Verteilte Algorithmen und Datenstrukturen

Kapitel 5: Prozessorientierte Datenstrukturen

Prof. Dr. Christian Scheideler

Institut für Informatik

Universität Paderborn

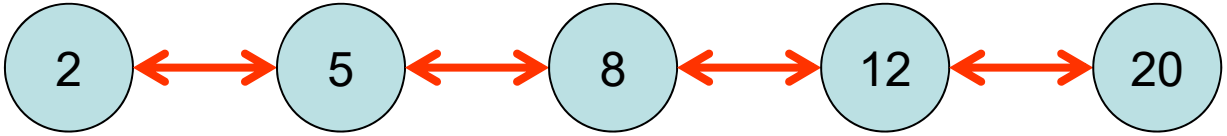
Prozessorientierte Datenstrukturen

Übersicht:

- Sortierte Liste
- Sortierter Kreis
- De Bruijn Graph
- Skip Graph

Sortierte Liste

Idealzustand: herzustellen durch Build-List Protokoll



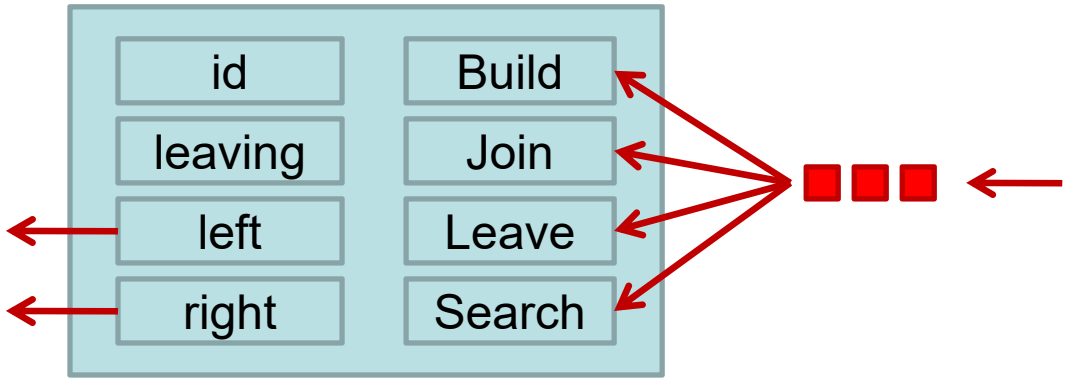
Operationen:

- Join(v): fügt Knoten v in Liste ein
- Leave(v): entfernt Knoten v aus Liste
- Search(id): sucht nach Knoten mit ID id in Liste

Sortierte Liste

Variablen innerhalb eines Knotens v:

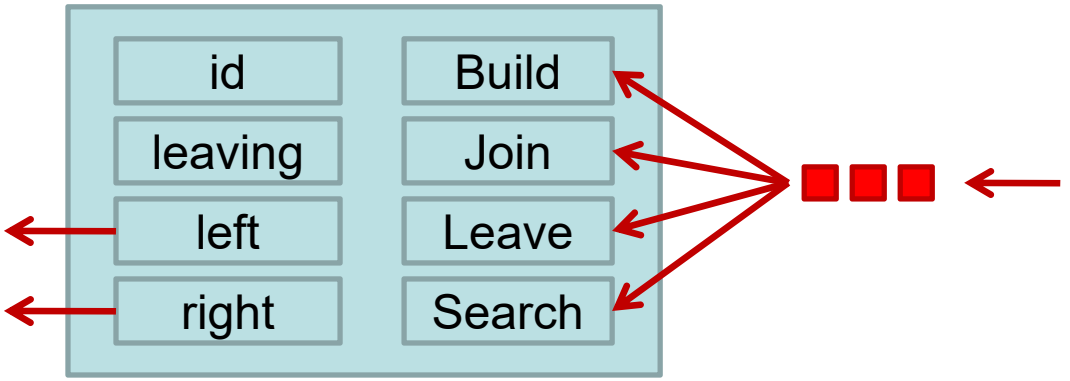
- id: eindeutiger Name von v (wir schreiben auch id(v))
- leaving$\in${true,false}: zeigt an, ob v System verlassen will
- left $\in V \cup \{\perp\}$: linker Nachbar von v, d.h. id(left)<id(v) (falls id(left) definiert ist)
- right $\in V \cup \{\perp\}$: rechter Nachbar von v, d.h. id(right)>id(v) (falls id(right) definiert ist)



Sortierte Liste

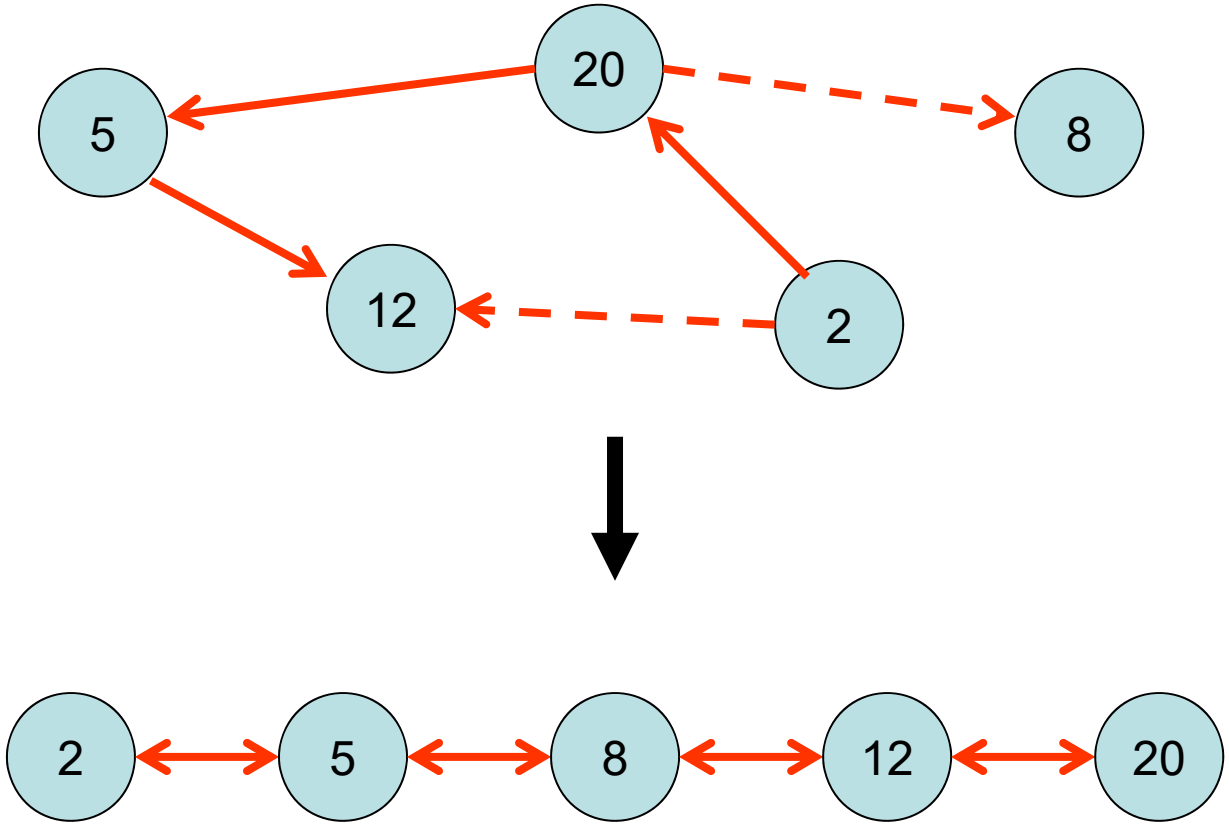
Vereinfachende Annahmen:

- id(v)=v (bzw. id(v)=f(v) für eine fest vorgegebene und bekannte Funktion f), d.h. wir interpretieren die Referenz (z.B. die IP-Adresse) eines Knotens v als seine ID.
- Es gibt keine Referenzen nicht (mehr) existierender Knoten im System (sonst bräuchten wir einen Fehlerdetektor).



Sortierte Liste

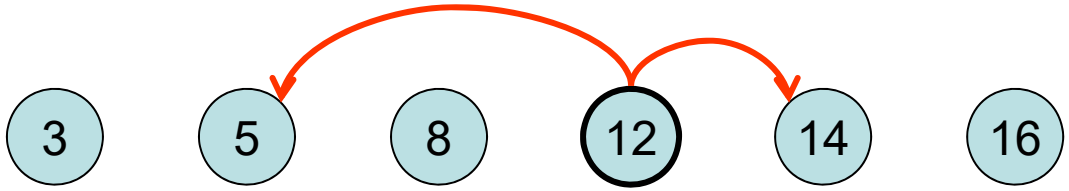
Build-List Protokoll:



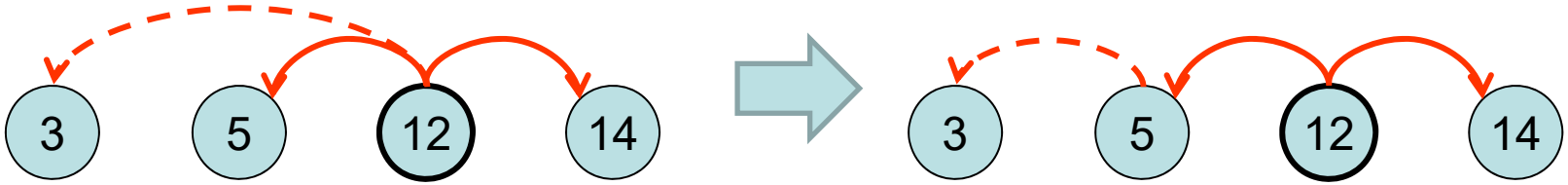
Build-List Protokoll

Listenaufbau über Linearisierung:

Idee: behalte Kanten zu nächsten Nachbarn und delegiere Rest weiter.



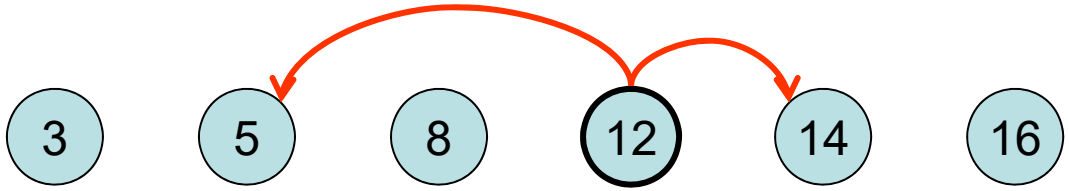
Bei Aufruf linearize(3): 12 generiert Anfrage 5←linearize(3)



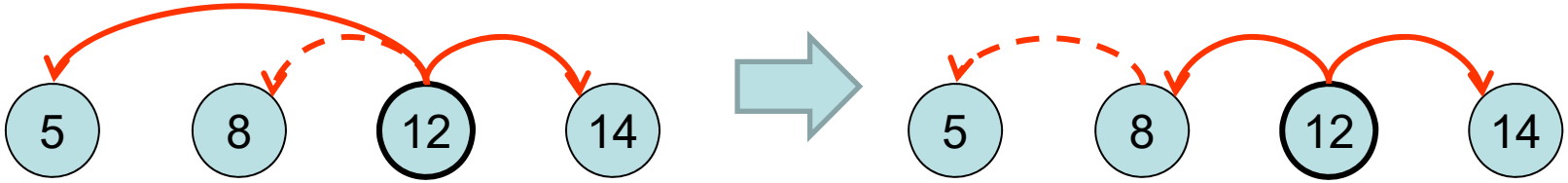
Build-List Protokoll

Listenaufbau über Linearisierung:

Idee: behalte Kanten zu nächsten Nachbarn und delegiere Rest weiter.



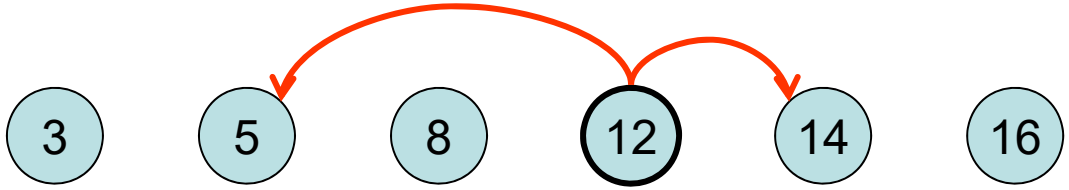
Bei Aufruf linearize(8): 12 setzt 12.left:=8 und generiert Anfrage 8←linearize(5)



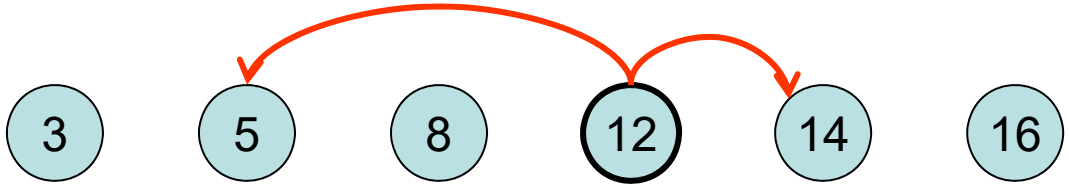
Build-List Protokoll

Listenaufbau über Linearisierung:

Idee: behalte Kanten zu nächsten Nachbarn und delegiere Rest weiter.



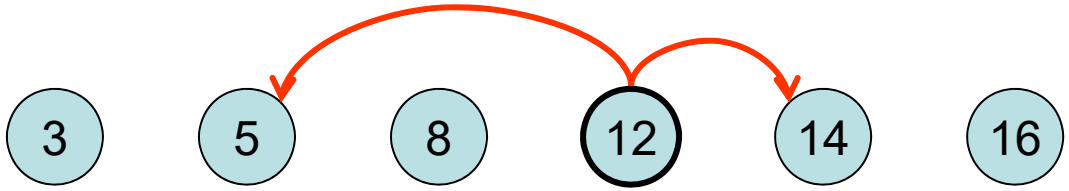
Bei Aufruf linearize(5) oder linearize(14): 12 tut nichts (außer Kante mit existierender zu verschmelzen)



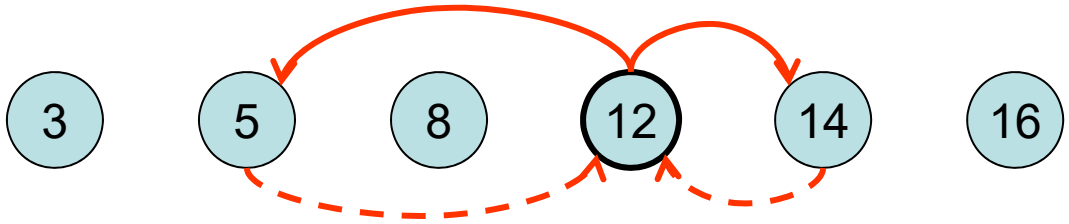
Build-List Protokoll

Listenaufbau über Linearisierung:

Idee: behalte Kanten zu nächsten Nachbarn und delegiere Rest weiter.



Bei timeout() generiert 12 Aufrufe 5←linearize(12) und 14←linearize(12), stellt sich also Nachbarn vor.



Build-List Protokoll

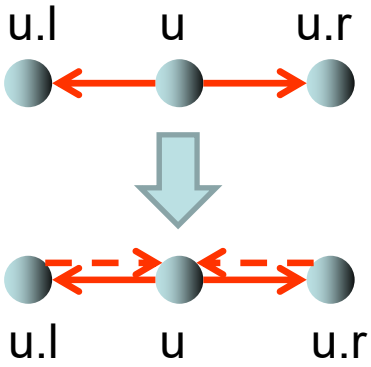
Vereinfachende Annahmen:

- Jedesmal wenn left=$\perp$ ist, nehmen wir für Vergleiche an, dass id(left)=-∞ ist.
- Jedesmal wenn right=$\perp$ ist, nehmen wir für Vergleiche an, dass id(right)=+∞ ist.
- Ein Aufruf u$\leftarrow$action(v) findet nur dann statt, wenn u und v nicht leer sind.

Build-List Protokoll

```
timeout: true →
   { durchgeführt von Knoten u }
   if id(left)<id then
         left←linearize(this)
      else
         this←linearize(left)
         left:= ⊥
   if id(right)>id then
         right←linearize(this)
      else
         this←linearize(right)
         right:= ⊥
```

id(u.left)<id(u) und
id(u.right)>id(u):



In Bildern: u.l statt u.left, u.r statt u.right

Build-List Protokoll

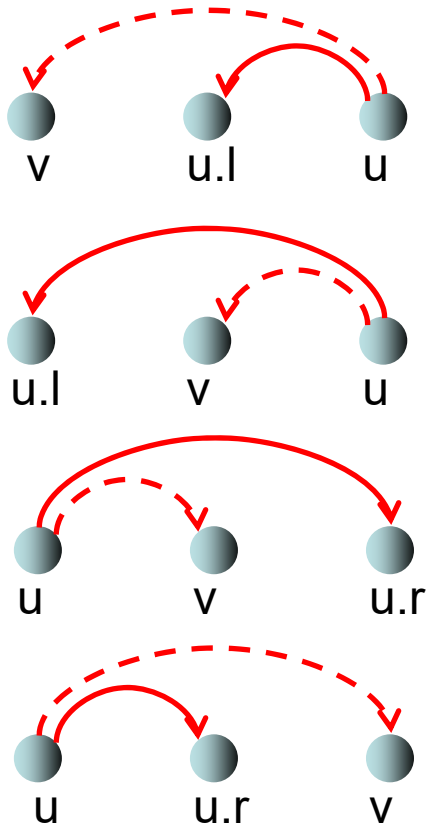
```
timeout: true →
   { durchgeführt von Knoten u }
   if id(left)<id then
         left←linearize(this)
      else
         this←linearize(left)
         left:= ⊥
   if id(right)>id then
         right←linearize(this)
      else
         this←linearize(right)
         right:= ⊥
```

id(u.left)>id(u) oder id(u.right)<id(u): Referenz wird weg delegiert, so dass falsche Belegung verschwindet

In Bildern: u.l statt u.left, u.r statt u.right

Build-List Protokoll

```
linearize(v) →
   { ausgeführt in Knoten u }
   if id(v)<id(left) then
      left←linearize(v)
   if id(left)<id(v)<id then
      v←linearize(left)
      left:=v
   if id<id(v)<id(right) then
      v←linearize(right)
      right:=v
   if id(right)<id(v) then
      right←linearize(v)
```



Build-List Protokoll

```
linearize(v) →
    { ausgeführt in Knoten u }
    if id(v)<id(left) then
        left←linearize(v)
    if id(left)<id(v)<id then
        v←linearize(left)
        left:=v
    if id<id(v)<id(right) then
        v←linearize(right)
        right:=v
    if id(right)<id(v) then
        right←linearize(v)
```

Satz 5.1: Referenz v geht nur dann in linearize verloren, wenn id(v)∈{id,id(left),id(right)} ist, selbst wenn die Belegungen von left und right falsch sind, d.h. id(left)≥id oder id(right)≤id.

Beweis: Übung (Fallunterscheidungen).

Sortierte Liste

Erinnerung: Sei DS eine Datenstruktur.

Definition 3.6: Build-DS stabilisiert die Datenstruktur DS, falls Build-DS

1. DS für einen beliebigen Anfangszustand mit schwachem Zusammenhang und eine beliebige faire Rechnung in endlicher Zeit in einen legalen Zustand überführt (Konvergenz) und
2. DS für einen beliebigen legalen Anfangszustand in einem legalen Zustand belässt (Abgeschlossenheit),

sofern keine Operationen auf DS ausgeführt werden und keine Fehler auftreten.

Legaler Zustand für sortierte Liste:

- explizite Kanten formen sortierte Liste, wobei für jedes u gilt, dass id(u.left)<id(u)<id(u.right)
- IDs nicht korrumpiert (kein Problem, da diese hier an Referenzen gekoppelt sind)

Sortierte Liste

Satz 5.2 (Konvergenz): Build-List erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ eine sortierte Liste (sofern E_M nur aus linearize Anfragen besteht).

Beweis:

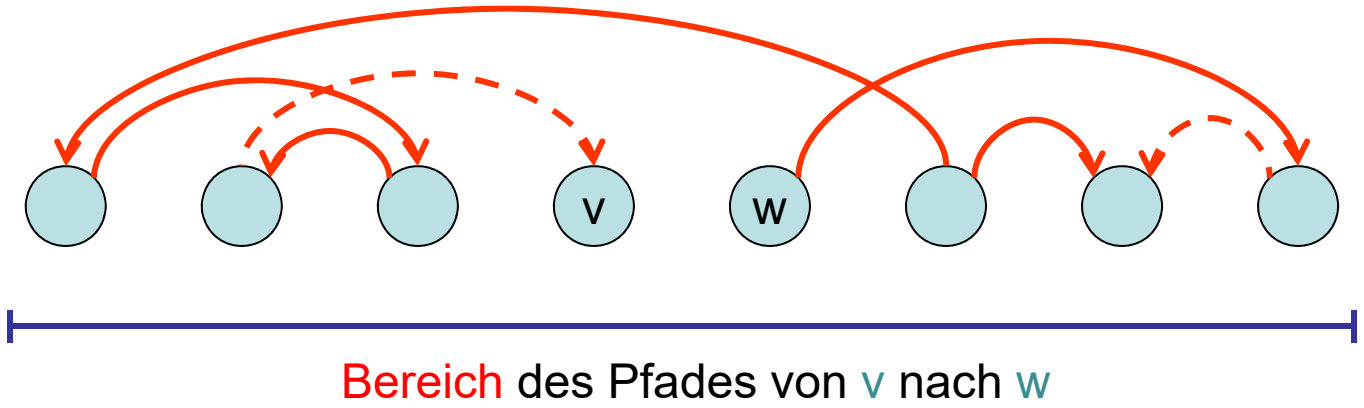
- Aufgrund der Annahme, dass Rechnungen fair sein müssen, wird in endlicher Zeit jeder Knoten timeout ausgeführt haben, so dass es danach keinen Knoten v mehr gibt mit $id(v.left) \geq id(v)$ oder $id(v.right) \leq id(v)$.
- Wir können also ohne Beschränkung der Allgemeinheit im Folgenden annehmen, dass für alle Knoten v $id(v.left)<id(v)$ und $id(v.right)>id(v)$ ist.

Sortierte Liste

Satz 5.2 (Konvergenz): Build-List erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ eine sortierte Liste (sofern E_M nur aus linearize Anfragen besteht).

Beweis:

- Betrachte beliebiges Nachbarpaar v,w bzgl. der sortierten Liste.
- Da G schwach zusammenhängend ist, gibt es einen (nicht notwendigerweise gerichteten) Pfad in G von v nach w.

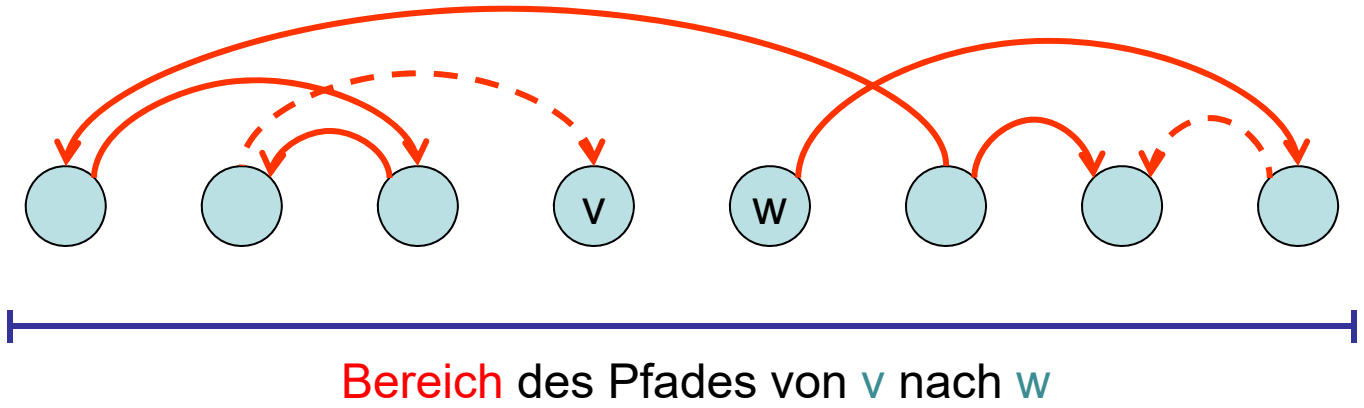


Sortierte Liste

Satz 5.2 (Konvergenz): Build-List erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ eine sortierte Liste (sofern E_M nur aus linearize Anfragen besteht).

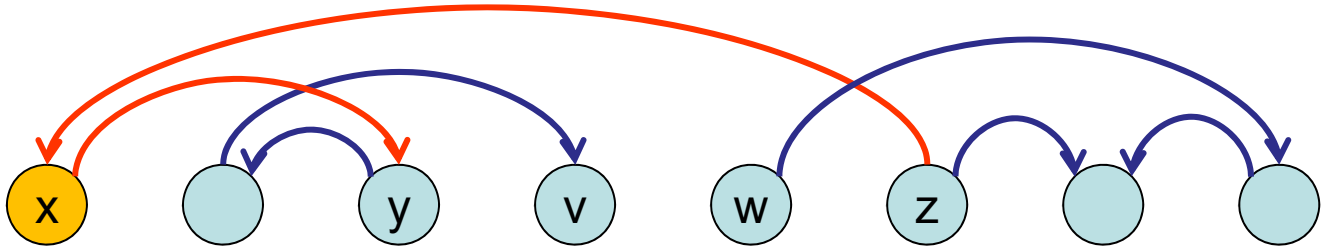
Beweis:

- Wir wollen zeigen, dass sich der Bereich des Pfades von v nach w sukzessive verkleinert. Da G endlich viele Knoten hat, sind dann irgendwann v und w direkt verbunden.

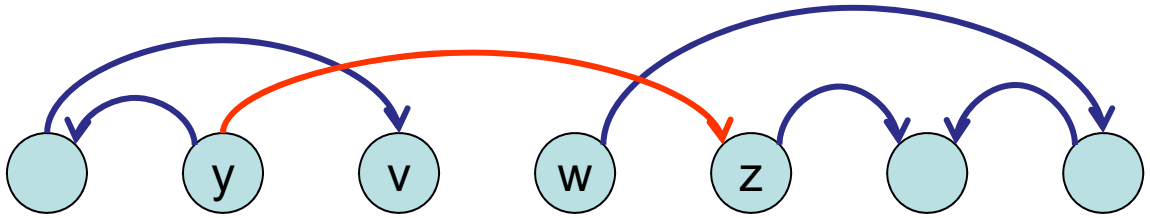


Sortierte Liste

Beweis (Intuition):

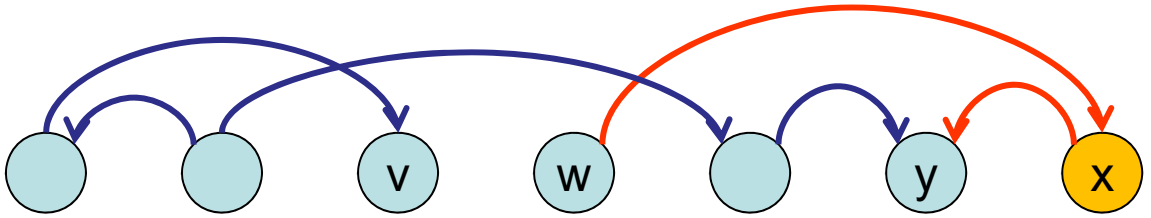


- z führt irgendwann beim timeout() Aufruf x←linearize(z) aus
- x delegiert z weiter an y
- danach kürzerer Bereich für Pfad, da x unnötig:

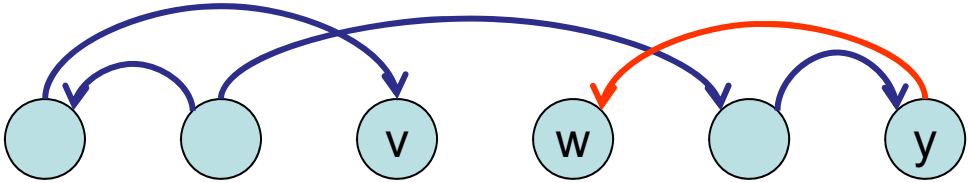


Sortierte Liste

Beweis (Intuition):

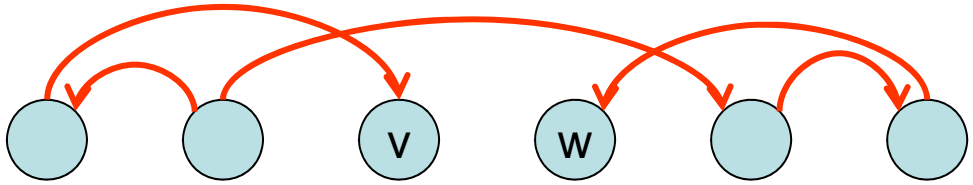


- w führt irgendwann beim timeout() Aufruf x←linearize(w) aus
- x delegiert w weiter an y
- danach kürzerer Bereich für Pfad, da x unnötig:



Sortierte Liste

Beweis (Intuition):



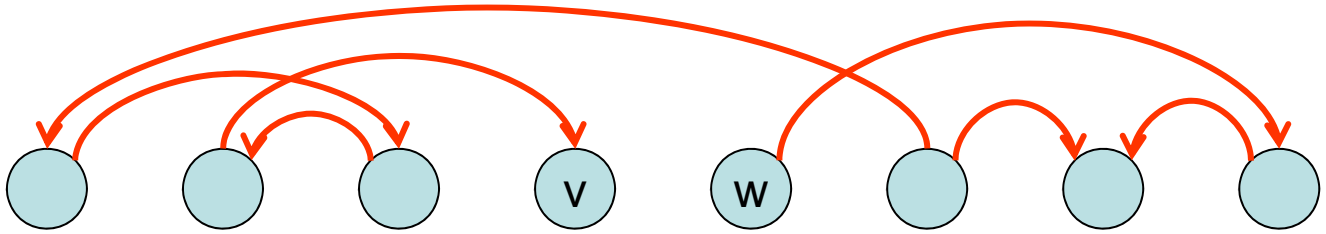
- Randknoten des Pfades können also nach und nach aus dem Pfad ausgeklammert werden, so dass Bereich des Pfades schrumpft und irgendwann v und w direkt (über eine explizite Kante) verbunden sind.

- Gilt das für alle direkten Nachbarn, formen die expliziten Kanten eine sortierte Liste, d.h. wir haben einen legalen Zustand erreicht.

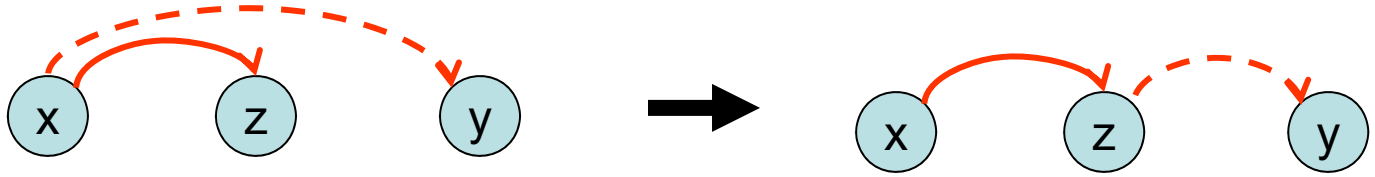
Sortierte Liste

Beweis (formal):

- Betrachte einen Pfad, der v und w miteinander verbindet:



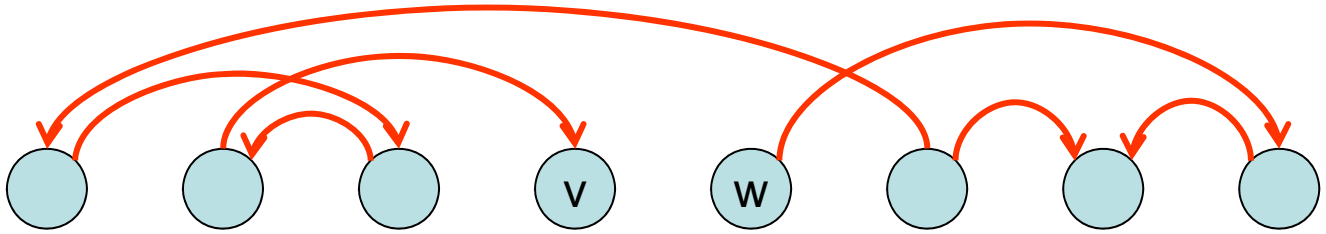
- Dieser Pfad kann so erhalten werden, dass der Bereich, der vom Pfad überdeckt wird, nie anwächst. Dazu reicht es, eine einzelne Kante (x,y) zu betrachten, die Teil des Pfades ist.
- Wird (x,y) durch einen linearize Aufruf aufgelöst, kann diese nur durch zwei Kanten ersetzt werden, die innerhalb des Bereichs von (x,y) verlaufen.
- Fall 1: linearize(y) → ersetze (x,y) durch (x,z),(z,y)



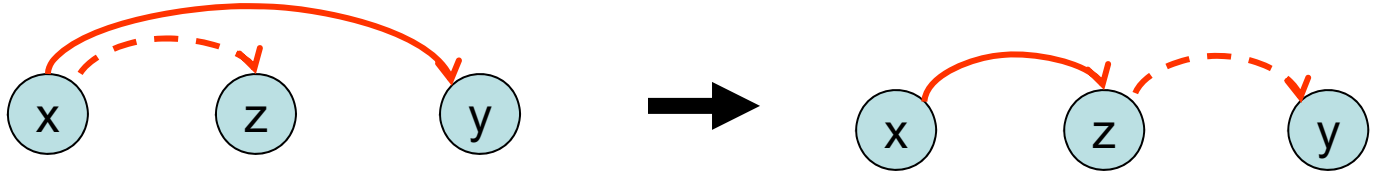
Sortierte Liste

Beweis (formal):

- Betrachte einen Pfad, der v und w miteinander verbindet:



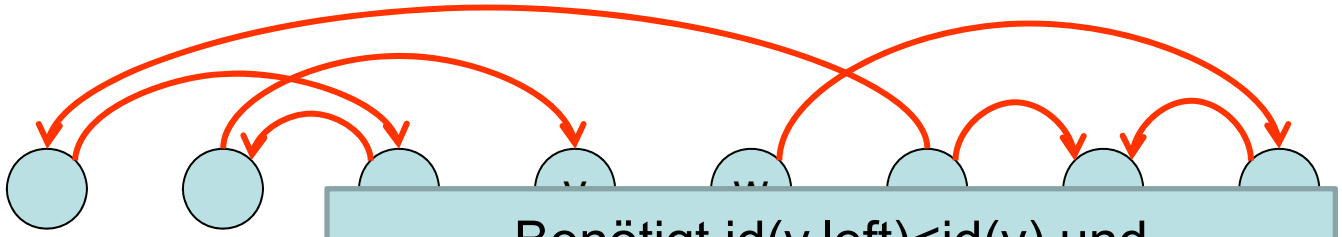
- Dieser Pfad kann so erhalten werden, dass der Bereich, der vom Pfad überdeckt wird, nie anwächst. Dazu reicht es, eine einzelne Kante (x,y) zu betrachten, die Teil des Pfades ist.
- Wird (x,y) durch einen linearize Aufruf aufgelöst, kann diese nur durch zwei Kanten ersetzt werden, die innerhalb des Bereichs von (x,y) verlaufen.
- Fall 2: linearize(z) $\rightarrow$ ersetze (x,y) durch (x,z),(z,y)



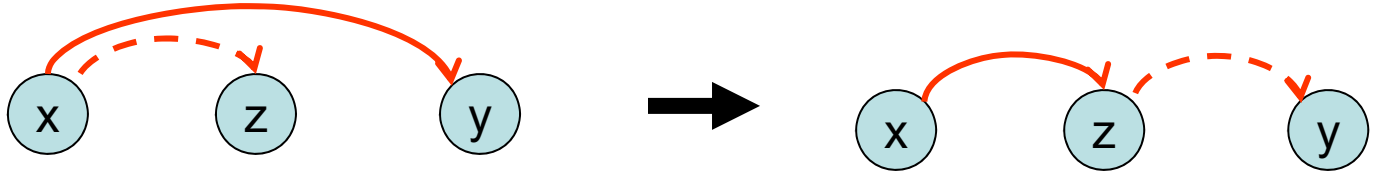
Sortierte Liste

Beweis (formal):

- Betrachte einen Pfad, der v und w miteinander verbindet:



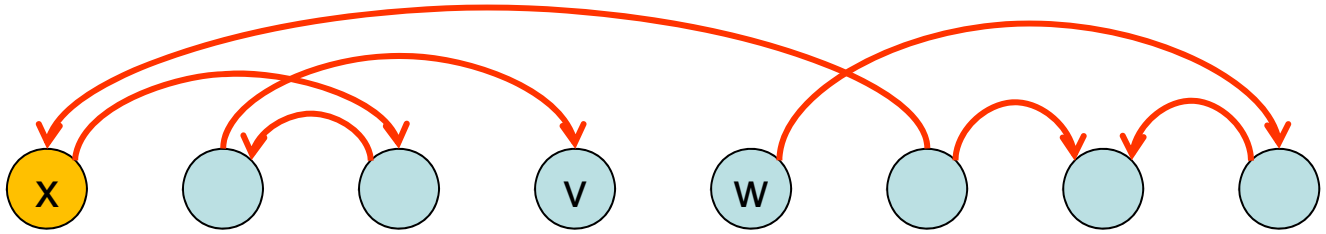
- Dieser Pfad kann so
  ad
  überdeckt wird, nie a
  (x,y) zu
  betrachten, die Teil des Pfades ist.
- Wird (x,y) durch einen linearize Aufruf aufgelöst, kann diese nur durch zwei Kanten ersetzt werden, die innerhalb des Bereichs von (x,y) verlaufen.
- Fall 2: linearize(z) → ersetze (x,y) durch (x,z),(z,y)



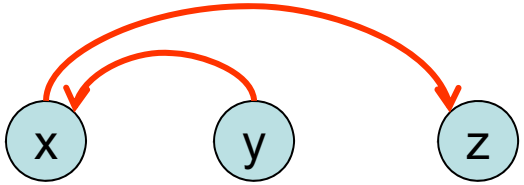
Sortierte Liste

Beweis (formal):

- Betrachte nun einen Randknoten des Pfades, z.B. x:



- Wir müssen alle möglichen Fälle für x betrachten:

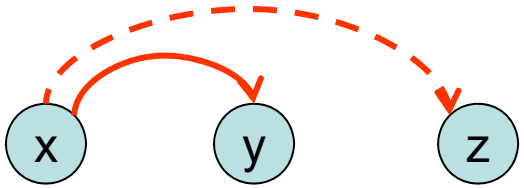


Pfad durch y,x,z

Die zwei Kanten können explizit bzw. implizit sein oder auf x zeigen bzw. von x weg zeigen. O.B.d.A. sei y näher an x als z.

Sortierte Liste

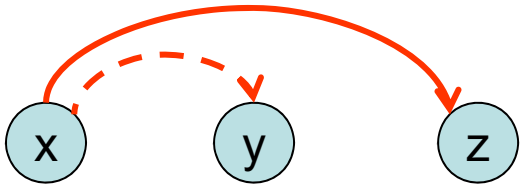
Fall 1a:



- x leitet über y←linearize(z) die Verbindung zu z an y weiter
- damit ist Weg von v nach w verkürzbar von y→x→z auf y→z, d.h. x kann aus dem Weg entfernt werden

Sortierte Liste

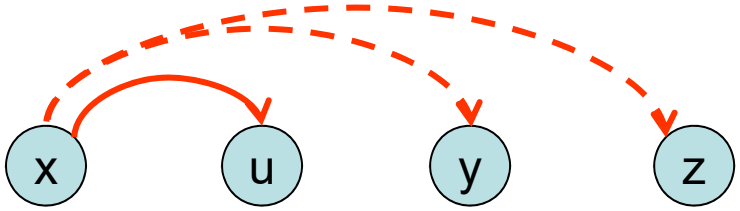
Fall 1b:



- x wandelt bei Aufruf linearize(y) (x,y) in eine explizite Kante (da y näher an x als z ist) und (x,z) in eine implizite Kante um
- damit Reduktion auf Fall 1a

Sortierte Liste

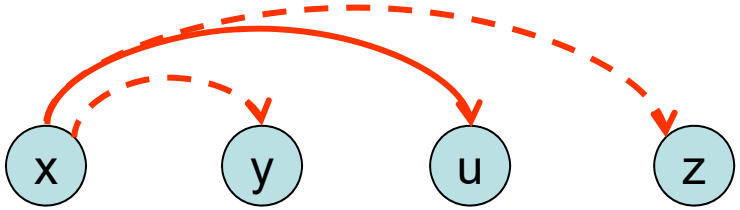
Fall 1c:



- wird zuerst y von x bearbeitet, dann reicht x y an u weiter
- damit können wir den Teilweg y→x→z umwandeln in y→u→x→z, d.h. u wird das neue y, so dass wir Fall 1a erhalten
- wird zuerst z von x bearbeitet, dann reicht x z an u weiter
- damit können wir den Teilweg y→x→z umwandeln in y→x→u→z, d.h. u wird das neue z, so dass wir (bei Vertauschung von y und z, da y immer der nächste Knoten an x sein soll) auch hier Fall 1a erhalten

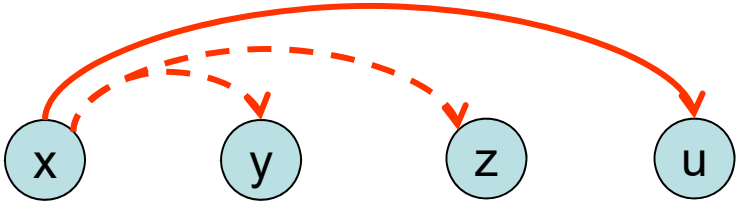
Sortierte Liste

Fall 1d:



- reduziert sich (je nachdem, ob y oder z zuerst bearbeitet wird) auf Fall 1a oder 1b

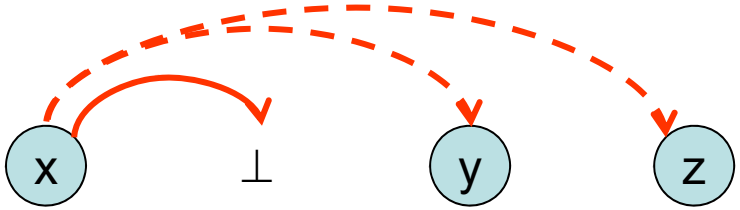
Fall 1e:



- reduziert sich auch (je nachdem, ob y oder z zuerst bearbeitet wird) auf Fall 1a oder 1b

Sortierte Liste

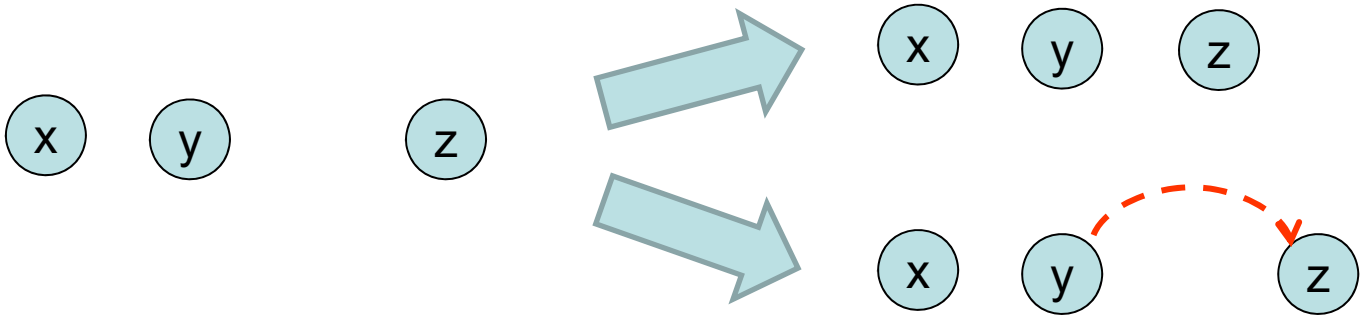
Fall 1f:



- reduziert sich (je nachdem, ob y oder z zuerst bearbeitet wird) auf Fall 1a oder 1b
- Restliche Fälle (z.B. Kanten von y und/oder z nach x orientiert): Übung

Sortierte Liste

- Genauer gesagt können wir nachweisen, dass wir durch Umbenennung von y und z die Knoten y und z immer näher an x heranführen können, bis irgendwann x eine Verbindung zwischen y und z erzeugen muss, was dazu führt, dass x aus dem Pfad herausgenommen werden kann.

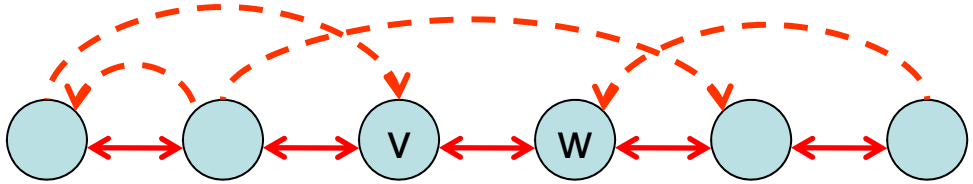


- Damit erhalten wir Satz 5.2.

Sortierte Liste

Satz 5.3 (Abgeschlossenheit): Formen die expliziten Kanten bereits eine sortierte Liste und gilt id(v.left)<id(v)<id(r.right) für alle v, wird diese bei beliebigen timeout und linearize Aufrufen erhalten.

Beweis:



- Eine explizite Kante würde nur bei einem näheren Nachbarn wieder abgebaut werden.
- Formen die expliziten Kanten erst einmal eine sortierte Liste, ist das nicht mehr möglich.

- In der Tat werden dann die impliziten Kanten nur noch weiter-delegiert, bis diese mit einer expliziten Kante verschmelzen.

Sortierte Liste

Bemerkungen zu Satz 5.2:

- Die Gesamtarbeit eines Knotens (gemessen an empfangenen und ausgesendeten linearize Anfragen) kann sehr groß werden, bis die sortierte Liste erreicht ist. Wir haben 2013 ein verbessertes Build-List Protokoll vorgestellt, aber das ist viel komplexer!
- Da eine Listenkante nie wieder verloren geht, wird die sortierte Liste monoton vom Build-List Protokoll aufgebaut.

Bedeutet der letzte Punkt auch, dass Build-List (gemäß Def. 3.7) die Liste monoton stabilisiert?

Dazu müssen wir erstmal die Search-Operation spezifizieren.

Sortierte Liste

Search-Operation:
(Annahme: left=⊥: id(left):=-∞, right=⊥: id(right):=+∞)

Search(sid) →
    if sid=id then „Erfolg“, stop
    if (id(left)<sid<id or id<sid<id(right) then
        „Misserfolg“, stop { garantiert liveness }
    if sid<id then left←Search(sid)
    if sid>id then right←Search(sid)

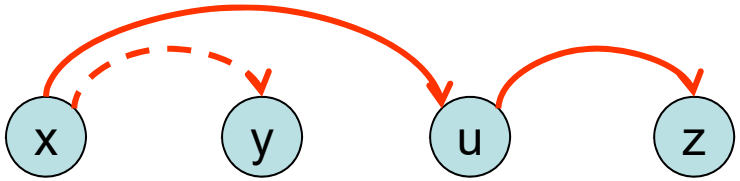
Ziele:

- Liveness: jede Search Anfrage terminiert in endlicher Zeit
- Safety: Monotone Suchbarkeit

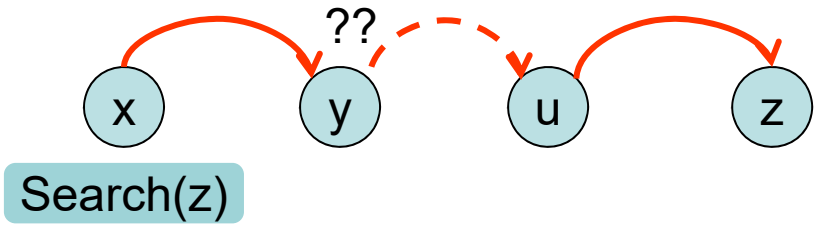
Sortierte Liste

Build-List erfüllt nicht monotone Suchbarkeit.

- Search(z) kann von x nach z gelangen:

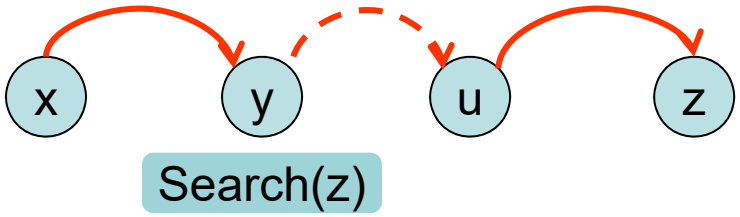


- Nach linearize(y) ist das nicht mehr garantiert:



Sortierte Liste

- Wenn in diesem Fall Search(z) bei y wartet, ist liveness nicht mehr garantiert.

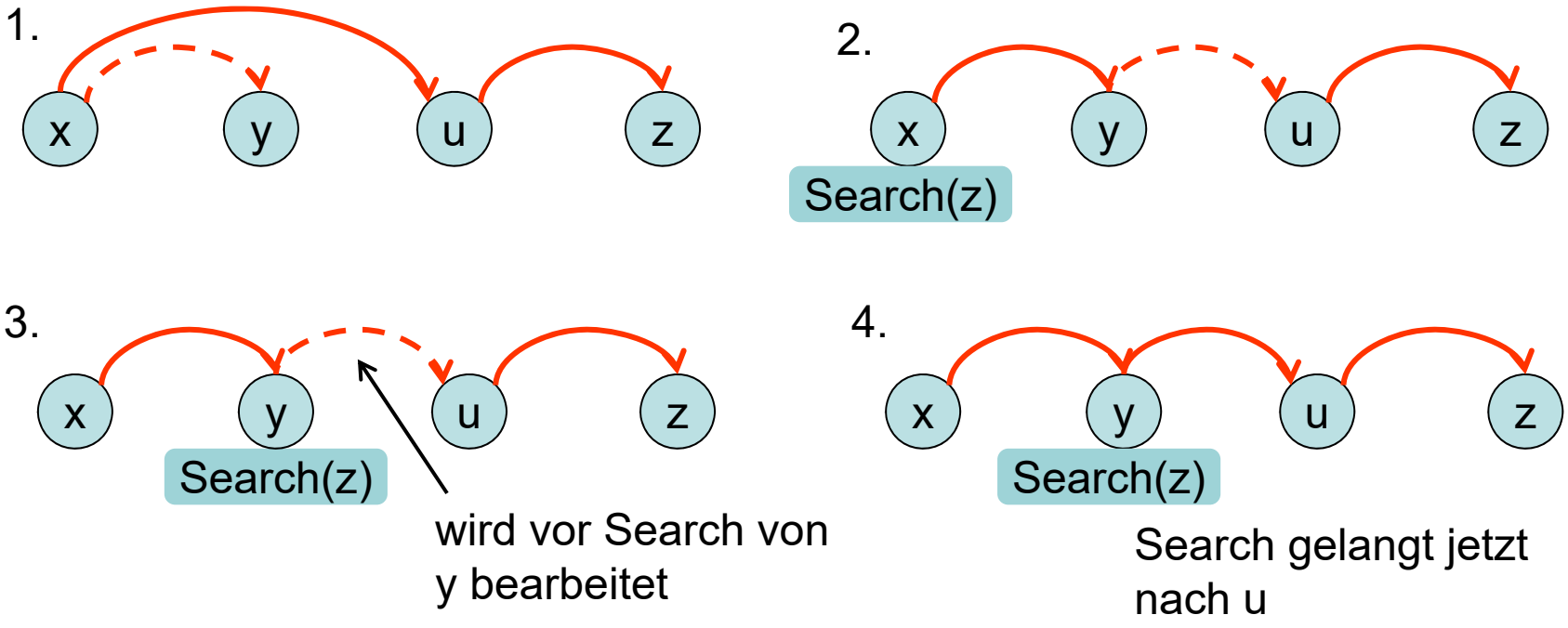


Warum?

- y mag keine Ahnung darüber haben, dass noch ein linearize(u) von x unterwegs ist.
- Selbst wenn y Ahnung davon hätte, könnte diese Information falsch sein (wir betrachten selbststabilisierende Systeme!), d.h. y könnte vergebens warten.

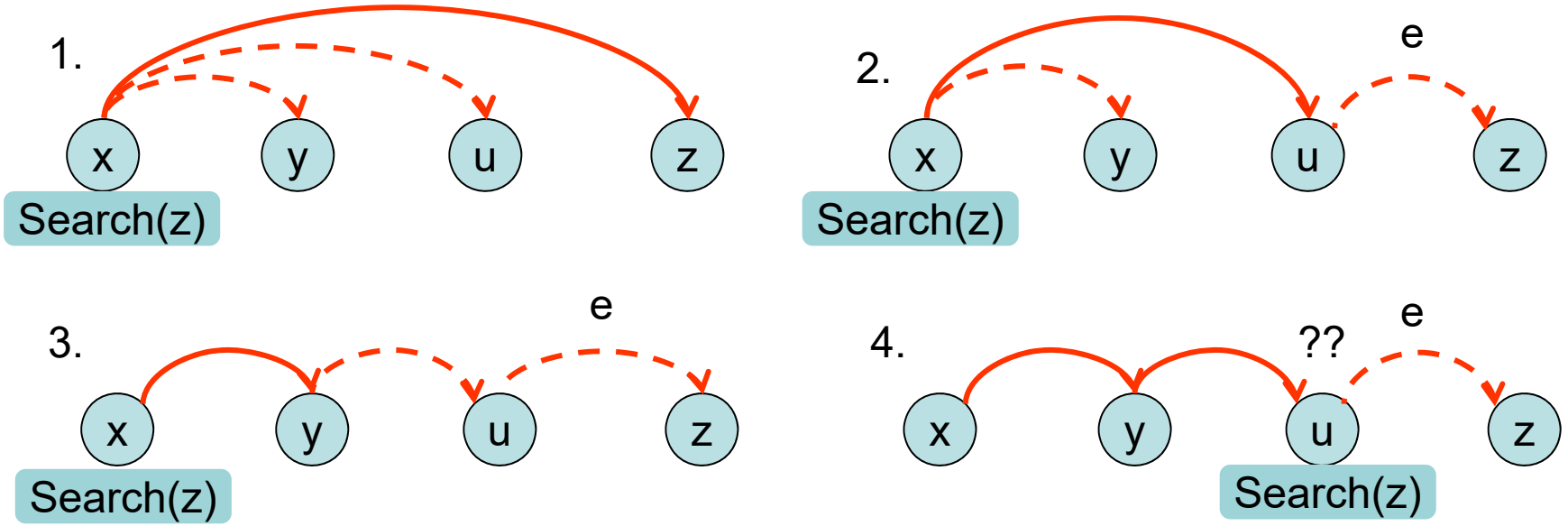
Sortierte Liste

Idee: Vielleicht hilft ja die Annahme, dass für beliebige zwei Konten v,w die Anfragen in FIFO-Ordnung von v nach w geschickt werden?



Sortierte Liste

Aber auch hier gibt es Gegenbeispiel!



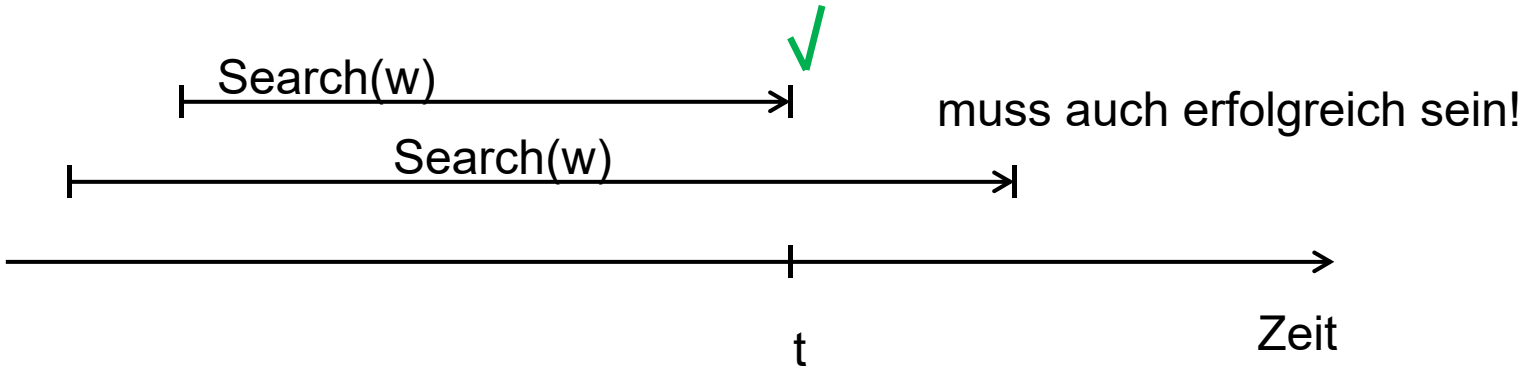
Search(z) kommt von y, aber e von x, d.h. u führt trotz FIFO Annahme eventuell Search(z) vor e aus.

Sortierte Liste

In welcher Form kann monotone Suchbarkeit sichergestellt werden?

1. Falls ein von v initiiertes Search(w) Prozess w zur Zeit t erfolgreich erreicht, dann gilt das auch für alle anderen von v initiierten Search(w) Anfragen, die bis dahin noch nicht w erreicht haben.

Anschaulich:



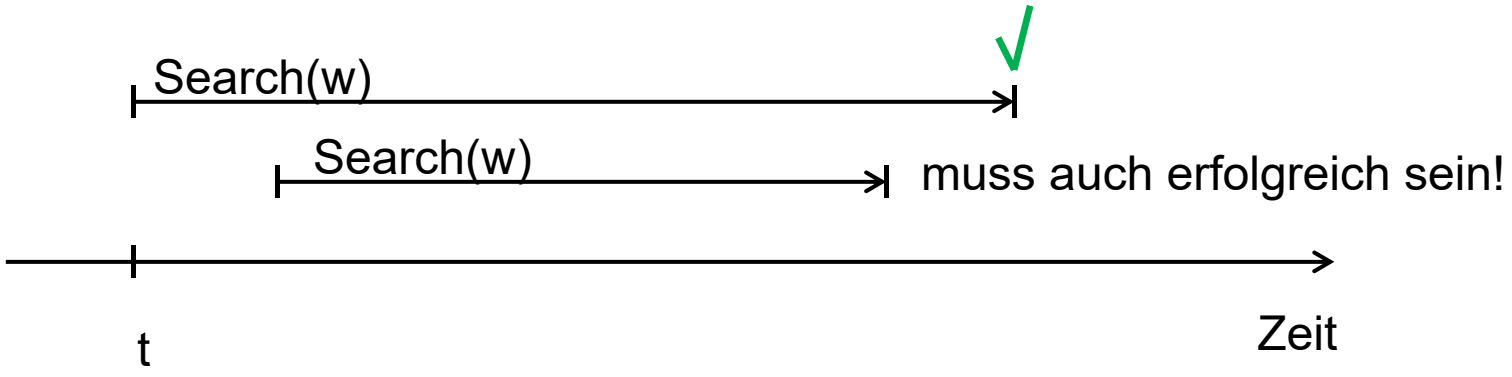
Problem: dafür gibt es Gegenbeispiel! (Übung)

Sortierte Liste

In welcher Form kann monotone Suchbarkeit sichergestellt werden?

2. Falls ein von v zur Zeit t initiiertes Search(w) Prozess w erfolgreich erreicht, dann gilt das auch für alle anderen von v nach dem Zeitpunkt t initiierten Search(w) Anfragen.

Anschaulich:



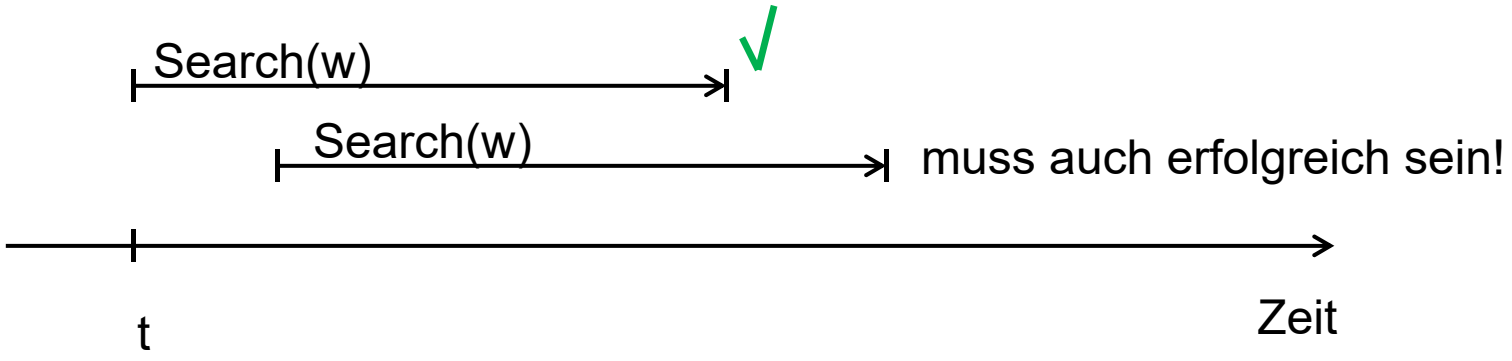
Auch hier gibt es ein Problem!

Sortierte Liste

In welcher Form kann monotone Suchbarkeit sichergestellt werden?

2. Falls ein von v zur Zeit t initiiertes Search(w) Prozess w erfolgreich erreicht, dann gilt das auch für alle anderen von v nach dem Zeitpunkt t initiierten Search(w) Anfragen.

Anschaulich:



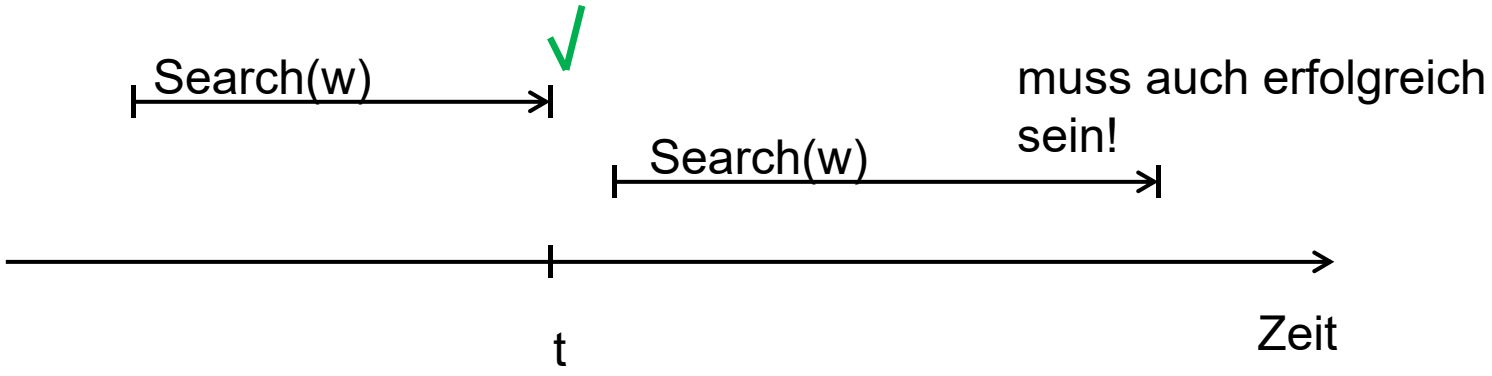
Es muss in diesem Fall FIFO Ordnung erzwungen werden! Aber wie??

Sortierte Liste

In welcher Form kann monotone Suchbarkeit sichergestellt werden?

3. Falls ein von v initiiertes Search(w) Prozess w zur Zeit t erfolgreich erreicht, dann gilt das auch für alle anderen von v nach dem Zeitpunkt t initiierten Search(w) Anfragen.

Anschaulich:

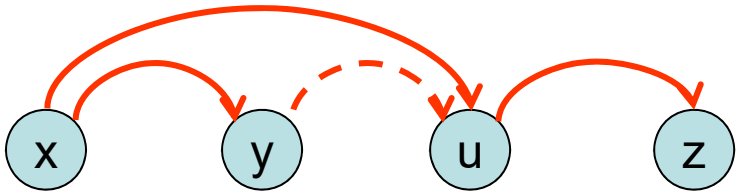


Das ist tatsächlich für die sortierte Liste ohne FIFO Annahme möglich!

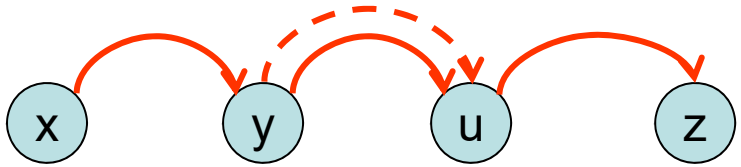
Sortierte Liste

Idee: monotone Erreichbarkeit über explizite Kanten

- statt u zu delegieren, stellt x u dem Knoten y zunächst nur vor:

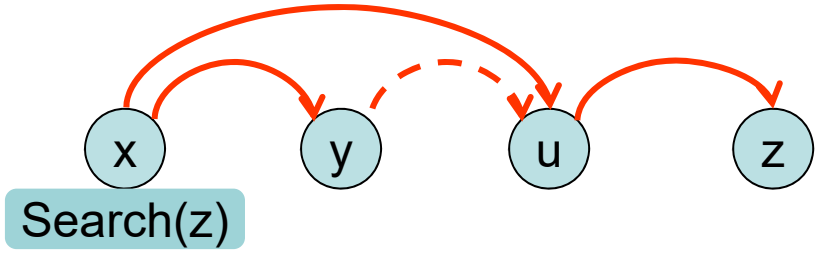


- erst wenn y die Vorstellung an x bestätigt hat, delegiert x u weg nach y



Sortierte Liste

Problem: Welchen Weg soll dann aber Search(z) nehmen, da x jetzt zwei Alternativen hat?

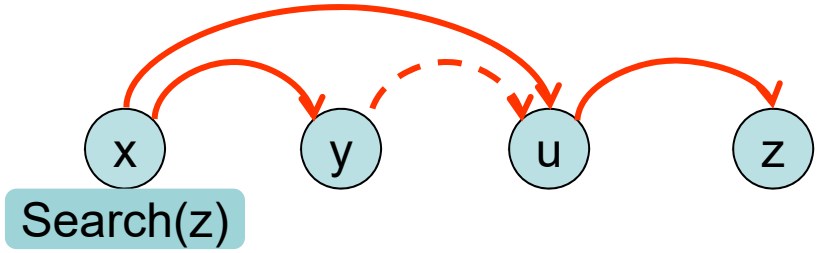


Idee 1: Search(z) wartet solange bei x, bis x nur einen rechten Nachbarn hat. Das wird im Selbststabilisierungsfall irgendwann der Fall sein (keine neuen Knoten kommen hinzu), aber in der Praxis könnte eine Search Anfrage ewig warten.

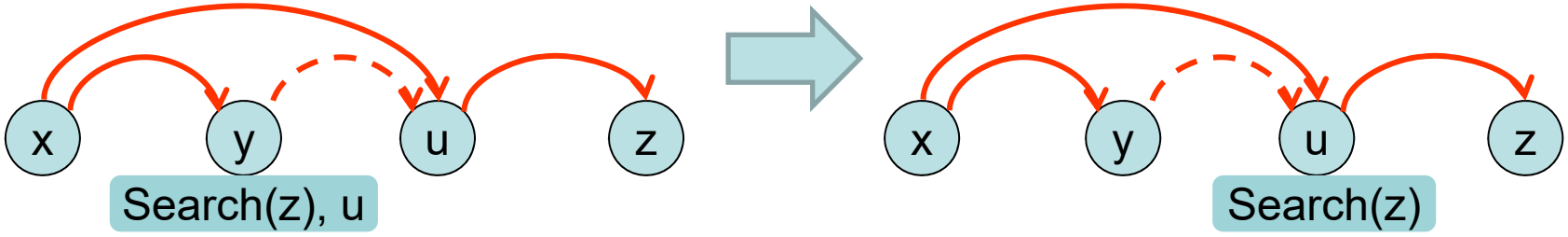
Idee 2: Search(z) wird entlang aller Kanten in Richtung z geschickt. Dann kann die Anzahl der Search(z) Anfragen aber exponentiell über die Zeit anwachsen!

Sortierte Liste

Problem: Welchen Weg soll dann aber Search(z) nehmen, da x jetzt zwei Alternativen hat?



Alternative Idee: Search wird immer zum nächsten Nachbarn weitergeleitet, aber alle anderen rechten Nachbarn werden in der Search Anfrage vermerkt. Dadurch kann dann Search(z) von y nach u gelangen:



Sortierte Liste

Erweitertes Search Protokoll:

- Jeder Knoten v hat Knotenmengen Left und Right
- Jede Search Anfrage hat Menge potenzieller Zielknoten in Next gespeichert.

Vereinfachende Annahmen:

- Für jeden Knoten v gilt, dass v.Left nur Knoten w mit $id(w)<id(v)$ enthält und v.Right nur Knoten w mit $id(w)>id(v)$ enthält (muss sonst in timeout korrigiert werden).
- Für jede Search(sid,Next) Anfrage, die in oder unterwegs zu einem Knoten v ist, muss für alle Knoten w in Next gelten, dass $id(v)\leq id(w)\leq sid$ oder $sid\leq id(w)\leq id(v)$ ist (muss sonst in Search Aktion korrigiert werden).

Übung: wie genau müssten die Korrekturen umgesetzt werden, so dass der schwache Zusammenhang bewahrt bleibt?

Sortierte Liste

Erweitertes Search Protokoll:

- Jeder Knoten v hat Knotenmengen Left und Right
- Jede Search Anfrage hat Menge potenzieller Zielknoten in Next gespeichert.

```
Search(sid, Next) →
    if sid=id then „Erfolg“, stop
    if sid<id then
        Next:=Next∪{w∈Left | id(w)≥sid}
        if Next=∅ then „Misserfolg“, stop
        else
            v:= argmax{ id(w) | w∈Next }
            v←Search(sid,Next\{v}), Left:=Left∪{v}
    else
        Next:=Next∪{w∈Right | id(w)≤sid }
        if Next=∅ then „Misserfolg“, stop
        else
            v:= argmin{ id(w) | w∈Next }
            v←Search(sid,Next\{v}), Right:=Right∪{v}
```

Wichtig für Selbststabilisierung, da v kritisch für den Zusammenhang sein könnte!

Sortierte Liste

Erweitertes Search Protokoll:

- Jeder Knoten v hat Knotenmengen Left und Right
- Jede Search Anfrage hat Menge potenzieller Zielknoten in Next gespeichert.

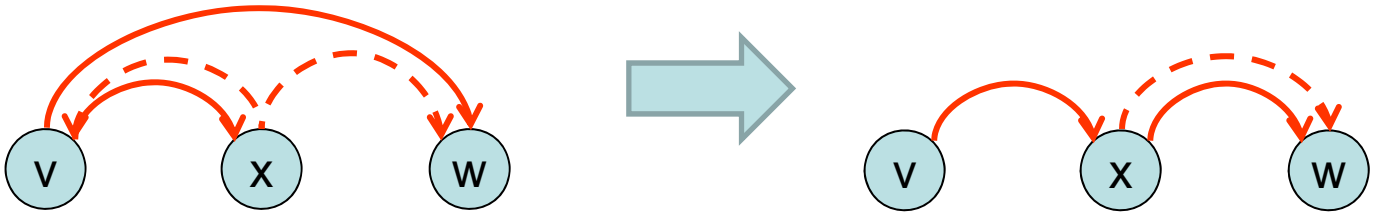
```
Search(sid, Next) →
    if sid=id then „Erfolg“, stop
    if sid<id then
        Next:=Next∪{w∈Left | id(w)≥sid}
        if Next=∅ then „Misserfolg“, stop
        else
            v:= argmax{ id(w) | w∈Next }
            v←Search(sid,Next\{v}), Left:=Left∪{v}
    else
        Next:=Next∪{w∈Right | id(w)≤sid }
        if Next=∅ then „Misserfolg“, stop
        else
            v:= argmin{ id(w) | w∈Next }
            v←Search(sid,Next\{v}), Right:=Right∪{v}
```

Sollte Next korrumpiert sein, dann delegiere alle w in Next mit id(w)∉[sid,id] weg, indem linearize(w) aufgerufen wird.

Sortierte Liste

Angepasstes Build-List Protokoll (Build-List$^+$):

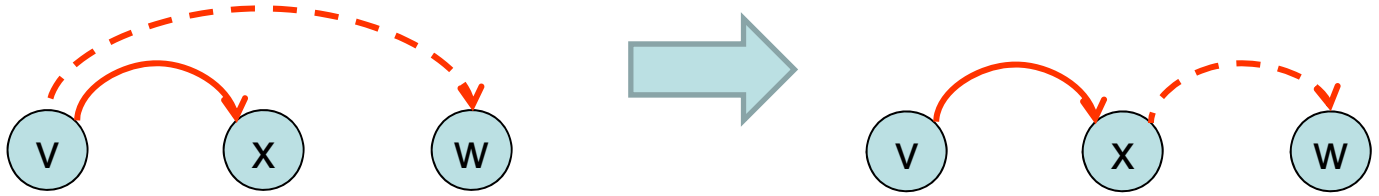
- Jeder Prozess v hat Knotenmengen Left und Right.
- Ist $w \in Left \cup Right$ kein nächster Nachbar von v, dann ruft v in timeout für diesen $x \leftarrow$ introduce(w,v) für einen passenden Prozess $x \in Left \cup Right$ auf.
- Wird in x introduce(w,v) aufgerufen, speichert x die Referenz w in Left bzw. Right ab und meldet via delegate(w) an v, dass v jetzt w wegdelegieren kann.



Sortierte Liste

Angepasstes Build-List Protokoll (Build-List$^+$):

- Jeder Prozess v hat Knotenmengen Left und Right.
- Lernt v einen neuen Prozess w über linearize kennen, behandelt er ihn ähnlich zum alten linearize (d.h. w wird weitergeleitet, falls er nicht der nächste Nachbar ist). Das ist in Ordnung, weil noch keine Anfrage von v nach w aufgrund dieser Vorstellung gelaufen sein kann, d.h. es besteht keine Gefahr auf Verletzung der monotonen Suchbarkeit.

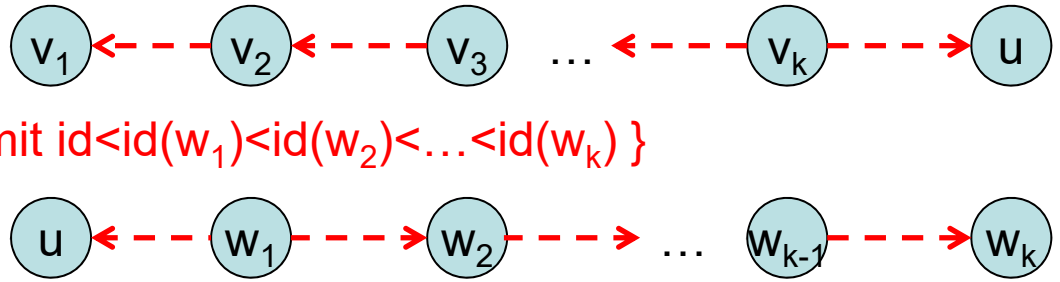


Sortierte Liste

Angepasstes Build-List$^+$ Protokoll:

Überprüfung, ob alle ids in Left < id sind! (Hier weggelassen.)

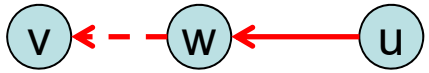
timeout: true →
  { Sei Left = {$v_1, v_2,\ldots,v_k$} mit $id(v_1)<id(v_2)<\ldots<id(v_k)<id$ }
  for all $v_i \in$ Left with $i<k$ do
    $v_{i+1} \leftarrow$ introduce(v_i,this)
  $v_k \leftarrow$ linearize(this)
  { Sei Right = {$w_1, w_2,\ldots,w_k$} mit $id<id(w_1)<id(w_2)<\ldots<id(w_k)$ }
  for all $w_i \in$ Right with $i>1$ do
    $w_{i-1} \leftarrow$ introduce(w_i,this)
  $w_1 \leftarrow$ linearize(this)

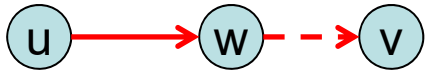


introduce(v,w) →
  if id(v)<id then
    Left:=Left∪{v}
    w←delegate(v) { v kann von w wegdelegiert werden }
  if id(v)>id then
    Right:=Right∪{v}
    w←delegate(v) { v kann von w wegdelegiert werden }

Sortierte Liste

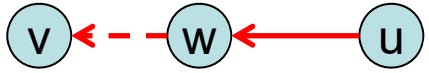
```
delegate(v) →
    if id(v)<id then
        w:= argmax { id(w´) | w´∈Left }
        if w=⊥ or id(w)≤id(v) then Left:=Left∪{v}          { Fehlerfall? }
        else                                                { v ist nicht nächster Nachbar }
           Left:=Left \ {v}
           w:= argmin { id(w´) | w´∈Left und id(w´)>id(v) }
           w←linearize(v)
    if id(v)>id then
        w:= argmin { id(w´) | w´∈Right }
        if w=⊥ or id(w)≥id(v) then Right:=Right∪{v}        { Fehlerfall? }
        else                                                { v ist nicht nächster Nachbar }
            Right:=Right \ {v}
            w:= argmax { id(w´) | w´∈Right und id(w´)<id(v) }
            w←linearize(v)
```

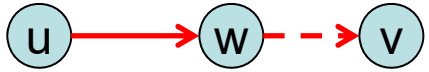




Sortierte Liste

```
linearize(v) →
    if id(v)<id then
        w:= argmin { id(w´) | w´∈Left und id(w´)≥id(v) }
        if w=⊥ then Left:=Left∪{v}      { v ist nächster Nachbar? }
        else
            if w≠v then w←linearize(v)
    if id(v)>id then
        w:= argmax { id(w´) | w´∈Right und id(w´)≤id(v) }
        if w=⊥ then Right:=Right∪{v}  { v ist nächster Nachbar? }
        else
            if w≠v then w←linearize(v)
```





Sortierte Liste

Satz 5.4: Build-List$^+$ garantiert die monotone Suchbarkeit gemäß Fall 3, sofern keine korrumpierten Informationen mehr im System sind.

Was genau bedeutet „korrumpierte Informationen“?

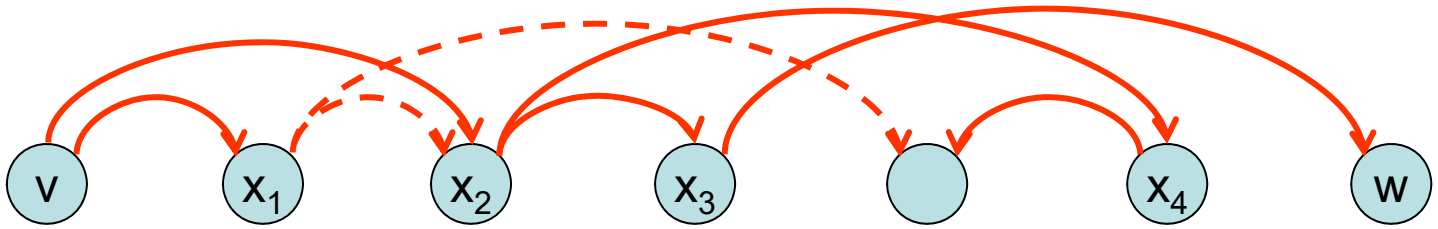
- Für jeden Prozess v sind in v.Left nur Prozesse w mit id(w)<id(v) und in v.Right nur Prozesse w mit id(w)>id(v).
- Jede Anfrage delegate(v) ist durch ein introduce(v,w) in einem Prozess x ausgelöst worden, für den id(w)<id(x)<id(v) oder id(v)<id(x)<id(w) gilt.

Sortierte Liste

Satz 5.4: Build-List$^+$ garantiert die monotone Suchbarkeit gemäß Fall 3, sofern keine korrumpierten Informationen mehr im System sind.

Beweis:

- Betrachte eine beliebige Anfrage Search(w), die in Knoten v gestartet ist.
- O.B.d.A. sei id(v)<id(w). Sei R(v,w) die Menge aller Prozesse x mit id(v)<id(x)≤id(w), für die es einen gerichteten Pfad von v nach x über explizite Kanten (y,z) mit id(y)<id(z) gibt.



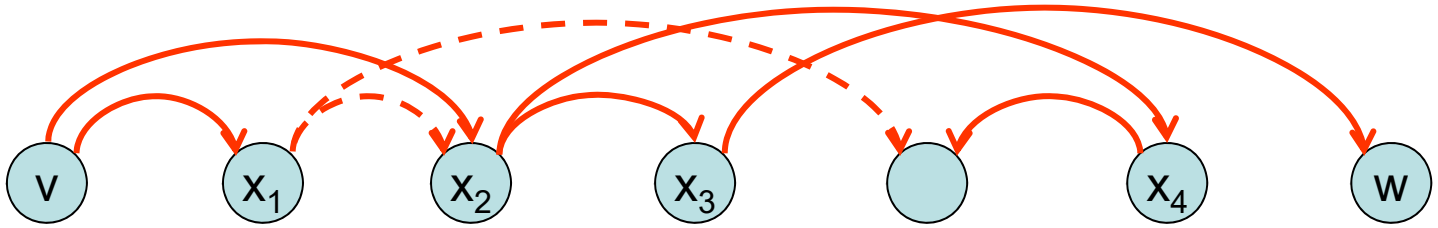
$R(v,w)=\{x_1,x_2,x_3,x_4,w\}$

Sortierte Liste

Satz 5.4: Build-List$^+$ garantiert die monotone Suchbarkeit gemäß Fall 3, sofern keine korrumpierten Informationen mehr im System sind.

Beweis:

- Betrachte eine beliebige Anfrage Search(w), die in Knoten v gestartet ist.
- O.B.d.A. sei id(v)<id(w). Sei R(v,w) die Menge aller Prozesse x mit id(v)<id(x)≤id(w), für die es einen gerichteten Pfad von v nach x über explizite Kanten (y,z) mit id(y)<id(z) gibt.



Allgemein gilt: $R(v,w)=\bigcup_{x \in v.Right:\ id(x) \le id(w)} (\{x\} \cup R(x,w))$

Sortierte Liste

Satz 5.4: Build-List$^+$ garantiert die monotone Suchbarkeit gemäß Fall 3, sofern keine korrumpierten Informationen mehr im System sind.

Beweis:

- Betrachte eine beliebige Anfrage Search(w), die in Knoten v gestartet ist.
- O.B.d.A. sei $id(v)<id(w)$. Sei R(v,w) die Menge aller Prozesse x mit $id(v)<id(x)\leq id(w)$, für die es einen gerichteten Pfad von v nach x über explizite Kanten (y,z) mit $id(y)<id(z)$ gibt.

Wir wollen zeigen:

- R(v,w) wächst monoton über die Zeit.
- Sei $R_0(v,w)$ die Menge R(v,w) zu dem Zeitpunkt, an dem Search(w) von v initiiert worden ist. So gilt zu jedem späteren Zeitpunkt, an dem Search(w) in einem Knoten x ist, dass
  (a) Search(w) alle Knoten $y\in R_0(v,w)$ mit $id(y)<id(x)$ besucht hat und
  (b) $\{ y\in R_0(v,w) \mid id(y)>id(x) \}\subseteq Next\cup(\cup_{y\in Next} R(y,w))$ ist.

Aus diesen beiden Aussagen würde Satz 5.4 folgen. Warum?

Sortierte Liste

Lemma 5.5: R(v,w) wächst monoton über die Zeit.

Beweis:

Induktion über die Aktionen einer beliebigen Rechnung:

- Der einzige kritische Punkt ist, wenn eine explizite Kante (x,y) für einen Prozess x aufgelöst wird, der in R(v,w) ist. D.h. vor diesem Zeitpunkt gilt, dass $x,y \in R(v,w)$.
- Die Auflösung geschieht aber nur, wenn delegate(y) in x aufgerufen wird.
- In diesem Fall muss es aber einen Prozess z gegeben haben, der $x \leftarrow$ delegate(y) in einem von x herbeigeführten introduce(y,x) Aufruf aufgerufen hat, d.h. zu diesem Zeitpunkt galt, dass $z \in R(x,w)$ und $y \in R(z,w)$.
- Nach Induktionsvoraussetzung gilt dann auch direkt vor der Auflösung von (x,y), dass $z \in R(x,w)$ und $y \in R(z,w)$. Weiterhin ist $x \in R(v,w)$.
- Da $id(v)<id(x)<id(z)<id(y)$, muss auch nach Auflösung der Kante (x,y) noch gelten, dass $x \in R(v,w)$, $z \in R(x,w)$ und $y \in R(z,w)$. Damit gibt es aber nach wie vor einen gerichteten Pfad von v (über x und z) nach y, d.h. $y \in R(v,w)$, was die Induktion abschließt.

Sortierte Liste

Lemma 5.6: Sei $R_0(v,w)$ die Menge $R(v,w)$ zu dem Zeitpunkt, an dem Search(w) von v initiiert worden ist. So gilt zu jedem späteren Zeitpunkt, an dem Search(w) in einem Knoten x ist, dass
(a) Search(w) alle Knoten $y \in R_0(v,w)$ mit $id(y)<id(x)$ besucht hat und
(b) $\{ y \in R_0(v,w) \mid id(y)>id(x) \} \subseteq Next \cup (\bigcup_{y \in Next} R(y,w))$ ist.

Beweis:

Induktion über die Aktionen einer beliebigen Rechnung:

- Initial gelten (a) und (b), da $Next=\{x \in v.Right \mid id(x) \leq id(w)\}$ ist.
- Wird Search(w) weitergeleitet, dann an den nächsten Knoten y in Next. Wegen (b) ist danach dann auch (a) erfüllt.
- Für (b) gilt für die neue Menge Next (die wir hier mit Next´ bezeichnen) im nächsten Knoten x´, der besucht wird, dass
$$Next' = (Next \setminus \{x'\}) \cup \{y \in Right(x') \mid id(y) \leq id(w)\}$$
- Damit folgt aufgrund von Lemma 5.5, dass
$$\{ y \in R_0(v,w) \mid id(y)>id(x') \} \subseteq (Next \setminus \{x'\}) \cup (\bigcup_{y \in Next} R(y,w)) \subseteq Next' \cup (\bigcup_{y \in Next'} R(y,w))$$
- D.h. es gilt (b) für x´, was die Induktion abschließt.

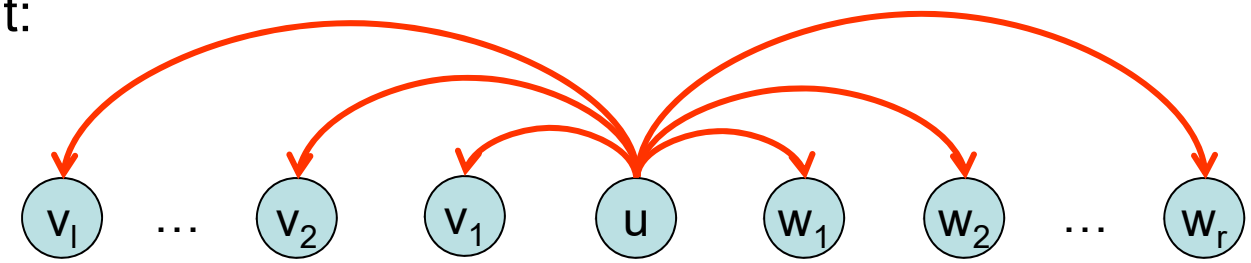
Sortierte Multiliste

Problem: Overhead recht hoch, da |Next| recht groß sein kann und BuildList+ teurer als BuildList ist.

Alternative für monotone Suchbarkeit:

- Eine Kante wird wie vorher explizit, wenn sie zu einem nächsten Nachbarn führt.
- ABER: Eine Kante, die einmal explizit geworden ist, wird nicht mehr wegdelegiert.

Ergebnis: für die Zeitpunkte t(w), zu denen eine Kante zu w aufgebaut wurde, gilt:



$t(v_l) < \ldots < t(v_2) < t(v_1)$ und $t(w_1) > t(w_2) > \ldots > t(w_r)$

Sortierte Multiliste

Angepasstes timeout:

(Annahmen: Left und Right nicht korrumpiert.)

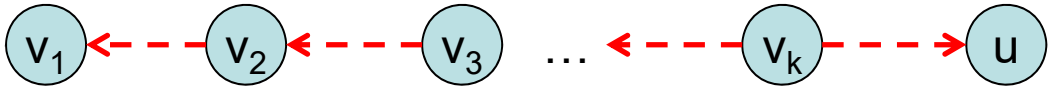
timeout: true →

{ Sei Left = $\{v_1, v_2, \ldots, v_k\}$ mit $id(v_1)<id(v_2)<\ldots<id(v_k)<id$ }

for all $v_i \in$ Left with $i<k$ do

$v_{i+1} \leftarrow$ linearize(v_i)

$v_k \leftarrow$ linearize(this)

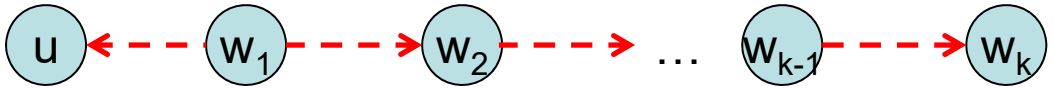


{ Sei Right = $\{w_1, w_2, \ldots, w_k\}$ mit $id<id(w_1)<id(w_2)<\ldots<id(w_k)$ }

for all $w_i \in$ Right with $i>1$ do

$w_{i-1} \leftarrow$ linearize(w_i)

$w_1 \leftarrow$ linearize(this)

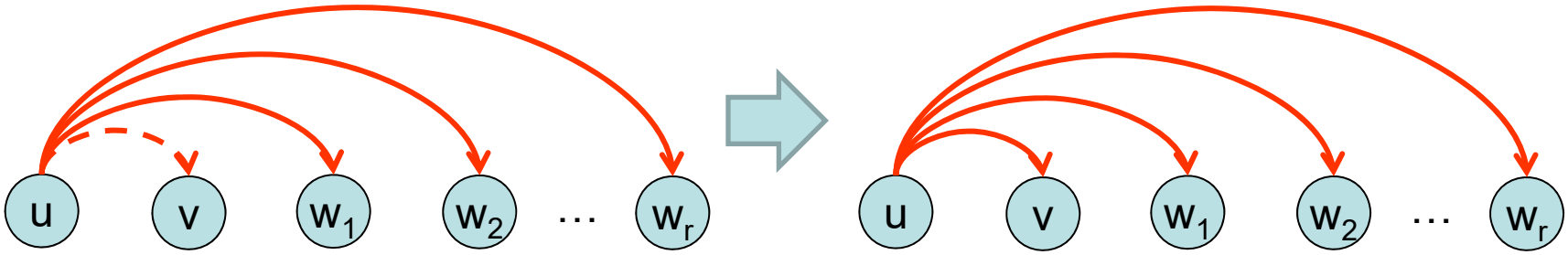


D.h. kein introduce und delegate mehr notwendig.

Sortierte Multiliste

linearize(v) (wie bisher):

Fall 1: v ist näher als alle rechten (bzw. linken) Nachbarn

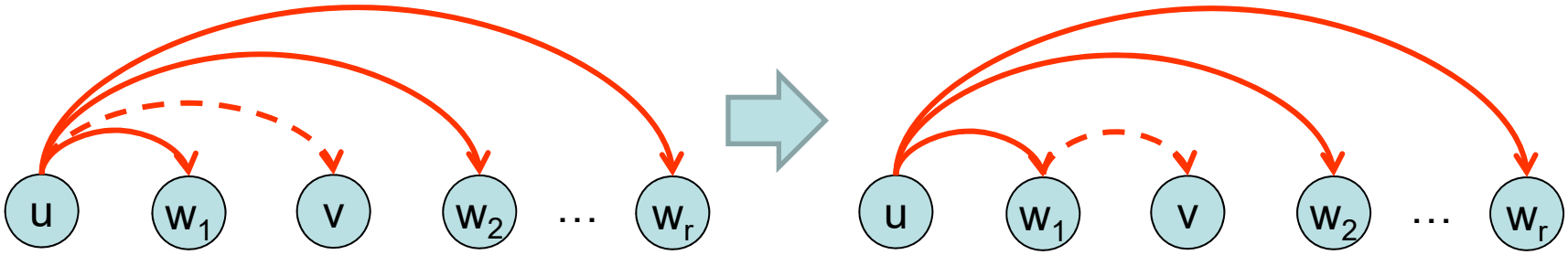


Nimm v als neuen, permanenten Nachbar auf.

Sortierte Multiliste

linearize(v):

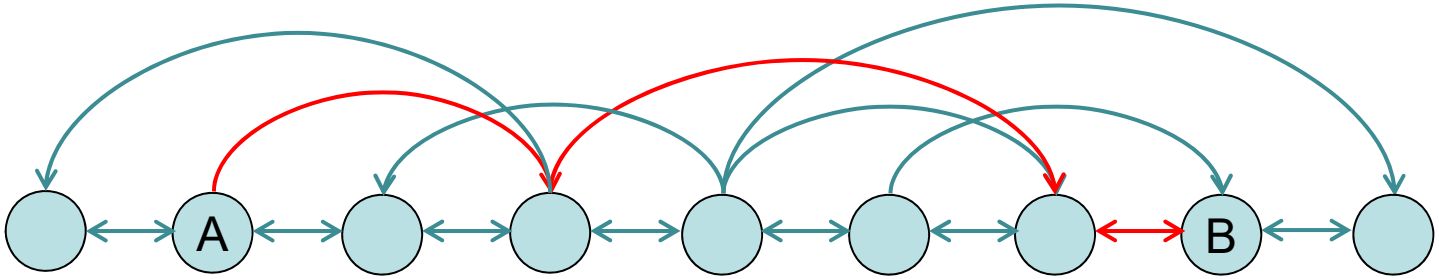
Fall 2: v ist weiter weg als der nächste rechte (bzw. linke) Nachbar



Delegiere v an nächsten Nachbarn zu v vor v (hier w_1).

Sortierte Multiliste

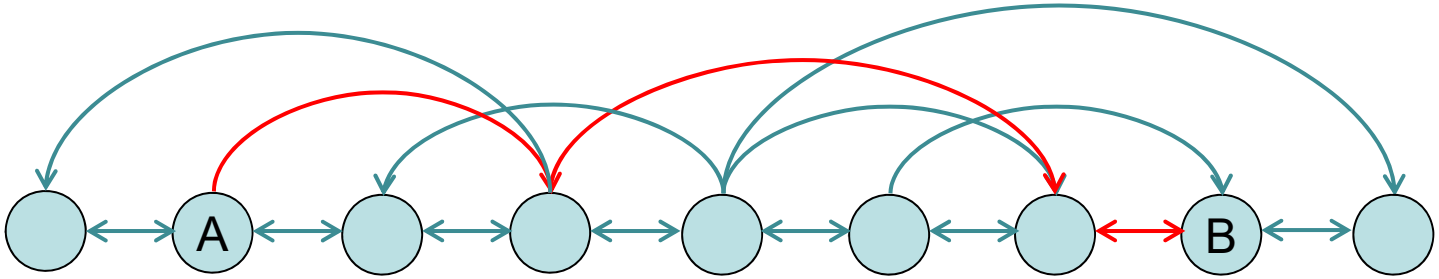
Stabiler Fall: Multiliste



Vorteile der Multiliste:

- Greedy Search Operation (gehe immer zum am nächsten am Ziel liegenden Nachbarn, der vor dem Ziel liegt) reicht, um monotone Suchbarkeit zu erreichen (Beweis: Übung)
- robuster gegenüber Churn (ständigem Wechsel der Knotenmenge), da instabile neue Knoten den Zusammenhang alter Knoten nicht gefährden können

Sortierte Multiliste



Probleme:

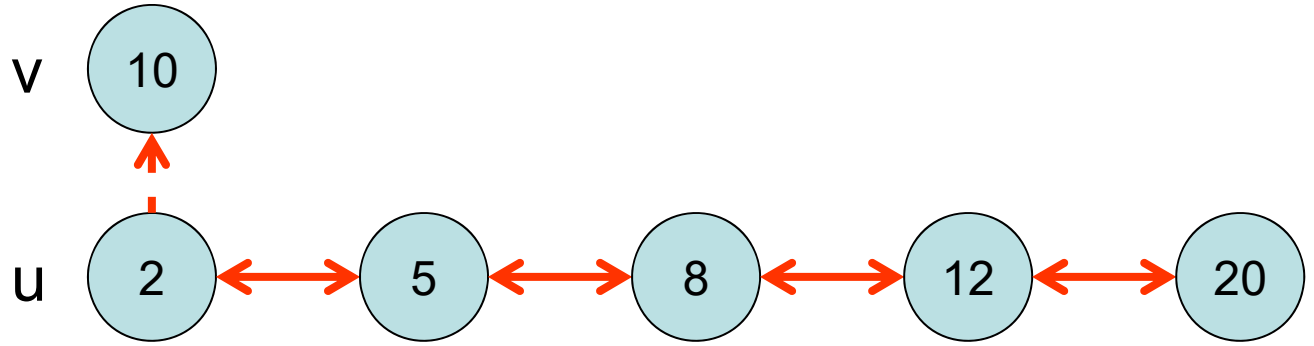
- hoher Grad
- Hohe Kosten im stabilen Fall durch die Weiterleitung der in timeout vorgestellten Kanten

Probleme behebbar, falls jeder neue Knoten zufällige ID hat, da dann z.B. der Grad nur logarithmisch ist, wenn man mit einer Ein-Prozess-Liste startet (evtl. Übung).

Sortierte Liste

Join(v):

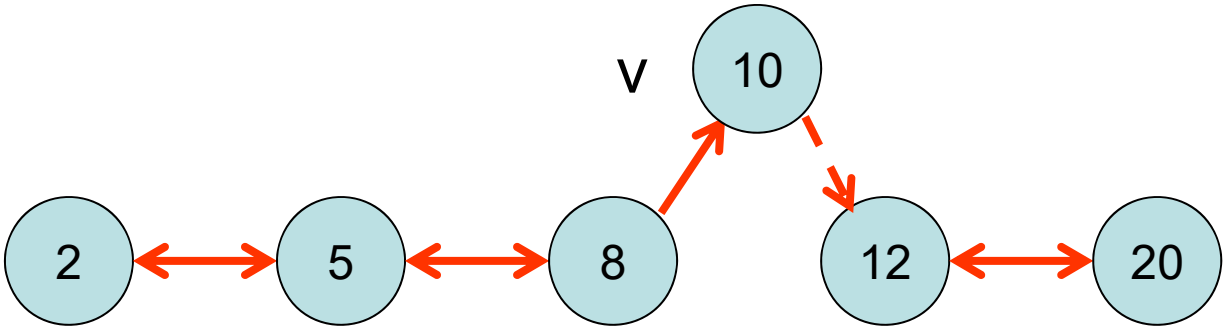
- Angenommen, u bekommt die Anfrage Join(v).
- Dann ruft u einfach u←linearize(v) auf.
- Das Build-List Protokoll wird dann v korrekt in die sortierte Liste einbinden, d.h. Build-List stabilisiert die Join Operation.



Sortierte Liste

Join(v):

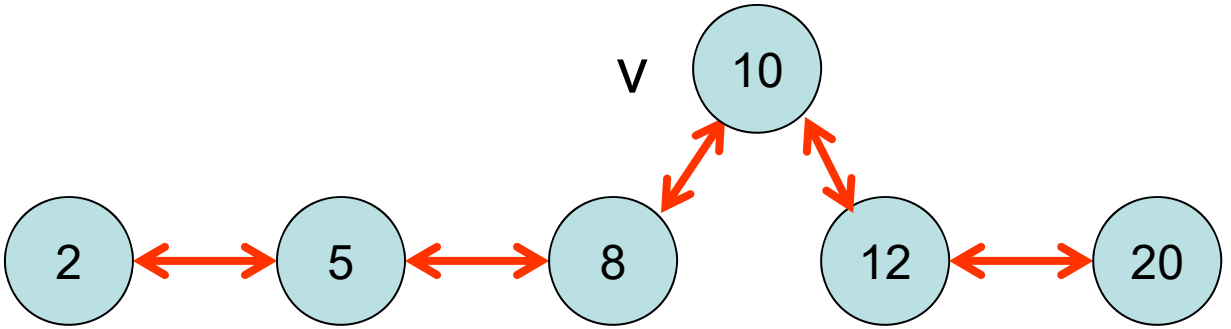
- Angenommen, u bekommt die Anfrage Join(v).
- Dann ruft u einfach u←linearize(v) auf.
- Das Build-List Protokoll wird dann v korrekt in die sortierte Liste einbinden, d.h. Build-List stabilisiert die Join Operation.



Sortierte Liste

Join(v):

- Angenommen, u bekommt die Anfrage Join(v).
- Dann ruft u einfach u←linearize(v) auf.
- Das Build-List Protokoll wird dann v korrekt in die sortierte Liste einbinden, d.h. Build-List stabilisiert die Join Operation.



Sortierte Liste

Wir sagen: Build-List hat die Join(v) Operation für die lineare Liste stabilisiert, wenn pred(v) und succ(v) mit v verbunden sind und v mit pred(v) und succ(v), wobei

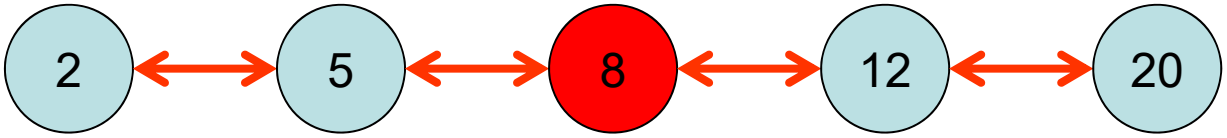
- pred(v): aktueller Vorgänger von v bzgl. IDs
- succ(v): aktueller Nachfolger von v bzgl. IDs

Satz 5.7: Im legalen Zustand (d.h. die sortierte Liste ist bereits erreicht worden) stabilisiert Build-List die Join(v) Operation in maximal O(n) linearize Aufrufen (jenseits der durch timeout erzeugten linearize Aufrufe).

Beweis: Übung

Sortierte Liste

- Leave(v): wir nehmen an, dass Knoten v nur sich selbst aus dem System nehmen kann.



- Idealerweise wird v bei Leave(v) einfach aus dem System genommen und Build-List kümmert sich um die Stabilisierung. Dafür benötigen wir aber eine Topologie mit hoher Expansion!

Leave Problematik

Zentrale Frage: Ist es möglich, ein Leave Protokoll zu entwerfen, so dass Knoten, die das System verlassen wollen, dies tun können, ohne den Zusammenhang zu gefährden sofern keine Fehler auftreten?

Wir führen zwei neue Befehle/Zustände ein:

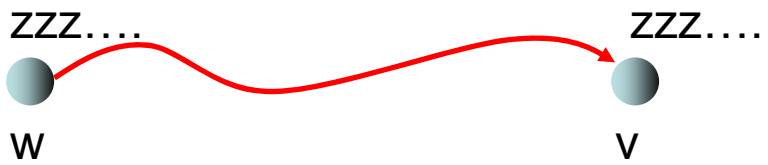
- sleep Befehl/Zustand: Knoten v tut solange nichts, bis er durch eine Anfrage an ihn wieder aufgeweckt wird
- exit Befehl/Zustand: Knoten v will/hat das System endgültig verlassen

Leave(v) Operation:

- Knoten v setzt leaving(v):=true
- Rest soll dann Build-List Protokoll übernehmen

Leave Problematik

- Anfangssituation: beliebiger Systemzustand mit schwachem Zusammenhang, in dem für all die Knoten v, die das System verlassen wollen, leaving(v)=true und für alle bleibenden Knoten leaving(v)=false ist. leaving ist read-only und darf damit nicht verändert werden.
- C_v: Menge der eingehenden Anfragen für Knoten v.
- Ein Knoten ist wach, wenn er weder im Schlaf- noch im Exit-Zustand ist.
- Ein Knoten ist tot, wenn er im Exit-Zustand ist.
- Ein Knoten v ist scheintot, wenn er schläft und $C_v=\emptyset$ ist und für alle Knoten w mit gerichtetem Pfad zu v auch w schläft und $C_w=\emptyset$ ist.



Satz 5.8: Für jeden Algorithmus, in dem alle wachen Knoten in endlicher Zeit eine Nachricht über jede Kante schicken und jeden Systemzustand gilt: ein Knoten ist genau dann permanent am schlafen wenn er scheintot ist.

Beweis: Übung

Leave Problematik

Annahmen über initialen Systemzustand:

- Keine Referenzen nicht existierender Knoten
- Kein Knoten ist initial scheintot oder tot (solche Knoten hätten keinen Nutzen für das Protokoll)
- Schwacher Zusammenhang aller Knoten

Ein Systemzustand ist legal wenn

1. jeder bleibende Knoten wach ist,
2. jeder verlassende Knoten scheintot oder tot ist, und
3. alle bleibenden Knoten eine schwache Zusammenhangskomponente bilden.

Leave Problematik

Ein Systemzustand ist legal wenn

1. jeder bleibende Knoten wach ist,
2. jeder verlassende Knoten scheintot oder tot ist, und
3. alle bleibenden Knoten eine schwache Zusammenhangskomponente bilden.

Probleme:

- FDP (Finite Departure) Problem:
  Erreiche in endlicher Zeit einen legalen Zustand für den Fall, dass nur der exit-Befehl verfügbar ist.

- FSP (Finite Sleep) Problem:
  Erreiche in endlicher Zeit einen legalen Zustand für den Fall, dass nur der sleep-Befehl verfügbar ist.

- Unser Ziel: finde selbststabilisierendes Protokoll für das FDP bzw. FSP Problem, d.h. ein legaler Zustand ist von jedem Ausgangszustand (gem. unserer Annahmen) erreichbar und die Abgeschlossenheit gilt.

Leave Problematik

Unser Ziel: finde selbststabilisierendes Protokoll für das FDP bzw. FSP Problem

Wir werden zeigen:

Satz 5.9: Es gibt kein selbststabilisierendes Protokoll für das FDP Problem.

Satz 5.10: Es gibt ein selbststabilisierendes Protokoll für das FSP Problem.

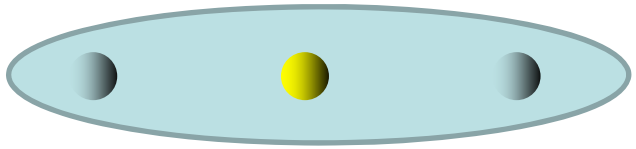
Konsequenz: Es ist unmöglich lokal zu entscheiden, wann es sicher ist, das System endgültig zu verlassen. Erfordern wir aber keine Entscheidung sondern die Knoten sollen lediglich irgendwann permanent schlafen, ist es möglich, aus dem System in endlicher Zeit ausgeschlossen zu werden.

Leave Problematik

Satz 5.9: Es gibt kein selbststabilisierendes Protokoll für das FDP Problem.

Beweis:

- Angenommen, es gäbe ein selbststabilisierendes Protokoll P, das das FDP Problem lösen kann.
- Betrachte folgenden Anfangszustand S_0:



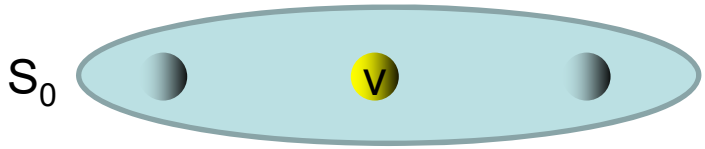
bel. schwach zusammenhängend

: leaving(v)=true

: im exit Zustand

Leave Problematik

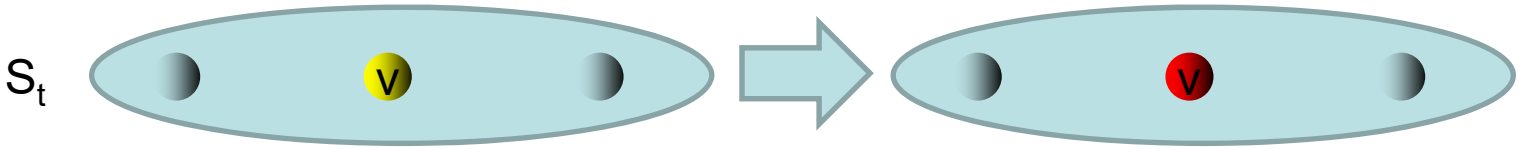
Protokoll P:



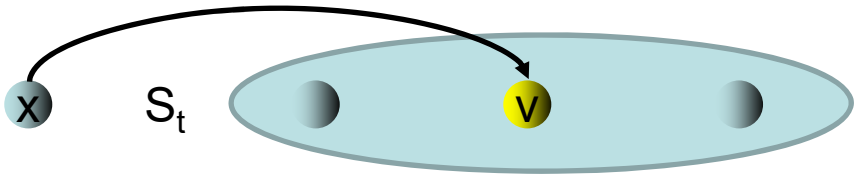
: leaving(v)=true

: im exit Zustand

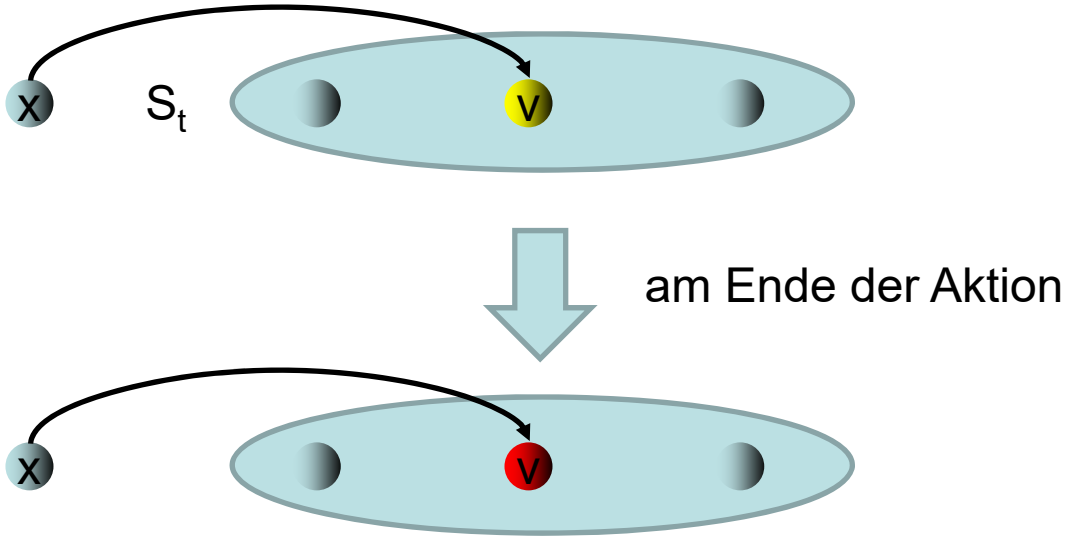
irgendwann der erste Zeitpunkt t, so dass
am Ende von t Knoten v im exit Zustand



Betrachte nun folgenden Anfangszustand:



Leave Problematik



Problem: v wird trotzdem das System verlassen, da er denselben lokalen Zustand wie vorher hat und damit nach wie vor dieselbe Aktion wie vorher ausführen kann. Geht v aber in den Exit-Zustand, dann wäre x isoliert, d.h. das Protokoll würde das FDP Problem nicht lösen. Widerspruch!

Leave Problematik

Um das FDP Problem zu lösen, benötigen wir Orakel.

- Orakel: Prädikat über dem Systemzustand, das von Knoten v abgefragt werden kann.
- Beispiel: O(v)=wahr $\Leftrightarrow$ G ist ohne v noch schwach zusammenhängend.

Weitere Orakel:

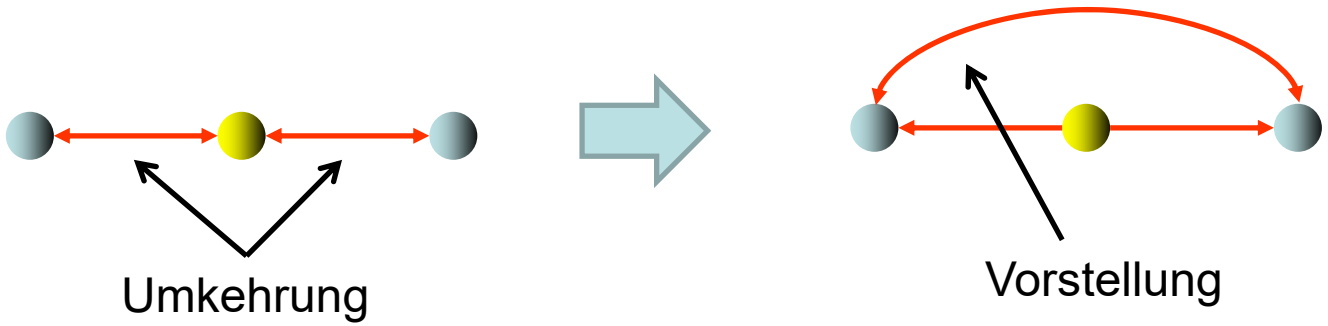
- NID(v)=wahr $\Leftrightarrow$ G hat keine eingehende (implizite oder explizite) Kante von einem Knoten zu v, der nicht tot oder scheintot ist (NID: no ID)
- EC(v)=wahr $\Leftrightarrow$ $C_v=\emptyset$ (EC: empty channel)
- NIDEC(v)=wahr $\Leftrightarrow$ NID(v)=wahr und EC(v)=wahr
- ONESID(v)=wahr $\Leftrightarrow$ v hat Kanten mit höchstens einem Knoten in G, der nicht tot oder scheintot ist

Leave Problematik

Idee:

- Verwende dieselben Variablen wie für die Liste (d.h. ein Knoten hat maximal zwei explizite Nachbarn)
- Versuche, über das Umkehrungsprimitiv Kanten zu verlassenden Knoten so umzulegen, dass diese nicht mehr von bleibenden Knoten erreicht werden können.

Beispiel für die Liste:



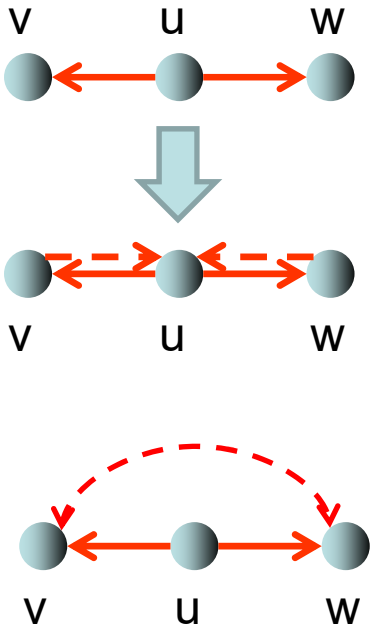
FDP-Protokoll

Vereinfachende Annahmen:

- Jedesmal wenn left=$\perp$ ist, nehmen wir für Vergleiche an, dass id(left)=-∞ ist.
- Jedesmal wenn right=$\perp$ ist, nehmen wir für Vergleiche an, dass id(right)=+∞ ist.
- Ein Aufruf u$\leftarrow$action(v) findet nur dann statt, wenn u und v nicht leer sind.
- Wir kontrollieren nicht in timeout und linearize, ob die linken und rechten Nachbarn korrekt sind (d.h. ob id(left)<id und id(right)>id), d.h. wir nehmen vereinfachend an, dass diese richtig gesetzt sind.

FDP-Protokoll

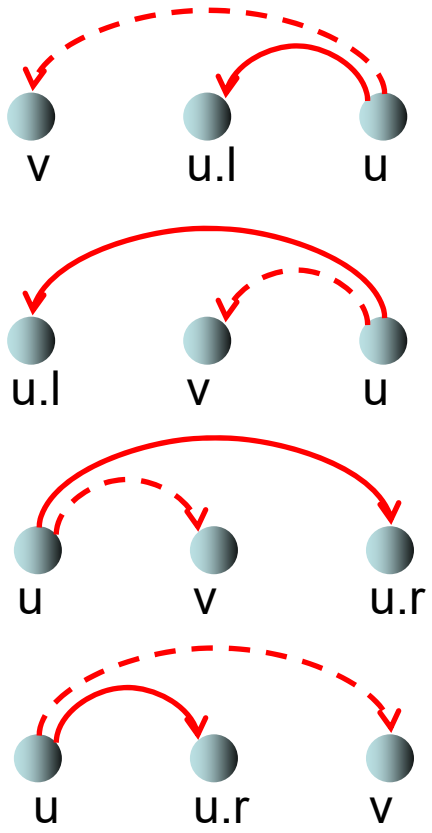
```
timeout: true →
   { ausgeführt in Knoten u }
   if not leaving then
      left←linearize(this)
      right←linearize(this)
   else  { leaving }
       if NIDEC then
          left←linearize(right)
          right←linearize(left)
          exit
       else
          left←reverse(revright)
          right←reverse(revleft)
```



Kantenumkehrung anfordern

FDP-Protokoll

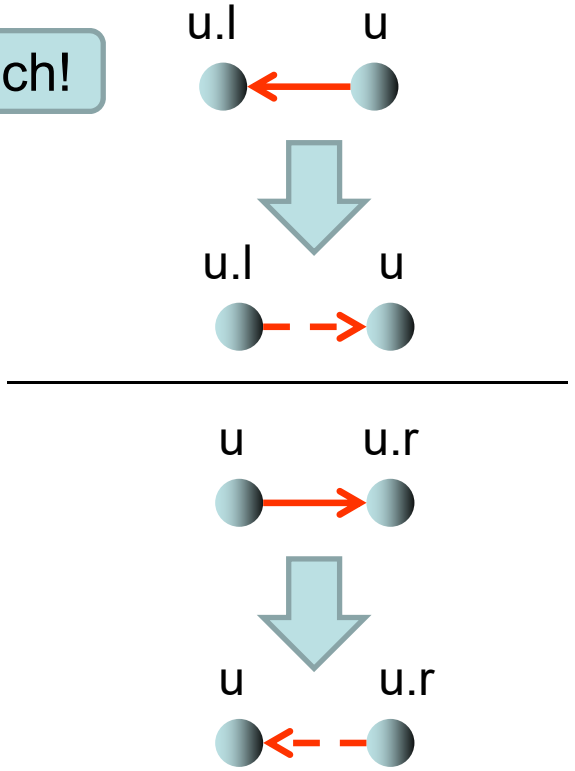
```
linearize(v) →
   { ausgeführt in Knoten u }
   if id(v)<id(left) then
      u.l←linearize(v)
   if id(left)<id(v)<id then
      v←linearize(left)
      left:=v
   if id<id(v)<id(right) then
      v←linearize(right)
      right:=v
   if id(right)<id(v) then
      right←linearize(v)
```



FDP-Protokoll

```
reverse(dir∈{revleft,revright}) →
   { ausgeführt in Knoten u }
   if dir=revleft then
      if not leaving then
         left←linearize(this)
         left:=⊥
   else { revright }
      right←linearize(this)
      right:=⊥
```

Symmetriebruch!

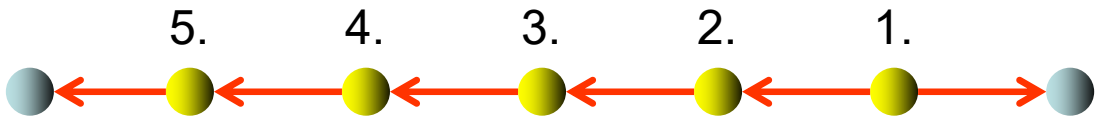


FDP-Protokoll

Satz 5.11: Das FDP-Protokoll mit dem NIDEC Orakel ist eine selbststabilisierende Lösung für das FDP-Problem.

Beweisidee:

- Es bilden sich irgendwann stabile Ketten verlassender Knoten (d.h. die Kanten werden nicht mehr weiterdelegiert)



- Verlassende Knoten können dann in der Reihenfolge der Nummerierung das System verlassen

- Die verbleibenden Knoten werden danach in endlicher Zeit eine sortierte Liste bilden

FDP-Protokoll

Gilt denn mit dem NIDEC-Orakel noch der folgende Satz?

Satz 3.3: Innerhalb unseres Prozess- und Netzwerkmodells kann jede lokal atomare Aktionsausführung in eine äquivalente global atomare Aktionsausführung transformiert werden.

Ein Orakel O heißt persistent, falls solange O(v) wahr ist, die Aktionen anderer Knoten die Ausgabe von O(v) nicht verändern können.

- Man kann zeigen: Für das FDP-Problem gilt Satz 3.3 solange ein persistentes Orakel verwendet wird, mit dem das FDP-Problem für global atomare Aktionsausführungen gelöst werden kann.
- NIDEC ist persistent. ONESID ist allerdings nicht persistent.

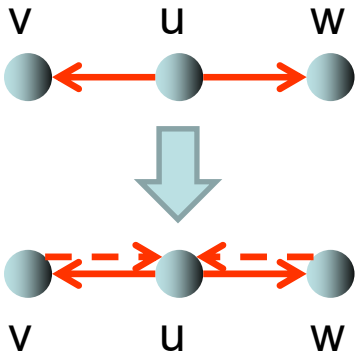
FSP-Protokoll

Satz 5.10: Es gibt ein selbststabilisierendes Protokoll für das FSP Problem.

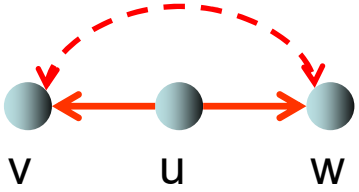
FSP-Protokoll: sehr ähnlich zum FDP-Protokoll, nur dass kein NIDEC verwendet wird und statt exit der sleep Befehl verwendet wird (d.h. nur timeout ist anders).

FSP-Protokoll

```
timeout: true →
    { ausgeführt in Knoten u }
    if not leaving then
        left←linearize(this)
        right←linearize(this)
    else  { leaving }
        left←reverse(revright)
        right←reverse(revleft)
        left←linearize(right)
        right←linearize(left)
        sleep
```



Kantenumkehrung anfordern und



Mögliche Verbesserung: Zusammenfassung von reverse und linearize in einem Aktionsaufruf im else Fall.

FSP-Protokoll

Beweis von Satz 5.10:

Für einen sich schlafen legenden Knoten v gilt: v ist scheintot genau dann wenn NIDEC(v) wahr ist.

„⇐“: Wenn NIDEC(v) wahr ist, dann kann kein Knoten v etwas schicken und es gibt keine weitere Anfrage für v. v ist daher nach Definition scheintot.

„⇒“: Ist v scheintot, dann gibt es keinen gerichteten Pfad von einem nicht toten oder scheintoten Knoten zu v und keine eingehende Anfrage für v, was bedeutet, dass NIDEC(v) wahr ist.

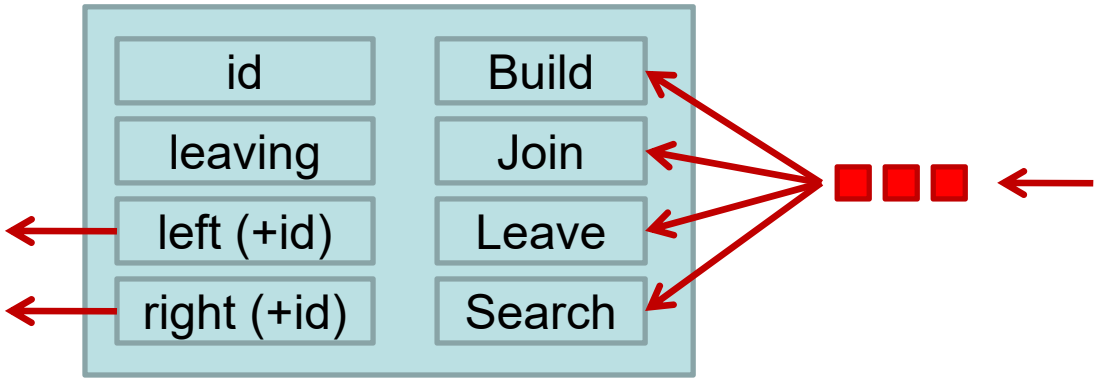
- Das FSP-Protokoll stellt sicher, dass alle nicht scheintoten Knoten irgendwann wieder aufwachen. (Beweis ist Übung)
- Weiterhin stellt das FSP-Protokoll sicher, dass ein Knoten, der sich schlafen legt, nicht dazu führen kann, dass ein anderer Knoten scheintot wird. (Beweis ist Übung)
- Also arbeitet das FSP-Protokoll wie das FDP-Protokoll, nur mit dem Unterschied, dass die verlassenden Knoten am Ende nicht tot sondern scheintot sind.

Sortierte Liste

Variablen innerhalb eines Knotens v:

- id: eindeutiger Name von v (wir schreiben auch id(v))
- leaving$\in${true,false}: zeigt an, ob v System verlassen will
- left $\in V \cup \{\perp\}$: linker Nachbar von v, d.h. id(left)<id(v) (falls left definiert ist)
- right $\in V \cup \{\perp\}$: rechter Nachbar von v, d.h. id(right)>id(v) (falls right definiert ist)

Wir nehmen jetzt an, dass v.id unabhängig zur Referenz von v gewählt ist, d.h. Info über id(left) bzw. id(right) kann falsch sein.

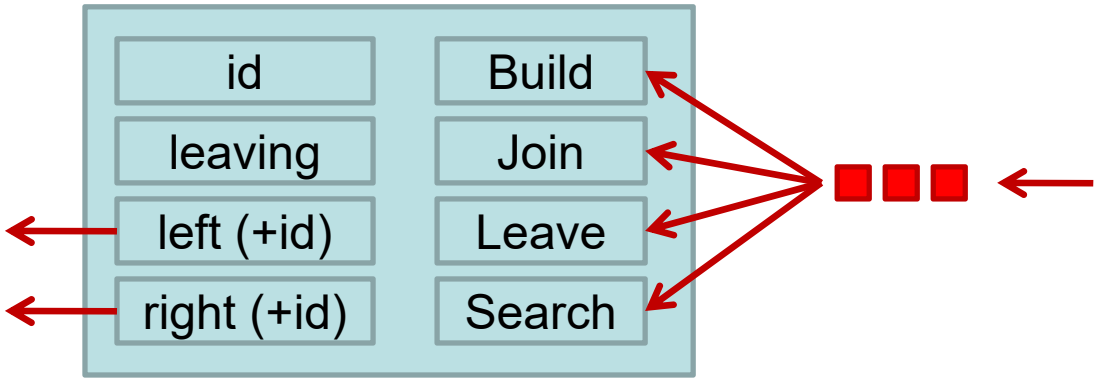


Sortierte Liste

Variablen innerhalb eines Knotens v:

- id: eindeutiger Name von v (wir schreiben auch id(v))
- leaving$\in${true,false}: zeigt an, ob v System verlassen will
- left $\in V \cup \{\perp\}$: linker Nachbar von v, d.h. id(left)<id(v)
  (falls left definiert ist)
- right $\in V \cup \{\perp\}$: rechter Nachbar von v, d.h. id(right)>id(v)
  (falls right definiert ist)

D.h. mit jedem verschickten v wird jetzt auch v.id mit verschickt, damit die IDs bei Bedarf korrigiert werden können.



Sortierte Liste

Sei DS eine Datenstruktur. Zur Erinnerung:

Definition 3.6: Build-DS stabilisiert die Datenstruktur DS, falls Build-DS

1. DS für einen beliebigen Anfangszustand mit schwachem Zusammenhang und eine beliebige faire Rechnung in endlicher Zeit in einen legalen Zustand überführt (Konvergenz) und
2. DS für einen beliebigen legalen Anfangszustand in einem legalen Zustand belässt (Abgeschlossenheit),

sofern keine Operationen auf DS ausgeführt werden und keine Fehler auftreten.

Legaler Zustand für sortierte Liste:

- explizite Kanten formen sortierte Liste
- keine korrumpierten IDs mehr im System

jetzt nichttrivial!

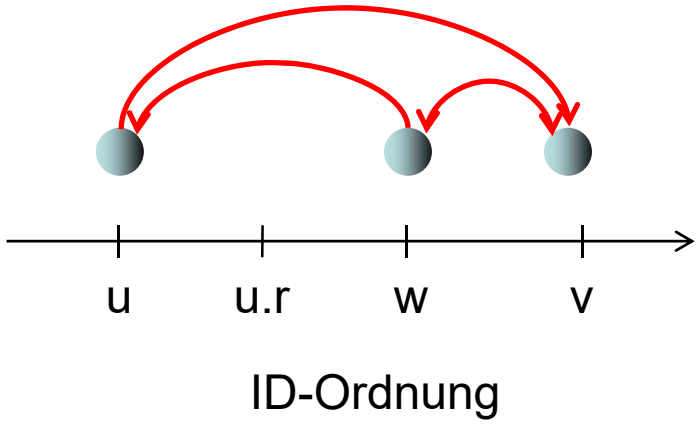
Wir nehmen an: alle korrekten IDs sind verschieden, aber korrumpierte IDs könnten gleich korrekter IDs anderer Prozesse sein.

Sortierte Liste

Problem: Das bisherige Build-List Protokoll funktioniert nicht für korrumpierte IDs.

Beispiel:

- v=u.right, aber id(u.right)<id(v)



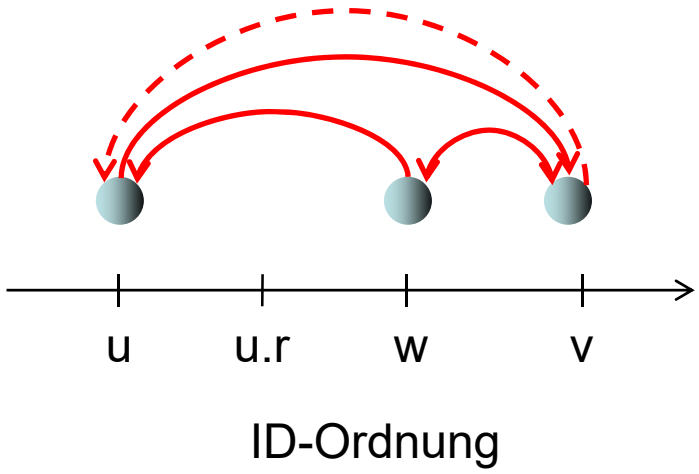
u wird nie ID von v erhalten, da u nicht v´s nächster Nachbar ist, so dass u id(u.right) nicht korrigieren kann

Sortierte Liste

Problem: Das bisherige Build-List Protokoll funktioniert nicht für korrumpierte IDs.

Beispiel:

- v=u.right, aber id(u.right)<id(v)



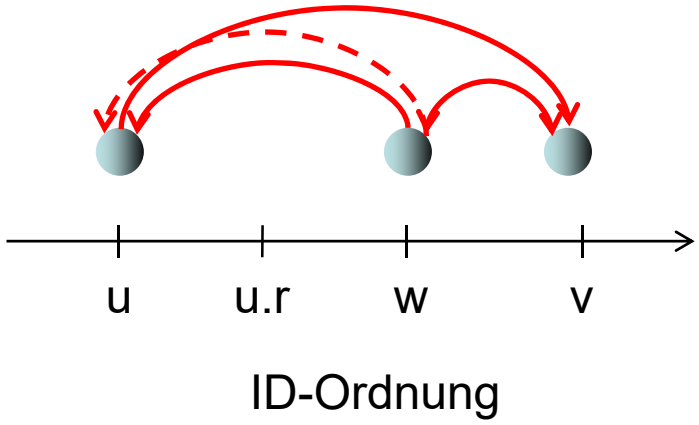
u wird nie ID von v erhalten, da u nicht v´s nächster Nachbar ist, so dass u id(u.right) nicht korrigieren kann

Sortierte Liste

Problem: Das bisherige Build-List Protokoll funktioniert nicht für korrumpierte IDs.

Beispiel:

- v=u.right, aber id(u.right)<id(v)



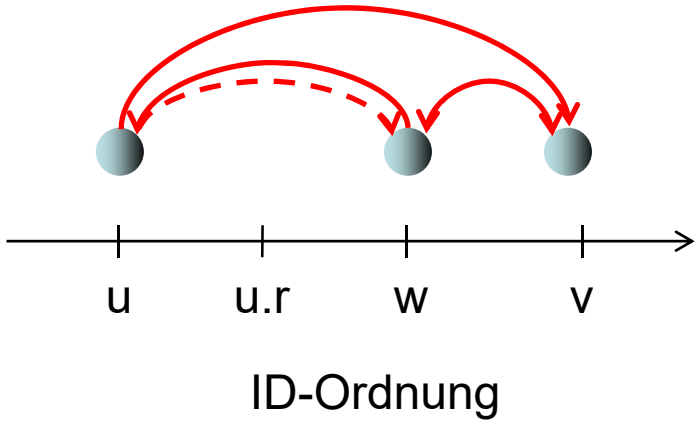
u wird nie ID von v erhalten, da u nicht v´s nächster Nachbar ist, so dass u id(u.right) nicht korrigieren kann

Sortierte Liste

Problem: Das bisherige Build-List Protokoll funktioniert nicht für korrumpierte IDs.

Beispiel:

- v=u.right, aber id(u.right)<id(v)



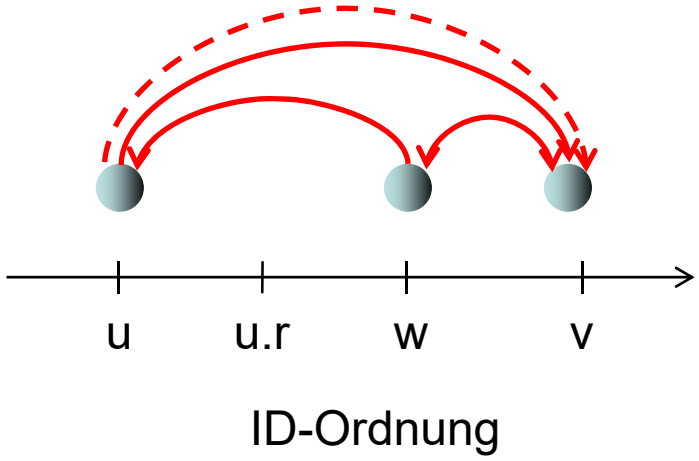
u wird nie ID von v erhalten, da u nicht v´s nächster Nachbar ist, so dass u id(u.right) nicht korrigieren kann

Sortierte Liste

Problem: Das bisherige Build-List Protokoll funktioniert nicht für korrumpierte IDs.

Beispiel:

- v=u.right, aber id(u.right)<id(v)

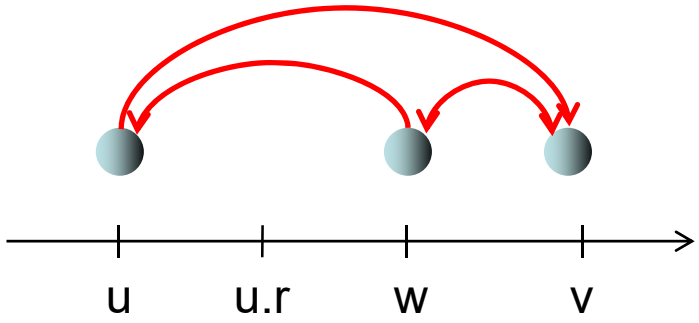


u wird nie ID von v erhalten, da u nicht v´s nächster Nachbar ist, so dass u id(u.right) nicht korrigieren kann

Sortierte Liste

Ergänzungen:

- u ruft statt u.right←linearize(u) in timeout() u.right←check(u, id(u.right)) auf
- Für jedes verschickte v wird auch v.id verschickt.



ID-Ordnung

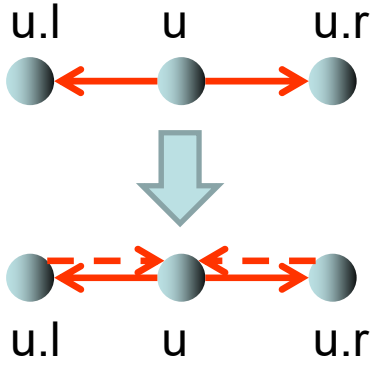
u wird nie ID von v erhalten, da u nicht v´s nächster Nachbar ist, so dass u id(u.right) nicht korrigieren kann

Build-List Protokoll

```
timeout: true →
    { durchgeführt von Knoten u }
    if id(left)≤id then
        left←check(this, id(left))
    else
        this←linearize(left)
        left:= ⊥
    if id(right)≥id then
        right←check(this, id(right))
    else
        this←linearize(right)
        right:= ⊥

check(v, idu) →
    if id≠idu then
        v←linearize(this)          { teile v korrektes id(u) mit }
    else
        linearize(v)               { sonst wie bisher }
```

id(u.l)<id(u) und
id(u.r)>id(u):



Build-List Protokoll

```
linearize(v) →
    { ausgeführt in Knoten u }
    if v=left or v=right then                    { muss left oder right aktualisiert werden? }
        if v=left and id(v)≠id(left) then
            if id(v)≤id then id(left):=id(v)     { id immer noch links von u oder gleich u? }
            else
                this←linearize(v)
                left:= ⊥
        if v=right and id(v)≠id(right) then
            if id(v)≥id then id(right):=id(v)    { id immer noch rechts von oder gleich u?}
            else
                this←linearize(v)
                right:= ⊥
    else                                 { sonst ähnlich zum alten linearize }
        if id(v)≤id(left) then
            left←linearize(v)
        if id(left)<id(v)≤id then
            v←linearize(left)
            left:=v; id(left):=id(v)
        if id<id(v)<id(right) then
            v←linearize(right)
            right:=v; id(right):=id(v)
        if id(right)≤id(v) then
            right←linearize(v)
```

Sortierte Liste

Lemma 5.12: Für alle initialen Systemzustände S erreicht Bild-List in endlicher Zeit einen Zustand ohne korrumpierte IDs.

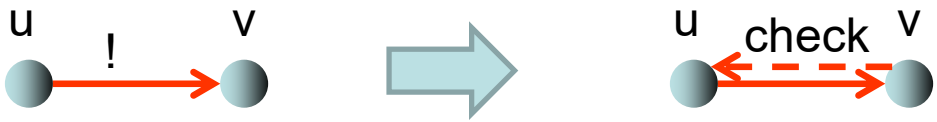
Beweis:

- Spur der Kante (u,u.right): Folge $(v_1,v_2,\ldots)$ von Knoten, die u.right durchläuft, ohne dass id(u.right) korrigiert wird.
- Da die Anzahl aller Knoten-IDs (korrumpiert oder korrekt) endlich ist, ist auch die Spur von (u,u.right) endlich, da die IDs von u.right in der Spur streng monoton sinken.
- Sei v_k der Endknoten der Spur der Kante (u,u.right). Dann wird für (u,u.right) mit u.right=v_k in endlicher Zeit timeout ausgeführt, was dazu führt, dass eine check(u,id(v_k))-Anfrage an v_k geschickt wird. Dadurch wird eine eventuell fehlerhafte ID von u.right korrigiert, was die Anzahl der Kanten mit fehlerhafter ID-Information absenkt. Dadurch kann sich aber id(u.right) erhöhen, was eine neue Spur verursachen kann.
- Da die Anzahl korrumpierter Knoten-IDs endlich ist und nicht vervielfältigt wird, ist damit auch die Gesamtzahl der Spuren endlich, d.h. in endlicher Zeit sind alle (u,u.right)-Kanten (und analog auch alle (u,u.left)-Kanten) stabil.
- Wenn alle (u,u.right)-Kanten und alle (u,u.left)-Kanten stabil sind, kann es keine Kanten mehr mit korrumpierten IDs geben. (Warum?)

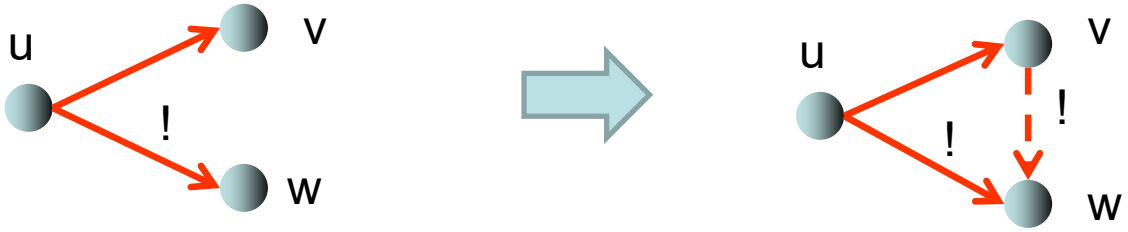
Problem mit korrumpierten IDs

Allgemeine Vorgehensweise:

- Bei Weiterdelegierung keine Überprüfung der korrumpierten Information nötig, da dadurch korrumpierte Information im System nicht erhöht wird.
- Bei Vorstellung müssen wir aber die korrumpierte Information überprüfen, da sie sonst vervielfältigt werden könnte!
- Selbstvorstellung: zusätzlich Überprüfung von u´s Wissen über v durchführen wie bei Build-List Protokoll:



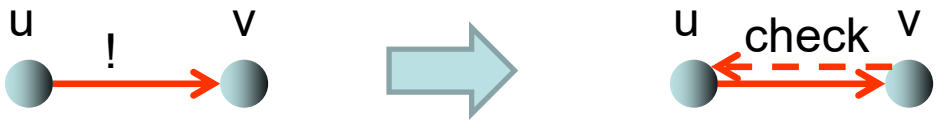
- Fremdvorstellung: Bisherige Vorgehensweise problematisch, da evtl. korrumpierte Information über w an v weitergegeben wird!



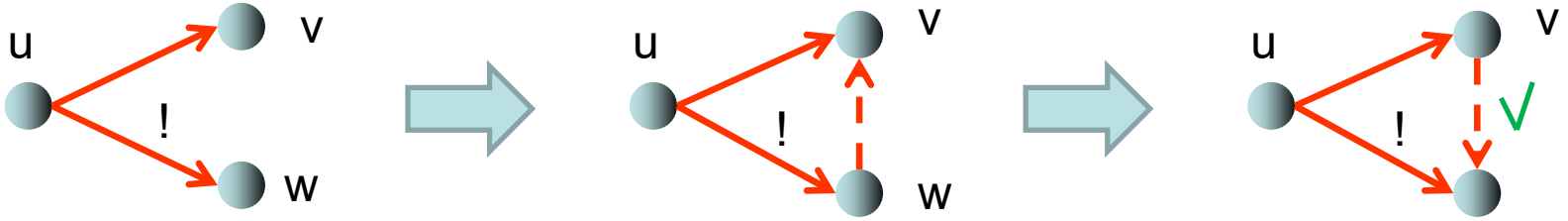
Eingrenzung korrumpierter IDs

Allgemeine Vorgehensweise:

- Bei Weiterdelegierung keine Überprüfung der korrumpierten Information nötig, da dadurch korrumpierte Information im System nicht erhöht wird.
- Bei Vorstellung müssen wir aber die korrumpierte Information überprüfen, da sie sonst vervielfältigt werden könnte!
- Selbstvorstellung: zusätzlich Überprüfung von u´s Wissen über v durchführen wie bei Build-List Protokoll:



- Fremdvorstellung: Stattdessen besser w darum bitten, sich bei v vorzustellen, damit w u´s Information über w korrigieren kann.



Sortierte Liste

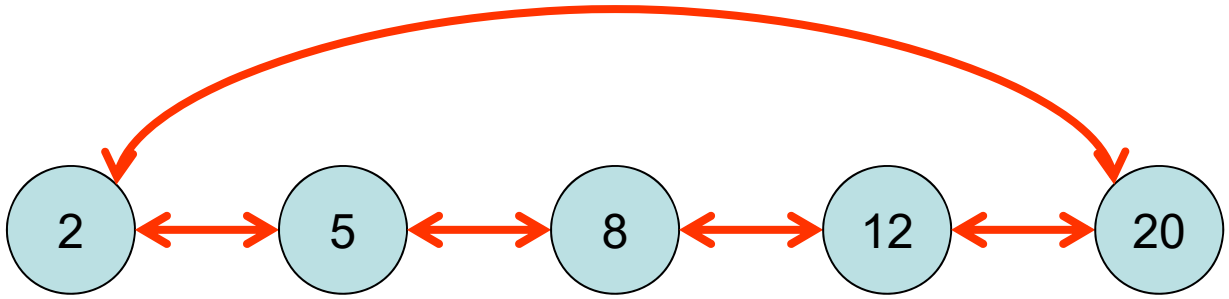
Zusammenfassung:

- Build-List Protokoll für Fall, dass v=id(v)
- Sehr einfache Join, Leave, Search Operationen
- Build-List Erweiterung für monotone Suchbarkeit
- Build-List Erweiterung für Fall, dass Knoten das System verlassen wollen
- Build-List Erweiterung für Fall, dass IDs unabhängig von Referenzen sind

Sortierte Liste

Problem: Eine Liste ist sehr verwundbar gegenüber Ausfällen und gegnerischem Verhalten.

Bessere Lösung: organisiere Knoten im Kreis



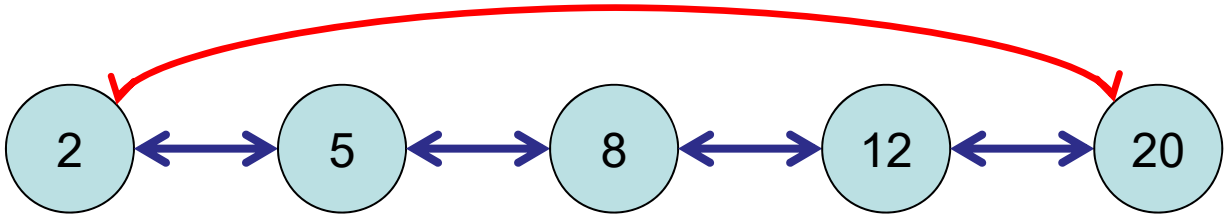
Prozessorientierte Datenstrukturen

Übersicht:

- Sortierte Liste
- Sortierter Kreis
- De Bruijn Graph
- Skip Graph

Sortierter Kreis

Idealzustand: herzustellen durch Build-Cycle Protokoll



Operationen:

- Join(v): Füge Knoten v in Kreis ein
- Leave(v): Entferne Knoten v aus Kreis
- Search(id): Suche nach Knoten mit ID id im Kreis

Sortierter Kreis

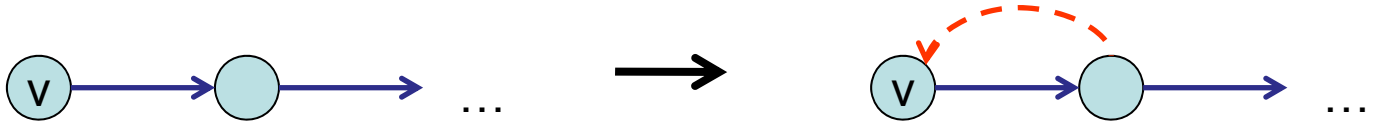
Build-Cycle Protokoll:

- Wir unterscheiden zwischen zwei Typen von Kanten: Listenkanten ($\rightarrow$ und $\dashrightarrow$) und Kreiskanten ($\rightarrow$ und $\dashrightarrow$)
- Alle Kanten zusammengenommen formen anfangs einen schwach zusammenhängenden Graphen.

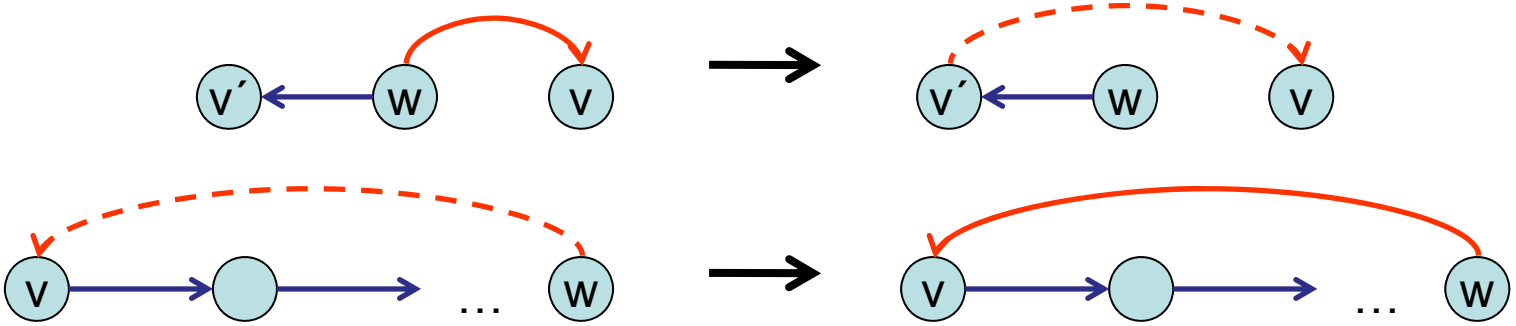
Sortierter Kreis

Regeln für Build-Cycle Protokoll:

- Behandlung der Listenkanten wie bei Build-List.
- Hat Knoten v keinen linken (bzw. rechten) Nachbarn und noch keine Kreiskante, erzeugt v bei timeout eine Kreiskantenanfrage mit Referenz an sich und schickt diese an v.r (bzw. v.l).



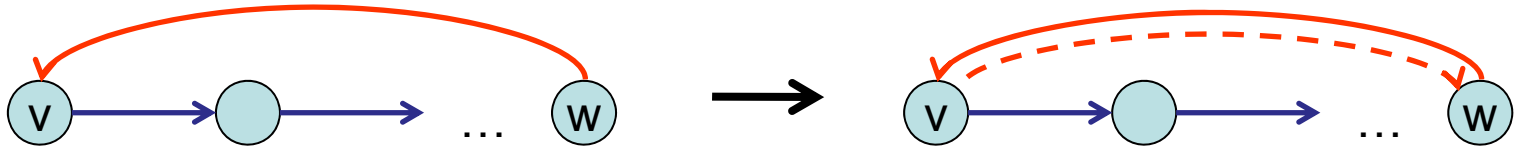
- Hat Knoten w eine Kreiskante bzw. Kreiskantenanfrage zu einem Knoten v mit v<w (bzw. v>w) und ist w.r≠⊥ (bzw. w.l≠⊥), leitet w die Anfrage an w.r (bzw. w.l) weiter. Ist das nicht möglich, erzeugt w eine Kreiskante zu v.



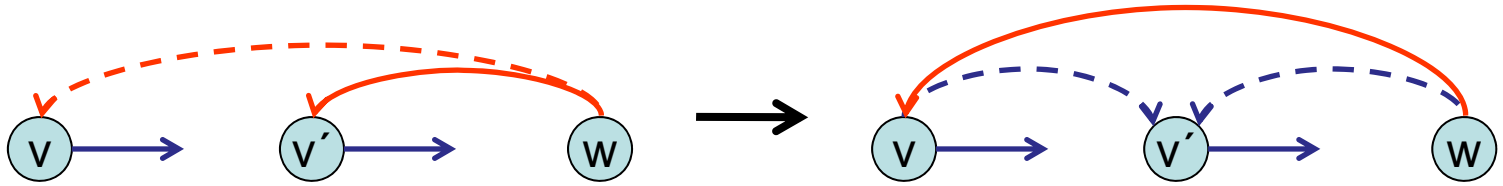
Sortierter Kreis

Regeln für Build-Cycle Protokoll:

- Hat w eine Kreiskante zu v, die nicht weiterleitbar ist, dann erzeugt w bei timeout eine Kreiskantenanfrage mit Referenz an sich und schickt diese an v.

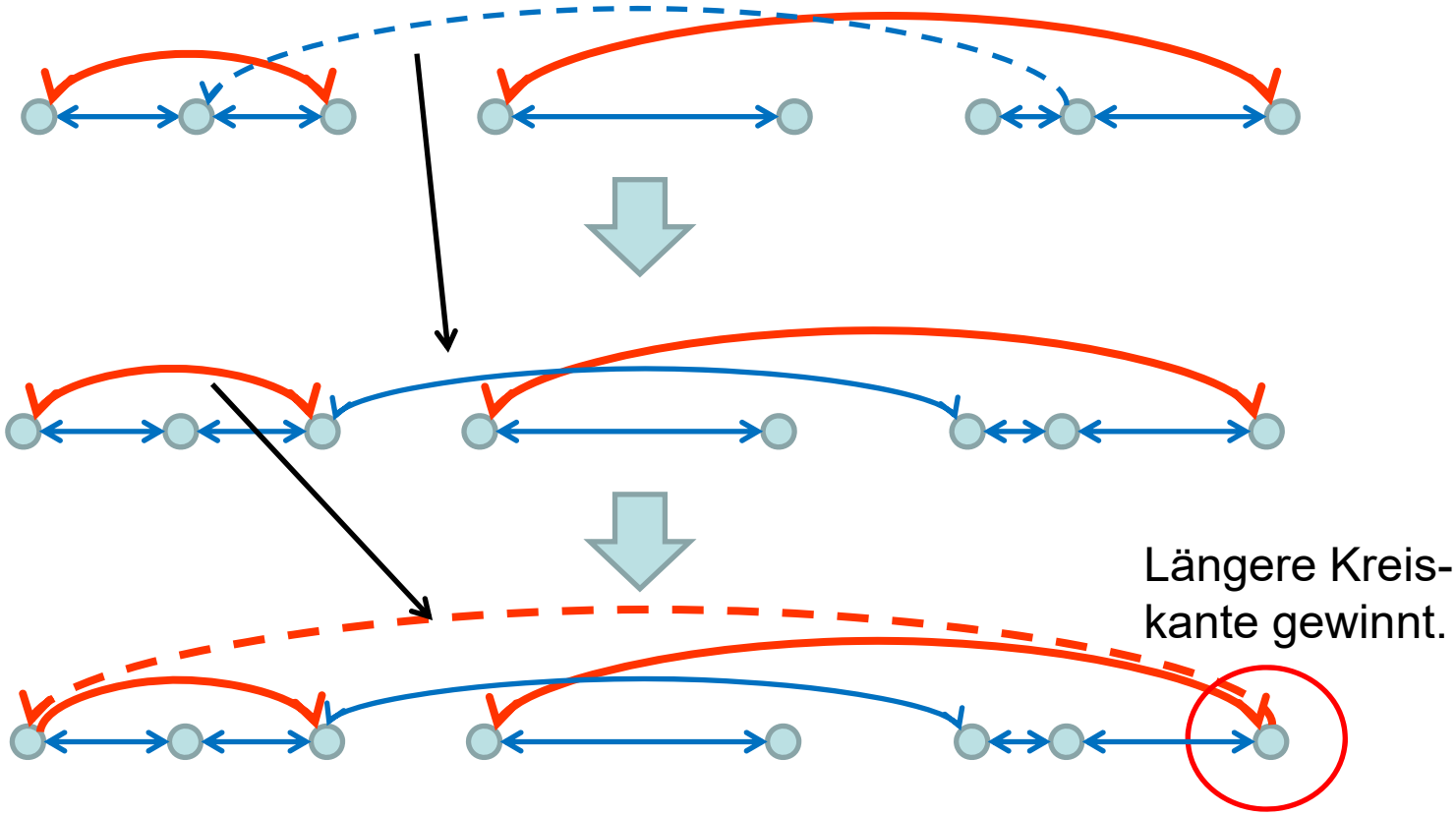


- Hat w bereits eine Kreiskante zu v´ und erhält dann eine Kreiskantenanfrage eines Knotens v≠v´, überlebt die zum weiter entfernten Knoten und zwei Listenkanten werden wie angegeben erzeugt.



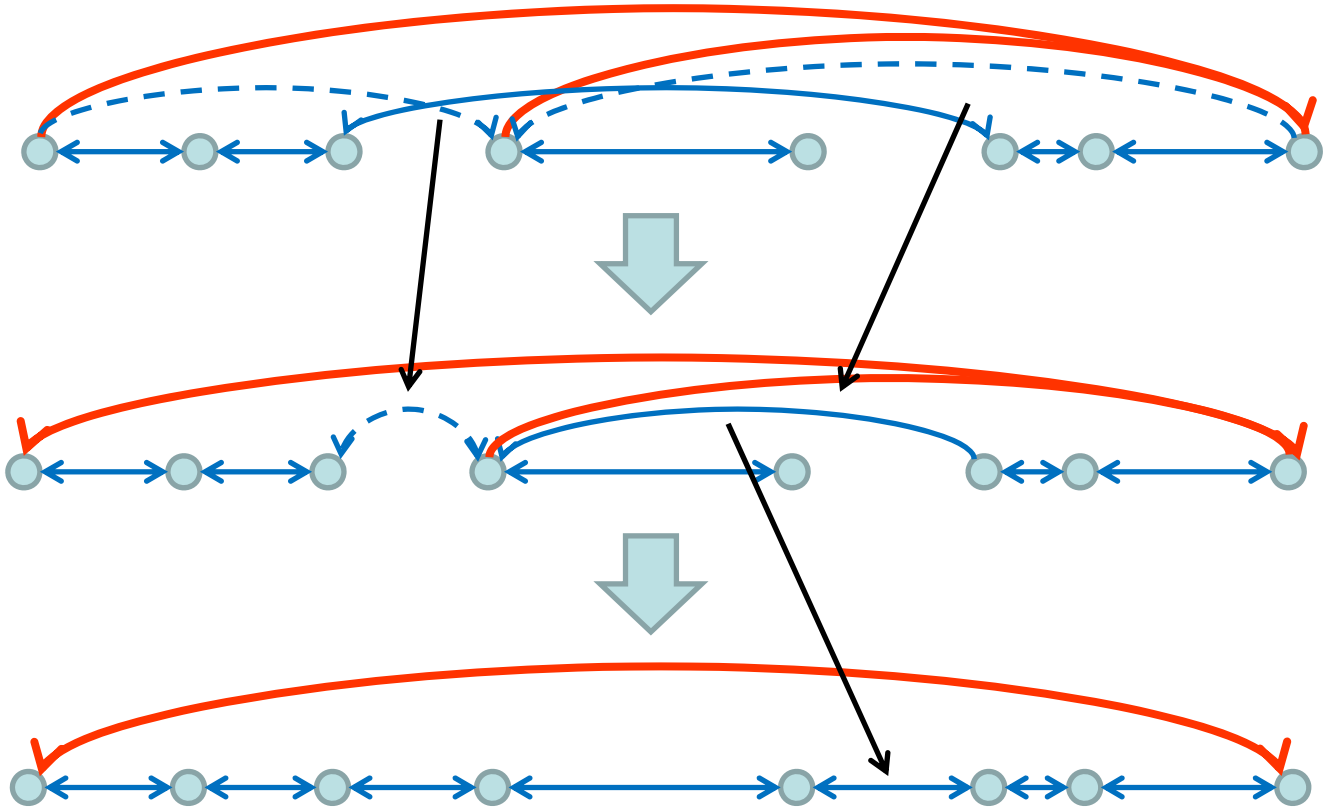
Sortierter Kreis

Kreiskanten behindern nicht Linearisierung:



Sortierter Kreis

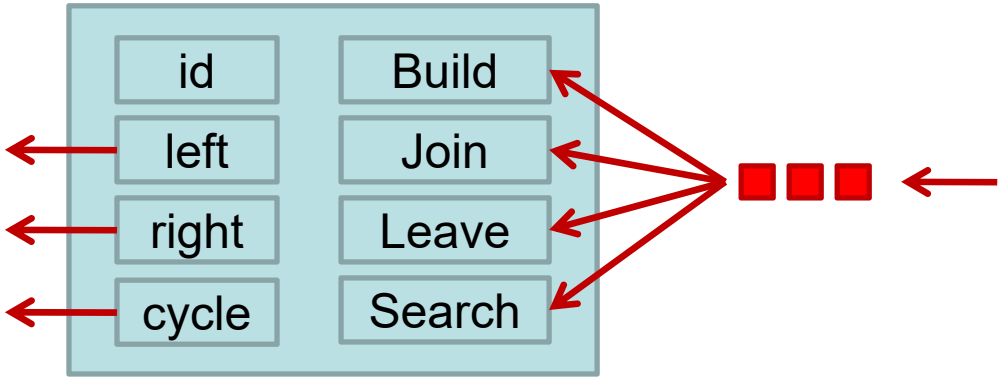
Kreiskanten behindern nicht Linearisierung:



Sortierter Kreis

Variablen innerhalb eines Knotens v:

- id: eindeutiger Name von v (wir schreiben auch id(v))
- left $\in V \cup \{\perp\}$: linker Nachbar von v, d.h. id(left)<id(v) (falls left definiert ist)
- right $\in V \cup \{\perp\}$: rechter Nachbar von v, d.h. id(right)>id(v) (falls right definiert ist)
- cycle $\in V \cup \{\perp\}$: Kreiskante von v



Sortierter Kreis

timeout: true →

- cycle=⊥:
  left=⊥ & right≠⊥: right←introduce(this, CYC)
  right=⊥ & left≠⊥: left←introduce(this, CYC)
- cycle≠⊥:
  left≠⊥ & id(cycle)>id: left←introduce(cycle, CYC); cycle:=⊥
  right≠⊥ & id(cycle)<id: right←introduce(cycle, CYC); cycle:=⊥
  left=⊥ & id(cycle)>id oder right=⊥ & id(cycle)<id:
  cycle←introduce(this, CYC) (erzeuge Rückwärtskante)
- Behandlung von left und right wie in timeout in Build-List mit Aufrufen introduce(this,LIN).

Kreiskantenanfrage

Normale Linearisierungsanfrage

Sortierter Kreis

introduce(v,CYC) →

- cycle=⊥:
  id(v)<id & right=⊥ oder id(v)>id & left=⊥:
  setze cycle auf v
  id(v)<id & right≠⊥: right←introduce(v,CYC)
  id(v)>id & left≠⊥: left←introduce(v,CYC)
- cycle≠⊥:
  v liegt auf derselben Seite von u wie cycle:
  Sei w∈{v,cycle} der weiter entfernte Knoten zu u und
  w´∈ {v,cycle} der andere Knoten.
  cycle:=w
  this←introduce(w´,LIN)
  w←introduce(w´,LIN)

  v liegt auf der entgegengesetzten Seite von cycle:
  this←introduce(v,LIN)
  this←introduce(cycle,LIN)
  cycle:=⊥

Normale Linearisierungsanfrage

Sortierter Kreis

Aufruf von introduce(v,LIN):

- wie in linearize(v)

Satz 5.13 (Konvergenz): Build-Cycle erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen sortierten Kreis (sofern E_M nur aus introduce Anfragen besteht).

Beweis: in drei Phasen, für die die folgenden Ziele erreicht werden sollen

- Phase 1: Graph schwach zusammenhängend bzgl. Listenkanten
  Zeige, dass für jede Kreiskante, die zwei Zusammenhangskomponenten miteinander verbindet, in endlicher Zeit eine Listenkante erzeugt wird, die diese Zusammenhangskomponenten miteinander verbindet. (Übung)
- Phase 2: Listenkanten der Knoten formen sortierte Liste
  Beweis wie für Build-List
- Phase 3: Kreiskante wird geformt
  Zeige, dass sobald sich die sortierte Liste geformt hat, alle überflüssigen Kreiskanten irgendwann verschwinden und die korrekte Kreiskante übrigbleibt. (Übung)

Sortierter Kreis

Satz 5.14 (Abgeschlossenheit): Formen die expliziten Kanten bereits einen sortierten Kreis, wird dieser bei beliebigen introduce Aufrufen erhalten.

Beweis:
Ähnlich zu Satz 5.2. Korrekte Kreiskante wird nie abgebaut.

Monotone Suchbarkeit: Hier muss derselbe Aufwand betrieben werden wie bei der sortierten Liste:

- Kanten werden erst vorgestellt, bevor sie weitergeleitet werden
- Search Anfragen sammeln alle Knoten auf, die sie unterwegs antreffen
- Alternativ: Multikreis

Anpassung der Regeln in Build-Cycle Protokoll: Übung

Sortierter Kreis

Join Operation: wie für die Liste, d.h. für einen neuen Knoten u wird lediglich introduce(u, LIN) aufgerufen.

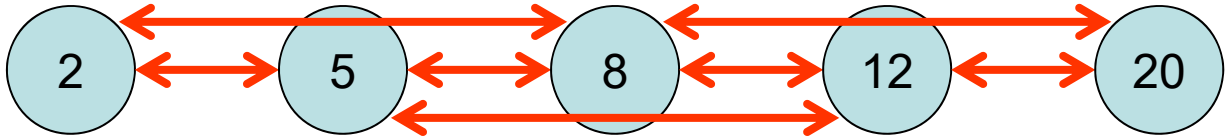
Leave Operation: wie für Liste, d.h. Knoten v setzt lediglich leaving=true. Aber das FDP/FSP-Protokoll muss dann entsprechend an den Kreis angepasst werden. Das kann z.B. eine Herausforderung für das Programmierprojekt sein.

Sortierter Kreis

Problem: Auch ein Kreis ist sehr verwundbar gegenüber Ausfällen und gegnerischem Verhalten.

Alternative Lösung: sortierte d-fache Liste, in der jeder Knoten mit seinen d Vorgängern und Nachfolgern verbunden ist.

Beispiel für d=2:



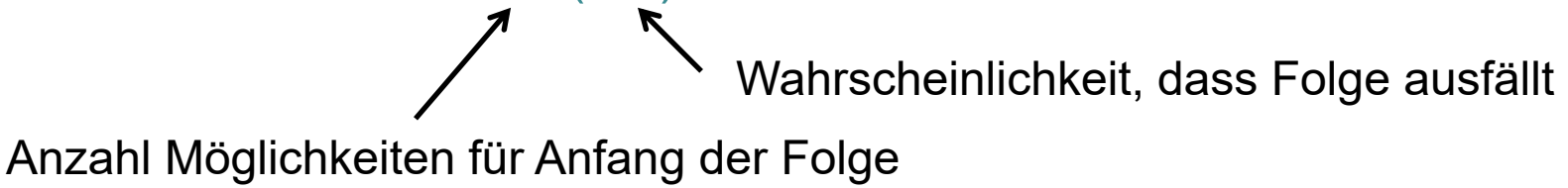
Sortierte d-fache Liste

Satz 5.15: Falls jeder Knoten mit seinen $c \log n$ vielen Vorgängern und Nachfolgern (für eine genügend große Konstante c) verbunden ist, dann ist die Knotenliste auch bei einer Ausfallwahrscheinlichkeit von $1/2$ pro Knoten noch mit hoher Wahrscheinlichkeit zusammenhängend.

Beweis:

- Die Liste zerfällt nur dann, wenn $c \log n$ viele aufeinanderfolgende Knoten ausfallen.
- Die Wahrscheinlichkeit dafür ist höchstens

$$n \, (1/2)^{c \log n} = n^{-c+1}$$



Problem: Wie weiß ein Knoten, wieviel $c \log n$ ist (da er n nicht kennt)?

Sortierte d-fache Liste

Lösung (für zufällige IDs aus [0,1)):

- Knoten v verbindet sich mit allen Knoten in einer Entfernung bis zu $1/2^j$, für die die Gleichung

$$j = N(j)/c - \log N(j) \qquad (*)$$

möglichst gut erfüllt ist, wobei $N(j)$ die Anzahl der Nachbarn in $[v, v+1/2^j)$ ist.

Begründung für die Regel:

- Angenommen, für das gewählte j ist $N(j) = \alpha \cdot c \log n$ für ein α.
- Idealerweise sollte $N(j)$ gleich der erwarteten Anzahl der Knoten in $[v, v+1/2^j)$ sein, d.h. $N(j) = n/2^j$. (Das gilt auch bis auf eine $(1 \pm \varepsilon)$-Abweichung mit hoher Wahrscheinlichkeit, wenn $n/2^j = \Omega(\log n)$ ist.)
- In diesem Fall gilt

$$N(j)/\alpha = c \log n = c \log (2^j N(j))$$
$$= c(j + \log N(j))$$

und daher

$$j = N(j)/(\alpha \cdot c) - \log N(j)$$

- D.h. die Gleichung $(*)$ gilt wenn $\alpha = 1$.

Sortierte d-fache Liste

Umsetzung: Knoten v erweitert seine Nachbarschaft nach rechts (bzw. links), bis zum erstenmal $j<N(j)/c - \log N(j)$ ist. Bzw. wenn das schon für eine kleinere Nachbarschaft gilt, wird diese abgebaut.

Übung: Regel für selbststabilisierende Liste mit Kanten zu d Vorgängern und Nachfolgern (für festes d).

Problem: Struktur jetzt zwar deutlich robuster, aber Durchmesser noch sehr hoch.

Alternative: de Bruijn Graph

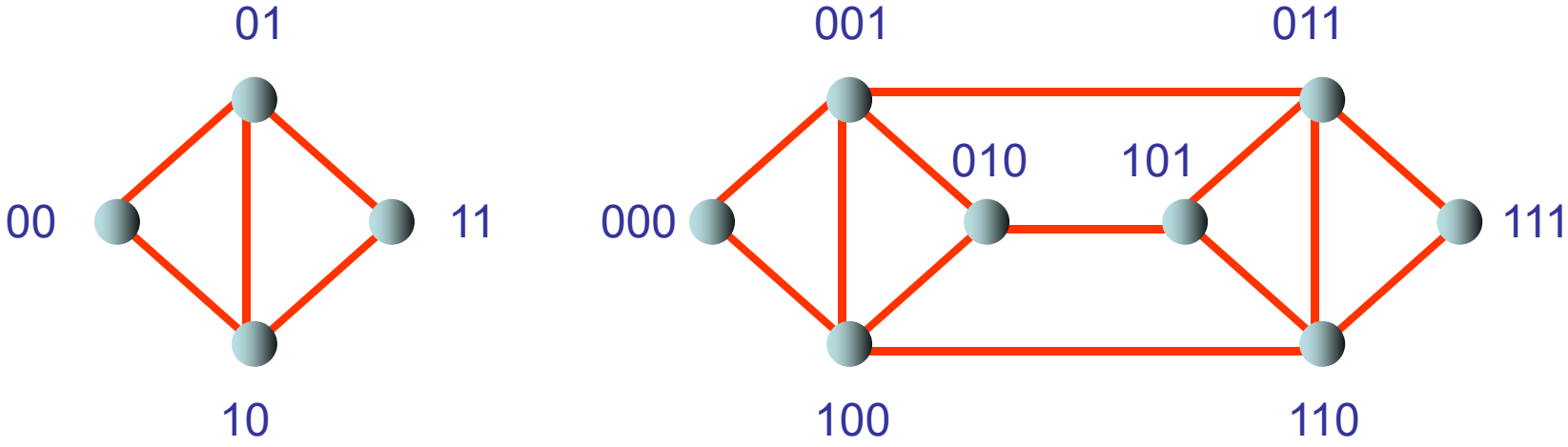
Prozessorientierte Datenstrukturen

Übersicht:

- Sortierte Liste
- Sortierter Kreis
- De Bruijn Graph
- Skip Graph

De Bruijn Graph

- Knoten: $(x_1,\ldots,x_d) \in \{0,1\}^d$
- Kanten: $(x_1,\ldots,x_d) \rightarrow (0,x_1,\ldots,x_{d-1})$
  $(1,x_1,\ldots,x_{d-1})$

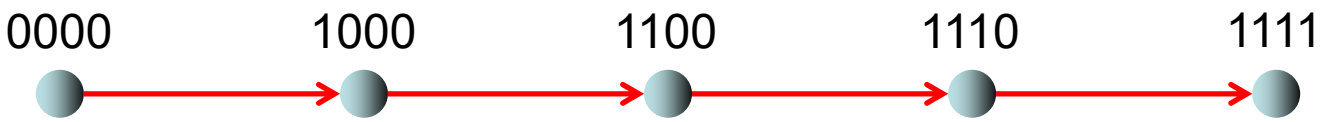


De Bruijn Graph

Routing im de Bruijn Graph: Bitshifting

Anzahl Hops: maximal log n bei n Knoten.

Beispiel: Routing von 0000 nach 1111.



De Bruijn Graph

Klassischer d-dim. de Bruijn Graph G=(V,E):

- $V = \{0,1\}^d$
- $E = \{ \{x,y\} \mid x=(x_1,\ldots,x_d),\ y=(b,x_1,\ldots,x_{d-1}),$ $b \in \{0,1\}$ beliebig$\}$

- Betrachte $(x_1,\ldots,x_d)$ als $0.x_1\ x_2 \ldots x_d \in [0,1)$, d.h. $0.101 = 1\cdot(1/2) + 0\cdot(1/4) + 1\cdot(1/8)$
- Setze $d \to \infty$

De Bruijn Graph

Klassischer d-dim. de Bruijn Graph G=(V,E)

- $V = \{0,1\}^d$
- $E = \{ \{x,y\} \mid x=(x_1,\ldots,x_d),\ y=(b,x_1,\ldots,x_{d-1}),\ b\in\{0,1\}$ beliebig$\}$

Ergebnis für $d\to\infty$:

- $V = [0,1)$
- $E = \{ \{x,y\}\in[0,1)^2 \mid y=x/2,\ y=(1+x)/2 \}$

De Bruijn Graph

Kontinuierlicher de Bruijn Graph:

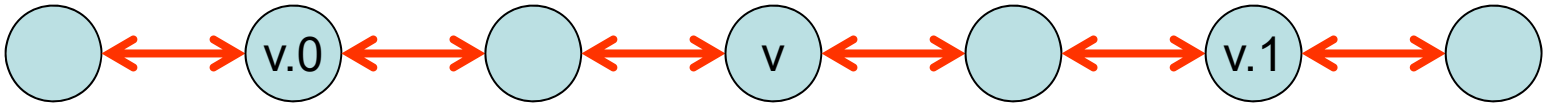
- $V = [0,1)$
- $E = \{ \{x,y\} \in [0,1)^2 \mid y=x/2, y=(1+x)/2 \}$

Dynamischer de Bruijn Graph:

- Wähle pseudo-zufällige Hashfunktion $h:V \rightarrow [0,1)$, die allen Prozessen a priori bekannt ist.
- Weise jedem Prozess $v \in V$ den Punkt $h(v) \in [0,1)$ zu.
- Damit ist bei beliebiger Join-Leave Folge (die unabhängig von den gewählten Punkten ist) die Punktmenge der aktuellen Prozesse gleichverteilt über $[0,1)$.
- Jeder Prozess v simuliert drei Knoten: v, $v.0$ und $v.1$ mit Positionen $h(v)$, $h(v)/2$, und $(1+h(v))/2$ in $[0,1)$.
- Im folgenden ist die ID eines Knoten v gleich seinem Hashwert, d.h. $id(v)=h(v)$.

De Bruijn Graph

Idealzustand: sortierte Liste der Knoten, herzustellen durch Build-deBruijn Protokoll



Operationen:

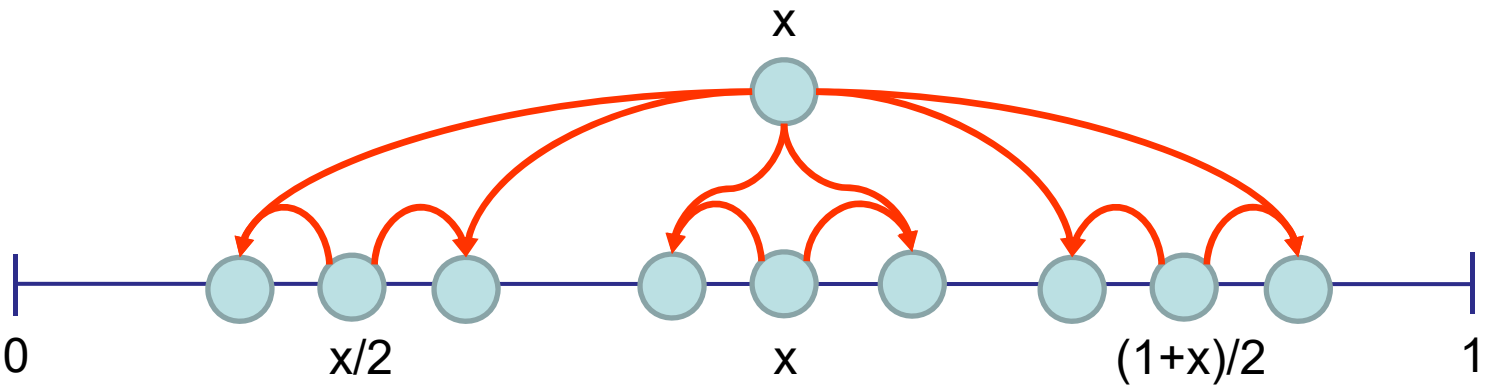
- Join(v): Füge Prozess v in de Bruijn Graph ein
- Leave(v): Entferne Prozess v aus de Bruijn Graph
- Search(id): Suche nach Knoten mit ID id

De Bruijn Graph

Warum sortierte Liste über den Knoten?

Ein Prozess x hält dann folgende Verbindungen.

Er kann damit de Bruijn Hops simulieren.

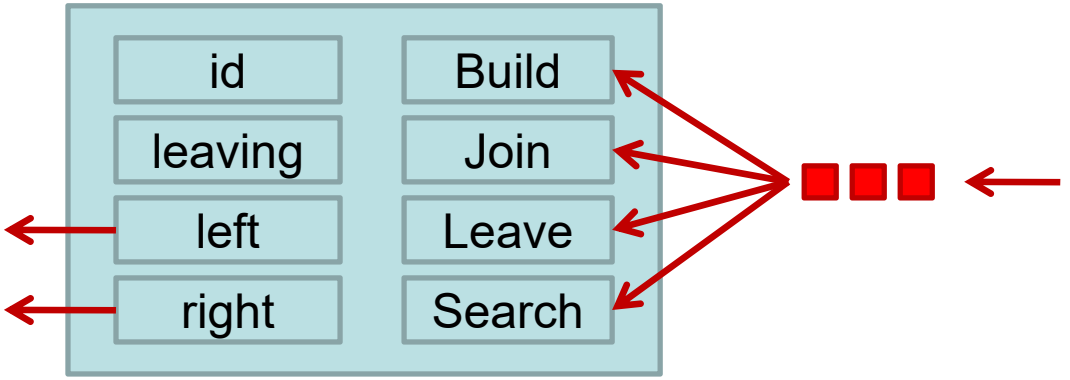


Wie wir sehen werden, reicht das aus, so dass Routing wie im klassischen de Bruijn Graph möglich ist.

De Bruijn Graph

Variablen innerhalb des Knotens $u \in \{v, v.0, v.1\}$:

- id: eindeutige Position von u in [0,1) (ergibt sich aus seiner Referenz und h)
- left $\in V \cup \{\perp\}$: linker Nachbar von u, d.h. id(left)<id(u) (falls id(left) definiert ist)
- right $\in V \cup \{\perp\}$: rechter Nachbar von u, d.h. id(right)>id(u) (falls id(right) definiert ist)



De Bruijn Graph

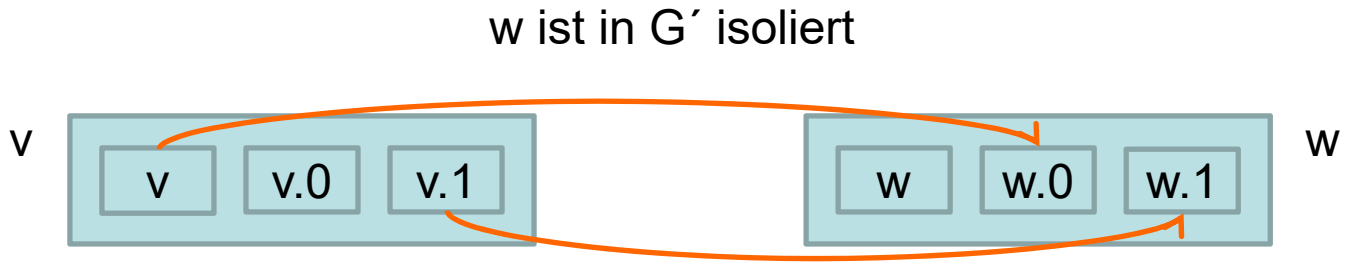
Generelles Vorgehen: Build-deBruijn arbeitet wie Build-List

Beobachtung: Falls der Graph $G'=(V',E')$ mit
- Knotenmenge $V'=\{ v, v.0, v.1 \mid v \in V \}$ (V: Menge der Prozesse) und
- Kantenmenge E'

schwach zusammenhängend ist, dann konvertiert Build-deBruijn diesen in eine sortierte Liste.

Problem: G' muss nicht schwach zusammenhängend sein, obwohl G (der Graph über den Prozessen) schwach zusammenhängend ist.

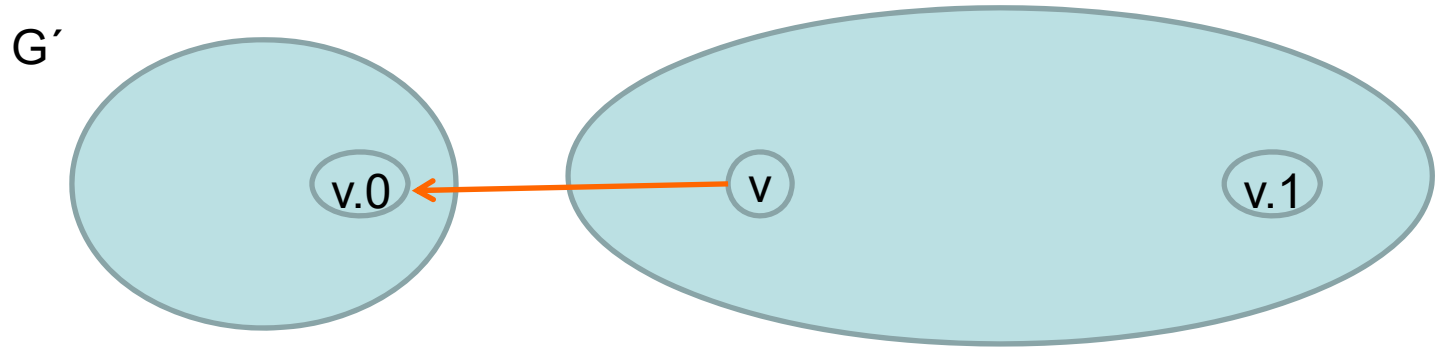
Beispiel:



De Bruijn Graph

Idee: Probing.

Jeder Prozess v testet kontinuierlich, ob für seine Knoten gilt, dass v und v.0 sowie v und v.1 in einer (schwachen) Zusammenhangskomponente in G´ sind. Falls nicht, dann verbindet sich v mit v.0 bzw. v.1 (duch Aufruf von linearize(v.0) bzw. linearize(v.1) in v).

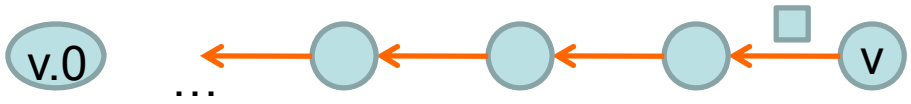


De Bruijn Graph

Naive Strategie für v.0: v schickt Probe entlang der Kanten (w,w.left) (über spezielle Aktion probe(v)) bis einer der folgenden Fälle eintritt:

1. v.0 wird erreicht
2. Die Probe bleibt bei einem Knoten w>v/2 hängen, der keine linke Kante besitzt
3. Die Probe gelangt zu einem Knoten w>v/2 mit w.left<v/2

In den letzten beiden Fällen initiiert v linearize(v.0).



De Bruijn Graph

Probleme mit naiver Strategie:

- Probe eventuell lange unterwegs
- Im stabilen Zustand hohe Congestion (da jeder Knoten kontinuierlich zwei Proben über einen Weg der Länge bis zu $\Theta(n)$ schickt).

Besser: nutze de Bruijn Kanten aus.

Idee: (für v nach v.0; v nach v.1 ähnlich)

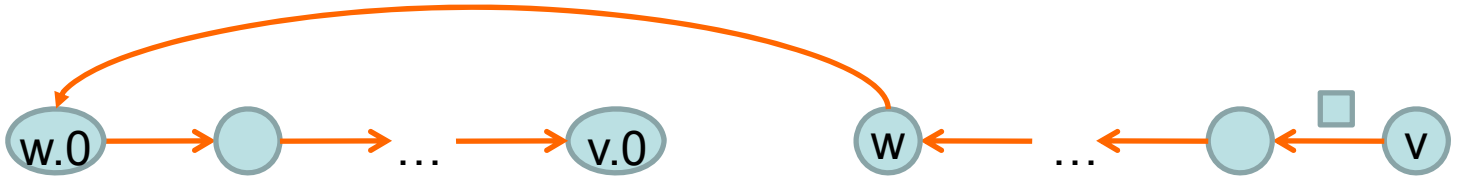
- v schickt Probe nach links bis ein Knoten w mit gleichnamigem Prozess w erreicht wird (so einen Knoten nennen wir deBruijn-Knoten und die anderen Listenknoten)
- w schickt Probe nach w.0 (das ist keine Kante in G´ sondern funktioniert nur deshalb, weil w und w.0 im selben Prozess sind!)
- w.0 schickt Probe nach rechts bis v.0 erreicht wird

De Bruijn Graph

De Bruijn Probing: (für v nach v.0; v nach v.1 ähnlich)

- v schickt Probe nach links bis ein deBruijn-Knoten w erreicht wird
- w schickt Probe nach w.0
- w.0 schickt Probe nach rechts bis v.0 erreicht wird

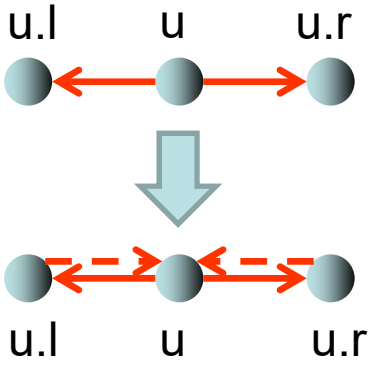
Sollte die Probe hängen bleiben oder v.0 überlaufen,
dann initiiert v linearize(v.0).



Erwartete Länge des Weges im stabilen Zustand: O(1)

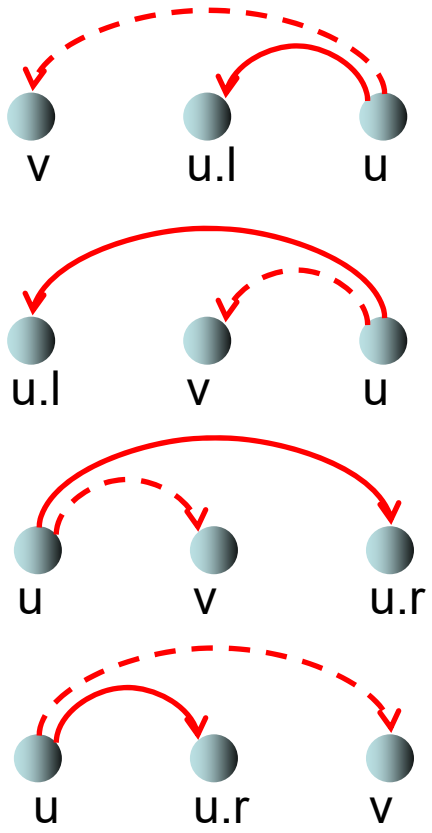
Build-deBruijn Protokoll

```
timeout: true →
    { durchgeführt von Knoten u }
    if id(left)<id then
            left←linearize(this)
        else
            this←linearize(left)
            left:= ⊥
    if id(right)>id then
            right←linearize(this)
        else
            this←linearize(right)
            right:= ⊥
    if u ist de Bruijn Knoten then
        left←probe(0,u.0)    { schicke Proben los }
        right←probe(1,u.1)
```



Build-deBruijn Protokoll

```
linearize(v) →
   { ausgeführt in Knoten u }
   if id(v)<id(left) then
      left←linearize(v)
   if id(left)<id(v)<id then
      v←linearize(left)
      left:=v
   if id<id(v)<id(right) then
      v←linearize(right)
      right:=v
   if id(right)<id(v) then
      right←linearize(v)
```



Build-deBruijn Protokoll

```
probe(x, vx) →   { wird von u ausgeführt }
    if x=1 then
        if id<id(vx) then
            if u is a deBruijn node then
                u.1←probe(x,vx)
            else
                if right≠⊥ and id(right)≤id(v.x) then
                    right←probe(x,vx)
                else
                    this←linearize(vx)      { verbinde u mit v.1: für Zusammenhang! }
                    vx←introduce()          { verbinde v mit v.1 }
        if id>id(vx) then
            if left≠⊥ and id(left)≥id(vx) then
                    left←probe(v,vx)
                else
                    this←linearize(vx)      { verbinde u mit v.1: für Zusammenhang!  }
                    vx←introduce()          { verbinde v mit v.1 }
        { sonst id=id(vx), Suche war also erfolgreich }
    else
        ….  { Fall x=0 ähnlich zu x=1; x weder 0 noch 1: linearize(vx) }
```

Build-deBruijn Protokoll

introduce() $\rightarrow$

{ ausgeführt von Knoten u }

if u ist v.0 für ein $v \in V$ then

$v \leftarrow$linearize(v.0)

if u ist v.1 für ein $v \in V$ then

$v \leftarrow$linearize(v.1)

{ sonst muss nichts vorgestellt werden }

De Bruijn Graph

Satz 5.22: Build-deBruijn transformiert jeden schwach zusammenhängenden Graphen G=(V,E) in eine doppelt verkettete sortierte Liste über der Menge der Knoten V´.

Beweis: besteht aus zwei Teilen.

1. Schwacher Zusammenhang von G $\rightarrow$ schwacher Zusammenhang von G´
2. Schwacher Zusammenhang von G´ $\rightarrow$ G´ formt sortierte Liste

- 2. folgt aus Analyse des Build-List Protokolls.
- Es bleibt also, 1. zu zeigen.

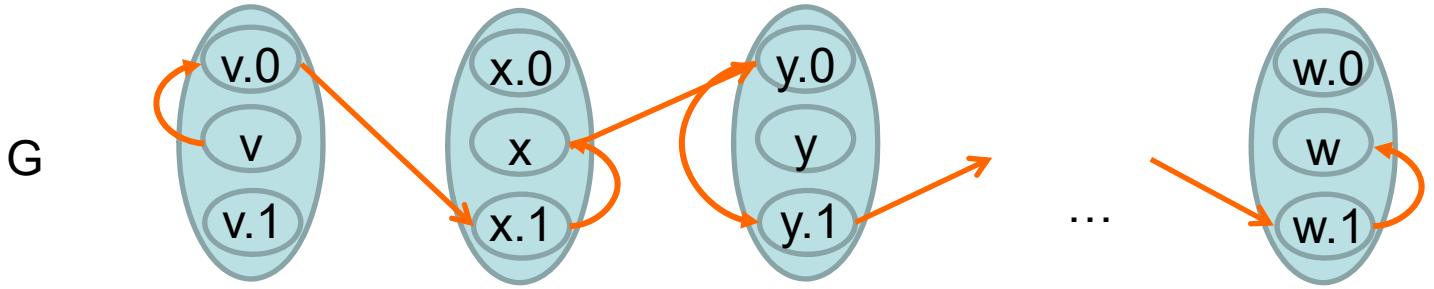
De Bruijn Graph

Build-deBruijn: führe linearize zusammen mit de Bruijn Probing aus.

Satz 5.22: Build-deBruijn transformiert jeden schwach zusammenhängenden Graphen G=(V,E) in eine doppelt verkettete sortierte Liste über der Menge der Knoten V´.

Beweis (Fortsetzung): Schwacher Zusammenhang von G → schwacher Zusammenhang von G´

- Es gilt: ist G schwach zusammenhängend und jeder Knoten v mit v.0 und v.1 in derselben schwachen Zusammenhangskomponente in G´, dann ist auch G´ schwach zusammenhängend.
  Beispiel: Pfad von v zu w



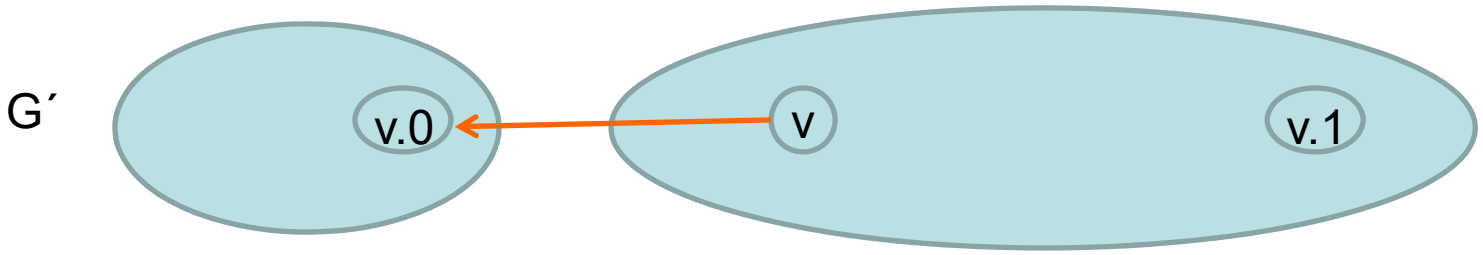
De Bruijn Graph

Build-deBruijn: führe linearize zusammen mit de Bruijn Probing aus.

Satz 5.22: Build-deBruijn transformiert jeden schwach zusammenhängenden Graphen G=(V,E) in eine doppelt verkettete sortierte Liste über der Menge der Knoten V´.

Beweis (Fortsetzung): Schwacher Zusammenhang von G → schwacher Zusammenhang von G´

- Es gilt: ist G schwach zusammenhängend und jeder Knoten v mit v.0 und v.1 in derselben schwachen Zusammenhangskomponente in G´, dann ist auch G´ schwach zusammenhängend.
- Es bleibt also zu gewährleisten, dass irgendwann v mit v.0 und v.1 in derselben schwachen Zusammenhangskomponente in G´ ist.



De Bruijn Graph

Build-deBruijn: führe linearize zusammen mit de Bruijn Probing aus.

Satz 5.22: Build-deBruijn transformiert jeden schwach zusammenhängenden Graphen G=(V,E) in eine doppelt verkettete sortierte Liste über die Menge der Knoten V´.

Beweis (Fortsetzung): Schwacher Zusammenhang von G $\rightarrow$ schwacher Zusammenhang von G´

- Betrachte am weitesten links liegenden Knoten v einer Zusammenhangskomponente (kurz ZHK), der noch nicht in derselben ZHK wie v.0 in G´ ist.
- Wir wollen dann zeigen, dass das de Bruijn Probing für v scheiten muss und sich damit v mit v.0 verbindet, so dass diese anschließend in einer ZHK sind.
- Damit gibt es irgendwann keinen Knoten v mehr, der nicht mit v.0 in einer ZHK ist. Dasselbe Argument kann für v.1 gezeigt werden. D.h. irgendwann muss G´ schwach zusammenhängend sein.

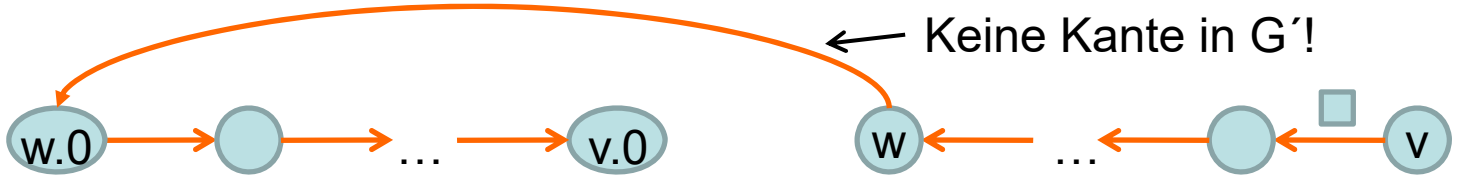
De Bruijn Graph

Build-deBruijn: führe linearize zusammen mit de Bruijn Probing aus.

Satz 5.22: Build-deBruijn transformiert jeden schwach zusammenhängenden Graphen G=(V,E) in eine doppelt verkettete sortierte Liste über die Menge der Knoten V´.

Beweis (Fortsetzung):

- Betrachte am weitesten links liegenden Knoten v, der noch nicht in derselben Zusammenhangskomponente (kurz ZHK) wie v.0 in G´ ist.
- Angenommen, das de Bruijn Probing für v scheitert nicht. Dann muss die Probe einen deBruijn-Knoten w erreichen, der diese dann an w.0 weiterreicht, denn das ist die einzige Möglichkeit, einen Hop durchzuführen, der keiner Kante in G´ entspricht. In diesem Fall wären v und w bzw. w.0 und v.0 wegen der Verwendung von Listenkanten in derselben ZHK. w und w.0 müssten aber aufgrund der Annahme über v in derselben ZHK sein, aber dann wären auch v und v.0 in derselben ZHK, ein Widerspruch zur Annahme.



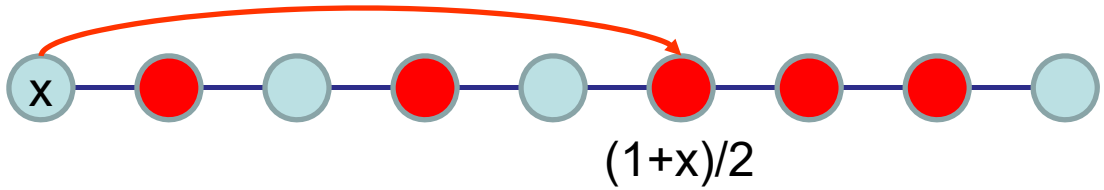
De Bruijn Graph

Routing im klassischen de Bruijn Graph:

$(x_1,\ldots,x_d) \rightarrow (y_d,x_1,\ldots,x_{d-1}) \rightarrow (y_{d-1},y_d,x_1,\ldots,x_{d-2}) \rightarrow \ldots \rightarrow (y_1,\ldots,y_d)$

Routing im dynamischen de Bruijn Graph:

- $(x_1,x_2,\ldots) \rightarrow (y_d,x_1,x_2,\ldots)$ möglich ohne den Prozess zu wechseln, da $(y_d,x_1,x_2,\ldots)$ entweder $x/2$ oder $(1+x)/2$ ist.



: deBruijn-Knoten (v)

: Listenknoten (v.0 oder v.1)

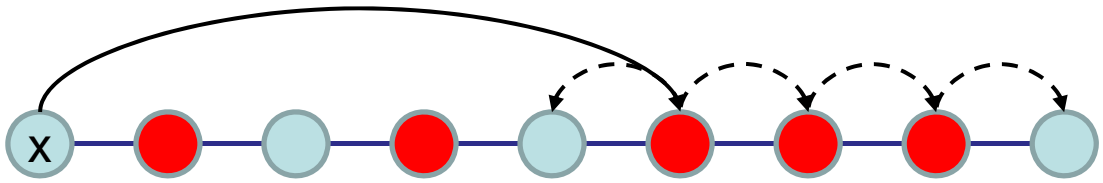
De Bruijn Graph

Routing im klassischen de Bruijn Graph:

$(x_1,\ldots,x_d) \rightarrow (y_d,x_1,\ldots,x_{d-1}) \rightarrow (y_{d-1},y_d,x_1,\ldots,x_{d-2}) \rightarrow \ldots \rightarrow (y_1,\ldots,y_d)$

Routing im dynamischen de Bruijn Graph:

- Dann aber Wechsel auf deBruijn-Knoten notwendig für nächsten de Bruijn hop. Dazu reicht die Suche entlang der linearen Liste.

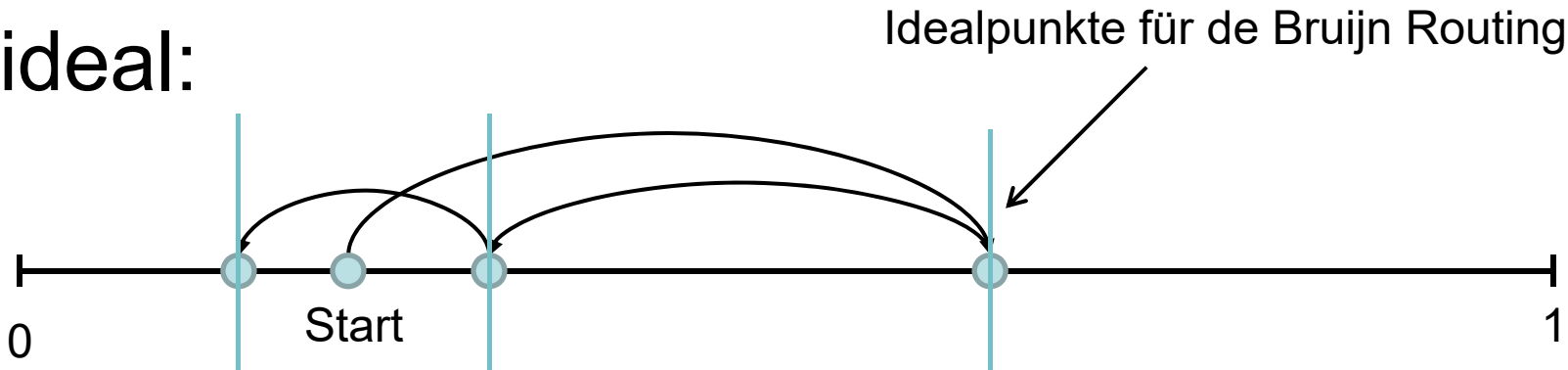


Nach links bzw. rechts bis zum nächsten deBruijn-Knoten.

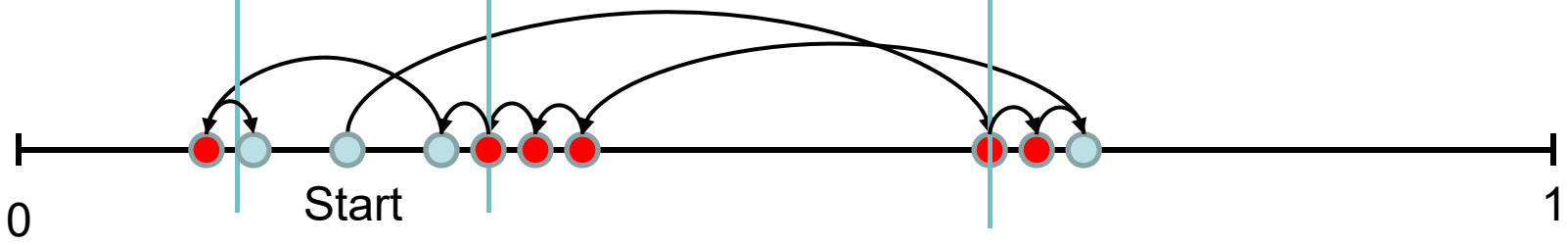
De Bruijn Graph

Beispiel:

- ideal:



- dynamischer de Bruijn Graph:



De Bruijn Graph

Routing im klassischen de Bruijn Graph:

$(x_1,\ldots,x_d) \rightarrow (y_d,x_1,\ldots,x_{d-1}) \rightarrow (y_{d-1},y_d,x_1,\ldots,x_{d-2}) \rightarrow \ldots \rightarrow (y_1,\ldots,y_d)$

Routing im dynamischen de Bruijn Graph:

- für jeden Schritt im klassischen de Bruijn Graphen zwei Phasen: (1) ein de Bruijn Schritt und (2) Suche nach nächstem deBruijn-Knoten in Richtung der Idealposition entlang der Liste
- am Ende (d.h. nach d Phasen), Suche nach Zielknoten entlang Liste

Lemma 5.23: Die Anzahl der Suchschritte pro Phase ist erwartungsgemäß konstant.

Beweis: Übung.

Satz 5.24: Das Routing im dynamischen de Bruijn Graph benötigt erwartungsgemäß $O(\log n)$ Schritte, um von einem Startknoten zu einem beliebigen Zielknoten zu gelangen (sofern die Knoten eine konstante Approximation von $\log n$ haben).

De Bruijn Graph

Routing im klassischen de Bruijn Graph:

$(x_1,\ldots,x_d) \rightarrow (y_d,x_1,\ldots,x_{d-1}) \rightarrow (y_{d-1},y_d,x_1,\ldots,x_{d-2}) \rightarrow \ldots \rightarrow (y_1,\ldots,y_d)$

Routing im dynamischen de Bruijn Graph:

- für jeden Schritt im klassischen de Bruijn Graphen zwei Phasen: (1) ein de Bruijn Schritt und (2) Suche nach nächstem deBruijn-Knoten in Richtung der Idealposition entlang der Liste
- am Ende (d.h. nach d Phasen), Suche nach Zielknoten entlang Liste

Lemma 5.23: Di tungs-
gemäß konst

Problem: Wie kann d(=log n) ermittelt werden?

Beweis: Übung.

Satz 5.24: Das Routing im dynamischen de Bruijn Graph benötigt erwartungsgemäß O(log n) Schritte, um von einem Startknoten zu einem beliebigen Zielknoten zu gelangen (sofern die Knoten eine konstante Approximation von log n haben).

De Bruijn Graph

Problem: finde gute Abschätzung d' von $d=\log n$, wobei n die aktuelle Anzahl der Prozesse im System ist.

Idee: Startknoten s betrachtet nächsten Nachfolger v entlang der sortierten Liste.

Man kann zeigen, dass für alle Startknoten s gilt:

- $|s\text{-}v| \in [1/n^3, 3 \cdot (\log n)/n]$ mit Wahrscheinlichkeit mindestens $1\text{-}1/n$.
- In diesem Fall ist $-\log|s\text{-}v| \in [(\log n)/2, 3 \cdot \log n]$
- Setzen wir also $d' = -2 \cdot \log|s\text{-}v|$, dann ist $d' \geq \log n$ und $d' = \Theta(\log n)$, was für unser Routing reicht. D.h. wir können d' für die Simulation des klassischen de Bruijn Routings im dynamischen de Bruijn Graph verwenden.

De Bruijn Graph

Join(v):

- Führe linearize(w) mit $w \in \{v, v.0, v.1\}$ von irgendeinem Knoten im dynamischen de Bruijn Graph aus.

Leave(v): (einfache Lösung)

- v, v.0 und v.1 verlassen de Bruijn Graph
- Build-deBruijn repariert diesen dann

De Bruijn Graph

Problem: Join(v) braucht wegen Build-List–Anwendung im worst case $\Theta(n)$ Kommunikationsrunden, um v, v.0 und v.1 in dynamischen de Bruijn Graphen einzubauen.

Lösung über Einbau von de Bruijn Routing in Selbst-stabilisierung?

Problem: Leave(v) kann zu Zerfall des Graphen führen.

Satz 5.25: Im stabilen de Bruijn Graphen bleibt der Prozessgraph G mit hoher Wahrscheinlichkeit nach Entfernung eines Prozesses zusammenhängend.

De Bruijn Graph

Satz 5.25: Im stabilen de Bruijn Graphen bleibt der Prozessgraph G mit hoher Wahrscheinlichkeit nach Entfernung eines Prozesses zusammenhängend.

Beweis:

- Zerlege $[0,1)$ in vier Regionen $R_1=[0,v.0)$, $R_2=[v.0,v)$, $R_3=[v,v.1)$ und $R_4=[v.1,1)$.
- Betrachte zunächst den Fall, dass ein deBruijn-Knoten in R_2 existiert und zeige, dass sowohl für $v\leq 1/2$ als auch $v>1/2$ der Prozessgraph G immer noch schwach zusammenhängend sein muss.
- Betrachte danach den Fall, dass ein deBruijn-Knoten in R_3 existiert und zeige, dass sowohl für $v\leq 1/2$ als auch $v>1/2$ der Prozessgraph G immer noch schwach zusammenhängend sein muss.
- Die einzige kritische Fall für den Zusammenhang ist daher, dass es keine deBruijn-Knoten in R_2 und R_3 gibt.
- Da $|R_2|+|R_3|=1/2$, ist die Wahrscheinlichkeit dafür aber $1/2^{n-1}$.
- Daraus folgt Satz 5.25. Der ausführliche Beweis ist eine Übung.

De Bruijn Graph

Bemerkung: Wir brauchen nicht unbedingt drei Knoten pro Prozess. Der Vorteil der Knoten war, dass wir dann die de Bruijn Kanten nicht explizit speichern müssen (wir wechseln z.B. einfach von v zu v.0, um einen de Bruijn Sprung durchzuführen) und damit das Problem des selbststabilisierenden de Bruijn Graphen auf das Problem der selbststabilisierenden Liste (zusammen mit einer Zusammen-hangsprüfung) reduzieren konnten. Wenn wir auf mehrere Knoten pro Prozess verzichten (d.h. es gibt nur einen Knoten pro Prozess), brauchen wir zusätzlich zu left und right explizite de Bruijn Kanten (welche in dafür vorgesehenen Variablen, z.B. deBuijn0 und deBruijn1, gespeichert werden müssen). Das vorgestellte Probing reicht nach wie vor aus um zu testen, ob der Zusammenhang über left und right Nachbarn sichergestellt ist, aber es muss zusätzlich überprüft werden, ob die de Bruijn Kanten auf die richtigen Knoten (d.h. denjenigen Knoten, der am nächsten an x/2 bzw. (1+x)/2 liegt) zeigen.

Durch die veränderte Konstruktion haben wir nach wie vor einen konstanten ausgehenden Grad, aber der eingehende Grad kann jetzt mehr als eine Konstante (aber maximal ~log n) sein. Warum?

Prozessorientierte Datenstrukturen

Übersicht:

- Sortierte Liste
- Sortierter Kreis
- De Bruijn Graph
- Skip Graph

Skip Graph

Betrachte eine beliebige Menge V an Knoten mit totaler Ordnung (d.h. die Knoten können bzgl. einer Ordnung ´<´ sortiert werden).

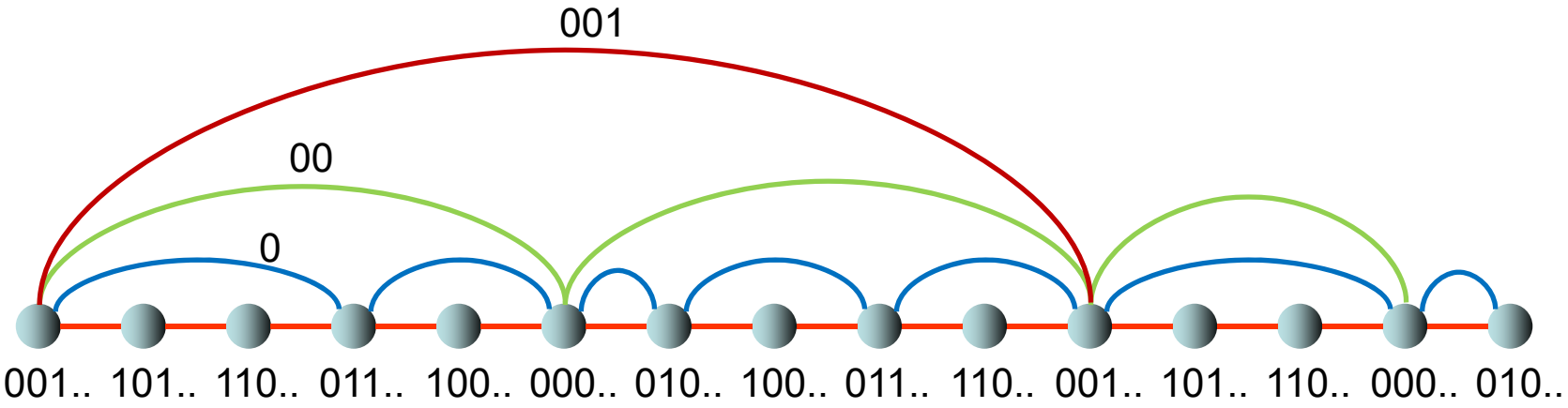
- Jeder Knoten v sei assoziiert mit einer zufälligen Bitfolge $r(v)$.
- $prefix_i(v)$: erste i Bits von $r(v)$
- $succ_i(v)$: nächster Nachfolger w von v (bzgl. der Ordnung ´<´) mit $prefix_i(w)=prefix_i(v)$.
- $pred_i(v)$: nächster Vorgänger w von v mit $prefix_i(w)=prefix_i(v)$.

Skip Graph Regel:
Für jeden Knoten v und jedes $i \in \mathbb{N}_0$:

- v hat eine Kante zu $pred_i(v)$ und $succ_i(v)$

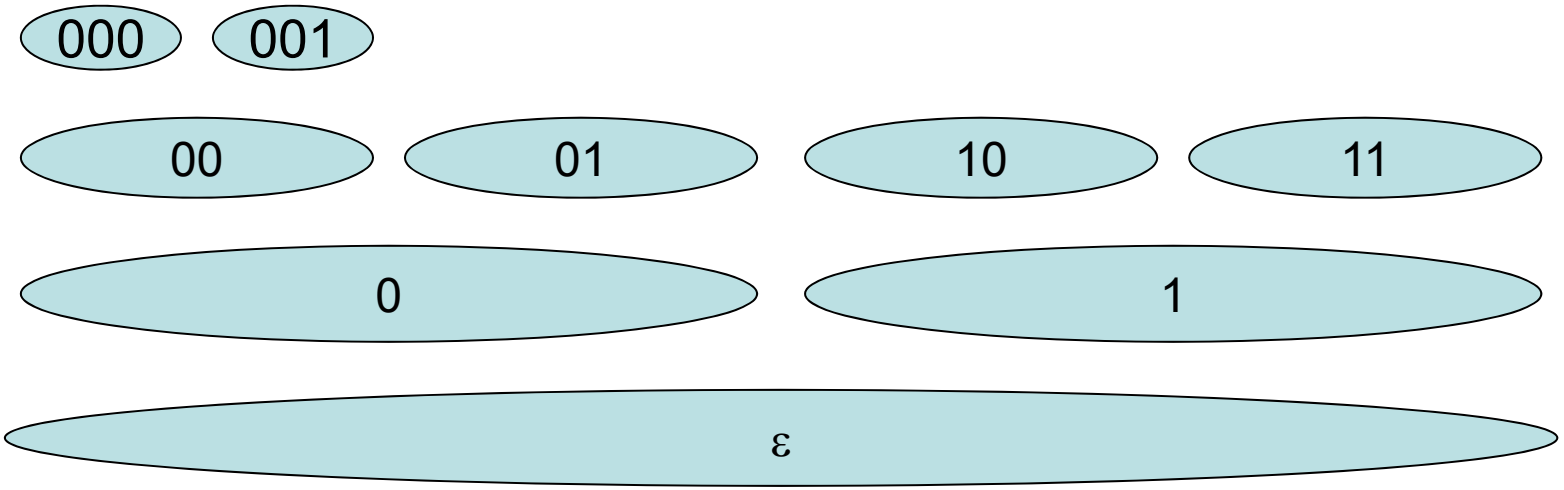
Skip Graph

Beispiel einiger Teillisten für verschiedene Präfixlängen im Skip Graphen:



Skip Graph

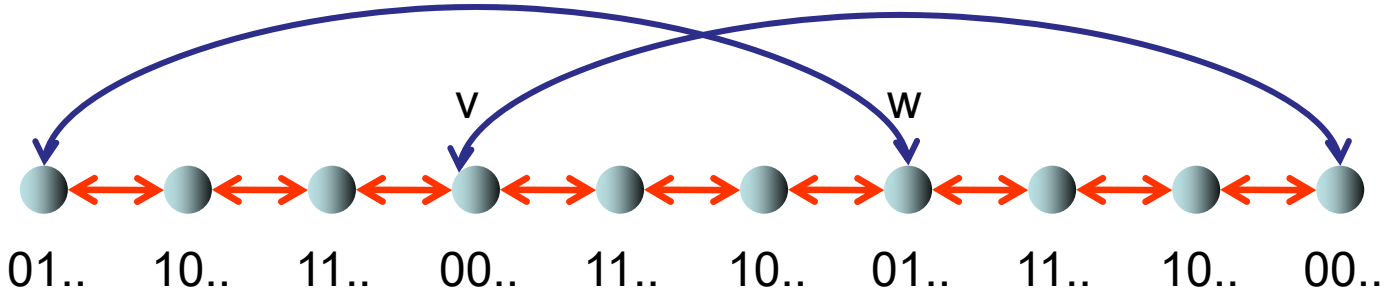
Hierarchische Sicht: geordnete Listen von Knoten mit demselben Präfix.



$\Theta(\log n)$ Grad, $\Theta(\log n)$ Durchmesser, $\Theta(1)$ Expansion (mit hoher Wahrscheinlichkeit)

Skip Graph

Problem: Der Skip Graph erlaubt keine lokale Überprüfung der Korrektheit seiner Struktur.



Aus der Sicht von v und w ist der Skip Graph korrekt.

Skip+ Graph

Problem: Der original Skip Graph erlaubt keine lokale Überprüfung der Korrektheit seiner Struktur.

Lösung: zusätzliche Kanten

Für jeden Knoten v sei

- $succ_i(v,b)$, $b \in \{0,1\}$: nächster Nachfolger von v mit Präfix $prefix_i(v) \cdot b$
- $pred_i(v,b)$, $b \in \{0,1\}$: nächster Vorgänger von v mit Präfix $prefix_i(v) \cdot b$

Skip Graph: $range_i(v)=[pred_i(v),succ_i(v)]$

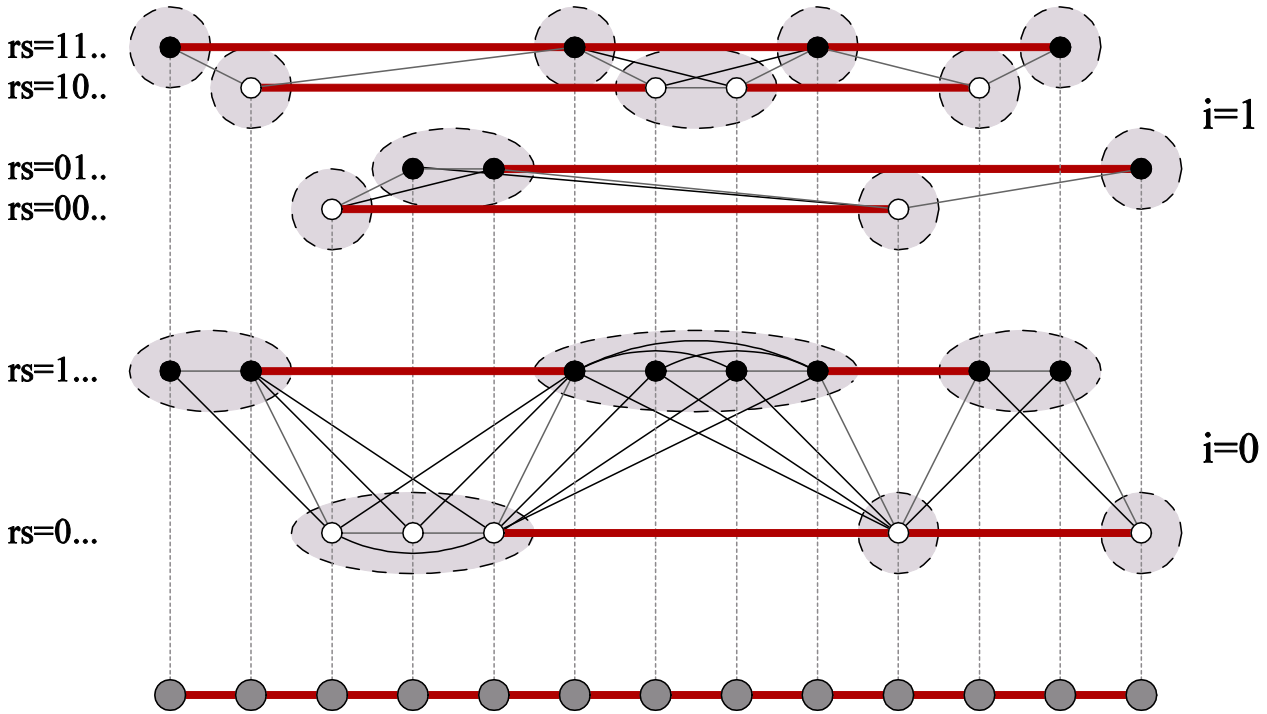
- $range_i(v)=[\min_b pred_i(v,b), \max_b succ_i(v,b)]$

v hat Kanten zu allen Knoten in $N_i(v)$ für jedes $i \geq 0$ wobei

$$N_i(v)=\{ w \in range_i(v) \mid prefix_i(w) = prefix_i(v) \}$$

Resultat: Skip+ Graph

Skip+ Graph



$\Theta(\log n)$ Grad, $\Theta(\log n)$ Durchmesser, $\Theta(1)$ Expansion mit hoher W.keit

Skip+ Graph

- $range_i(v)$: v´s aktueller Range in Level i
- $N_i(v)$: v´s aktuelle Nachbarschaft in Level i
  (d.h. für alle $w \in N_i(v)$, $w \in range_i(v)$ und $prefix_i(w) = prefix_i(v)$)

Das Build-Skip Protokoll besteht aus 2 Aktionen.

- timeout: führt Regel 1a und 1b aus
- linearize: führt Regel 2 aus

Regeln 1a, 1b und 2 werden im folgenden präsentiert.

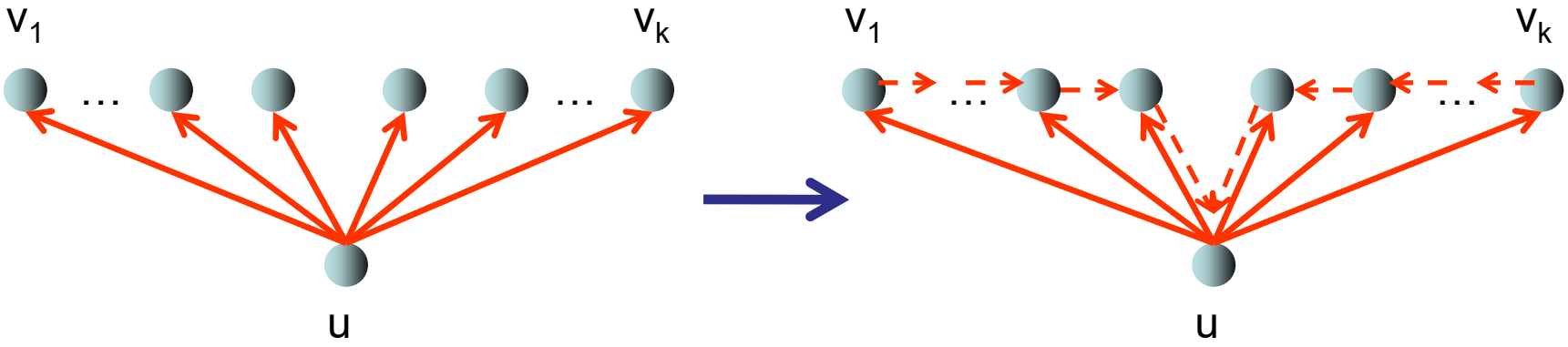
Skip+ Graph

- $range_i(v)$: v´s aktueller Range in Level i
- $N_i(v)$: v´s aktuelle Nachbarschaft in Level i
  (d.h. für alle $w \in N_i(v)$, $w \in range_i(v)$ und $prefix_i(w) = prefix_i(v)$)

Das Build-Skip Protokoll besteht aus 2 Aktionen.

Regel 1a: linearisiere

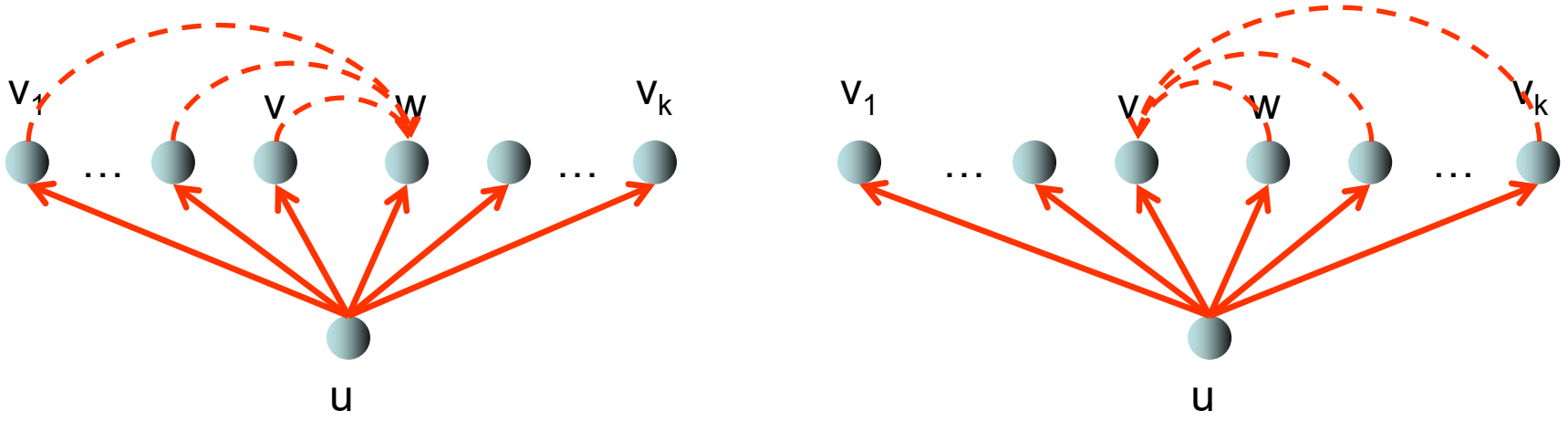
Für jeden Level i stellt jeder Knoten u periodisch alle Knoten in $N_i(u)$ in folgender Weise vor:



Skip+ Graph

Regel 1b: überbrücke

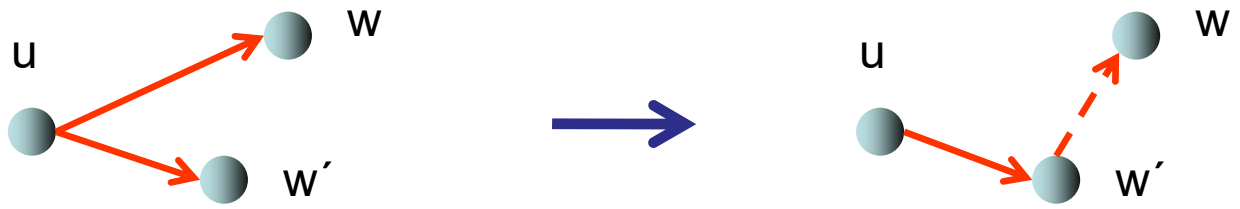
Für jeden Level i stellt jeder Knoten u periodisch seinen nächsten Vorgänger und Nachfolger den Knoten $v_j \in N_i(v)$ in folgender Weise vor wann immer aus u´s Sicht gilt, dass $v \in range_i(v_j)$ bzw. $w \in range_i(v_j)$.



Skip+ Graph

Regel 2: linearisiere

Bei Erhalt von v aktualisiert u seine Ranges und Nachbarschaften für jeden Level i. Für jeden Knoten w, den u nicht mehr benötigt (da er in keinem $range_i(u)$ ist), delegiert u w zu dem Knoten w´ in seiner neuen Nachbarschaft mit größter Präfixübereinstimmung zwischen w´.rs und w.rs, der am nächsten zu w ist.



Bemerkung:

Sei i der maximale Wert mit $prefix_i(u)=prefix_i(w)$. Dann muss gelten, dass $prefix_{i+1}(w)=prefix_{i+1}(w´)$ für den Knoten w´, zu dem w delegiert wird. Solch ein Knoten w´ existiert immer, da $w \notin range_i(u)$, und er muss zwischen u und w liegen, d.h. der ID-Bereich von (w´,w) ist innerhalb des ID-Bereichs von (u,w).

Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

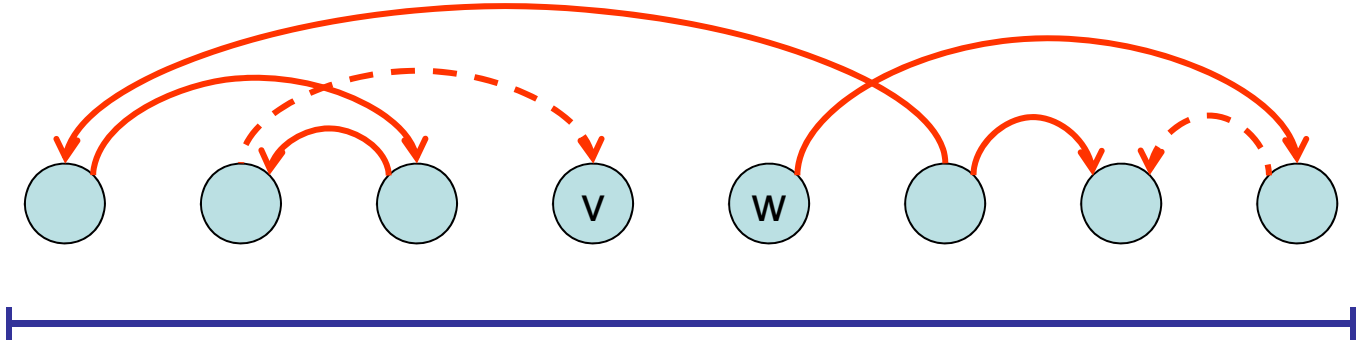
- Induktionsanfang: Build-Skip ordnet die Knoten in endlicher Zeit in einer sortierten Liste auf Level 0 an
  (zu zeigen: Konvergenz und Abgeschlossenheit bzgl. Level 0)
- Induktionsschritt:
  (a) Sobald sich die sortierten Listen in Level i gebildet haben, werden sich alle Skip+ Verbindungen in Level i bilden.
  (b) Sobald sich alle Skip+ Verbindungen in Level i gebildet haben, werden sich alle sortierten Listen in Level i+1 bilden.
  (zu zeigen: Konvergenz und Abgeschlossenheit bzgl. Level i+1)

Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

- Induktionsanfang: Build-Skip ordnet die Knoten in endlicher Zeit in einer sortierten Liste auf Level 0 an
  Wir können ähnliche Argumente wie für Build-List verwenden.



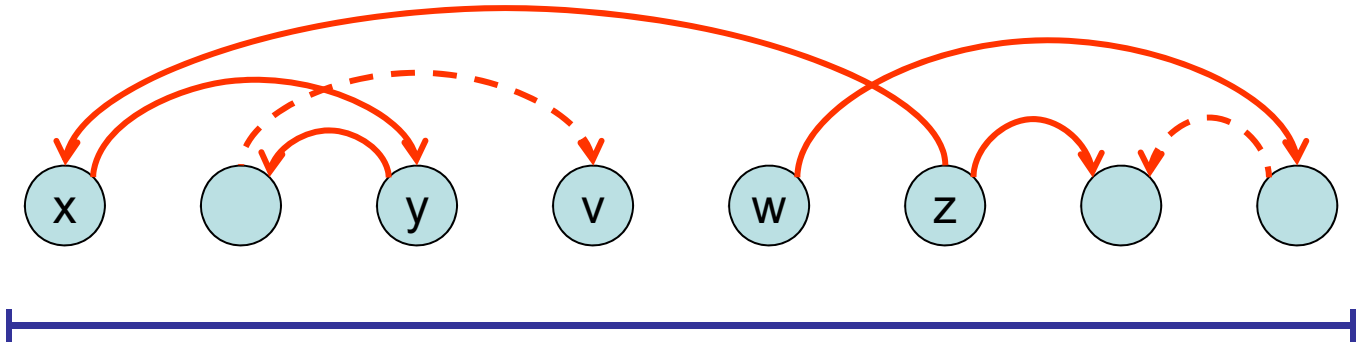
Bereich des Pfades von v nach w schrumpft über die Zeit

Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

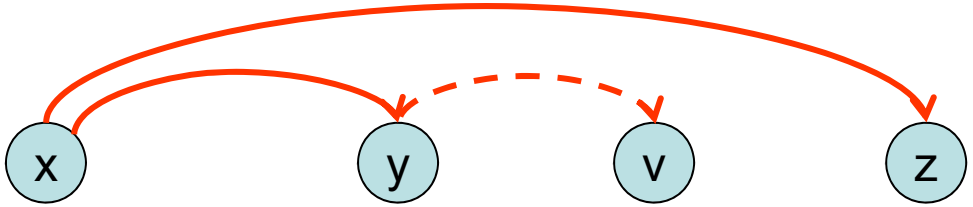
- Induktionsanfang: Build-Skip ordnet die Knoten in endlicher Zeit in einer sortierten Liste auf Level 0 an
  Betrachte als Beispiel den Fall, dass ein Randknoten x verbunden ist mit y und z.



Bereich des Pfades von v nach w schrumpft über die Zeit

Skip+ Graph

Betrachte als Beispiel den Fall, dass ein Randknoten x verbunden ist mit y und z.



- Angenommen, timeout wird bei Knoten x ausgeführt.
- Fall 1: Falls y und z im selben $N_i(x)$ sind, dann werden y und z durch Regel 1a verbunden sein, so dass wir x aus dem Pfad entfernen können.
- Fall 2: Sei i der maximale Wert, so dass $y \in N_i(x)$ und i´ der maximale Wert, so dass $z \in N_{i'}(x)$. O.B.d.A. nehmen wir an, dass $i<i'$.
- Dann muss $prefix_{i'}(x)=prefix_{i'}(z)$ sein, was bedeutet, dass es für alle $j \in \{i,\ldots,i'-1\}$ einen Knoten v zwischen y und z geben muss mit $v \in N_j(x)$, der auch in $N_{j+1}(x)$ ist.
- Also existiert insbesondere ein $v \in N_i(x)$ zwischen y und z , das auch in $N_{i+1}(x)$ ist. Dieses v wird nach Regel 1a mit y verbunden.
- Sei v das neue y. Falls $i=i'$, dann sind wir beim 1. Fall. Sonst fahren wir fort mit Fall 2. Aufgrund der Regel 1a müssen daher y und z mittels eine Pfades impliziter Kanten zwischen den Knoten y und z verbunden sein, so dass wir x aus dem Pfad entfernen können.

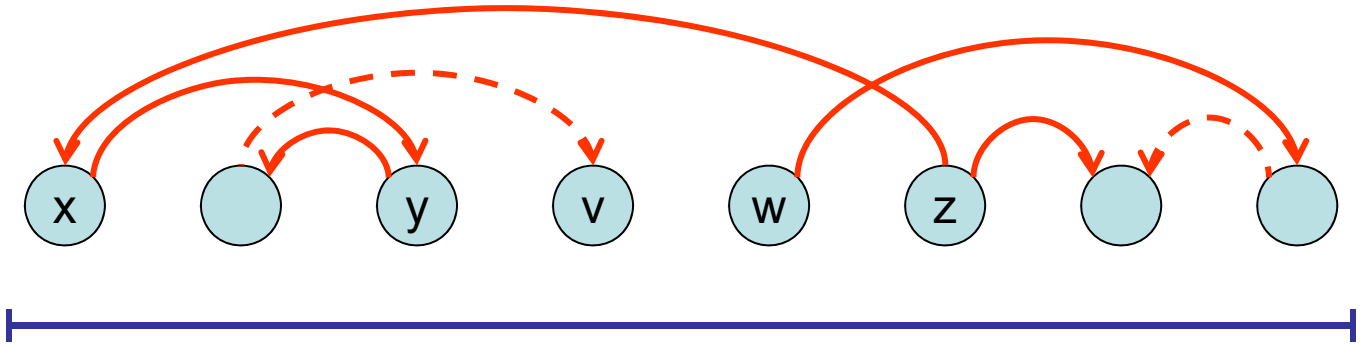
Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

- Induktionsanfang: Build-Skip ordnet die Knoten in endlicher Zeit in einer sortierten Liste auf Level 0 an

  Konvergenz: irgendwann sind v und w direkt verbunden.



Bereich des Pfades von v nach w schrumpft über die Zeit

Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

- Induktionsanfang: Build-Skip ordnet die Knoten in endlicher Zeit in einer sortierten Liste auf Level 0 an

  Konvergenz: irgendwann sind v und w direkt verbunden.

  Abgeschlossenheit: eine Kante auf Level 0 wird wie in sortierter Liste nur aufgelöst, wenn sich eine Kante zu einem näheren Knoten findet, was für v und w nicht möglich ist.

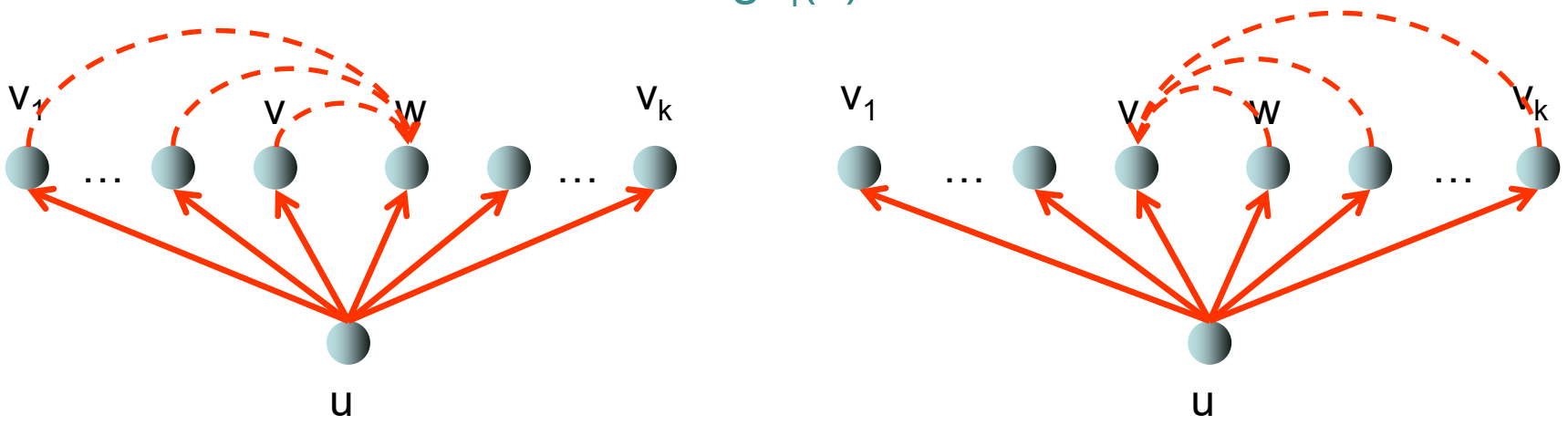
Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

- (a) Sobald sich die sortierten Listen in Level i gebildet haben, werden sich alle Skip+ Verbindungen in Level i bilden.

Wegen Regel 1b stellen sich die Knoten in Level i vor, bis jeder Knoten v alle Knoten in $range_i(v)$ kennt.

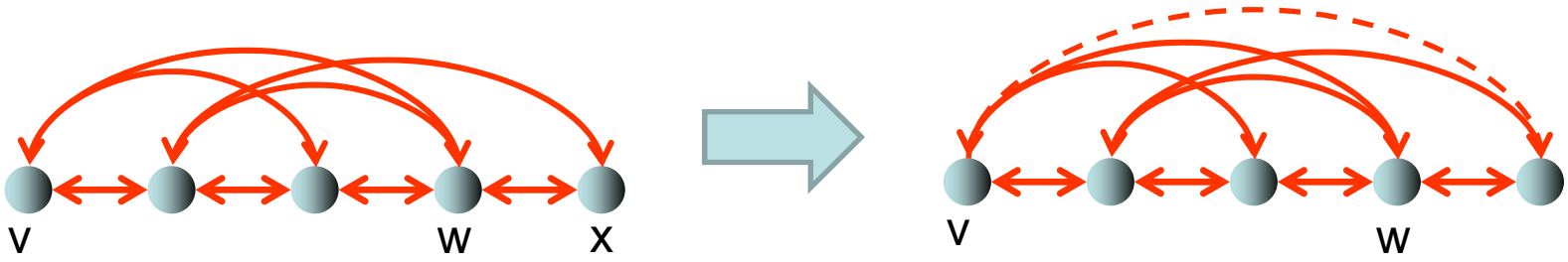


Skip+ Graph

Behauptung: Wegen Regel 1b stellen sich die Knoten in Level i vor bis jeder Knoten v alle Knoten in $range_i(v)$ kennt.

Beweisskizze:

- Sei w der aktuell rechteste Nachbar von v in einem Level i und v ein linker Nachbar von w.
- Wegen Regel 1b stellt w seinen nächsten Nachbar x in $N_i(w)$ Knoten v vor, so dass v $N_i(v)$ bei Bedarf erweitern kann.
- Also wird jeder Knoten v kontinuierlich neue Nachbarn von seinen linksten und rechtesten Nachbarn in Level i kennen lernen, bis er das korrekte $range_i(v)$ kennt.



Skip+ Graph

Satz 5.26 (Konvergenz): Build-Skip erzeugt aus einem beliebigen schwach zusammenhängenden Graphen $G=(V,E_L \cup E_M)$ einen Skip+ Graphen.

Beweis: durch Induktion

- (b) Sobald sich alle Skip+ Verbindungen in Level i gebildet haben, werden sich alle sortierten Listen in Level $i+1$ bilden.

Sobald jeder Knoten v alle Knoten in $range_i(v)$ kennt, folgt aus der Definition von $range_i(v)$, dass er auch seinen nächsten Vorgänger und Nachfolger in Level $i+1$ kennt, was den Beweis von Teil (b) abschließt.

Abgeschlossenheit: wie für Level 0 gilt diese nach Definition von Build-Skip auch für jede korrekte Level i Kante

Also sind am Ende alle Skip+ Kanten aufgebaut worden.

Skip+ Graph

Satz 5.27 (Abgeschlossenheit): Wenn die expliziten Kanten bereits den Skip+ Graphen formen, dann werden diese Kanten bei jedem Aufruf der Build-Skip Aktionen bewahrt.

Beweis:
folgt direkt aus dem Build-Skip Protokoll

Monotone Suchbarkeit: noch nicht erforscht

Operationen:

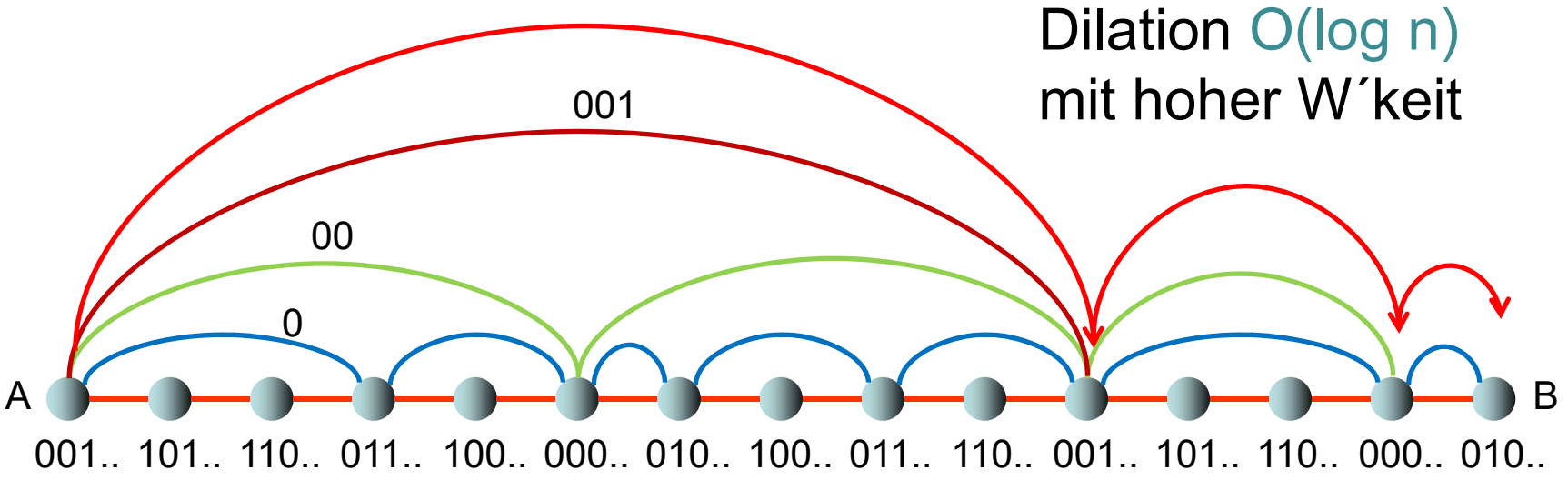
- Join(v): rufe einfach linearize(v) für einen Knoten u auf, der bereits im System ist
- Leave(v): einfache Strategie: entferne einfach v aus dem System

Satz 5.28: Jede Join oder Leave Operation im stabilisierten Skip+ Graphen benötigt mit hoher Wahrscheinlichkeit höchstens $O(\log^2 n)$ an zusätzlicher Arbeit (über die timeouts hinaus) um zu stabilisieren.

Beweis:
Übung

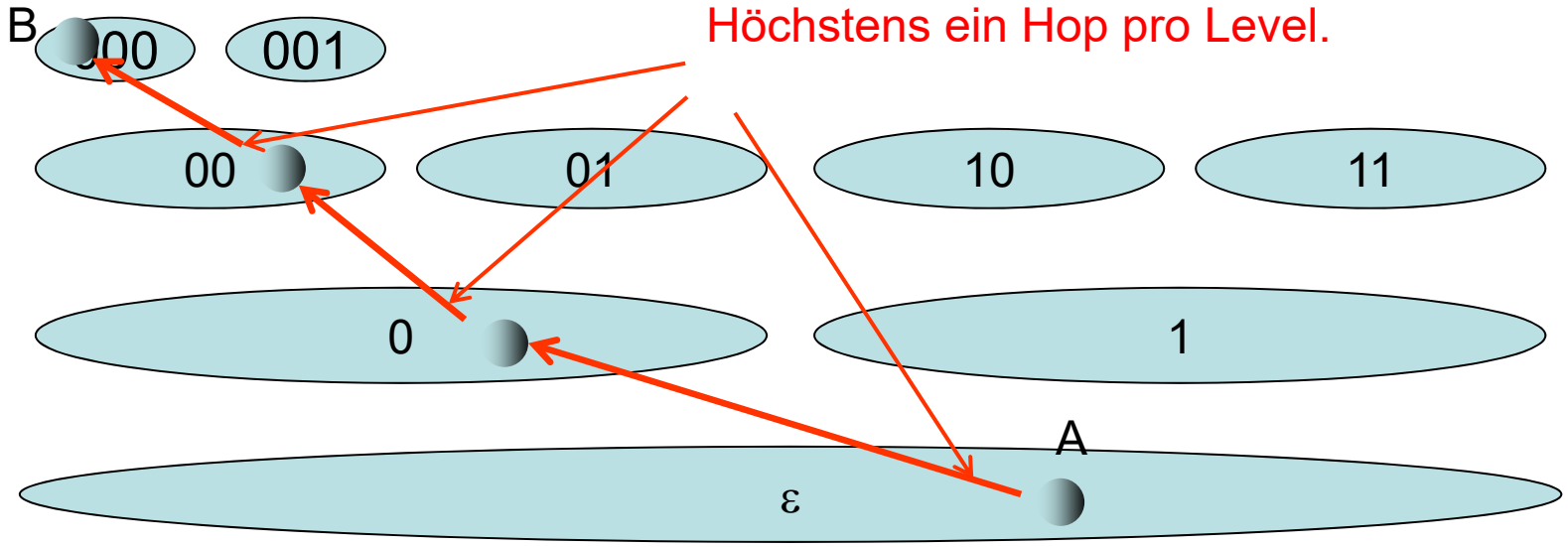
Oblivious Routing im Skip Graph

Search Operation (A kennt ID von B): wähle möglichst lange Kante in der Richtung des Ziels B (Greedy Routing)

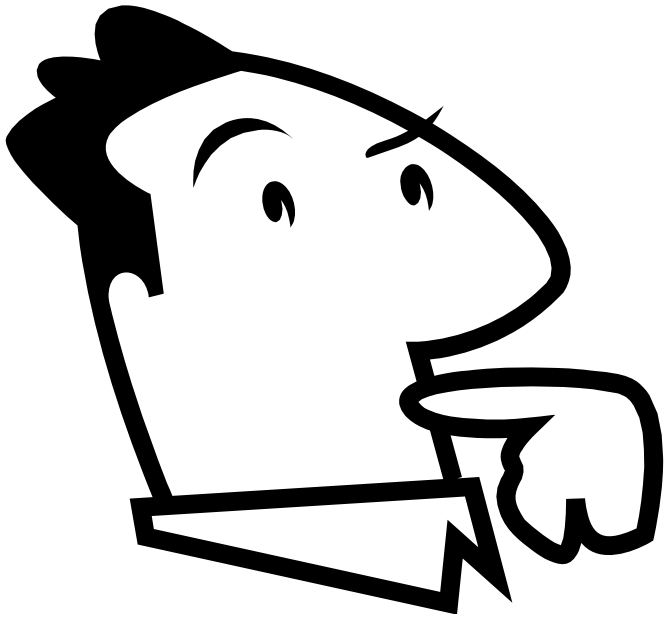


Skip+ Graph

Search Operation(A kennt r(B)): passe Bits eins nach dem anderen an r(B) an.



Anzahl Hops: O(log n) mit hoher W.keit



Fragen?