

8 Anwendung:

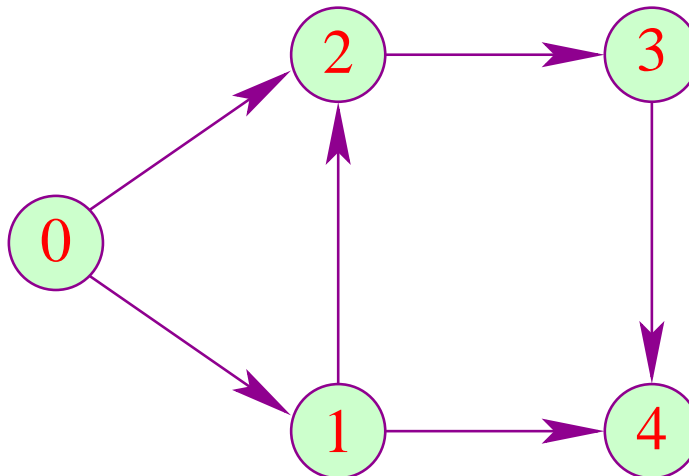
Grundlegende Graph-Algorithmen

- Gerichtete Graphen
- Erreichbarkeit und DFS
- Topologische Sortierung
- Kürzeste Wege

8.1 Gerichtete Graphen

Beobachtung:

- Viele Probleme lassen sich mit Hilfe gerichteter Graphen repräsentieren ...



8.1 Gerichtete Graphen

Beobachtung:

- Viele Probleme lassen sich mit Hilfe gerichteter Graphen repräsentieren ...

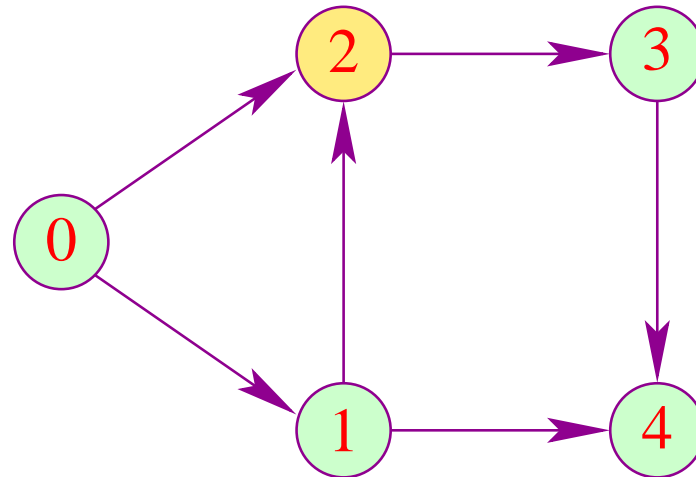
| | Knoten | Kanten |
|-----------------------|---------------------|----------|
| Betrieblicher Prozess | Bearbeitungsschritt | Abfolge |
| Software | Zustand | Änderung |
| Systemanalyse | Komponente | Einfluss |

- Oft sind die Kanten mit Zusatz-Informationen beschriftet :-)

8.2 Erreichbarkeit und DFS

Einfaches Problem:

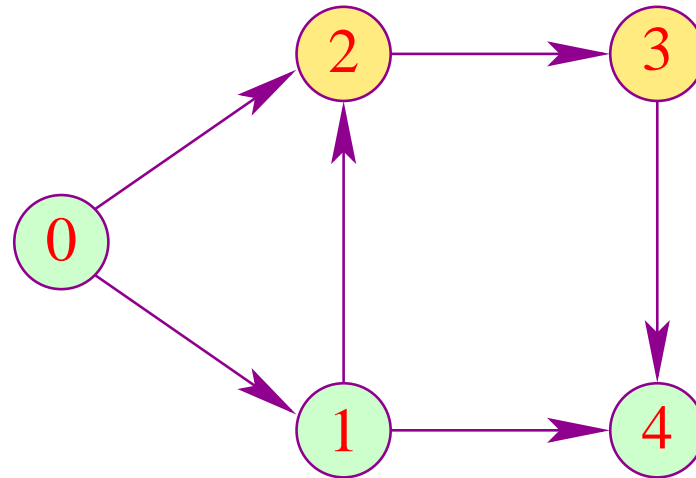
- Gegeben ein Knoten.
- Welche anderen Knoten im Graphen kann man erreichen?



8.2 Erreichbarkeit und DFS

Einfaches Problem:

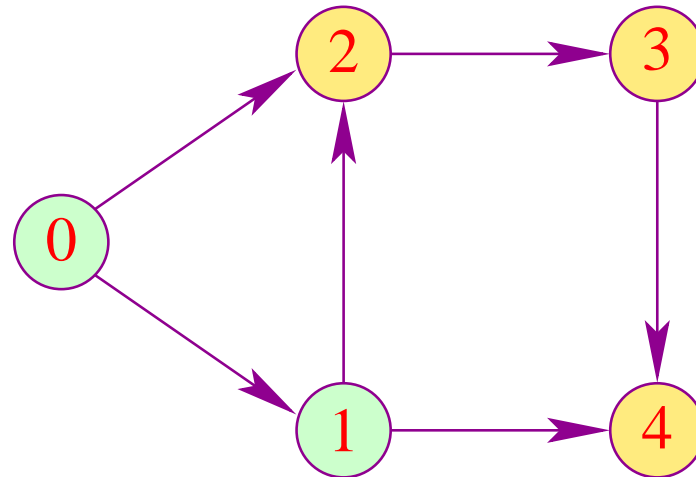
- Gegeben ein Knoten.
- Welche anderen Knoten im Graphen kann man erreichen?



8.2 Erreichbarkeit und DFS

Einfaches Problem:

- Gegeben ein Knoten.
- Welche anderen Knoten im Graphen kann man erreichen?



Eine Datenstruktur für Graphen:

- Wir nehmen an, die Knoten sind fortlaufend mit **0, 1, ...** durch-nummeriert.
- Für jeden Knoten gibt es eine **Liste** ausgehender Kanten.
- Damit wir schnellen Zugriff haben, speichern wir die Listen in einem **Array** ab ...

```
# type node      = int
# type 'a graph = ('a * node) list array
```

Eine Datenstruktur für Graphen:

- Wir nehmen an, die Knoten sind fortlaufend mit **0, 1, ...** durch-nummeriert.
- Für jeden Knoten gibt es eine **Liste** ausgehender Kanten.
- Damit wir schnellen Zugriff haben, speichern wir die Listen in einem **Array** ab ...

```
# type node      = int
# type 'a graph = ('a * node) list array

# let example = [| [(((),1); (((),2))]; [(((),2); (((),4))];
                 [(((),3)); [(((),4))]; [] |] |] ;;
```

Idee:

- Wir verwalten eine Datenstruktur `reachable : bool array`.
- Am Anfang haben alle Einträge `reachable.(x)` den Wert `false`.
- Besuchen wir einen Knoten `x`, der noch nicht erreicht wurde, setzen wir `reachable.(x) <- true` und besuchen alle Nachbarn von `x` :-)
- Kommen wir zu einem Knoten, der bereits besucht wurde, tun wir nix :-))

```

# let fold_li f a arr = let n = Array.length arr
                        in let rec doit i a = if i = -1 then a
                                                else doit (i-1) (f i a arr.(i))
                        in doit (n-1) a;;

# let reach edges x =
  let    n = Array.length edges
  in let reachable = Array.make n false
  in let extract_result () = fold_li
      (fun i acc b -> if b then i::acc
                       else acc) [] reachable
  ...

```

Kommentar:

- `fold_li : (int -> 'a -> 'b -> 'a) -> 'a -> 'b array -> 'a` erlaubt, über ein Array zu iterieren unter Berücksichtigung des Index wie der Werte für den Index :-)
- `Array.make : int -> 'a -> 'a array` legt ein neues initialisiertes Array an.
- Die entscheidende Berechnung erfolgt in der rekursiven Funktion `dfs ...`

```

...
in let rec dfs x = if not reachable.(x) then (
    reachable.(x) <- true;
    List.iter (fun (_,y) -> dfs y)
              edges.(x)
            )
in dfs x;
   extract_result ();;

```

- Die Technik heißt **Depth First Search** oder **Tiefensuche**, weil die Nachfolger eines Knotens **vor** den Nachbarn besucht werden :-)

Kommentar (Forts.):

Der Test am Anfang von `dfs` liefert für jeden Knoten nur `einmal true`.

⇒⇒ Jede Kante wird maximal `einmal` behandelt.

⇒⇒ Der Gesamtaufwand ist proportional zu $n + m$

// n == Anzahl der Knoten

// m == Anzahl der Kanten

8.3 Topologisches Sortieren

Ein **Pfad** oder **Weg** der Länge n in einem gerichteten Graphen ist eine Folge von Kanten:

$$(v_0, _, v_1)(v_1, _, v_2) \dots, (v_{n-1}, _, v_n)$$

Ein Pfad der Länge $n > 0$ von v nach v heißt **Kreis**.

Ein gerichteter Graph ohne Kreise heißt **azyklisch ...**

8.3 Topologisches Sortieren

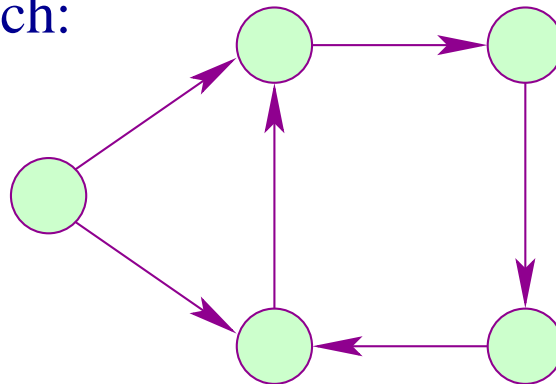
Ein **Pfad** oder **Weg** der Länge n in einem gerichteten Graphen ist eine Folge von Kanten:

$$(v_0, _, v_1)(v_1, _, v_2) \dots, (v_{n-1}, _, v_n)$$

Ein Pfad der Länge $n > 0$ von v nach v heißt **Kreis**.

Ein gerichteter Graph ohne Kreise heißt **azyklisch** ...

zyklisch:



8.3 Topologisches Sortieren

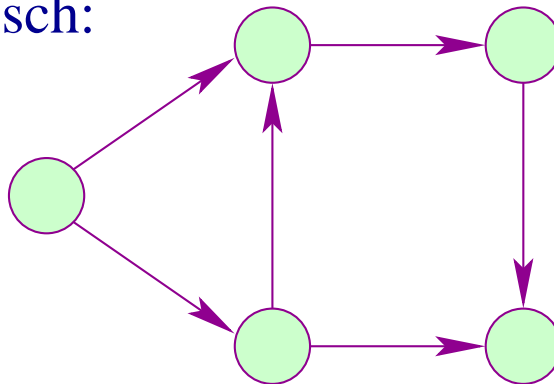
Ein **Pfad** oder **Weg** der Länge n in einem gerichteten Graphen ist eine Folge von Kanten:

$$(v_0, _, v_1)(v_1, _, v_2) \dots, (v_{n-1}, _, v_n)$$

Ein Pfad der Länge $n > 0$ von v nach v heißt **Kreis**.

Ein gerichteter Graph ohne Kreise heißt **azyklisch** ...

azyklisch:

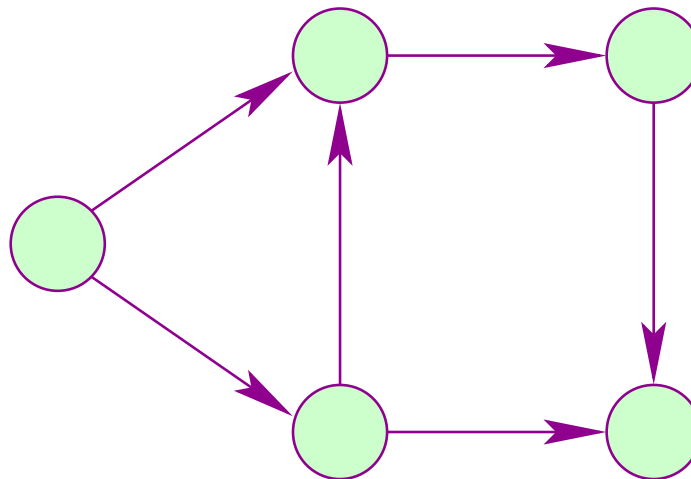


Aufgabe:

Gegeben: ein gerichteter Graph.

Frage: Ist der Graph azyklisch?

Wenn ja: Finde eine **lineare Anordnung** der Knoten!

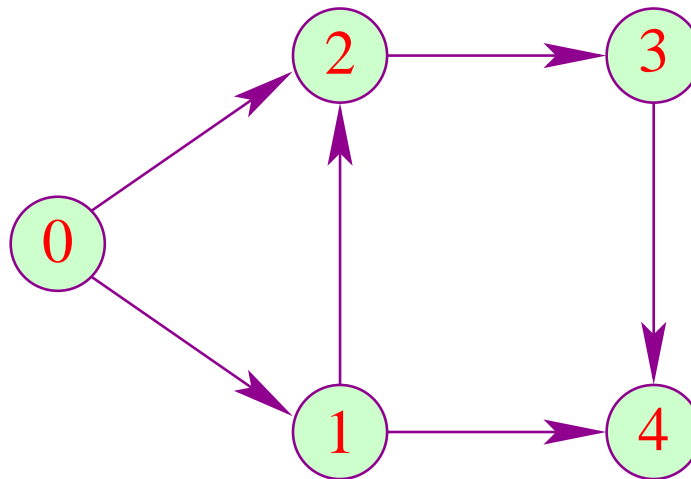


Aufgabe:

Gegeben: ein gerichteter Graph.

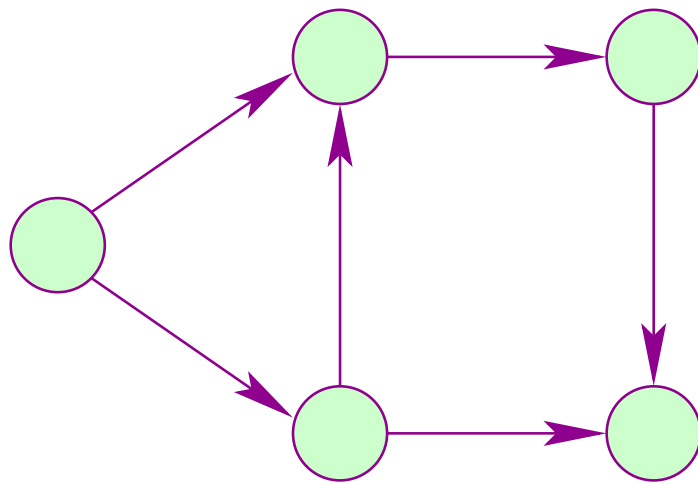
Frage: Ist der Graph azyklisch?

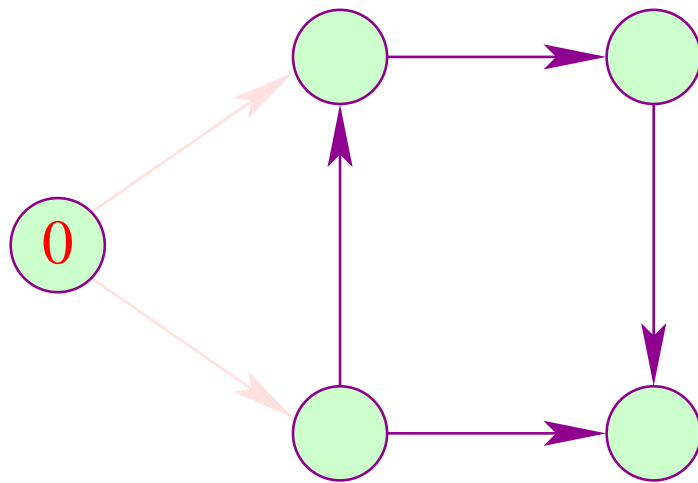
Wenn ja: Finde eine **lineare Anordnung** der Knoten!

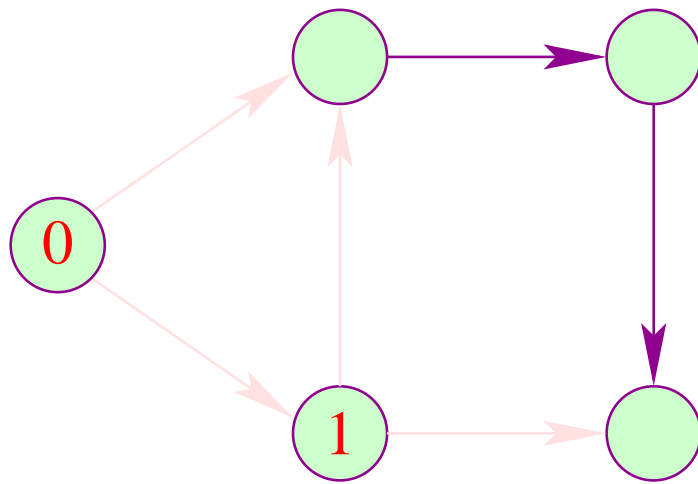


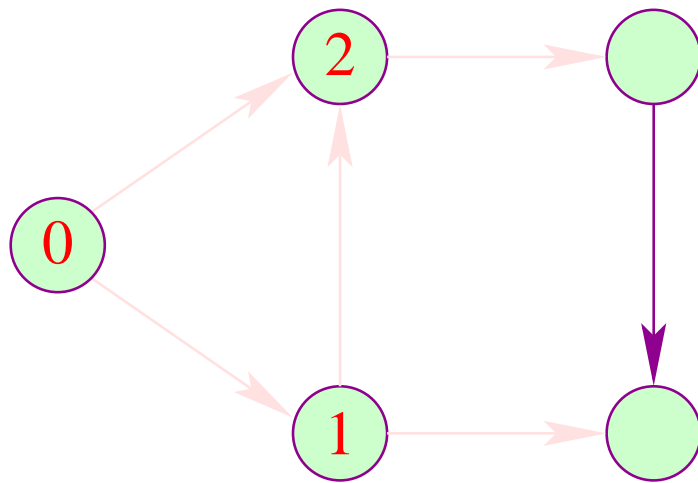
Idee:

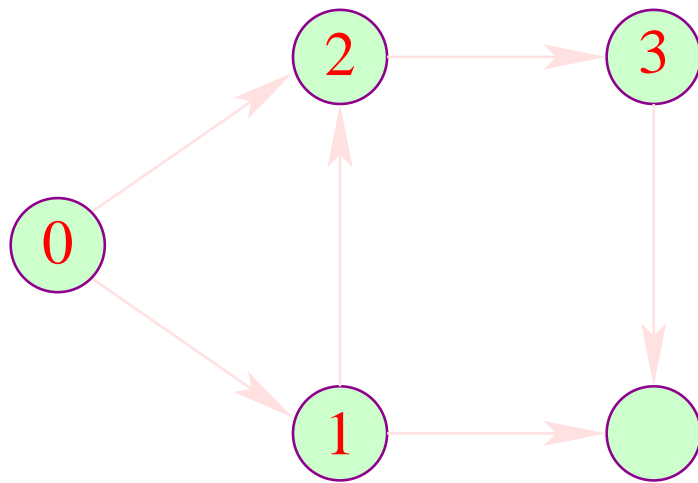
- Suche einen Knoten **ohne eingehende Kanten**.
- Besitzt jeder Knoten eingehende Kanten, muss der Graph einen Kreis enthalten **;-)**
- Gibt es einen Knoten ohne eingehende Kanten, können wir diesen als kleinstes Element auswählen **:-)**
- Entferne den Knoten mit allen seinen **ausgehenden Kanten !**
- Löse das verbleibende Problem ...

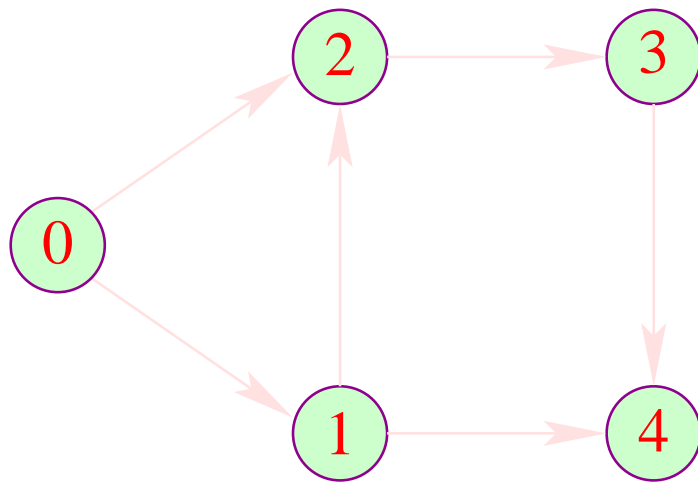












Implementierung:

- Für jeden Knoten x benutzen wir einen Zähler $\text{count}(x)$, der die Anzahl der eingehenden Kanten zählt.
- Haben wir einen Knoten, dessen Zähler 0 ist, nummerieren wir ihn :-)
- Dann durchlaufen wir seine ausgehenden Kanten.
- Für jeden Nachbarn dekrementieren wir den Zähler und fahren rekursiv fort.

⇒ wieder Depth-First-Search

```
# let top_sort edges =
    let n = Array.length edges
  in let count = Array.make n 0
  in let inc y = count.(y) <- count.(y) +1
  in let dec y = count.(y) <- count.(y) -1
  in for x=0 to n-1 do
      List.iter (fun (_,y) -> inc y) edges.(x)
    done ;
    let value = Array.make n (-1)
  ...
```

```

in let next = ref 0
in let number y = value.(y) <- !next;
      next := !next + 1
in let rec dfs x = if count.(x) = 0 then (
      number x;
      List.iter (fun (_,y) ->
        dec y;
        dfs y)
      edges.(x))

in for x=0 to n-1 do
  if value.(x) = -1 then dfs x
done; value;;

```

Diskussion:

- `for y=... to ... do ... done` ist eine vereinfachte `while`-Schleife ;-)
- Für die Initialisierung des Arrays `count` iterieren wir einmal über alle Knoten und für jeden Knoten einmal über alle ausgehenden Kanten.

⇒ Aufwand: proportional $n + m$

- Die Hauptschleife behandelt jeden Knoten maximal einmal.
- `dfs` wird maximal einmal für jeden Knoten und jede Kante aufgerufen und behandelt jede Kante maximal einmal.

⇒ Aufwand: proportional $n + m$

⇒ Der Gesamtaufwand ist linear :-)