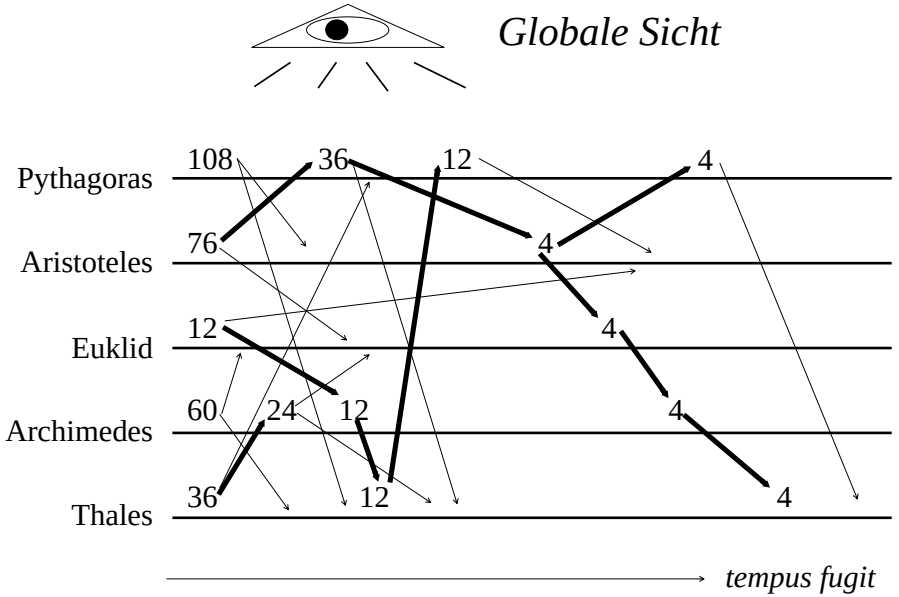
Berechnungsablauf und Zeitdiagramm



Ablauf einer *möglichen* "verteilten Berechnung" mit einem *Zeitdiagramm*.

Nicht-deterministisch, abhängig von der Nachrichtenlaufzeit!

- *Terminiert* wenn (?):

  Das ist unrealistisch!

  (a) jeder Prozess den ggT kennt,

  (b) alle den gleichen Wert haben,

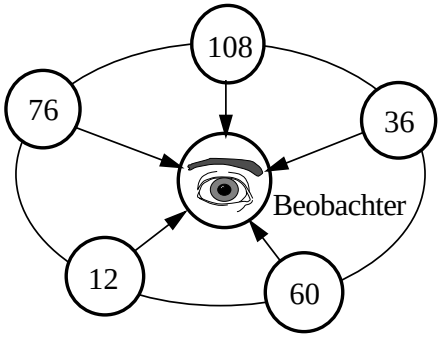
  Beweisbare math. Eigenschaft

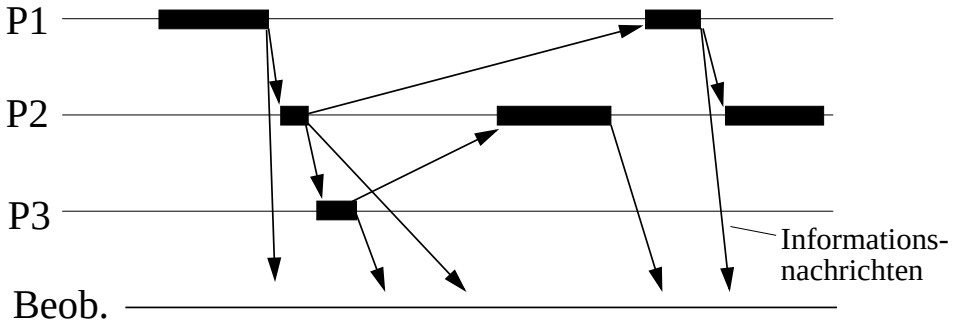
  (c) alle Prozesse passiv sind und keine Nachricht unterwegs ist.

  Diese "stabile" Stagnationseigenschaft ist *problemunabhängig*!

  - Was ist adäquat?
  - Wie feststellen?

Globaler ggT-Beobachter?





Bekommt der Beobachter ein "richtiges Bild" des Geschehens?

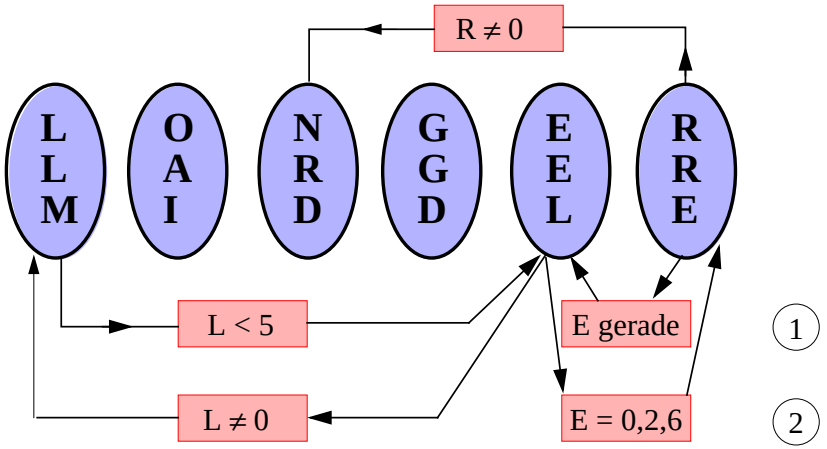
- was heisst das genau?
- und wenn Informationsnachrichten verschieden schnell sind?
- könnte der Beobachter einen Zwischenzustand (z.B. "alle haben den Wert 12") irrtümlich als Endzustand interpretieren?
- wie stellt er überhaupt das Ende der Berechnung fest?

Ein weiteres Beispielproblem: Paralleles Lösen von "Zahlenrätseln"

```
 LONGER          207563
+LARGER    ->   +283563
-------         -------
 MIDDLE          491126
```

Pro Spalte ein Prozess

- Reaktives Verhalten: Auslösen atomarer Aktionen
- Propagieren neuer Erkenntnisse (= Einschränkungen)
  ("parallele Constraint-Propagation")



Endergebnis:

$1 \le L \le 4$
$M \ge 2$
R = 1,3,5,6,8
E = 0,2,6
sonst keine Einschränkung

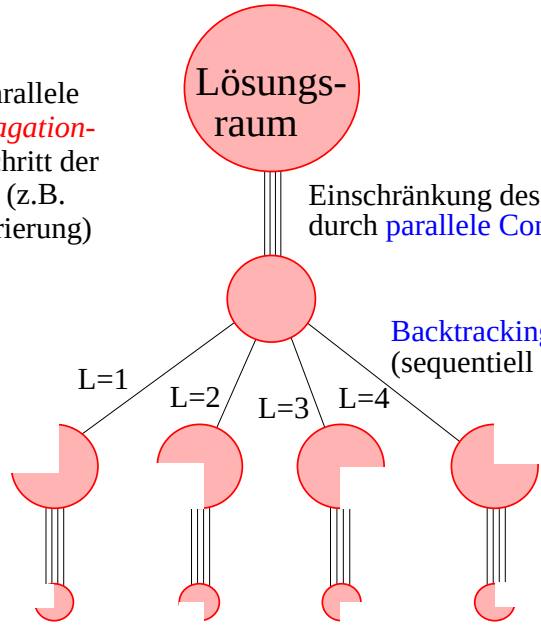
Probleme:

1) Keine eindeutige Lösung
   --> Backtrack-Algorithmus
2) Entdeckung der "Stagnation"?
   (Ende der Parallelphase)

Offensichtlich *gleiches Schema* wie ggT-Berechnung!

Der aufgesetzte Backtrack-Algorithmus

Abwechselnd: Parallele *Constraint-Propagation-Phase* und ein Schritt der *Backtrack-Phase* (z.B. Hypothesengenerierung)



- Hypothese = beliebige Menge von Constraints
  (von einem "Orakel" statt von einer Spalte)

- Einheit, die die Hypothesen generiert und verwaltet, muss die Terminierung einer Constraint-Phase feststellen können
  - wie würde man hier die Terminierung zweckmässigerweise definieren?
  - problembezogen wie bei ggT ("alle Werte identisch") hier nicht einfach
  - "alle passiv und keine sinnvolle Nachricht mehr unterwegs"?

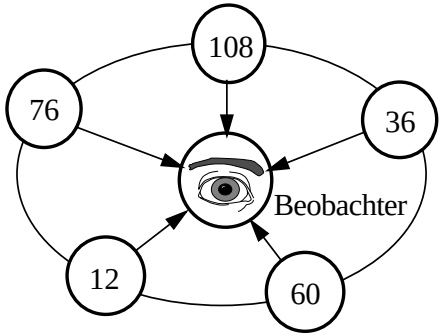
Bemerkungen:

- nicht unbedingt beste Parallelisierungsstrategie!
- Problem ist (in Verallgemeinerung) NP-vollständig

Übungen (1) zur Vorlesung "Verteilte Algorithmen"...

a) Man zeichne Raum-Zeit-Diagramme für verschiedene Abläufe des verteilten ggT-Algorithmus

b) Wie kann man beweisen, dass für *jeden* denkbaren Ablauf das Endergebnis stets der ggT ist?



c) Bleibt der Algorithmus (und/oder der Beweis) korrekt, wenn im Algorithmus $y<M_i$ durch $y \leq M_i$ ersetzt wird?

Verhaltensbeschreibung eines Prozesses P_i:

{Eine Nachricht $<y>$ ist eingetroffen}
if $y < M_i$ **then**
  M_i := mod(M_i-1,y)+1;
  send $<M_i>$ **to** all neighbours;
fi

...Übungen (1)

d) Man vergleiche die verteilte Berechnung des ggT-Algorithmus für zwei Zahlen mit dem üblichen sequentiellen ggT-Algorithmus für zwei Zahlen

e) Genügt es auch, nur in Uhrzeigerrichtung eine Nachricht zu senden (anstatt an beide Nachbarn)?

f) Kann statt des Ringes eine andere Topologie verwendet werden? Welche?

g) Formalisieren Sie für Zeitdiagramme den Begriff (potentiell, indirekt) "kausal abhängig" als Halbordnung über "Ereignissen"

h) Wie kann man erreichen, dass ein ggT-Beobachter (der über jede Wertänderung eines Prozesse informiert wird) eine "kausaltreue" Beobachtung macht?

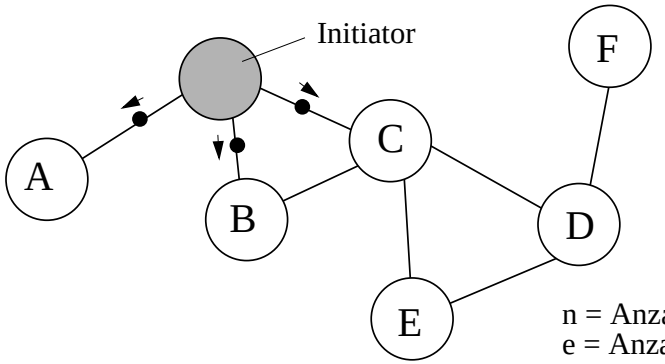
i) Beobachtungen sind eine lineare Ordnung von (beobachteten) Ereignissen. In welcher Beziehung steht die oben erwähnte Halbordnung zu dieser linearen Ordnung? Können Sie eine Vermutung darüber anstellen, was der Schnitt aller möglichen kausaltreuen Beobachtungen einer verteilten Berechnung aussagt?

j) Wie kann der Beobachter die Terminierung erkennen?

Flooding, Echo-Algorithmus, Broadcast

Informationsverteilung durch “Fluten”

“flooding”



n = Anzahl der Knoten
e = Anzahl der Kanten

- Voraussetzung: zusammenhängende Topologie
- Prinzip: jeder erzählt *neues* Gerücht allen anderen Freunden
- Kein Routing etc. notwendig

wieder das gleiche Prinzip wie beim ggT und beim Zahlenrätsel!

- Wieviele Nachrichten werden versendet?
  - jeder Knoten sendet über alle seine inzidenten Kanten (--> 2e)
  - jedoch nicht über seine Aktivierungskante zurück (--> -n)
  - Ausnahme: Initiator (--> +1)

  ==> *Also*: 2e - n + 1

- Frage: Wie *Terminierung* feststellen?

  d.h.: wie erfährt der Sender (= Initiator), wann alle erreicht wurden?
  (das ist für “sicheren” oder “synchronen” Broadcast notwendig)

Flooding-Algorithmus - eine etwas formalere Spezifikation

- Zwei *atomare Aktionen* für jeden Prozess:

- wechselseitig ausgeschlossen
- "schlagartig"?
- ununterbrechbar?

Assertion (muss wahr sein, damit Aktion ausgeführt wird)

R: {Eine Nachricht <info> kommt an}
if not informed **then**
send <info> **to** all other neighbors;
informed := true;
fi

Natürlich auch "merken" der per Nachricht erhaltenen Information <info>

I: {not informed}
send <info> **to** all neighbors;
informed := true;

- initial sei informed=false

- Aktion R wird nur bei Erhalt einer Nachricht ausgeführt
  - "message driven"

- Aktion I wird vom Initiator *spontan* ausgeführt
  - darf es mehrere *konkurrente* Initiatoren geben?

Terminierungserkennung von Flooding

1) Jeder Prozess informiert (ggf. indirekt) den Initiator (oder einen Beobachter) per *Kontrollnachricht*, wenn er eine *Basisnachricht* erhält; Initiator zählt bis 2e-n+1

   - Nachteile?
     - n und e müssen dem Initiator bekannt sein
     - indirektes Informieren kostet ggf. viele Einzelnachrichten

   - Nachrichtenkomplexität?

   - Variante: Prozess sendet Kontrollnachricht, wenn er *erstmalig* eine Basisnachricht erhält; Initiator zählt bis n-1
     - n muss dem Initiator bekannt sein
     - Terminierung in dem Sinne, dass alle informiert sind - es können dann aber noch (an sich nutzlose) Basisnachrichten unterwegs sein

2) *Überlagerung* eines geeigneten Kontrollalgorithmus, der die Berechnung des Flooding-Verfahrens beobachtet und die Terminierung meldet
   - später mehr zu überlagerten Terminierungserkennungsverfahren

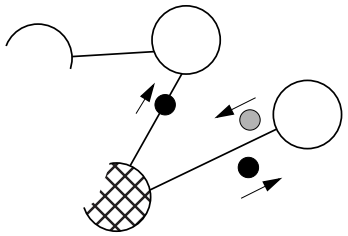
3) *Bestätigungsnachrichten* (acknowledgements)
   - direktes Bestätigen einer Nachricht funktioniert nicht
   - indirekte Bestätigungsnachrichten: ein Knoten sendet erst dann ein ack, wenn er selbst für alle seine Nachrichten acks erhalten hat
   - klappt diese Idee? auch wenn der Graph Zyklen enthält? wieso?

Flooding mit Quittungsmeldungen

(Originalversion des *Echo-Algorithmus* von Chang '82)

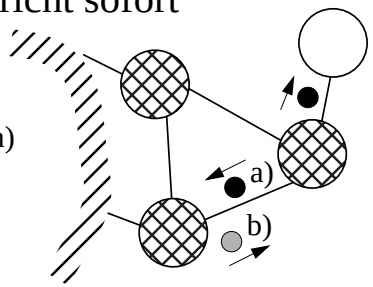
Prinzip: Ein Prozess versendet eine Quittung für eine empfangene Nachricht erst dann, wenn er für alle von ihm selbst versendeten Nachrichten Quittungen erhalten hat

- Ein Knoten mit Grad 1 ("Blatt") sendet sofort eine Quittung zurück



Nachricht des zugrundeliegenden (Flooding)-Algorithmus

- Ein Knoten, der bereits eine Basisnachricht erhalten hat, quittiert jede weitere Basisnachricht sofort
  - *Prinzip*: "bin schon informiert"
  - *Wirkung*: Zyklen werden aufgebrochen (als wäre die Kante gar nicht vorhanden)
  - *Konsequenz*: es entsteht ein Baum



- Terminiert, wenn Initiator alle Quittungen erhalten hat

- Wieviele Quittungen / Nachrichten insgesamt?

Der Echo-Algorithmus

(PIF-Variante von A. Segal, 1983)

Propagation of Information with Feedback

- Ähnliches Verfahren 1980 von Dijkstra/Scholten: "Diffusing Computations"

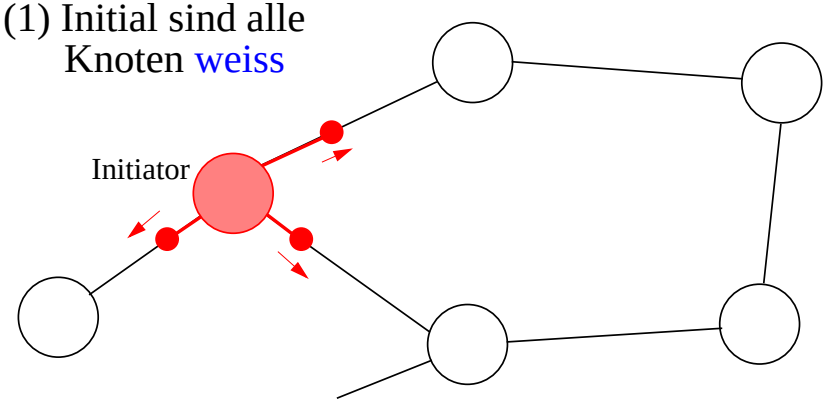
Ausgehend von einem einzigen Initiator:

- Paralleles *Traversieren* eines bel. (zusammenhängenden ungerichteten) Graphen mit 2e Nachrichten
- Terminierung klar durch "Vollzugsmeldung"
- Idee: *Indirektes* acknowledge
- Hinwelle durch "*Explorer*": Verteilen von Information
- Rückwelle durch "*Echos*": Einsammeln einer verteilten Information
- Aufbau eines *spannenden Baumes* ("Echo Kanten": jeder Knoten sendet genau ein Echo)

Paralleler *Wellenalgorithmus*:

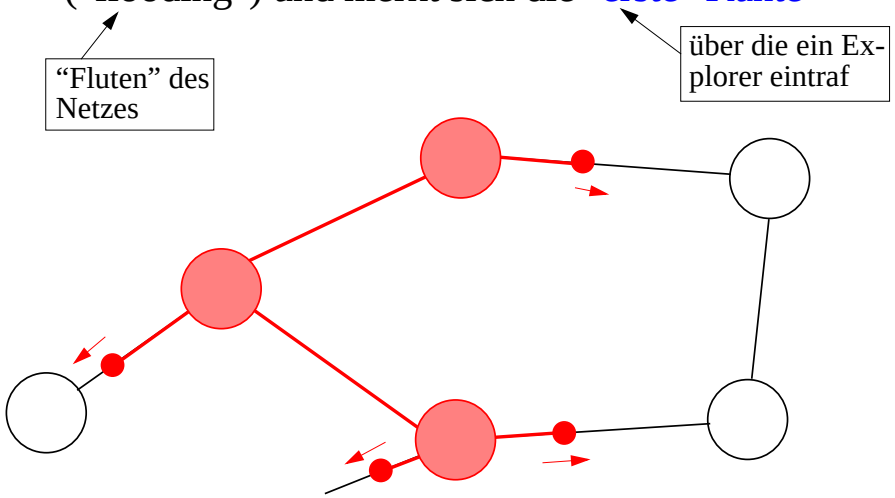
- virtueller broadcast
- *Basisalgorithmus* für andere Verfahren ("underlying algorithm"; "superposition")

(1) Initial sind alle Knoten weiss

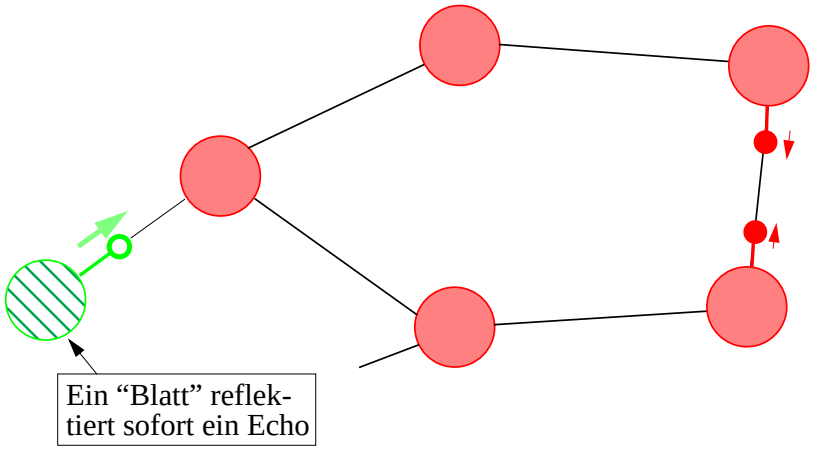


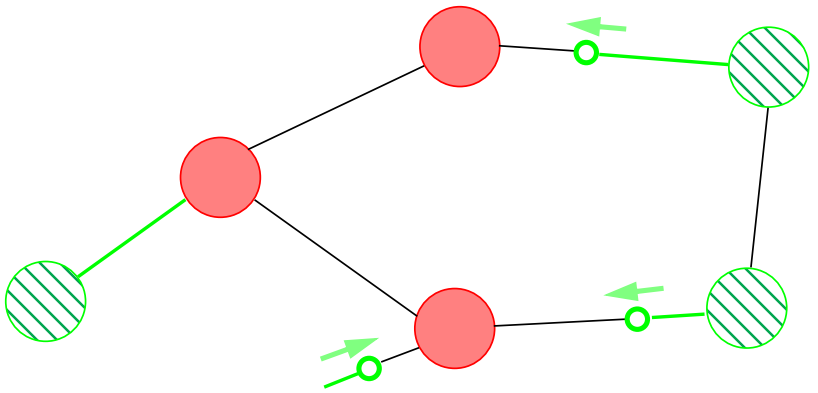
- Der (eindeutige) Initiator wird rot und sendet (rote) *Explorer* über alle seine Kanten

(2) Ein weisser Knoten, der einen Explorer bekommt, sendet Explorer über alle seine anderen Kanten ("flooding") und merkt sich die "erste" Kante



(3) Ein (roter) Knoten, der über alle seine Kanten einen Explorer *oder* ein Echo erhalten hat, wird grün und sendet ein (grünes) *Echo* über seine "erste" Kante

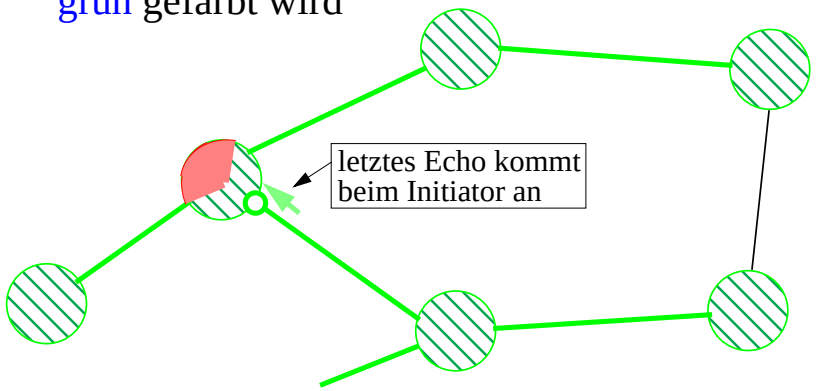




Beachte: *Atomare Aktionen* --> Explorer können sich höchstens auf Kanten begegnen, nicht aber bei Knoten!

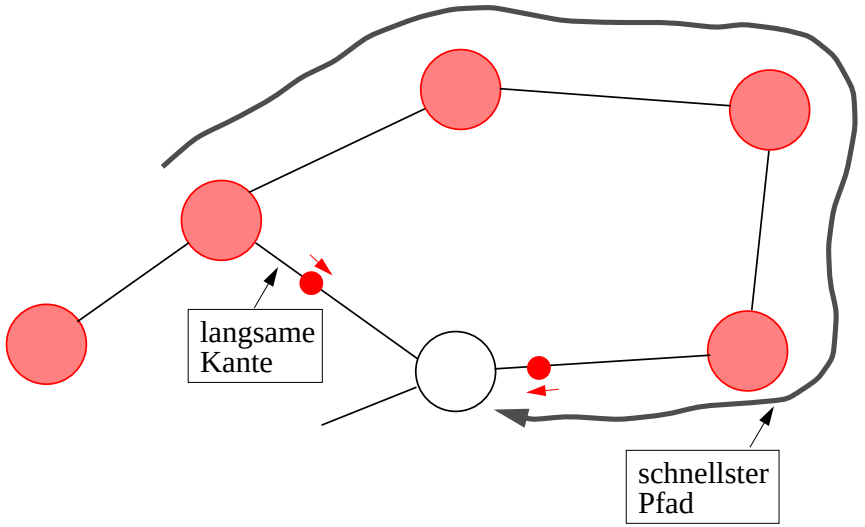
Auf einer Kante, wo sich zwei Explorer begegnen, wird der Zyklus aufgebrochen

(4) Das Verfahren ist beendet, wenn der Initiator grün gefärbt wird

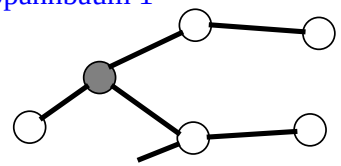


- Grüne Kanten bilden einen *Spannbaum*
  - alle Knoten bis auf den Initiator haben eine "erste" Kante
  - “grüner Graph” ist zusammenhängend
- Über jede Kante laufen genau 2 Nachrichten:
  - entweder ein Explorer und ein gegenläufiges Echo, oder zwei Explorer, die sich begegnen --> Nachrichtenkomplexität = 2e
- Verfahren ist *schnell*: parallel und "bester" (?) Baum
- Ereignis "rot werden" in jedem Prozess definiert eine Welle (*Hinwelle*)
- Ereignis "grün werden" in jedem Prozess --> *Rückwelle*
- Es darf nicht mehr als einen Initiator geben:
  - was geschieht sonst?
  - wie kann man dem Problem mehrerer Initiatoren begegnen?

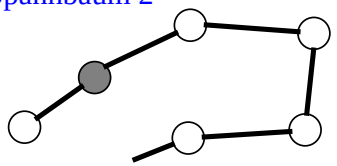
Echo-Algorithmus ist *nicht-deterministisch*, es können (je nach Geschwindigkeit einer "Leitung") *verschiedene* Spannbäume entstehen!



Spannbaum 1



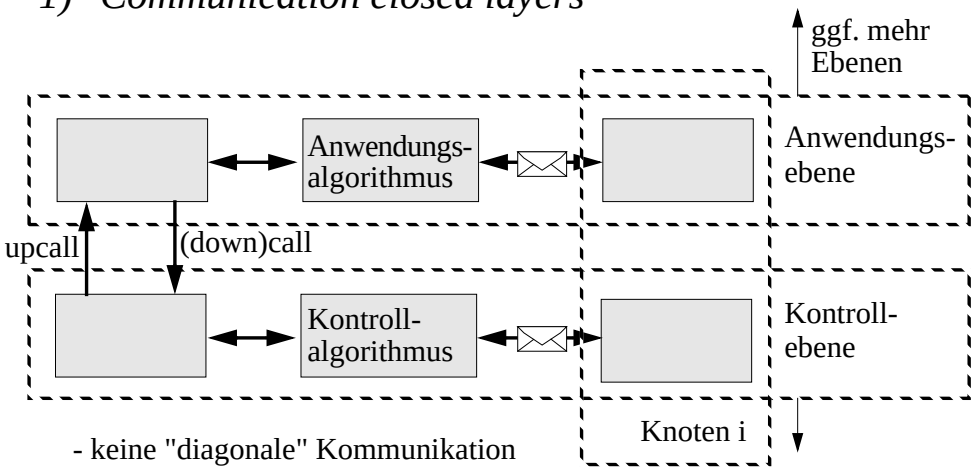
Spannbaum 2



- Inwiefern ist die PIF-Variante besser als die Originalversion?
  - Nachrichtenkomplexität
  - Einfachheit / Eleganz

Überlagerung ("Superposition")

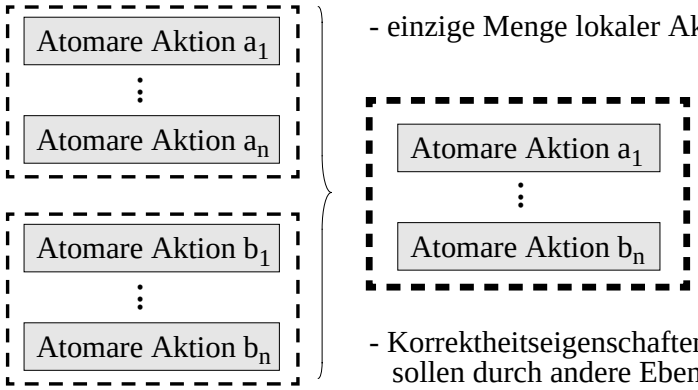
1) "Communication closed layers"



- keine "diagonale" Kommunikation
- call und upcall (innerhalb eines Knotens) nicht notw. mittels Nachrichten sondern i.a. durch Aufruf lokaler Aktionen (z.B. Prozeduren)
- Kommunikation zwischen der Anwendungs- und Kontrollebene (innerhalb eines Knotens) typw. über gemeinsame Variablen (wobei i.a. nur eine Ebene schreiben darf)

---

2) Zusammenbau und Vereinigung von Aktionen



- einzige Menge lokaler Aktionen
- Korrektheitseigenschaften einer Ebene sollen durch andere Ebenen nicht verletzt werden!

Echo-Algorithmus und upcall-Technik

```
receive <ECHO(...)> or <EXPLORER(...)> from p;
if COLOR = white then
    upcall "first EXPLORER(...) received";
    COLOR := red;
    N := 0; PRED := p;
    send <EXPLORER(...)> to neighbors\{PRED};
fi;
if echo received then upcall "ECHO(...) received"; fi;
N := N+1;
if N = | neighbors | then
    COLOR := green;
    if INITIATOR then upcall  "terminated";
                 else upcall "ready to send echo";
                      send <ECHO(...)> to PRED;
    fi;
fi;
```

Mit "upcalls" wird die darüberliegende Anwendung vom Echo-Algorithmus benachrichtigt; die Anwendung kann entweder die Kontrolle sofort zurückgeben oder z.B. Parameterwerte vorbereiten, die dann mit Nachrichten des Echo-Algorithmus mitgesendet werden.

```
{COLOR = white}
INITIATOR := true;
COLOR := red;
N := 0;
send <EXPLORER(...)> to neighbors;
```

Aktion, die von der Anwendung mit einem "normalen downcall" gestartet wird.

Echo-Algorithmus...

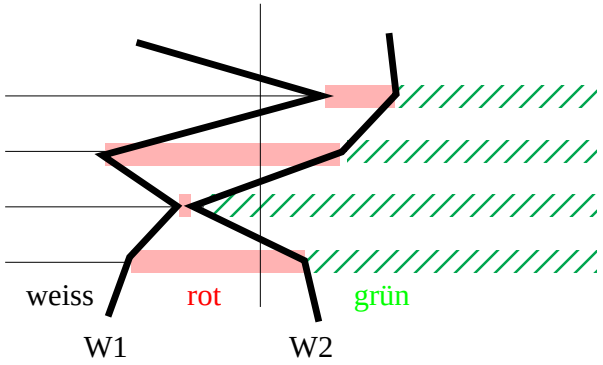
- Jeder Knoten wird erst *rot* und dann *grün*
- Rote Phase ist bei "Blättern" allerdings recht kurz
- Ein grüner Knoten hat keine weisse Nachbarn

==> Eine von einem grünen Knoten versendete Basisnachricht wird nicht von einem weissen Knoten empfangen

- beachte: gilt nur für *direkte* Nachrichten, nicht für *Nachrichtenketten!*

==> Weisse und grüne Phase sind in "gewisser Weise" disjunkt

- obwohl es globale Zeitpunkte geben kann, wo ein Knoten bereits grün ist, während ein anderer (nicht direkt benachbarter!) noch weiss ist!



- "Rot werden" und "grün werden" definieren zwei *Wellen*
  - mit diesen Wellen kann Information transportiert werden (verteilen bzw. akkumulieren)
  - Echo-Algorithmus wird daher oft als Basis für andere Verfahren verwendet