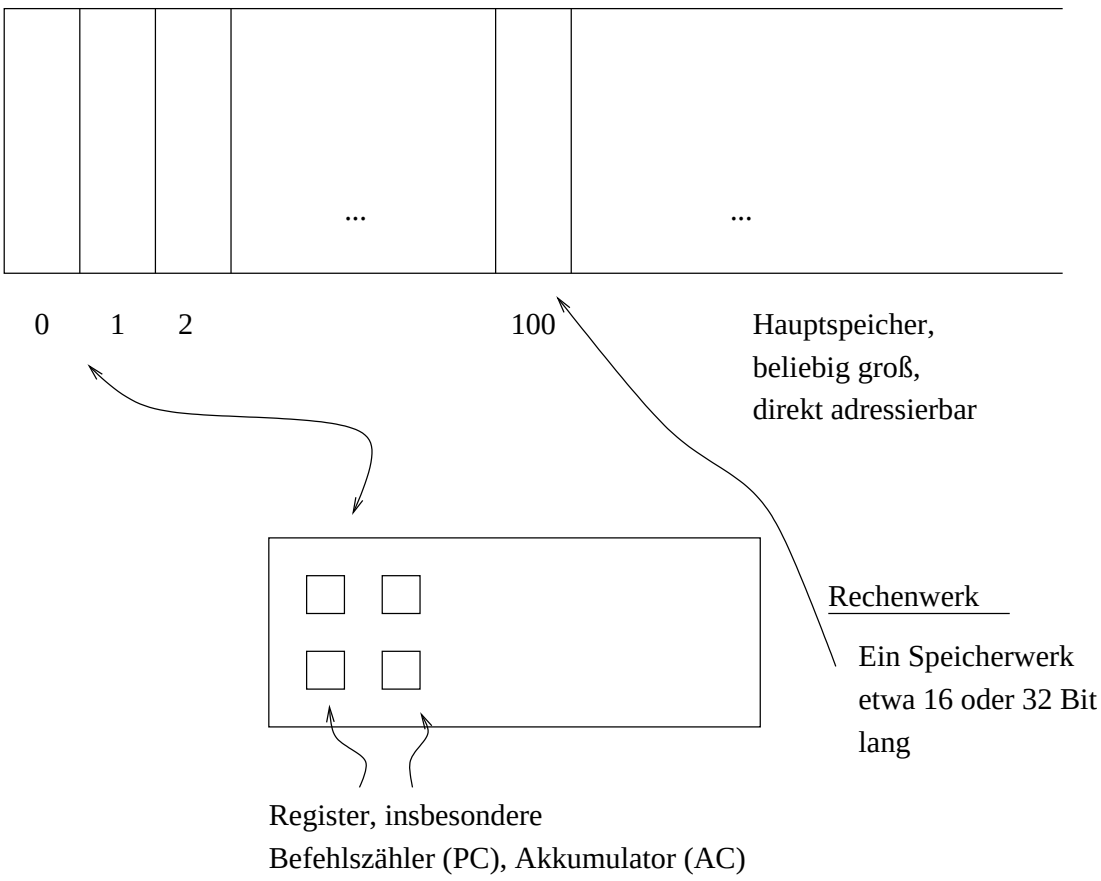
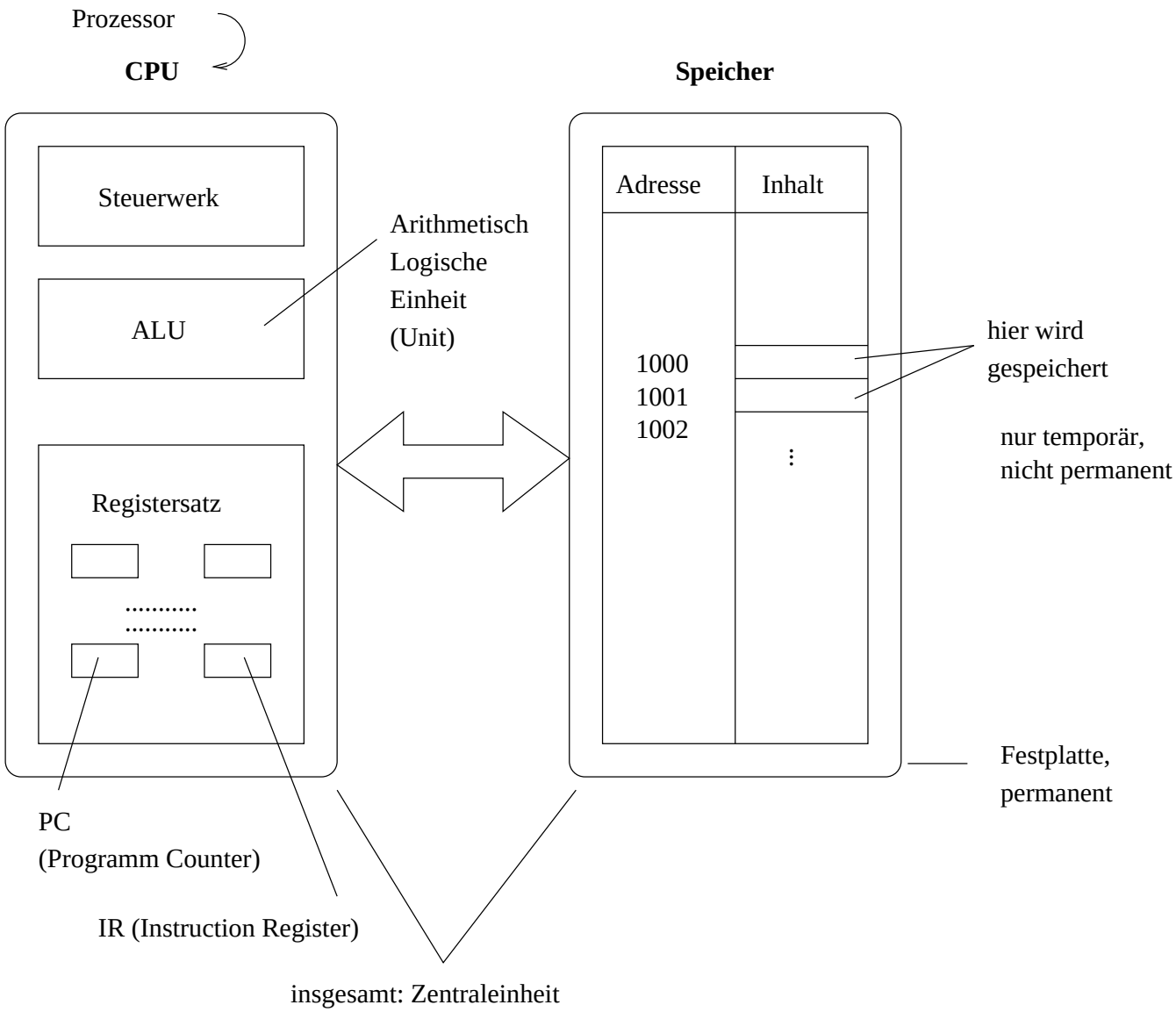
3 Zeit und Platz

Unsere Programme laufen letztlich immer auf einem Von-Neumann-Rechner (wie nachfolgend beschrieben) ab. Man spricht auch von einer random access machine (RAM), die dann vereinfacht so aussieht:





Typische Befehle der Art

Load 5000	Inhalt von Speicherplatz 5000 in Akkumulator
Add 1005	Inhalt von 1005 hinzuaddieren
Store 2000	Inhakt des Akkumulators in Platz 2000 speichern

Programme im Hauptspeicher.
Simulation von Schleifen durch Sprungbefehle, etwa:

Jp E	unbedingter Sprung nach E
Jpz E	Ist ein bestimmtes Register = 0, dann Sprung nach E (Jump if Zero)

Zur Realisierung von Pointern müssen Speicherinhalte als Adressen interpretiert werden. Deshalb auch folgende Befehle:

Load ↑x Inhalt des Platzes, dessen Adresse in Platz x steht, wird geladen
Store ↑x, Add ↑x, ...

Weitere Konventionen sind etwa:

- Eingabe in einem speziell reservierten Bereich des Hauptspeichers.
- Ebenso Ausgabe.

Listing 1: Java-Programm zum Primzahltest

```
1  // Hier einmal ein Primzahltest.
2  // Durch "return" wird das Programm beendet -- "Sprung ans Ende."
3
4  import Prog1Tools.IOTools;
5  public class PRIM{      // Name muss dem Dateinamen gleich sein!
6  public static void main(String[]args){
7    long c, d;
8    c = IOTools.readLong("Einlesen␣eins␣langen␣c␣>=␣zum␣Test:");
9    if (c == 2){
10     System.out.print(c +"ist␣PRIM");
11     return;
12   }
13   d = 2;
14   while(d * d <=c){      // Warum reicht es, bei D*D > c
15                          // aufzuhören?
16     if (c % d == 0){
17       System.out.println(c + "ist␣nicht␣PRIM,␣denn" + d + "teilt"
             + c);
18       return;            // Hier ist das Programm zuende,
19                          // Erläuterung siehe oben
20     }
21     d++;                 // Die Schleife hört auf, nach dem Lauf,
22                          // wo d so gesetzt wird, dass d*d > c
23                          // ist! An diesem Punkt muss immer
24                          // aufgepasst werden!
25   }
26   System.out.println("Die␣Schleife␣ist␣fertig␣ohne␣ordentlichen␣
        Teiler,␣ist" + c + "PRIM");
27 }
28 }
```

Das obige Programm wird etwa folgendermaßen für die RAM übersetzt. Zunächst die while-Schleife:

```
Load d          //Speicherplatz von d in Register (Akkumulator)
Mult d          //Multiplikation von Speicherplatz d mit Akkumulator
Store e         //Ergebnis in e
Analog in f e-c berechnen
Load f          // Haben d·d − c im Register
Jgz E           // Sprung ans Ende wenn d·d − c > 0, d.h. d·d > c
```

Das war nur der Kopf der while-Schleife.

Jetzt der Rumpf: if (c%d){...}

```
c % d in Speicherplatz m ausrechnen.
Load m
Jgz F                 // Sprung wenn > 0
Übersetzung von System.out.println(...)
JpE                   // Unbedingt ans Ende.
```

d++

```
F:  Load d
    Inc               // Register erhöhen
    JpA               // Zum Anfang
E:  Stop
```

Als Flussdiagramm AuP 9.14 folgende.

Wichtige Beobachtung: Jede einzelne Java-Zeile führt zur Abarbeitung einer konstanten Anzahl von Maschinenbefehlen (konstant ⇔ unabhängig von der Eingabe c, wohl abhängig von der eigentlichen Programmzeile.)

Für das Programm PRIM gilt: # ausgefüllte Java-Zeilen bei Eingabe $c \leq a\sqrt{c} + b$ mit $a \cdot \sqrt{c}$ ... Schleife (Kopf und Rumpf) und b ... Anfang und Ende
a, b konstant, d.h. unabhängig von c!

Also auch # ausgeführte Maschinenbefehle bei Eingabe $c \leq a' \cdot \sqrt{c} + b'$
Ausführung eines Maschinenbefehls:
Im Nanosekundenbereich
Nanosekunden bei Eingabe $c \leq a'' \cdot \sqrt{c} + b''$.
In jedem Fall gibt es eine Konstante k, so dass der „Verbrauch“ $\leq k\sqrt{c}$ ist ($k = a + b, a' + b', a'' + b''$).

Fazit: Laufzeit unterscheidet sich nur um einen konstanten Faktor von der Anzahl ausgeführter Programmzeilen.Schnellere Rechner ⇒ Maschinenbefehle schneller ⇒ Laufzeit nur um einen konstanten Faktor schneller. Bestimmen Laufzeit nur bis auf einen konstanten Faktor.

Definition 3.1(O-Notation): *Ist $f, g : \mathbb{N} \to \mathbb{R}$. Dann ist $f(n) = O(g(n))$ genau dann, wenn es eine Konstante $C > 0$ mit $|f(n)| \leq C \cdot g(n)$ gibt, für alle hinreichend großen n.*

Das Programm PRIM hat Zeit $O(\sqrt{c})$ bei Eingabe c. Platzbedarf (= # belegte Plätze) etwa 3, also $O(1)$.

Eine Java-Zeile $\Leftrightarrow$ endliche Anzahl Maschinenbefehle trifft nur bedingt zu:
Solange Operanden nicht zu groß (d.h. etwa in einem oder einer endlichen Zahl von Speicherplätzen). D.h., wir verwenden das *uniforme Kostenmaß*. Größe der Operanden ist uniform 1 oder $O(1)$.

Zur Laufzeit unseres Algorithmus $BFS(G, s)$. (Seite 14) Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$.
Datenstrukturen initialisieren (Speicherplatz reservieren usw.): $O(n)$.

1. Array col setzen: $O(n)$
2. O(1)
3. Kopf der while-Schleife
   Einmal $O(1)$
4. Einmal $O(1)$
5. Kopf einmal $O(1)$
   (Gehen $Adj[u]$ durch)
   Rumpf einmal $O(1)$
   In einem Lauf der while-Schleife
   wird 5. bis zu $n-1$-mal durchlaufen.
   Also 5. in $O(n)$.
   Rest $O(1)$

Rest $O(1)$.

Also: while-Schleife einmal:

$$O(1) + O(n) \text{ also } O(1) + O(1) + O(1) \text{ und } O(n) \text{ für 5.}$$

Durchläufe von 3. $\leq n$, denn schwarz bleibt schwarz. Also Zeit $O(1) + O(n^2) = O(n^2)$.

Aber das ist nicht gut genug. Bei $E = \emptyset$, also keine Kante, haben wir nur eine Zeit von $O(n)$, da 5. jedesmal nur $O(1)$ braucht.

Wie oft wird 5. insgesamt (über das ganze Programm hinweg) betreten? Für jede Kante genau einmal. Also das Programm hat in 5. die Zeit $O(m + n)$, n für die Betrachtung von $Adj[u]$ an sich, auch wenn $m = 0$.

Der Rest des Programmes hat Zeit von $O(n)$ und wir bekommen $O(m+n) \leq O(n^2)$. Beachte $O(m+n)$ ist bestmöglich, da allein das Lesen des ganzen Graphen $O(m+n)$ erfordert.
Regel: Die Multiplikationsregel für geschachtelte Schleifen:

Schleife 1 $\leq n$ Läufe
Schleife 2 $\leq m$ Läufe,

also Gesamtzahl Läufe von Schleife 1 und Schleife 2 ist $m \cdot n$ gilt nur bedingt. Besser ist es (oft), Schleife 2 insgesamt zu zählen. Globales Zählen.

Betrachten nun das Topologische Sortieren von Seite 22
$G = (V, E), |V| = n, |E| = m$
Initialisierung: $O(n)$

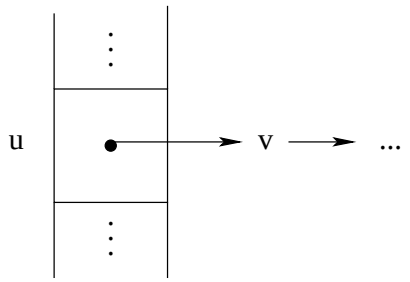
1. Array Egrad bestimmen: $O(m)$
   Knoten mit Grad 0 suchen: $O(n)$
   Knoten in top. Sortierung tun und Adjazenzlisten anpassen: $O(1)$
2. Wieder genauso: $O(m) + O(n) + O(1)$
   $\vdots$

Nach Seite 23 bekommen wir aber Linearzeit heraus:
1. und 2. Egrad ermitteln und Q setzen: $O(m)$
3. Ein Lauf durch 3. $O(1)$! (keine Schleifen), insgesamt n Läufe. Also: $O(n)$
4. insgesamt $O(n)$
5. jede Kante einmal, also insgesamt $O(m)$

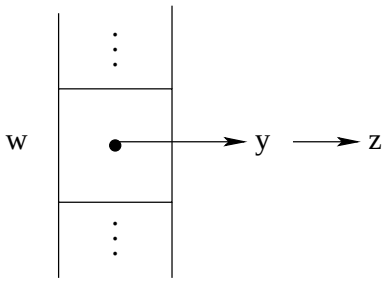
Zeit $O(m + n)$, Zeitersparnis durch geschicktes Merken.

4 Tiefensuche in gerichteten Graphen

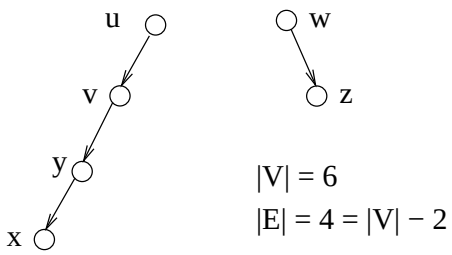
Fangen bei n zum Zeitpunkt 1 an, Gehen eine Kante, entdecken v zum Zeitpunkt 2, (b). Adjazenzliste



v ist der erste Knoten. Dann wird y in (c) entdeckt. u, v, y sind offen = grau. In (d) entdecken wir das x. Kante (x, v) wird zwar gegangen, aber v nicht darüber entdeckt.
Dann bei w weiter. Adjazenzliste ist:



Tiefensuchwald = Kanten, über die entdeckt wurde.



Wald = Menge von Bäumen
$\Pi[x] = y, \Pi[y] = v, \Pi[v] = u, \Pi[u] = u$ (alternativ nil), $\Pi[w] = w$ (alternativ nil)

4.1 Algorithmus Tiefensuche

Eingabe $G = (V, E)$ in Adjazenzlistendarstellung, gerichteter Graph, $n = |V|$.

d[1 ... n]	...	Entdeckzeit(discovery-Zeit) Nicht(!) Entfernung wie vorher. Knoten wird grau.
f[1 ... n]	...	Beendezeit (finishing), Knoten wird schwarz.
Π[1 ... n]	...	Tiefensuchwald, wobei $\Pi[u] = v \Longleftrightarrow$  $\Pi[u]$ = Knoten, über den u entdeckt wurde
col[1 ... n]	...	aktuelle Farbe

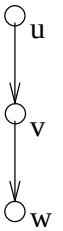
Algorithmus 1 : DFS(G)

```
   /* 1. Initialisierung */
 1 foreach u ∈ V do
 2     col[u] = weiß;
 3     Π[u] = nil;
 4 end
 5 time = 0;
   /* 2. Hauptschleife. Aufruf von DFS-visit nur, wenn col[u] =
      weiß */
 6 foreach u ∈ V do
 7     if col[u] == weiß then
 8         DFS-visit(u);
 9     end
10 end
```

Prozedur `DFS-visit(`u`)`

```
 1 col[u] = grau;                          /* Damit ist u entdeckt */
 2 d[u] = time;
 3 time = time + 1;
 4 foreach v ∈ Adj[u] do                    /* u wird bearbeitet */
 5     if col[v] == weiß then               /* (u,v) untersucht */
 6         Π[v] = u;                              /* v entdeckt */
 7         DFS-visit(v);
 8     end
 9 end
   /* Die Bearbeitung von u ist hier zuende. Sind alle Knoten aus
      Adj[u] grau oder schwarz, so wird u direkt schwarz. */
10 col[u] = schwarz;
11 f[u] = time;
12 time = time + 1;          /* Zeitzähler geht hoch bei Entdecken und
   Beenden */
```

Es ist $\{d[1], \ldots, d[m], f[1], \ldots, f[m]\} = \{1, \ldots, 2n\}$. Man kann über die Reihenfolge nicht viel sagen. Möglich ist:



Einmal:
$d[u] = 1, d[v] = 2, d[w] = 3$
und
$f[w] = 4, f[v] = 5, f[u] = 6.$
Aber auch:
$d[w] = 1, f[w] = 2, d[v] = 3, f[v] = 4,$
$d[u] = 5, f[u] = 6$

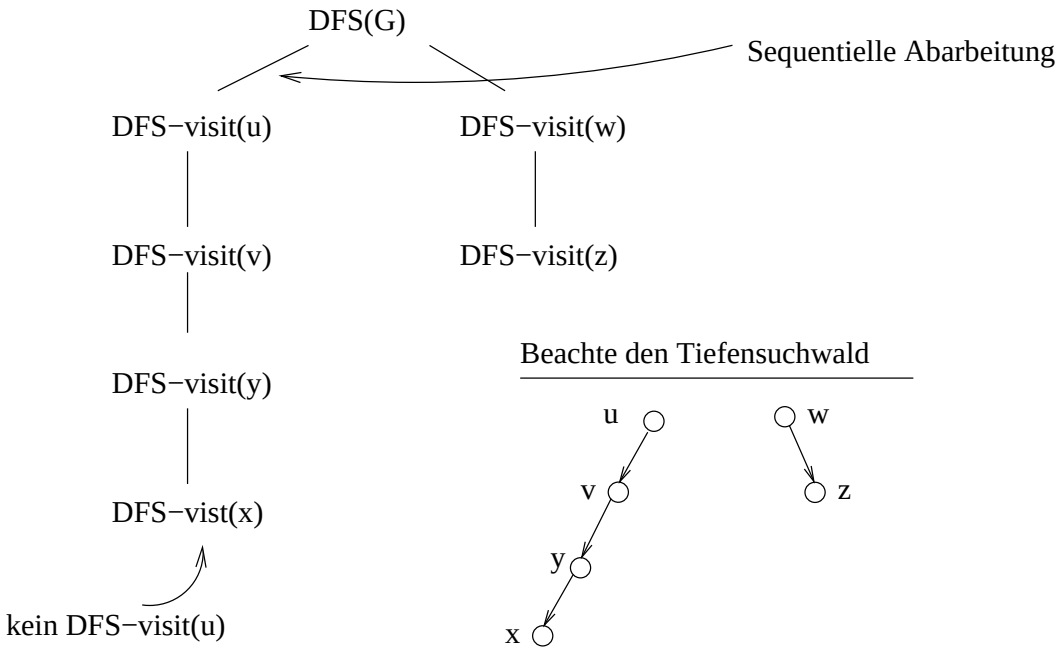
DFS-visit(u) wird nur dann aufgerufen, wenn col[u] = weiß ist. Die Rekursion geht nur in weiße Knoten.

Nachdem alle von u aus über weiße(!) Knoten erreichbare Knoten besucht sind, wird col[u] = schwarz.

Im Nachhinein war während eines Laufes

- col[u] = weiß, solange time < d[u]
- col[u] = grau, solange d[u] < time < f[u]
- col[u] = schwarz, solange f[u] < time

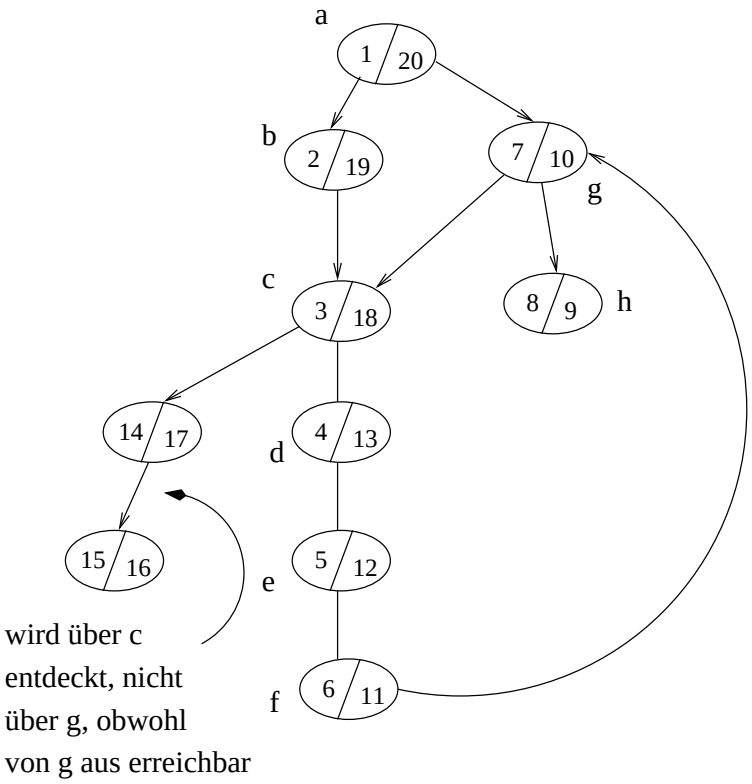
Unser Eingangsbeispiel führt zu folgendem Prozeduraufrufbaum:



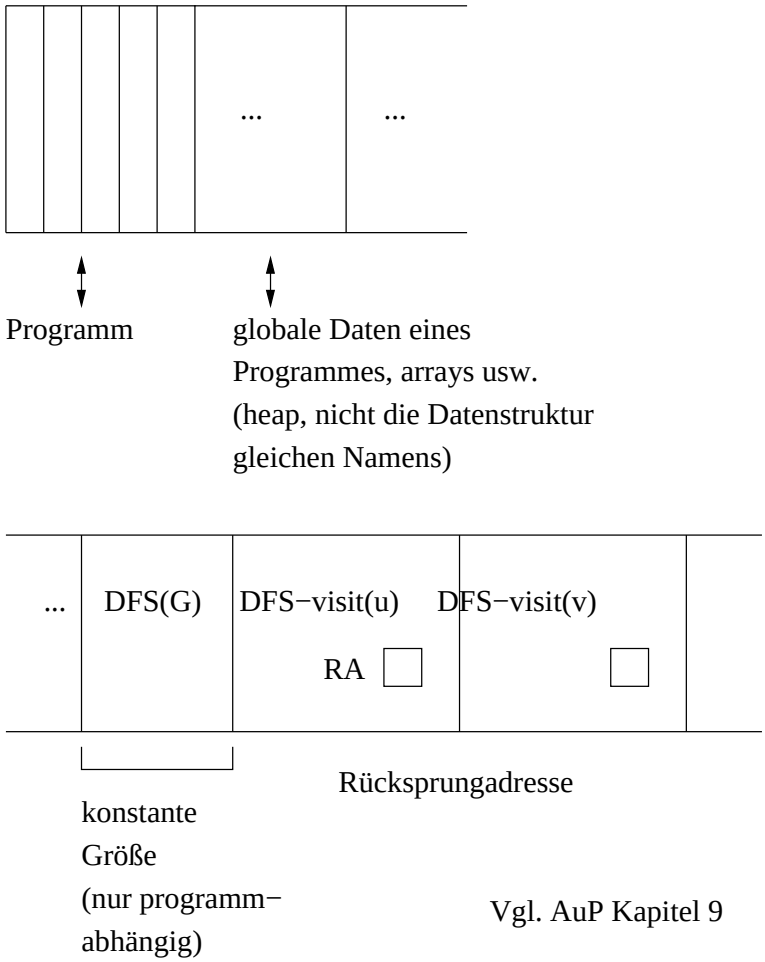
Erzeugung: Präorder(Vater vor Sohn)
Beendigung: Postorder(Sohn vor Vater)

Betrachten wir nun noch einmal folgendes Beispiel:

Beispiel 4.1:



Wie erfolgt die Ausführung der Prozeduraufrufe auf unserem Maschinenmodell der RAM? Organisation des Hauptspeichers als Keller von frames:



Verwaltungsaufwand zum Einrichten und Löschen eines frames: $O(1)$, programmabhängig. Hier werden im wesentlichen Adressen umgesetzt.

Merkregel zur Zeitermittlung:
Durchlauf jeder Programmzeile inklusive Prozeduraufruf ist $O(1)$.
Bei Prozeduraufruf zählen wir so:
Zeit für den Aufruf selbst (Verwaltungsaufwand) $O(1)$ + Zeit bei der eigentlichen Ausführung.

Satz 4.1: *Für $G = (V, E)$ mit $n = |V|$ und $m = |E|$ braucht DFS(G) eine Zeit $O(n + m)$.*

Beweis.
DFS(G) 1. $O(n)$, 2. $O(n)$ ohne Zeit in DFS-visit(u).
DFS-visit(u) $O(n)$ für Verwaktungsaufwand insgesamt.

DFS-visit(u) wird nur dann aufgerufen, wenn col[u] = weiß ist. Am Ende des Aufrufs wird col[u] = schwarz.

1. und 2. einmal $O(1)$ insgesamt $O(n)$.
3., 4. 5. 6. ohne Zeit in DFS-visit(v) insgesamt $O(m)$.

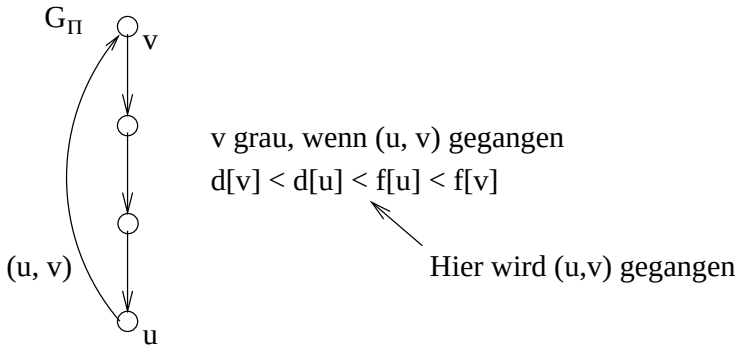
(3., 4. für jede Kante einmal, also $O(m)$. 5. und 6. für jedes Entdecken, also $O(n)$).
7. und 8. $O(n)$ insgesamt.
Also tatsächlich $O(n + m)$ □

Definition 4.1(Tiefensuchwald): *Sei $G = (V, E)$ und sei DFS(G) gelaufen. Sei $G_\Pi = (V, E_\Pi)$ mit $(u, v) \in E_\Pi \iff \Pi[v] = u$ der Tiefensuchwald der Suche.*

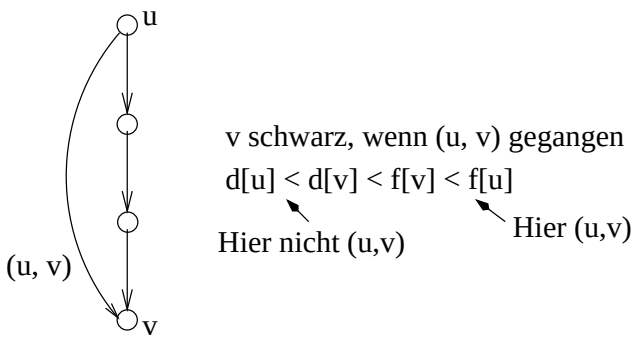
Klassifizieren der Kanten von G.
Sei $(u, v) \in E$.

(a) (u, v) Baumkante $\iff (u, v) \in E_\Pi (\Pi[v] = u)$

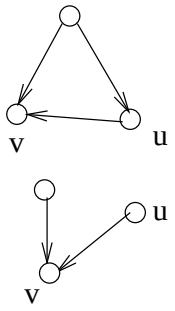
(b) (u, v) Rückwärtskante $\iff (u, v) \notin E_\Pi$ und v Vorgänger von u in G_Π



(c) (u, v) Vorwärtskante $\iff (u, v) \notin E_\Pi$ und v Nachfolger von u in G_Π



(d) (u, v) Kreuzkante $\iff (u, v) \notin E_\Pi$ und u weder Nachfolger noch Vorgänger von v in G_Π



v schwarz, wenn (u, v) gegangen
d[v] < f[v] < d[u] < f[u]

□

Noch eine Beobachtung für Knoten u, v. Für die Intervalle, in denen die Knoten aktiv (offen, grau) sind. gilt:
Entweder
$d[u] < d[v] < f[v] < f[u]$

DFS-visit(u)

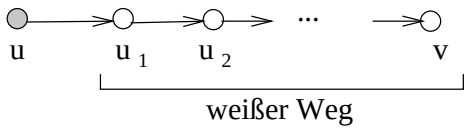
⋮

DFS-visit(v)

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

oder umgekehrt. Folgt aus Kellerstruktur des Laufzeitkellers.

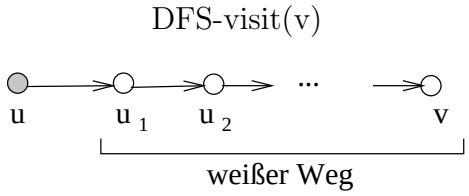
Satz 4.2(Weißer-Weg-Satz): *v wird über u entdeckt (d.h. innerhalb von DFS-visit(u) wird DFS-vist(v) aufgerufen)*
$\Longleftrightarrow$
Zum Zeitpunkt d[u] gibt es einen Weg



in G.

Beweis. „⇒“ Aufrufstruktur, wenn DFS-visit(v) aufgerufen wird:

DFS-visit(u)
|
DFS-visit(u_1)
|
DFS-visit(u_2)
|



„⇐“ Liege also zu $d[u]$ der weiße Weg u $\to v_1 \to v_2 \to \cdots \to v_k \to$ v vor. Das Problem ist, dass dieswe Weg keineswegs von der Suche genommen werden muss. Trotzdem, angenommen v wird nicht über u entdeckt, dann gilt nicht

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

sondern

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

Dann aber auch nach Programm

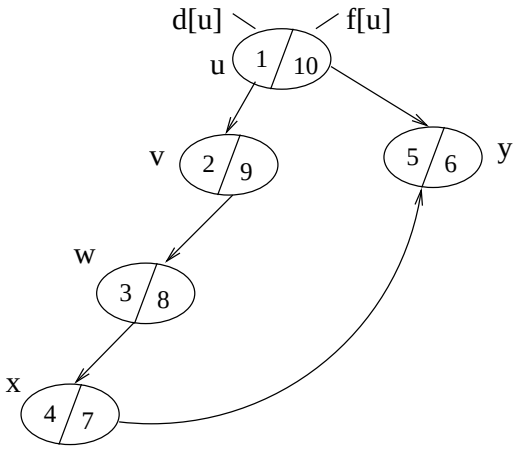
$$d[u] < f[u] < d[v_k] < f[v_k]$$
$$\ldots$$

dann

$$d[u] < f[u] < d[v_1] < f[v_1]$$

was dem Programm widerspricht. □

Noch ein Beispiel:
Beispiel 4.2:

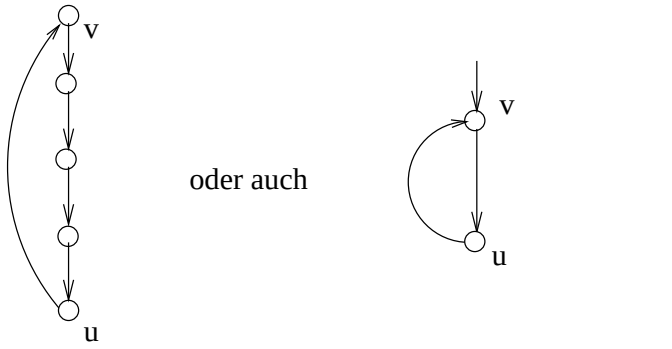


Zum Zeitpunkt 1 ist (u, y) ein weißer Weg. Dieser wird nicht gegangen, sondern ein anderer. Aber y wird in jedem Fall über u entdeckt!

Kreis feststellen!

Satz 4.3: *Ist $G = (V, E)$ gerichtet, G hat Kreis $\Longleftrightarrow$ DFS(G) ergibt eine Rückwärtskante.*

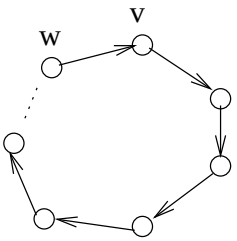
Beweis. „$\Leftarrow$“ Sei (u, v) eine Rückwärtskante. Dann in G_Π, dem Tiefensuchwald



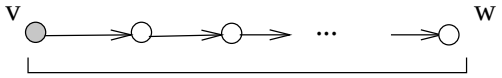
Rückwärtskante (u, v) Rückwärtskante (u, v)

Also Kreis in G.

„$\Rightarrow$“ Hat G einen Kreis, dann also



Sei v der erste Knoten auf dem Kreis, den DFS(G) entdeckt. Dann zu dem Zeitpunkt weißer Weg

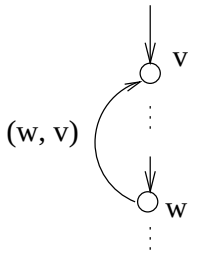


Der Kreis herum.

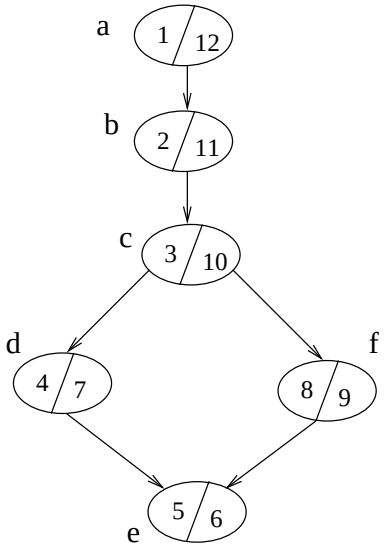
□

Satz 4.4(Weißer-Weg-Satz): $d[v] < d[w] < f[w] < f[v]$

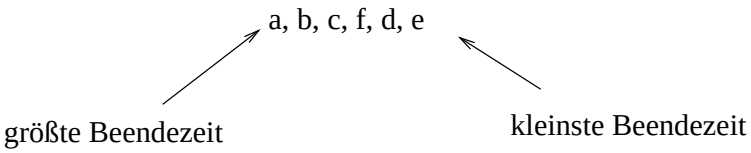
Also wenn (w, v) gegangen und ist $col[v] = grau$ also Rückwärtskante. Alternativ, mit Weißer-Weg-Satz:



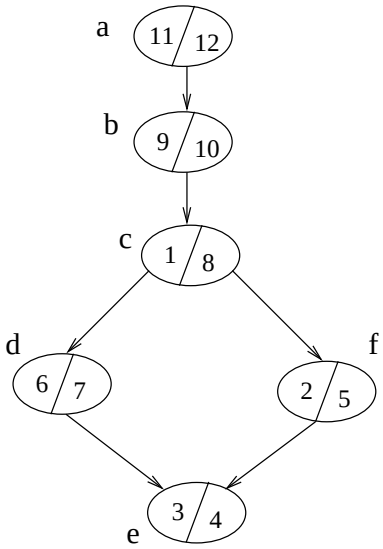
w ist irgendwann unter v und dann ist (w, v) Rückwärtskante. Man vergleiche hier den Satz und den Beweis zu Kreisen in ungerichteten Graphen in Kapitel 2.
Dort betrachte man den zuletzt(!) auf dem Kreis entdeckten Knoten.
Wir können auch topologisch sortieren, wenn kreisfrei.



Nach absteigender Beendezeit sorteieren.



Aber auch



Absteigende Beendezeit: a, b, c, d, f, e.

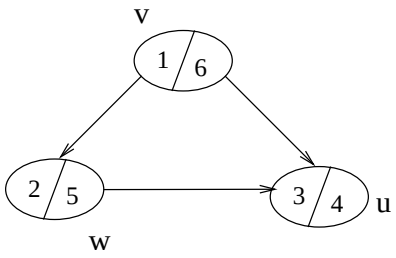
Satz 4.5: *Sei $G = (V, E)$ kreisfrei. Lassen nun DFS(G) laufen, dann gilt für alle $u, v \in V, u \notin v$, dass*
$f[u] < f[v] \Rightarrow (u, v) \notin E$ Keine Kante geht von kleinerer nach größerer Beendezeit.

Beweis. Wir zeigen: $(u, v) \in E \Rightarrow f[u] > f[v]$.

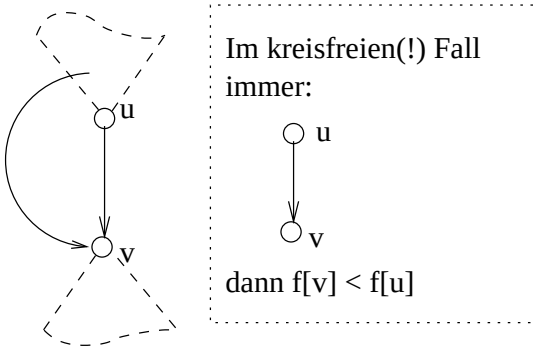
1. Fall:$d[u] < d[v]$
   Dann Weißer Weg (u, v) zu $d[u]$, also $d[u] < d[v] < f[v] < f[u]$ wegen Weißer-Weg-Satz.
2. Fall:$d[u] > d[v]$
   Zum Zeitpunkt $d[v]$ ist $col[u] = wei$ß. Aber da kreisfrei, wird u nicht von v aus entdeckt, da sonst (u, v) Rückwärtskante ist und damit ein Kreis vorliegt. Also kann nur folgendes sein:
   $d[v] < f[v] < d[u] < f[u]$ und es ist $f[v] < f[u]$

□

Beachte aber



$(u, v) \in E$ zbd $f[u] < f[v]$, der Satz gilt nicht. Aber wir haben ja auch einen Kreis! $\neg(A \Rightarrow B)$ bedeutet $(A \wedge \neg B)$ Im kreisfreien Fall etwa so:



Wird u vor v weiß, dann ist klar $f[v] < f[u]$ Wird aber v vor u weiß, dann wird u nicht nachfolger von v, also auch dann $f[v] < f[u]$, Weg kreisfrei.