SSA-BASED REGISTER ALLOCATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

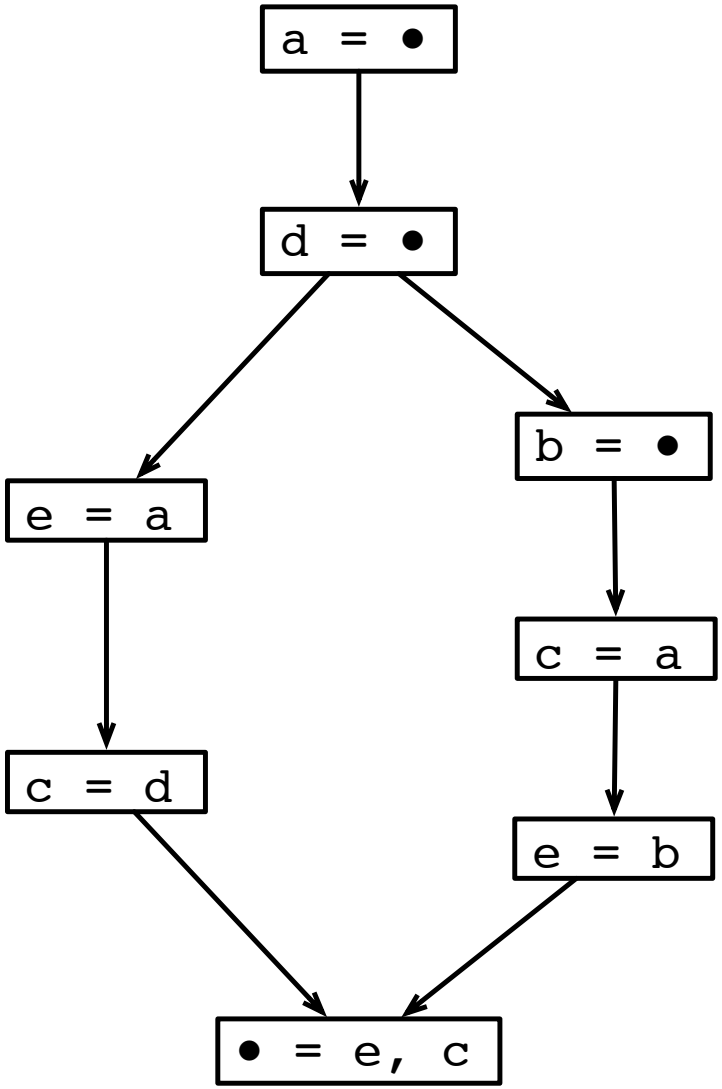
fernando@dcc.ufmg.br

A Start-up Example

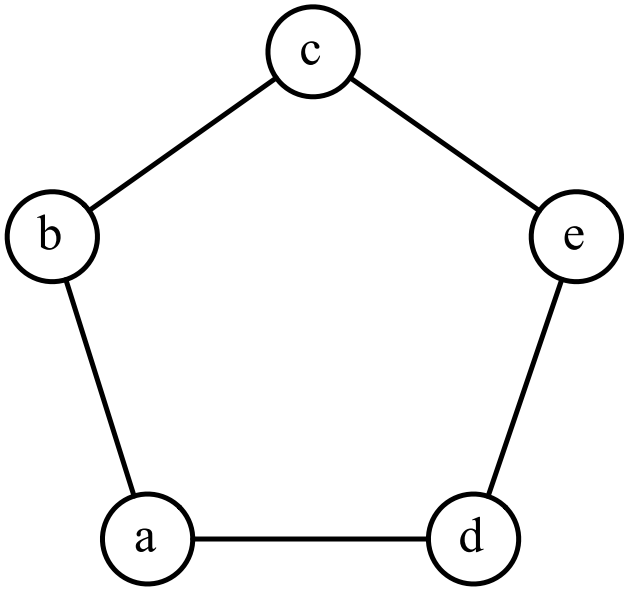
We shall use this example to motivate SSA-based register allocation. We have seen this program before, in our previous class on register allocation.

1) What is the maximum number of variables alive at any program point?

2) How is the interference graph of this program?

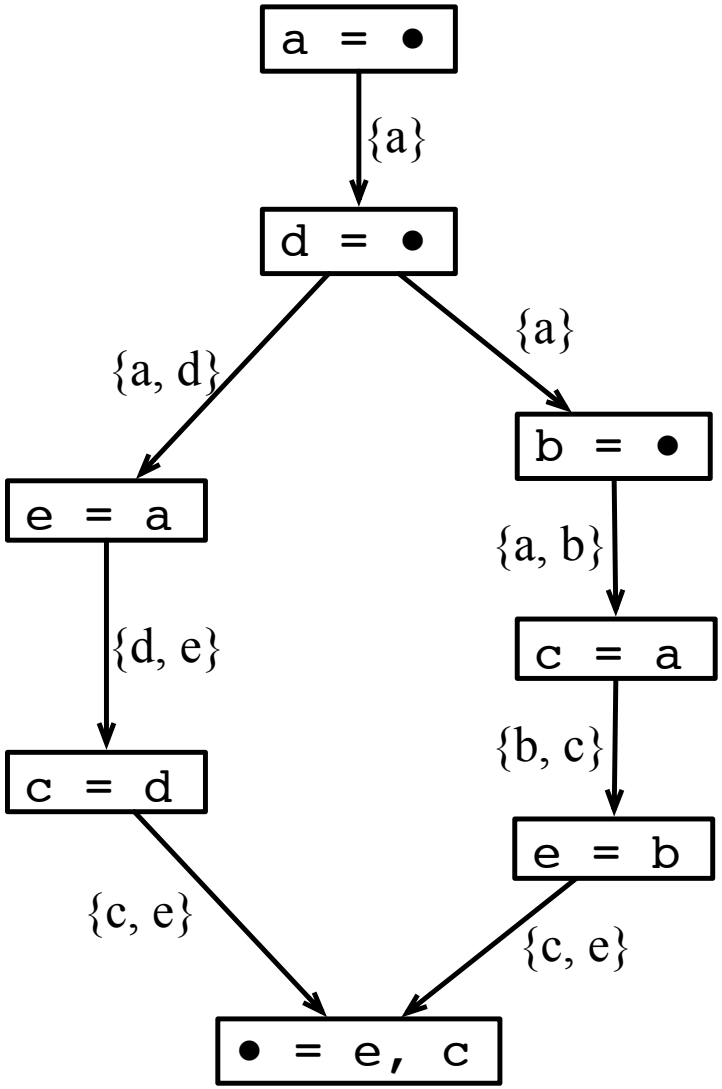


The Example's Interference Graph



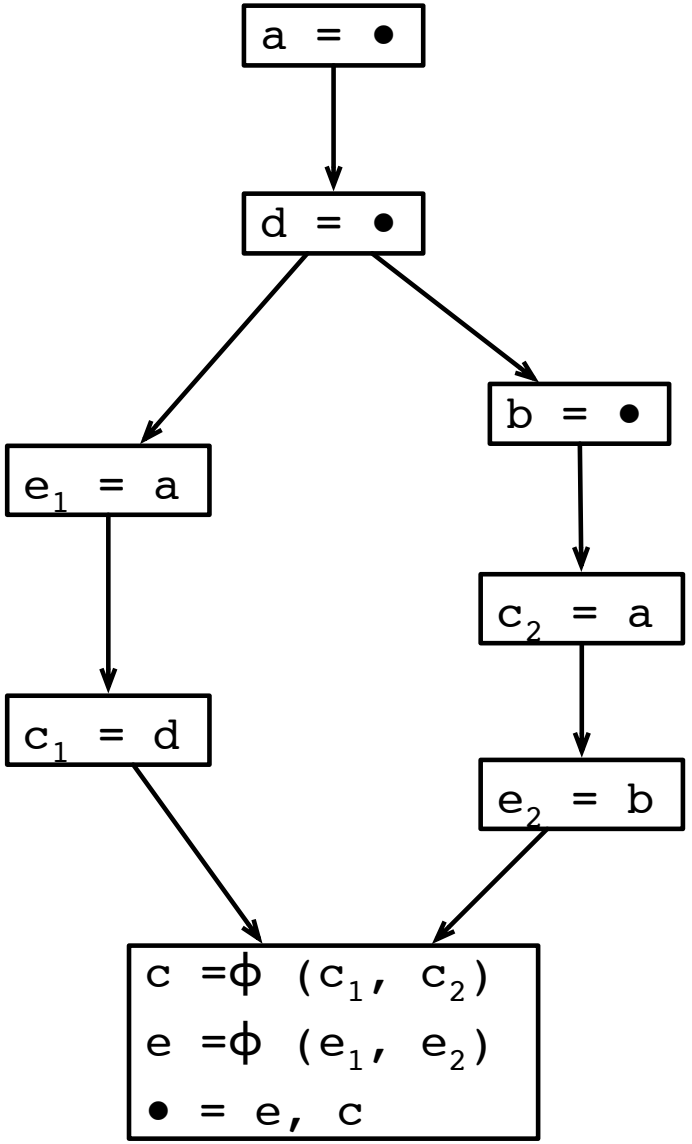
1) How many registers do we need, if we want to compile this program without spilling?

2) How this example would look like in SSA-form?



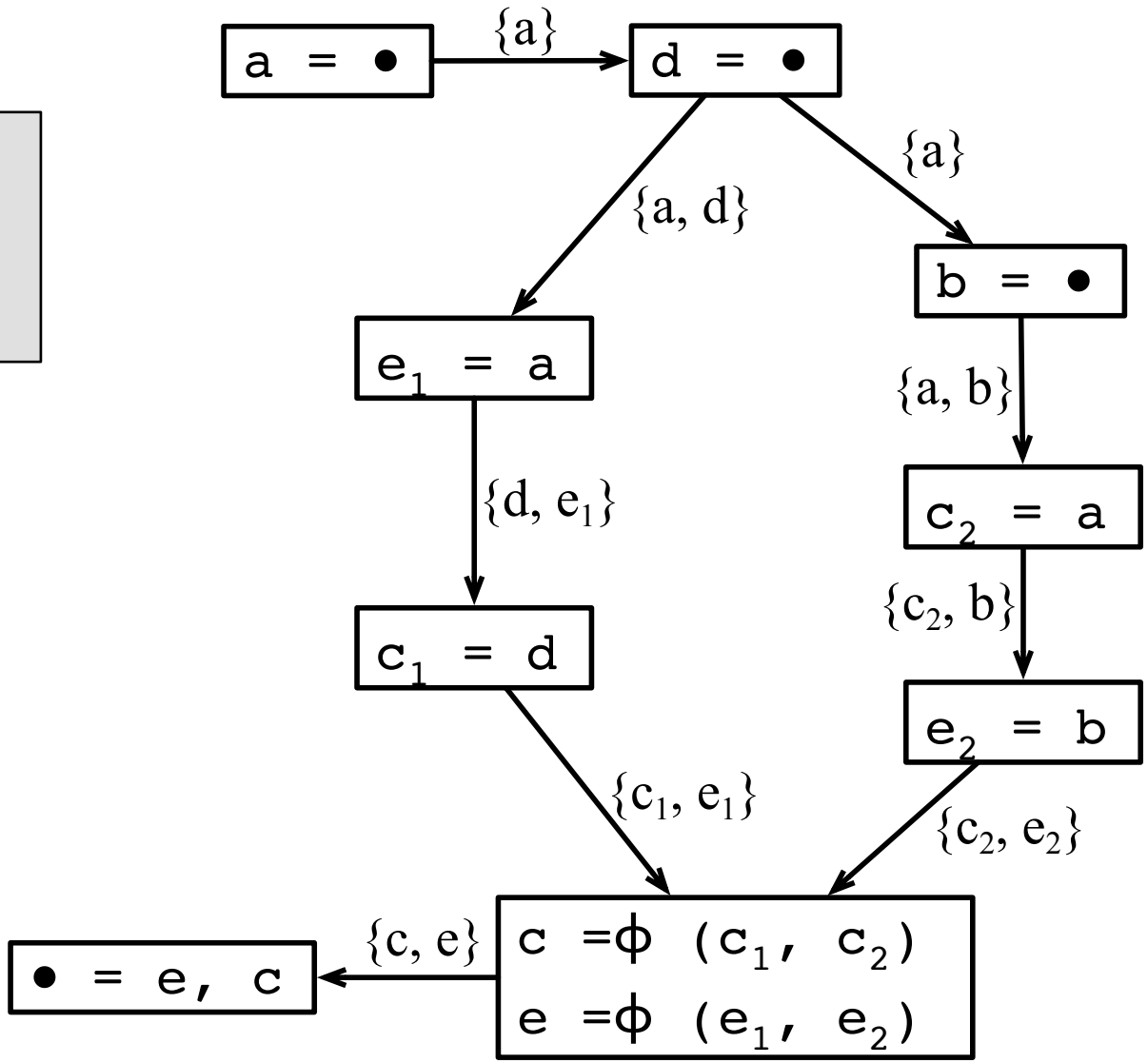
Example in SSA form

Can you run a liveness analysis algorithm on this program?



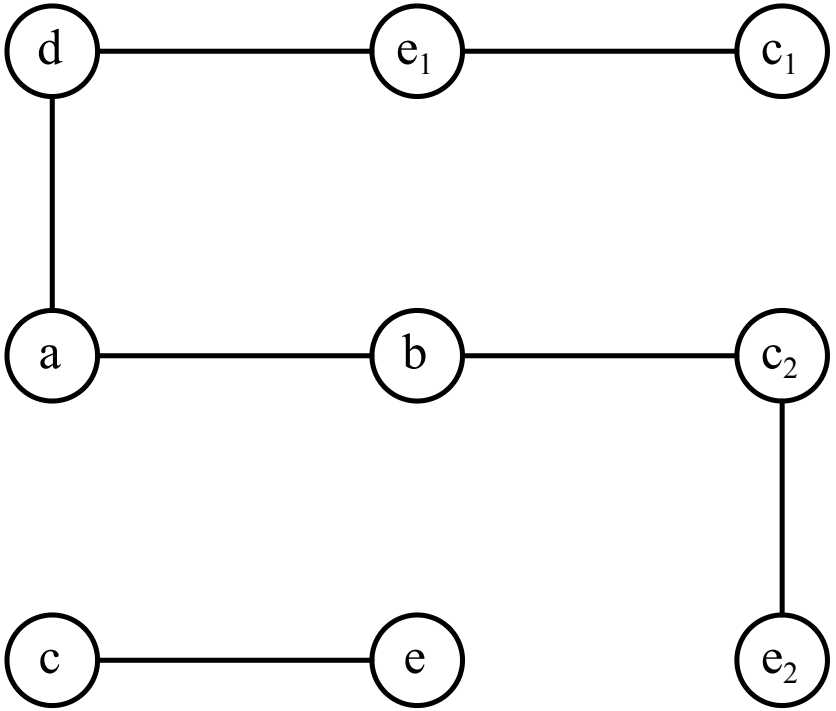
Example in SSA form

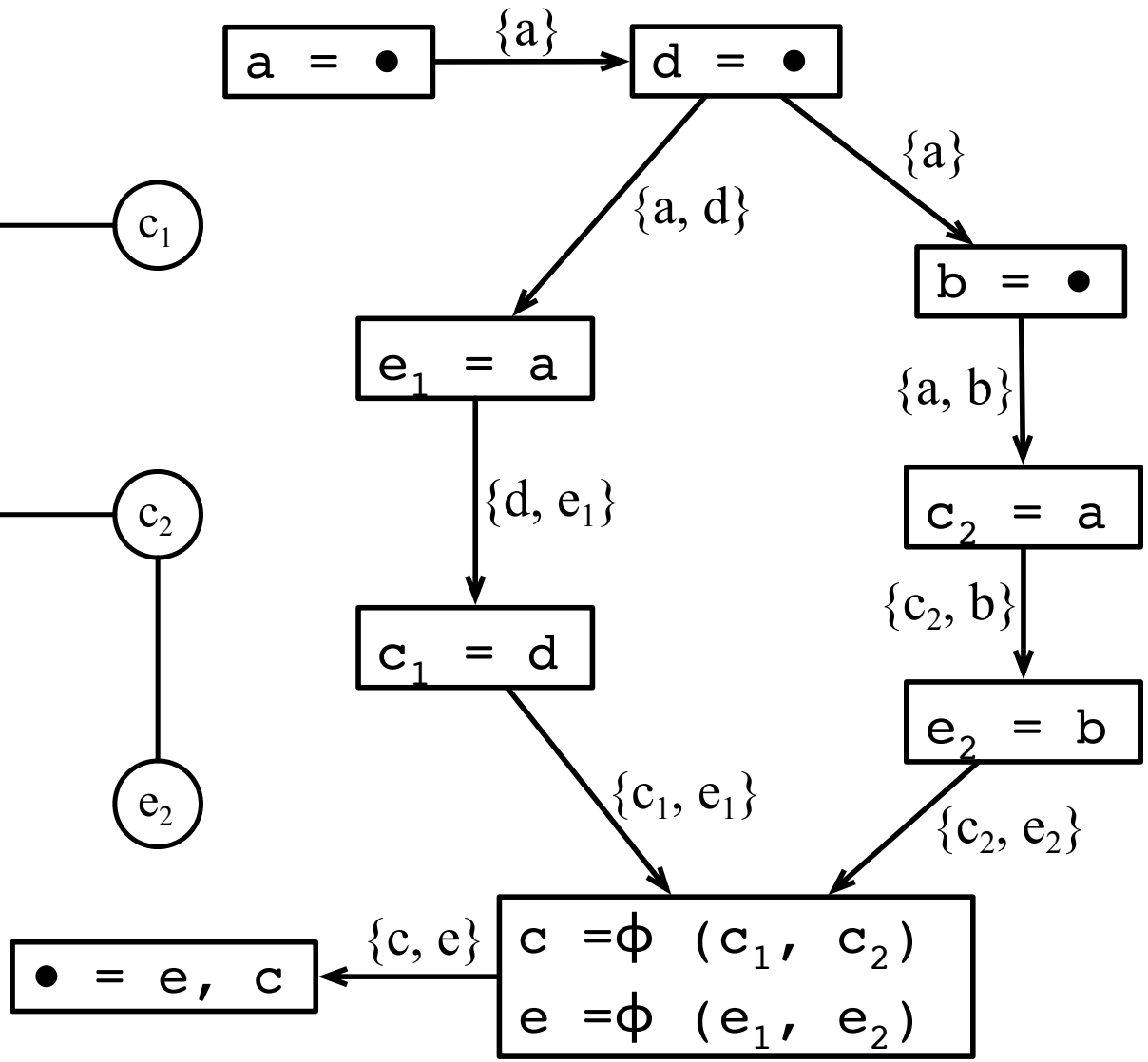
How is the interference graph of this example?



Example in SSA form

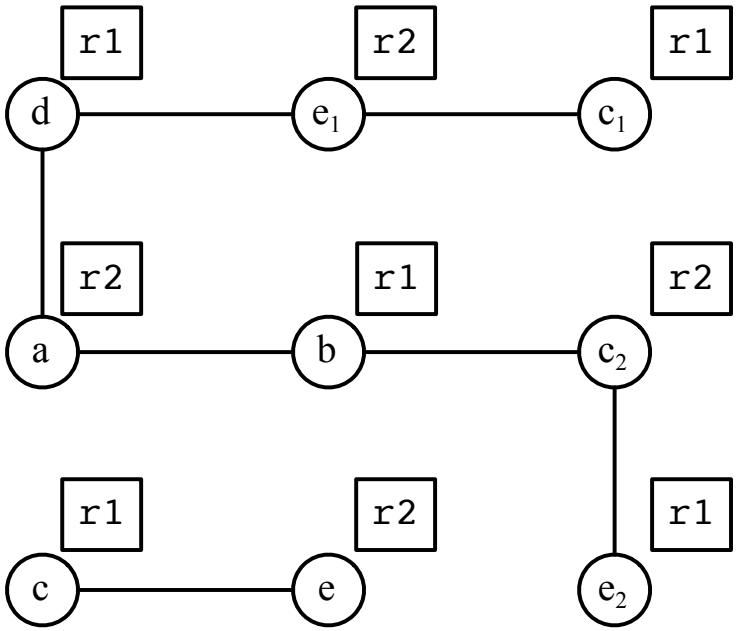
What's the chromatic number of this graph?

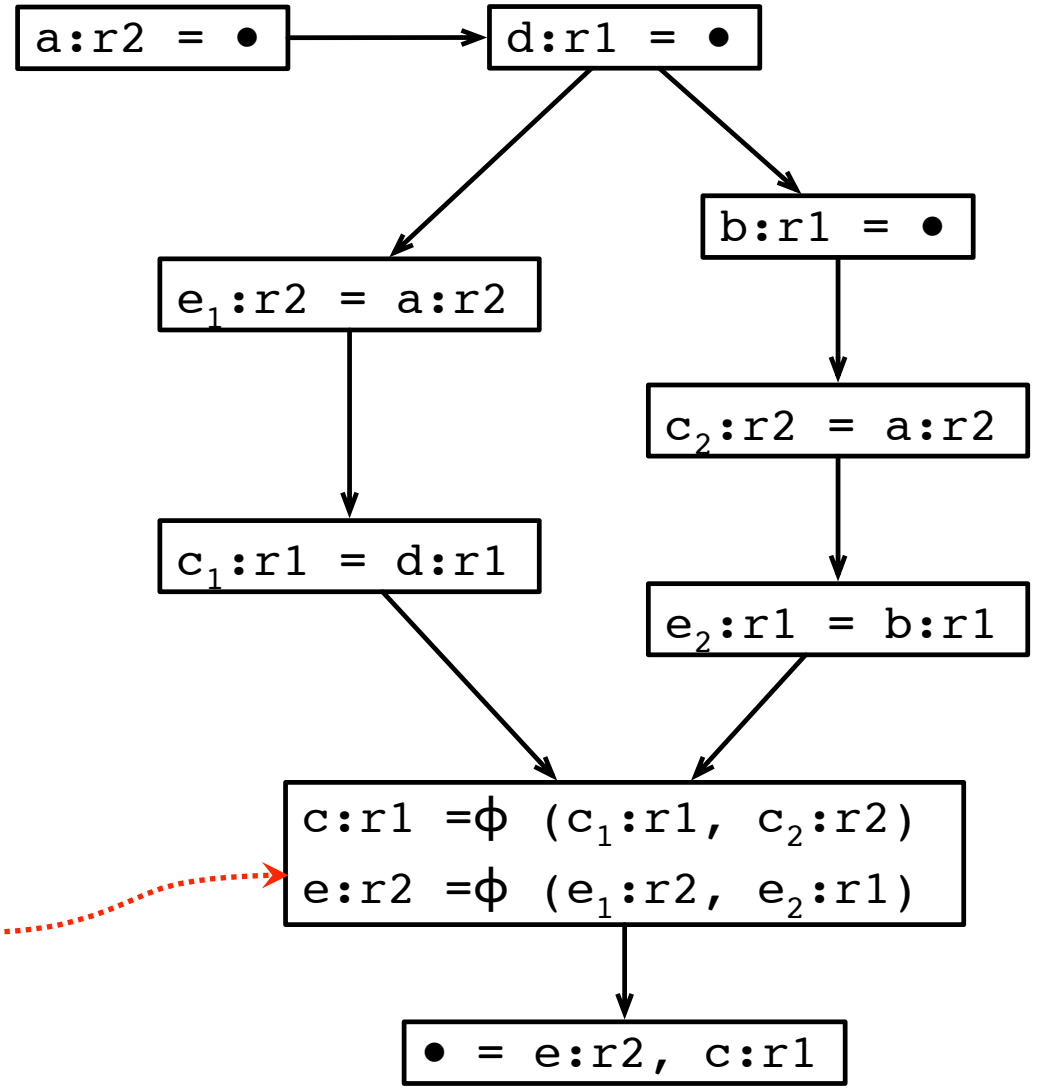




MinReg = MaxLive

This result is no coincidence.
We shall talk more about it!





But we still have a very serious problem: how can we translate **these phi-functions** to assembly, respecting the register allocation?

Swaps

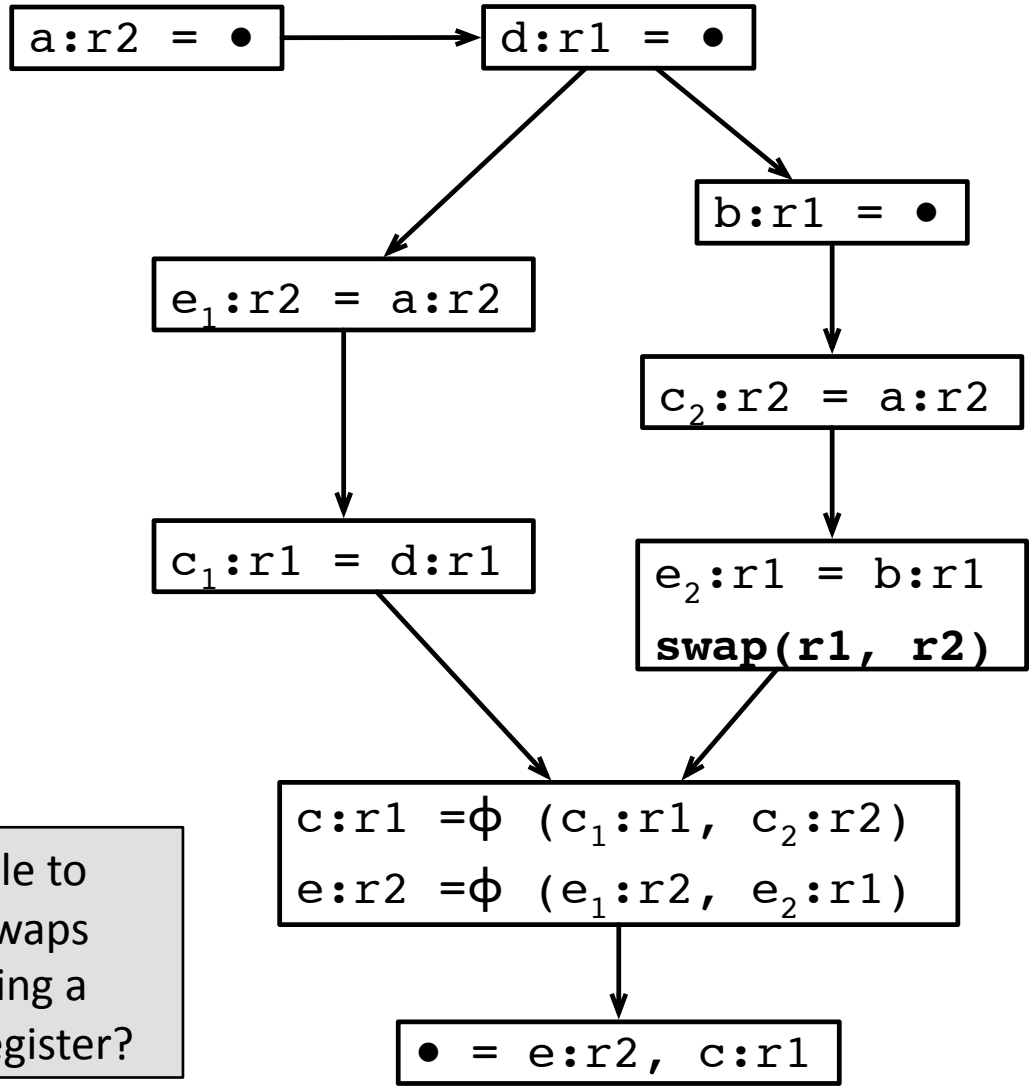
We need to copy the contents of e_2 to e. Similarly, we need to copy c_2 to c. But these variables have been allocated to different registers. If we have a third register to spare, we could do a swap like:

```
tmp = r1
r1 = r2
r2 = tmp
```

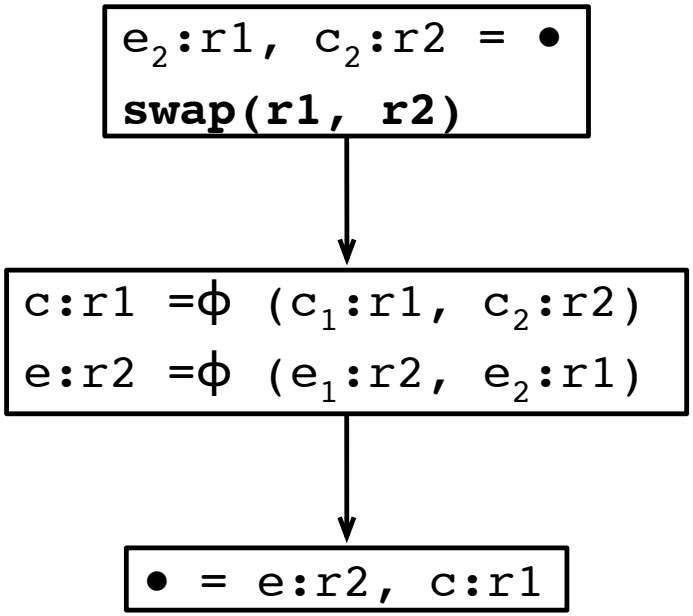
Yet, we may not have this register.

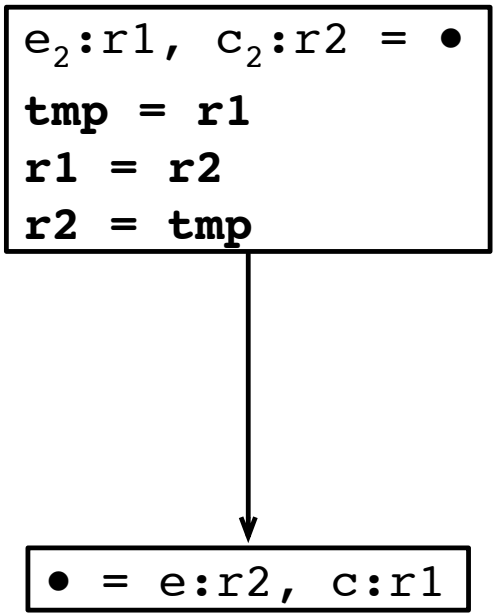
1) What is the problem of separating a register to do the swaps?

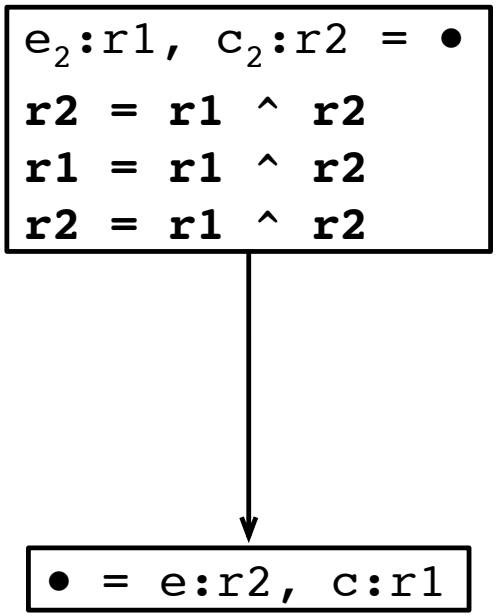
2) Is it possible to implement swaps without sparing a temporary register?



Swaps



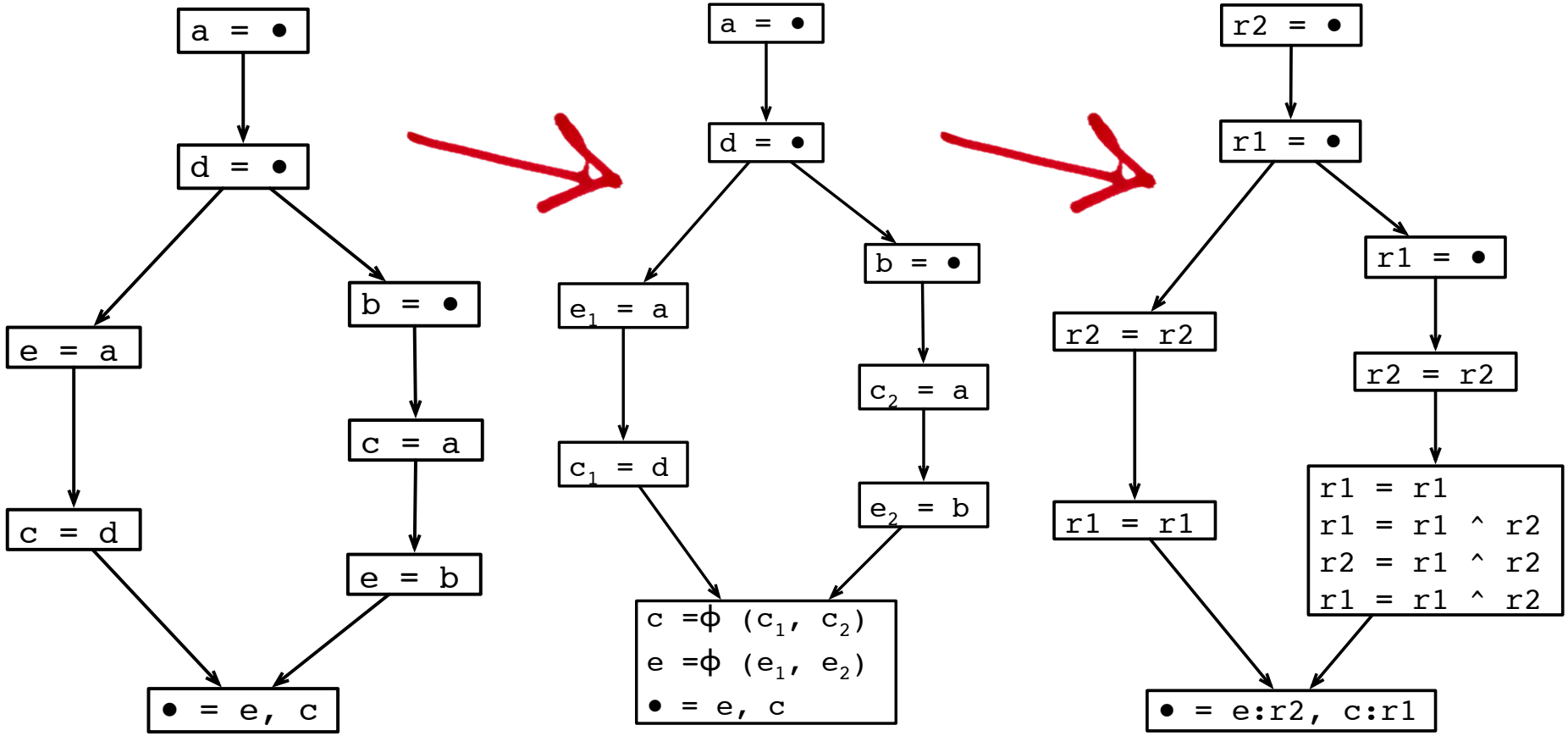




There are ways to implement swaps, without the need of a temporary location. One of these ways is the well-known hacking of using three xor operations to exchange two integer locations. There are other ways, though. Some architectures provide instructions to swap two registers. The x86, for instance, provides the instruction xchg(r1, r2), which exchanges the contents of r1 and r2.

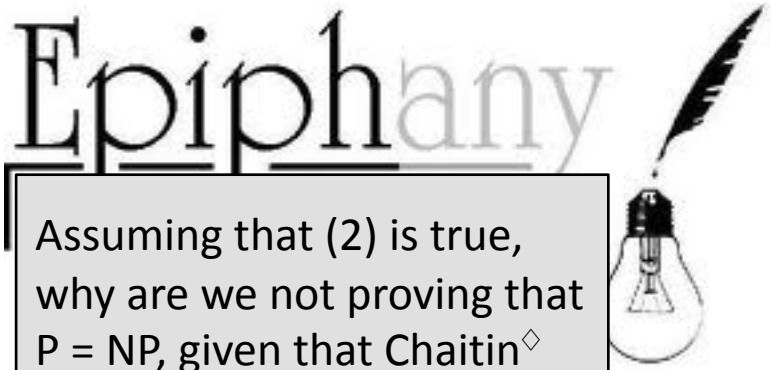
Can you think about other ways to swap the contents of registers?

So, in the end we get…



SSA-Based Register Allocation

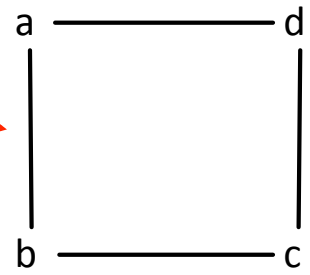
- We have been able to compile the SSA-form program with less registers than the minimum that the original program requires.
- This result is not a coincidence:
  1. The SSA-form program will never require more registers than the original program.
  2. And we can find the minimum number of registers that the SSA-form program needs in polynomial time.

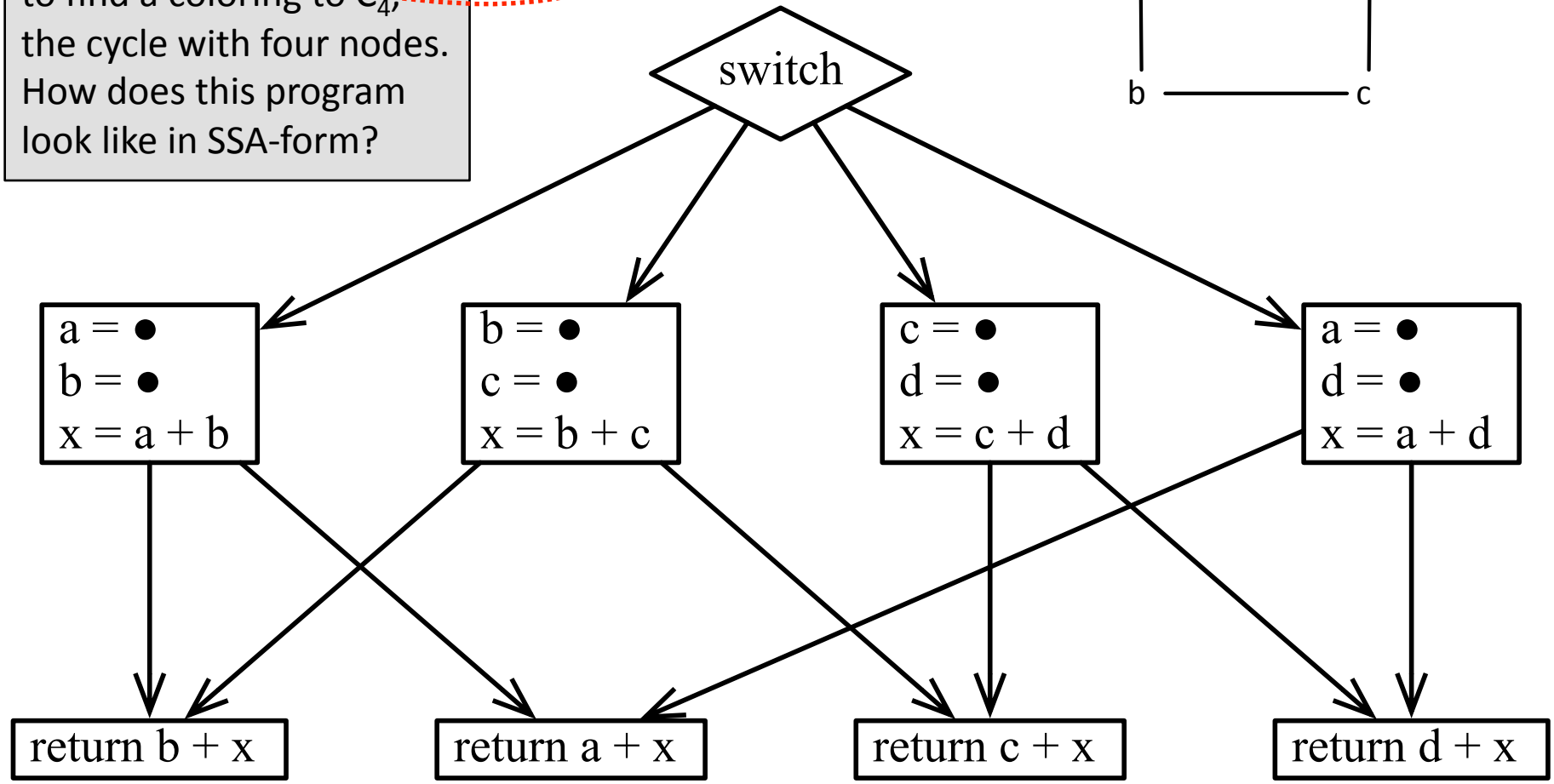


◇: Register allocation via coloring, 1981

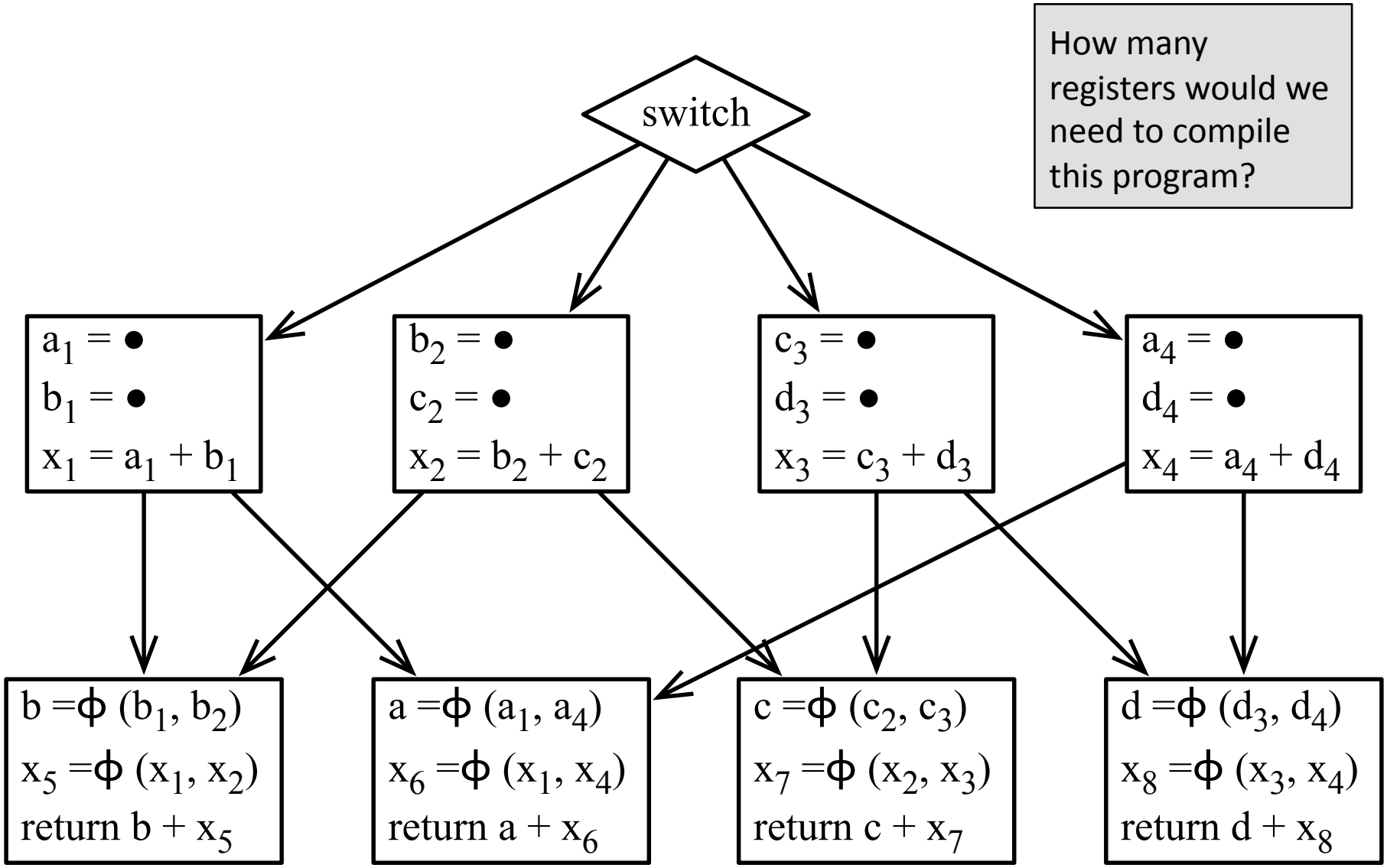
Chaitin's Proof in SSA-form Programs

This is the program that Chaitin would construct to find a coloring to C_4, the cycle with four nodes. How does this program look like in SSA-form?





Chaitin's Proof in SSA-form Programs



SSA-Based Register Allocation

- SSA-based register allocation is a technique to perform register allocation in SSA-form programs.
  - Simpler algorithm.
    - Decoupling of spilling and register assignment
  - Less spilling.
    - Smaller live ranges
    - Polynomial time minimum register assignment

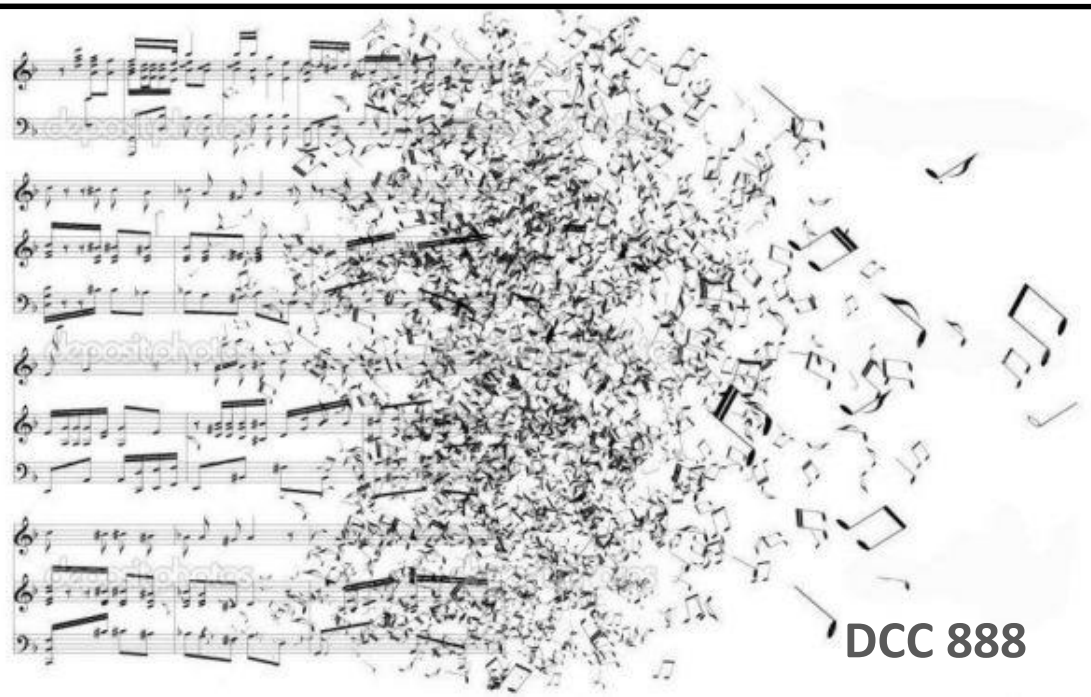
Traditional Register Allocation

SSA-Based Register Allocation

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

SSA-Form and Chordal Graphs

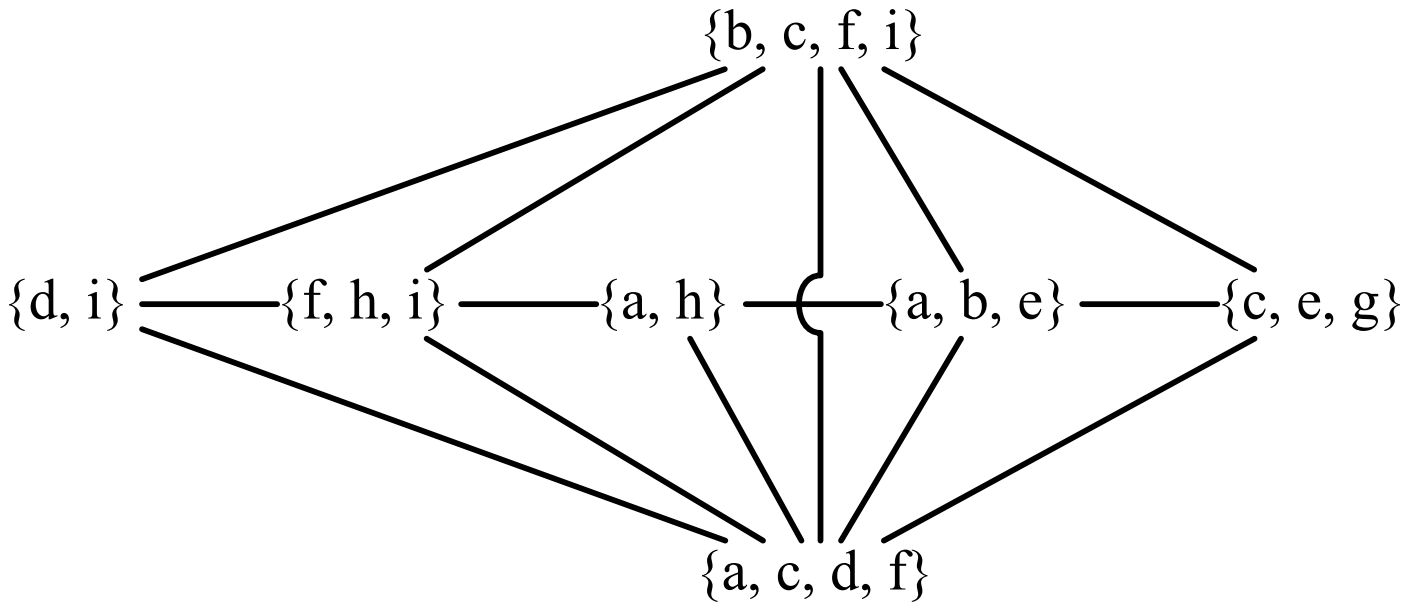


DCC 888

Intersection Graphs

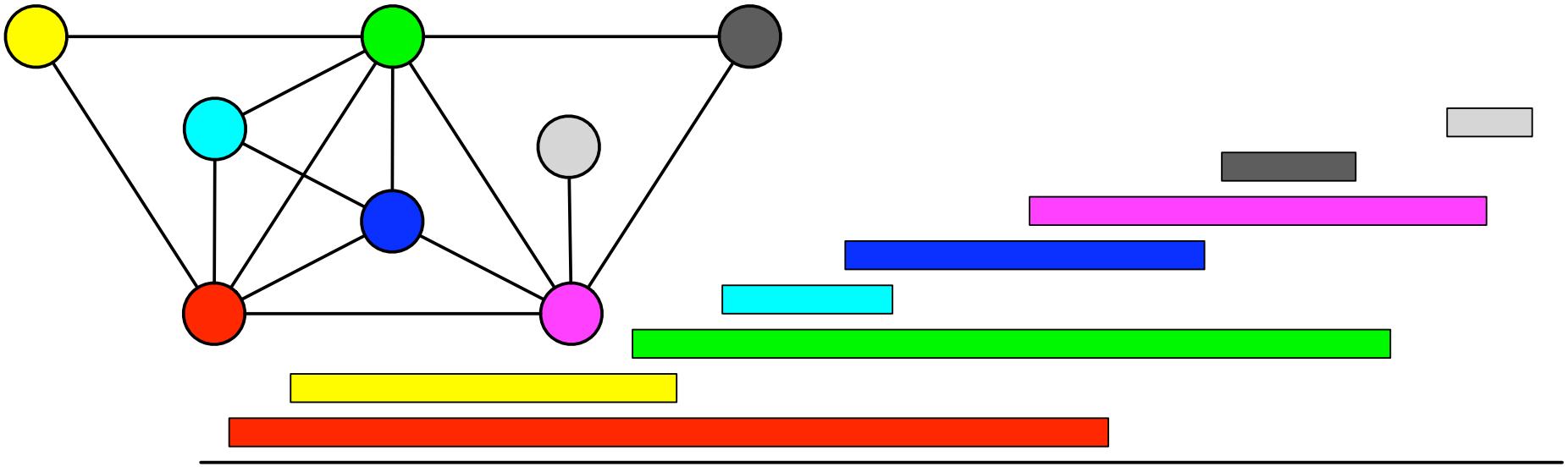
- If S is a set of sets, then we define an intersection graph G = (V, E) as follows:
  - For each set $s \in S$, we have a vertex $v \in V$
  - If $s_0, s_1 \in S$, and $s_0 \cap s_1 \neq \{\}$, then we have an edge $(v_0, v_1) \in E$

```
{
  {a, b, e},
  {a, c, d, f},
  {b, c, f, i},
  {a, h},
  {f, h, i},
  {c, e, g},
  {d, i}
}
```



Interval Graphs

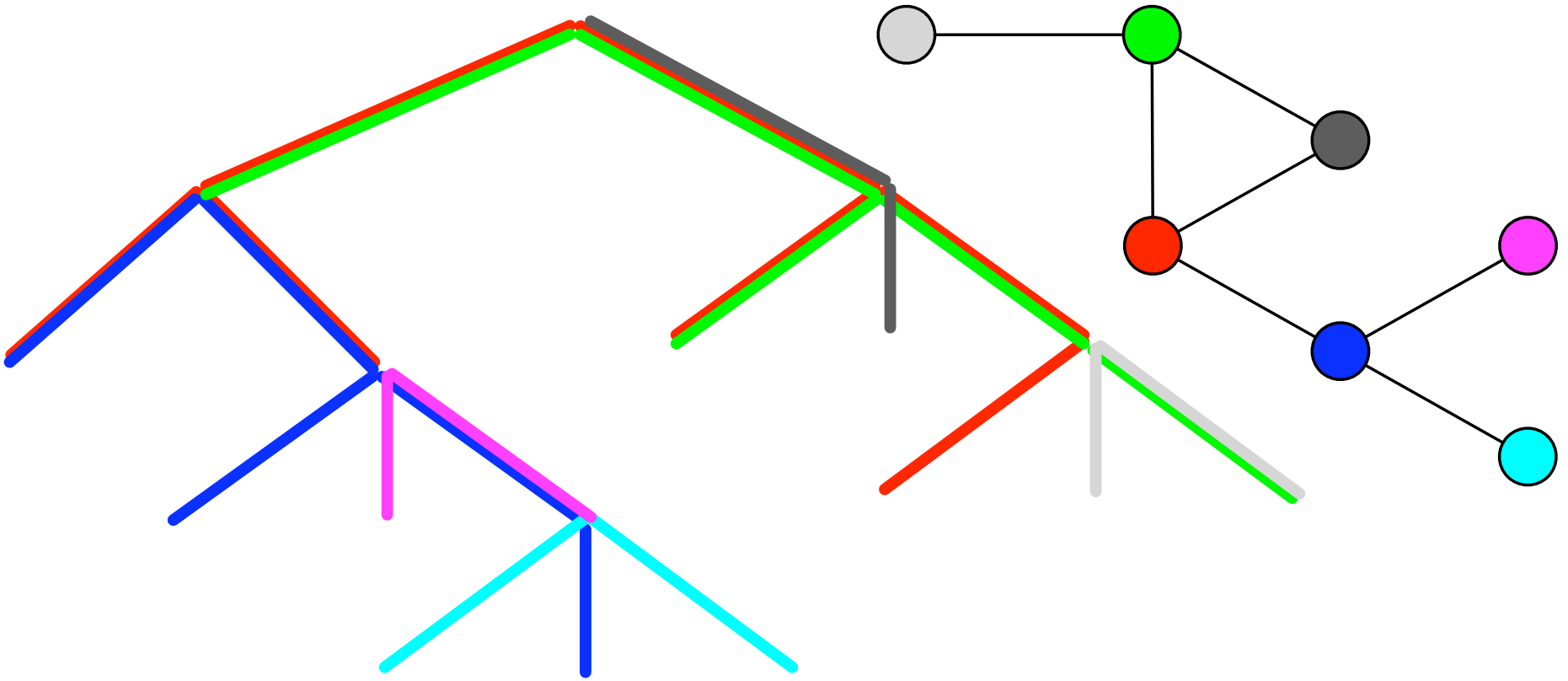
- We have already seen interval graphs, in the context of register allocation:
  - We have the intersection graph of segments on a line:
    - We have a node for each segment
    - We have an edge between two nodes whose corresponding segments overlap on the line



Chordal Graphs

- The intersection graph of subtrees of a tree is a *chordal graph*.

The interference graph of programs in SSA form is chordal. Any intuition on why?

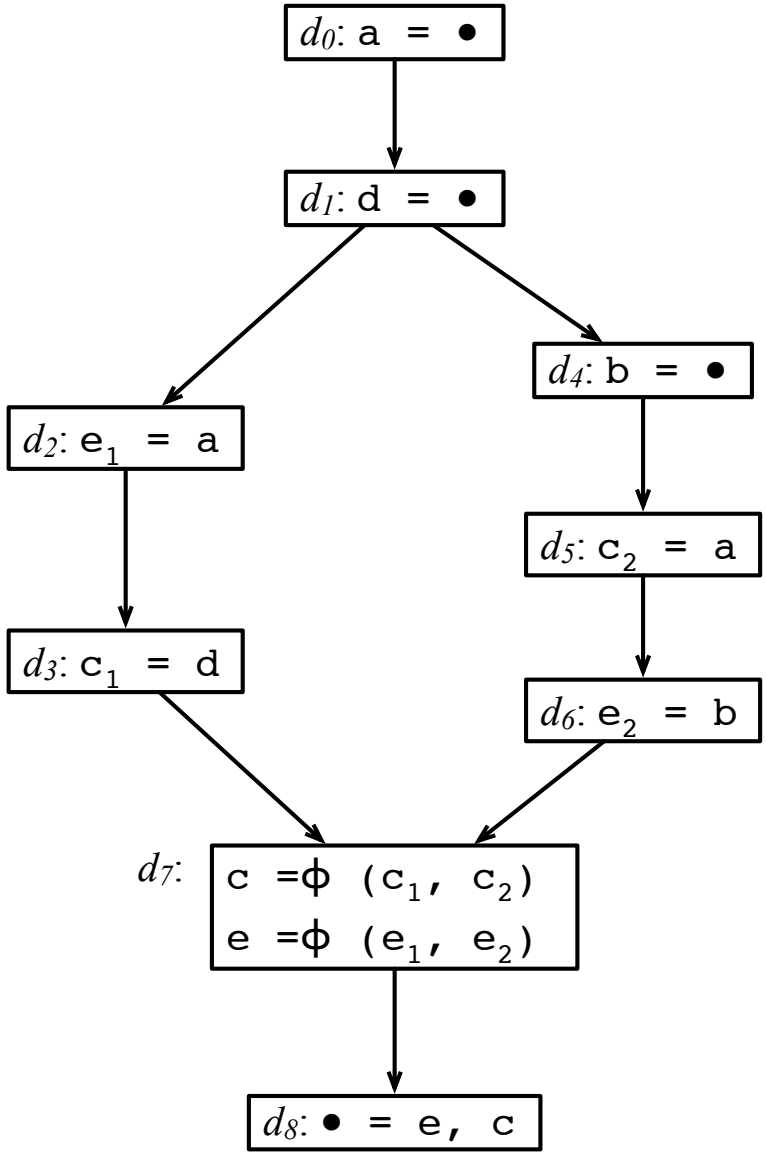


Dominance Trees

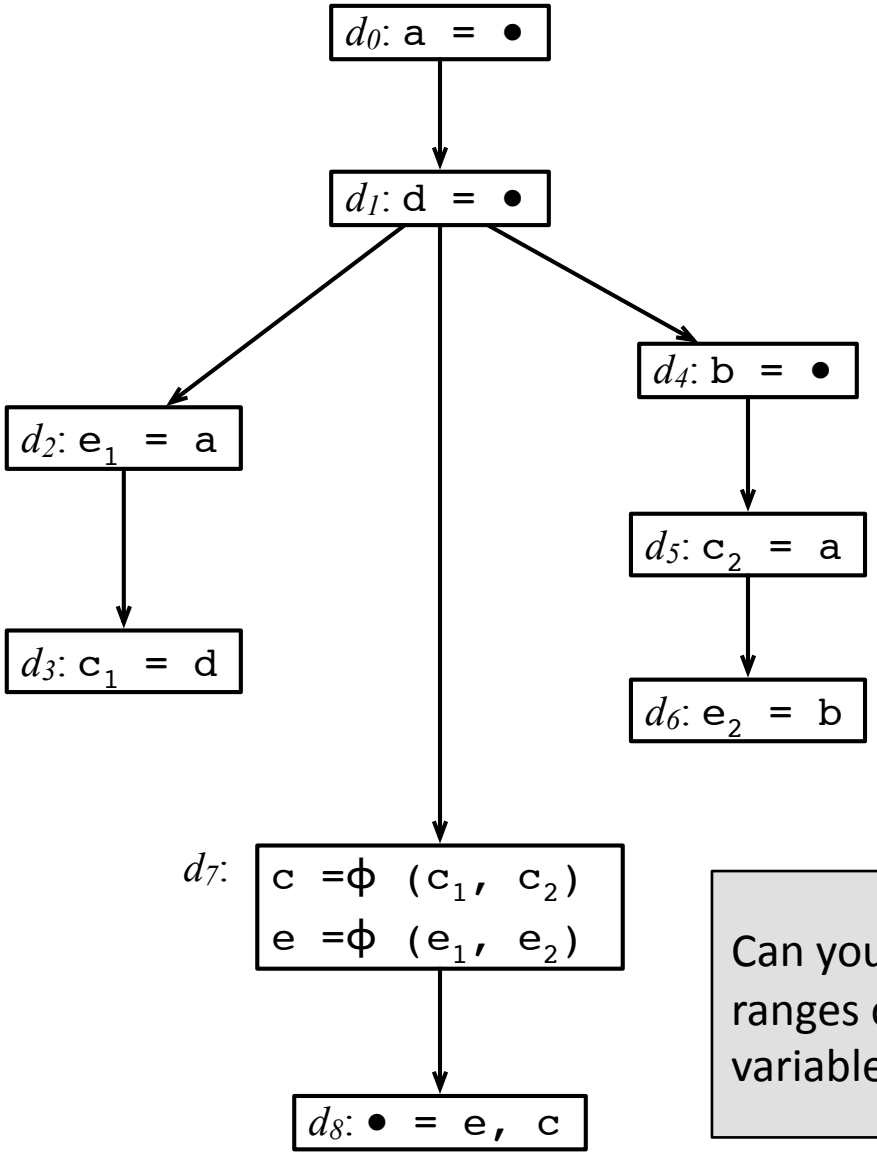
1) Do you remember what is dominance tree?

2) When had we talked about this data-structure before?

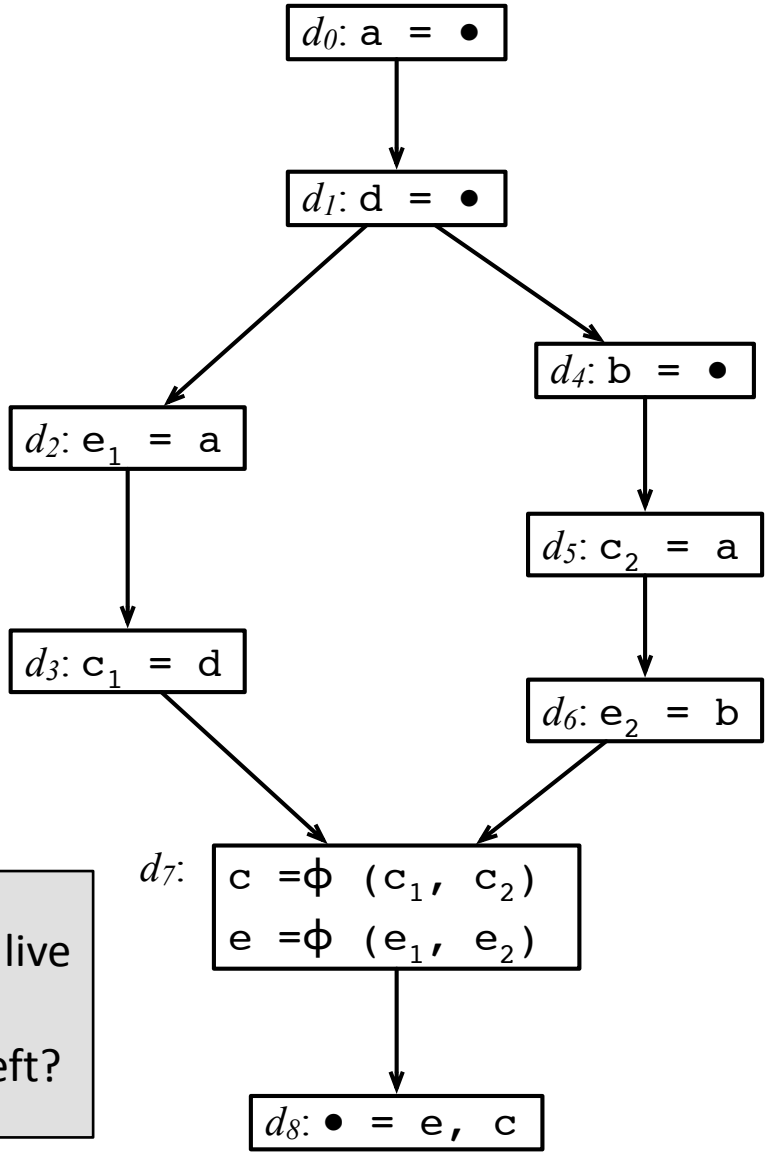
3) How is the dominance tree of this program on the right?



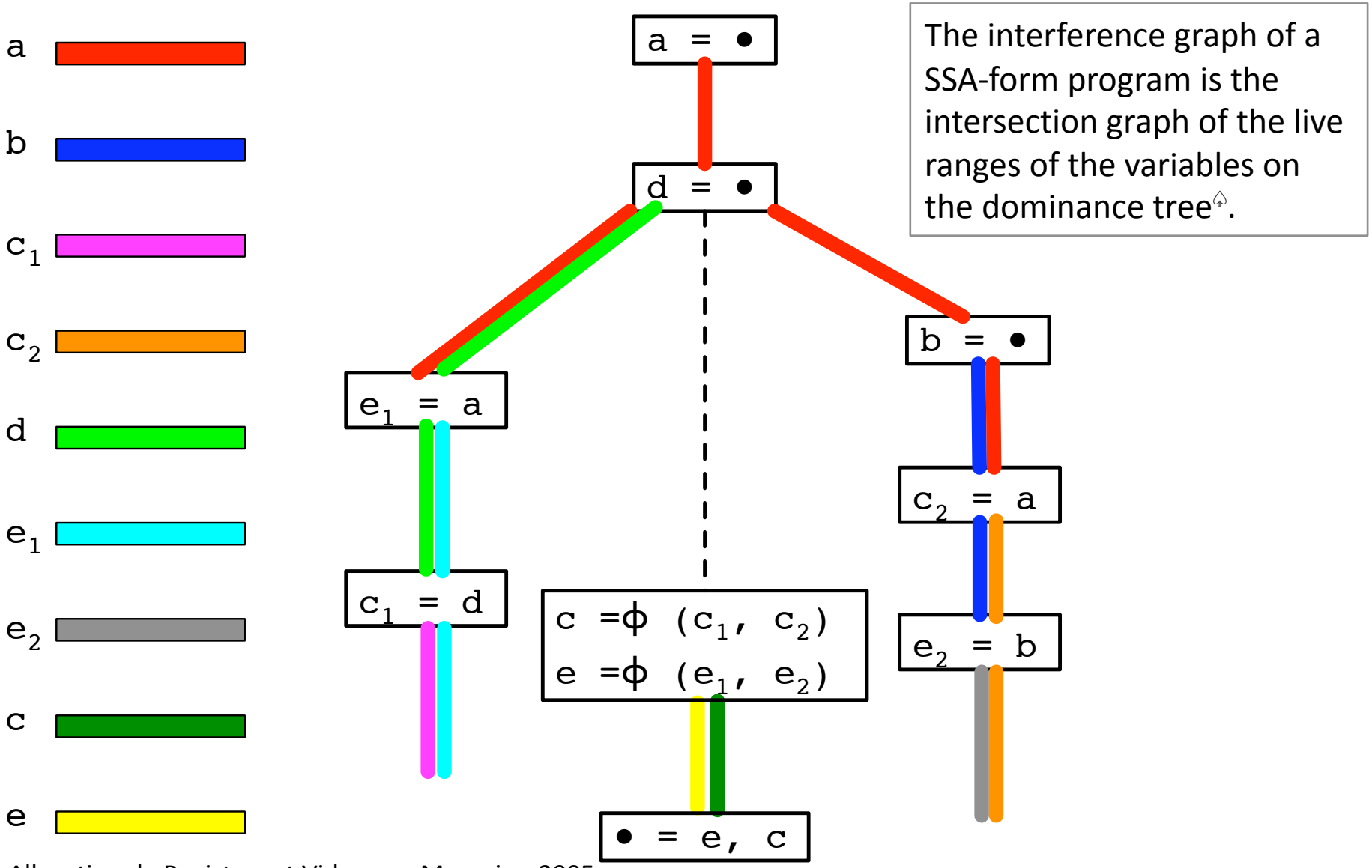
Dominance Trees



Can you draw the live ranges of the variables on the left?



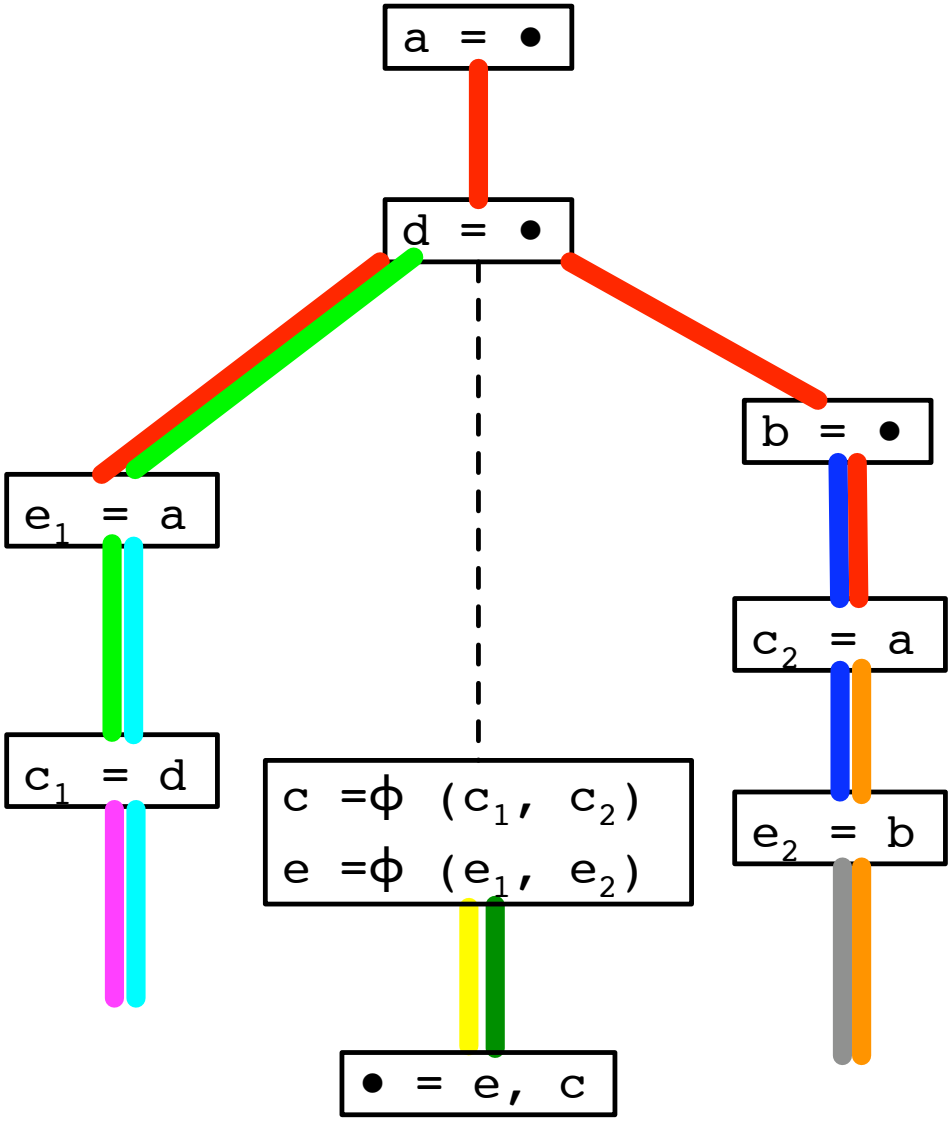
Dominance Trees

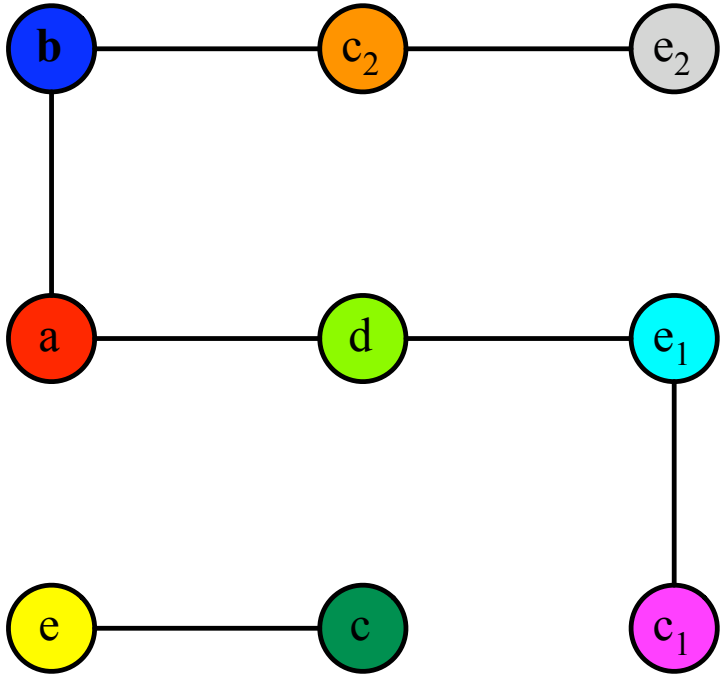


The interference graph of a SSA-form program is the intersection graph of the live ranges of the variables on the dominance tree♤.

♤: Allocation de Registres et Vidage en Memoire, 2005

Intersection Graph of Live Ranges





Triangular Graph = Chordal Graph

- There are other ways to define chordal graphs. We will need this definition below in our proof that SSA form programs have chordal interference graphs:

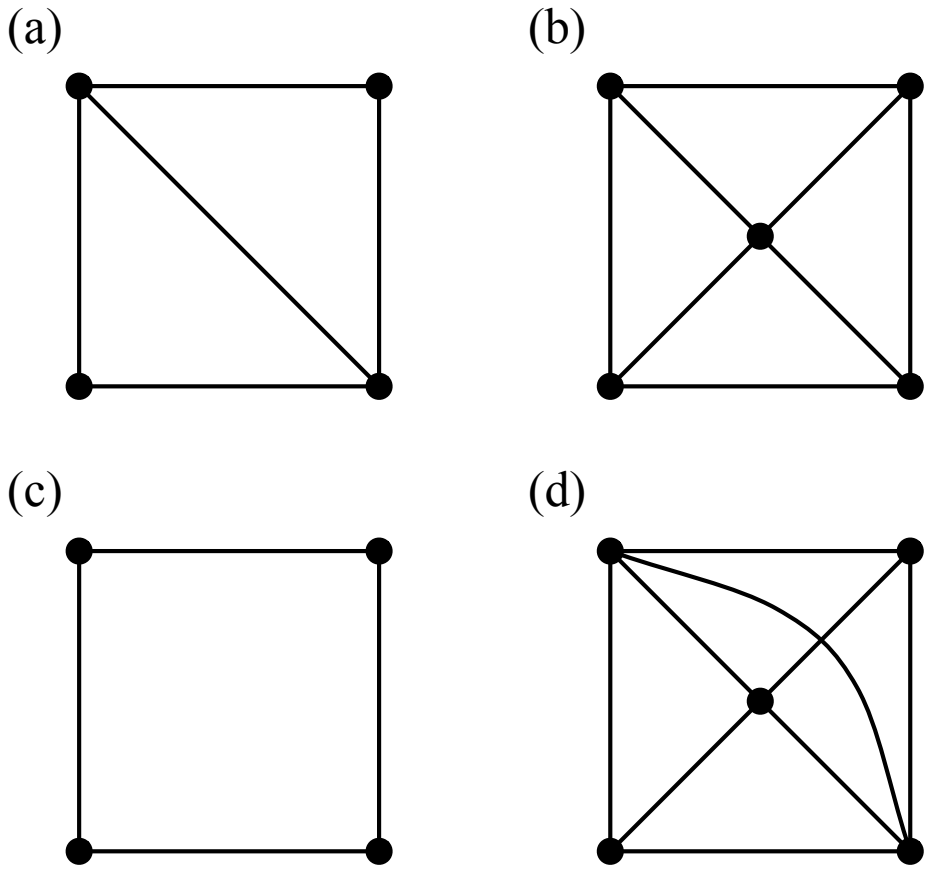
A Graph is Chordal If, and Only If, it has No Induced Subgraph Isomorphic to Cn, Where Cn is the Cycle with N Nodes, N > 3.

What is an induced subgraph?

Triangular Graph = Chordal Graph

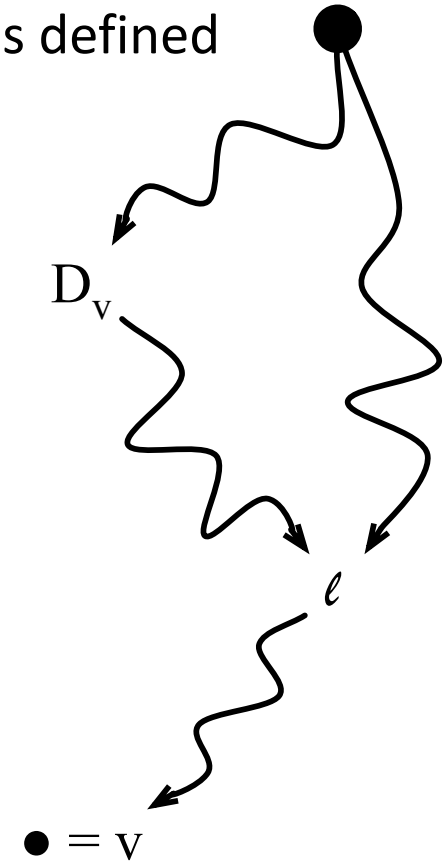
- If G = (V, E) is a graph, then S = (V', E') is an induced subgraph of G if $V' \subseteq V$, and $(v_i, v_j) \in E'$ if, and only if, $(v_i, v_j) \in E$.

Which graphs on the right are chordal?



Dominance and Interference

- Label ℓ **dominates** label ℓ' if every path from the beginning of the CFG to ℓ' must go through ℓ. We write that $\ell < \ell'$.
- If v is a variable, we denote by D_v the label where v is defined in the program.
- We say that a program is *strict* if a variable can only be used if it is defined before.



Theorem 1: in a strict SSA-form program, D_v dominates every label where v is alive[◇].

Can you prove this theorem? You must rely on the definition of a strict program, SSA, liveness and dominance. There is a hint on the right.

◇: Fast copy coalescing and live-range identification, PLDI (2002)

Dominance and Interference

- Label ℓ **dominates** label ℓ' if every path from the beginning of the CFG to ℓ' must go through ℓ. We write that $\ell < \ell'$.
- If v is a variable, we denote by D_v the label where v is defined in the program.
- We say that a program is *strict* if a variable can only be used if it is defined before.

Theorem 1: in a strict SSA-form program, D_v dominates every label where v is alive[◇].

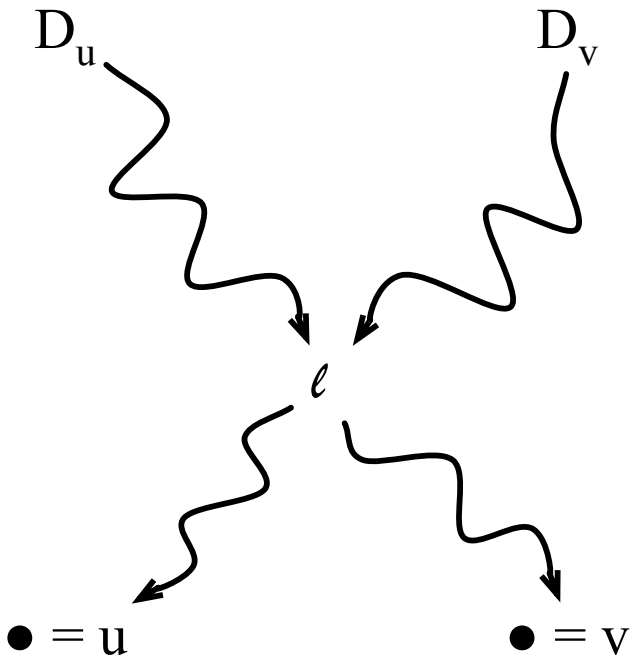
Proof: suppose not. Then there exists a label ℓ in the CFG where v is alive, but that is not dominated by v. Thus, there exists a path from ℓ to a usage of v. Therefore, there exists a path from the beginning of the program to a usage of v that does not go across the unique definition of v, and the program is not strict.

◇: Fast copy coalescing and live-range identification, PLDI (2002)

Dominance and Interference

Lemma 1: if two variables, *u* and *v*, interfere, in a strict SSA-form program, then either $D_u < D_v$, or $D_v < D_u$.

Can you prove this lemma?
There is a hint on the Right.
See if it helps.



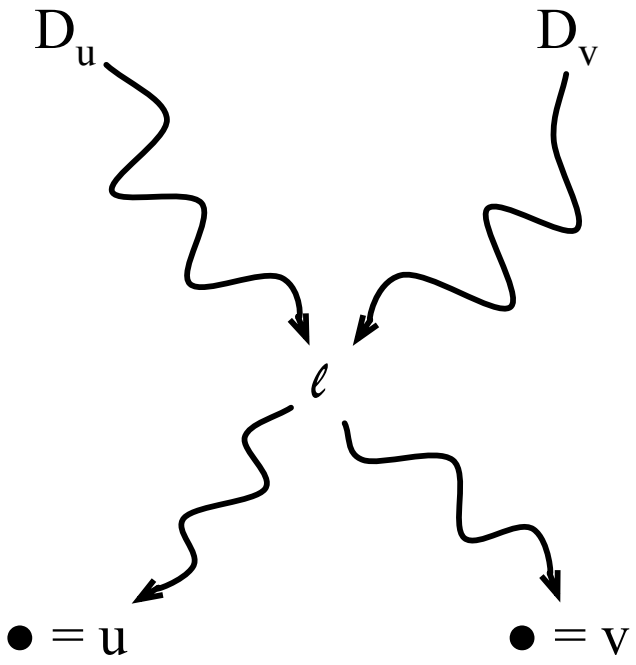
Dominance and Interference

Lemma 1: if two variables, *u* and *v*, interfere, in a strict SSA-form program, then either $D_u < D_v$, or $D_v < D_u$.

Proof: If *u* and *v* interfere, then there exists a label ℓ in the CFG where both variables are alive. By theorem 1, this label is dominated by D_u and D_v. Let's assume that neither $D_v < D_u$ nor $D_u < D_v$.
Then:

1. There exists a path from start to ℓ going through D_u that does not go across D_v (or else there must exist a path from start to ℓ going through D_v that does not go across D_u).
2. There exists also a path from ℓ to a usage of *v*, because *v* is alive at ℓ.

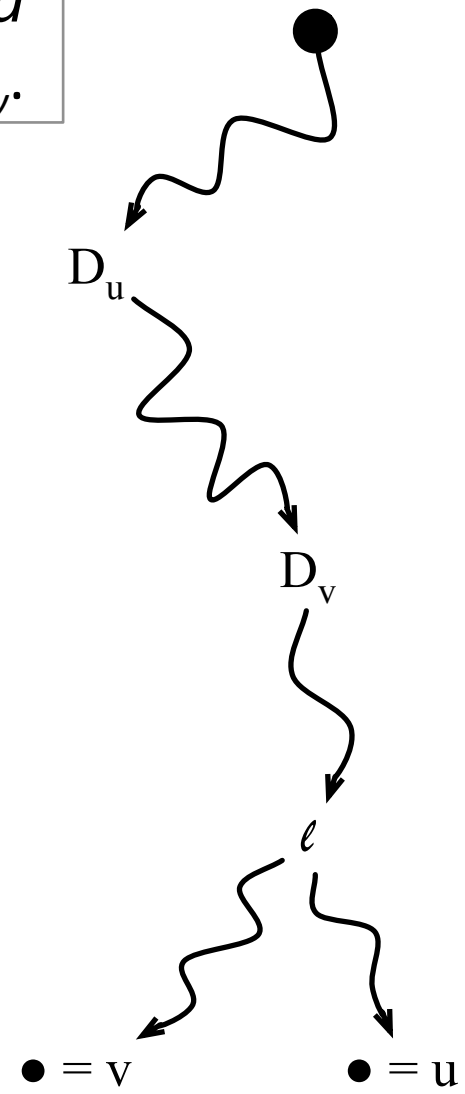
From (1) and (2) we conclude that there exists a path from start to a usage of *v* that does not go across a definition of *v*. Therefore, the program is not strict.



Dominance and Interference

Lemma 2: in a Strict program, if two variables, *u* and *v*, interfere and $D_u < D_v$, then *u* is alive at D_v.

Can you prove this lemma?
There is a hint on the right.
Try showing that every path in this graph is necessary.

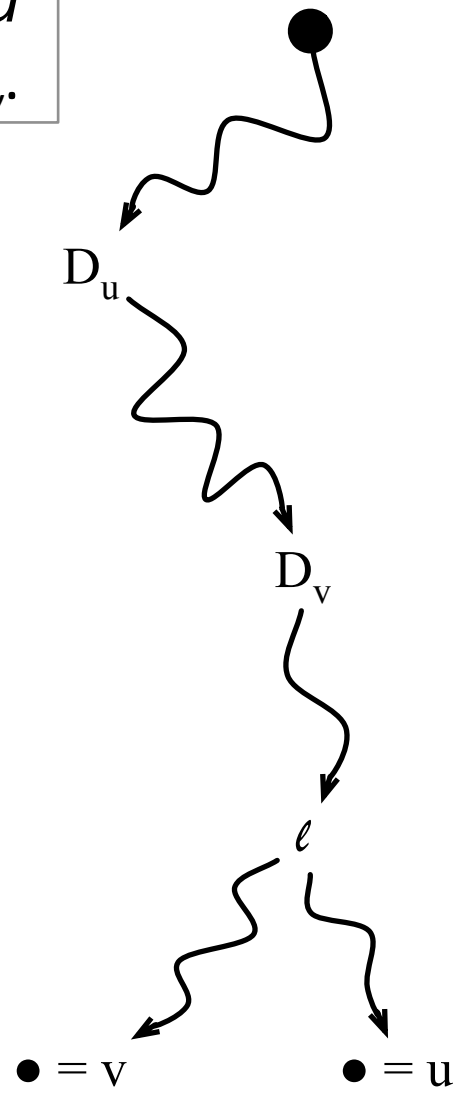


Dominance and Interference

Lemma 2: in a Strict program, if two variables, *u* and *v*, interfere and $D_u < D_v$, then *u* is alive at D_v.

Proof:

- There exists a label ℓ in the CFG where *v* and *u* are alive.
- There exists a path from D_u to D_v, given the definition of dominance.
- There exists a path from D_v to ℓ, given the definition of liveness, plus Theorem 1.
- There exists a path from ℓ to a usage of *u*, due to the definition of liveness, plus Theorem 1.



Transitivity of Dominances

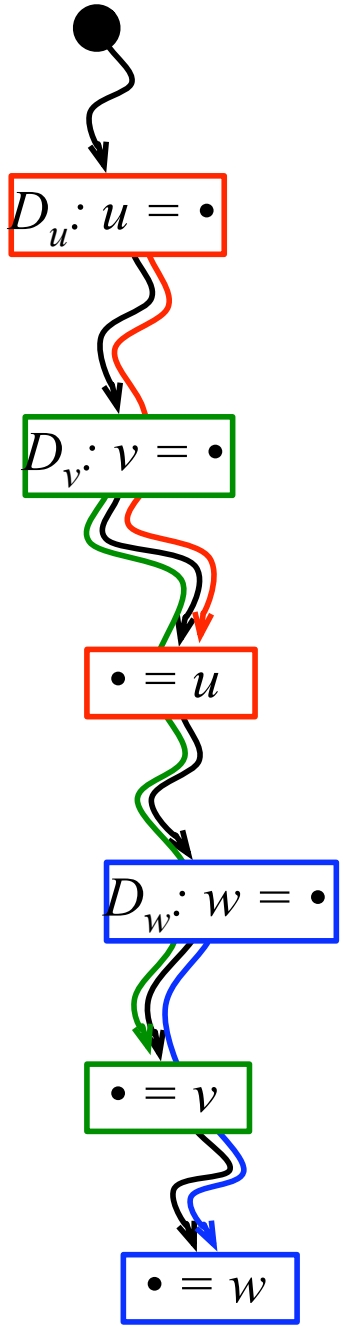
Lemma 3: let *u*, *v* and *w* be three program variables, where (*u, v*) and (*v, w*) interfere, but (*u*, *w*) do not, if $D_u < D_v$, then $D_v < D_w$

Proving this lemma is not too difficult, if you **remember** the previous lemmas. Can you do it?

In case you do not remember them, and that is *mildly* possible, **here** are the lemmas for you…

Lemma 1: if two variables, *u* and *v*, interfere, in a strict SSA-form program, then either $D_u < D_v$, or $D_v < D_u$.

Lemma 2: in a strict program, if two variables, *u* and *v*, interfere, $D_u < D_v$, then *u* is alive at D_v.



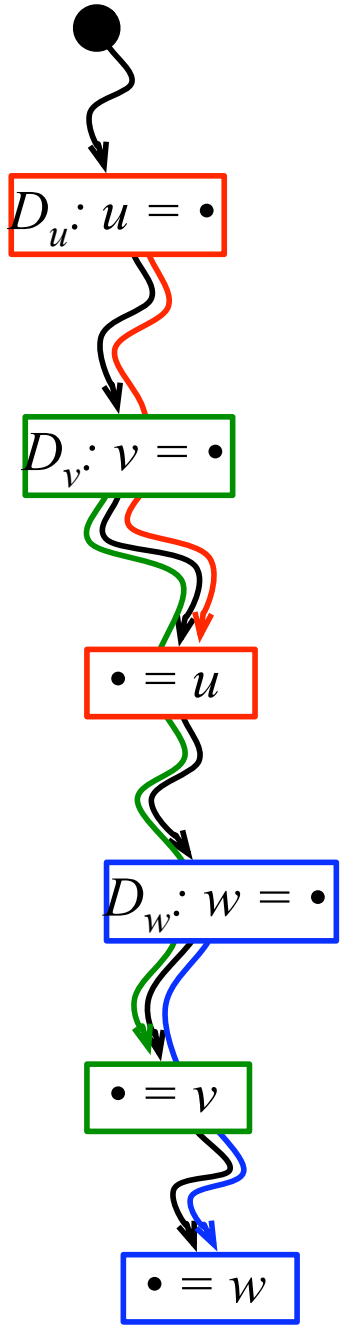
Transitivity of Dominances

Lemma 3: let *u*, *v* and *w* be three program variables, where (*u, v*) and (*v, w*) interfere, but (*u*, *w*) do not, if $D_u < D_v$, then $D_v < D_w$

Proof:

- We know that either $D_v < D_w$, or $D_w < D_v$. This is true because *v* and *w* are simultaneously alive, and we have the result of **Lemma 1**
- If we assume $D_w < D_v$, then, by **Lemma 2**, we have that *w* is alive at D_v
- Because *u* and *v* also interfere, and $D_u < D_v$, we know, also from **Lemma 2**, that *u* is also live at D_v
- From this absurd (by hypothesis u and w do not interfere), we know that $D_v < D_w$

Lemma 1: if two variables, *u* and *v*, interfere, in a strict SSA-form program, then either $D_u < D_v$, or $D_v < D_u$.
Lemma 2: in a strict program, if two variables, *u* and *v*, interfere, $D_u < D_v$, then *u* is alive at D_v.



Chordality

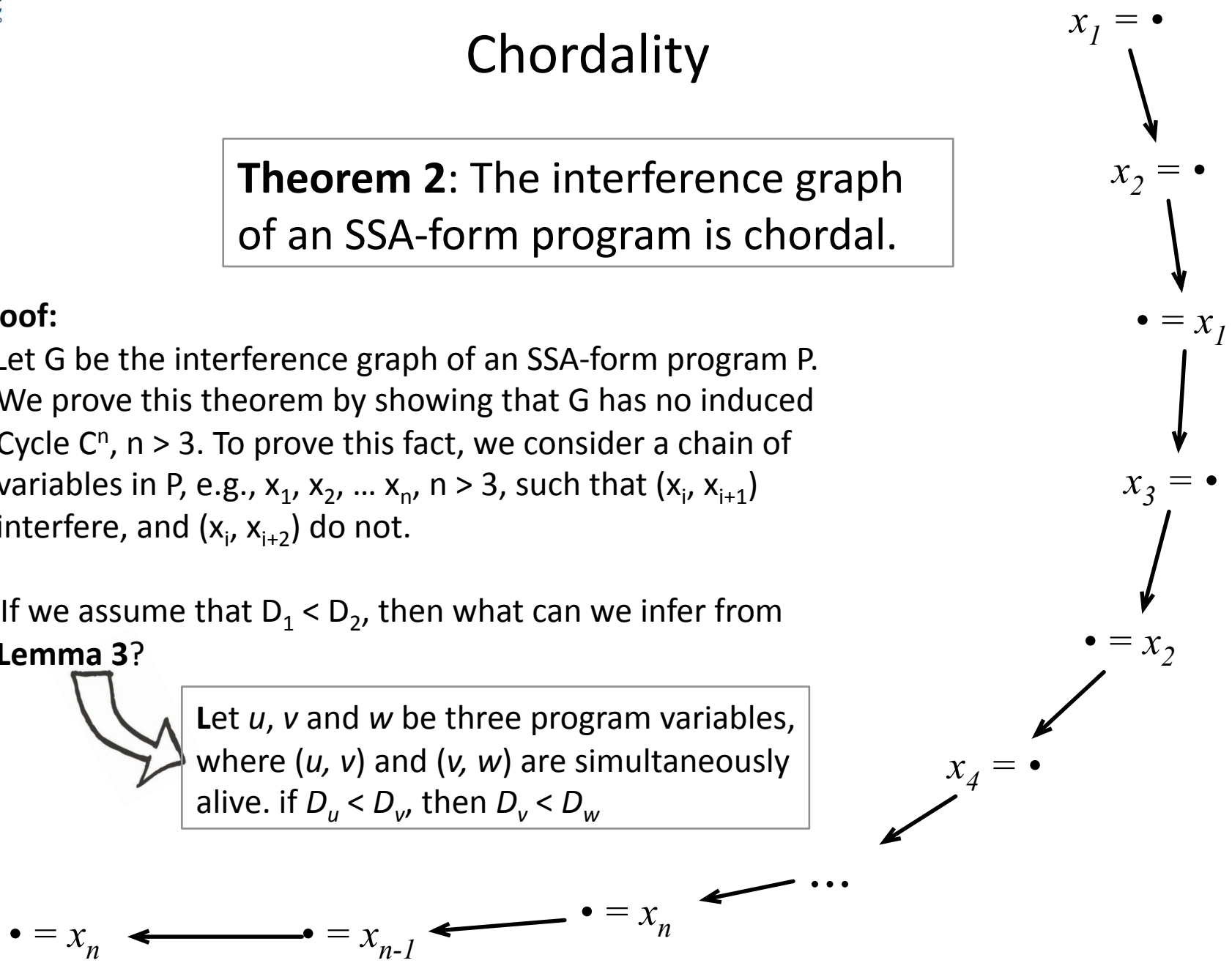
Theorem 2: The interference graph of an SSA-form program is chordal.

Proof:

Let G be the interference graph of an SSA-form program P. We prove this theorem by showing that G has no induced Cycle C^n, n > 3. To prove this fact, we consider a chain of variables in P, e.g., x_1, x_2, … x_n, n > 3, such that (x_i, x_{i+1}) interfere, and (x_i, x_{i+2}) do not.

If we assume that $D_1 < D_2$, then what can we infer from **Lemma 3**?

Let *u*, *v* and *w* be three program variables, where (*u, v*) and (*v, w*) are simultaneously alive. if $D_u < D_v$, then $D_v < D_w$



Chordality

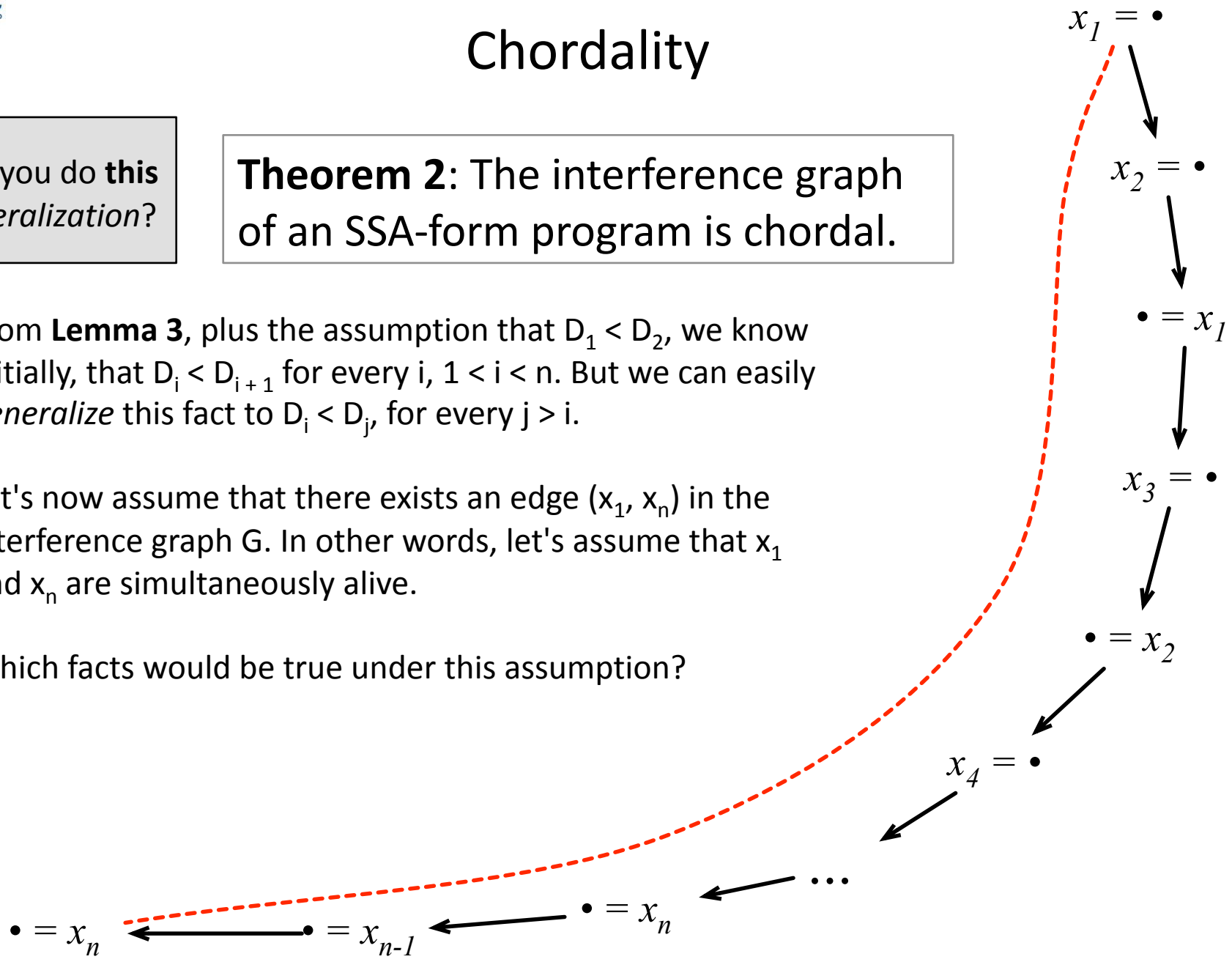
Can you do **this** *generalization*?

Theorem 2: The interference graph of an SSA-form program is chordal.

From **Lemma 3**, plus the assumption that $D_1 < D_2$, we know initially, that $D_i < D_{i+1}$ for every i, $1 < i < n$. But we can easily *generalize* this fact to $D_i < D_j$, for every $j > i$.

Let's now assume that there exists an edge (x_1, x_n) in the interference graph G. In other words, let's assume that x_1 and x_n are simultaneously alive.

Which facts would be true under this assumption?



Chordality

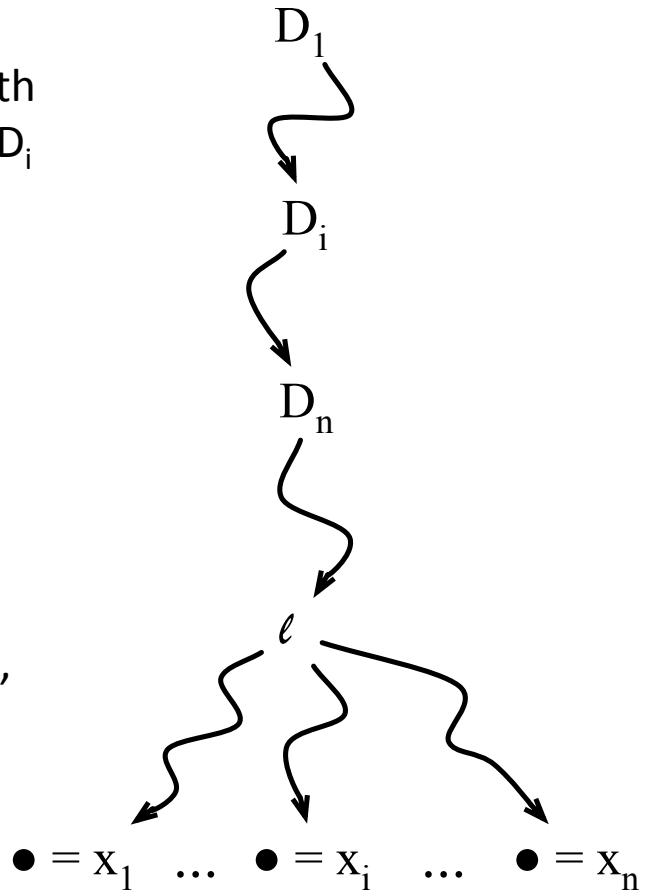
Theorem 2: The interference graph of an SSA-form program is chordal.

If x_1 and x_n are simultaneously alive, then there exists a program point ℓ that is dominated by the definition of both variables. Because ℓ is dominated by D_n, and every other D_i dominates D_n, we know that ℓ is dominated by every D_i.

If we consider any D_i, $1 < i < n$, then we know that:

- There exists a path from D_i to ℓ, because D_i dominates ℓ
- This path does not contain D_1, because D_i does not dominate D_1.
- There exists a path from D_1 to D_i, as $D_1 < D_i$.

Thus, x_1 is alive at D_i, contradicting our initial assumption, e.g., (x_i, x_{i+2}) do not interfere

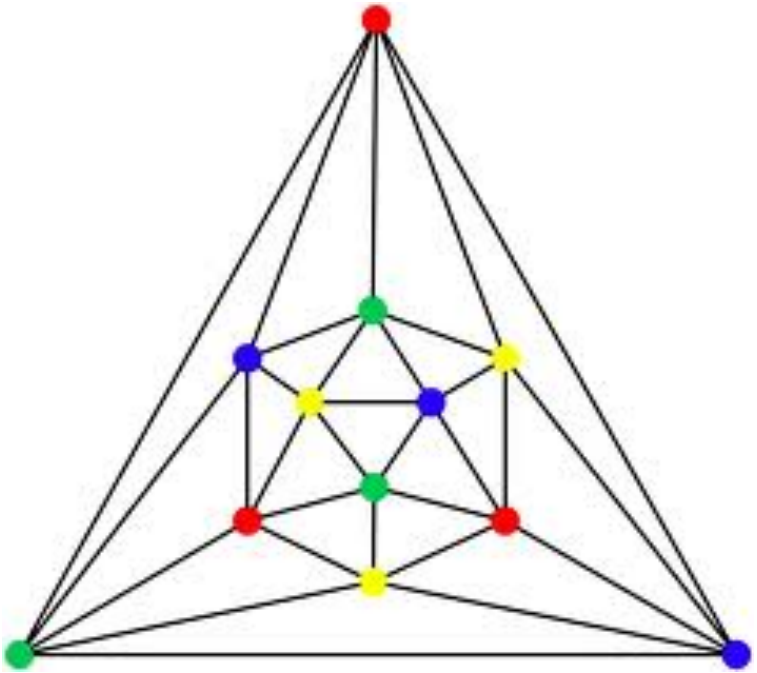


Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

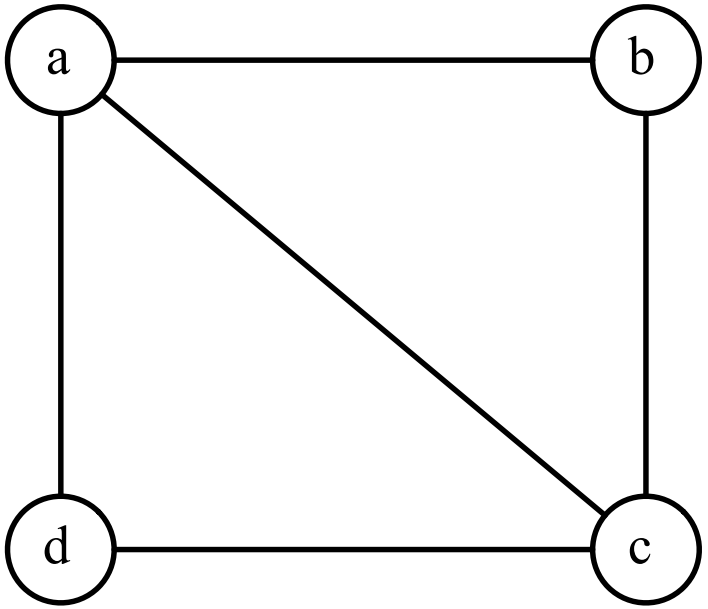
COLORING OF CHORDAL GRAPHS

DCC 888



Simplicial Elimination Ordering (SEO)

- If G = (V, E) is a graph, then a vertex v ∈ V is called *simplicial* if, and only if, its neighborhood in G is a clique.
- A *Simplicial Elimination Ordering* of G is a bijection σ: V(G) → {1, …, |V|}, such that every vertex v_i is a simplicial vertex in the subgraph induced by $\{v_1, \ldots, v_i\}$.



The nodes b and d are simplicial. For instance, b has two neighbors, a and c, and these neighbors form a clique. On the other hand, a and c are not simplicial.

SEO and Chordal Graphs

Theorem 3♧: An undirected graph without self-loops is chordal if, and only if, it has a simplicial elimination ordering

- The greedy coloring of a simplicial elimination ordering yields an optimal solution.
  - In the algorithm below, we let Δ(G) be the largest clique in G

Greedy Coloring

input: G = (V, E), a sequence S of vertices in V
output: a mapping m, m(v) = c, $0 \leq c$, $v \in V$
for all $v \in S$ do $m(v) \leftarrow \perp$
for $i \leftarrow 1$ to $|S|$ **do**
  let c be the lowest color not used in N(S[i]) **in**
    $m(S[i]) \leftarrow c$

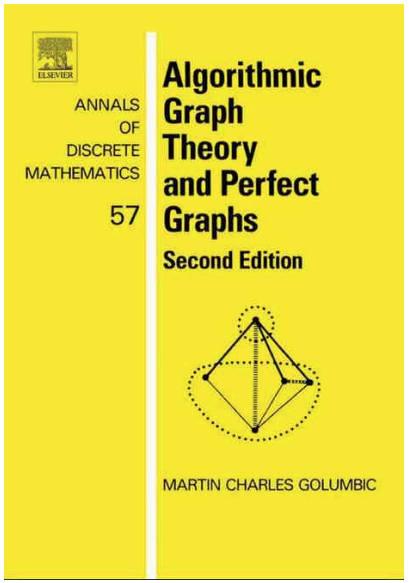
Neighborhood

But, how can we find a simplicial elimination ordering?

♧: On Rigid Circuit Graphs, G. A. Dirac (1991)

Maximum Cardinality Search

- There exist algorithms that sort the vertex of a graph in simplicial elimination order.
- We will use the Maximum Cardinality Search (MCS), which is given below:



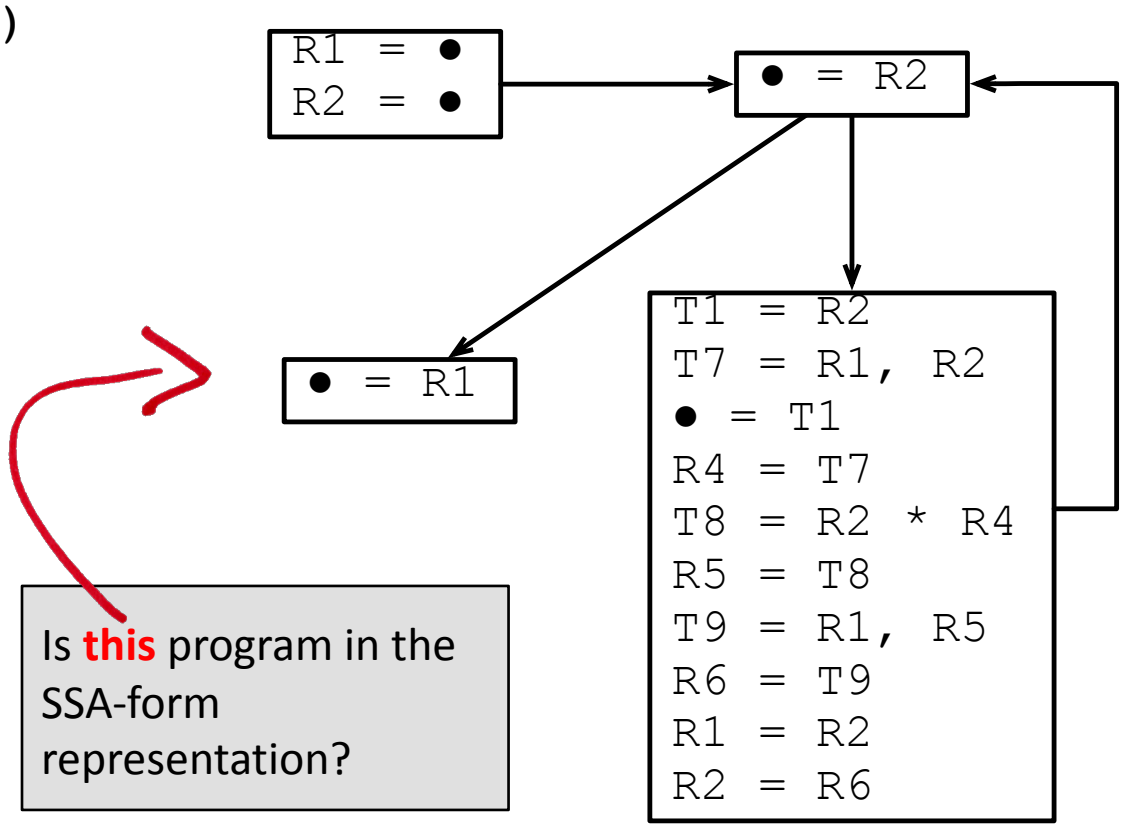
Maximum Cardinality Search

input: $G = (V, E)$
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $|V|$ **do**
  let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    for all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
    $V = V \setminus \{v\}$

Running Example

- We will be using an example throughout the rest of this presentation, to illustrate how we can take benefit from chordality to do better register allocation:

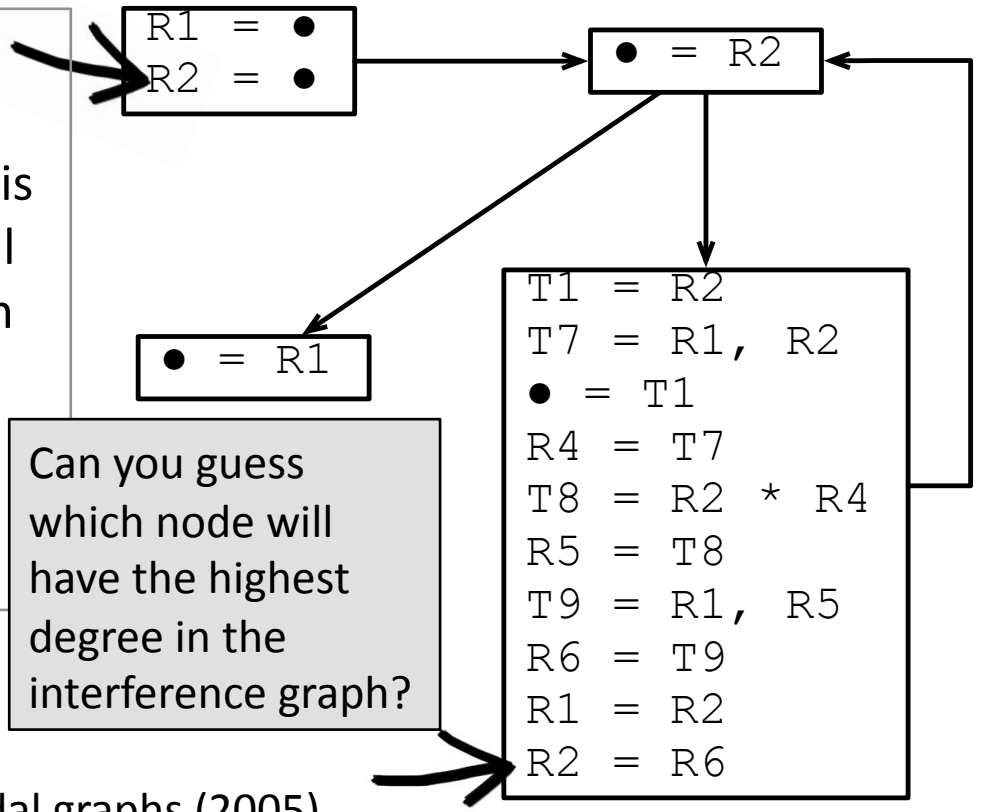
```
int gcd(int R1, int R2)
1.   if R2 != 0 goto 13
2.   T1 = is_zero R2
3.   T7 = R1 / R2
4.   check_exception T1
5.   R4 = T7
6.   T8 = R2 * R4
7.   R5 = T8
8.   T9 = R1 - R5
9.   R6 = T9
10.  R1 = R2
11.  R2 = R6
12.  goto (1)
13.  return R1
```



Running Example

- We will be using an example throughout the rest of this presentation, to illustrate how we can take benefit from chordality to do better register allocation:

This program is not in the SSA-form intermediate representation. Nevertheless, its interference graph is chordal. The algorithms that we shall see work for any chordal graph, even if that graph has not been derived from an SSA-form program. In fact, many non-SSA-form programs also have chordal graphs♧.



Can you guess which node will have the highest degree in the interference graph?

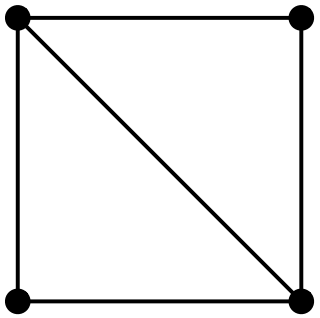
♧: Register allocation via the coloring of chordal graphs (2005)

Running Example

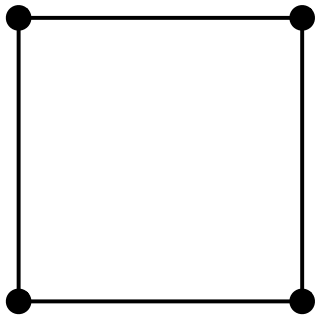
Below we see the interference graph of our running example. A graph G is chordal if, and only if, the largest chordless cycle that G contains has no more than three nodes. As we had said before, because of this definition, chordal graphs are also called triangular graphs.

1) Which are the three definitions of chordal graphs that we have seen in this class?

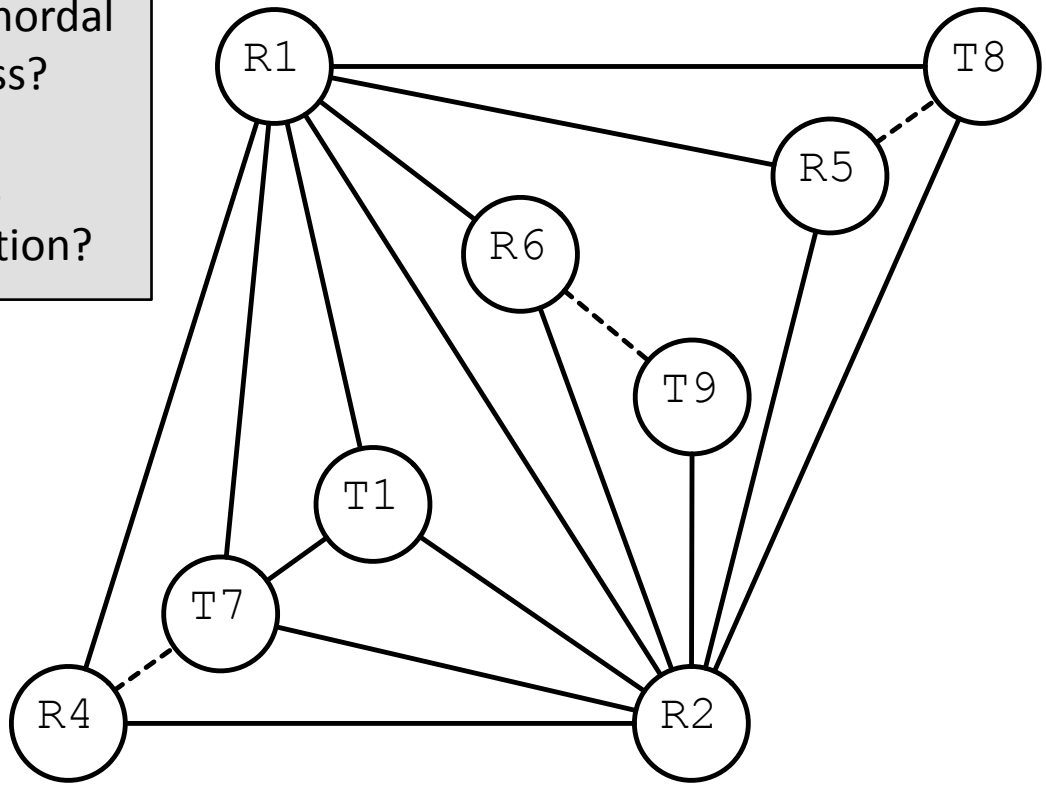
2) Can you make sure that this graph is chordal according to this new definition?



Chordal



Non-Chordal

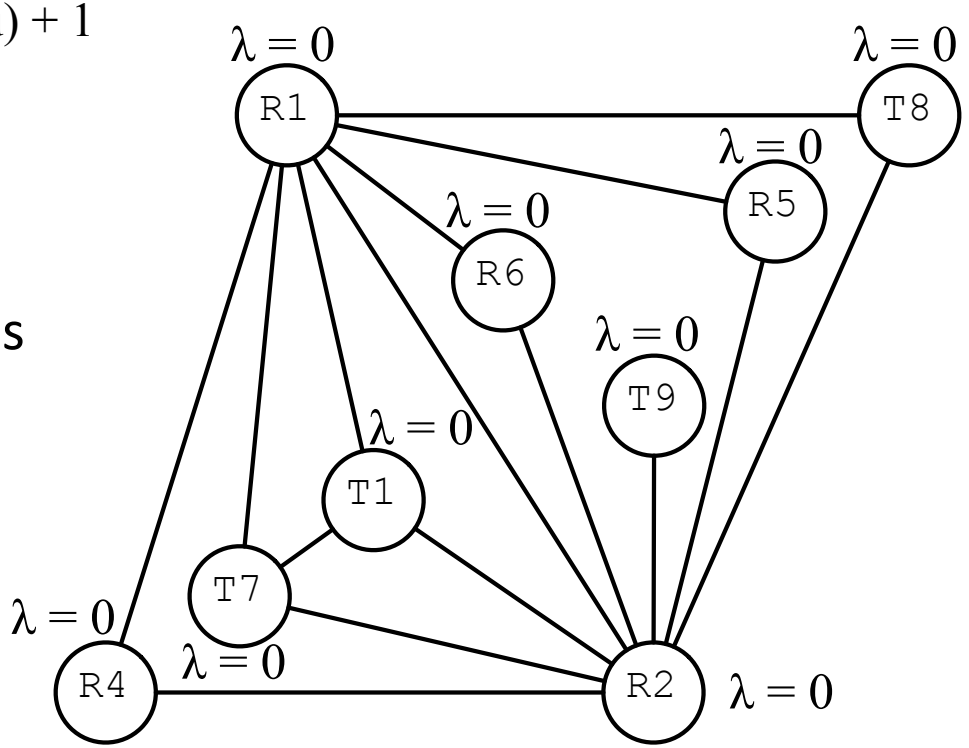


Applying MCS on the Example

Maximum Cardinality Search

input: G = (V, E)
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $|V|$ **do**
  let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    for all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
    $V = V \setminus \{v\}$

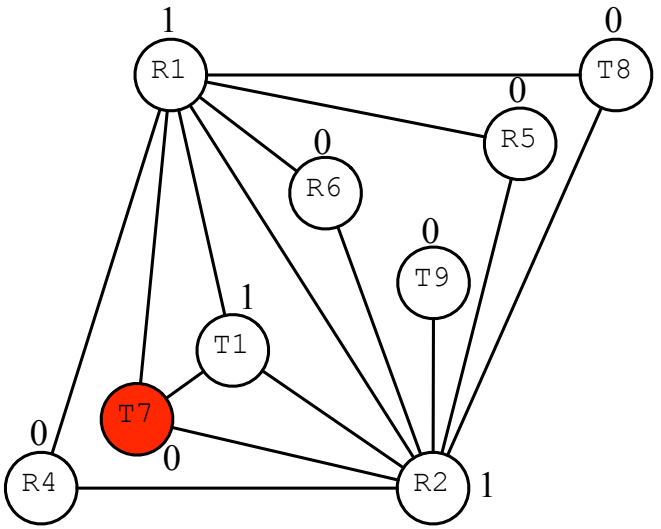
We let λ(v) be the label associated with node v. Initially every node has label zero.



Applying MCS on the Example

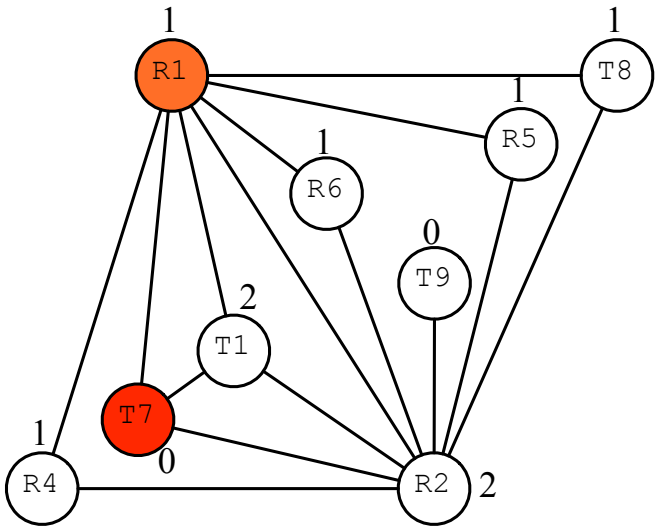
Maximum Cardinality Search

input: G = (V, E)
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $|V|$ **do**
  let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    for all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
    $V = V \setminus \{v\}$



σ = T7,

What is going to be the next node to be chosen by our MCF procedure?



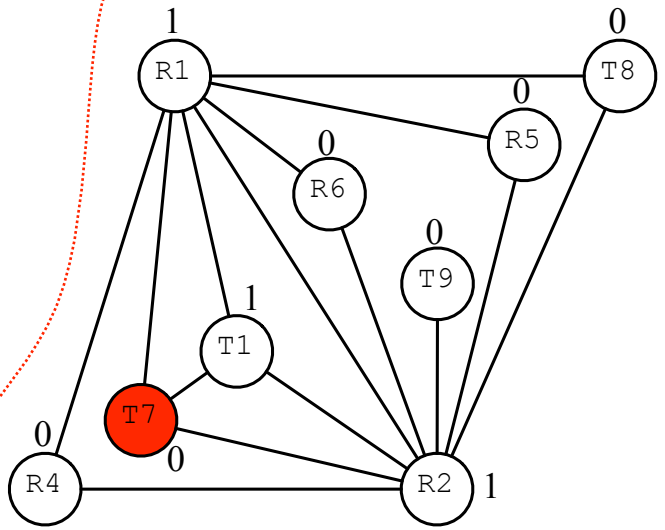
σ = T7, R1,

Applying MCS on the Example

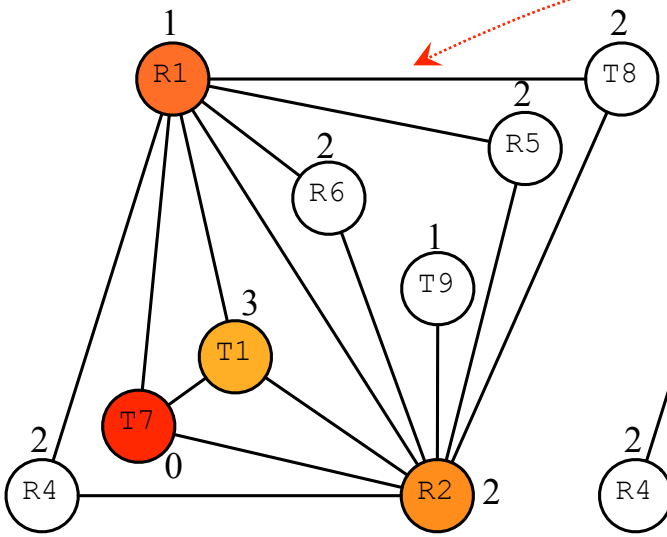
And now, what is the next node?

Maximum Cardinality Search

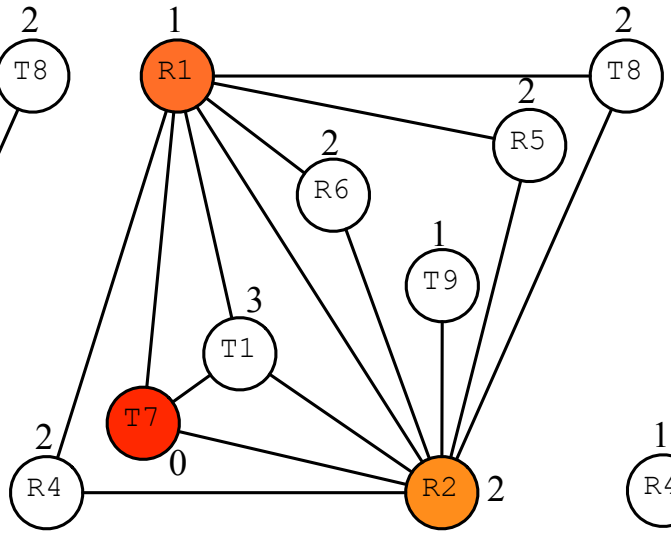
input: G = (V, E)
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $|V|$ **do**
  let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    for all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
    $V = V \setminus \{v\}$



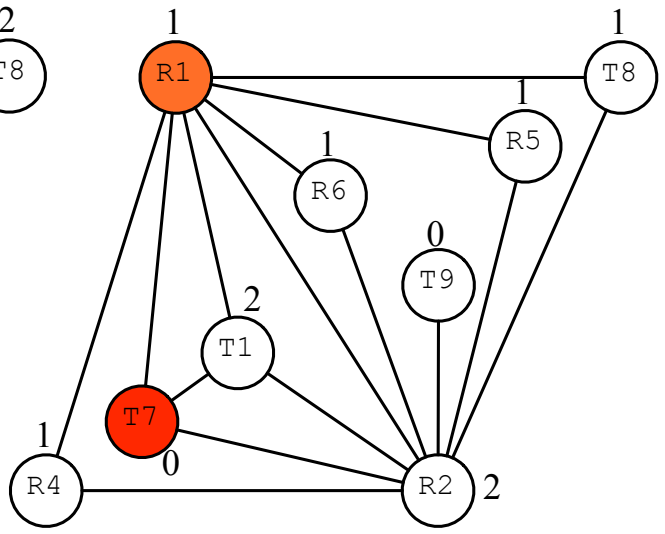
σ = T7,



σ = T7, R1, R2, T1,



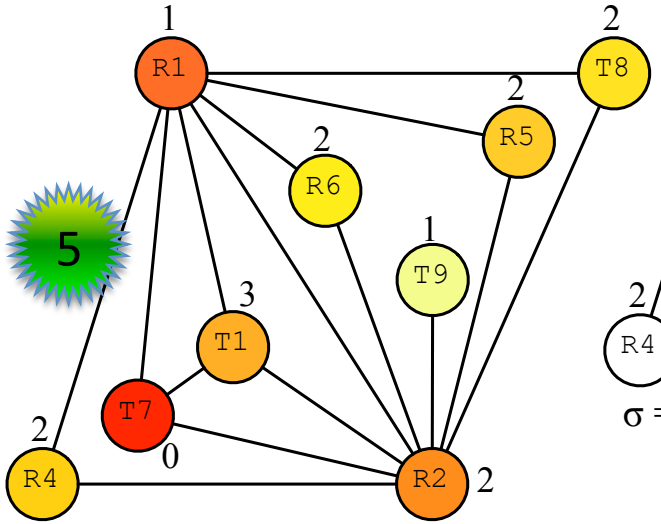
σ = T7, R1, R2,



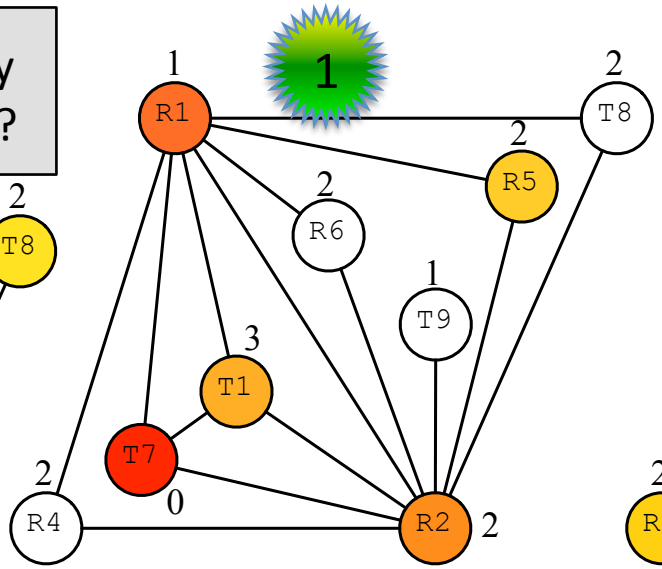
σ = T7, R1,

Applying MCS on the Example

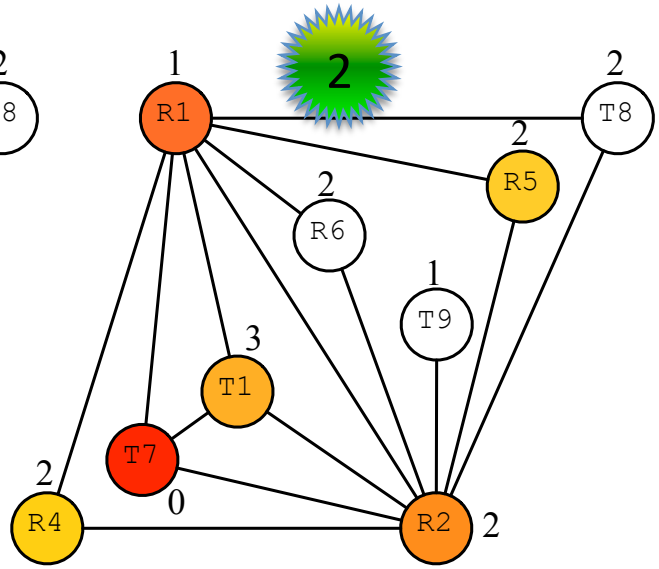
What is the complexity of the MCF procedure?



σ = T7, R1, R2, T1, R5, R4, T8, R6, T9

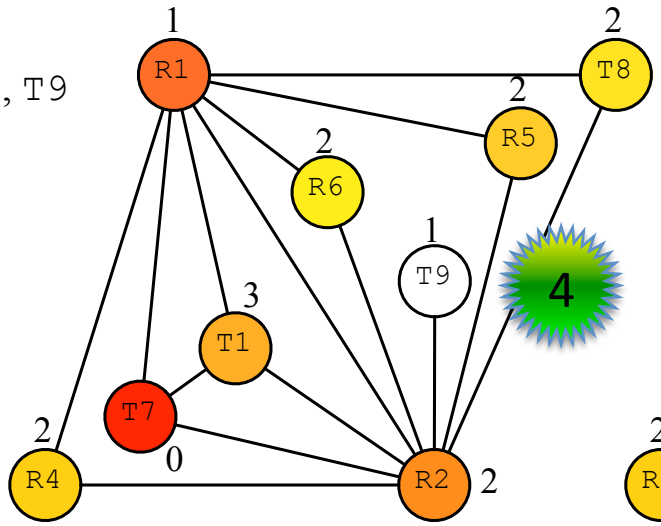


σ = T7, R1, R2, T1, R5,

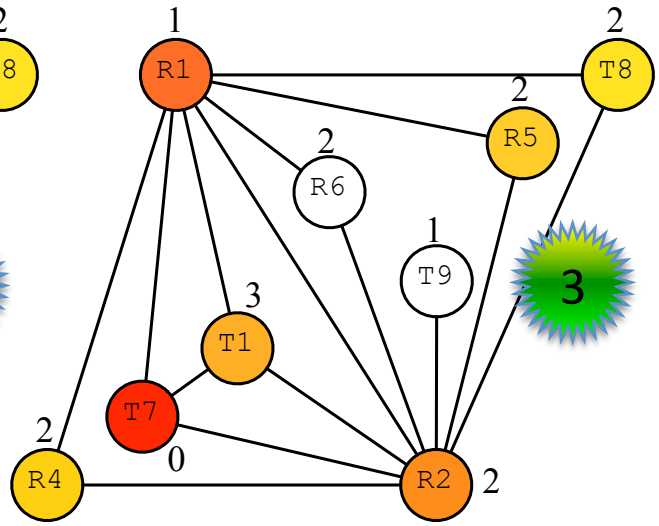


σ = T7, R1, R2, T1, R5, R4,

We are done! If we start eliminating nodes, from T9 to T7, we will see that they are all simplicial.

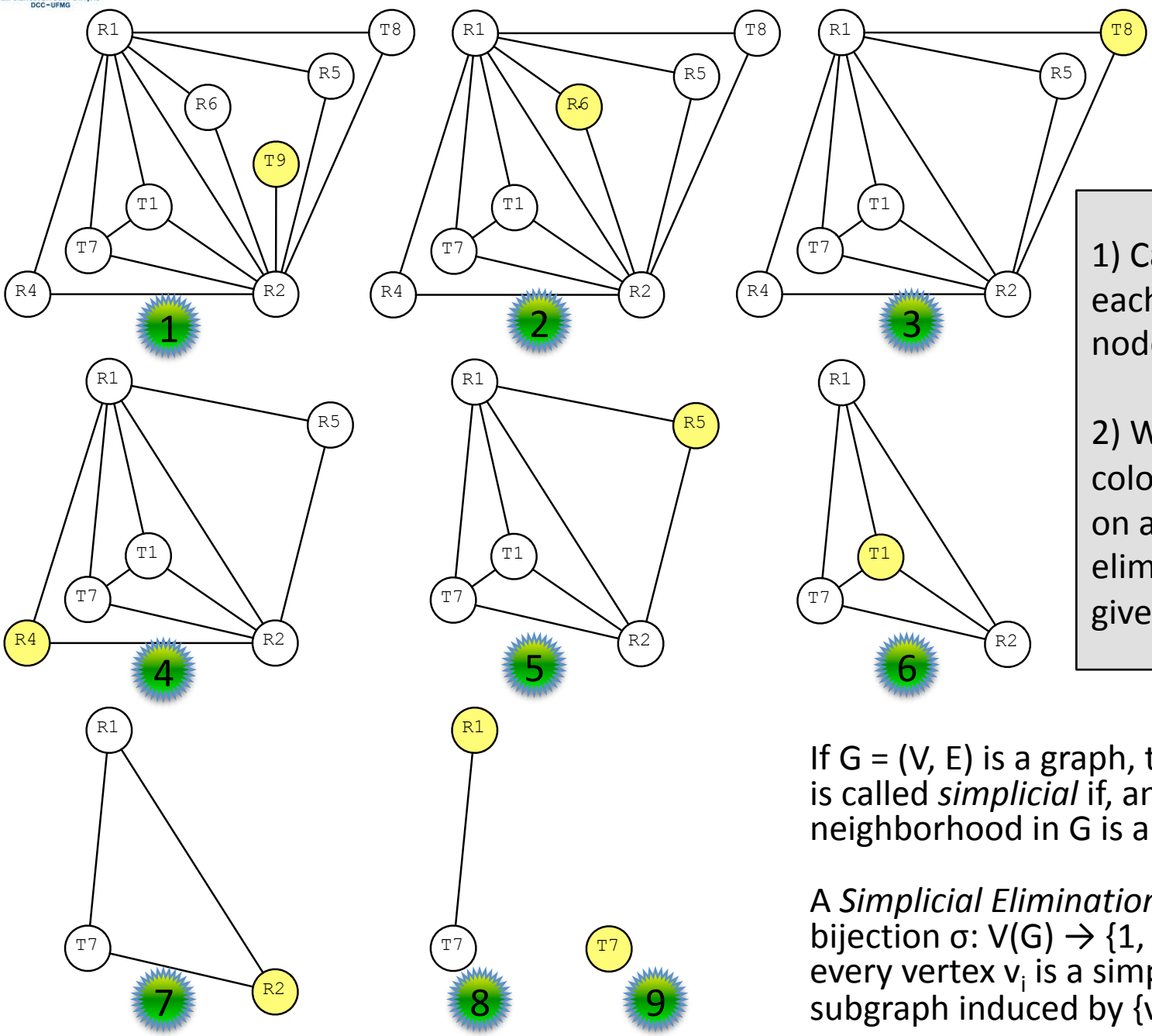


σ = T7, R1, R2, T1, R5, R4, T8, R6



σ = T7, R1, R2, T1, R5, R4, T8,

σ = T7, R1, R2, T1, R5, R4, T8, R6, T9



1) Can you show that each of these yellow nodes is simplicial?

2) Why does the greedy coloring, when applied on a simplicial elimination ordering, give us a tight coloring?

If G = (V, E) is a graph, then a vertex $v \in V$ is called *simplicial* if, and only if, its neighborhood in G is a clique.

A *Simplicial Elimination Ordering* of G is a bijection σ: V(G) $\rightarrow$ {1, …, |V|}, such that every vertex v_i is a simplicial vertex in the subgraph induced by $\{v_1, \ldots, v_i\}$.

Greedy Coloring and the SEO

Greedy coloring in the simplicial elimination ordering yields an optimal coloring.

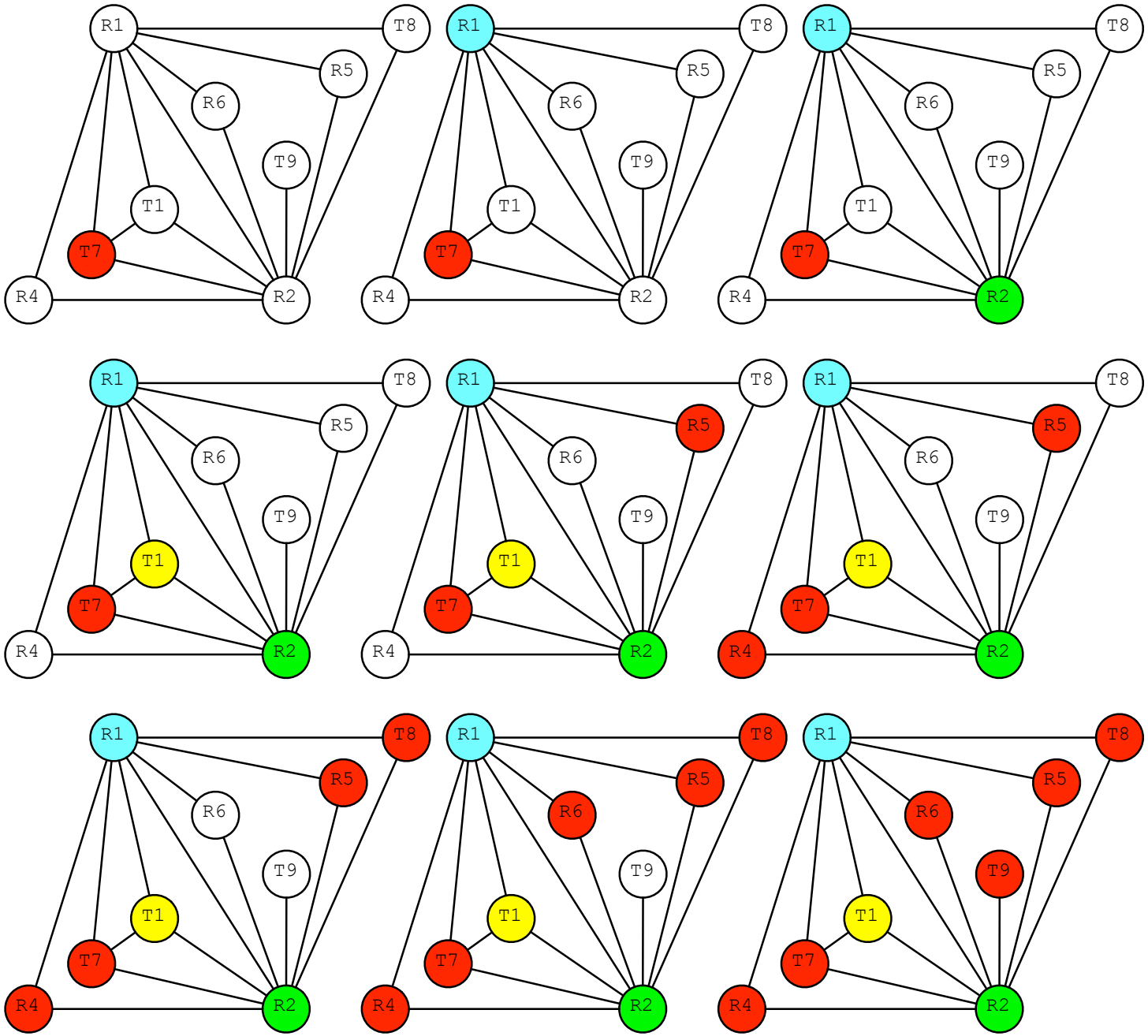
- If we greedily color the nodes in the order given by the SEO, then, upon finding the n-th node v in this ordering, all the neighbors of v that have been already colored form a clique.
- All the nodes in a clique must receive different colors.
- Thus, if v has M neighbors already colored, we will have to give it color M+1.

Consequently, in a chordal graph the size of the largest clique equals its chromatic number.

σ = T7, R1, R2, T1, R5, R4, T8, R6, T9

Colors: 1 2 3 4

Can you double check that whenever we color a node, all its neighbors that have already been colored form a clique?



Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

DECOUPLED REGISTER ALLOCATION

DCC 888

MaxLive = MaxClique

Theorem 4[♧]: Let P be an SSA-form program, and G = (V, E) be its interference graph. For each **clique** C = $\{c_1, \ldots, c_n\}$ in G there exists a label ℓ in P, where all the nodes c_i interfere.

Can you provide an intuition on why this theorem is true?

A clique of a graph G = (V, E) is a subgraph of G in which every two vertices are adjacent.

♧: Register allocation for programs in SSA-form, 2006

MaxLive = MaxClique

Theorem 4[♧]: Let P be an SSA-form program, and G = (V, E) be its interference graph. For each clique $C = \{c_1, \ldots, c_n\} \in V$ there exists a label ℓ in P, where all the nodes c_i interfere.

Proof:
Since C is a clique, $(c_i, c_j) \in E$ for each $1 \leq i < j \leq n$. From **Lemma 1**, the labels $\{D_1, \ldots, D_n\}$ form a totally ordered set. Thus, it is possible to find an ordering $D_x < D_y$ in the dominance relation. From **Lemma 2**, all the variables are alive at the definition of the lowest node in this ordering.

Lemma 1: if two variables, *u* and *v*, interfere, in a strict SSA-form program, then either $D_u < D_v$, or $D_v < D_u$.
Lemma 2: in a strict SSA form program, if two vars, *u* and *v*, interfere and $D_u < D_v$, then *u* is alive at D_v.

Ps.: Remember, the dominance relation is anti-symmetric and transitive.

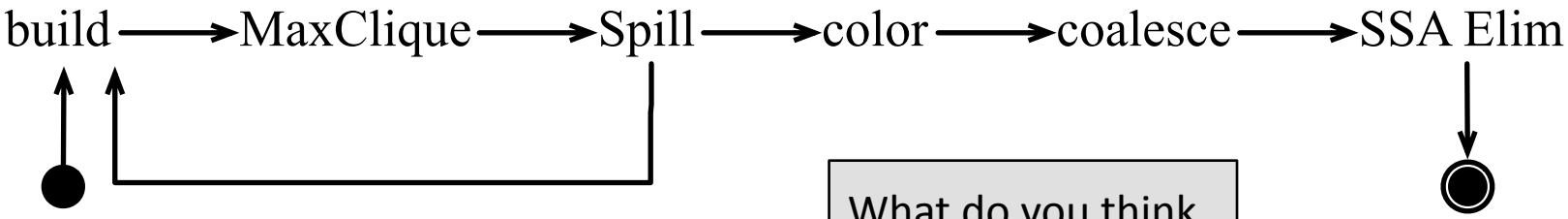
[♧]: Register allocation for programs in SSA-form, 2006

Decoupled Spilling

- Because the maximum clique of the interference graph equals the minimum number of registers necessary to compile the program, we can lower register pressure until MaxLive = K, and just then we perform register assignment.
- This technique is called the *decoupled approach* to register allocation.
  - First we spill
  - Then we do register assignment
- As we have already seen, there exist an exact, polynomial time, algorithm to find out the chromatic number of a chordal graph.

Decoupled Spilling

- The possibility of being able to spill, until we reach a colorable graph, gives us the opportunity to try many different algorithmic designs.
- Below we show the design used in the first register allocator based on the coloring of chordal graphs[◇]:

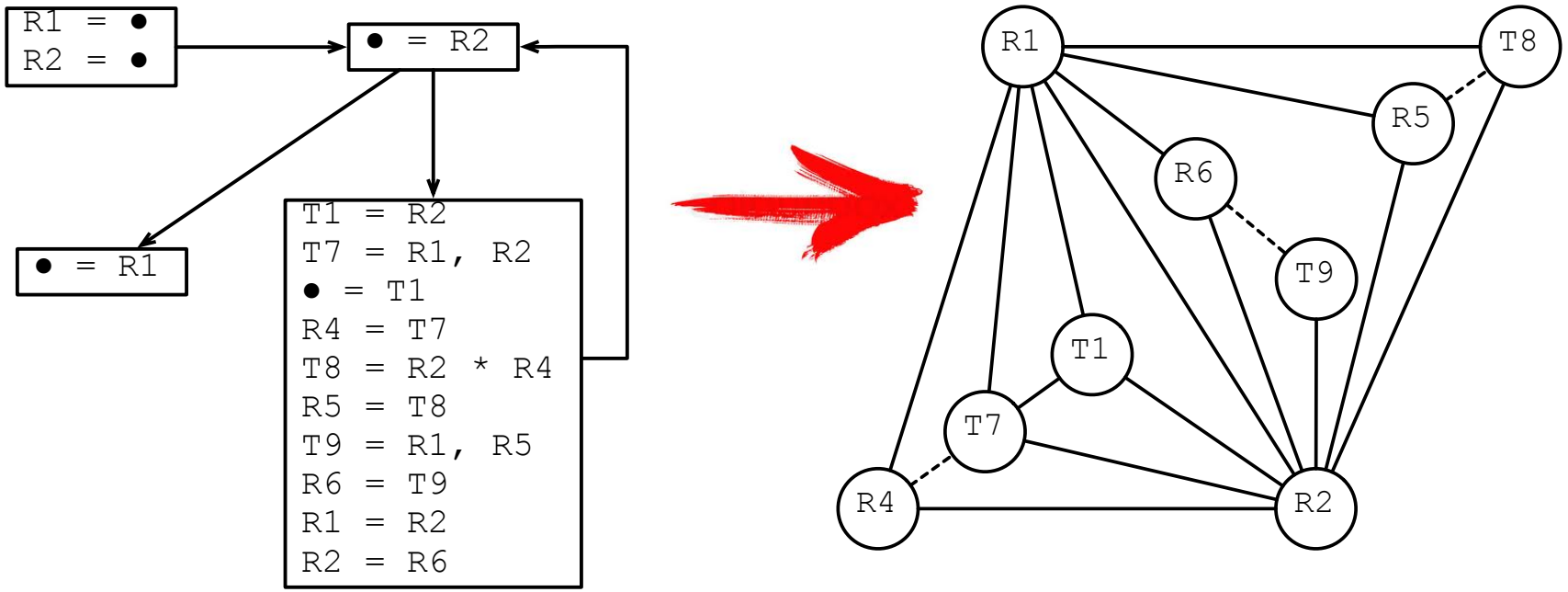


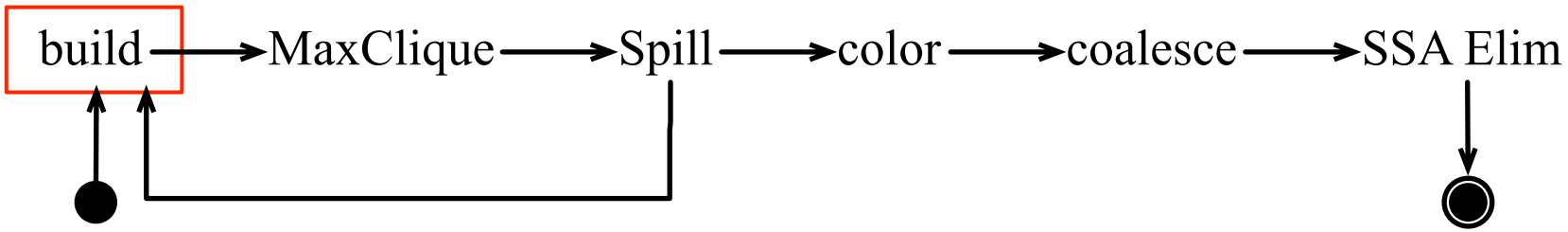
What do you think is the work of each of these phases?

[◇]: Register Allocation via the Coloring of Chordal Graphs (2005)

Build

- In the build phase we produce an interference graph out of liveness analysis.

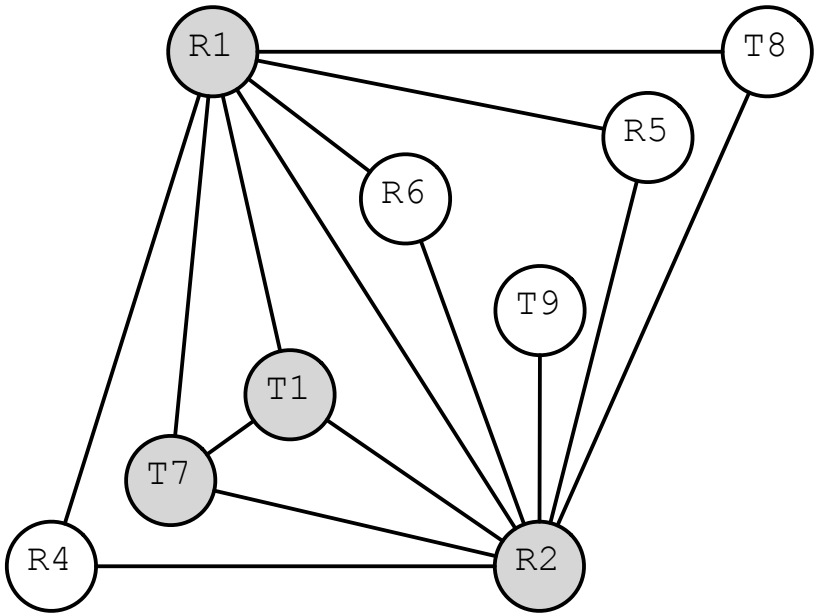




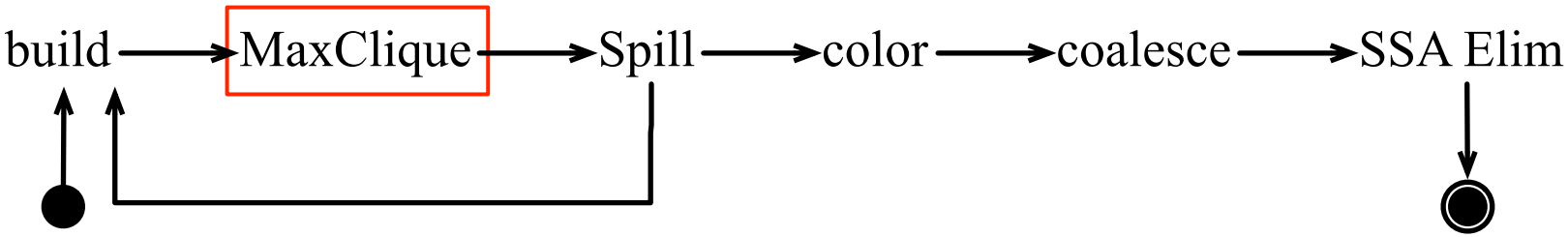
MaxClique

- In the MaxClique phase we try to find cliques with more than K nodes in the interference graph, where K is the maximum number of available registers.
- We find cliques using the MCF procedure that we have seen before.

$\sigma = \mathbf{T7, R1, R2, T1}, \mathtt{R5}, \mathtt{R4}, \mathtt{T8}, \mathtt{R6}, \mathtt{T9}$



How can we use the MCF method to find cliques?

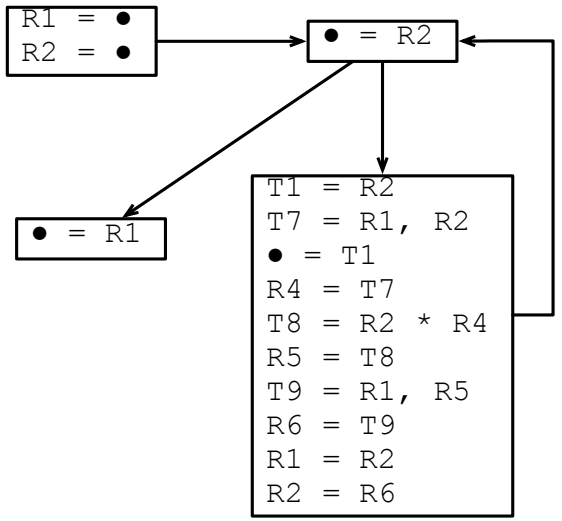


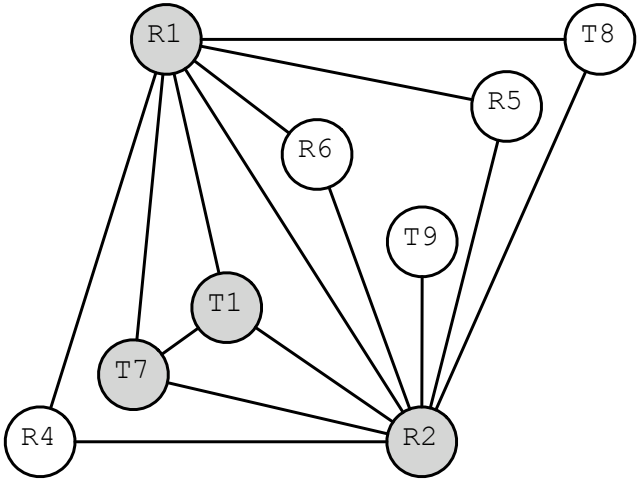
Spill

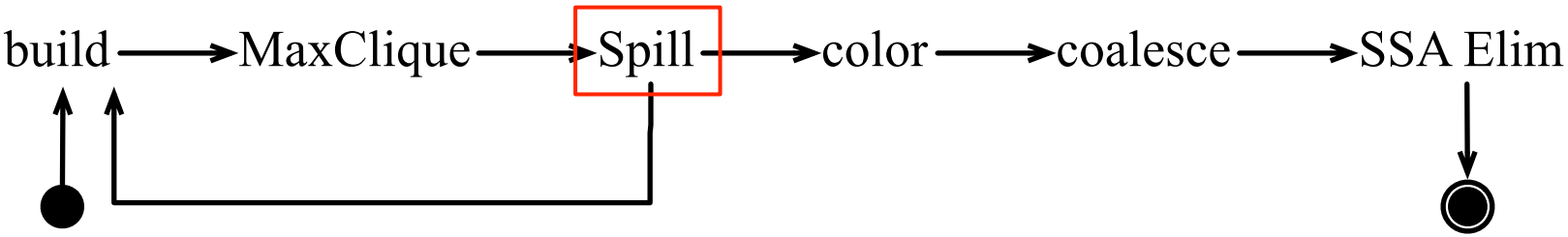
- If we have cliques with more than K nodes, then we must choose a few of these nodes to spill.
- The problem of finding the minimum number of nodes to spill, so that we get a K colorable graph is NP-complete♤.

1) How do we choose which node to spills?

2) In this example, we have a clique of four nodes. If we have only <u>three</u> registers, which node do we spill?



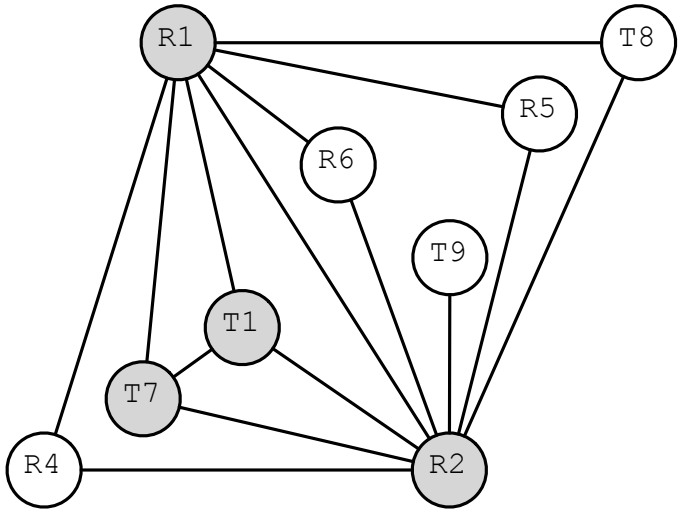




♤: The Maximum k-Colorable Subgraph Problem for Chordal Graphs (1987)

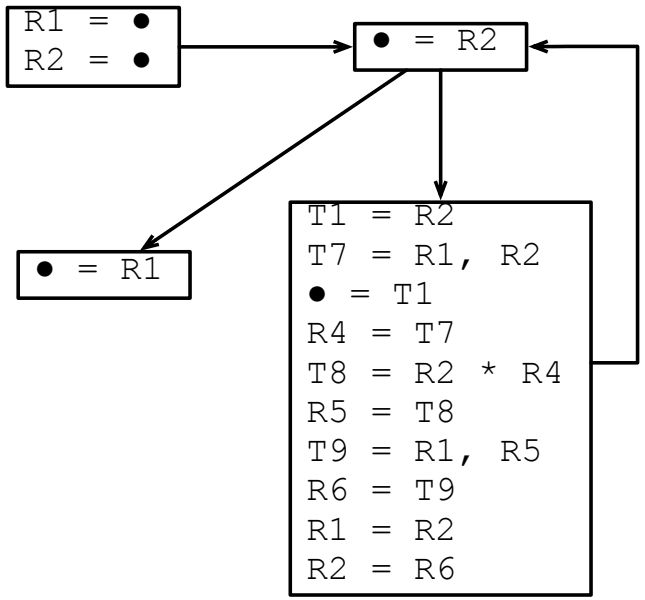
Spill

We can use the same formula that we have used in the design of Iterated Register Coalescing (Remember last class?) to compute spill costs. This formula takes into consideration the program, and the structure of its interference graph.



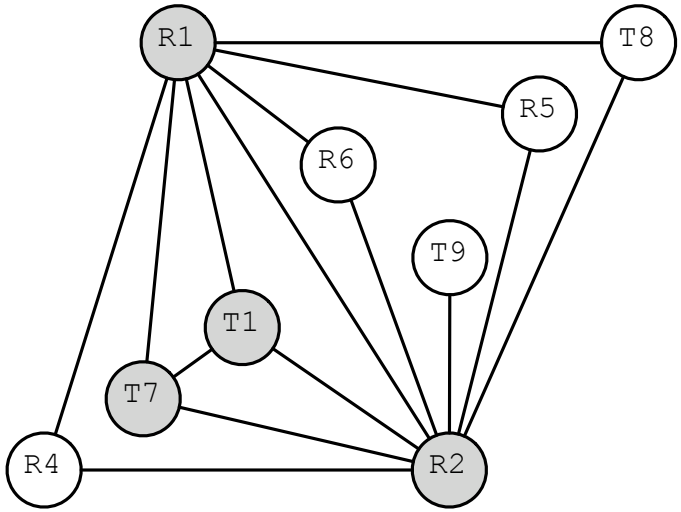
Which variable should we spill in this example?

Spill_Cost(v)
  cost = 0
  foreach definition at block B, or use at block B
    cost = (Σ ($S_B \times 10^N$))/D, where
      S_B is the number of uses and defs at B
      N is B's loop nesting factor
      D is v's degree in the interference graph



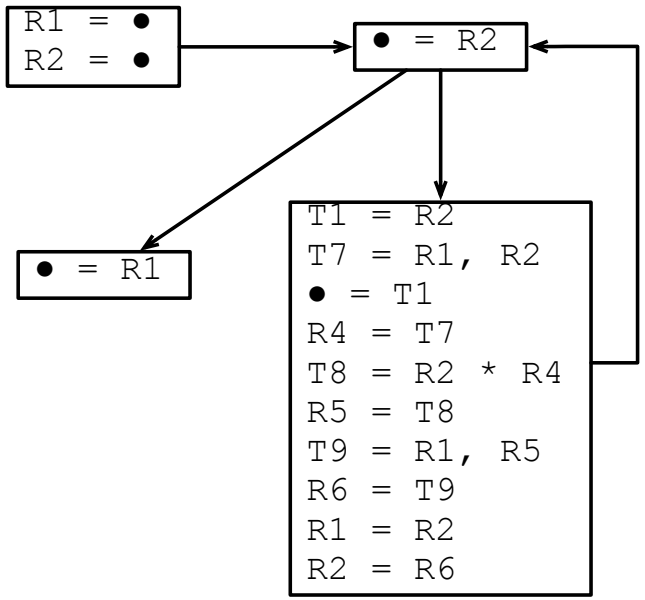
Spill

Node	Formula	Spilling Cost
T7	(2 * 10) / 3	6.66
R1	(1 + 1 + 3 * 10) / 7	4.57
R2	(1 + 1 + 5 * 10) / 8	6.5
T1	(2 * 10)/3	6.66



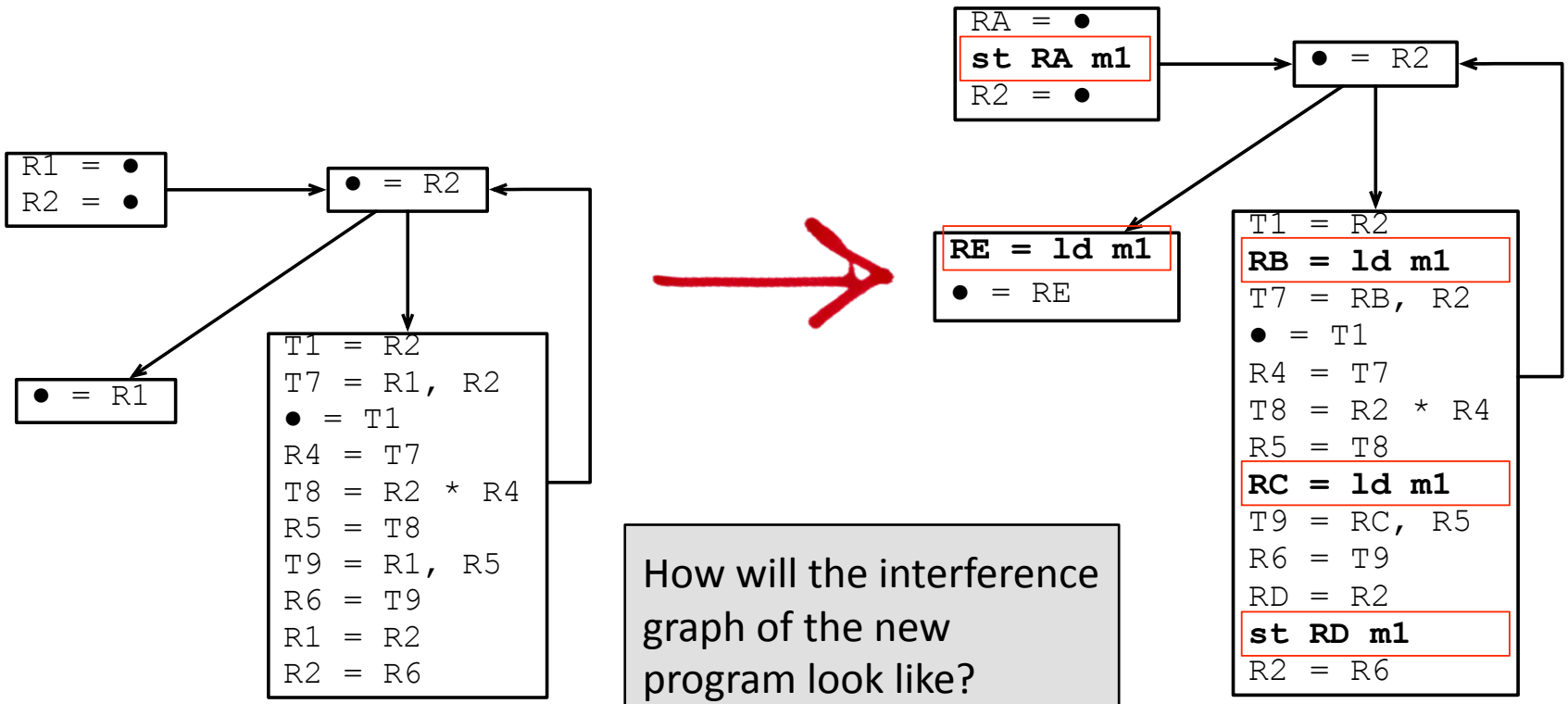
Which variable should we spill in this example?

Spill_Cost(v)
  cost = 0
  foreach definition at block B, or use at block B
    cost = (Σ ($S_B \times 10^N$))/D, where
      S_B is the number of uses and defs at B
      N is B's loop nesting factor
      D is v's degree in the interference graph



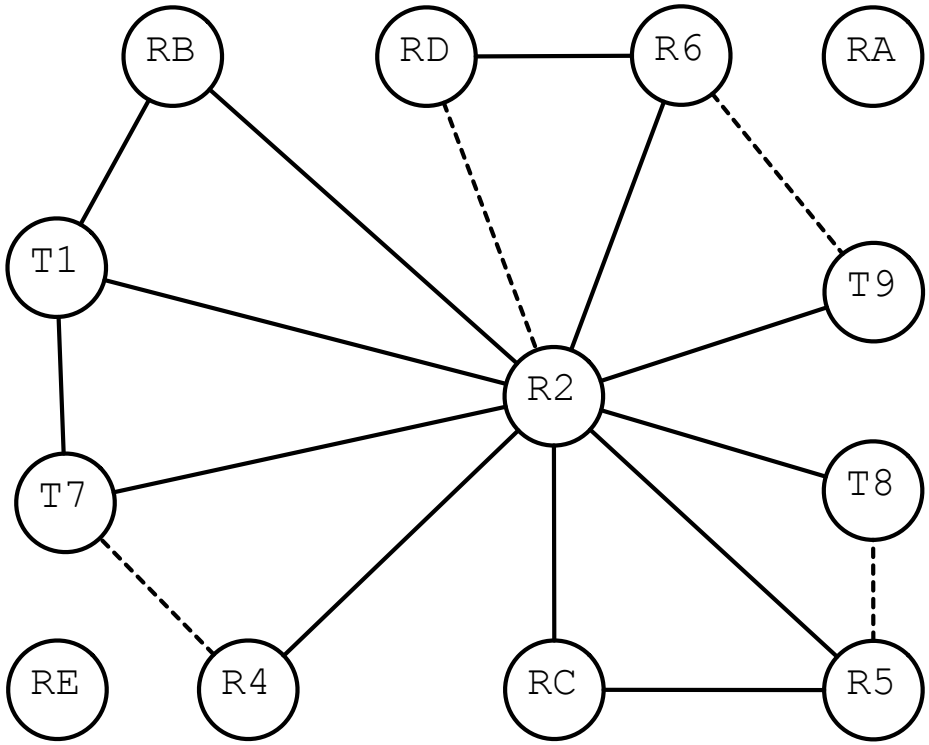
Rebuild

- Once we spill, we must insert loads and stores in the code, to preserve the semantics of the original program.
- After scattering loads and stores around, we rebuild the interference graph.



Rebuild

- Once we spill, we must insert loads and stores in the code, to preserve the semantics of the original program.
- After scattering loads and stores around, we rebuild the interference graph.




```
RA = ●
st RA m1
R2 = ●
```

```
● = R2
```

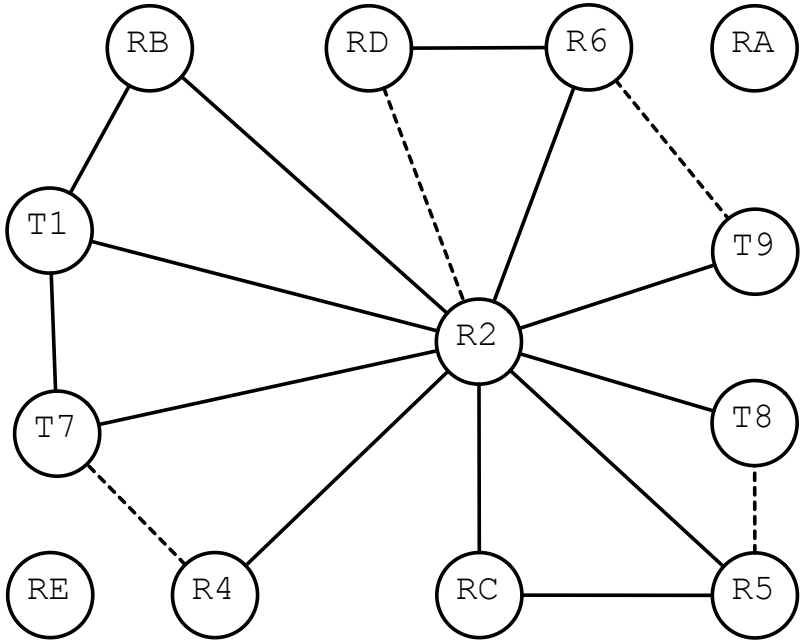
```
RE = ld m1
● = RE
```

```
T1 = R2
RB = ld m1
T7 = RB, R2
● = T1
R4 = T7
T8 = R2 * R4
R5 = T8
RC = ld m1
T9 = RC, R5
R6 = T9
RD = R2
st RD m1
R2 = R6
```

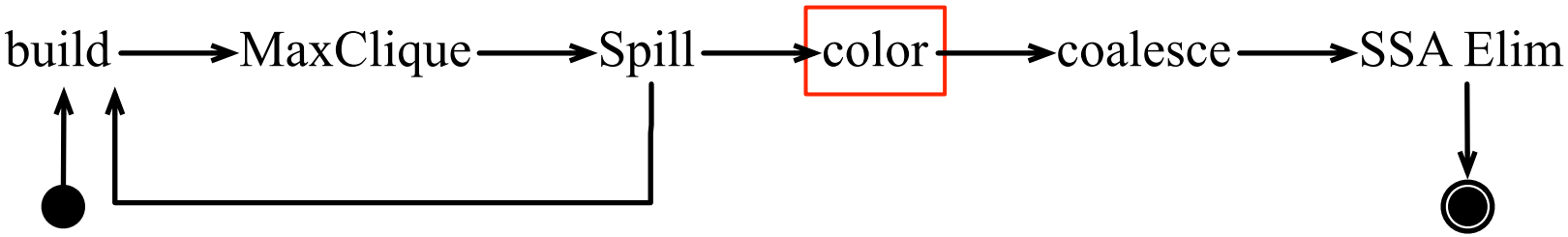
Is this graph 3-colorable?

Register Assignment

- Once we are down to a chordal graph whose largest clique has no more than K nodes, we are guaranteed to find a K-coloring to it.
- To find this coloring, we simply apply the greedy coloring on the simplicial elimination ordering that we obtain.

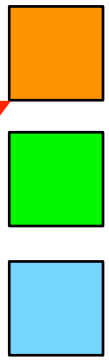


Can you find a SEO for this graph using our MCF procedure?

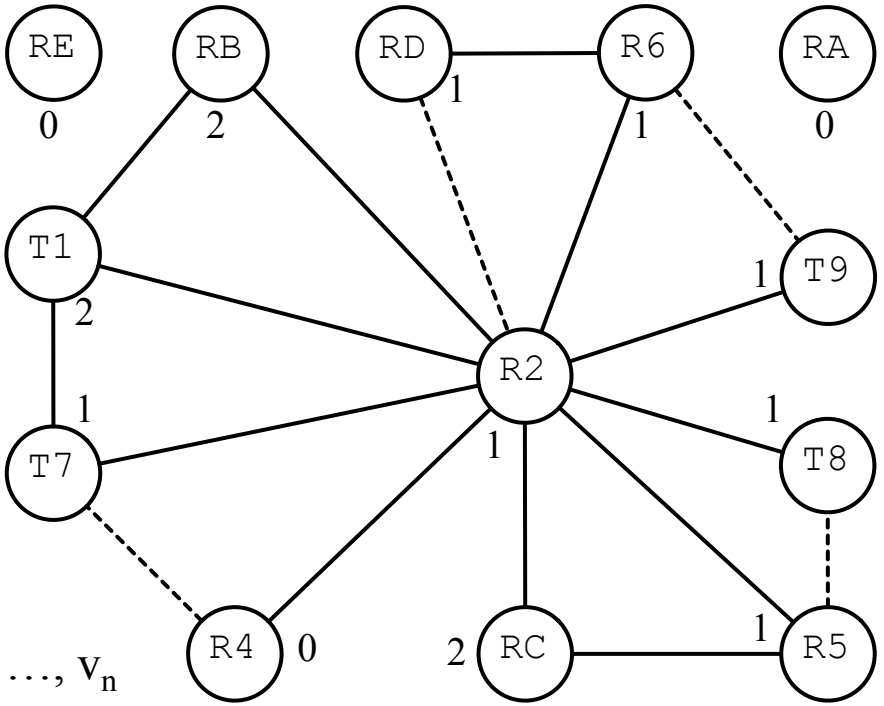


Register Assignment

Now, assuming that we have **three colors**, what do we get when we apply greed coloring on this σ?



σ = R4, R2, R6, T9, T8, T7, T1, RB, R5, RC, RD, RA, RE,



Maximum Cardinality Search

input: G = (V, E)
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $|V|$ **do**
  let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    for all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
    $V = V \setminus \{v\}$

Register Assignment

How can we map this coloring into a valid register assignment in our original program?

r1

r2

r3

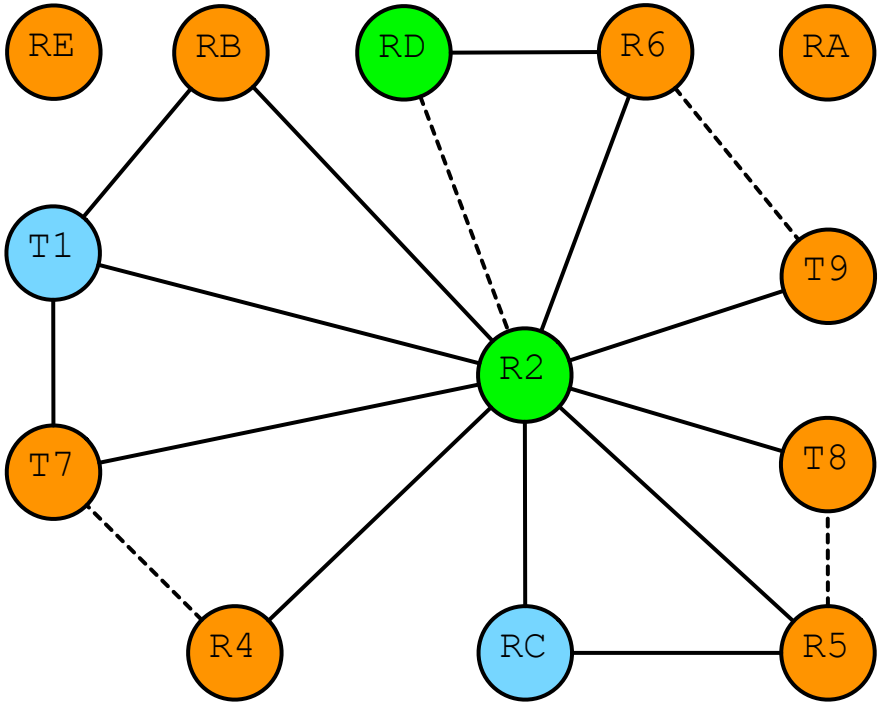
```
RA = ●
st RA m1
R2 = ●
```

```
● = R2
```

```
RE = ld m1
● = RE
```

```
T1 = R2
RB = ld m1
T7 = RB, R2
● = T1
R4 = T7
T8 = R2 * R4
R5 = T8
RC = ld m1
T9 = RC, R5
R6 = T9
RD = R2
st RD m1
R2 = R6
```

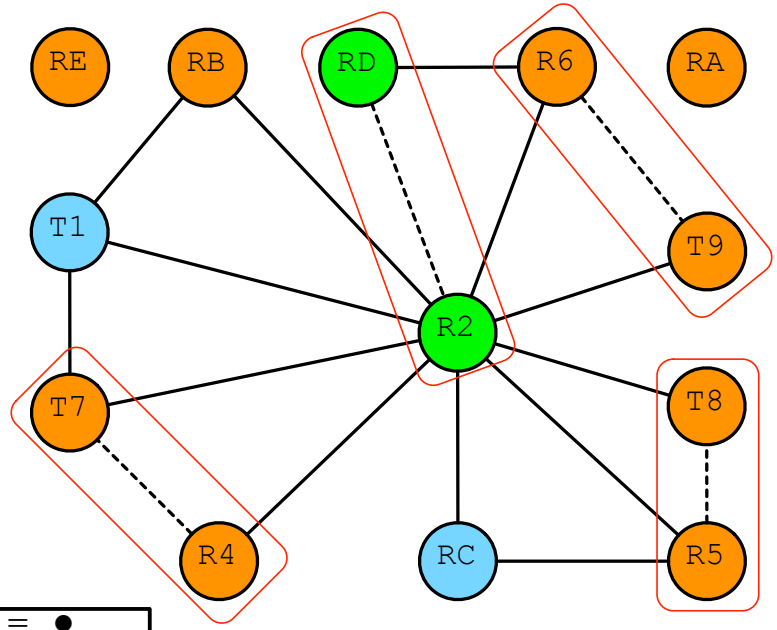
σ = R4, R2, R6, T9, T8, T7, T1, RB, R5, RC, RD, RA, RE,



Register Assignment

1) We have been lucky: all the coalescible nodes have been coalesced. Actually, the greedy coloring helps coalescing a little bit. Why?

2) Can you think about a simple coalescing strategy to our algorithm?




```
r1 = ●
st r1 m1
r2 = ●
```

```
● = r2
```

```
r1 = ld m1
● = r1
```

```
r3 = r2
r1 = ld m1
r1 = r1, r2
● = r3
r1 = r1
r1 = r2 * r1
r1 = r1
r3 = ld m1
r1 = r3, r1
r1 = r1
r2 = r2
st r2 m1
r2 = r1
```

```
r1 = ●
st r1 m1
r2 = ●
```

```
● = r2
```

```
r1 = ld m1
● = r1
```

```
r3 = r2
r1 = ld m1
r1 = r1, r2
● = r3
r1 = r2 * r1
r3 = ld m1
r1 = r3, r1
st r2 m1
r2 = r1
```

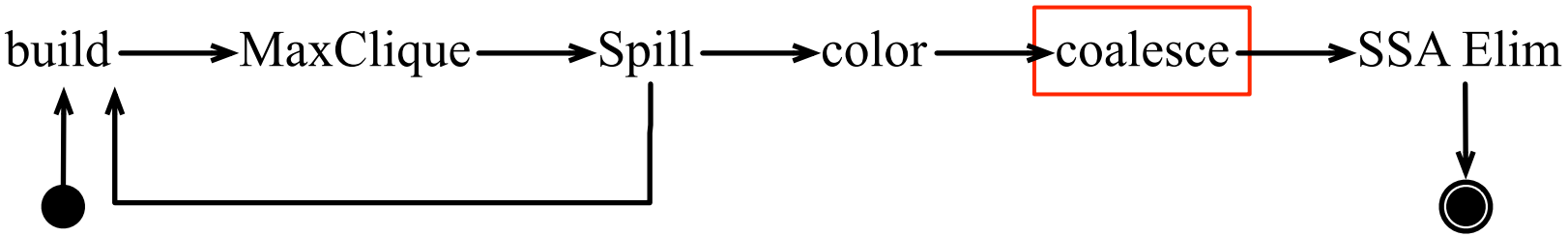
Best Effort Coalescing

- Because we have K colors to play with, some of them may end up not being used in some neighborhood of the interference graph.
- We can use these extra colors to maximize the amount of copy instructions that we can coalesce away.

1) How likely are we to have an unused color in some neighborhood of the interference graph?

2) This coalescing technique is rather simple. Can you think about anything stronger?

3) If we keep changing colors, our algorithm may oscillate without terminating. How can we ensure termination?



Best Effort Coalescing

Best Effort Coalescing

input: list L of copy instructions, G = (V, E), K
output: G', the coalesced graph G
G' = G
for all x = y ∈ L **do**
  let S_x be the set of colors in N(x)
  let S_y be the set of colors in N(y)
  if ∃ c, c < K, c ∉ $S_x \cup S_y$ **then**
    let xy, xy ∉ V be a new node **in**
      add xy to G' with color c
      make xy adjacent to every v, v ∈ N(x) ∪ N(y)
      replace occurrences of x or y in L by xy
      remove x from G'
      remove y from G'

What is the complexity of this algorithm?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

A Bit of History

- In this presentation we have used the algorithm introduced by Pereira and Palsberg.
- The proof that the interference graph of SSA-form programs are chordal was independently found by Hack et al., Florent Bouchez, and Brisk et al., in the mid 2000's.

- Pereira, F., and Palsberg, J., "Register Allocation via the Coloring of Chordal Graphs", APLAS, pp 315-329 (2005)
- Bouchez, F., "Allocation de Registres et Vidage en Memoire", MSc Dissertation, ENS Lyon, (2005)
- Hack, S., Grund, D., and Goos, G., "Register allocation for programs in SSA-form", CC, 247-262 (2006)
- Brisk, P., Dabiri, F., MacBeth, J., and Sarrafzadeh, M., "Polynomial-Time Graph Coloring Register Allocation", WLS, (2005)