Inhalt

Maschinenabhängige Übersetzung

Kernaufgaben

- Befehlsauswahl
- Befehlsanordnung
- Registerzuteilung

Problem: Diese Aufgaben beeinflussen sich gegenseitig.

▶ Phasenkopplungsproblem

Zusätzlich:

- Spezial-Optimierungen für die jeweilige Zielarchitektur
- Implementierung der Konventionen der Laufzeitumgebung
- Implementierung komplexer Hochsprachen-Elemente (z.B. Vererbung)

Übersetzung in SSA-Form

Erhalte SSA im Backend

- Einheitliche Programmrepräsentation im Übersetzer
- Optimierungen aus der Optimierungsphase wiederverwendbar

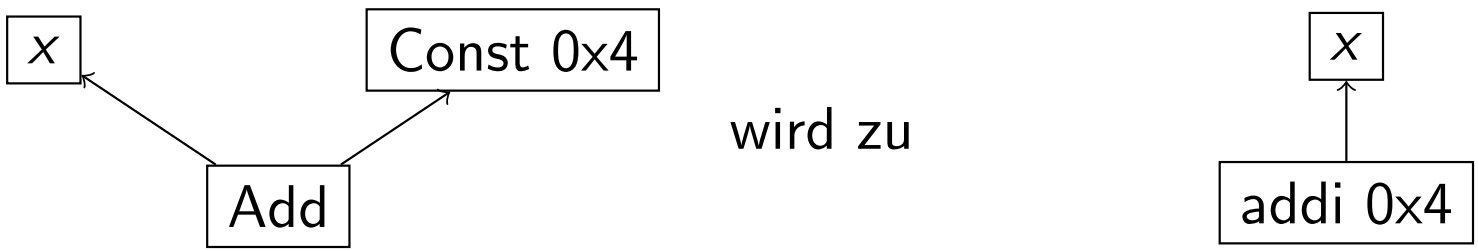
Ablauf der Übersetzung

- Befehlsauswahl
- Anordnung
- Registerzuteilung
- Nach-Anordnung im Rahmen der Registerzuteilung
- SSA-Abbau

Inhalt

Befehlsauswahl – Aufgabe

- Graphtransformation des Zwischensprachengraphs in einen Codegraph
- Muster aus Zwischensprachenecken werden in eine oder mehrere Codeecken überführt



Methoden

Anforderungen

- SSA-Eigenschaft muss erhalten bleiben
- Φ-Funktionen werden nicht angetastet
- Befehle haben *ein* Ergebnis (ggf. Projektionen aus Tupeln nötig)

Problem

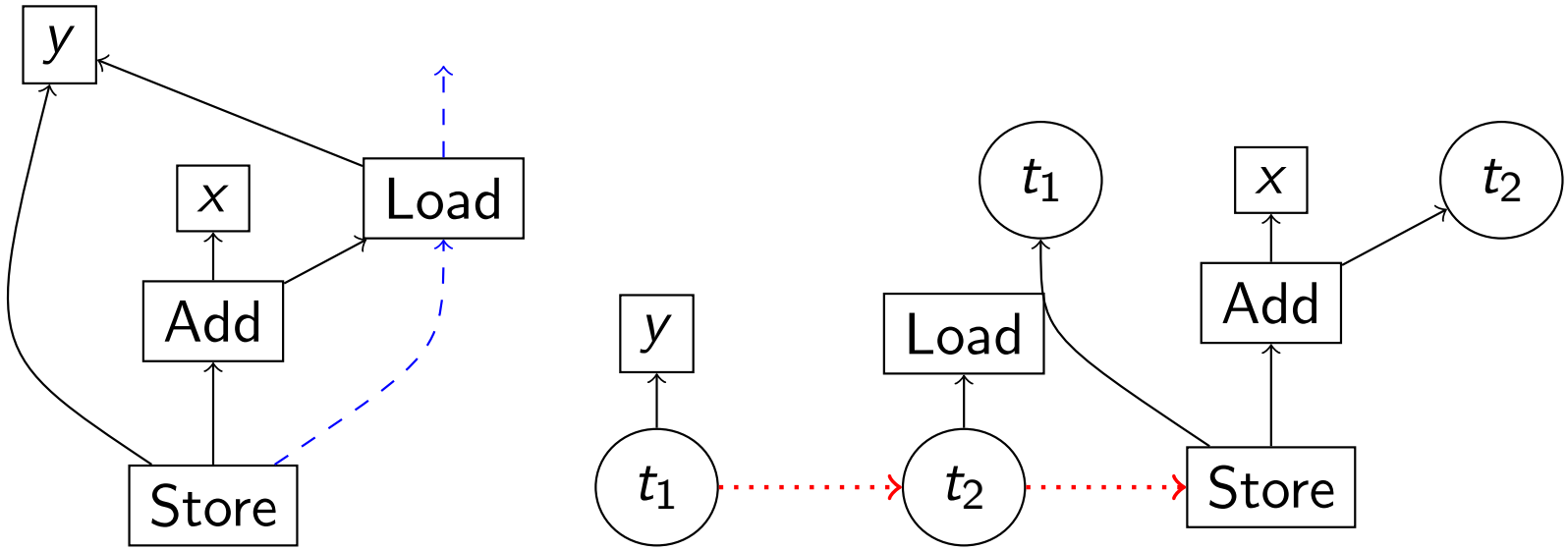
- Zwischendarstellung keine Wälder:
  - In Grundblöcken: DAGs auf Grund von Optimierungen
  - Grundblockübergreifend: DAGs auf Grund von Φ-Funktionen
- Termersetzung nur nach Aufbrechen der DAGs in Bäume anwendbar

Methoden

- Makrosubstitution
- Termersetzungsverfahren
  - CGGG von Boesler, IPD
  - BURS Code-Generator des Java Hotspot Compilers (SUN)
- Einziges Graphersetzungsverfahren:
  PBQP-Verfahren von Eckstein, König und Scholz, TU Wien

Aufbrechen des DAGs in Bäume

Load/Op/Store-Befehle bei ia32



- t_1, t_2 sind neue Hilfsvariablen die genau einmal als Wurzel und beliebig oft als Blätter auftreten können
- Die gepunktete rote Kante wird nicht von TES verwendet sondern gibt die Ausführungsreihenfolge vor
- Die gestrichelte blaue Speicherkante wird nicht mehr benötigt, da die Befehlsanordnung nun z.T. explizit ist
- Das TES kann optimalen Maschinenbefehl nicht finden

Inhalt

Befehlsanordnung

- Ziel: Bessere Nutzung der Parallelarbeit auf Befehlsebene durch geschickte Anordnung der Reihenfolge von Befehlen
- Spezialitäten:
  - Füllen des Verzögerungsschritts nach bedingten Befehlen
  - Vermeiden des Wartens auf den Speicher
  - Kollisionsvermeidung in den Rechenwerken bei mehrstufigen Befehlen
  - Software-Fließband für Schleifen
- Methodik:
  - gierige Algorithmen innerhalb eines Grundblocks
  - Optimierung für Schleifen und Teile des Dominatorbaums
- bei vielen neueren Prozessoren (moderne RISCs, Pentium Pro, Pentium II, III, ..., nicht: IA-64) durch Hardware (out-of-order-execution), aber trotzdem ist Vorarbeit durch den Übersetzer vorteilhaft.

Beispiel Verzögerungsschritt

- Fakt: Auf klassischen (heute veralteten) RISC-Prozessoren dauern bedingte Sprünge 2 Takte. Der 2. Takt kann mit Befehl gefüllt werden, der immer ausgeführt wird, unabhängig von Sprungbedingung. Sprungbedingung darf nicht von diesem Befehl abhängen.
- Beispiel MIPS:

		Tak
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	1
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	2
`beq $t2 $t3 marke`	wenn $\$t2 = \$t3$ dann gehe zu marke	3,4
`nop`	Verzögerungsschritt	4

Bemerkung: Bei heutigen superskalaren Prozessoren mit Sprungvorhersage und Caches ist die Lage nicht mehr so einfach. Ausnahmen: IA-64 (benötigt explizite Angaben was parallel / spekulativ auszuführen ist).

Beispiel Verzögerungsschritt II

- Original:

		Takt
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	1
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	2
`beq $t2 $t3 marke`	if $\$t2 = \$t3$ then goto marke	3,4
`nop`	Verzögerungsschritt	4

- unzulässig:

		Takt
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	1
`beq $t2 $t3 marke`	if $\$t2 = \$t3$ then goto marke	2,3
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	3

- zulässig:

		Takt
`addiu $t2 $t2 1`	$\$t2 := \$t2 + 1$	1
`beq $t2 $t3 marke`	if $\$t2 = \$t3$ then goto marke	2,3
`addiu $t1 $t1 1`	$\$t1 := \$t1 + 1$	3

Verfahren zur Anordnung

Methode (*list scheduling*): Konstruiere für jeden Grundblock Abhängigkeitsgraph (Info bereits in graphbasierter SSA-Form vorhanden!):

- b hängt von b' ab, wenn
  - In der Quelle b' vor b kommt (nur bei Java o.ä.)
  - mindestens einer der Befehle ein Register oder Speicherzelle beschreibt
  - der andere dasselbe Register (Speicherzelle) liest oder schreibt
- bewerte alle Befehle b mit (# Zyklen für längsten Pfad im Graph beginnend mit b, # abhängiger Befehle)
- $b < b'$ (höchste Priorität), wenn
  - $\#$ Zyklen$(b) > \#$ Zyklen(b') oder
  - $\#$ Zyklen$(b) = \#$ Zyklen(b') und
    $\#$ abh. Bef.$(b) > \#$ abh. Bef.(b')

Anordnung

- wähle nächsten Befehl b wenn:
  - alle Befehle, von denen b abhängt, bereits angeordnet und mit Beginn des laufenden Takts fertig
  - b hat höchste Priorität
  - ist bedingter Sprung und es bleiben höchstens Befehle, die in den Verzögerungsschritt passen
- gibt es keinen solchen Befehl, so wähle `nop`
- ordne den ausgewählten Befehl als nächsten an und streiche ihn samt Kanten aus dem Abhängigkeitsgraph

Beispiel Anordnung

Befehle:

Befehl	Bedeutung	Dauer (Takte)
`i := j` $+_i$ `k`	integer add	1
`x := y` $+_f$ `z`	float add	2
`i := j * k`	int/float mult	3
`i := j(k)`	lade $<< j > + k >$	2
`j(k) := i`	speichere $>< j > + <$	1
`if i` ρ `j then goto marke`	bedingter Sprung	4

Beispiel II

Programm
(inneres Produkt von a und b):

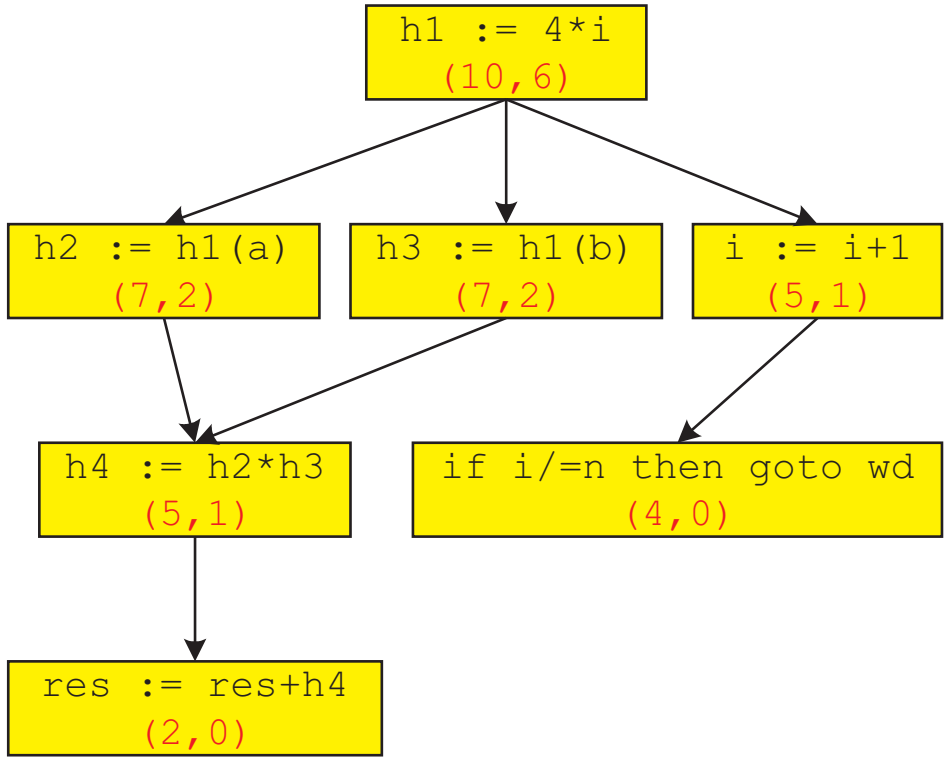
```
    res := 0;
    i := 1;
wd: h1 := 4*i;
    h2 := h1(a);
    h3 := h1(b);
    h4 := h2*h3;
    res := res+h4;
    i := i+1;
    if i/=n then goto wd
```

Ausführung Initiale Ordnung:

```
    res := 0;                    Takt:  0
    i := 1;                             1
wd: h1 := 4*i;                          2
    nop;                                3
    nop;                                4
    h2 := h1(a);                        5
    h3 := h1(b);                        6
    nop;                                7
    h4 := h2*h3;                        8
    nop;                                9
    nop;                               10
    res := res+h4;                     11
    i := i+1;                          12
    if i?n then goto wd                13
    nop;                               14
    nop;                               15
    nop;                               16
```

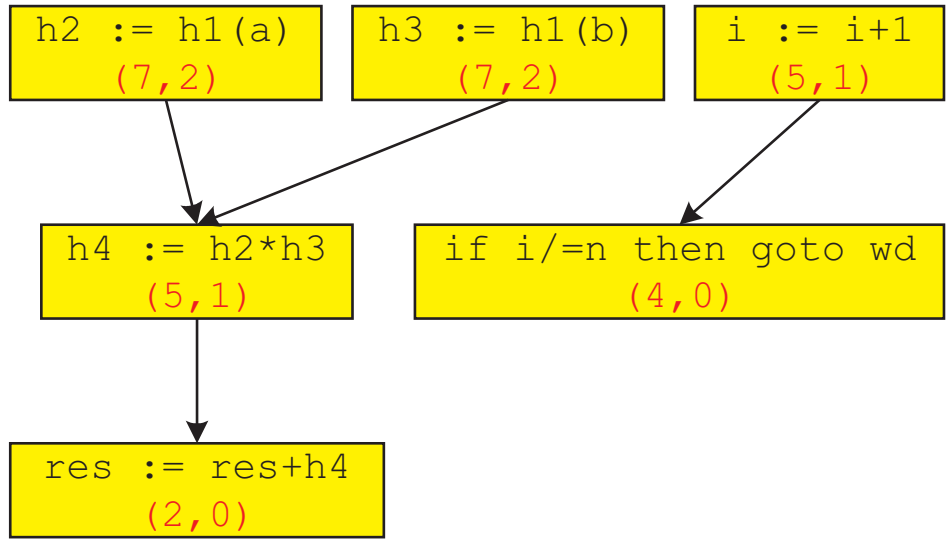

Beispiel

```
        res := 0;
        i := 1;
wd:
```



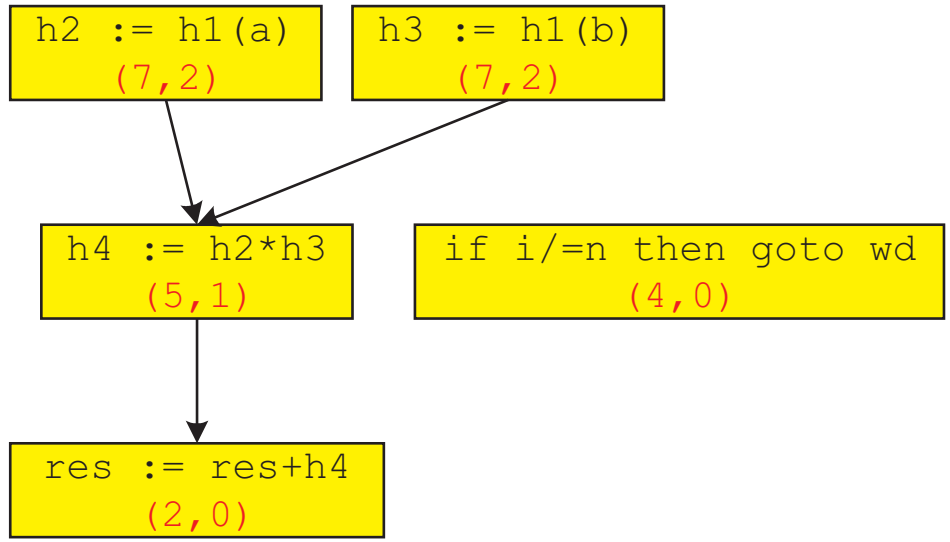
Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
```



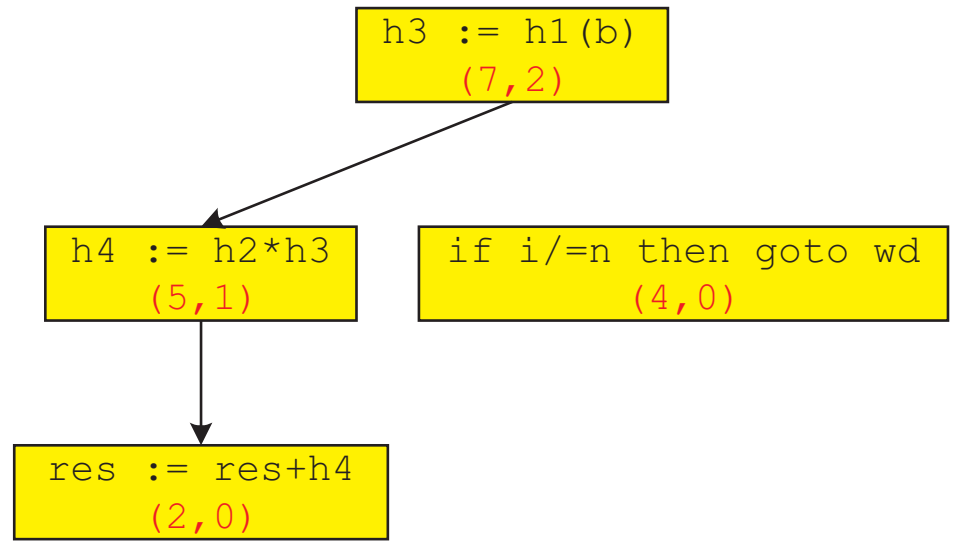
Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
```



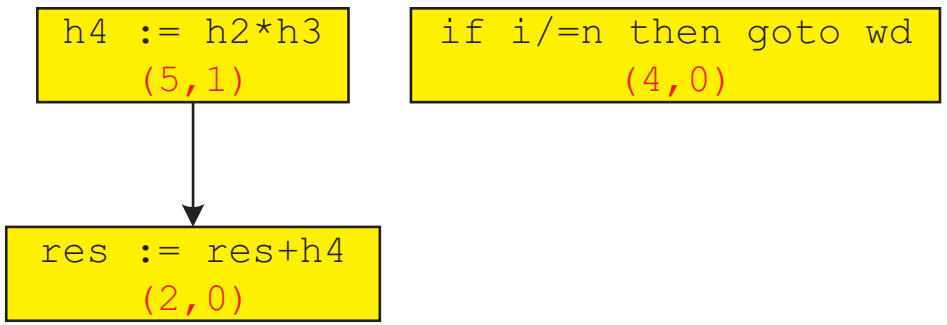
Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
        h2 := h1(a);
```



Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
        h2 := h1(a);
        h3 := h1(b);
        nop;
```



Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
        h2 := h1(a);
        h3 := h1(b);
        nop;
        h4 := h2*h3;
```

```
if i/=n then goto wd
      (4,0)
```

```
res := res+h4
   (2,0)
```


Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
        h2 := h1(a);
        h3 := h1(b);
        nop;
        h4 := h2*h3;
        if i/=n then goto wd;
```

```
res := res+h4
   (2,0)
```


Beispiel

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
        h2 := h1(a);
        h3 := h1(b);
        nop;
        h4 := h2*h3;
        if i/=n then goto wd;
        nop;
```

```
res := res+h4
    (2,0)
```


Beispiel XI

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        i := i+1;
        h2 := h1(a);
        h3 := h1(b);
        nop;
        h4 := h2*h3;
        if i/=n then goto wd;
        nop;
        res := res+h4;
```

Gesamtdauer:
11 Takte, Schleife 9 Takte

Verbesserung durch Schleifenausrollen

```
        res := 0;
        i := 1;
wd:     h1 := 4*i;
        j := i+1;
        f1 := 4*j;
        h2 := h1(a);
        h3 := h1(b);
        f2 := f1(a);
        f3 := f1(b);
        h4 := h2*h3;
        f4 := f2*f3;
        i := i+2;
        res := res+h4;
        if i<n then goto wd;
        res := res+f4;
        nop;
        nop;
```

Gesamtdauer der Doppelscheife: 15 Takte

(Behandlung ungerades n fehlt noch, ist aber außerhalb der Schleife möglich)

Weitere Verfahren

- Blockanordnung unter Berücksichtigung interner Verarbeitungseinheiten
- Spuranordnung (trace scheduling)
- Software-Fließband

Verarbeitungseinheiten berücksichtigen

- Fakt: Superskalare Prozessoren zerlegen Befehle in Aufgaben für verschiedene Verarbeitungseinheiten (Befehlsentschlüsselung, Speicherzugriff, ganzzahlige, logische Gleitpunkteinheit,...). Außer externen Kollisionen (noch nicht vorhandene Operanden) gibt es interne Kollisionen, weil die benötigte Verarbeitungseinheit noch belegt ist.
- Verfahren (Thomas Müller):
  - Berechne für Befehlsfolgen $b_1, \ldots, b_n$, meist $n = 2$, die minimale Verzögerung $v(b)$ für jeden Befehl b, damit dieser innerhalb der Sequenz kollisionsfrei ausgeführt werden kann
  - Konstruiere endlichen Automaten: Zustand entspricht Belegungszustand der Verarbeitungseinheiten, Eingabe sind Befehle b, Ausgabe $v(b)$.
  - Konstruiere Abhängigkeitsgraph. Wähle Folgebefehl mit $v(b)$ minimal, gleichzeitig Zustandsübergang im Automat

Spuranordnung

- Betrachte häufig ausgeführte Pfade (Spuren) im Ablaufgraph als Einheit und ordne sie gemeinsam an.
  - Spuren aus Ausführungsprofil ermitteln (falls vorhanden)
  - erweiterte Grundblöcke als Spuren
  - allgemeiner: Spuren sind Wege im Dominanzbaum
- Spuranordnung kann Befehle vorzeitig platzieren, bevor klar ist, dass sie wirklich benötigt werden (spekulative Ausführung)
  - Problem: anschließende Korrekturphase nötig, um unerwünschte Wirkung spekulativer Befehle zu beseitigen
  - Korrekturmethode: spekulative Befehle dürfen nur Register, keinen Speicherzustand verändern; Registerinhalt anschließend nicht mehr lebendig
- Problem: Block- und Spuranordnung verändern (meist: erhöhen) die Anzahl der belegten Register: neue Registerzuteilung nötig

Software-Fließband

- Problem: am Ende des Schleifenrumpfs sind alle Verarbeitungseinheiten frei, werden erst im nächsten Schleifendurchgang wieder gefüllt.
  Wie kann fortlaufende Belegung erreicht werden?
- Methoden:
  - Schleifenausrollen, siehe früheres Beispiel
  - vorzeitiger, spekulativer Start des nächsten Schleifendurchlaufs (mit Vorlauf vor Schleifenbeginn und Korrekturphase nach Schleifenende)
  - vorzeitiger Beginn des nächsten Schleifendurchlaufs (Abschlußbefehle des vorigen Schleifendurchlaufs an den Beginn des nächsten) mit entsprechender Korrekturphase
  - Konsequenz: In den meisten Fällen belegen Werte in aufeinanderfolgenden Durchgängen unterschiedliche Register!

  Bemerkung: Die IA-64 unterstützt diese Maßnahme von der Hardwareseite durch „roulierende Register“.

Inhalt

Registerzuteilung

- Problem: Annahme während der Optimierungsphase: Anzahl verfügbarer Register ist unbeschränkt. Aufgabe: Reduktion auf die tatsächlich verfügbaren, endlich vielen Register.
- Prinzip: Alle Register nach Möglichkeit gleich behandeln. Unterscheide Register nach:
  - Gebrauch durch Hardware festgelegt (Befehlszähler, Bedingungsanzeige, Gleitpunkt-/allgemeine Register, Adressregister, gerade/ungerade z.B. `mult`, ...)
  - Gebrauch durch Konventionen Laufzeitsystem festgelegt
  - freie Register
- Verfahren zur Zuteilung freier Register:
  - Lokale Registerzuteilung mit Freiliste (*on the fly*)
  - *linear-scan register allocation*
  - Graphfärben

Registerzuteilung – Literatur

- G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, *Register Allocation via Coloring*, Computer Languages, 6(1), January 1981.
- P. Briggs, K. Cooper, and L. Torczon, *Improvements to Graph Coloring Register Allocation*, Transactions on Programming Languages and Systems, 16(3), May 1994.
- F. Chow and J. Hennessy, *The Priority-Based Coloring Approach to Register Allocation*, Transactions on Programming Languages and Systems, 12(4), October 1990.
- Traub, Holloway, and Smith, *Optimized interval splitting in a linear scan register allocator*, VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, 2005, pp. 132–141.
- Sebastian Hack, Daniel Grund, Gerhard Goos, *Register allocation for programs in SSA-form*, Compiler Construction 2006, Springer, March 2006.

Registerzuteilung: Wann/Wo

Möglichkeiten „Wann“:

- Nach Codeselektion; Probleme:
  - Kosten in Codeselektion von Registerzuteilung abhängig,
  - Auslagerungscode (*spill-code*) von Registerzuteilung abhängig,
  - Bestimmter Code nur mit bestimmten Registern auswählbar.
- Vor Codeselektion; Probleme:
  - Codeselektion definiert Anzahl der benötigten Register
  - Manche Werte werden nie explizit berechnet, z.B. Werte auf Adressierungspfaden
- Codeselektion – Registerzuteilung – Codeselektion
- Während der Codeselektion (*on the fly*)
- Aber, Lebendigkeit nur definiert nach Anordnung der Befehle

Möglichkeiten „Wo“:

- Ausdrücke
- Grundblöcke (lokal)
- Schleifen
- Prozeduren (global)
- Programme

Registerzuteilung – Aufgaben

- Aufgabe der Registerzuteilung ist Abbildung der Programmvariablen auf Prozessorregister
- Aufgaben im Detail:

  Zuteilen Finde Abbildung von Programmvariablen auf Prozessorregister

  Auslagern Lagere Variablen in den Hauptspeicher aus, falls nicht genug Register verfügbar sind

  Verschmelzen Eliminiere unnötiges Kopieren von Variablen in dem Programm

Partitionierung des Registersatzes

1. Dringend (in Register) verfügbare Werte: Kellerpegel, Haldenpegel, Schachtel, etc.
2. Globale Werte – (Prozedur-)globale Zuteilung
3. Zwischenergebnisse in Ausdrucksbäumen
   - 4-5 Register freihalten
   - *on the fly* zuteilen

Beobachtung: Unterscheidung zwischen 2. und 3. nicht zwingend - wird heutzutage oft einheitlich behandelt. Register aus 1. werden fest vorgegeben und bei Registerzuteilung ignoriert.

Inhalt

Lokale Registerzuteilung mit Freiliste (*on the fly*)

- Bestimme letzte Verwendungen von Werten im Grundblock.
- Durchlaufe Grundblock von Anfang bis Ende:
  - Falls Register benötigt wird rufe *allocReg*() auf; nach letzter Verwendung *freeReg*(r).
  - *allocReg*(): Gibt ein freies Register zurück, falls keines mehr frei ist, löse Ausname aus. Entferne Register aus der Freiliste.
  - *freeReg*(r): Füge Register r in die Freiliste ein.
- Am Ende des Grundblocks (teilweise auch Ausdrucks) werden alle Variablen in den Speicher geschrieben.
- Achtung: Falls Register fehlen, kann kein Programm erzeugt werden (die ersten Turbo Pascal Übersetzer funktionierten wirklich so), ggf. muss dieses Verfahren um die Möglichkeit des Auslagerns erweitert werden.

Erweiterungen

Oft Kombination von lokalen mit globalen Methoden:

- Teile Variablen deren Lebenszeiten komplett innerhalb eines Grundblocks liegen mit lokalem Verfahren zu.
- Teile übrige Variablen mit globalem Verfahren zu. Beachte dabei Interferenzen mit bereits lokal vergebenen Registern.
- Lokales Verfahren kann oft mit Codeauswahl kombiniert werden.

Nutze höhere Geschwindigkeit/besseres Auslagerungsverhalten für lokale Variablen ohne, dass Werte an Grundblockgrenzen zurück in den Speicher geschrieben werden.

Inhalt

linear-scan register allocation

- Die Laufzeit der Graphfärbung ist (sehr) hoch.
- Codequalität von rein lokalen Verfahren schlecht.
- Linear-scan register allocation ist eine Erweiterung des *on the fly* Ansatzes.

Algorithmus

- Berechne Lebendigkeitsinformation und Ablauflaufgraph.
- Durchlaufe das Programm in umgekehrter Postfixordnung: Dies erzeugt linear Liste aller Grundblöcke.
- Berechne Lebendigkeitsinterval für jede Variable.
- Durchlaufe sortierte Intervalliste:
  - Verwende *allocReg*() und *freeReg*() wie beim *on the fly* Ansatz.
  - Auslagern bei Bedarf. Lagere längste verbleibende Intervalle zuerst aus.

linear-scan – Erweiterungen

Es existieren verschiedene Verbesserungen um schwächen des Original Linear-Scan Ansatz zu beheben:

- Mehrere Intervalle pro Variablen: Ausnutzen von „Lücken” in den Lebenszeiten.
- Handhabung von Register-Constraints
- Kein Freihalten von Registern für Reloads; erzeuge stattdessen neue Intervalle
- Splitten von Intervallen

Inhalt

Registerzuteilung mit Graphfärbung (nach Chaitin)

Prinzip (Chaitin 1981):

- Konstruiere für jede Prozedur einen Interferenzgraph
- Ecken sind die Variablen (auch temporäre) des Programms
- Ecken e, e' durch Kante verbinden, wenn sie nicht gleichzeitig dasselbe Register belegen können.
  - Grund von Unverträglichkeit: überlappende Lebensdauer.
  - Information: Definition und Benutzung von Werten ($\rightarrow$ SSA!).
- Graphfärbung mit minimaler Farbanzahl (*chromatische Zahl* $\chi(G)$) liefert die Minimalanzahl benötigter Register und gleichzeitig die Registerzuordnung.

Globale Register-Vergabe

- Interferenzgraph (*register inference graph*):
  - Ungerichtet
  - Ecken: symbolische Register – Werte aus Wertnumerierung
  - Kanten: zwischen Ecken, die gleichzeitig aktiven Werten entsprechen
- Konstruktion des Interferenzgraph:
  - Topologisches Sortieren der halbgeordneten Berechnungen in SSA Regionen
    Achtung: Das ist Aufgabe der Befehlsanordnung, für guten Code reicht topologisches Sortieren nicht
  - Definition-Benutzungs-Information explizit im SSA Graphen
  - Registerinterferenz direkt bestimmbar

Graphenfärben

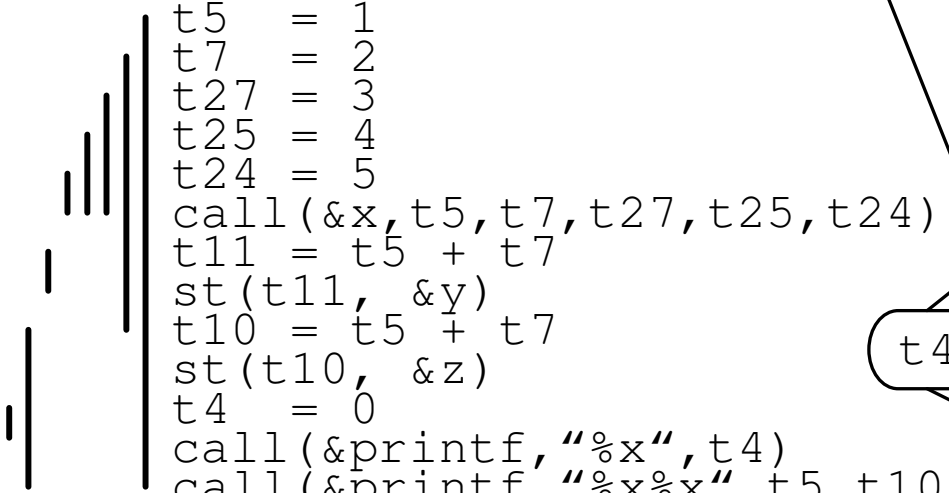
- Ist Register-Interferenz-Graph mit k = Anzahl der Register färbbar?

  Aber: Bestimmung der chromatischen Zahl ist *NP-Problem*.
- Hinreichendes Kriterium liefert folgende linear laufende Heuristik:
  1. Wähle Ecke n mit Grad kleiner k aus.
  2. Nicht möglich? Ausgabe: Weiß nicht, ob k-färbbar.
  3. Sonst eliminiere n und seine Kanten.
  4. Gehe zu 1. wenn Graph nicht leer.
     Sonst Ausgabe: k-färbbar.
- Färbe Graphen in umgekehrter Eliminierungsfolge.

Beispiel – Interferenzgraph

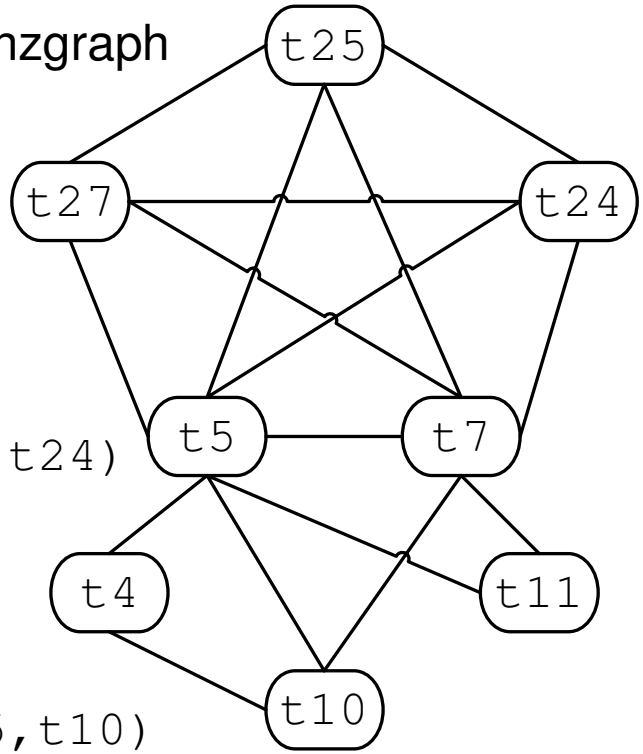
Lebens-
zeiten

Programm



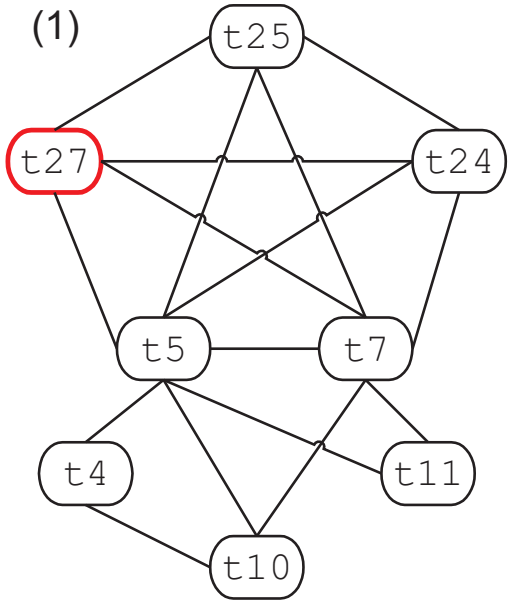
```
t5  = 1
t7  = 2
t27 = 3
t25 = 4
t24 = 5
call(&x,t5,t7,t27,t25,t24)
t11 = t5 + t7
st(t11, &y)
t10 = t5 + t7
st(t10, &z)
t4  = 0
call(&printf,“%x“,t4)
call(&printf,“%x%x“,t5,t10)
```

Interferenzgraph

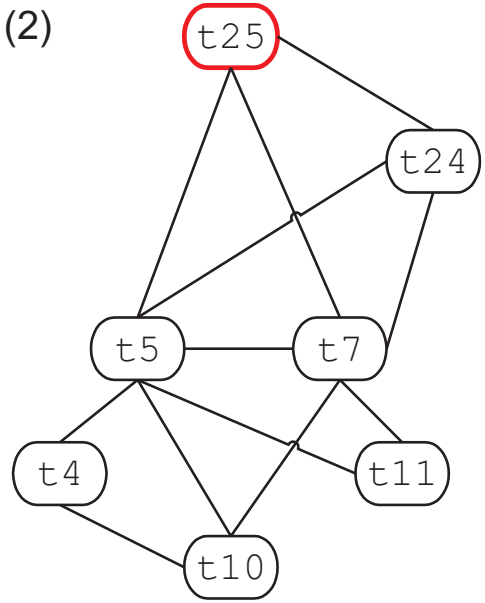


Beispiel – Ecken eliminieren ($k = 5$)

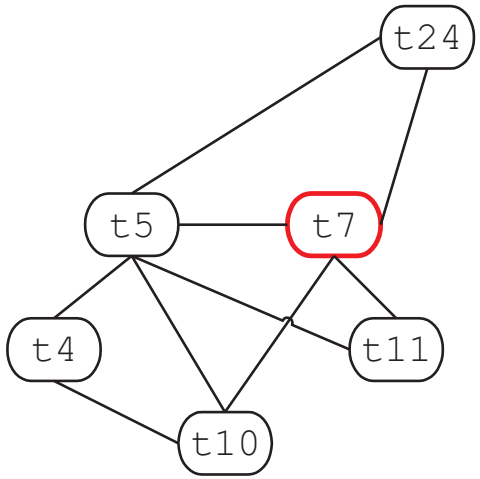
(1)



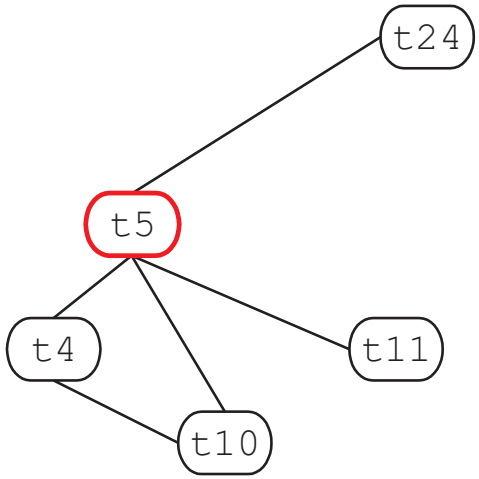
(2)



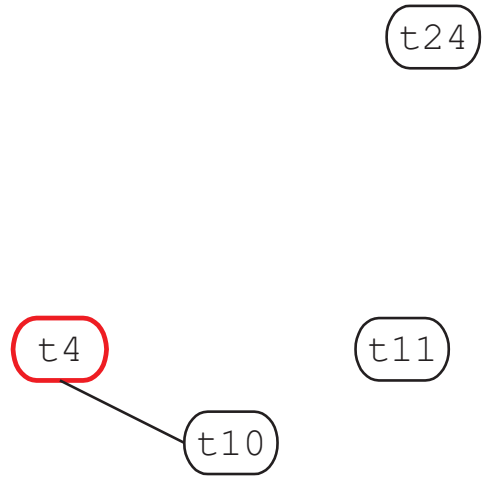
(3)



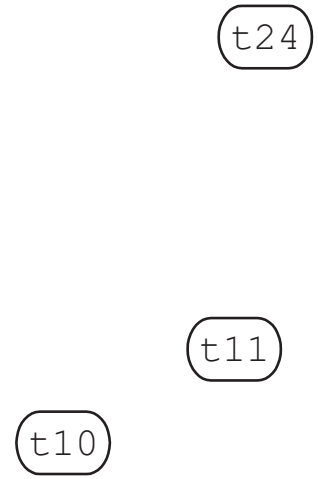
(4)



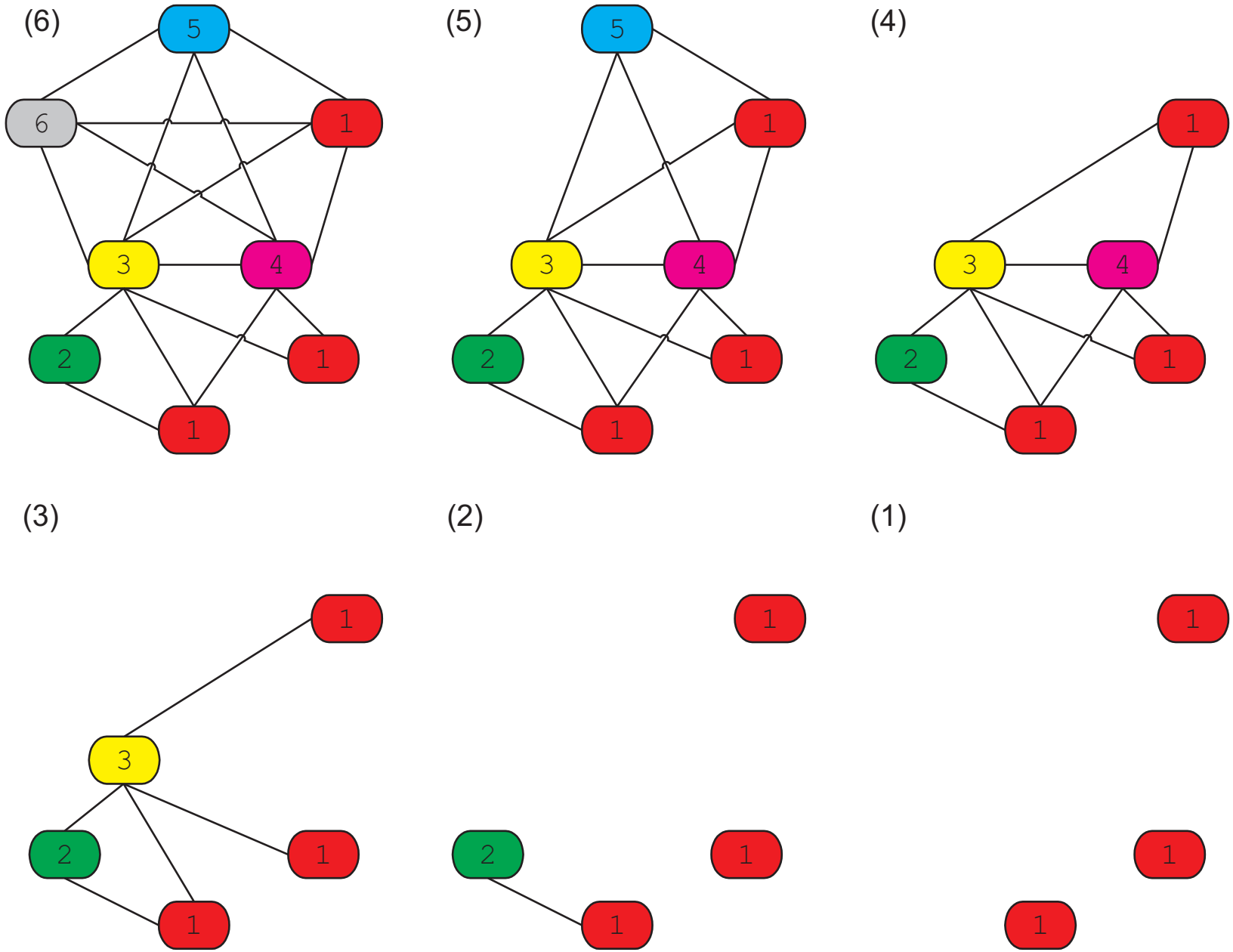
(5)



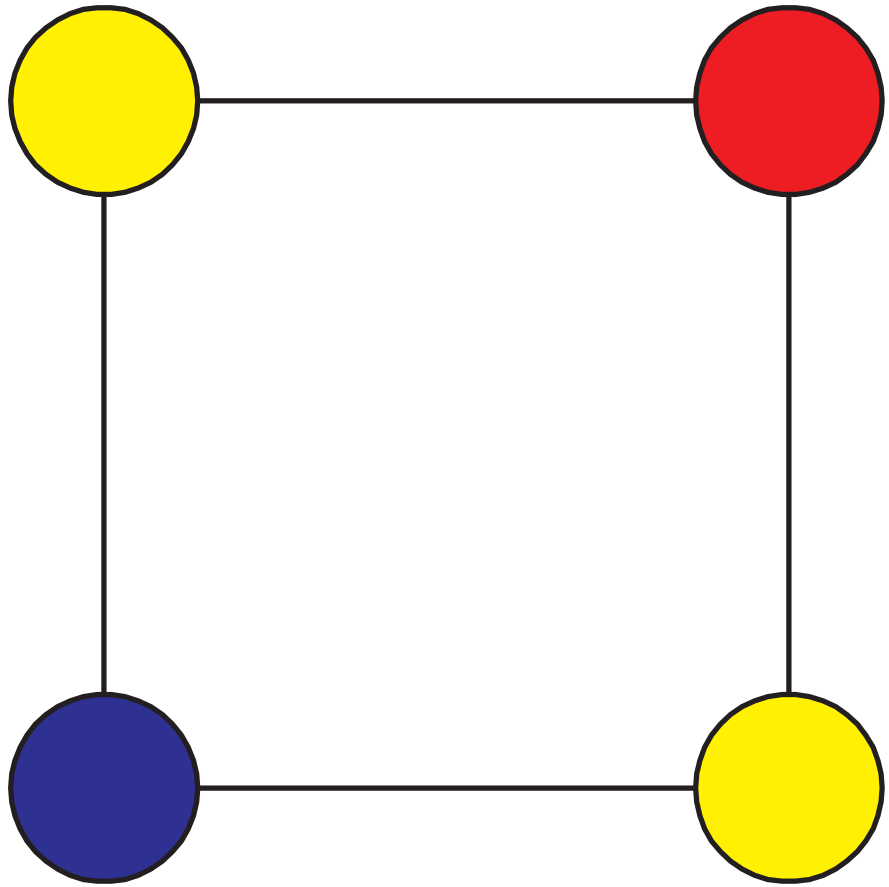
(6)



Beispiel – Färbung der Ecken



Offensichtlich 2-färbbar aber Heuristik scheitert



Heuristik unterstellt, daß alle Nachbarecken unterschiedlich gefärbt sein müssen.

Färbung nicht gefunden

Wenn Färbung nicht gefunden wurde, kann die Heuristik iteriert werden:

- Eliminiere eine Ecke m aus Register-Interferenz-Graph
- Entsprechender Wert kommt nicht in globales Register, sondern wird in den Speicher ausgelagert.
- Neuer Versuch:
  Graphenfärben des Rest-Register-Interferenz-Graphen
- Auswahl von m heuristisch siehe 1. bis 4. der nachfolgenden Rangfolge

Register auslagern

- Genügen die Register nicht („Registerdruck zu hoch“, Graph nicht k-färbbar), so müssen Werte in den Speicher ausgelagert werden
- Auswahl der auszulagernden Werte (Rangfolge):
  1. Wert kann mit einem (oder wenigen) Befehlen aus anderen Registerinhalten wieder berechnet werden
  2. Wert schon im Speicher vorhanden oder mit einem Speicherzugriff wiederberechenbar
  3. Wert wird möglichst lange nicht benötigt
  4. Wert interferiert mit vielen anderen
- Bei 1. und 2. kein Auslagern nötig, 3. nur angenähert beurteilbar, z.B. innerhalb eines Grundblocks
- Probleme:
  - Auslagern kann während der Registerzuteilung, aber auch danach nötig werden, z.B. während Befehlsanordnung
  - Befehlsanordnung kann die Bedingungen verändern

Weitere Verbesserungen

Bevor die Registerzuteilung beginnt (Chow & Hennessy):

- Konstanten aufspalten, d.h. die Konstanten die in SSA-Form zusammengezogen wurden, unmittelbar vor ihrer Verwendung in den Code platzieren.
- Allgemeiner – Rematerialisierung: Werte die sich leicht (wenige Takte $<$ Speicherzugriffzeit) wiederberechnen lassen, nicht in Register belassen

Experimenteller Vergleich

Angegeben: Jeweils Quotient aus *linear-scan* / Graphenfärben

Prozedur	Datenstruktur [Elemente]	Optimiererlaufzeit [s]
cvrin.c	245 / 1061	1.5 / 0.4
twldrv.f	6218 / 51796	3.7 / 8.8
fpppp.f	6697 / 116926	4.5 / 15.8
field()	7611 / 86741	4.9 / 14.9

Benchmark	dyn. Befehlsanzahl	Ausführungszeit
alvinn	1.000	0.995
doduc	1.002	1.018
eqntott	1.000	1.003
espresso	1.013	1.060
fpppp	1.052	1.043
li	1.018	0.966
tomcatv	1.000	0.995
compress	1.002	1.020
m88ksim	1.008	1.024
sort	1.035	1.082
wc	1.000	1.011

Inhalt

Registerzuteilung mit Graphfärbung (nach Chaitin/Briggs)

Architektur mit Iteration (non-SSA)

- Jeder ungerichtete Graph kann als Interferenzgraph (IG) auftreten
- Die Bestimmung der chromatischen Zahl ist NP-vollständig
- Färben ist eine Heuristik $\Rightarrow$ Iteration notwendig
  Selbst wenn Heuristik $n + k$ Farben schätzt, muss nach Auslagen von n Registern keine Lösung mit k Registern zu finden sein
- Auslagern ist auf IG fokussiert

Registerzuteilung mit Graphfärbung (nach Hack/Goos)

Architektur ohne Iteration (SSA)

$\longrightarrow$ Auslagern $\longrightarrow$ Färben $\longrightarrow$ Verschmelzen $\longrightarrow$ SSA-Abbau $\longrightarrow$

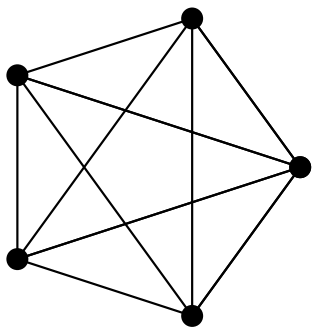
- Wegen Chordalität[1] der SSA IGs:
  - Trennen von Auslagern und Verschmelzen möglich
  - Färben in polynomieller Zeit
- Registerdruck ist ein präzises Maß für Zahl benötigter Register
- Wir wissen vor dem Färben welche Werte ausgelagert werden müssen
  - ▶ Alle Marken bei denen der Registerdruck größer als k ist
- Auslagern kann vor dem Färben geschehen und
- Das Färben scheitert anschließend nie!
- Verschmelzen neu formuliert als Optimierungsproblem mit einer Kostenfunktion für Färbungen ▶ Komplexität ist jetzt hier

[1] Definition folgt noch

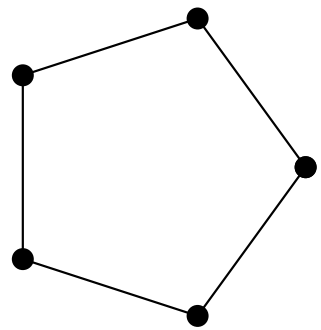
Inhalt

Vollständige Graphen and Zyklen

Grundlagen



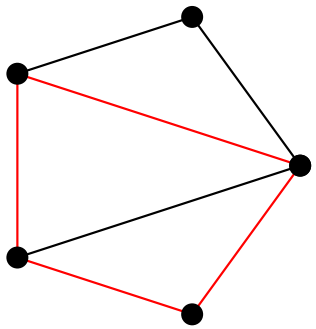
Vollständiger Graph K^5



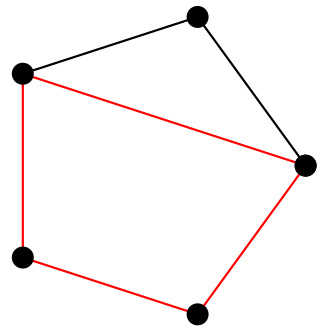
Zyklus C^5

Teilgraphen / Untergraphen

Grundlagen



Graph mit einem C^4-Teilgraphen



Graph mit einem C^4-Untergraphen

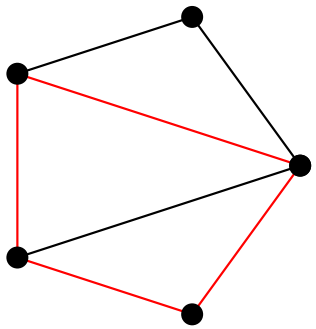
Untergraphen werden auch induzierte Teilgraphen genannt

Definition

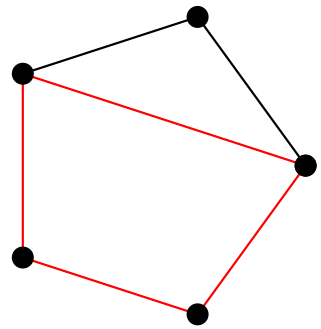
Komplette Untergraphen heißen Cliquen

Teilgraphen / Untergraphen

Grundlagen



Graph mit einem C^4-Teilgraphen



Graph mit einem C^4-Untergraphen

Untergraphen werden auch induzierte Teilgraphen genannt

Definition

Komplette Untergraphen heißen Cliquen

Cliquenzahl and chromatische Zahl

Grundlagen

$\omega(G)$ Eckenzahl der größten Clique in G

$\chi(G)$ Anzahl der Farben in einer minimalen Färbung von G

Corollary

$\omega(G) \leq \chi(G)$ gilt für jeden Graphen G

$\omega(G)$	3	2	2	3
$\chi(G)$	3	2	3	3

Cliquenzahl and chromatische Zahl

Grundlagen

$\omega(G)$ Eckenzahl der größten Clique in G

$\chi(G)$ Anzahl der Farben in einer minimalen Färbung von G

Corollary

$\omega(G) \leq \chi(G)$ *gilt für jeden Graphen* G

$\omega(G)$	3	2	2	3
$\chi(G)$	3	2	3	3

Cliquenzahl and chromatische Zahl

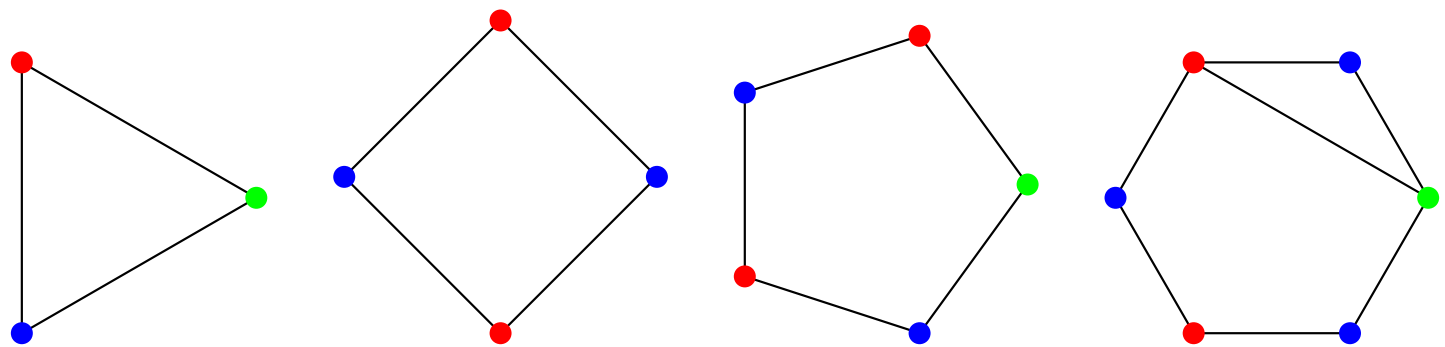
Grundlagen

$\omega(G)$ Eckenzahl der größten Clique in G

$\chi(G)$ Anzahl der Farben in einer minimalen Färbung von G

Corollary

$\omega(G) \leq \chi(G)$ *gilt für jeden Graphen* G



$\omega(G)$	3	2	2	3
$\chi(G)$	3	2	3	3

Perfekte Graphen

Grundlagen

Definition

G ist perfekt $\iff$ $\chi(H) = \omega(H)$ gilt für alle Untergraphen H von G

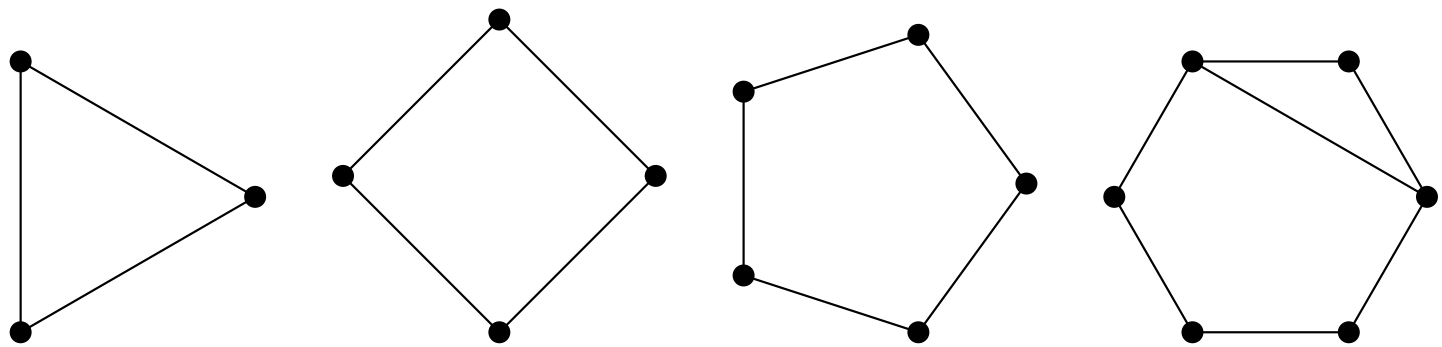
perfekt? ✓ ✓

Perfekte Graphen

Grundlagen

Definition

G ist perfekt $\iff$ $\chi(H) = \omega(H)$ gilt für alle Untergraphen H von G



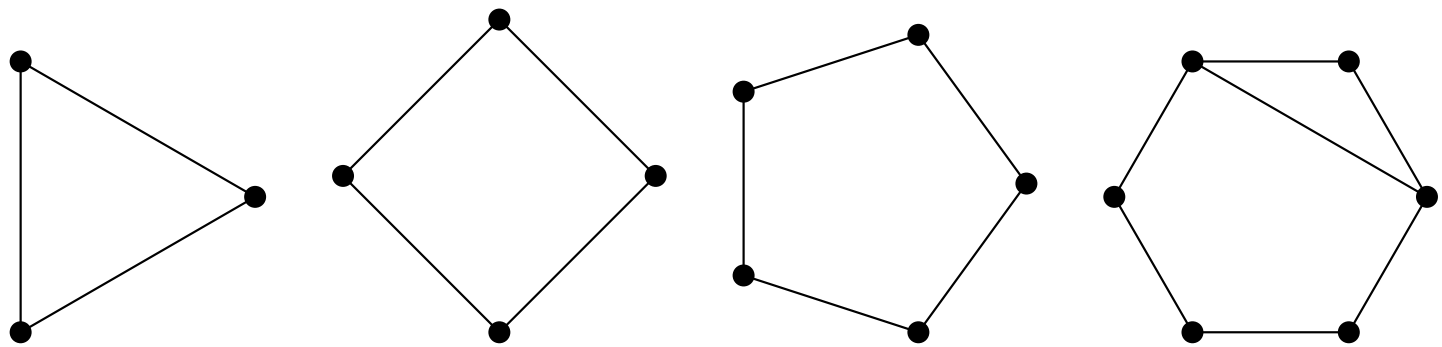
perfekt?	✓	✓		

Perfekte Graphen

Grundlagen

Definition

G ist perfekt $\iff$ $\chi(H) = \omega(H)$ gilt für alle Untergraphen H von G



perfekt?	✓	✓

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3

chordal?	✓	✓	

Theorem

Chordale Graphen sind perfekt

Theorem

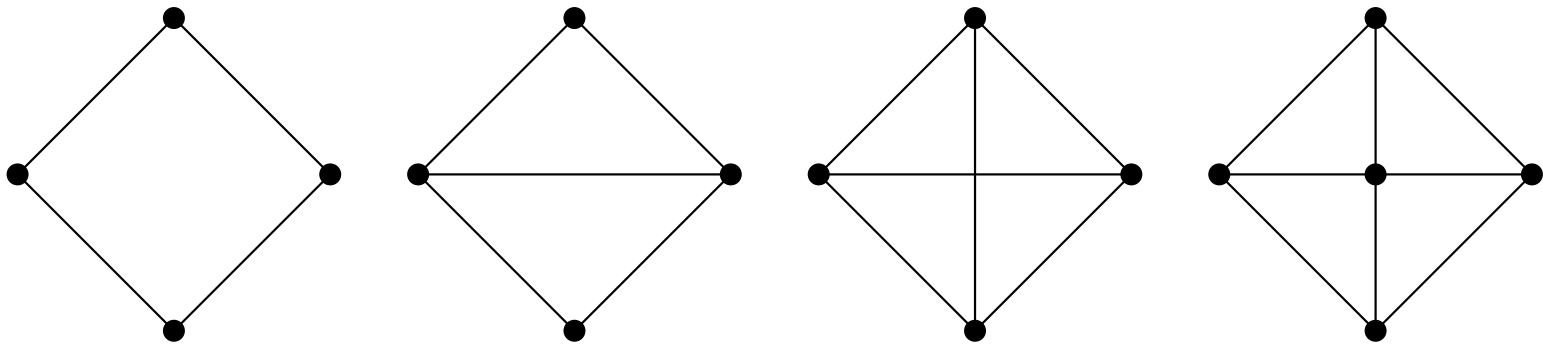
Chordale Graphen sind in $O(\lvert V\rvert \cdot \omega(G))$ optimal färbbar

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3



chordal?	✓	✓	

Theorem

Chordale Graphen sind perfekt

Theorem

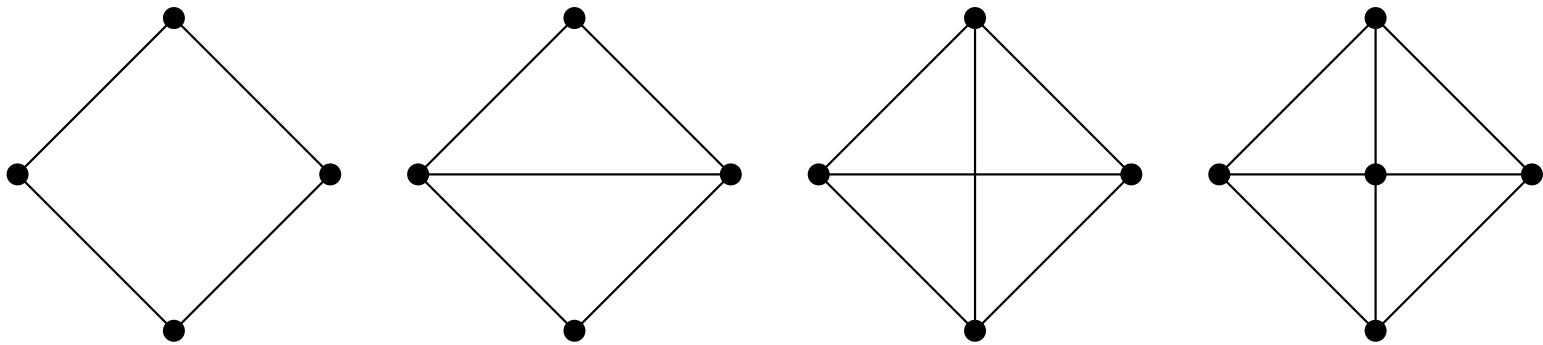
Chordale Graphen sind in $O(\lvert V\rvert \cdot \omega(G))$ optimal färbbar

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3



chordal?		✓	✓

Theorem

Chordale Graphen sind perfekt

Theorem

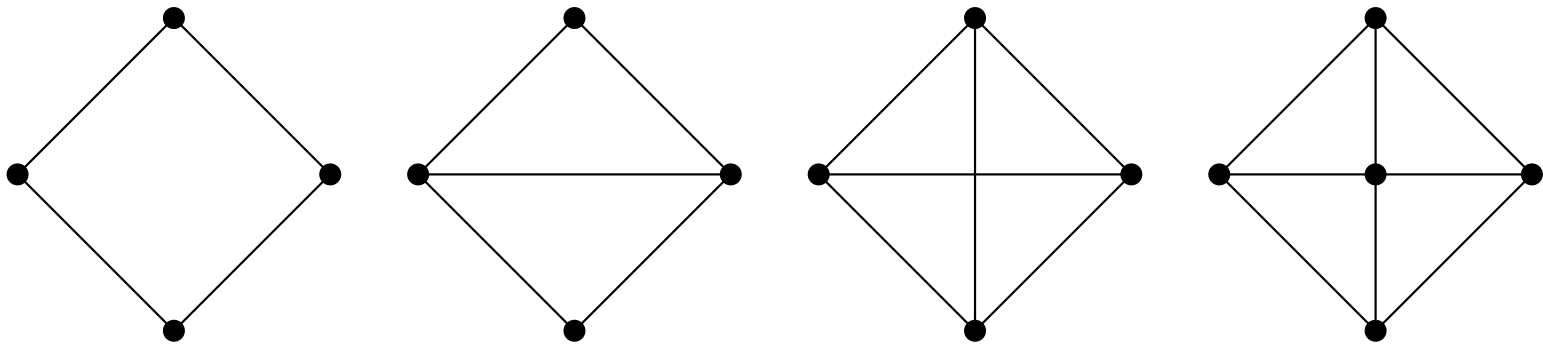
Chordale Graphen sind in $O(\lvert V\rvert \cdot \omega(G))$ optimal färbbar

Chordale Graphen

Grundlagen

Definition

G ist chordal $\iff$ G enthält keine Untergraphen mit Zyklen länger 3



chordal?		✓	✓

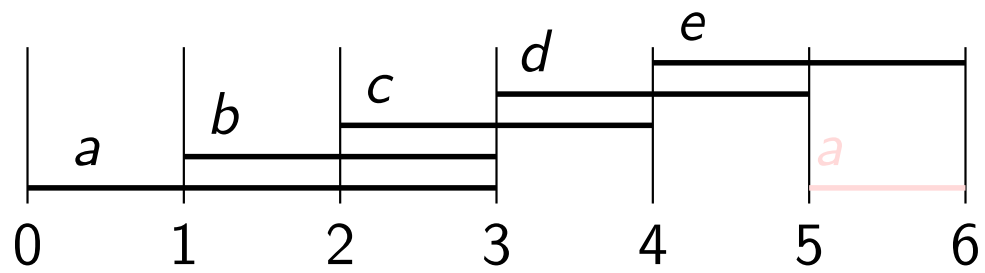
Theorem

Chordale Graphen sind perfekt

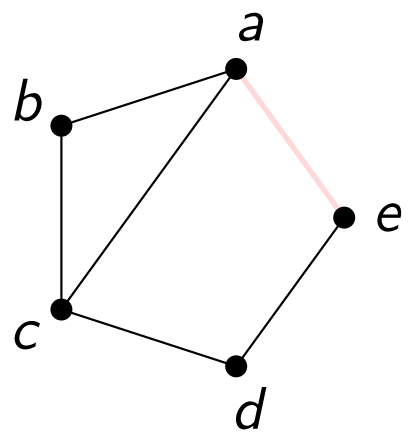
Theorem

Chordale Graphen sind in $O(|V| \cdot \omega(G))$ optimal färbbar

Warum sind SSA-IGs chordal? — intuitiv

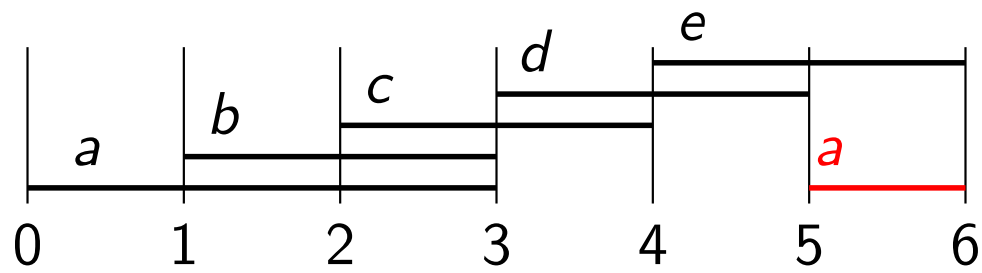


- Jedes Intervall entspricht der Lebenszeit einer Variablen
  - einer Ecke im IG

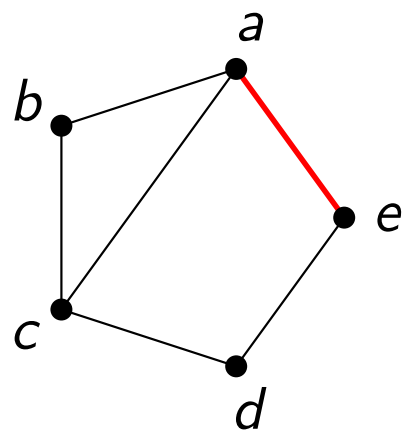


- Kann durch Einfügen einer Kante (a, e) ein Zyklus gebildet werden?
- Das geht nur wenn a erneut bei 5 beginnt
- Das verletzt jedoch die SSA-Eigenschaft, da a 2 Definitionen hätte!

Warum sind SSA-IGs chordal? — intuitiv

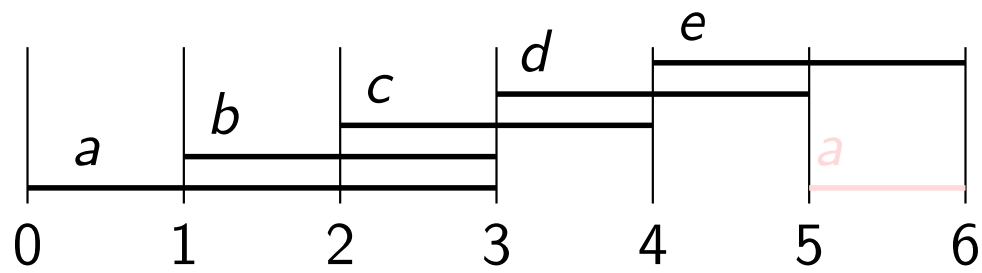


- Jedes Intervall entspricht der Lebenszeit einer Variablen
  - einer Ecke im IG

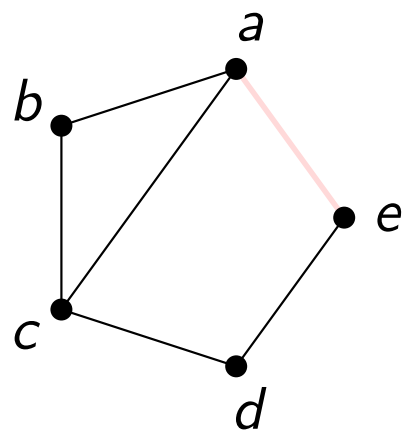


- Kann durch Einfügen einer Kante (a, e) ein Zyklus gebildet werden?
- Das geht nur wenn a erneut bei 5 beginnt
- Das verletzt jedoch die SSA-Eigenschaft, da a 2 Definitionen hätte!

Warum sind SSA-IGs chordal? — intuitiv



- Jedes Intervall entspricht der Lebenszeit einer Variablen
  - einer Ecke im IG

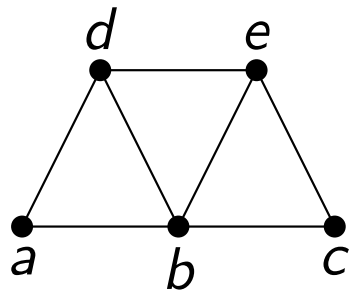


- Kann durch Einfügen einer Kante (a, e) ein Zyklus gebildet werden?
- Das geht nur wenn a erneut bei 5 beginnt
- Das verletzt jedoch die SSA-Eigenschaft, da a 2 Definitionen hätte!

Inhalt

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



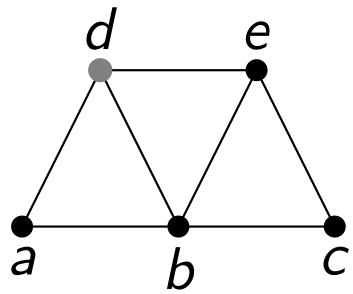
Eliminationsschema

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema

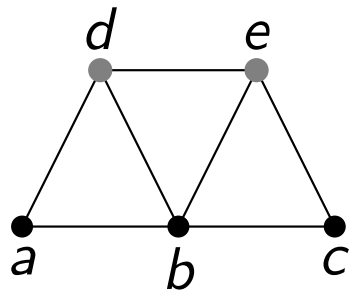
d,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



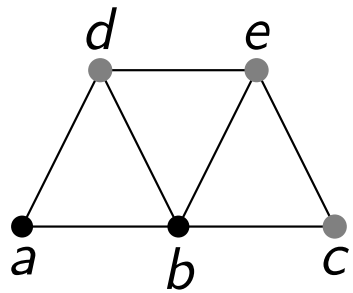
Eliminationsschema
d, e,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



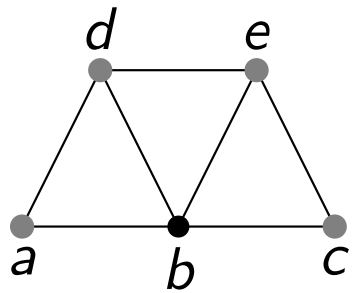
Eliminationsschema
d, e, c,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema

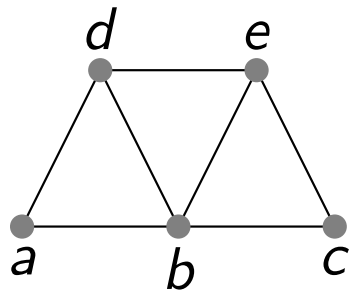
d, e, c, a,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



Eliminationsschema

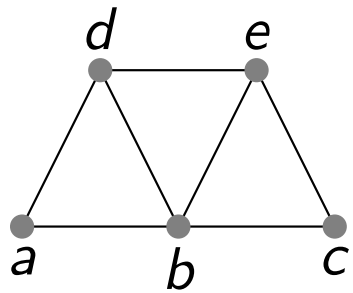
d, e, c, a, b

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



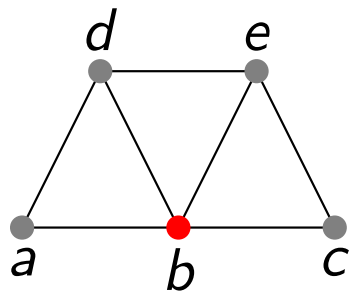
Eliminationsschema
d, e, c, a, b

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



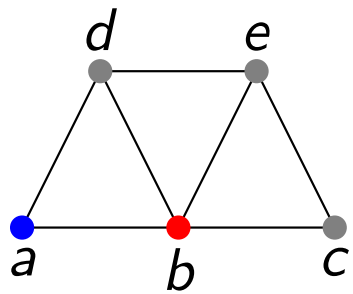
Eliminationsschema
d, e, c, a,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



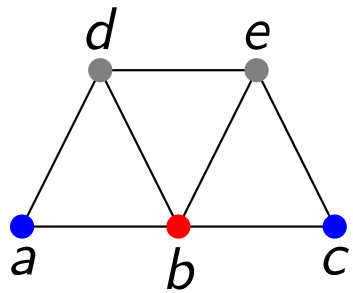
Eliminationsschema
d, e, c,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



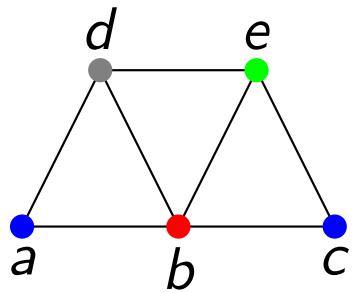
Eliminationsschema
d, e,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



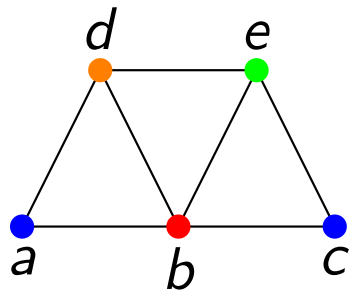
Eliminationsschema
d,

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



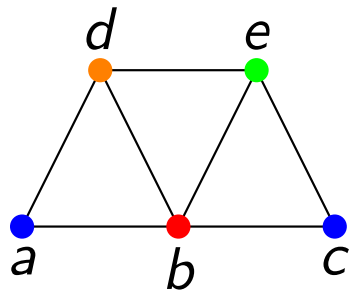
Eliminationsschema

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Färbung

- Entferne schrittweise Ecken aus dem Graphen
- Füge die Ecken in umgekehrter Reihenfolge wieder ein
- Weise jeder Ecke die nächste verfügbare Farbe zu



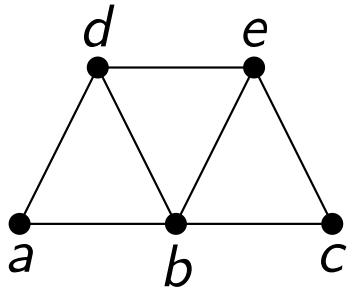
Eliminationsschema

Theorem (Graphentheorie)

Für jeden Graph existiert ein Eliminationsschema, dass zu einer optimalen Färbung führt.

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



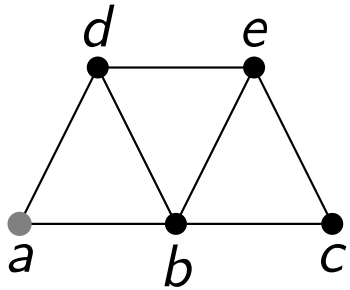
Eliminationschema

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema

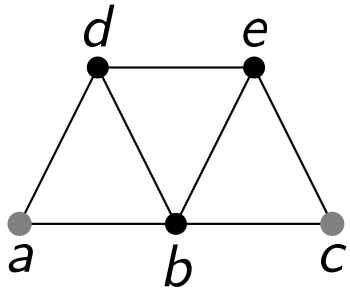
a,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



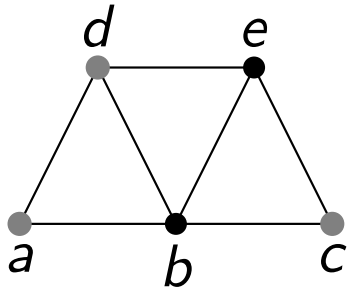
Eliminationschema
a, c,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



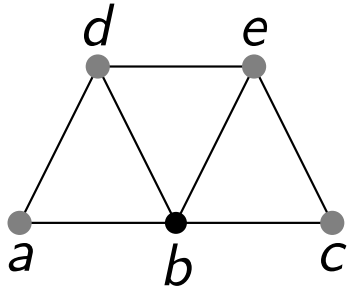
Eliminationschema
a, c, d,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



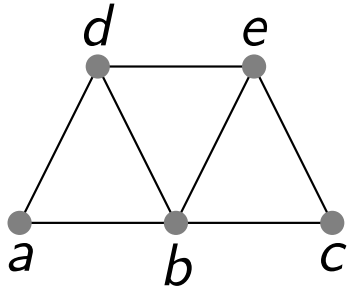
Eliminationschema
a, c, d, e,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema

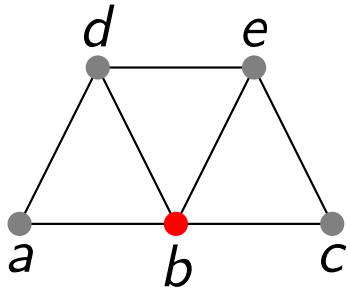
a, c, d, e, b

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



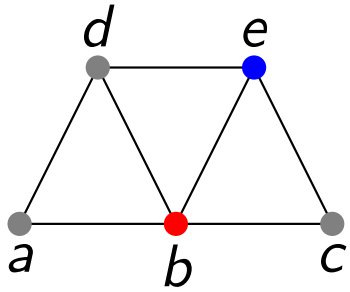
Eliminationschema
a, c, d, e,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



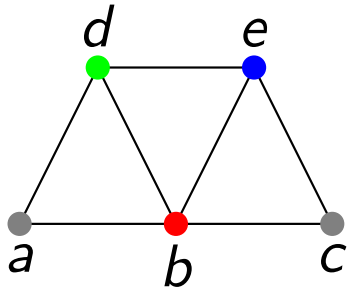
Eliminationschema
a, c, d,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



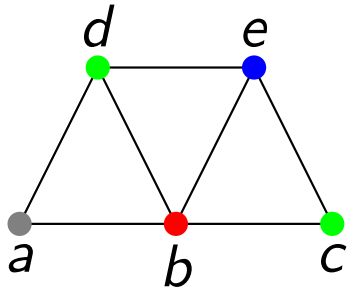
Eliminationschema
a, c,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



Eliminationschema

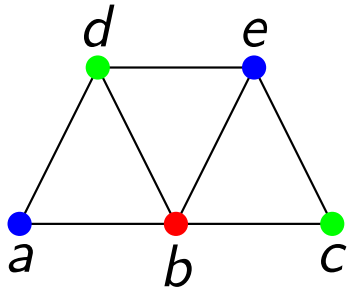
a,

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



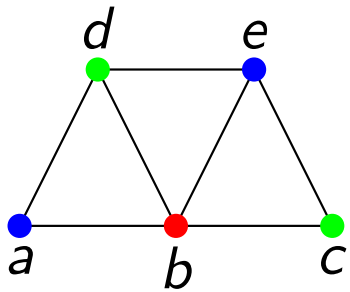
Eliminationschema

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Annahme: alle (noch nicht eliminierten) Nachbarn einer Ecke n bilden eine Clique



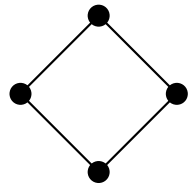
Eliminationschema

Theorem (Graphentheorie)

- *Ein PES erlaubt eine optimale Färbung in polynomieller Zeit*
- *Die Anzahl der benötigten Farben ist durch die größte Clique beschränkt*

Perfekte Eliminationsschemata

- Graphen mit Untergraphen, die Zyklen größer 3 sind, haben keine PES, z.B.



- Graphen, die ein PES besitzen, sind *chordal*
- Chordale Graphen sind *perfekt*, d.h.

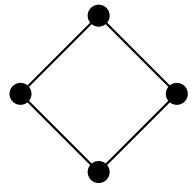
$$\chi(H) = \omega(H) \text{ für jedes } H \subseteq G$$

Theorem

- Die Dominanzrelation von SSA-Programmen induziert ein PES im IG.
- SSA IGs sind also chordal

Perfekte Eliminationsschemata

- Graphen mit Untergraphen, die Zyklen größer 3 sind, haben keine PES, z.B.



- Graphen, die ein PES besitzen, sind *chordal*
- Chordale Graphen sind *perfekt*, d.h.

$$\chi(H) = \omega(H) \text{ für jedes } H \subseteq G$$

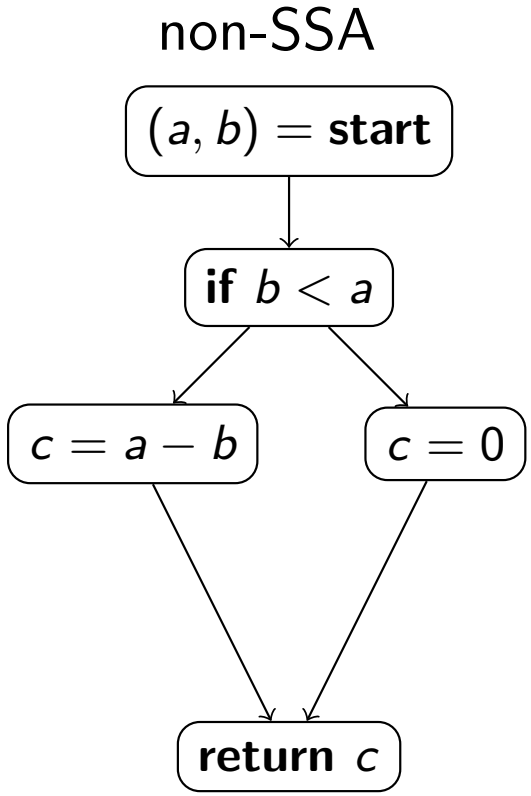
Theorem

- Die Dominanzrelation von SSA-Programmen induziert ein PES im IG.
- SSA IGs sind also chordal

SSA-Form

Beweis – SSA IGs sind chordal

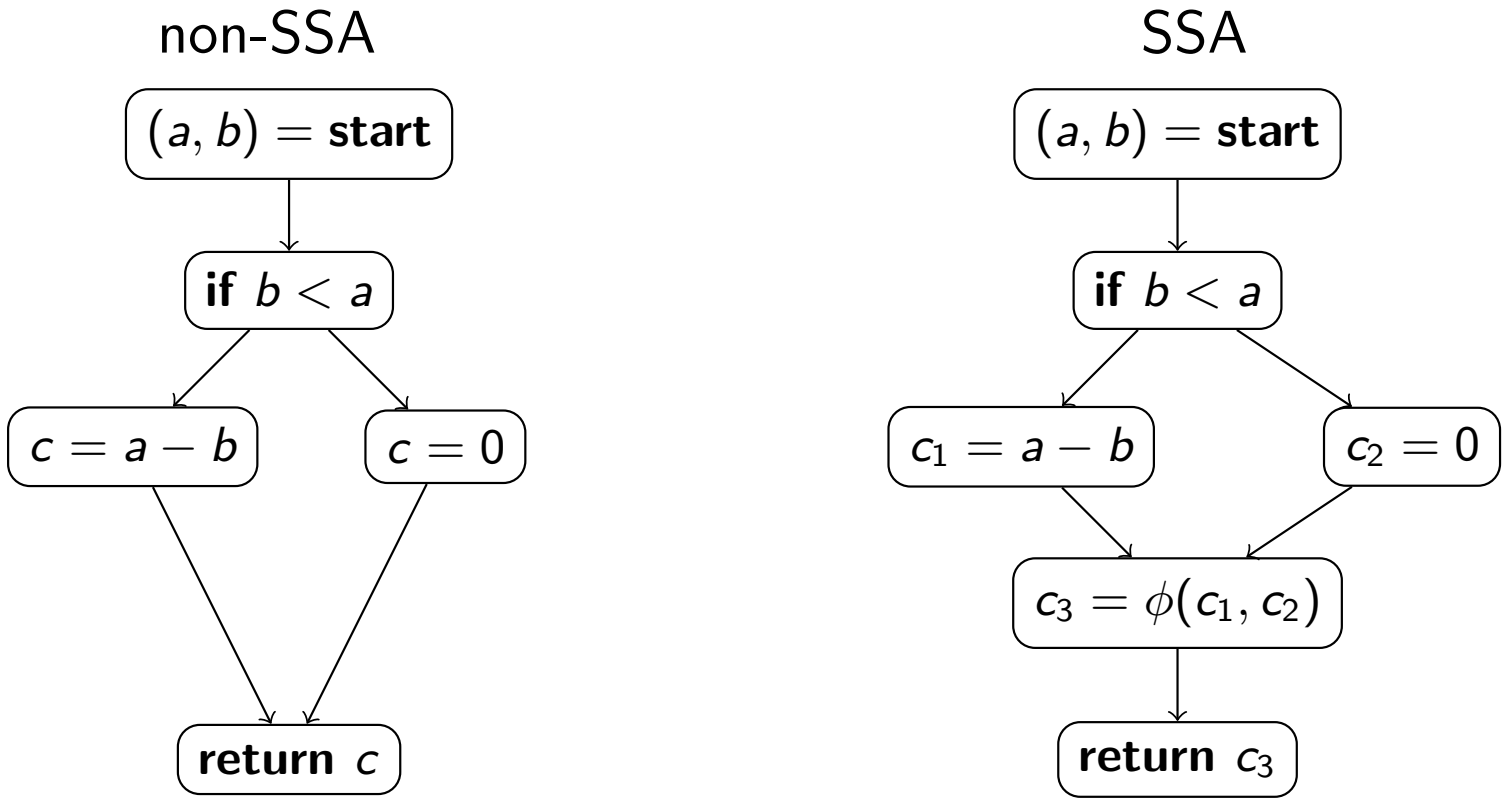
- Jede Variable hat genau eine Definition
  - Variablen sind dynamische Konstanten (Werte)
- ϕ-Funktionen wählen Werte abhängig vom Steuerfluss aus



SSA-Form

Beweis – SSA IGs sind chordal

- Jede Variable hat genau eine Definition
  - Variablen sind dynamische Konstanten (Werte)
- ϕ-Funktionen wählen Werte abhängig vom Steuerfluss aus



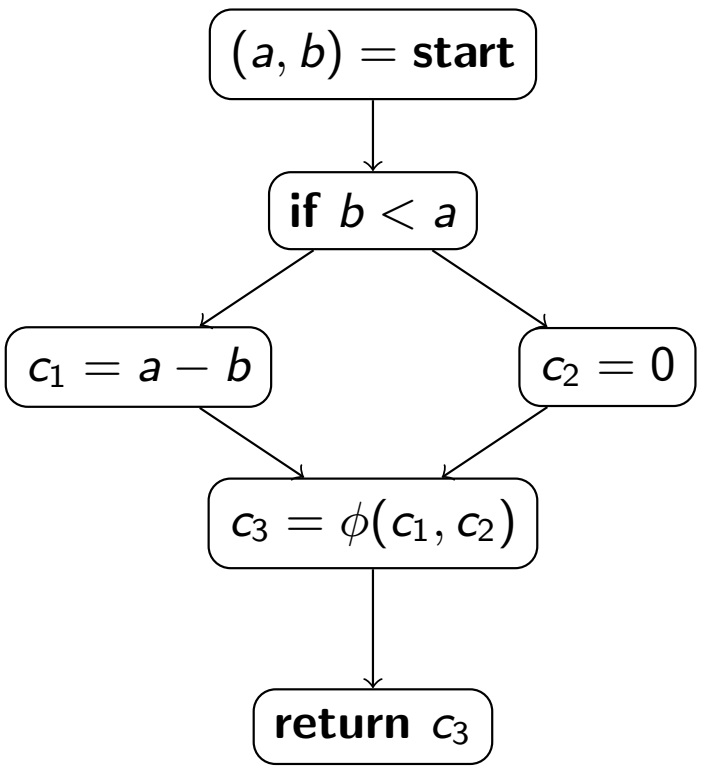
Dominanzrelation

Beweis – SSA IGs sind chordal

Entscheidend für SSA-Programme ist die Dominanzrelation:

Definition

ℓ_1 dominatiert ℓ_2 wenn jeder Pfad von **Start** nach ℓ_2 über ℓ_1 führt



- Jede Ecke hat einen eindeutigen *direkten Dominator*
- Die Dominanzrelation induziert also einen Baum im Steuerflussgraphen
- Die Dominanzrelation ist somit eine Halbordnung

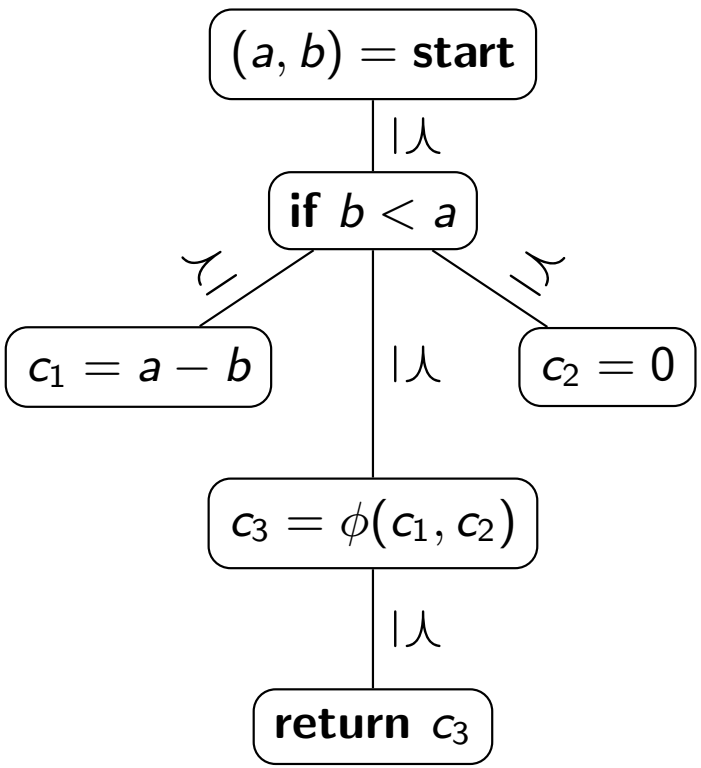
Dominanzrelation

Beweis – SSA IGs sind chordal

Entscheidend für SSA-Programme ist die Dominanzrelation:

Definition

ℓ_1 dominatiert ℓ_2 wenn jeder Pfad von **Start** nach ℓ_2 über ℓ_1 führt

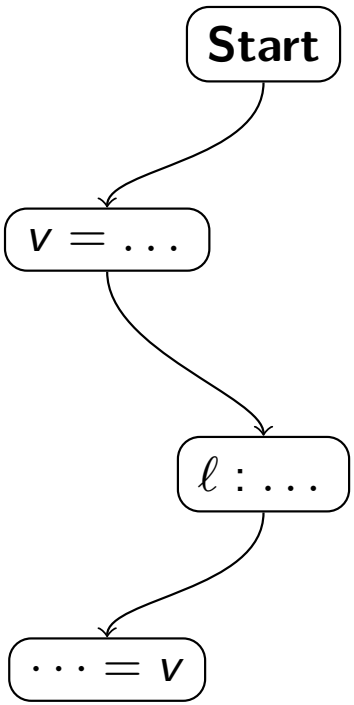


- Jede Ecke hat einen eindeutigen *direkten Dominator*
- Die Dominanzrelation induziert also einen Baum im Steuerflussgraphen
- Die Dominanzrelation ist somit eine Halbordnung

Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



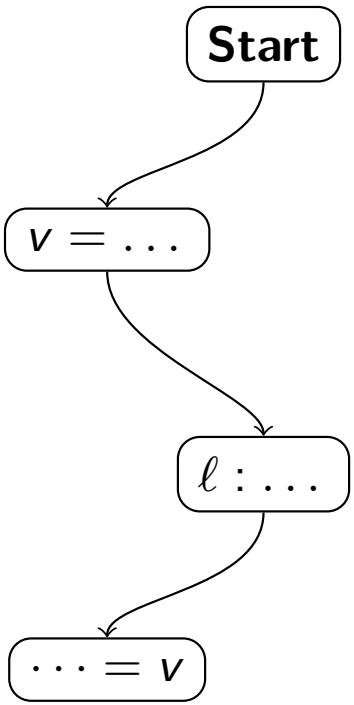
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.
- Dann existiert ein Pfad von **Start** zu einer Benutzung von v, der die Definition von v nicht enthält.
- Widerspruch: Jeder Wert muss vor seiner Benutzung definiert werden.

Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



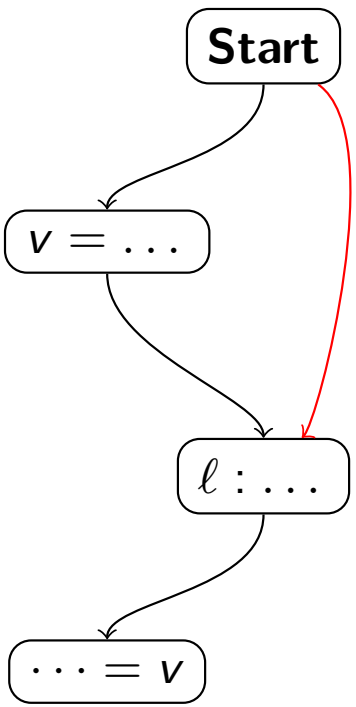
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.
- Dann existiert ein Pfad von **Start** zu einer Benutzung von v, der die Definition von v nicht enthält.
- Widerspruch: Jeder Wert muss vor seiner Benutzung definiert werden.

Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



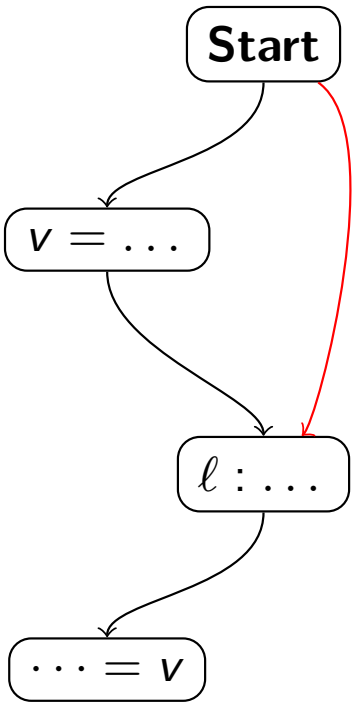
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.
- Dann existiert ein Pfad von **Start** zu einer Benutzung von v, der die Definition von v nicht enthält.
- Widerspruch: Jeder Wert muss vor seiner Benutzung definiert werden.

Lebendigkeit und Dominanz

Budimlić, PLDI '02

- Jede Programmstelle ℓ, an der ein Wert v lebendig ist, wird durch die Definition von v dominiert.
- Wir schreiben: $\mathcal{D}_v \preceq \ell$



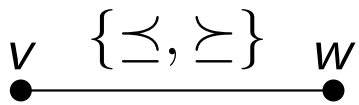
Widerspruchsbeweis

- Sei ℓ nicht von $\mathcal{D}_v$ dominiert.
- Dann existiert ein Pfad von **Start** zu einer Benutzung von v, der die Definition von v nicht enthält.
- Widerspruch: Jeder Wert muss vor seiner Benutzung definiert werden.

Interferenz und Dominanz I

Beweis – SSA IGs sind chordal

- Annahme: v, w interferieren, d.h. sie sind beide an einer Programmstelle ℓ lebendig
- Dann gilt: $\mathcal{D}_v \preceq \ell$ und $\mathcal{D}_w \preceq \ell$
- Da die Dominanzrelation einen Baum bildet, gilt entweder $\mathcal{D}_v \preceq \mathcal{D}_w$ oder $\mathcal{D}_w \preceq \mathcal{D}_v$



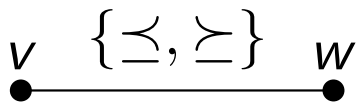
Folgerungen

- Jede Kante im IG ist in Bezug auf die Dominanz gerichtet
- Der IG ist ein "Extrakt" der Dominanzrelation

Interferenz und Dominanz I

Beweis – SSA IGs sind chordal

- Annahme: v, w interferieren, d.h. sie sind beide an einer Programmstelle ℓ lebendig
- Dann gilt: $\mathcal{D}_v \preceq \ell$ und $\mathcal{D}_w \preceq \ell$
- Da die Dominanzrelation einen Baum bildet, gilt entweder $\mathcal{D}_v \preceq \mathcal{D}_w$ oder $\mathcal{D}_w \preceq \mathcal{D}_v$



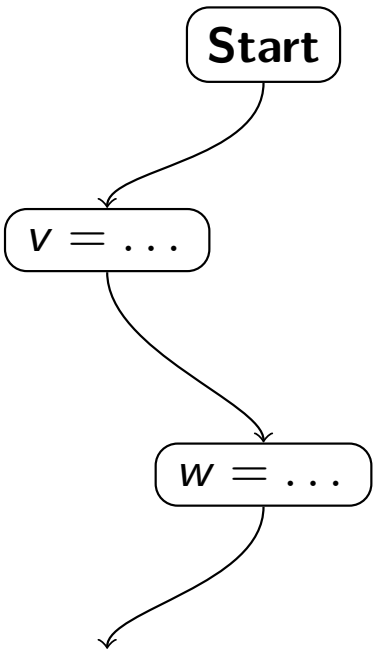
Folgerungen

- Jede Kante im IG ist in Bezug auf die Dominanz gerichtet
- Der IG ist ein “Extrakt” der Dominanzrelation

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



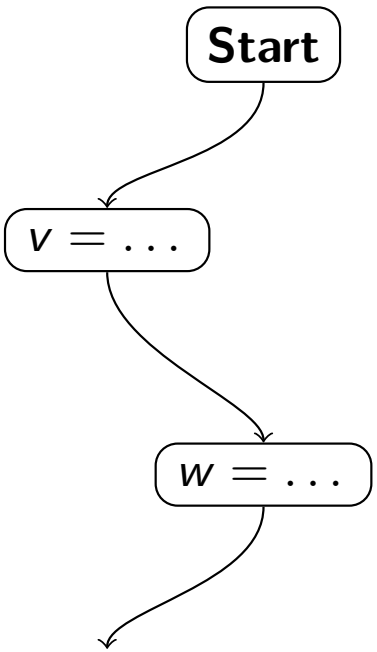
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$
- Dann existiert kein Pfad von $\mathcal{D}_w$ zu einer beliebigen Benutzung von v
- Also interferieren v und w nicht $\lightning$

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



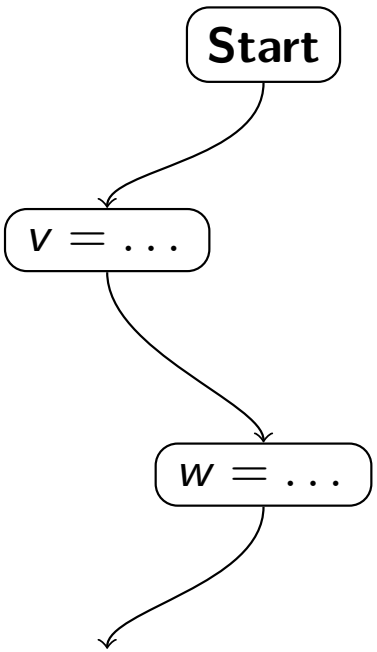
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$
- Dann existiert kein Pfad von $\mathcal{D}_w$ zu einer beliebigen Benutzung von v
- Also interferieren v und w nicht ↯

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



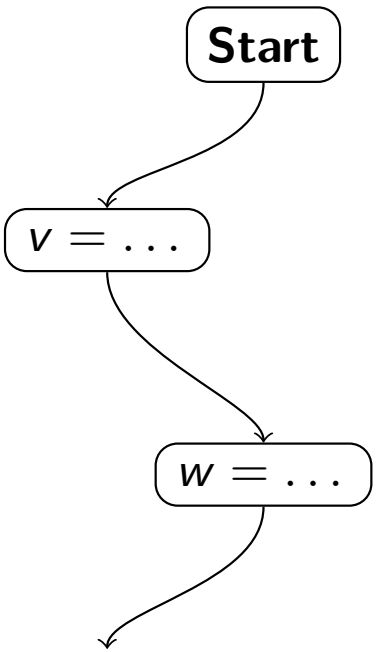
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$
- Dann existiert kein Pfad von $\mathcal{D}_w$ zu einer beliebigen Benutzung von v
- Also interferieren v und w nicht ↯

Interferenz and Dominanz II

Budimlić, PLDI '02

- Sei $v \preceq w$
- Dann ist v lebendig an $\mathcal{D}_w$



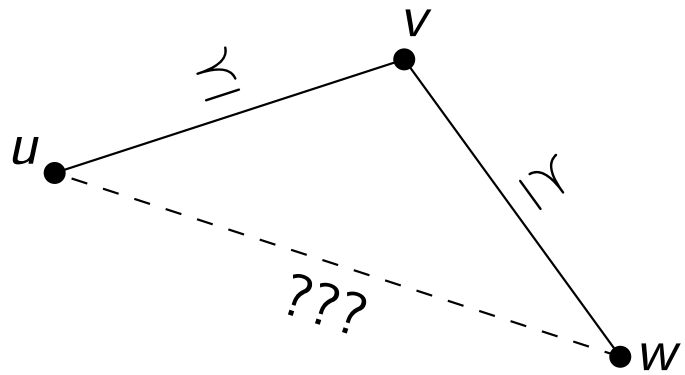
Widerspruchsbeweis

- Sei v nicht lebendig an $\mathcal{D}_w$
- Dann existiert kein Pfad von $\mathcal{D}_w$ zu einer beliebigen Benutzung von v
- Also interferieren v und w nicht $\lightning$

Interferenz and Dominanz III

Beweis – SSA IGs sind chordal

- Betrachten wir drei Ecken u, v, w im IG:



  - u, w sind lebendig an $\mathcal{D}_v$
  - Somit interferieren sie

Schlussfolgerung

Alle Werte

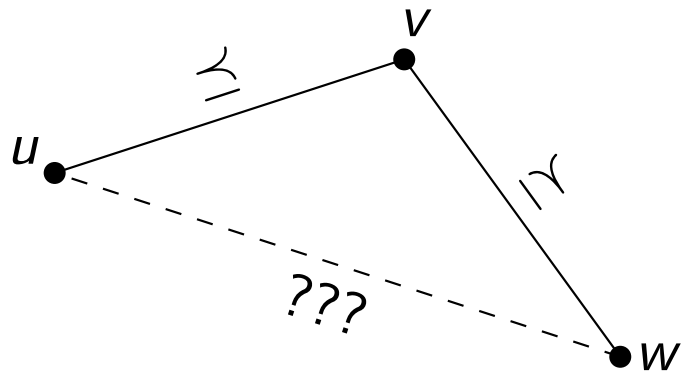
  - Interferieren mit v
  - Ihre Definitionen dominieren die Definition von v

sind Mitglieder derselben Clique

Interferenz and Dominanz III

Beweis – SSA IGs sind chordal

- Betrachten wir drei Ecken u, v, w im IG:



- u, w sind lebendig an $\mathcal{D}_v$
- Somit interferieren sie

Schlussfolgerung

Alle Werte

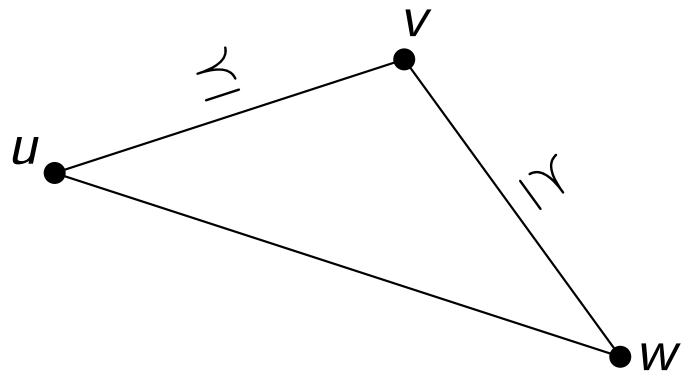
- Interferieren mit v
- Ihre Definitionen dominieren die Definition von v

sind Mitglieder derselben Clique

Interferenz and Dominanz III

Beweis – SSA IGs sind chordal

- Betrachten wir drei Ecken u, v, w im IG:



- u, w sind lebendig an $\mathcal{D}_v$
- Somit interferieren sie

Schlussfolgerung

Alle Werte

- Interferieren mit v
- Ihre Definitionen dominieren die Definition von v

sind Mitglieder derselben Clique

Dominanz and PESs

- Bevor ein Wert v zu einem PES hinzugefügt wird, füge alle Werte, deren Definitionen von $\mathcal{D}_v$ dominiert werden, ein
- Somit führt eine Post-Order Besuchsreihenfolge des Dominanzbaums zu einem PES
- IGs von SSA-Programmen können in $O(\chi(G) \cdot |V|)$ gefärbt werden
- Dazu muss der IG selbst nicht einmal berechnet werden

Inhalt

Auslagern

Theorem

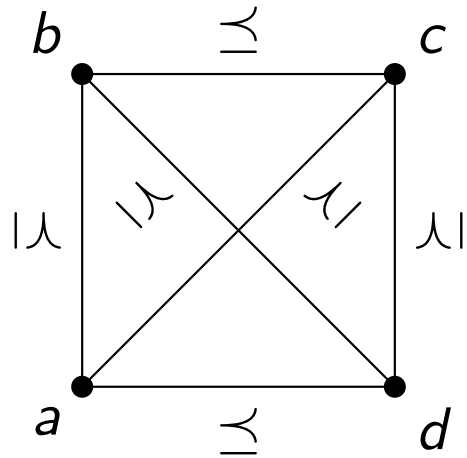
Für jede Clique in einem IG existiert eine Programmstelle, an der alle Ecken der Clique lebendig sind.

- Die Dominanz induziert also eine *totale Ordnung* innerhalb der Clique
  $\Rightarrow$ Es existiert ein "größter" Wert d
- Alle anderen sind lebendig an der Definitionsstelle von d
- Existieren nur drei Register, muss a, b oder c an $\mathcal{D}_d$ ausgelagert sein

Auslagern

Theorem

Für jede Clique in einem IG existiert eine Programmstelle, an der alle Ecken der Clique lebendig sind.



- Die Dominanz induziert also eine *totale Ordnung* innerhalb der Clique
  $\Rightarrow$ Es existiert ein "größter" Wert d
- Alle anderen sind lebendig an der Definitionsstelle von d
- Existieren nur drei Register, muss a, b oder c an $\mathcal{D}_d$ ausgelagert sein

Auslagern

Konsequenzen

- Die chromatische Zahl des IG ist exakt durch die Anzahl der lebendigen Werte an den Programmstellen festgelegt
- Reduktion der Anzahl der lebendigen Werte an jeder Programmstelle auf k mach den IG k-färbbar
- Wir kennen a-priori die Programmstellen, an denen Werte ausgelagert werden müssen
  - An allen Programmstellen mit einem Registerdruck größer k
- Auslagern kann also vor dem Färben durchgeführt werden und
- Färben mit k Farben gelingt danach stets

Schlussfolgerung

- Im Gegensatz zu Chaitin/Briggs-Zuteilern keine Iteration notwendig
- Der IG muss höchstens einmal aufgebaut werden (falls überhaupt)

Auslagern

Konsequenzen

- Die chromatische Zahl des IG ist exakt durch die Anzahl der lebendigen Werte an den Programmstellen festgelegt
- Reduktion der Anzahl der lebendigen Werte an jeder Programmstelle auf k mach den IG k-färbbar
- Wir kennen a-priori die Programmstellen, an denen Werte ausgelagert werden müssen
  - An allen Programmstellen mit einem Registerdruck größer k
- Auslagern kann also vor dem Färben durchgeführt werden und
- Färben mit k Farben gelingt danach stets

Schlussfolgerung

- Im Gegensatz zu Chaitin/Briggs-Zuteilern keine Iteration notwendig
- Der IG muss höchstens einmal aufgebaut werden (falls überhaupt)

Auslagern

Allgemein zu lösende Probleme beim Auslagern

- Wenn mehrere Werte ausgelagert werden könnten, welchen wählen?
- Wo genau platziert man die Auslagerungs-/Einlagerungsinstruktionen?
- Ist die Wiederberechnung (Rematerialisierung) manchmal günstiger als ein Auslagern?
  - Konstanten
  - Prozedurargumente auf dem Keller
  - Einfache Berechnungen ...
- Auslagerungsplätze zusammenfassen (um Platz zu sparen)?

Farach und Liberatore

- Optimales Auslagern innerhalb eines Grundblocks ist NP-vollständig.

Auslagern

Allgemein zu lösende Probleme beim Auslagern

- Wenn mehrere Werte ausgelagert werden könnten, welchen wählen?
- Wo genau platziert man die Auslagerungs-/Einlagerungsinstruktionen?
- Ist die Wiederberechnung (Rematerialisierung) manchmal günstiger als ein Auslagern?
  - Konstanten
  - Prozedurargumente auf dem Keller
  - Einfache Berechnungen ...
- Auslagerungsplätze zusammenfassen (um Platz zu sparen)?

Farach und Liberatore

- Optimales Auslagern innerhalb eines Grundblocks ist NP-vollständig.

Auslagern

Spezielle SSA-Probleme

- ϕ-Funktionen werden gleichzeitig und am Anfang eines Grundblocks ausgeführt, deshalb kann weder vor noch zwischen ihnen aus-/eingelagert werden
- Deshalb müssen Werte vor Betreten des Grundblocks ausgelagert werden
  - Argumente von ϕ-Funktionen sind ausgelagert
  - ϕ-Funktionen bei denen alle Argumente ausgelagert sind benötigen kein Register
  - Vorsicht: Beim SSA-Abbau entstehen hier Speicherkopien statt Registerkopien

Auslagern

Implementierung

- Heuristik nach Belady:
  - Bevorzuge diejenigen Werte bei der Auslagerung, deren nächste Benutzung am weitesten in der Zukunft liegt
    - Was ist die "Entfernung" bei Verzweigungen?
  - Beachte: Benutzungen außerhalb von Schleifen sind stets "weiter" in der Zukunft als innerhalb!
  - Einlagerungsinstruktionen möglich vor einer Schleife platzieren
- Auch als ILP formulierbar

Inhalt

SSA-Abbau

- Gegeben eine k-Färbung eines SSA-IGs
- Können wir daraus eine gültige Registerzuteilung mit k Registern für das zugehörige non-SSA Programm erzeugen?

Zentrale Frage

Wie behandeln wir ϕ-Funktionen?

SSA-Abbau

- Gegeben eine k-Färbung eines SSA-IGs
- Können wir daraus eine gültige Registerzuteilung mit k Registern für das zugehörige non-SSA Programm erzeugen?

Zentrale Frage

Wie behandeln wir ϕ-Funktionen?

ϕ-Funktionen

- Alle ϕ-Funktionen in einem Grundblock

$$\begin{aligned} y_1 &\leftarrow \phi(x_{11}, \ldots, x_{n1}) \\ &\vdots \\ y_m &\leftarrow \phi(x_{1m}, \ldots, x_{nm}) \end{aligned}$$

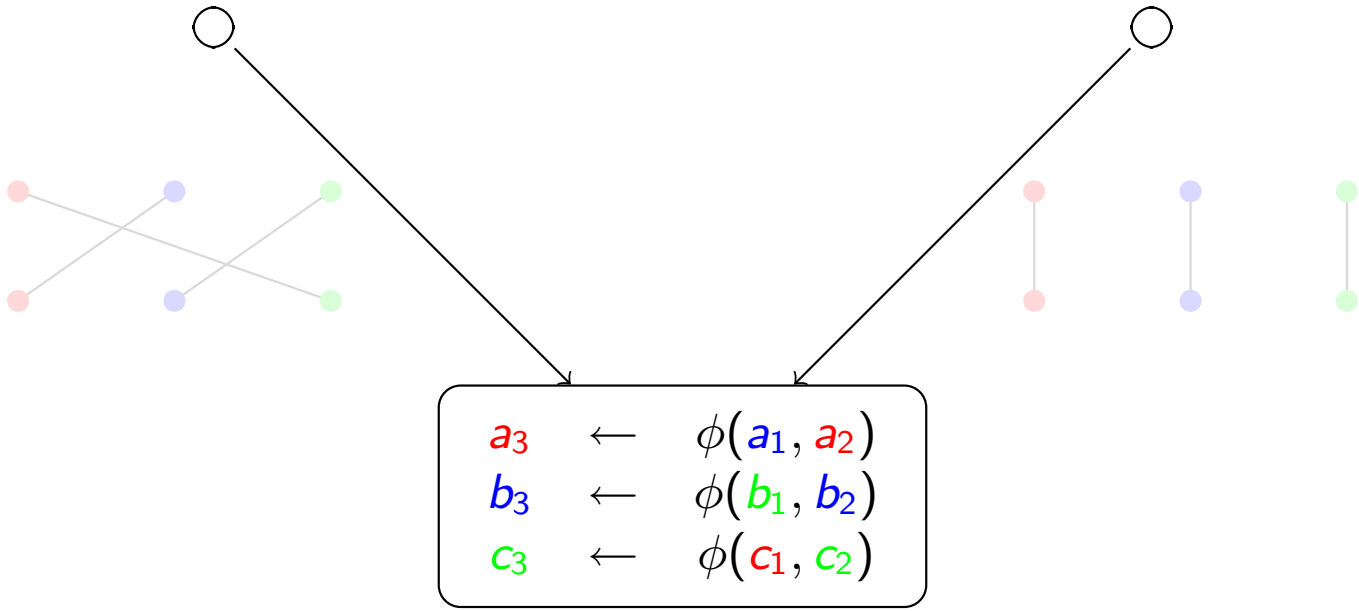
  werden gleichzeitig und vor allen anderen Instruktionen in diesem Grundblock ausgeführt.

- Betreten wir diesen Block über die i-te Kante, wirken die ϕ-Funktionen wie eine parallele Kopierinstruktion

$$(y_1, \ldots, y_m) \leftarrow (x_{i1}, \ldots, x_{im})$$

ϕ-Funktionen

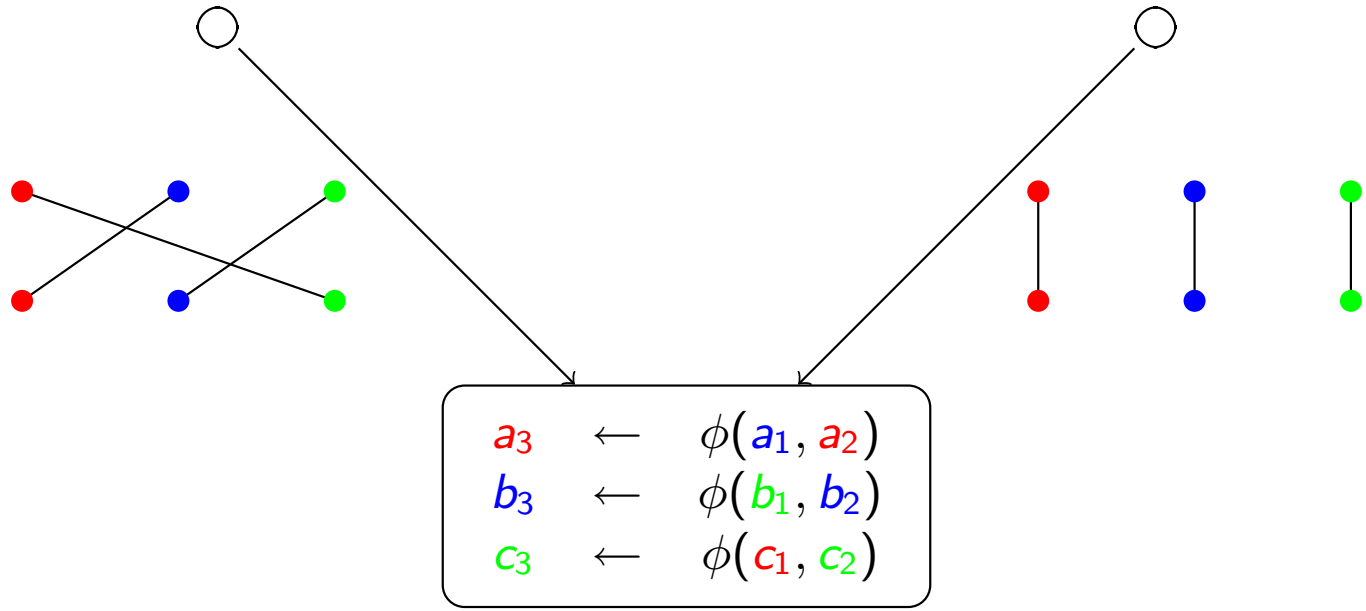
- Beispiel



- Die ϕs stellen Registerpermutationen auf den Steuerflusskanten dar

ϕ-Funktionen

- Beispiel



- Die ϕs stellen Registerpermutationen auf den Steuerflusskanten dar

ϕ-Funktionen

- Beispiel

$$
\begin{aligned}
a_3 &\leftarrow \phi(a_1, a_2) \\
b_3 &\leftarrow \phi(b_1, b_2) \\
c_3 &\leftarrow \phi(c_1, c_2)
\end{aligned}
$$

- Die ϕs stellen Registerpermutationen auf den Steuerflusskanten dar

Auf Kante 1

Auf Kante 2

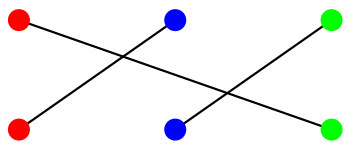
ϕ-Funktionen

- Beispiel

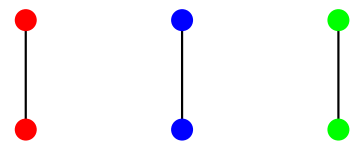
$$
\begin{aligned}
a_3 &\leftarrow \phi(a_1, a_2) \\
b_3 &\leftarrow \phi(b_1, b_2) \\
c_3 &\leftarrow \phi(c_1, c_2)
\end{aligned}
$$

- Die ϕs stellen Registerpermutationen auf den Steuerflusskanten dar

Auf Kante 1

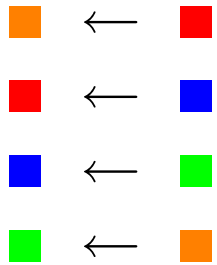


Auf Kante 2



Permutationen

- Eine Permutation kann mit Hilfe von Kopien implementiert werden, wenn ein Hilfsregister ■ verfügbar ist

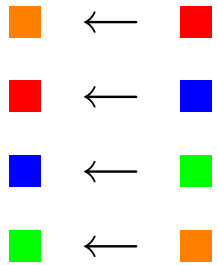


- Permutationen können als Folge von Transpositionen (also Vertauschungen) implementiert werden

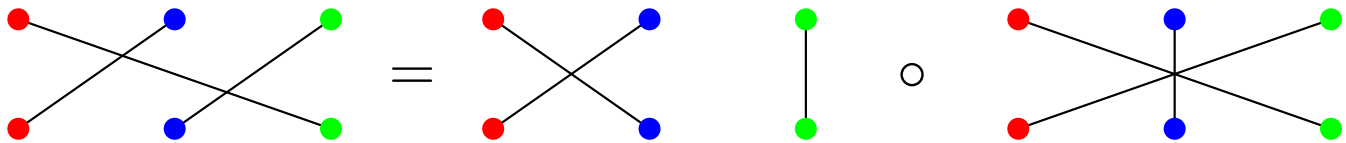
- Eine Transposition kann als Folge von drei `xor`-Instruktionen ohne Verwendung eines zusätzlichen Registers implementiert werden

Permutationen

- Eine Permutation kann mit Hilfe von Kopien implementiert werden, wenn ein Hilfsregister ■ verfügbar ist



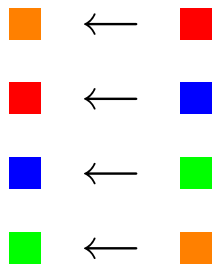
- Permutationen können als Folge von Transpositionen (also Vertauschungen) implementiert werden



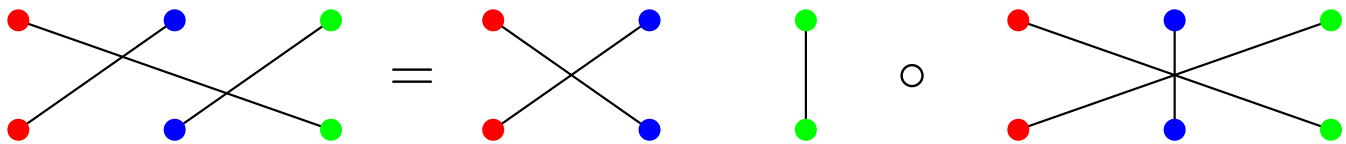
- Eine Transposition kann als Folge von drei `xor`-Instruktionen ohne Verwendung eines zusätzlichen Registers implementiert werden

Permutationen

- Eine Permutation kann mit Hilfe von Kopien implementiert werden, wenn ein Hilfsregister ■ verfügbar ist



- Permutationen können als Folge von Transpositionen (also Vertauschungen) implementiert werden



- Eine Transposition kann als Folge von drei `xor`-Instruktionen ohne Verwendung eines zusätzlichen Registers implementiert werden

Unterschiede zum klassischen SSA-Abbau

Parallele Kopien

$$(a', b', c', d') \leftarrow (a, b, c, d)$$

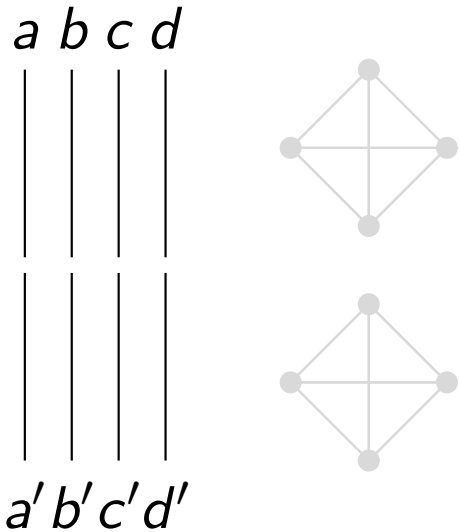
Sequentielle Kopien

$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$

Unterschiede zum klassischen SSA-Abbau

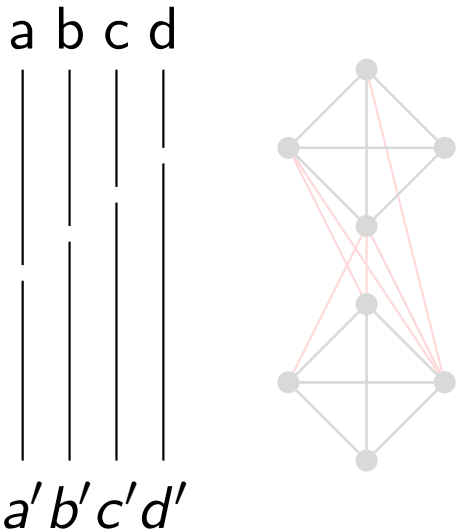
Parallele Kopien

$$(a', b', c', d') \leftarrow (a, b, c, d)$$



Sequentielle Kopien

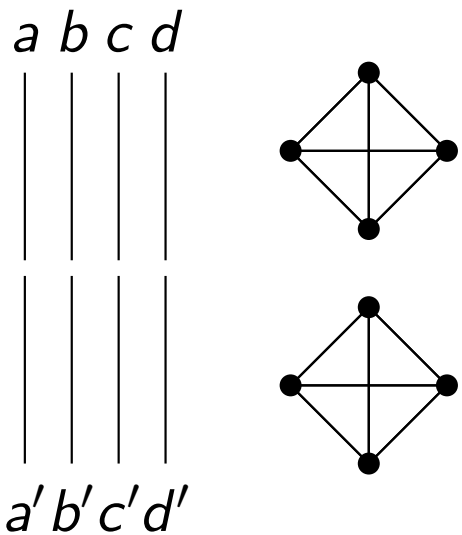
$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$



Unterschiede zum klassischen SSA-Abbau

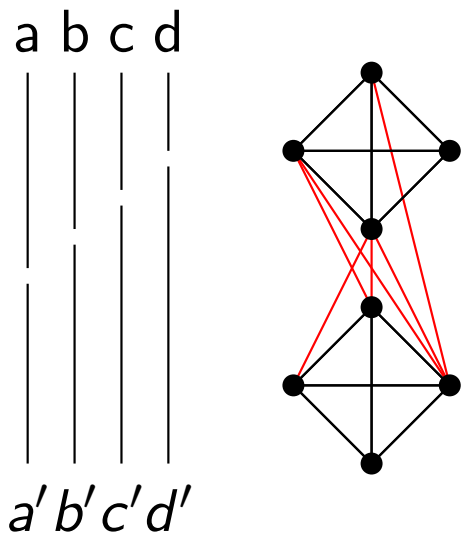
Parallele Kopien

$$(a', b', c', d') \leftarrow (a, b, c, d)$$



Sequentielle Kopien

$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$



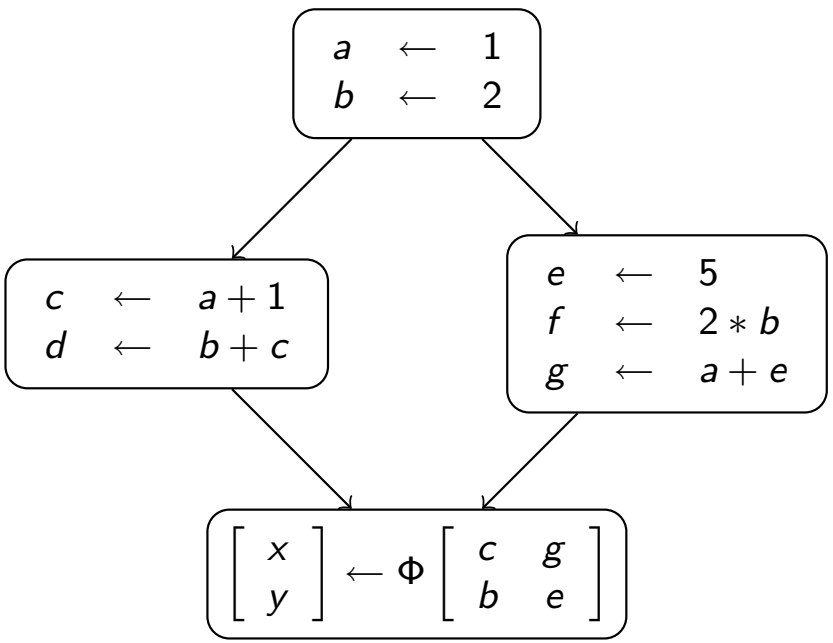
Verschmelzung

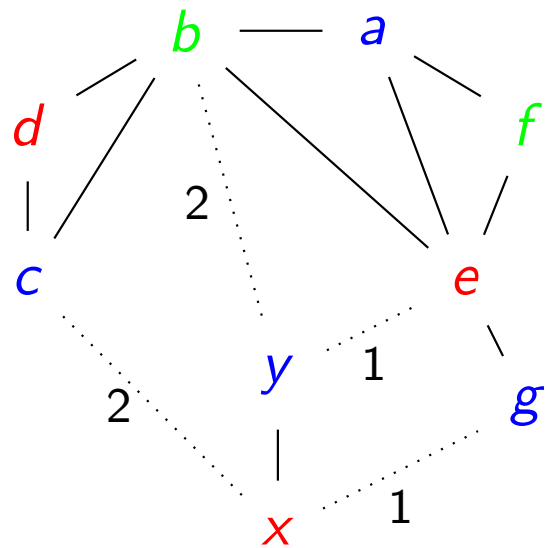
- Minimiere die Anzahl der dynamisch ausgeführten (statisch eingefügten) Kopierinstruktionen
- Üblich: Verschmelzung von Ecken im IG, um Kopien zu vermeiden
  - Führt zu nicht-chordalen Graphen
    - Man verliert die sichere Kenntnis der chromatischen Zahl
  - Werden aggressiv Ecken verschmolzen, können Spills entstehen, um Kopien zu vermeiden

Verschmelzung

Problemmodell

- Gegeben: Eine minimale Färbung des IG
- Gesucht: Eine erlaubte ($< k$) Färbung mit minimalen Kosten
  - Kosten sind die gewichtete Summe der Gleichfärbekanten
  - Unbenutzte Farben dürfen verwendet werden
  - Weder IG noch Programm dürfen geändert werden

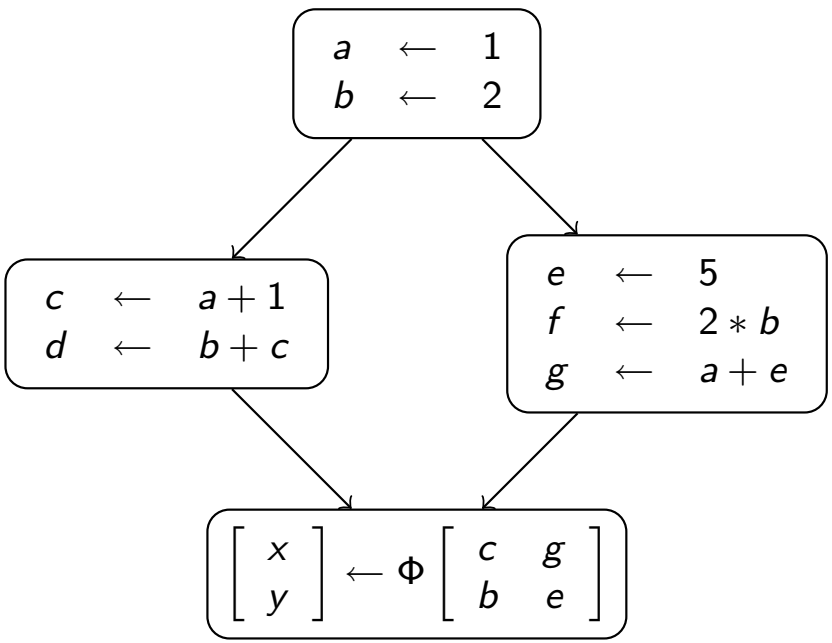


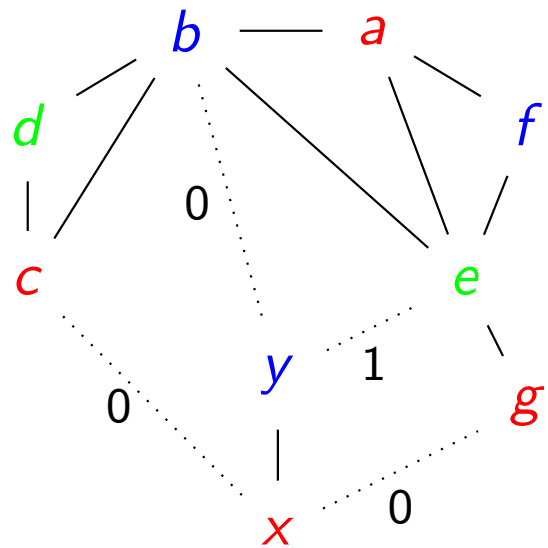


Verschmelzung

Problemmodell

- Gegeben: Eine minimale Färbung des IG
- Gesucht: Eine erlaubte ($< k$) Färbung mit minimalen Kosten
  - Kosten sind die gewichtete Summe der Gleichfärbekanten
  - Unbenutzte Farben dürfen verwendet werden
  - Weder IG noch Programm dürfen geändert werden





Verschmelzung

Formal

- Finde eine k-Färbung $\mathcal{C}$ des IG, die so vielen ϕ-Operanden und Ergebnissen wie möglich die gleiche Farbe zuweist.

$$\min_{\mathcal{C}} \sum_{\phi} costs(\mathcal{C}, \phi)$$

mit

$$costs\left(\mathcal{C}, y \leftarrow \phi(x_1, \ldots, x_n)\right) = \sum_{i=1}^{n} \begin{cases} 0 & \text{falls } \mathcal{C}(y) = \mathcal{C}(x_i) \\ w_{yx_i} & \text{sonst} \end{cases}$$

Verschmelzung

Lösungsstrategien

Komplexität

- Problem ist NP-vollständig in der Anzahl der ϕs

Algorithmen

- Eine Greedy-Heuristik
- Eine optimale Methode, die ILP[2] benutzt

[2] Integer Linear Programming

Verschmelzung

Heuristik

- Idee: Ändere die Farben um besser zusammenpassende Paare zu erhalten
- Problem: Ob der Farbwechsel möglich ist, ist nicht lokal entscheidbar
- Darum
  - Betrachte jedes ϕ separat
  - Versuche den ϕ-Operanden und dem Ergebnis dieselbe Farbe zu geben
  - Löse Farbunverträglichkeiten rekursiv durch den IG
  - Falls dies fehlschlägt markiere den Konflikt lokal und setze fort

Verschmelzung

Formalisierung als ILP

- Entscheidungsvariablen stellen Zustände / Färbungen dar
- Färbung: $x_{ic} = 1 \Leftrightarrow$ Ecke i hat Farbe c
- Zulässigkeit: $y_{ij} = 1 \Leftrightarrow$ Ecke i und j haben verschiedene Farben

$$
\begin{aligned}
\min f = \quad & \sum_{e \in Q} w_e \cdot y_{ij} & \\
\textit{mit} \quad & \sum_{c} x_{ic} = 1 & v_i \in V \\
& x_{ic} + x_{jc} \leq 1 & [v_i, v_j] \in E \\
& y_{ij} \geq x_{ic} - x_{jc} & [v_i, v_j] \in Q \\
& \\
& y_{ij}, x_{ic} \in \{0, 1\} &
\end{aligned}
$$

Verschmelzung

ILP Beispiel mit 3 Ecken und 3 Farben

$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
 & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

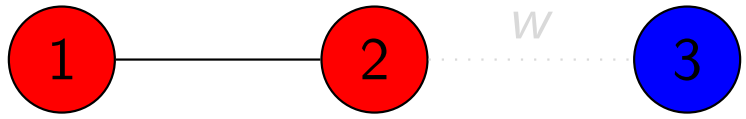
Verschmelzung

ILP Beispiel mit 3 Ecken und 3 Farben

$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
 & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

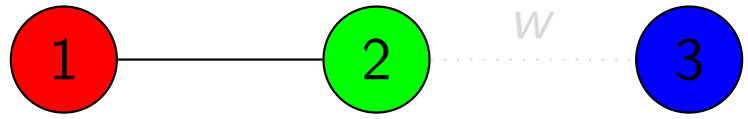
ILP Beispiel mit 3 Ecken und 3 Farben



$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
\lightning & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
\lightning & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

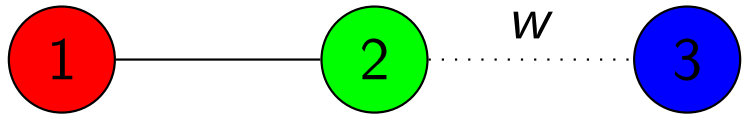
ILP Beispiel mit 3 Ecken und 3 Farben



$$
\begin{array}{lll}
\text{min} & ??? & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
& x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
& x_{31} + x_{32} + x_{33} = 1 & \\
\\
& x_{11} + x_{21} \leq 1 & \\
& x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
& x_{13} + x_{23} \leq 1 & \\
\\
& y_{23} \geq x_{21} - x_{31} & \\
& y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
& y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

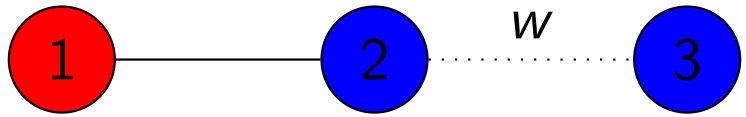
ILP Beispiel mit 3 Ecken und 3 Farben



$$
\begin{array}{lll}
\text{min} & w \cdot y_{23} & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
\searrow & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

ILP Beispiel mit 3 Ecken und 3 Farben

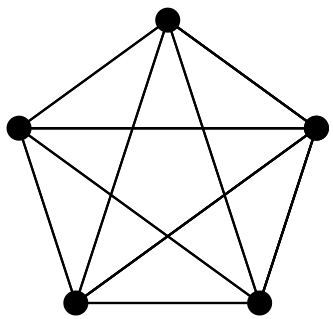


$$
\begin{array}{lll}
\text{min} & w \cdot y_{23} & \\
\textit{mit} & x_{11} + x_{12} + x_{13} = 1 & \\
 & x_{21} + x_{22} + x_{23} = 1 & \text{Färbung} \\
 & x_{31} + x_{32} + x_{33} = 1 & \\
 & & \\
 & x_{11} + x_{21} \leq 1 & \\
 & x_{12} + x_{22} \leq 1 & \text{Interferenz} \\
 & x_{13} + x_{23} \leq 1 & \\
 & & \\
 & y_{23} \geq x_{21} - x_{31} & \\
 & y_{23} \geq x_{22} - x_{32} & \text{Affinität} \\
 & y_{23} \geq x_{23} - x_{33} &
\end{array}
$$

Verschmelzung

Verbesserung der ILP Lösungszeit

- Cliquen Ungleichungen
- Pfad Ungleichungen
- Cliquen-Pfad Ungleichungen

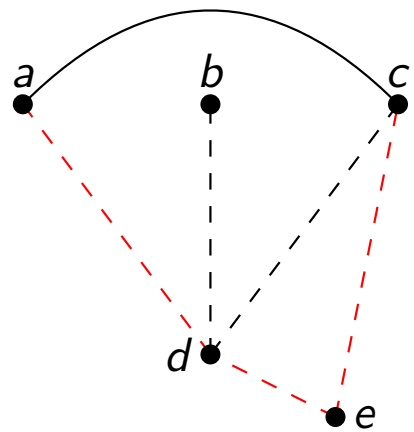


Ersetze $O(n^2)$ Ungleichungen $x_{ic} + x_{jc} \leq 1$ durch eine: $\sum_{i=1}^{n} x_{ic} \leq 1$

Verschmelzung

Verbesserung der ILP Lösungszeit

- Cliquen Ungleichungen
- Pfad Ungleichungen
- Cliquen-Pfad Ungleichungen



Verwende gegenseitigen Ausschluss von Interferenz- und Gleichfärbekanten

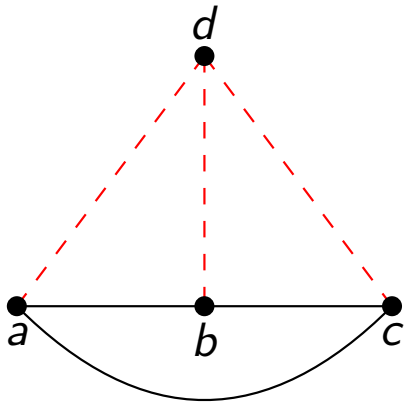
$y_{ad} + y_{cd} \geq 1$

$y_{ad} + y_{de} + y_{ec} \geq 1$

Verschmelzung

Verbesserung der ILP Lösungszeit

- Cliquen Ungleichungen
- Pfad Ungleichungen
- Cliquen-Pfad Ungleichungen



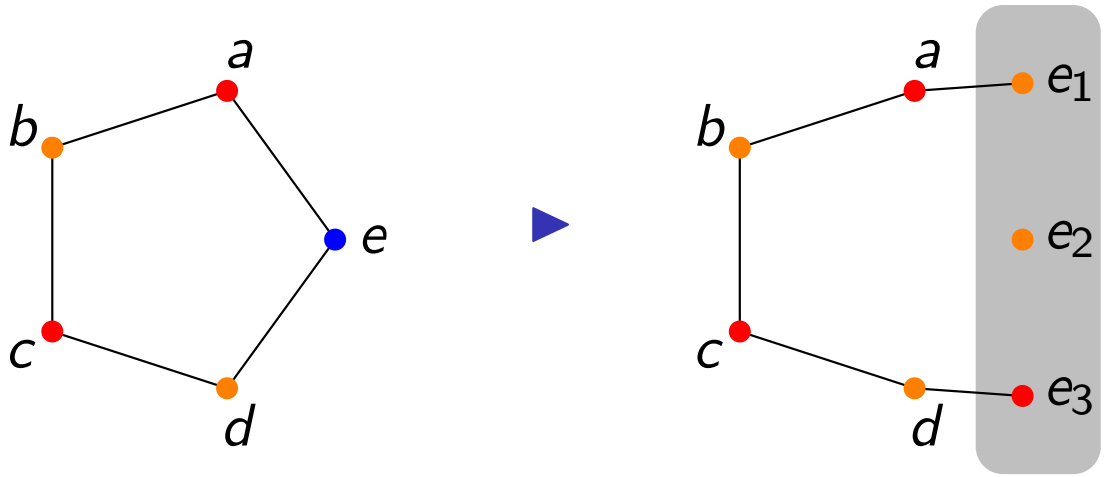
Mehrfache Verwendung desselben Arguments, z.B. $\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \Phi \begin{pmatrix} d & e \\ d & f \\ d & g \end{pmatrix}$

führt zu $y_{ad} + y_{bd} + y_{cd} \geq 2$

Schlussfolgerung

SSA-Aufbau

- SSA-Aufbau erzeugt Kopien
  (ϕ-Funktionen sind Kopien auf Steuerflusskanten)
- Diese Kopien "zerreißen" den IG
- Ecken werden durch stabile Mengen (von Ecken) ersetzt

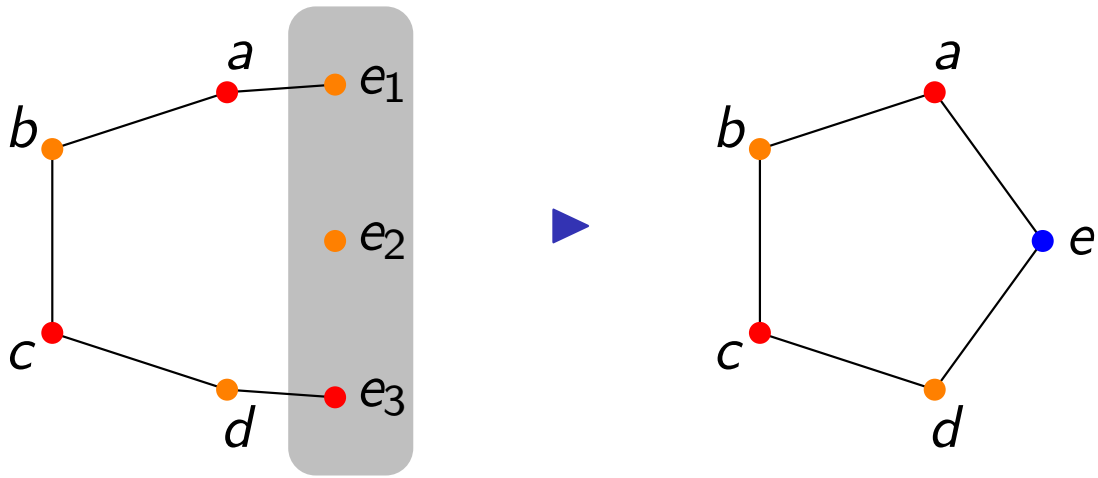


- Zyklen im IG werden aufgebrochen
- Der IG wird chordal

Schlussfolgerung

Klassischer SSA-Abbau

- SSA-Abbau verschmilzt aggressiv Kopien
  ohne die Anzahl der noch verfügbaren Register zu beachten
- Stabile Mengen werden zu Ecken



- Möglicherweise werden Zyklen erzeugt
- Dadurch kann sich die chromatische Zahl erhöhen

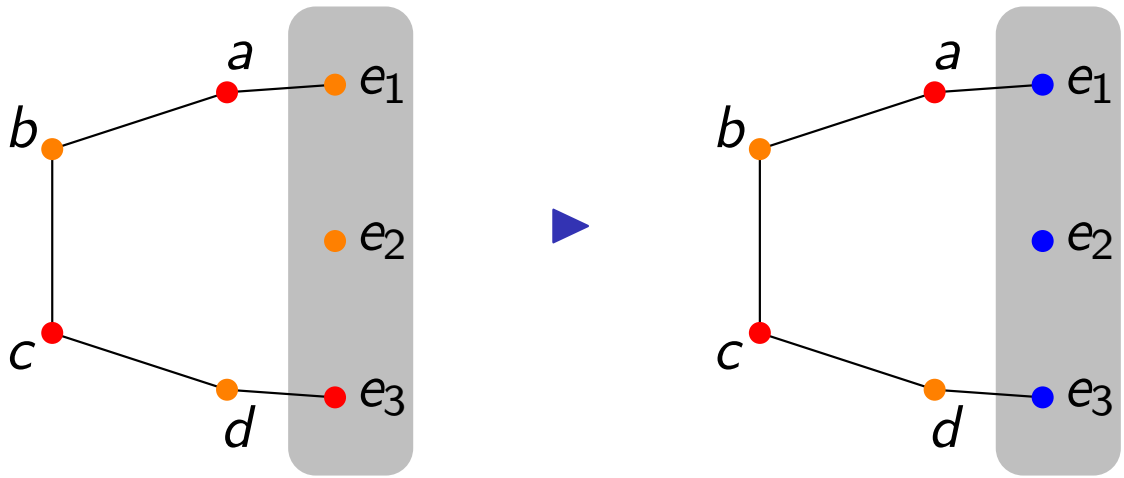
Schlussfolgerung

- Die vom SSA-Aufbau eingeführten Kopien sollten (möglichst) entfernt werden
- Sei $\chi(G) = 2$ und $k = 3$ Farben (Register) verfügbar
- Dann kann die freie Farbe für das Verschmelzen verwendet werden

- Falls nur zwei Farben existieren: Kopie von e_2 nach e_3 stehen lassen
- Erkenntnis: Es ist besser, Verschmelzen nicht als das Zusammenlegen von Ecken zu verstehen

Schlussfolgerung

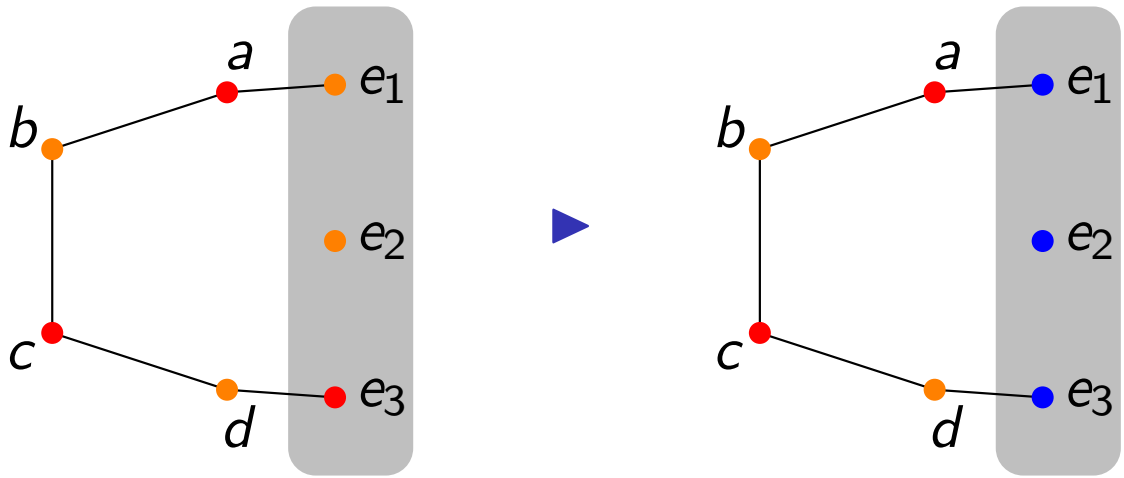
- Die vom SSA-Aufbau eingeführten Kopien sollten (möglichst) entfernt werden
- Sei $\chi(G) = 2$ und $k = 3$ Farben (Register) verfügbar
- Dann kann die freie Farbe für das Verschmelzen verwendet werden



- Falls nur zwei Farben existieren: Kopie von e_2 nach e_3 stehen lassen
- Erkenntnis: Es ist besser, Verschmelzen nicht als das Zusammenlegen von Ecken zu verstehen

Schlussfolgerung

- Die vom SSA-Aufbau eingeführten Kopien sollten (möglichst) entfernt werden
- Sei $\chi(G) = 2$ und $k = 3$ Farben (Register) verfügbar
- Dann kann die freie Farbe für das Verschmelzen verwendet werden



- Falls nur zwei Farben existieren: Kopie von e_2 nach e_3 stehen lassen
- Erkenntnis: Es ist besser, Verschmelzen nicht als das Zusammenlegen von Ecken zu verstehen

Registerzuteilung – Zusammenfassung

- Registerzuteilung ist NP-vollständig
- Hack/Goos liefert ein polynomielles Verfahren zur Graphfärbung, das die Registerzuteilung in 3 sequentielle Einzelschritte zerlegt
- Verschmelzung und Auslagerung bleibt aber NP-vollständig
- Selbst bei Verwendung heuristischer Verfahren kann die Registerzuteilung den Großteil der Übersetzungszeit brauchen
- Für große Graphen liefert *linear-scan* in vielen Anwendungsszenarien schneller eine brauchbare Lösung
- Ein weiterer Ansatz: Formulierung als ganzzahliges lineares Programm (ILP) (langsam)
- Das Zusammenspiel der zielmaschinenabhängigen Optimierungen ist offen

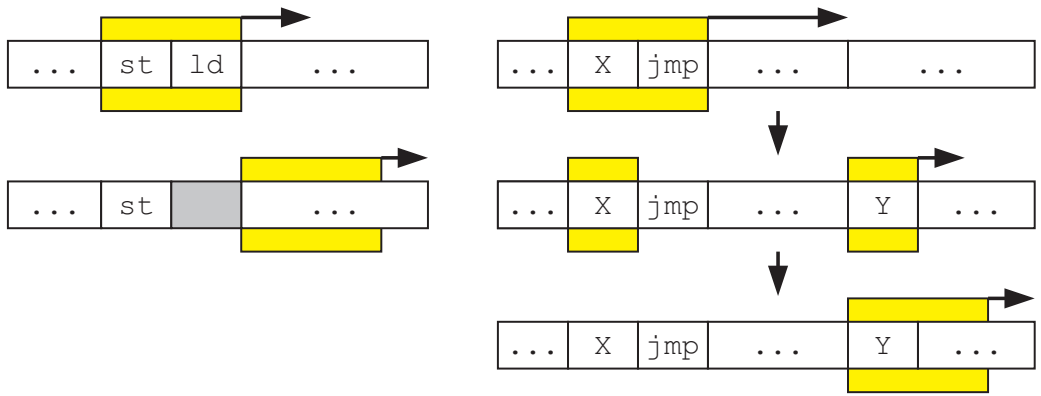
Inhalt

Nachoptimierung

- Grundidee (McKeeman, peephole optimization, 1965): schiebe ein Fenster, das n Befehle umfasst, über den erzeugten Code und ersetze die Befehlsfolge im Fenster durch eine kostengünstigere
- meist $n = 2$
- bei bedingten Sprüngen folgt das Fenster beiden Ausführungspfaden
- Grundlage der Vereinfachung: Maschinensimulation (wie bei einfacher Codeerzeugung)
- besonders wirksam bei CISC-Architekturen:
  - an Fugen zwischen dem Code unabhängiger Ausdrucksbäume
  - um komplizierte Adressierungsmodi zu nutzen: `a[i]; i++` $\longrightarrow$ `a[i++]`
  - zur Komposition oder Zerlegung von Komplexbefehlen
- Hinweis: Eigentlich vor Befehlsanordnung (da Befehle geändert werden $\longrightarrow$Pipeline), aber nicht ohne Befehls-Anordnung bzw. -Auswahl machbar.

Nachoptimierung nach Davidson/Fraser

- Methode:
  - betrachte Befehle als (endliche) Zustandstransduktoren, die den endlichen Prozessor- und Speicherzustand verändern, z.B. Befehle auf RTL-Ebene betrachten
  - berechne vorab die Wirkung von Paaren ($n = 2$) oder Tripeln ($n = 3$) solcher Befehle und überdecke sie durch andere, billigere Befehlsfolgen
  - konstruiere Tabelle ersetzbarer Befehlskombinationen
  - schiebe das Fenster über die Befehlsfolgen und ersetze entsprechend Tabelle unter Fortschaltung Maschinenzustand



Typische Beispiele

`store R,a; load a,R`	$\longrightarrow$	`store R,a`	überflüssiges Laden
`imul 2,R; xxx`	$\longrightarrow$	`ashl 2,R; xxx`	Konstantenmult.
`iadd x,R; cmp 0,R`	$\longrightarrow$	`iadd x,R`	überflüssige Vergleic
`if B then a:=x end`	$\longrightarrow$	`t:=B; a:=(t) x`	Grundblock verlänge

Weitere Nachoptimierungen I

- Zusammenziehen von Sprüngen:
  - `goto M; ... M: goto L` $\rightarrow$ `goto L; ... M: goto L`
  - `if B then goto M; ... M: if B' then goto L` $\rightarrow$
    `if B then goto L; ... M: if B' then goto L`
    falls `B'` aus `B` folgt
- Zusammenziehen von Grundblöcken, wenn der erste mit unbedingtem Sprung endigt und der zweite nur diesen Vorgänger hat
- Bedingungsumkehr und statische Sprungvorhersage:
  - `if B then goto M; L: ...` $\rightarrow$
    `if` $\neg$`B then goto L; M: ...`
    falls dies zur schnelleren Ausführung des häufigsten Pfades führt
    häufigster Pfad: Schleifen werden wiederholt und plausible Annahmen, z.B. ganze Zahlen sind nicht negativ
  - `if B then goto M; goto L: M: ...` $\rightarrow$
    `if` $\neg$`B then goto L; M: ...`

Weitere Nachoptimierungen II

- Enden verschmelzen
- leere Schleifen streichen: `while i<n loop i := i+1 end` $\longrightarrow \emptyset$
  wenn anschließend `i` nicht mehr benötigt
  Vorsicht: diese Transformation ist partiell korrekt, könnte also nicht terminierende Programme zum Terminieren bringen!
- Endrekursion beseitigen (wenn nicht vorher geschehen):
  `p(x) is M: ... p(y); return; ... end` $\longrightarrow$
  `p(x) is M: ... x := y; goto M; return; ... end`
- Ausdrücke, die Fehler verursachen durch Ausnahmeaufruf ersetzen:
  `print 1/0` $\longrightarrow$ `raise ZERO_DIVIDE`
- Explizite Operationen auf Adressierungspfad setzen:
  `R := R + const; load (R),R'` $\longrightarrow$ `load const(R),R'`
  wenn anschließend <R> nicht mehr benötigt

Nachoptimierungen – Zusammenfassung

- Automatenmodell
- viele Einzelaufgaben, wenig Systematik, oft stark prozessorabhängig
- eigentlich als Optimierungsmaßnahmen auf dem Zielcode entwickelt, z.T. aber auch vor der Codeerzeugung auf der Zwischensprache durchführbar
- Nachoptimierung beeinflusst Registervergabe, Befehlsanordnung, Konstantenfaltung, Lebendigkeitsanalyse
- Laufzeitgewinn von weit über 10% erzielbar
- manche Übersetzer (z.b. Urform lcc) benutzen nur Nachoptimierung

Zusammenfassung

- Registerzuteilung, Befehlsanordnung sind für die Funktion nötig, Nachoptimierung ist wegen Performance unerlässlich.
- Hier ist besonders zwischen Codequalität und Übersetzergeschwindigkeit abzuwägen. (Extremfälle: JIT-Übersetzer, eingebettete Systeme)
- Eine Kombination / Integration der maschinenabhängigen Optimierungen ist bis heute nur teilweise gelungen.

  Kombination von Befehlsanordnung und Registerzuteilung: Proebsting & Fischer, *Optimal Code Scheduling for Delayed-Load Architectures*, ACM Transactions on Programming Language Design and Implementation, Toronto, Canada, June 1991, pp. 256-267.
- Unbehandeltes Problem: Berücksichtigung und Nutzung bedingter Befehle (predicated instructions) der IA-64