 2 und 9)
Sei *d[v] < f[u]*, also wurde *v* entdeckt, während *u* grau war. Das impliziert, dass *v* ein Nachfolgerknoten von *u* ist. Da *v* später als *u* entdeckt wurde, werden weiterhin zuerst alle von *v* ausgehenden Kanten erforscht, bevor der Algorithmus zu *u* zurückkehrt und dessen Durchsuchung beendet. Deshalb ist auch *f[v] < f[u]* und *[d[f], f[f]]* ist vollständig in *[d[u], f[u]]* enthalten.
Sei *d[v] > f[u]*, also auch *f[u] < d[v]*. Daraus folgt automatisch, dass die Intervalle *[d[u], f[u]]* und *[d[v], f[v]]* disjunkt sind.

Der Fall *d[v] < d[u]* lässt sich analog beweisen, indem man in obigem Beweis *u* und *v* vertauscht.

q.e.d.

Korollar zur Verschachtelung der Intervalle: *(Korollar 1)*

Ein Knoten *v* ist genau dann ein Nachfolgerknoten von *u*, wenn gilt:

$$d[u] < d[v] < f[v] < f[u]$$

Der Beweis folgt direkt aus dem Klammer-Theorem.

Das nächste Theorem gibt einen weiteren Anhaltspunkt dafür, wann ein Knoten *v* ein Nachfolgerknoten von *u* ist.

„Weißer-Pfad"-Theorem: *(Theorem 2)*

In einem Tiefensuche-Wald eines (gerichteten oder ungerichteten) Graphen *G = (V,E)* ist ein Knoten *v* genau dann ein Nachfolgerknoten von *u*, wenn *v* zum Zeitpunkt *d[u]*, also zum Zeitpunkt der Entdeckung von *u*, von *u* aus über einen Pfad erreicht werden kann, der vollständig aus weißen Knoten besteht.

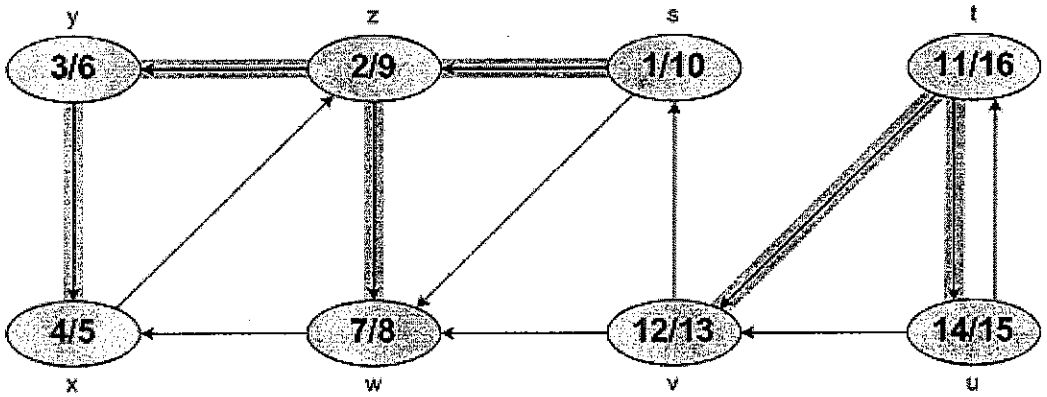
Beweis:

⇒ : Angeommen, *v* ist ein Nachfolgerknoten von *u*. Sei *w* ein Knoten auf dem Pfad von *u* nach *v* im Tiefensuche-Wald und sei *w* ein Nachfolgerknoten von *u*. Nach dem Korollar zur Verschachtelung der Intervalle ist dann *d[u] < d[w]*, also ist *w* zum Zeitpunkt *d[u]* weiß.

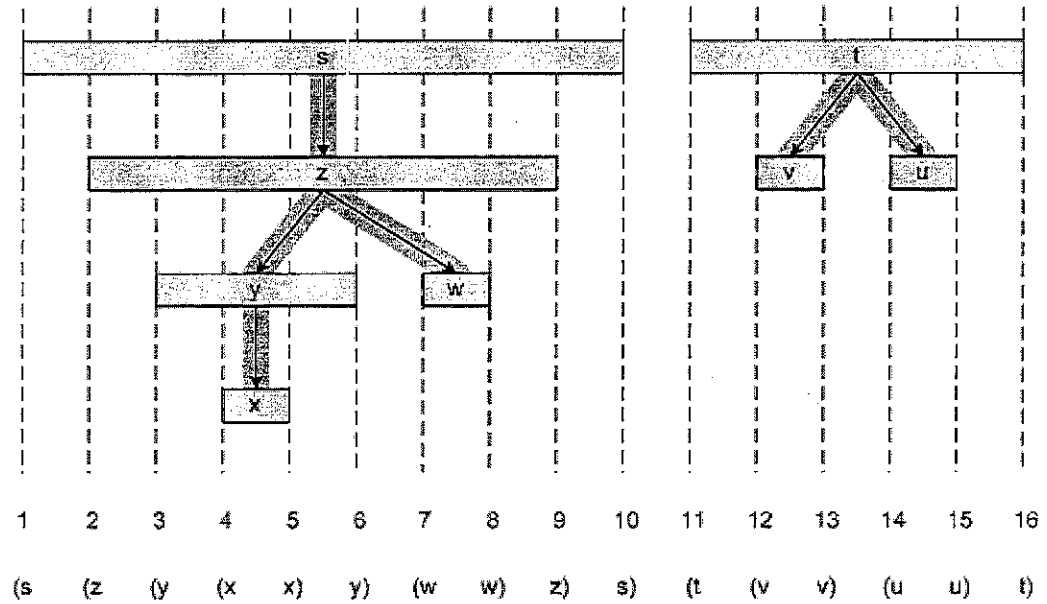
⇐ : Es existiere zum Zeitpunkt *d[u]* ein Pfad von *u* nach *v*, der nur aus weißen Knoten besteht, aber *v* sei kein Nachfolgerknoten von *u* im Tiefensuche-Wald. Sei, ohne Beschränkung der Allgemeinheit, jeder andere Knoten dieses Pfads ein Nachfolgerknoten von *u*. Sei nun *w* der Vorgängerknoten von *v* und Nachfolgerknoten von *u* (*u* und *w* können auch derselbe Knoten sein). Nach dem Korollar zur Verschachtelung der Intervalle gilt *f[w] < f[u]*. Nach dem Klammer-Theorem gilt weiterhin *d[w] < d[v] < f[v] < f[w]* und *d[u] < d[w] < f[w] < f[u]*.
Daraus folgt *d[u] < d[v] < f[v] < f[u]*. Nach dem Klammer-Theorem gilt damit, dass *v* ein Nachfolgerknoten von *u* ist.

q.e.d.

a)



b)



c)

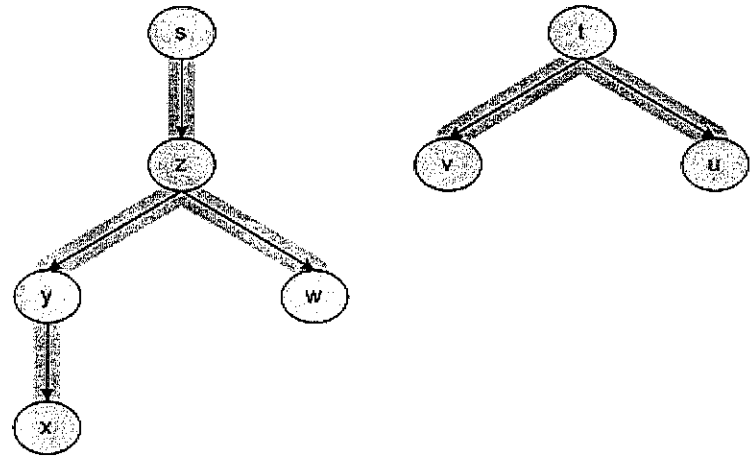


Bild 2: a) Graph G mit Zeitmarkierungen nach Tiefensuche
b) Klammer-Struktur der Zeitmarkierungen
c) resultierender Tiefensuche-Wald

Es bleibt zu sagen, dass die Ergebnisse einer Tiefensuche davon abhängen, welcher Knoten des Graphen *G* als Startknoten ausgewählt wird. Jedoch verursacht das normalerweise keine Probleme, da jede eine Tiefensuche effektiv eingesetzt werden kann und üblicherweise grundlegend äquivalente Ergebnisse ergbit.

1.7. Klassifizierung von Kanten

Mittels der Tiefensuche lassen sich alle Kanten eines Graphen *G* in vier Kategorien einteilen: Baumkanten, Rückkanten, Vorwärtskanten und Kreuzkanten.

- Baumkanten sind diejenigen Kanten, die zur Menge der Kanten des Tiefensuche-Walds gehören, also alle Kanten (u,v) mit $\pi[v] = u$.
- Rückkanten sind alle Kanten, die einen Knoten *u* mit einem seiner Vorgängerknoten *v* verbinden. Auch Kanten (u,u) werden als Rückkanten gewertet.
- Vorwärtskanten sind alle Kanten, die einen Knoten *u* mit einem seiner Nachfolgerknoten *v* verbinden, aber nicht zum Tiefensuche-Wald gehören.
- Kreuzkanten sind alle anderen Kanten. Kreuzkanten kann es zwischen zwei verschiedenen Bäumen des Tiefensuche-Walds geben oder zwischen zwei Knoten desselben Baums, die weder Nachfolger- noch Vorgängerknoten des jeweils anderen sind.

Die Einteilung der Kanten kann erfolgen, ohne dass der Tiefensuche Algorithmus verändert werden muss:
Trifft man von einem Knoten *u* aus auf einen Knoten *v*, so reicht bereits dessen Farbe, um die Kante als Baumkante, Rückkante oder als Vorwärtskante/Kreuzkante zu klassifizieren. Ist *v* zum Zeitpunkt seiner Entdeckung weiß, handelt es sich bei (u,v) um eine Baumkante. Ist *v* grau, muss es sich um eine Rückkante handeln, da *u* immer der graue Knoten mit der aktuell größten Tiefe ist, d.h. wenn *v* grau ist, ist seine Tiefe geringer als die Tiefe von u mit $d[v] < d[u] < f[v]$, woraus nach dem Klammer-Theorem folgt, dass *v* ein Vorgängerknoten von *u* sein muss.
Die Unterscheidung zwischen Vorwärtskante und Kreuzkante kann nur mittels Überprüfung von $d[u]$ und $d[v]$ erfolgen. Ist $d[u] < d[v]$, so handelt es sich bei der Kante (u,v) um eine Vorwärtskante, andernfalls um eine Kreuzkante.
Anzumerken bleibt noch, dass in der Tiefensuche eines ungerichteten Graphen G nur Baum- und Rückkanten auftreten.

1.8. Beispiel zur Kantenklassifizierung

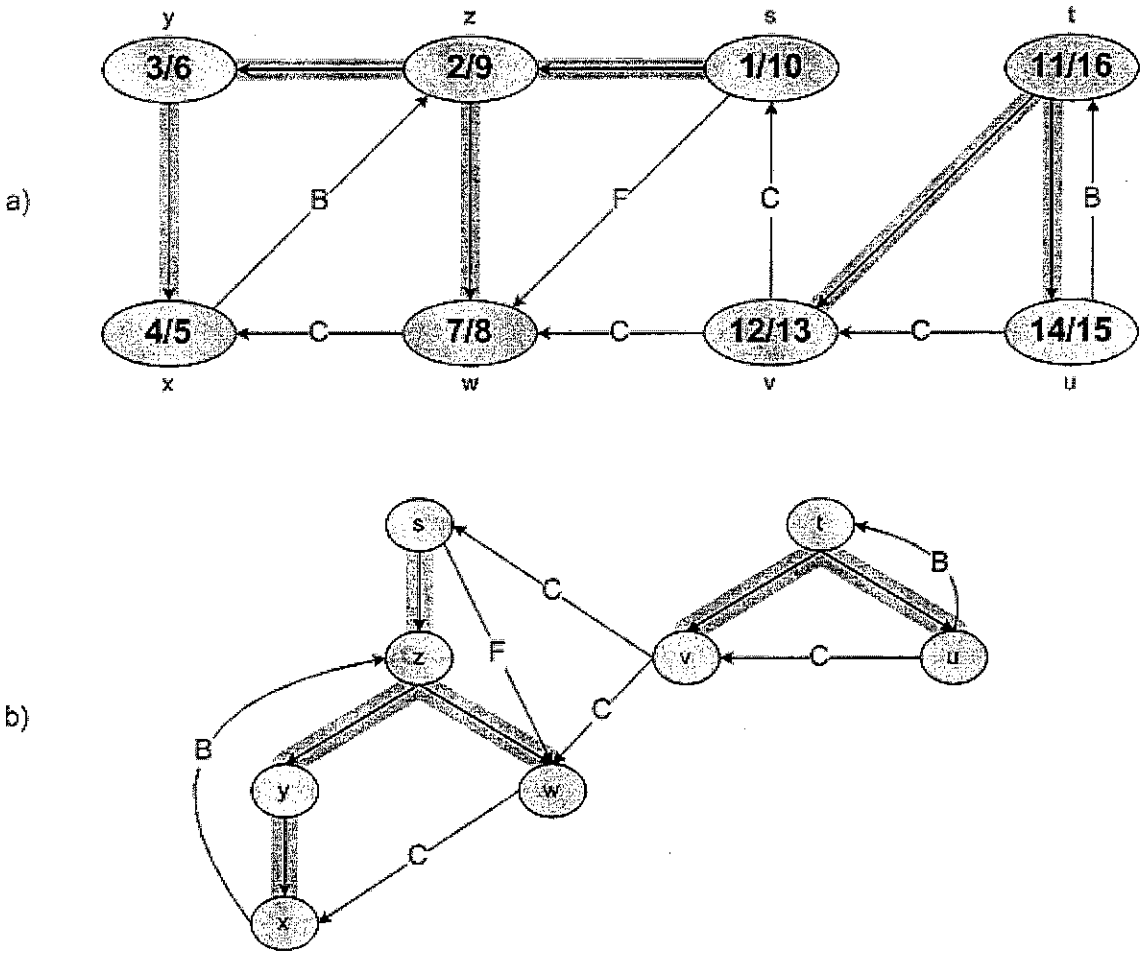


Bild 2: a) Graph G mit Zeitmarkierungen nach Tiefensuche und Kantenklassifizierungen. B bezeichnet Rückkanten, C Kreuzkanten und F Vorwärtskanten. Die grau schattierten Kanten sind die Baumkanten.
b) resultierender Tiefensuche-Wald mit eingezeichneten „Nicht-Baum-Kanten"

1.9. Folgerungen der Kantenklassifizierung

Theorem 3:
In der Tiefensuche eines ungerichteten Graphen *G* ist jede Kante entweder eine Baumkante oder eine Rückkante.

Beweis:
Sei *(u,v)* eine Kante des Graphen *G* und sei (ohne Beschränkung der Allgemeinheit) *d[u] < d[v]*. Da *v* in *u*´s Adjazenzliste enthalten ist, muss *v* entdeckt und seine Durchsuchung abgeschlossen sein, bevor die Durchsuchung von *u* abgeschlossen ist, also während *u* grau ist. Wird die Kante *(u,v)* zuerst in der Richtung von *u* nach *v* erforscht wird, so ist *v* zu diesem Zeitpunkt weiß, sonst wäre *(u,v)* schon von *v* aus erforscht worden. Dann ist *(u,v)* eine Baumkante. Wird *(u,v)* zuerst von *v* aus erforscht, so handelt es sich bei *(u,v)* um eine Rückkante, weil *u* zum Zeitpunkt der ersten Erforschung der Kante *(u,v)* noch grau ist.

Lemma 1:
Ein gerichteter Graph *G* ist genau dann azyklisch, wenn eine Tiefensuche auf *G* keine Rückkanten ergibt.

Beweis:
$\Rightarrow$: Sei *(u,v)* eine Rückkante zwischen *u* und *v*. Dann ist der Knoten *v* ein Vorgängerknoten von *u*. Daraus folgt, dass eine Kante *(v,u)* von *v* nach *u* in *G* existiert und die Kanten *(u,v)* und *(v,u)* ergeben einen Zyklus.

$\Leftarrow$: *G* enthalte einen Zyklus *c*. Zu zeigen ist, dass eine Tiefensuche auf *G* eine Rückkante ergibt. Sei *v* der erste Knoten im Zyklus *c*, der entdeckt wird und sei *(u,v)* eine Kante im Zyklus *c*. Zum Zeitpunkt *d[v]* stellen die Knoten des Zyklus *c* einen Pfad aus weißen Knoten von *v* nach *u* dar.
Mit dem „Weißer-Pfad"-Theorem folgt hieraus, dass *u* ein Nachfolgerknoten von *v* ist. Deshalb ist *(u,v)* eine Rückkante.

2. Topologische Sortierung

2.1. Topologische Sortierung

Unter einer topologischen Sortierung eines Graphen *G* versteht man eine lineare Anordnung der Knoten dieses Graphen, so dass der Knoten *u* vor dem Knoten *v* in der Anordnung erscheint, wenn es eine Kante von *u* nach *v* im Graphen gibt. Einfacher ausgedrückt: Man schreibt alle Knoten des Graphen *G* so auf eine Linie, dass alle Kanten des Graphen von links nach rechts zeigen. Aus dieser Definition ergibt sich, dass eine solche Anordnung nur sinnvoll möglich ist, wenn *G* gerichtet ist und keine Zyklen enthält, also azyklisch ist. G muss also ein gerichteter azyklischer Graph sein, damit eine topologische Sortierung sinnvoll und überhaupt möglich ist.
Gerichtete azyklische Graphen können verwendet werden, um zeitlich abhängige Ereignisse darzustellen, z.B. müssen bei der Herstellung eines bestimmten Produkts die Edukte bereits vorliegen, die aber wiederum erst produziert werden müssen. Hat man nun einen Graphen, der einen kompletten Produktionsvorgang darstellt, kann man eine topologische Sortierung dieses Graphen vornehmen, um eine Liste zu erstellen, die eine Reihenfolge enthält, in der man zum gewünschten Ziel kommt.

2.2. Algorithmus

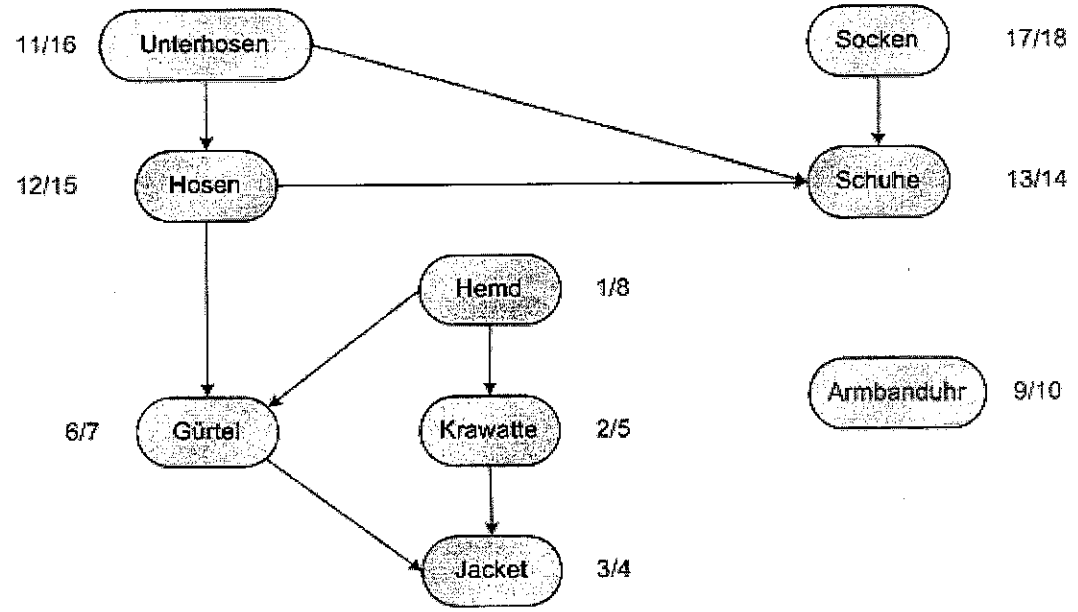
```
TOPOLOGICAL-SORT(G)
1    DFS(G) aufrufen
2    Knoten v vorne in eine Liste L einfügen, wenn dessen
     Durchsuchung beendet ist
3    L zurückgeben
```

2.3. Erklärung des Algorithmus

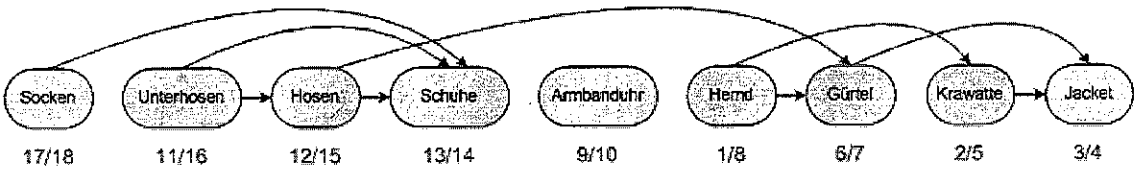
Führt man auf dem Graphen G eine Tiefensuche (siehe 1.2. und 1.3.) durch, fügt jeden Knoten *v* vorne in eine Liste *T* ein, wenn *f[v]* erreicht ist (Kap.1.2., DFS-VISIT, Zeile 10), dann enthält die *T* eine topologische Sortierung von *G*. Die topologische Sortierung *T* enthält also alle Knoten des Graphen in umgekehrter Reihenfolge ihrer Zeitmarkierung *f*.

2.4. Beispiel anhand eines Graphen

Der folgende Graph stellt einen einfachen Vorgang dar: das morgentliche Anlegen der Garderobe. Das bietet sich als Beispiel an, da es einige Kleidungsstücke gibt, die erst angezogen werden sollten, wenn bereits die darunter liegenden angezogen wurden, z.B. sollte eine Hose erst nach der Unterhose angezogen werden. Es gibt jedoch auch Kleidungsstücke, die von anderen unabhängig sind, wie z.B. die Armbanduhr.



Aus diesem Graphen lässt sich folgende topologische Sortierung ableiten:



2.5. Terminierung und Aufwandsberechnung

Da das Einfügen des Knotens in eine Liste die Komplexität $O(1)$ hat und ansonsten nur eine Tiefensuche durchgeführt wird, hat der vorgestellte Algorithmus die selbe Komplexität wie die Tiefensuche, also $O(V+E)$ für einen Graphen $G = (V, E)$. Da die Tiefensuche wie gezeigt terminiert, terminiert auch die topologische Sortierung.

2.6. Beweis

Zu zeigen ist, dass TOPOLOGICAL-SORT(G) eine topologische Sortierung eines gerichteten, azyklischen Graphen *G* erstellt.

Beweis:

Es werde DFS(G) auf einem gerichteten, azyklischen Graphen *G* ausgeführt, um *f[u]* für alle Knoten *u* in *G* zu berechnen. Dann reicht es aus, zu zeigen, dass für jedes Paar Knoten *u,v* in *G* gilt: Wenn es eine Kante von *u* nach *v* gibt, dann ist *f[v] < f[u].* Angenommen, DFS(G) erforscht eine beliebige Kante *(u,v)* in *G*. Zu diesem Zeitpunkt kann *v* nicht grau sein, denn sonst wäre *v* ein Vorgängerknoten von *u*, was *Lemma 1* widersprechen würde (*G* ist azyklisch). Deshalb muss *v* entweder weiß oder schwarz sein. Wäre *v* weiß, so würde *v* ein Nachfolgerknoten von *u* werden und es würde *f[v] < f[u]* gelten. Wäre *v* schwarz, wäre die Durchsuchung von *v* bereits abgeschlossen und *f[v]* wäre bereits gespeichert. Da die Zeitvariable nie verringert wird, deshalb stetig wächst und *f[u]* noch nicht gespeichert wurde, wird *f[v] < f[u]* gelten, sobald *f[u]* gespeichert wird.
Deshalb gilt für jede Kante *(u,v) f[v] < f[u]* und TOPOLOGICAL-SORT(G) erstellt eine topologische Sortierung von G.

q.e.d.