4. Architektursynthese

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

4.1 Bindung

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

4.1.1 Das Problem der Bindung

Definition

Eine **Bindung** eines Sequenzgraphens $G_S=(V_S,E_S)$ bzgl. eines Ressourcengraphen $G_R=(V_S \cup V_T,E_R)$ und einer Allokation $\alpha:V_T\rightarrow N_0$ ist ein Paar von Funktionen $\beta:V_S\rightarrow V_T$ und $\gamma:V_S\rightarrow N$ mit

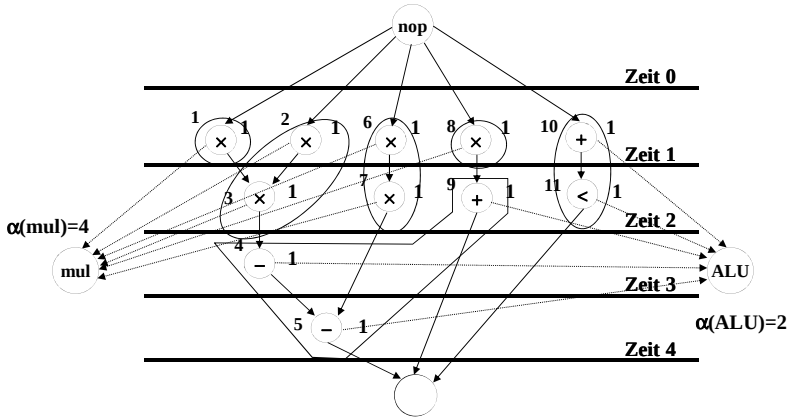
- $\forall v_s \in V_S$: $(v_s,\beta(v_s)) \in E_R$
- $\forall v_s \in V_S$: $\gamma(v_s) \leq \alpha(\beta(v_s))$

d.h. die Bindung ordnet jeder Aufgabe eine verfügbare Instanz einer Ressource zu, die diese Aufgabe ausführen kann.

Für einen legalen Ablaufplan muss gelten, dass zu jedem Zeitpunkt *t* jeder Ressourceninstanz nur eine Aufgabe zugeordnet ist.

Man kann das Problem, eine Bindung zu erzeugen, vor oder nach Vorliegen eines Ablaufplanes betrachten, oder durch Einsatz von ILP zusammen mit der Ablaufplanung.

Bindung: Beispiel



Ausführungslaufzeiten

Verträglichkeit von Aufgaben

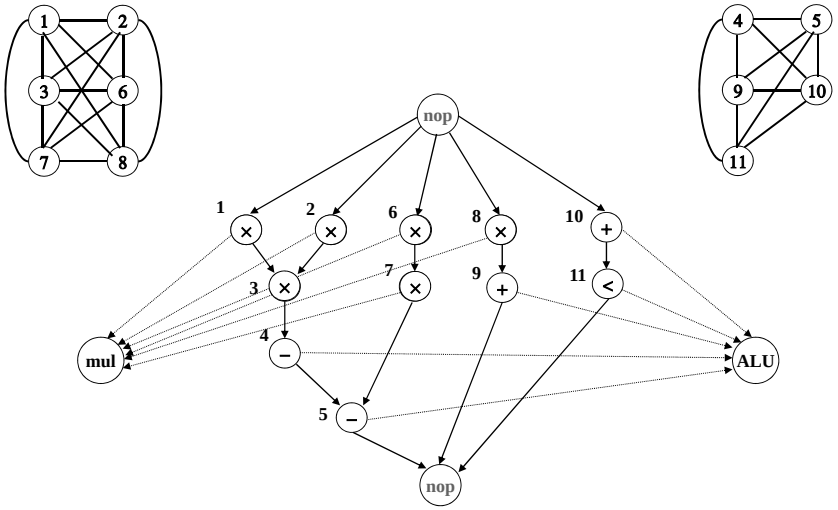
- Aufgaben können an ein und dieselbe Ressource gebunden werden, wenn
  - sie vom gleichen Ressourcentyp sind, und
  - sich die Ausführungszeiten nicht überlappen.
- Aufgaben können genau dann nebenläufig ausgeführt werden, wenn
  - sie nicht auf einem gerichteten Pfad im Sequenzgraph liegen.

Definition

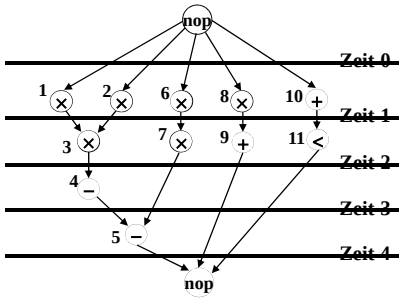
Ein Knotenpaar $(v_i,v_j) \in V_S$ heißt

- **schwach verträglich**, falls $\exists r_k \in V_R$ mit $(v_i,r_k) \in E_R$ und $(v_j,r_k) \in E_R$
- **ablaufplanverträglich**, falls $\exists r_k \in V_R$ mit $(v_i,r_k) \in E_R$ und $(v_j,r_k) \in E_R$ und $\tau(v_i) \geq \tau(v_j) + w(r_k)$ oder $\tau(v_j) \geq \tau(v_i) + w(r_k)$ gilt
- **stark verträglich**, falls das Paar schwach verträglich ist und es im Sequenzgraphen G_S ein Pfad von v_i nach v_j oder umgekehrt gibt.

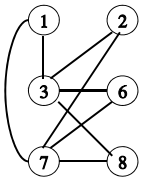
Beispiel zur schwachen Verträglichkeit



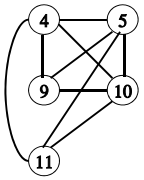
Beispiel zur Ablaufverträglichkeit



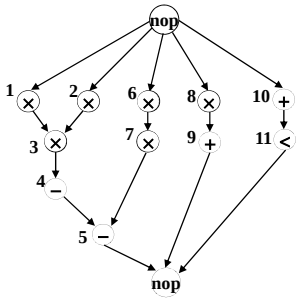
1 2
3 6
7 8



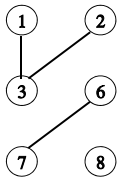
4 5
9 10
11



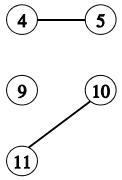
Beispiel zur starken Verträglichkeit



1 2
3 6
7 8



4 5
9 10
11



Beobachtungen

- Schwach verträgliche Knoten kann man grundsätzlich an ein und dieselbe Ressource binden. Allerdings sind dann ggf. nur noch sehr stark sequentielle Ablaufpläne unter dieser Bindung gültig.
  Will man die Kosten ohne Rücksicht auf Performanz minimieren, dann muss man das Problem lösen, eine Bindung unter schwacher Verträglichkeit mit minimaler Zahl von Ressourcen zu finden.
- Ablaufplanverträgliche Knoten kann man unter Beibehaltung der Gültigkeit eines gegebenen Ablaufplanes an ein und dieselbe Ressource binden.
  Hat man unter Performanzaspekten also einen Ablaufplan erstellt, so besteht eine Minimierung der Kosten für diesen Ablaufplan in dem Problem, eine Bindung unter Ablaufplanverträglichkeit mit minimaler Zahl von Ressourcen zu finden.
- Stark verträgliche Knoten kann man stets an ein und dieselbe Ressource binden, ohne die Gültigkeit von Ablaufplänen zu zerstören.
  Die Minimierung der Zahl der Ressourcen unter starker Verträglichkeit, liefert uns eine Kostenschranke, unter der sicher jeder legale Ablaufplan gültig bleibt.

Der Verträglichkeitsgraph

Definition

Der (Ressourcen-)**Verträglichkeitsgraph** $G_V=(V_S,E_V)$ ist ein ungerichteter Graph mit Kantenmenge E_V, der genau dann eine Kante zwischen zwei Knoten v_i und v_j enthält, wenn das Knotenpaar $\{v_i,v_j\}$ (schwach, ablauf-, stark) verträglich ist.

Beobachtungen

- Ist jeder Aufgabe genau ein Ressourcentyp zugeordnet, dann enthält ein Verträglichkeitsgraph mindestens $|V_T|$ unabhängige Mengen, d.h. mindestens so viele wie es Ressourcen-Typen gibt.
- Eine Menge gegenseitig verträglicher Aufgaben entspricht einer Clique, d.h. einem vollständigen Untergraphen, des Verträglichkeitsgraphen.
- Eine maximale Verträglichkeitsmenge entspricht einer maximalen Clique des Verträglichkeitsgraphen.

Wir können das Problem der Minimierung der Kosten auf das Problem, den Verträglichkeitsgraphen mit einer minimalen Zahl von Cliquen zu überdecken, zurückführen.

4.1.2 Kostenminimale Bindung

... das Problem einer kostenoptimalen Bindung ist mittels Clique-Cover ungerichteter Graphen lösbar.

Definition -- Clique-Cover

Gegeben: ungerichteter Graph $G_V=(V_S,E_V)$

Gesucht: minimale Partition $\{V_1,...,V_k\}$ von V_S, so dass für alle $j\in\{1,...,k\}$ der durch V_j induzierte Teilgraph von G_V eine Clique ist.

Anmerkung:

Clique-Cover ist NP-vollständig. Allerdings ist das Problem, zu einem gegebenen Verträglichkeitsgraphen eine kostenminimale Bindung zu finden bis auf einige Ausnahmen ebenfalls NP-vollständig. Da es zu Clique-Covering brauchbare Heuristiken gibt, ist diese Reduktion also durchaus legitim.

- ⊙ Berechne hinreichend viele große Cliquen (Maxclique Heuristik)
- ⊙ Löse damit ein **Überdeckungsproblem**

4.1.3 Knotenüberdeckungsprobleme

Motivation

Kennt man (alle oder hinreichend viele) Cliquen des Verträglichkeitsgraphen kann man eine kostengünstige Bindung dadurch bestimmen, dass man ein Knotenüberdeckungsproblem in folgendem Hypergraphen löst:

- Knoten: alle $v\in V_S$
- Kanten: alle betrachteten Cliquen G_c des Verträglichkeitsgraphen mit dem Rand $\rho(c)=\{v \mid v$ gehört zu $G_c\}$

Eine Clique überdeckt also stets alle Knoten, die zu ihr gehören. Ziel ist es nun, den Graphen mit möglichst wenig Cliquen zu überdecken.

Definitionen

Ein **Hypergraph** $H = (V,E,\rho)$ ist gegeben durch eine Menge V von Knoten (Punkten, Ecken, *vertices*), eine Menge E von (Hyper-)Kanten (*edges*) und eine Zuordnung

$$\rho : E \to 2^V$$

die jeder Kante e ihren Rand $\rho(e)$ zuordnet. Sei $c: E \to N_0$ eine Bewertung der Kanten. Dann nennen wir (H,c) einen bewerteten Hypergraphen

Knotenüberdeckungsproblem
Gegeben: ein bewerteter Hypergraph (H,c)
Gesucht: eine Kantenmenge $Y \subseteq E$ mit

(1) $\bigcup_{e \in Y} \rho(e) = V$ (Y überdeckt V)

(2) $c(Y) := \sum_{e \in Y} c(e)$ ist minimal über alle $Z \subseteq E$ mit (1)

Knotenüberdeckungsprobleme

Wir können nun zu diesem Optimierungsproblem auch das zugehörige Entscheidungsproblem betrachten:

MVC (Minimum Vertex Cover)

Gegeben sei ein bewerteter Hypergraph (H,c) und eine Konstante k.

Entscheide, ob es eine Kantenmenge Y gibt, mit Y überdeckt V und $c(Y) \leq k$.

Lemma -- vgl. Garey & Johnson: Computers and Intractability
Die Sprache

$$MVC = \{(H,c,k) \mid \exists Z \subseteq E(H): \bigcup_{e \in Z} \rho(e) = V(H) \text{ und } c(Z) \leq k\}$$

ist NP-vollständig.

Sie bleibt NP-vollständig, auch wenn H ein Graph und $\forall e: c(e) = 1$.

Wesentliche Knoten -- Reduktion erster Art

Es kann Knoten geben, die nur auf einer einzigen Kante liegen. Wir nennen solche Knoten **wesentliche Knoten** (essentials) und Kanten, auf denen ein wesentlicher Knoten liegt, **wesentliche Kanten**. Diese Kanten werden offensichtlich stets zu einer vollständigen Überdeckung der Knotenmenge benötigt, sind also Teil jeder Lösung des Überdeckungsproblems.

Definition

Sei $Ess(H) := \{ e \mid e \text{ wesentlich} \} \subseteq E$

Dann nennen wir $(H',c') := ((V',E',\rho'),c')$

reduziertes Überdeckungsproblem erster Art, wobei

$$V' = V \setminus \bigcup_{e \in Ess(H)} \rho(e), \quad E' = E \setminus Ess(H)$$
$$\rho'(e) = \rho(e) \cap V', \quad c' = c|_{E'}$$

Kantendominanz (Red. zweiter Art)

Wir können Kanten ausmachen, die offenbar wichtiger als andere Kanten sind, weil sie insbesondere alle ihre Knoten überdecken, und auch noch billiger sind:

Definition

Sei (H,c) ein Überdeckungsproblem. Eine Kante e **dominiert** eine Kante e'

$$:\Leftrightarrow \rho(e) \supseteq \rho(e') \text{ und } c(e) \leq c(e')$$

$(H',c') := ((V',E',\rho'),c')$ heißt **reduziertes Überdeckungsproblem zweiter Art** zu (H,c)

$$:\Leftrightarrow V' = V,\ E' = E \setminus \{e'\},\ \rho' = \rho|_{E'},\ c' = c|_{E'}$$

Knotendominanz (Red. dritter Art)

Wir können ebenso Knoten ausmachen, die offenbar wichtiger als andere Knoten sind, weil ihre Überdeckung die Überdeckung eines anderen Knotens automatisch sicherstellt:

Definition

Sei *(H,c)* ein Überdeckungsproblem. Ein Knoten *v* **dominiert** einen Knoten *v'*

$:\Leftrightarrow \{ e \mid v \in \rho(e) \} \subseteq \{ e \mid v' \in \rho(e) \}$ $(\Leftrightarrow \forall e \in E: v \in \rho(e) \Rightarrow v' \in \rho(e))$

(H',c') := ((V',E',ρ'),c') heißt **reduziertes Überdeckungsproblem dritter Art** zu *(H,c)*

$:\Leftrightarrow V' = V \setminus \{v'\}, E' = E, \rho' = \rho(e) \cap V', c' = c$

Reduzierte Überdeckungsprobleme

Satz

(1) ist *(H',c')* reduziertes Problem erster Art zu *(H,c),* dann ist zu jeder kostenminimalen Überdeckung *Y* von *H'* schon $Y \cup Ess(H)$ kostenminimale Überdeckung für *(H,c).*

(2) ist *(H',c')* reduziertes Problem zweiter Art zu *(H,c),* dann ist jede kostenminimale Überdeckung *Y* von *H'* schon kostenminimale Überdeckung für *(H,c).*

(3) ist *(H',c')* reduziertes Problem dritter Art zu *(H,c),* dann ist jede kostenminimale Überdeckung *Y* von *H'* schon kostenminimale Überdeckung für *(H,c).*

Wir nennen ein Überdeckungsproblem **irreduzibel** bzw. **zyklisch**, wenn keine Reduktion 1., 2. oder 3. Art anwendbar ist.

4.1.4 Petriks Methode

Idee:

Stelle zu *H = (V,E)* eine Schaltfunktion *covers:* $B^{\#E} \rightarrow B$ auf, die auf jedem Bitvektor 1 wird, der eine Kantenteilmenge kodiert, die *V* komplett überdeckt. Eine konjunktive Form für *covers* leitet sich direkt aus der Überdeckungsbedingung ab:

$$\forall v \in V\ \exists e \in Y : v \in \rho(e)$$
$$\Leftrightarrow \forall v \in V : \left(\bigvee_{v \in \rho(e)} e\right) = 1 \Leftrightarrow \boxed{\prod_{v \in V} \left(\bigvee_{v \in \rho(e)} e\right) = 1}$$

Beobachtung:
covers ist monoton in jeder Variablen $e \in E$. Also erhält man die Primimplikanten von *covers* durch Ausmultiplizieren und reduce by containment. Nun gilt für jeden Primimplikanten *c* von *covers*:

supp(c) ist eine irredundante Überdeckung von *V*

denn *supp(c)* entspricht gerade einem minimalen Punkt des On-Sets von *covers*. Macht man in diesem Punkt eine 1 zu 0, wird *covers* 0, d.h. die kodierte Teilmenge ist keine Überdeckung mehr.

Petriks Methode -- ff

Da eine optimale Überdeckung insbesondere irredundant ist, kann man nun alle irredundanten Überdeckungen bewerten und eine billigste davon nehmen.

Das Verfahren scheitert, wenn es sehr viele irredundate Überdeckungen gibt.

Beispiel:

	00001	00010	00100	00011	00110	00101	
000-1	x			x			a
00-01	x					x	b
0001-		x		x			c
00-10		x			x		d
001-0			x		x		e
0010-			x			x	f

$$\prod_{v \in V} \left(\bigvee_{v \in \rho(e)} e\right)$$
$$= (a \vee b)(c \vee d)(e \vee f)(a \vee c)(d \vee e)(b \vee f)$$
$$= \underbrace{(a \vee b)(a \vee c)}\underbrace{(c \vee d)(d \vee e)}\underbrace{(e \vee f)(b \vee f)}$$
$$cd \vee ce \vee d \vee de = d \vee ce$$
$$a \vee ac \vee ab \vee bc = a \vee bc$$
$$= (a \vee bc)(d \vee ce)(f \vee be)$$
$$= (ad \vee ace \vee bcd \vee bce)(f \vee be)$$
$$= adf \vee abde \vee acef \vee \cancel{abce} \vee bcdf \vee \cancel{bcde} \vee \cancel{bcef} \vee bce$$

Petriks Methode mit OBDDs

Stellt man *covers* statt durch eine dsijunktive Form durch ein OBDD (vgl. Script Logiksynthese) dar, so korrespondieren Überdeckungen zu Wegen von der Wurzel zur 1 Senke.

Gibt man einer von einem mit *e* markierten Knoten ausgehenden 1-Kante die Kosten *c(e)* und den 0-Kanten die Kosten 0, so findet man eine billigste Überdeckung durch die 1-Kanten entlang eines kürzesten Weges von der Wurzel zur 1 Senke.

Solch ein Weg lässt sich ggf. durch depth-first-search effizient bestimmen, selbst wenn es viele irredundante Überdeckungen gibt, das OBDD aber moderat in seiner Größe bleibt.

Das Verfahren scheitert wenn das OBDD zu covers zu groß wird.

4.1.5 Branch&Bound Methode

Branch & Bound ist ein allgemeines Paradigma, das man sehr häufig bei Optimierungsproblemen einsetzen kann,

- deren Suchraum sich gut durch einen Entscheidungsbaum strukturieren lässt, und
- bei denen man leicht untere Schranken für Teilräume des Suchraums berechnen kann.

Wir wollen nun dieses Paradigma zur Lösung unseres Überdeckungsproblems einsetzen:

Idee:

Branch: Teile den Suchraum durch Wahl einer geeigneten Kante *e* auf in (1) den Unterraum der nur aus Teilmengen $Y \subseteq E\backslash\{e\}$ besteht

(2) den Unterraum der nur aus $Y \subseteq E$ mit $e \in Y$ besteht.

Bound: Berechne für jeden dieser Unterräume eine untere Schranke. Ist diese größer als eine bisher gefundene Lösung, dann durchsuche den Unterraum nicht.

Branch & Bound -- Rahmenalgorithmus

```
find_min_sol (V,E,Z)
-- V: Knotenmenge, die noch zu überdecken ist
-- E: Kantenmenge, die noch zum Überdecken benutzt werden kann
-- (V,E) bildet demnach den Resthypergraphen
-- Z Menge der Kanten, die in jeder noch zu betrachtenden Lösung benutzt werden müssen.
-- global: best_sol beste bisher gefundene Lösung,
--         best_cost Kosten von best_sol
begin
    reduce(V,E,Z);  -- reduziere (V,E) und nehme ggf. wesentliche Kanten nach Z hinzu.
    if V == { } then if c(Z) < best_cost then best_cost = c(Z); best_sol = Z fi;
                return fi;
    lb = lower_bound(V,E);
    if c(Z) + lb >= best_cost then return
       else  wähle e ∈ E; -- (*)
             find_min_sol(V \ ρ(e), E \ {e}, Z ∪ {e}); -- Restproblem bei Nutzung von e
             find_min_sol(V, E \ {e}, Z); -- Restproblem bei Verzicht auf e
    fi;
end;
```

Branch&Bound Methode -- ff

Aufruf: Aufzurufen ist die Methode mit

```
best_cost = MAX_INT; best_sol = NIL; Z = { };
find_min_sol (V,E,Z);
```

Beobachtungen: Man erkennt leicht

Gute untere Schranken können die Berechnung stark beschleunigen:

Wann immer c(Z) + lb mindestens so groß wie die Kosten der besten bisher gefundenen Lösung ist, kann man sich das weitere Durchsuchen sparen, weil man sicher weiß, dass keine bessere Lösung mehr gefunden werden kann.

Das Verfahren findet schnell eine Lösung:

Wir haben den Rahmen so programmiert, dass zuerst in die Tiefe mit Hinzunahme von Kanten nach Z gesucht wird. Sobald alle Knoten überdeckt sind, hat man eine erste Lösung gefunden.

Wichtig: Man sollte möglichst schnell auch möglichst gute Lösungen finden, damit untere Schranken schneller greifen!

Noch Offen: (1) Wahl von *e*;

(2) Berechnung unterer Schranken

Der Branch Schritt: Wahl der Kante e

Wir gehen heuristisch vor:

Ziel: Wähle eine Kante, von der man hoffen darf, dass sie in guten Lösungen sicher enthalten ist. Betrachte dazu für jeden Knoten *v*

$$cp(v) := \frac{\#edges(v)}{\#E}$$

cp(v) ist also die Überdeckungswahrscheinlichkeit eines Knoten, zieht man zufällig eine Kante bei Gleichverteilung. Knoten mit kleinem *cp(v)* sind scheinbar schwieriger zu überdecken, als Knoten mit hohem *cp(v).* Betrachte nun $L(e) := \sum_{v \in \rho(e)} cp(v)$

L(e) gibt an wie "leicht" die Randknoten von e zu überdecken sind. Ist *L(e)* groß, dann ist auch die Chance groß, dass die Randknoten von e auch anders überdeckt werden können.

Heuristik: Bestimme in *E* alle *e* mit *L(e)* minimal, wähle daraus dann ein *e* mit minimalen Kosten.

Der Bound Schritt: untere Schranken für ((V,E),c)

Betrachte zum Restgraphen *(H = (V,E), c)* stets folgenden, ungerichteten Graphen *G(H) = (V,K)* mit der gleichen Knotenmenge *V* und der Kantenmenge

$$K := \{ \{s,t\} \mid edges_H(s) \cap edges_H(t) = \{ \} \}$$

für jede Kante $\{s,t\} \in K$ sei $\{s,t\}$ auch zugleich der Rand. Ferner habe *G(H)* folgende Knotengewichtung $weight: V \to N$

$$weight(v) := min \{ c(e) \mid e \in edges_H(v) \}$$

Es gilt nun

V' Clique :⇔ ∀v,w ∈ V': {v,w} ∈ K

Lemma:

Sei $V' \subseteq V$ eine Clique von *G(H),* dann ist für jede Knotenüberdeckung $Z \subseteq E$ des Graphen *H* schon

$$\sum_{v \in V'} weight(v) \leq c(Z)$$

Untere Schranken für ((V,E),c) -- Beweis

Sei *Z* eine beliebige Knotenüberdeckung von *(H,c).* Dann gilt für jede Clique *V'* insbesondere

$$V' \subseteq V = \bigcup_{e \in Z} \rho(e)$$

Für jedes Paar $v,t \in V'$ gibt es eine Kante $k = \{v,t\} \in K$, also gilt für jedes Paar $v,t \in V'$:

$$edges_H(v) \cap edges_H(t) = \{ \}$$

Damit gilt aber auch

$$(Z \cap edges_H(v)) \cap (Z \cap edges_H(t)) = \{ \}$$

für alle Paare *v,t*. Also ist unter disjunkter Vereinigung

$$Z \supseteq \bigcup_{v \in V'} (Z \cap edges_H(v))$$

und damit

$$c(Z) \geq \sum_{v \in V'} c(Z \cap edges_H(v)) \geq \sum_{v \in V'} weight(v)$$

4.1.6 Bestimmung einer maximalen Clique

Nach dem Lemma liefert also das Gewicht jeder Clique eine untere Schranke, große Cliquen liefern demnach große untere Schranken.

Umgekehrt möchten wir zur Konstruktion des Überdeckungsproblems selbst ja auch viele möglichst große Cliquen des Verträglichkeitsgraphen bestimmen. Also sind MaxClique Algorithmen in zweierlei Hinsicht für uns hier interessant.

Problem: Max Clique ist ebenfalls ein NP-vollständiges Problem!

Ausweg:

Wir nähern eine gute untere Schranke durch folgende Heuristik

Bestimmung einer maximalen Clique

Wir nähern eine gute untere Schranke durch folgende Heuristik

Max-Clique Heuristik
-- gegeben sei *G=(V,E,w)*, ungerichteter Graph mit Knotengewicht *w*
1. Bestimme D_{max} **:= { v | #nachbarn(v) maximal über V}**
2. Wähle aus D_{max} einen Knoten ***u*** mit **maximalem Gewicht**. ***VC := {u};***
3. ***Kand := nachbarn(u);*** -- Menge der Kandidaten zur Clique
4. **while** *Kand* $\neq \{\}$
   do berechne ***D := { t | t ∈ Kand und #(nachbarn(t) ∩ Kand) maximal}***
      wähle ***u*** $\in$ *D mit* ***maximalem Gewicht****;* ***VC = VC ∪ {u};***
      Kand = Kand ∩ nachbarn(u);
   od;

4.1.7 Eine duale Sicht auf das Bindungsproblem

Definition

Der (Ressourcen-)**Konfliktgraph** $G_K=(V_S,E_K)$ bezeichnet den ungerichteten Graphen mit Kantenmenge E_K, die genau dann eine Kante zwischen zwei Knoten v_i und v_j enthält, wenn das Knotenpaar $\{v_i,v_j\}$ nicht (schwach, ablauf-, stark) verträglich ist.

Beobachtungen

- Der Konfliktgraph ist quasi das Komplement des Verträglichkeitsgraphen.
- Eine Menge gegenseitig verträglicher Aufgaben entspricht einer unabhängigen Knotenmenge des Konfliktgraphen, d.h. einer Knotenmenge, die einen Teilgraphen von G_K induziert, der keine Kante enthält.
- Eine maximale Verträglichkeitsmenge entspricht einer maximalen unabhängigen Menge des Konfliktgraphen.
- Eine maximale unabhängige Menge des Konfliktgraphen entspricht einer maximalen Clique des Verträglichkeitsgraphen.

Kostenminimale Bindung durch Graphfärbung

... das Problem einer kostenoptimalen Bindung kann auf ein Graphfärbungsproblem zurückgeführt werden:

Definition -- Graph Coloring

Gegeben: ungerichteter Graph $G_K=(V_S,E_K)$

Gesucht: minimales $k\in$ N, so dass es eine Abbildung $\phi:V_S\rightarrow\{1,...,k\}$ gibt, mit $\phi(v_i)$ verschieden von $\phi(v_j)$, wenn $\{v_i,v_j\}\in E_K$ gilt.

Anmerkung
Graph Coloring ist NP-vollständig. Es gilt das entsprechend zu Clique Covering Bemerkte.

4.1.8 Kostenoptimale Bindung mit Ablaufplanung

Ablaufplanverträglichkeit liefert einen Konfliktgraphen, für den als Spezialfall das Färbungsproblem in polynomieller Zeit exakt gelöst werden kann, wenn zu jedem Knoten nur ein Ressourcentyp existiert, oder die Ressourcentypbindung schon vorgegeben ist:

Gegeben

- Sequenzgraph $G_S=(V_S,E_S)$ und Ressourcengraph $G_R=(V_S\cup V_T,E_R)$
- Ablaufplan $\tau:V_S\rightarrow N_0$
- Ressourcetypbindung $\beta:V_S\rightarrow V_T$

Gesucht

- Bindung $\gamma:V_S\rightarrow$ N,
  - ⊙ die je zwei Aufgaben v_i,v_j, deren Verarbeitungsintervalle nicht disjunkt sind, verschiedene Ressourcen zuordnet, und
  - ⊙ die unter dieser Bedingung die Anzahl $\sum_{v\in V_T}\alpha(v)$ der benutzten Ressourcen minimiert.

Intervallgraphen

Das Färbungsproblem lässt sich in diesem Spezialfall effizient lösen, weil der Konfliktgraph eine spezielle Struktur hat:

Definition

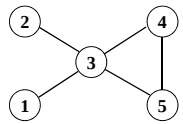
Ein ungerichteter Graph $G=(V,E)$ heißt **Intervallgraph**, wenn man jedem Knoten $v_i \in V$ ein Intervall $[u_i,o_i)$ mit $u_i<o_i$ zuordnen kann, so dass die Kante $\{v_i,v_j\}$ genau dann in E ist, wenn sich die beiden Intervalle $[u_i,o_i)$ und $[u_j,o_j)$ schneiden.

Jeder Aufgabe $v_s \in V_S$ wird durch die Ablaufplanung τ ein Zeitintervall $[\ \tau(v_s),\tau(v_s)+w(\beta(v_s))\ )$ zugeordnet. Also erhält man in einfacher Weise zu jedem Ressourcetyp $r \in V_T$ für die Menge aller Knoten v, mit $\beta(v) = r$ den Konfliktgraphen als Intervallgraphen.

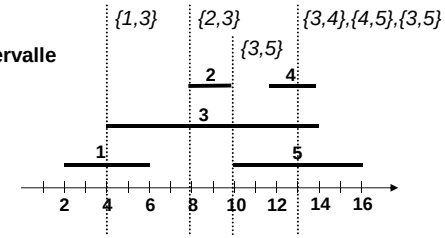
Da die Kanten implizit über die Intervalle definiert sind, genügt es, die Intervalle zu kennen, und darauf einen optimalen Färbungsalgorithmus zu entwickeln.

Intervallgraph: Beispiel

Intervallgraph



mögliche, dazugehörige Intervalle



Graph Coloring -- Der left edge Algorithmus

Folgender Algorithmus löst das Färbungsproblem effizient.
Voraussetzungen:

- Die Intervalle zu $u \in V$ sind aufsteigend sortiert nach *left(u)* gegeben,

$$V = \{u_1,\cdots,u_n\}; \forall 1 \le i < n : left(u_i) \le left(u_{i+1})$$

- Ferner kennen wir die **Dichte** *d* des Problems, die gegeben ist als die maximale Zahl von Intervallen, die sich zu einem Zeitpunkt *t* schneiden.

Dies lässt sich durch Sortieren der Intervallmenge und anschließendes, sequentielles Durchmustern leicht berechnen.

Wir fassen nun die Farben als Spuren auf, auf denen die Intervalle nichtüberlappend zu platzieren sind:

- In einem Array *int* free[i] halten wir die erste freie Position auf Spur *i*.
- In einer Priority Queue *pr_queue<int>* Tracks(p) halten wir die Menge aller Spuren mit der Priorität *p(i) := - free[i];* d.h. die am weitesten links freie Spur hat höchste Priorität.

left edge Algorithmus -- Schleife

Der Algorithmus:

```
-- Initialisierung
for i=0,i++,i<d do free[i] = 0; Tracks.add(i); od;
-- Hauptschleife
for  i=0,i++,i<n
do      u = Nets[i];
        s = Tracks.getmax();
        C[i] = s;
        free[s] = u.right;
        Tracks.add(s);
od;
```

Klar ist, dass die Dichte *d* eine untere Schranke für die chromatische Zahl des Konfliktgraphens ist, und damit wäre die Zahl der benutzten Farben minimal.
Liefert die so berechnete Zuweisung aber auch eine korrekte Färbung *C*?

Wir zeigen, dass in der Schleife folgende Invariante nach jedem Durchlauf *i* gilt:

$$\forall j < i : C(u_j) = C(u_i) \Rightarrow right(u_j) \le left(u_i)$$

Zu Beginn, für $i = 0$, gilt die Invariante, da kein $j < i$ existiert.

left edge Algorithmus -- Korrektheit

Annahme: Im *i*-ten Durchlauf wird die Invariante erstmals verletzt. Sei *s* die zugeordnete Farbe und sei *j(s) < i* mit

$$C(u_{j(s)}) = s = C(u_i) \wedge right(u_{j(s)}) > left(u_i)$$

Da *s* maximale Priorität hatte, gilt

$$\forall s' \neq s : -free[s] \geq -free[s']$$
$$\Leftrightarrow \forall s' \neq s : free[s] \leq free[s']$$

Sei *j(t)* der Index des *t* zuletzt zugewiesenen Intervalls. Dieser existiert für jedes *t*, da *free[s] > 0*, aber *-free[s]* maximal war. Damit ist für alle *t*

$$j(t) < i \wedge free[t] = right(u_{j(t)}) \geq free[s] = right(u_{j(s)})$$

Also gilt für alle *d* Spuren *t* wegen *j(t) < i*

$$left(u_{j(t)}) \leq left(u_i) < right(u_{j(s)}) \leq right(u_{j(t)})$$

und damit ist die Dichte größer als *d.* $\lightning$

left edge Algorithmus -- alternativ

Folgerung:
Der Konfliktgraph zu einem Ressourcentyp *r* lässt sich in *O(#V log d)* in Dichte vielen Farben optimal färben.

Anmerkung:
Das Sortieren der Intervalle braucht ohnehin Zeit *O(#V log #V).* Deshalb findet man in der Literatur auch häufig eine alternative "Urform" des Left egde Algorithmus, die Zeit *O(#V log #V)* braucht, verwaltet man die Intervalle in einem balancierten Baum *Q* nach *left(u)* geordnet. Diese ist asymptotisch nicht schlechter, aber praktisch.

Q = Menge der Intervalle;
for i=1,i++,i<=d
do pos = 0;
  while (es gibt u in Q mit u.left ≥ pos)
  do u = Q.get_smallest_greater_or_equal_than(pos);
    C[u] = i; pos = u.right;
  od;
od;

Ur – left edge Algorithmus am Beispiel

Sorry!
macht nur mit Animation einen Sinn

track 1

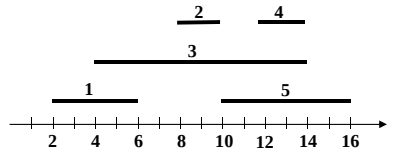
track 2

track 3

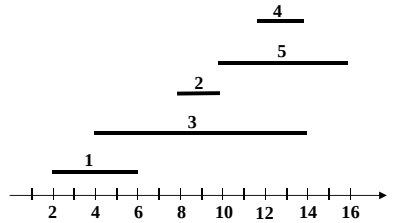
track 4

Intervallgraph-Coloring: weiteres Beispiel

Dazugehörige Intervalle

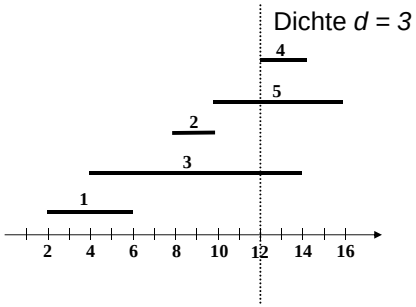


geordnete Intervalle

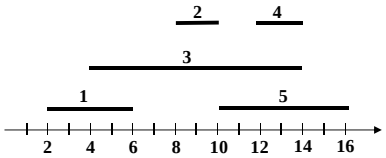


Intervallgraph: Graph-Coloring Beispiel

Sortierte Intervalle

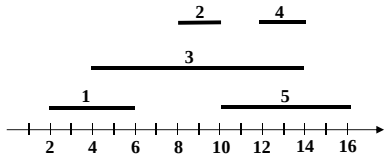


Färbung nach left edge

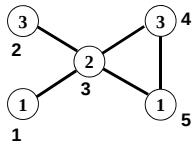


Intervallgraph: Graph-Coloring Beispiel

Färbung nach left edge



Färbung des Intervallgraphen



4.1.9 Bindung unter periodischer Ablaufplanung

Auch im Falle eines periodischen Ablaufplans liefert die Ablaufplan-verträglichkeit einen speziellen Konfliktgraphen. Allerdings ist für diesen das Färbungsproblem schon NP-vollständig.

Gegeben

- Iterativer Problemgraph $G_S=(V_S,E_S,s)$ und Ressourcengraph $G_R=(V_S \cup V_T,E_R)$
- Ein gültiger periodischer Ablaufplan $t:V_S \rightarrow N_0$ mit Periode P unter statischer Bindung
- Ressourcetypbindung $\beta:V_S \rightarrow V_T$

Gesucht

- Bindung $\gamma:V_S \rightarrow N$,
  - ⊙ die je zwei Aufgaben v_i,v_j, deren Verarbeitungsintervalle nicht disjunkt sind, verschiedene Ressourcen zuordnet, und
  - ⊙ die unter dieser Bedingung die Anzahl $\sum_{v \in V_T} \alpha(v)$ der benutzten Ressourcen minimiert.

Periodische Intervalle

Wir müssen zunächst einmal den Konfliktgraphen konstruieren.

Betrachtet man eine Aufgabe v des iterativen Problemgraphens, so ergeben sich unter dem Ablaufplan t unendlich viele Bearbeitungs-intervalle $[t(v)+n*P, t(v) + w(\beta(v)) + n*P), n \in \mathbf{N}_0$. Diese werden definiert durch das Tripel $(P,t(v),w(\beta(v)))$. Wir nennen $(P,t(v),w(\beta(v)))$ auch das **periodische Intervall** von v.

Wenn sich nun zwei Instanzen von Bearbeitungsintervallen zweier Aufgaben u,v schneiden, dann gilt

entweder $(1)\ t(u) + n*P \in [t(v)+k*P, t(v) + w(\beta(v)) + k*P), n,k \in \mathbf{N}_0$

oder $(2)\ t(v) + k*P \in [t(u)+n*P, t(u) + w(\beta(u)) + n*P), n,k \in \mathbf{N}_0$

also $(1)\ t(v) + (k-n)P \leq t(u) < t(v) + w(\beta(v)) + (k-n)*P$

oder $(2)\ t(u) + (n-k)P \leq t(v) < t(u) + w(\beta(u)) + (n-k)*P$

da $t(x),w(\beta(x)) < P$ für alle Knoten x sein muss, gilt ferner

$-1 \leq (k-n) < 1$ oder $-1 \leq (n-k) < 1$

Periodische Intervalle -- ff

Es bleiben also nur folgende Fälle:

a. Für $(k\text{-}n) = 0$ schneiden sich die Intervalle

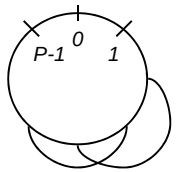
$$[t(v),\ t(v) + w(\beta(v))\ ) \text{ und } [t(u),\ t(u) + w(\beta(u))\ )$$

b. Für $(k\text{-}n) = -1$ gilt $t(u) < t(v) + w(\beta(v)) - P$

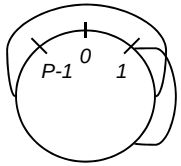
c. Für $(n\text{-}k) = -1$ gilt $t(v) < t(u) + w(\beta(u)) - P$

Trägt man die Paare $(t(u),\ t(u)+w(\beta(u)) \bmod P)$ auf einem Kreis im Uhrzeigersinn ab, dann bedeutet dies, dass sich die Ausführungszeitintervalle von u und v genau dann schneiden, wenn sich die Kreissegmente im Uhrzeigersinn überlappen:

Fall a.



Fall b,c.



Graphen mit zirkularen Kanten

Der Konfliktgraph hat auch hier eine spezielle Struktur:

Definition

Ein ungerichteter Graph $G=(V,E)$ heißt **Graph mit zirkularen Kanten**, wenn man jedem Knoten $v_i \in V$ ein Kreissegment $[l_i,r_i)$ zuordnen kann, so dass die Kante $\{v_i,v_j\}$ genau dann in E ist, wenn sich die beiden Segmente $[l_i,r_i)$ und $[l_j,r_j)$ schneiden.

Da die Kanten wieder implizit über die zirkularen Segmente definiert sind, genügt es, die Segmente zu kennen, und darauf einen Färbungsalgorithmus zu entwickeln.

Die Dichte $d(t)$ an der Stelle t für $0 \le t < P$ ist definiert als die Zahl der Segmente, die t enthalten. Ein Segment enthält t offenbar genau dann, wenn $l_i \le t < r_i$ für $l_i < r_i$ oder $t < r_i \le l_i$ bzw. $r_i \le l_i \le t$.

Es sei $d_{min} := min\{\ d(t) \mid 0 \le t < P\}$ und $d_{max} := max\{\ d(t) \mid 0 \le t < P\}$

Folgende Heuristik färbt nun einen Graphen mit zirkularen Kanten in höchstens $2*d_{max}$ Farben, deren Zahl mindestens d_{max} sein muss:

Der SORT & MATCH Algorithmus

Algorithmus -- SORT & MATCH

1. Bestimme d_{min}, d_{max} und sowie ein t mit $d_{min} = d(t)$ und die Menge V_A aller v, mit t liegt auf dem Segment $(l(v),r(v))$.
2. Überführe alle Segmente für $v \in V_B := V \setminus V_A$, in
   $l'(v) = (l(v) + P - t) \bmod P,\ r'(v) = (r(v) + P - t) \bmod P$.
3. Färbe die Knoten aus V_B nach dem left edge Algorithmus.
   -- Dies ist möglich, weil für $v \in V_B$ stets l'(v) < r'(v) ist, d.h die Segmente bilden wieder Intervalle.
4. Bestimme eine möglichst große Teilmenge aus V_A, die mit bisher vergebenen Farben gefärbt werden kann.
5. Gebe jedem anderen Knoten aus V_A eine noch nicht vergebene Farbe.

Durch die Transformation im Schritt 2 legen wir den Nullpunkt des Kreises auf t. Alle Segmente mit $l'(v) > r'(v)$ enthalten diesen Nullpunkt, korrespondieren also zu Knoten aus V_A.

Der SORT & MATCH Algorithmus -- ff

Anmerkungen:

Die Auswahl von V_A im Schritt 1 sorgt dafür, dass $\#V_A = d_{min}$,
Der left edge Algorithmus garantiert uns ferner eine Färbung des von V_B aufgespannten Untergraphens in höchstens d_{max} Farben.
Die Auswahl im Schritt 4 ist für die Analyse nicht kritisch, sie kann rein heuristisch durchgeführt werden. Sie dient nur zur pragmatischen Reduktion der Zahl der benötigten Farben.
Schritt 4 und 5 zusammen stellen sicher, dass selbst wenn man in Schritt 4 keine Teilmenge findet, höchstens $\#V_A = d_{min}$ weitere Farben benötigt werden.
Der Algorithmus braucht daher höchstens

$$d_{max} + d_{min} \le 2*d_{max}$$

Farben.
Da d_{max} eine unterer Schranke für die chromatische Zahl des Konfliktgraphens ist, ist SORT&MATCH ein heuristisches Verfahren, das höchstens um den Faktor 2 schlechter ist als ein optimales Verfahren.

4.1.10 Minimierung der Zahl der Register

Der left edge Algorithmus im nicht periodischen Fall sowie der Sort&Match Algorithmus im periodischen Fall können auch eingesetzt werden, um die Bindung der Kanten des Problemgraphens an Register zu optimieren:

Gegeben

- Sequenzgraph $G=(V,E)$ oder ein periodischer Problemgraph $G=(V,E,s)$ mit $s(e) \leq 1$ (sonst braucht man echte Pufferspeicher)
- Ablaufplan $\tau: V \rightarrow N_0$ bzw. $t: V \rightarrow N_0$

Gesucht

- Registerbindung $\rho: V \rightarrow N$,
  - ⊙ die jeder Aufgabe v ein Ergebnisregister so zuordnet, dass keine Aufgabe u mit $\rho(u) = \rho(v)$ endet, solange eine Aufgabe, die das Ergebnis von v braucht, noch nicht gestartet ist, und
  - ⊙ die unter dieser Bedingung die Anzahl der benutzten Register minimiert.

Registerzahlminimierung in Sequenzgraphen

Im Falle eines gerichteten azyklischen Sequenzgraphen kann man die Registerminimierung direkt auf einen Intervallgraphen zurückführen und mit dem left edge Algorithmus durchführen:

Gegeben sei

- ⊙ Sequenzgraph $G=(V,E)$
- ⊙ Ablaufplan $\tau: V \rightarrow N_0$

Lebensdauer

Für jede Operation u sei die Lebensdauer $d(u)$ gegeben durch

$$d(u) := max\{\tau(v) | e=(u,v) \in E\} - (\tau(u) + w(u))$$

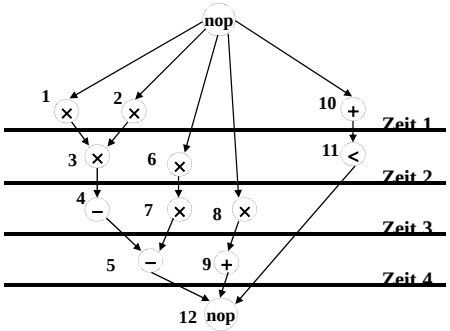
Dann wird das Ergebnis der Operation u benötigt im **Lebenszeitintervall**

$$[\tau(u) + w(u),\ \tau(u) + w(u) + d(u)+1)$$

Über dieses Intervall darf keine Operation v, deren Ergebnis dort entsteht, an dasselbe Register gebunden werden.

Der Konfliktgraph ist also ein Intervallgraph.

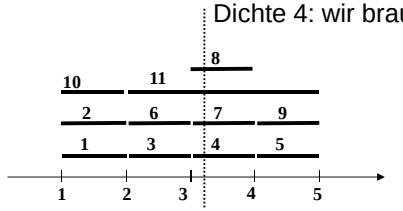
Registerzahlminimierung: Beispiel 1



Wir betrachten folgenden Ablaufplan unseres DGL-Lösers:
Lebenszeitintervalle:

1: [1,2)	2: [1,2)
3: [2,3)	4: [3,4)
5: [4,5)	6: [2,3)
7: [3,4)	8: [3,4)
9: [4,5)	10: [1,2)
11: [2,5)	

Dichte 4: wir brauchen 4 Register



Die Quellen und Senken stehen nun aber auch für Register, von denen Argumente kommen bzw. Resultate aufgenommen werden. Will man diese ebenfalls zuordnen, genügt es nicht mehr, nur einen Quell- bzw. Zielknoten zu betrachten.

Minimierung in periodischen Problemen

Im Falle eines periodischen Problemgraphens kann man die Registerminimierung direkt auf einen Graphen mit zirkularen Kanten zurückführen und mit dem SORT&MATCH-Algorithmus durchführen:

Gegeben sei

- periodischer Problemgraph $G=(V,E,s)$, $s(e) \leq 1$
- periodischer Ablaufplan $t: V \rightarrow N_0$

Lebensdauer:

Für jede Operation u sei nun das „Lebenszeitintervall“ gegeben durch das Tupel

$$(t(u) + w(u),\ max\{t(v) | (u,v) \in E\} + 1)$$

falls für alle $(u,v) \in E$ gilt $t(v) \geq t(u) + w(u)$.

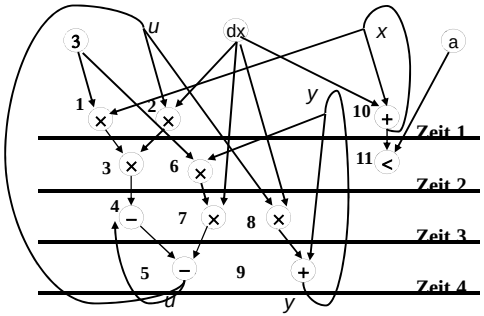
$$(t(u) + w(u),\ max\{t(v) | (u,v) \in E \text{ und } t(v) < t(u) + w(u)\} + 1)$$

falls es $(u,v) \in E$ gibt, mit $t(v) < t(u) + w(u)$.

Der Konfliktgraph ist nun ein Graph mit zirkularen Kanten.

!! Knoten, deren Intervall $(t(u)+w(u),\ t(u)+w(u))$ ist, stehen für Aufgaben, die über die ganze Iteration gebraucht werden. Sie belegen stets ein Register für sich.

Registerzahlminimierung: Beispiel 2

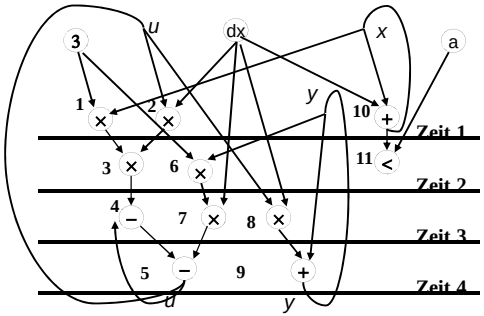


Lebenszeitintervalle:

1: (1,2)	2: (1,2)
3: (2,3)	4: (3,4)
5 = u: (4,3)	6: (2,3)
7: (3,4)	8: (3,4)
9 = y: (4,4)	10=x: (1,1)

Wir betrachten folgenden, periodischen Ablaufplan unseres DGL-Lösers: Er enthält 3 Quellen, *3, dx* und *a*, die wir fest an Argumentregister binden, weil sie in allen Iterationen gebraucht werden. Die Senke 11 binden wir ebenfalls fest, weil sie in die Kontrolle geht. Für die restlichen Knoten ist eine Bindung zu bestimmen.

Registerzahlminimierung: Beispiel 2

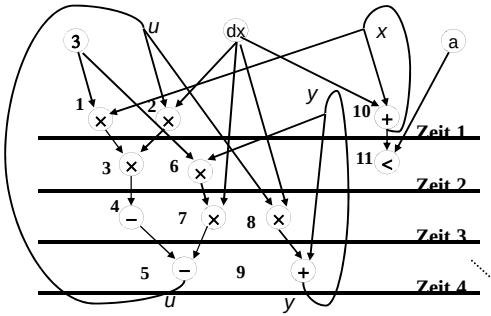


Lebenszeitintervalle:

1: (1,2)	2: (1,2)
3: (2,3)	4: (3,4)
5 = u: (4,3)	6: (2,3)
7: (3,4)	8: (3,4)
9 = y: (4,4)	10=x: (1,1)

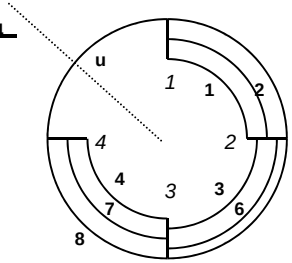
Wir beobachten ferner, dass die Lebenszeitintervalle zu *x,y* zyklisch den gesamten Zeitraum ((4,4), (1,1)) überspannen. Sie müssen also ebenfalls fest an ein Register gebunden werden. Die restlichen Intervalle ergeben folgenden Graph mit zirkularen Kanten:

Registerzahlminimierung: Beispiel 2



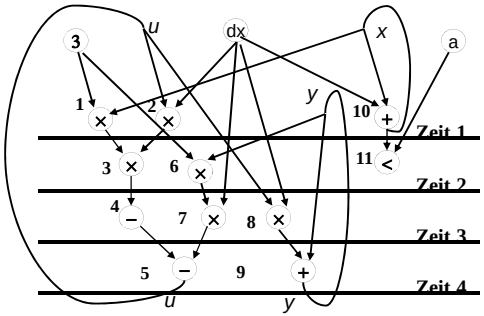
Lebenszeitintervalle:

1: (1,2)	2: (1,2)
3: (2,3)	4: (3,4)
5 = u: (4,3)	6: (2,3)
7: (3,4)	8: (3,4)



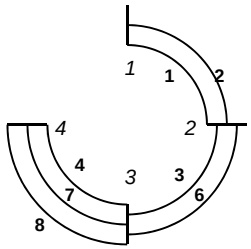
Die niedrigste Dichte haben wir zwischen *4* und *1*, also schneiden wir den Zyklus dort auf.
Der Graph zerfällt nach Herausnahme von *u* in einen Intervallgraphen, den wir mit left edge färben:

Registerzahlminimierung: Beispiel 2

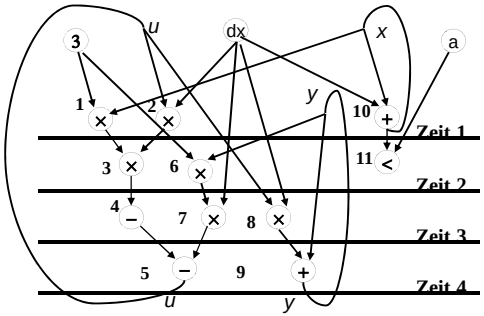


Lebenszeitintervalle:

1: (1,2)	2: (1,2)
3: (2,3)	4: (3,4)
5 = u: (4,3)	6: (2,3)
7: (3,4)	8: (3,4)

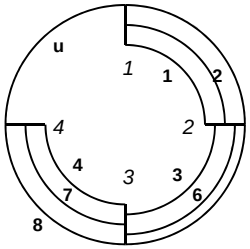


Registerzahlminimierung: Beispiel 2



Lebenszeitintervalle:

1: (1,2)	2: (1,2)
3: (2,3)	4: (3,4)
5 = u: (4,3)	6: (2,3)
7: (3,4)	8: (3,4)



left edge braucht 3 Farben, da wir im Abschnitt (3,4) die maximale Dichte von 3 haben.
Da im zyklischen Intervall von *u* (4,3) die rote Farbe nicht benötigt wird, können wir *u* mit der vorhandenen Farbe färben. **Ergebnis:** 3 Register