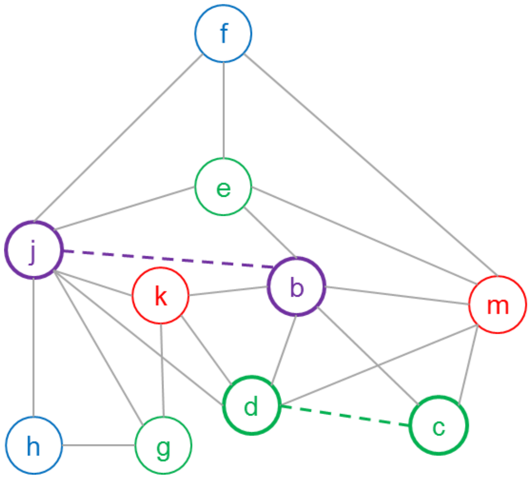
Register Allocation via Graph Coloring



4190.409 Compilers, Spring 2016

Overview

- Prerequisites
  - Live range computation through iterative dataflow analysis
  - Interference graph
- Global register allocation via graph coloring
  - Some Theory
  - Graph coloring: main idea
  - Optimistic Coloring
  - Coloring with Copy Propagation (Coalescing)

Register Allocation

Input

Program with
n names/registers

Register Allocation

Output

Program with
m registers

Typically unlimited number of
virtual registers

Limited to *physical* registers
of machine

…
if (c_{61} != 0) a_{127} = b_{23} / c_{61};
else a_{128} = 0;
a_{129} = φ(a_{127}, a_{128});
…

…
if (r_2 != 0) r_4 = r_1 / r_2;
else r_4 = 0;
~~r_4 = r_4;~~
…

Assumptions, Goals & Prerequisites

- **Assumption**
  Register-to-register model

- **Goal**
  Produce fast code
  - keep as many values in registers as possible, for as long as possible

  Ok, so:
  - which values?
  - from where to where?

- **Prerequisites**
  need to know the which values cannot be kept in the same register at the same time

Interference and Live Ranges

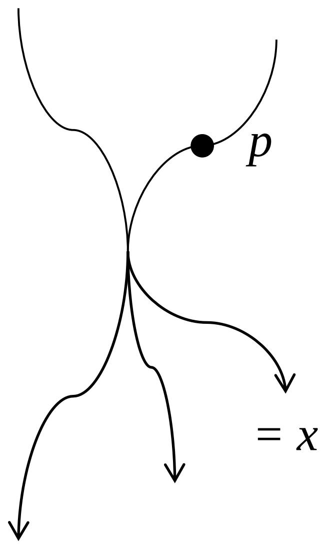
```
a = 0;
do {
  b = a+1;
  c = c+1;
  a = 2*b;
while (a < N);
return c;
```

- Can a and b occupy the same register (i.e., are they both *live* at some point in the program, do they *interfere*)? What about a and c? And b and c?

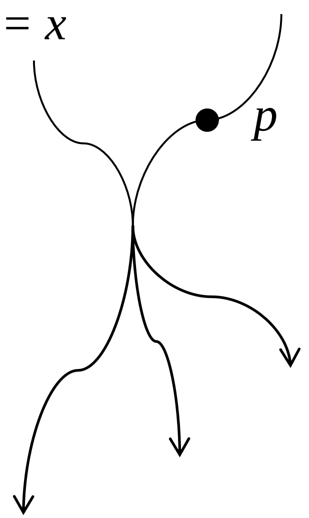
CSE 컴퓨터공학부
Department of Computer Science & Engineering

Liveness of a Variable

- Def. **Liveness**
  Variable x is *live* at point p iff the value of x at p could be used along some path in the flow graph starting at p. Otherwise, x is dead at p.



x is *live* at point p



x is *dead* at point p

Liveness Analysis

- Live Ranges for the three variables a, b, and c

```
a = 0;
do {
   b = a+1;
   c = c+1;
   a = 2*b;
while (a < N);
return c;
```

Liveness Analysis

- Live Ranges for the three variables a, b, and c

```
a = 0;
do {
   b = a+1;
   c = c+1;
   a = 2*b;
while (a < N);
return c;
```

Liveness Analysis

- Live Ranges for the three variables a, b, and c

```
a = 0;
do {
   b = a+1;
   c = c+1;
   a = 2*b;
while (a < N);
return c;
```

Liveness Analysis

- Live Ranges for the three variables a, b, and c

```
a = 0;
do {
   b = a+1;
   c = c+1;
   a = 2*b;
while (a < N);
return c;
```

Liveness Analysis

- Live Ranges for the three variables a, b, and c

```
a = 0;
do {
   b = a+1;
   c = c+1;
   a = 2*b;
while (a < N);
return c;
```

Iterative Dataflow Analysis in a Nutshell

- **Given:**
  CGF $G = (V, E)$
  $gen(v)/kill(v)$ functions
  a transfer function $f(v)$ and a meet operator $\sqcap$
  a boundary condition and initialization functions
  a direction (forward/backward)

- **Algorithm**

  Forward flow problem:

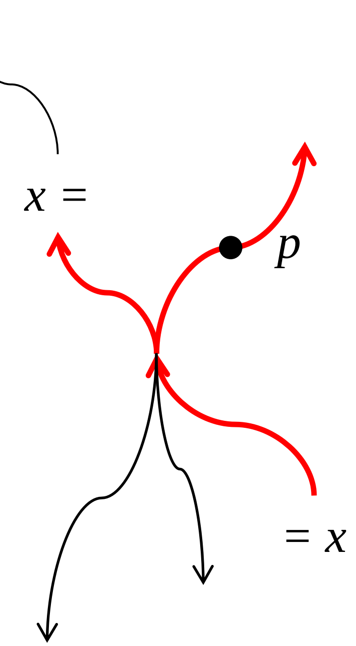
  $\mathrm{OUT}[\mathrm{ENTRY}]$ = boundary condition
  run initialization functions
  do {
    for each node v other than ENTRY {
      $\mathrm{IN}[v] = \sqcap_{p \text{ a predecessor of } v} \mathrm{OUT}[p]$
      $\mathrm{OUT}[v] = f(v)$
    }
  } until no changes to any OUT occur

  Backward flow problem:

  $\mathrm{IN}[\mathrm{EXIT}]$ = boundary condition
  run initialization functions
  do {
    for each node v other than EXIT {
      $\mathrm{OUT}[v] = \sqcap_{s \text{ a successor of } v} \mathrm{IN}[s]$
      $\mathrm{IN}[v] = f(v)$
    }
  } until no changes in any IN occur

Liveness as an Iterative Dataflow Analysis Formulation

- **Liveness is a "backward flow problem"**
  Start at a use of a variable x and look backwards (in terms of the control flow) towards ENTRY and find all points p from which may reach the use of x without a redefinition of x on the way.



Liveness as an Iterative Dataflow Analysis Formulation

- ***gen(v)* / *kill(v)* functions**

  gen(v) = *use(v)* = node *v* uses a variable *x*
  kill(v) = *def(v)* = node *v* defines a variable *x*

- **Transfer function *f(v)***
  f(v) defines what happens when the dataflow crosses node v

  IN[*v*] = *f(v)*, with

$$f(v) = use(v) \cup (\mathrm{OUT}[v] - def(v))$$

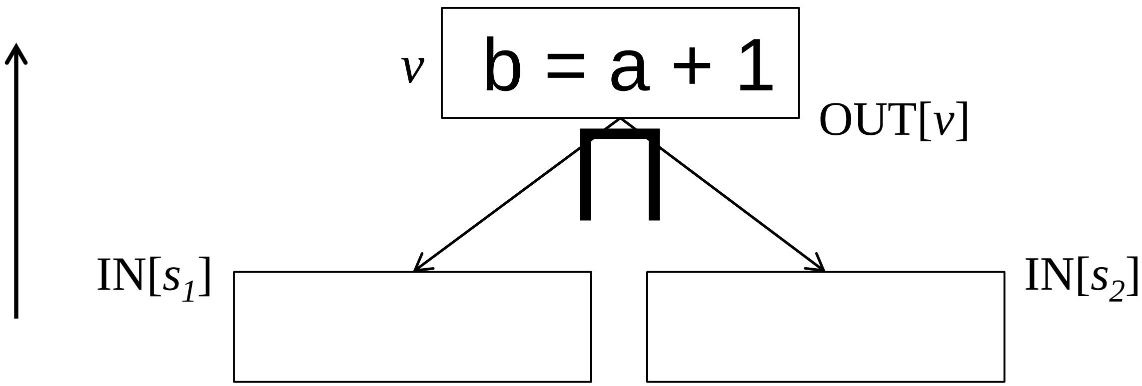
IN[*v*]

$\uparrow$ *v* $\boxed{\text{b = a + 1}}$ *use(v)* = { a }, *def(v)* = { b }

OUT[*v*]

Liveness as an Iterative Dataflow Analysis Formulation

- **Meet operator ⊓**
  ⊓ defines what happens at the entrance to node v ("crossing edges")



$$\mathrm{OUT}[v] = \sqcap_{s \text{ a successor of } v} \mathrm{IN}[s]$$

$$\sqcap = \text{union operator}$$

$$\mathrm{OUT}[v] = \bigcup_{s \text{ a successor of } v} \mathrm{IN}[s]$$

Liveness as an Iterative Dataflow Analysis Formulation

- **Boundary Condition**

  $\mathrm{IN}[\mathrm{EXIT}] = \{\}$

- **Initialization Functions**

  for all nodes v except EXIT do $\mathrm{IN}[v] = \{\}$

Liveness as an Iterative Dataflow Analysis Formulation

- **Iterative Dataflow Analysis to Compute Liveness**

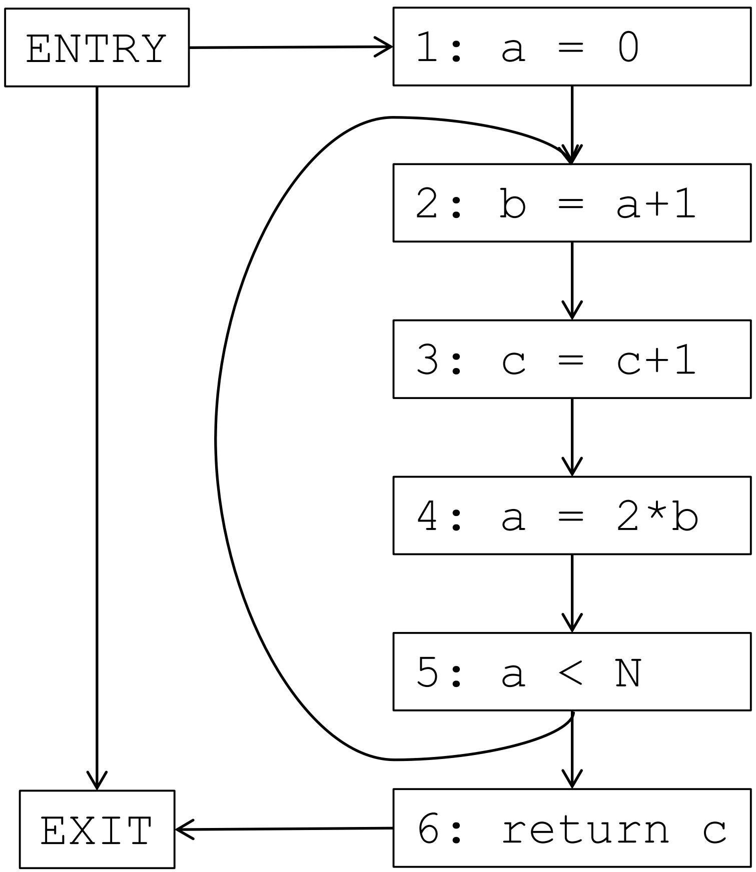
$\mathrm{IN}[\mathrm{EXIT}] = \{\}$
for all nodes v except EXIT do $\mathrm{IN}[v] = \{\}$

do {
  for each node v other than EXIT {
    $\mathrm{OUT}[v] = \cup_{s \text{ a successor of } v}\, \mathrm{IN}[s]$
    $\mathrm{IN}[v] = use(v) \cup (\mathrm{OUT}[v] - def(v))$
  }
} until no changes in any IN occur

Liveness Analysis: Example

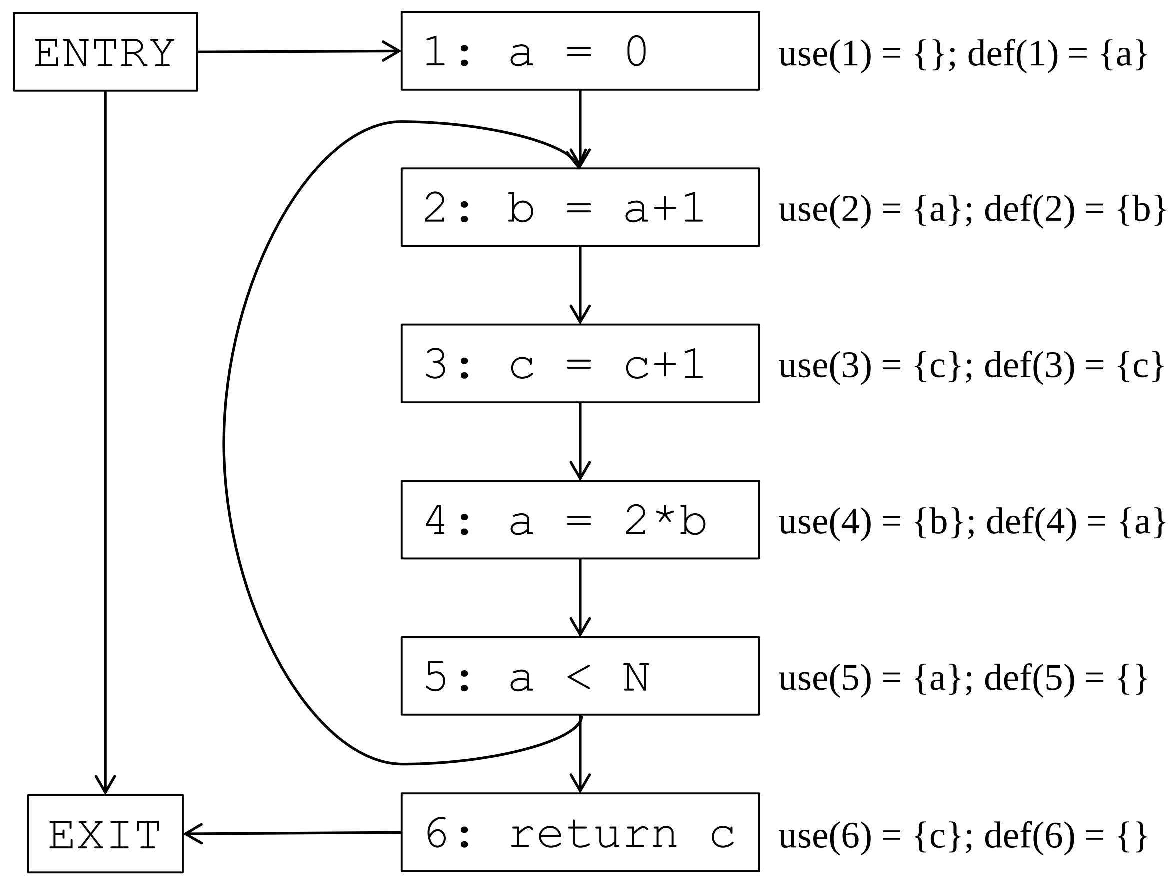
- Build Control Flow Graph

```
a = 0;
do {
   b = a+1;
   c = c+1;
   a = 2*b;
while (a < N);
return c;
```



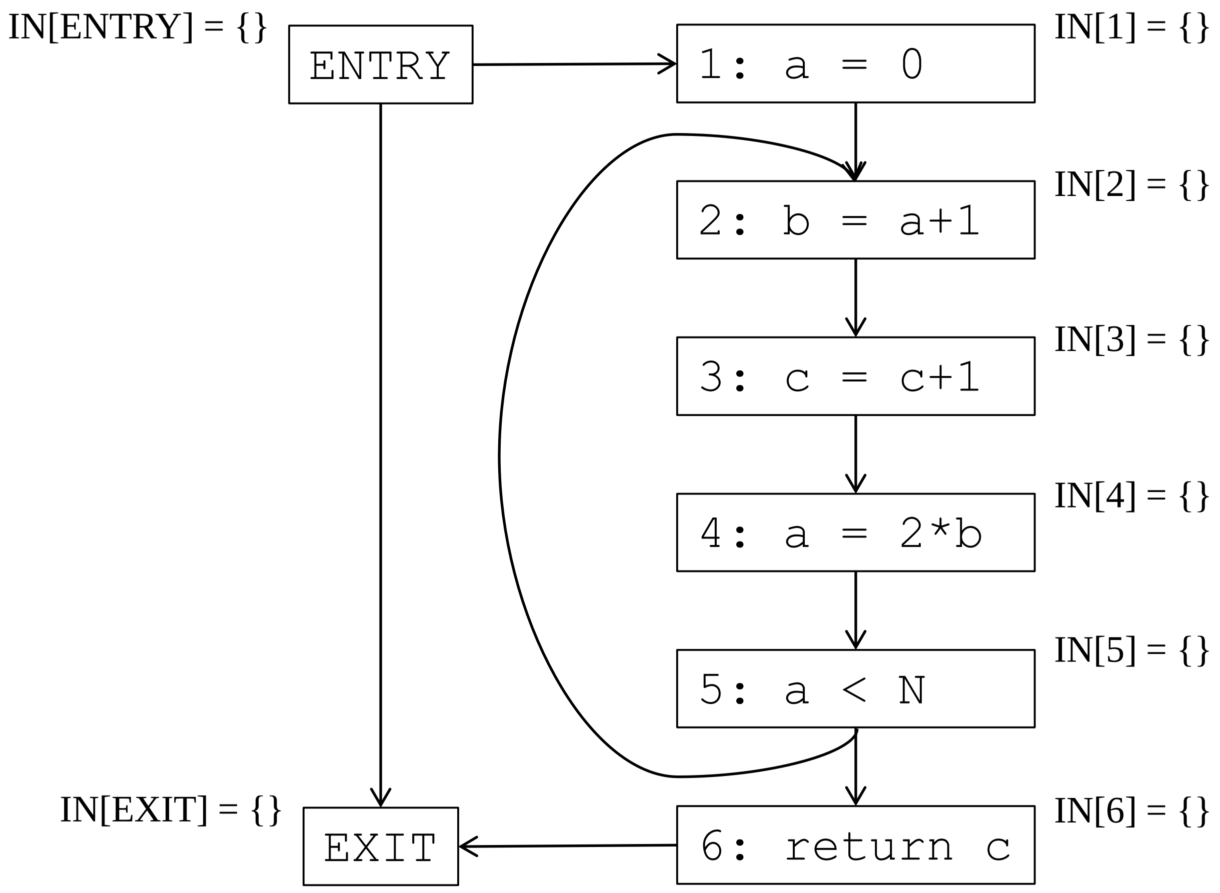
Liveness Analysis: Example

- Compute *use(v)* / *def(v)*



Liveness Analysis: Example

- Init Border Condition and Execute Initialization Functions

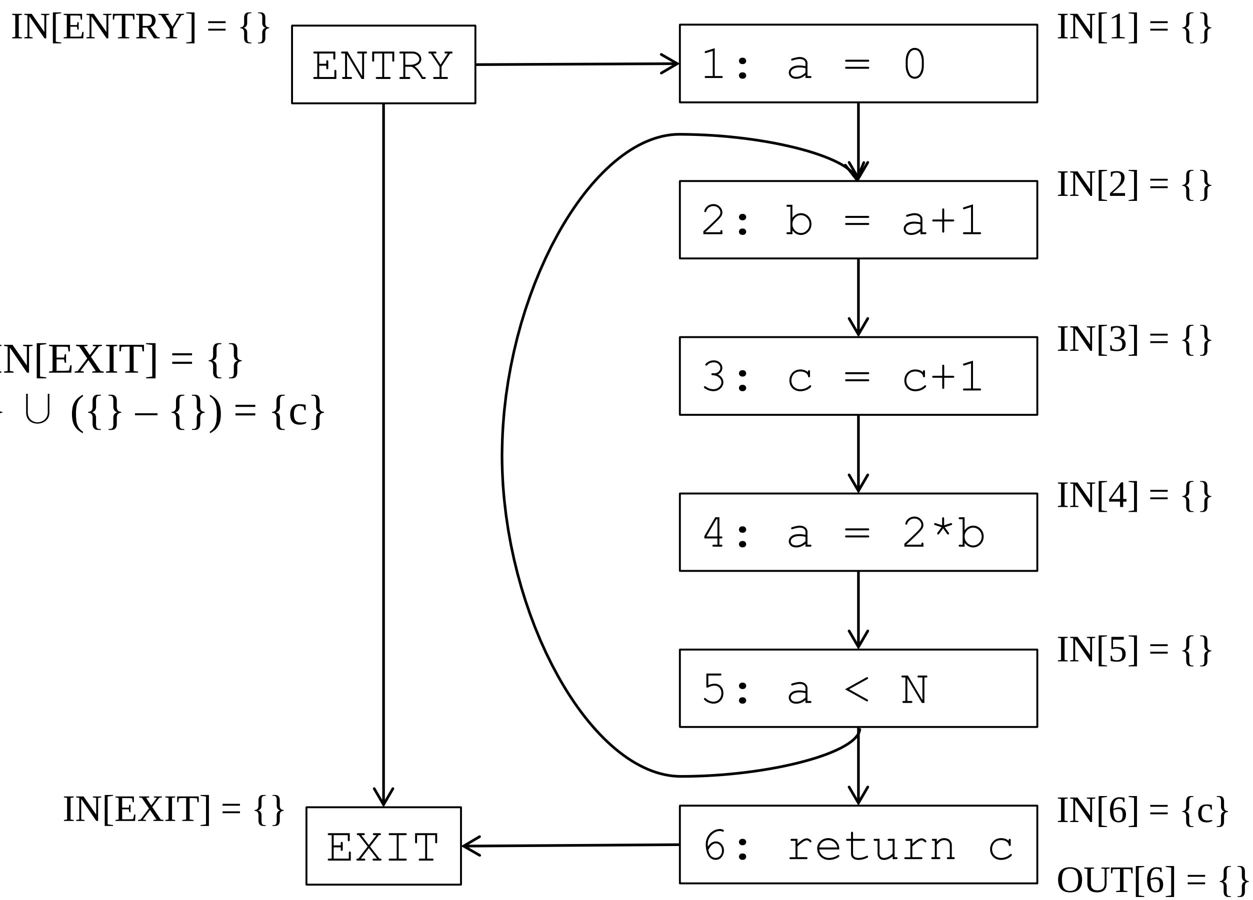


Liveness Analysis: Example

```
do {
  for each node v other than EXIT {
    OUT[v] = ∪ s a successor of v IN[s]
    IN[v] = use(v) ∪ (OUT[v] – def(v))
  }
} until no changes in any IN occur
```

- Iteration 1: let's start with node 6

OUT[6] = IN[EXIT] = {}
IN[6] = {c} ∪ ({} – {}) = {c}



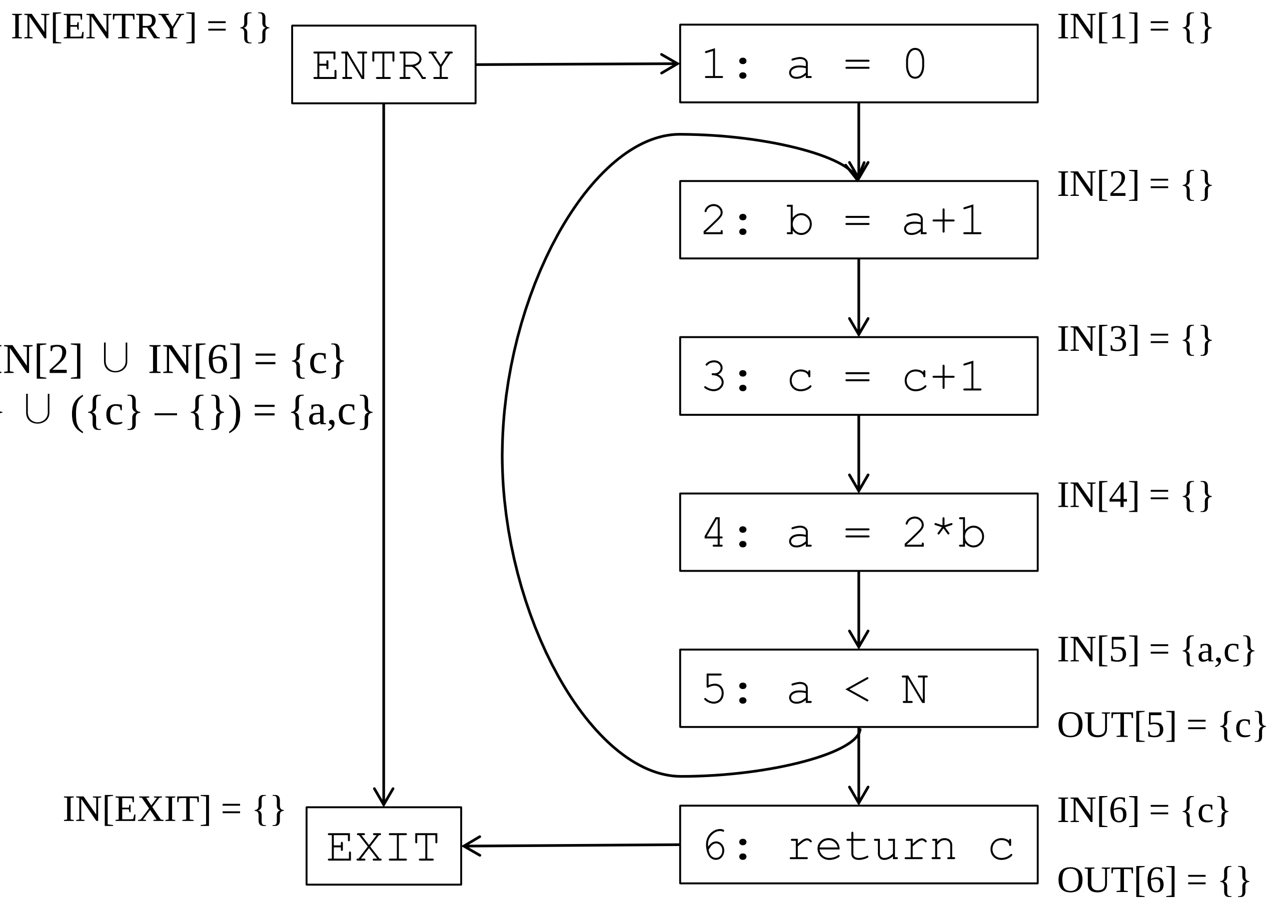
Liveness Analysis: Example

do {
  for each node v other than EXIT {
    $\mathrm{OUT}[v] = \bigcup_{s \text{ a successor of } v} \mathrm{IN}[s]$
    $\mathrm{IN}[v] = use(v) \cup (\mathrm{OUT}[v] - def(v))$
  }
} until no changes in any IN occur

- Iteration 1: node 5

$\mathrm{OUT}[5] = \mathrm{IN}[2] \cup \mathrm{IN}[6] = \{c\}$

$\mathrm{IN}[5] = \{a\} \cup (\{c\} - \{\}) = \{a,c\}$

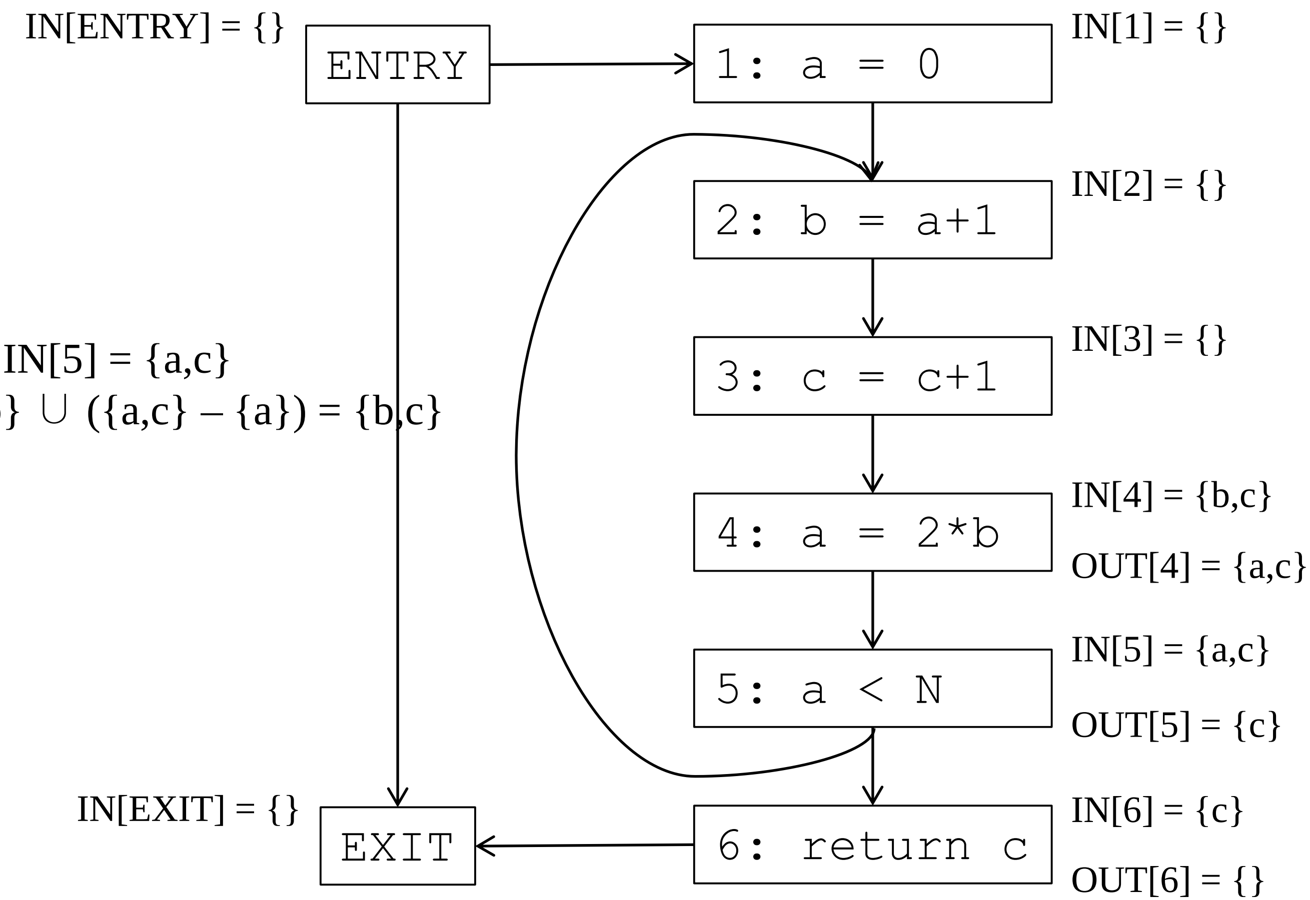


Liveness Analysis: Example

do {
  for each node v other than EXIT {
    $\text{OUT}[v] = \cup_{s \text{ a successor of } v} \text{IN}[s]$
    $\text{IN}[v] = use(v) \cup (\text{OUT}[v] - def(v))$
  }
} until no changes in any IN occur

- Iteration 1: node 4

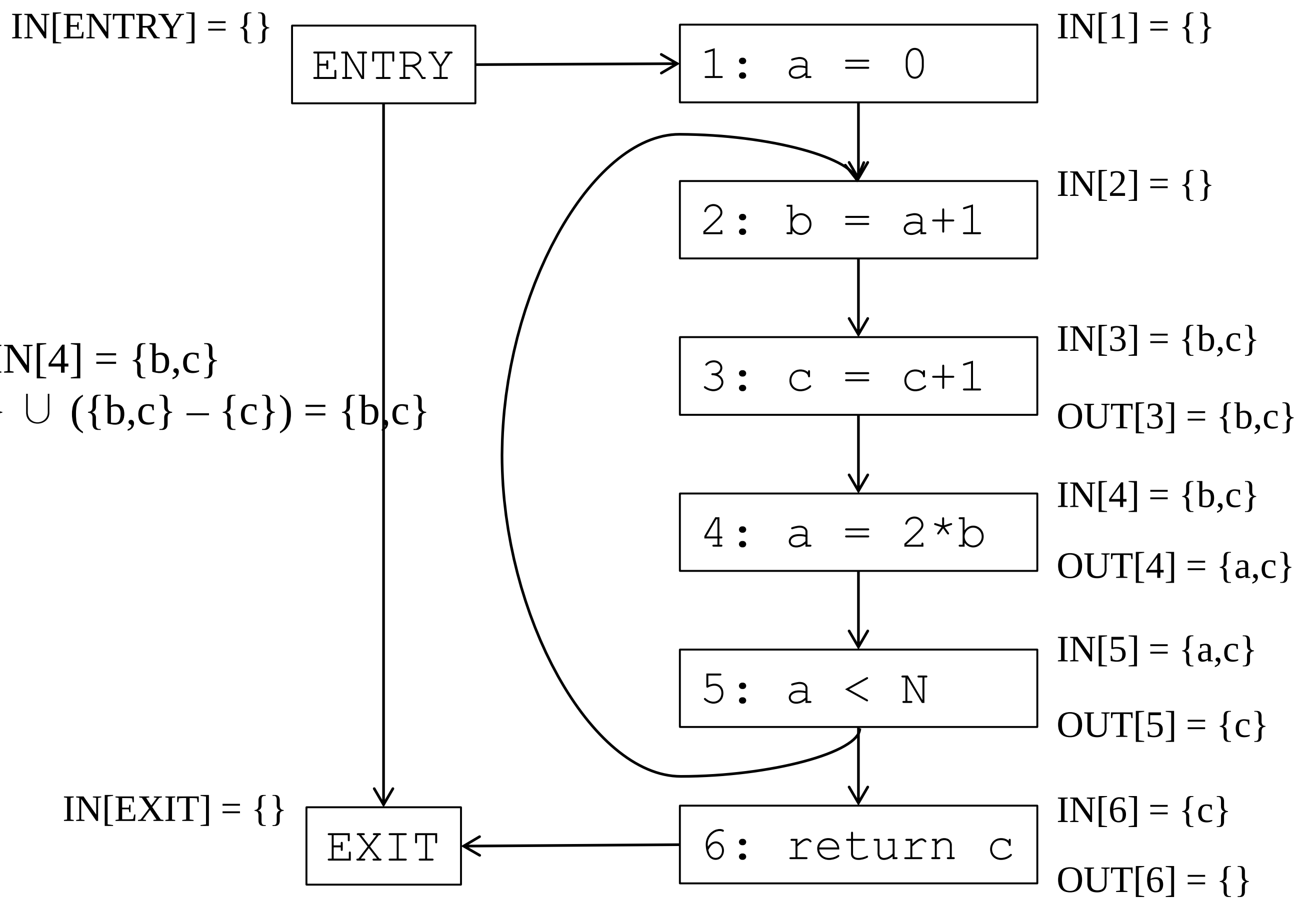
$\text{OUT}[4] = \text{IN}[5] = \{a,c\}$
$\text{IN}[4] = \{b\} \cup (\{a,c\} - \{a\}) = \{b,c\}$



Liveness Analysis: Example

```
do {
  for each node v other than EXIT {
    OUT[v] = ∪ s a successor of v IN[s]
    IN[v] = use(v) ∪ (OUT[v] – def(v))
  }
} until no changes in any IN occur
```

- Iteration 1: node 3

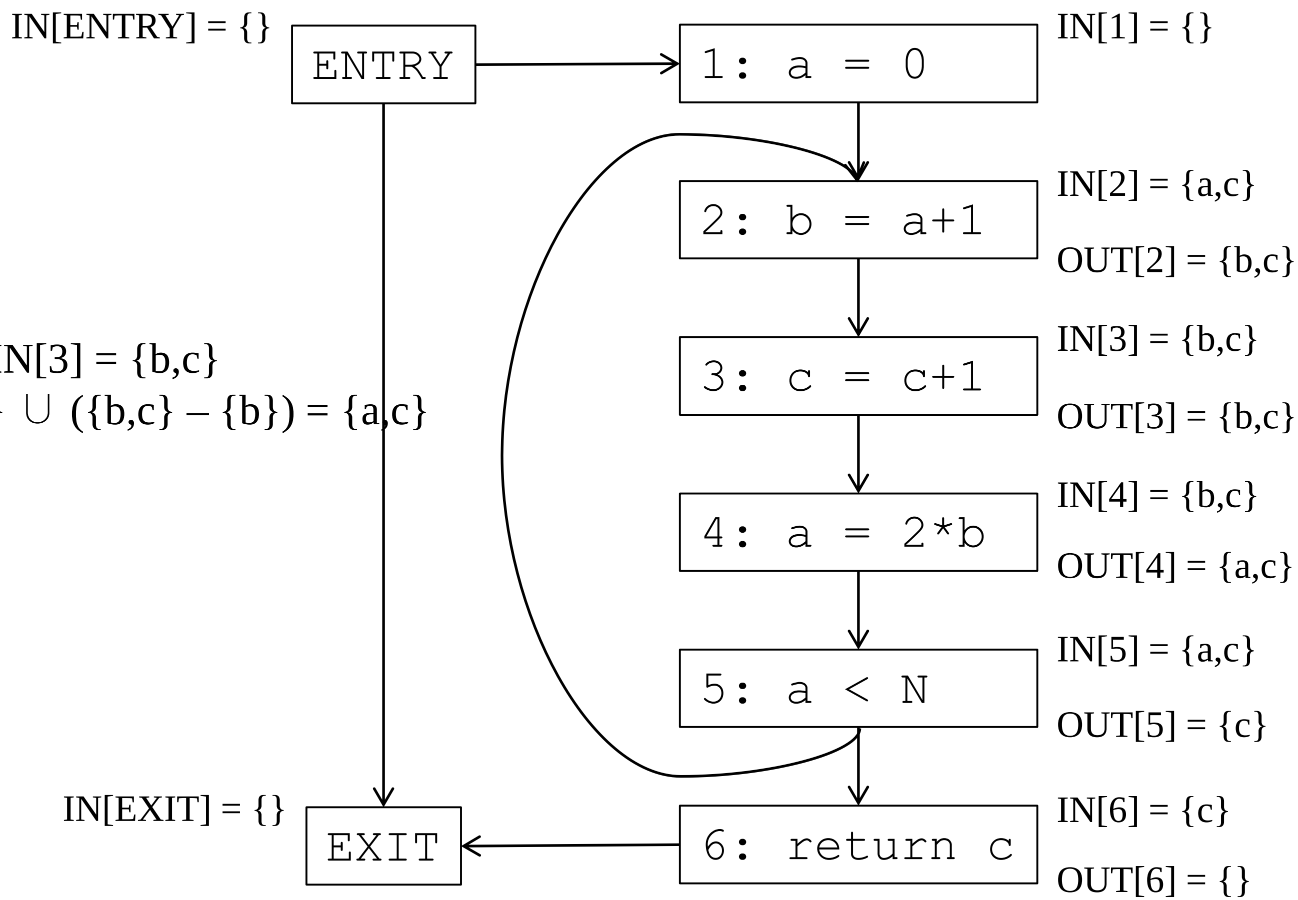


OUT[3] = IN[4] = {b,c}
IN[3] = {c} ∪ ({b,c} – {c}) = {b,c}

Liveness Analysis: Example

do {
  for each node v other than EXIT {
    $OUT[v] = \bigcup_{s \text{ a successor of } v} IN[s]$
    $IN[v] = use(v) \cup (OUT[v] - def(v))$
  }
} until no changes in any IN occur

- Iteration 1: node 2



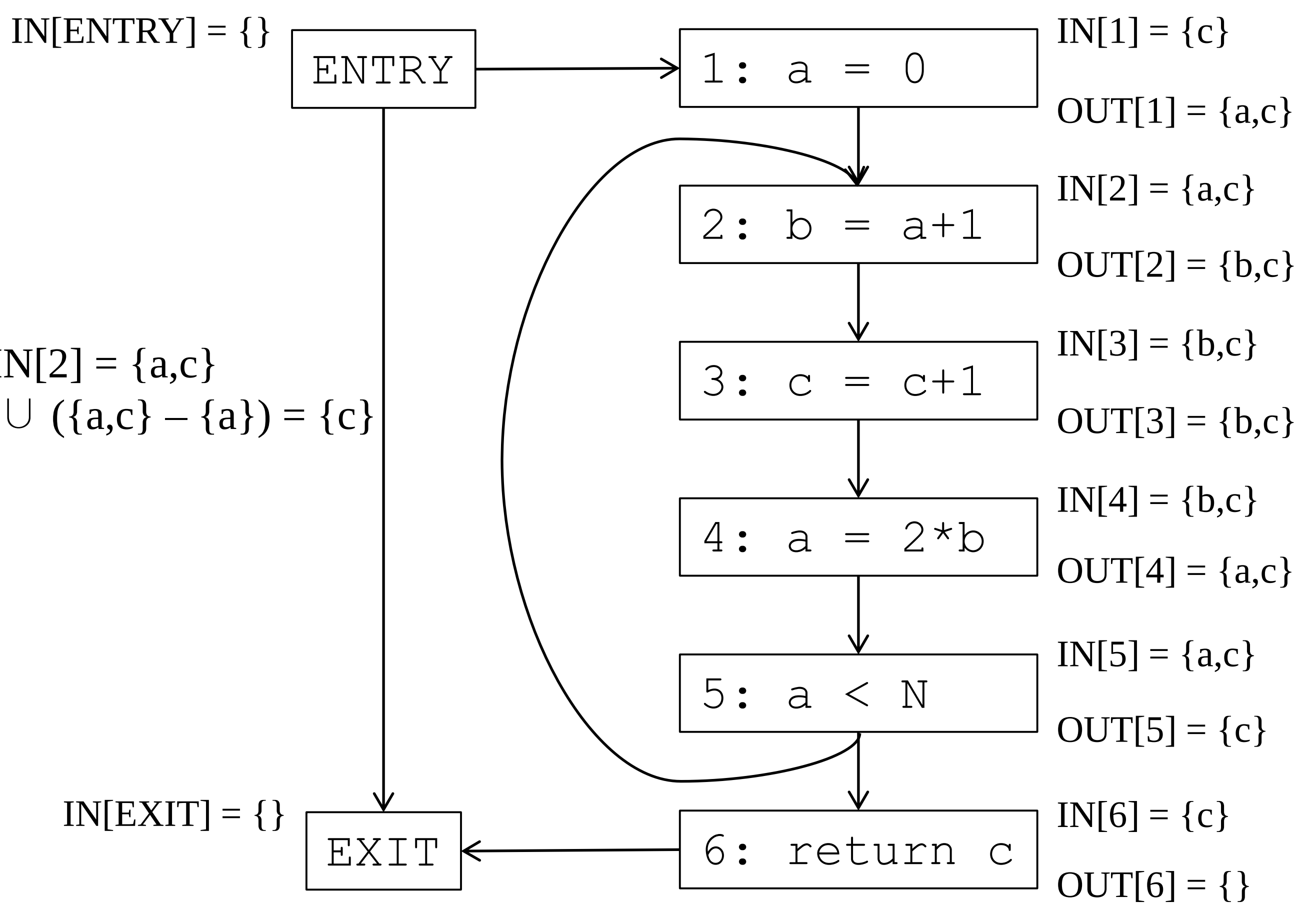
$OUT[2] = IN[3] = \{b,c\}$
$IN[2] = \{a\} \cup (\{b,c\} - \{b\}) = \{a,c\}$

Liveness Analysis: Example

```
do {
  for each node v other than EXIT {
    OUT[v] = ∪ s a successor of v IN[s]
    IN[v] = use(v) ∪ (OUT[v] – def(v))
  }
} until no changes in any IN occur
```

- Iteration 1: node 1

OUT[1] = IN[2] = {a,c}
IN[1] = {} ∪ ({a,c} – {a}) = {c}



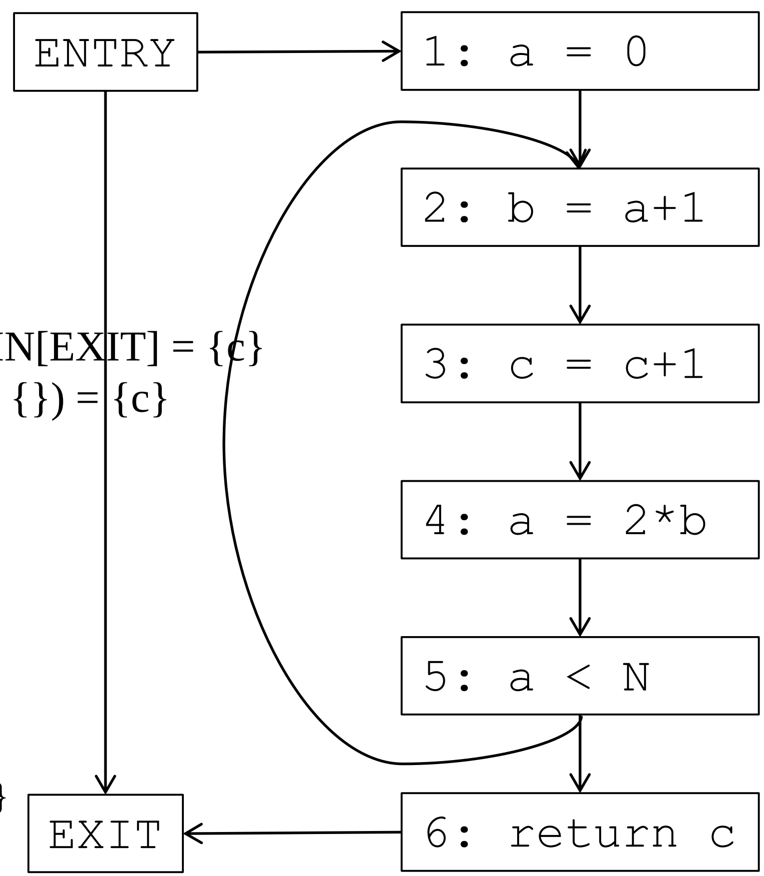
Liveness Analysis: Example

```
do {
  for each node v other than EXIT {
    OUT[v] = ∪_{s a successor of v} IN[s]
    IN[v] = use(v) ∪ (OUT[v] – def(v))
  }
} until no changes in any IN occur
```

- Iteration 1: node ENTRY

IN[ENTRY] = {c}

OUT[ENTRY] = {c}



IN[1] = {c}

OUT[1] = {a,c}

IN[2] = {a,c}

OUT[2] = {b,c}

IN[3] = {b,c}

OUT[3] = {b,c}

IN[4] = {b,c}

OUT[4] = {a,c}

IN[5] = {a,c}

OUT[5] = {c}

IN[6] = {c}

OUT[6] = {}

IN[EXIT] = {}

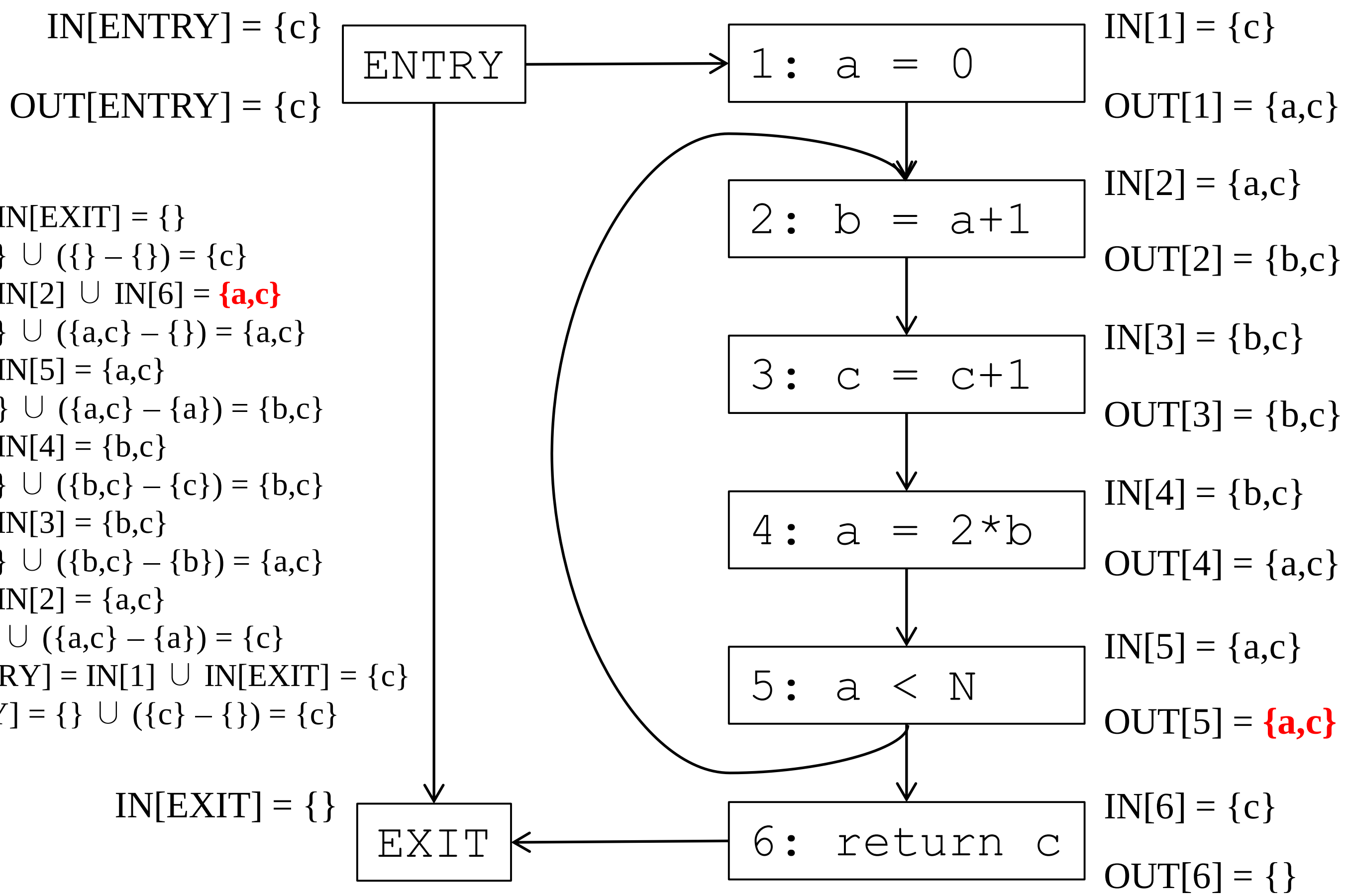
OUT[ENTRY] = IN[1] ∪ IN[EXIT] = {c}

IN[ENTRY] = {} ∪ ({c} – {}) = {c}

Liveness Analysis: Example

```
do {
  for each node v other than EXIT {
    OUT[v] = ∪ s a successor of v IN[s]
    IN[v] = use(v) ∪ (OUT[v] – def(v))
  }
} until no changes in any IN occur
```

- Iteration 2: nodes 6,5,4,3,2,1,ENTRY



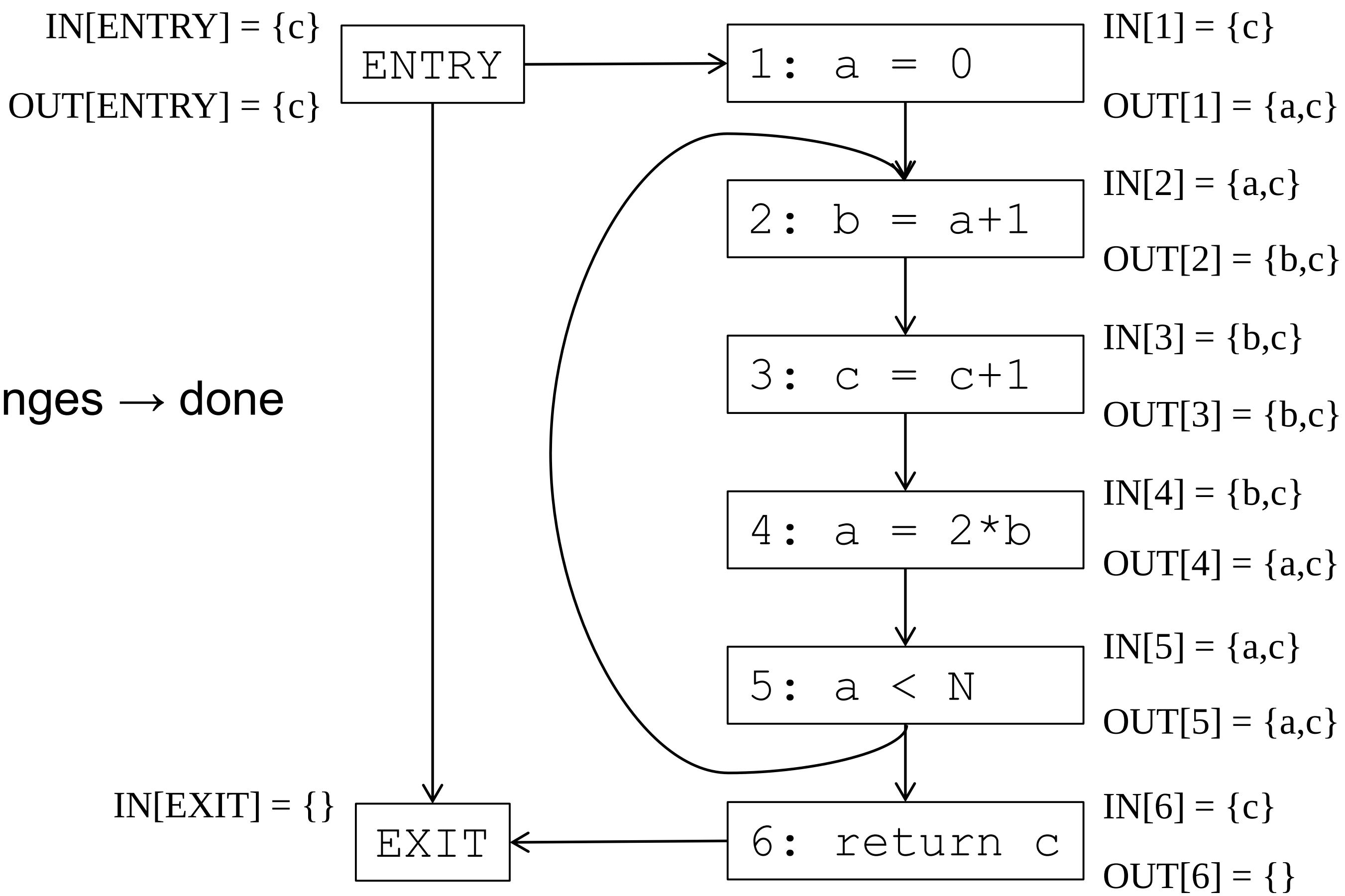
OUT[6] = IN[EXIT] = {}
IN[6] = {c} ∪ ({} – {}) = {c}
OUT[5] = IN[2] ∪ IN[6] = **{a,c}**
IN[5] = {a} ∪ ({a,c} – {}) = {a,c}
OUT[4] = IN[5] = {a,c}
IN[4] = {b} ∪ ({a,c} – {a}) = {b,c}
OUT[3] = IN[4] = {b,c}
IN[3] = {c} ∪ ({b,c} – {c}) = {b,c}
OUT[2] = IN[3] = {b,c}
IN[2] = {a} ∪ ({b,c} – {b}) = {a,c}
OUT[1] = IN[2] = {a,c}
IN[1] = {} ∪ ({a,c} – {a}) = {c}
OUT[ENTRY] = IN[1] ∪ IN[EXIT] = {c}
IN[ENTRY] = {} ∪ ({c} – {}) = {c}

Liveness Analysis: Example

do {
  for each node v other than EXIT {
    $\mathrm{OUT}[v] = \bigcup_{s \text{ a successor of } v} \mathrm{IN}[s]$
    $\mathrm{IN}[v] = use(v) \cup (\mathrm{OUT}[v] - def(v))$
  }
} until no changes in any IN occur

- Iteration 3: nodes 6,5,4,3,2,1,ENTRY

no changes → done



Iterative Dataflow Analysis Considerations

- **Termination**
  does the iterative liveness analysis algorithm always terminate?

  do {
    for each node v other than EXIT {
      $\text{OUT}[v] = \cup_{s \text{ a successor of } v} \text{IN}[s]$
      $\text{IN}[v] = use(v) \cup (\text{OUT}[v] - def(v))$
    }
  } until no changes in any IN occur

- **Speed of Convergence**
  what order of nodes provides the fastest convergence speed?

- **Node Size**
  single statements vs. basic blocks

Useful Iterative Dataflow Analyses: Overview

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Sets of variables	Sets of expressions
Direction	forwards	backwards	forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\mathrm{OUT}[\mathrm{ENTRY}] = \emptyset$	$\mathrm{IN}[\mathrm{EXIT}] = \emptyset$	$\mathrm{OUT}[\mathrm{ENTRY}] = \emptyset$
Initialize	$\mathrm{OUT}[B] = \emptyset$	$\mathrm{IN}[B] = \emptyset$	$\mathrm{OUT}[B] = \top$
Meet	$\cup$	$\cup$	$\cap$
Equations	$\mathrm{IN}[B] = \cup_{P \in \mathrm{Pred}(B)} \mathrm{OUT}[P]$ $\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$	$\mathrm{OUT}[B] = \cup_{S \in \mathrm{Succ}(B)} \mathrm{IN}[S]$ $\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$	$\mathrm{IN}[B] = \cup_{P \in \mathrm{Pred}\,(B)} \mathrm{OUT}\,[P]$ $\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$

Building the Interference Graph

- **Interference Graph IG = (V, E)**

  Given:

  - $def(s)$, $\mathrm{OUT}[s]$ for all statements s (containing a definition) obtained through liveness analysis
  - all machine registers M

  Construction:

  - $\mathrm{V} = \{$ one node for each variable v and machine register m $\}$
  - $\mathrm{E} = \{ x \leftrightarrow y \mid x \in \{ def(s) \}, y \in \mathrm{OUT}[s], x \neq y \parallel x \in M, \mathrm{y} \in M, x \neq y \}$

Interference Graph: Example

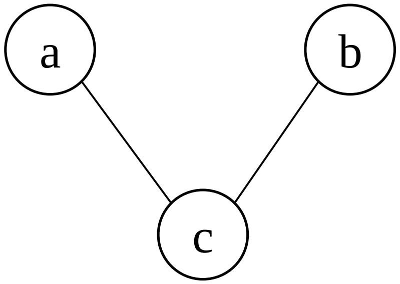
- **Interference Graph IG = (V, E)**

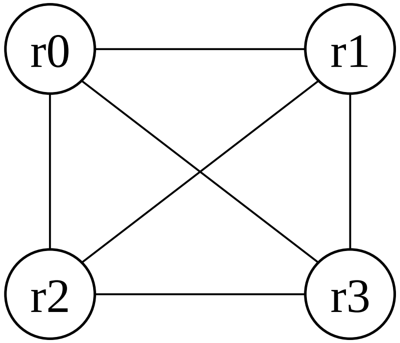
```
1: a = 0
```
def(1) = {a}
OUT[1] = {a,c}

```
2: b = a+1
```
def(2) = {b}
OUT[2] = {b,c}

```
3: c = c+1
```
def(3) = {c}
OUT[3] = {b,c}

```
4: a = 2*b
```
def(4) = {a}
OUT[4] = {a,c}





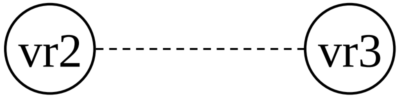
Treatment of MOVE Instructions

- **Special Treatment of MOVE Instructions to Simplify Later Coalescing**

$$
\begin{array}{l}
vr3 \leftarrow vr2 \\
\quad \ldots \\
\quad \leftarrow \ldots vr2 \ldots \\
\quad \ldots \\
\quad \leftarrow \ldots vr3 \ldots
\end{array}
$$

In principle, vr2 and vr3 are both live and interfere. Do we need to use separate registers?

No: both vr2 and vr3 contain the same value, no need to add an interference edge. However, register allocation will need to know which nodes are copies of each other, therefore we introduce special ‘move edges’:



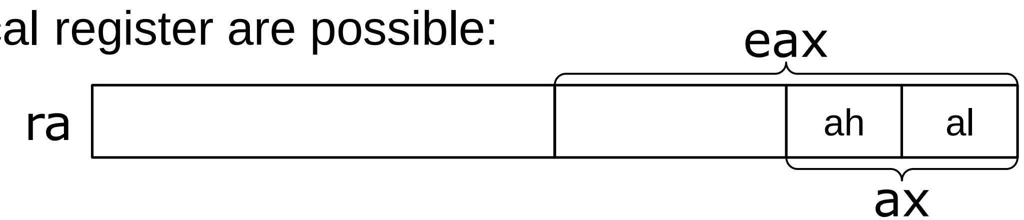
Interference Graph: additional Considerations

- **Interference with Zero-length live ranges**
  A value defined at statement *s* that is not live after its definition has a zero-length live range. We have to add interference edges to the IG.

- **Interference with Machine Registers**
  Some instructions cannot write the generated value to certain machine registers; in that case we also add interference edges to these registers.
  Example: floating point registers in x86 cannot be used by integer operations.

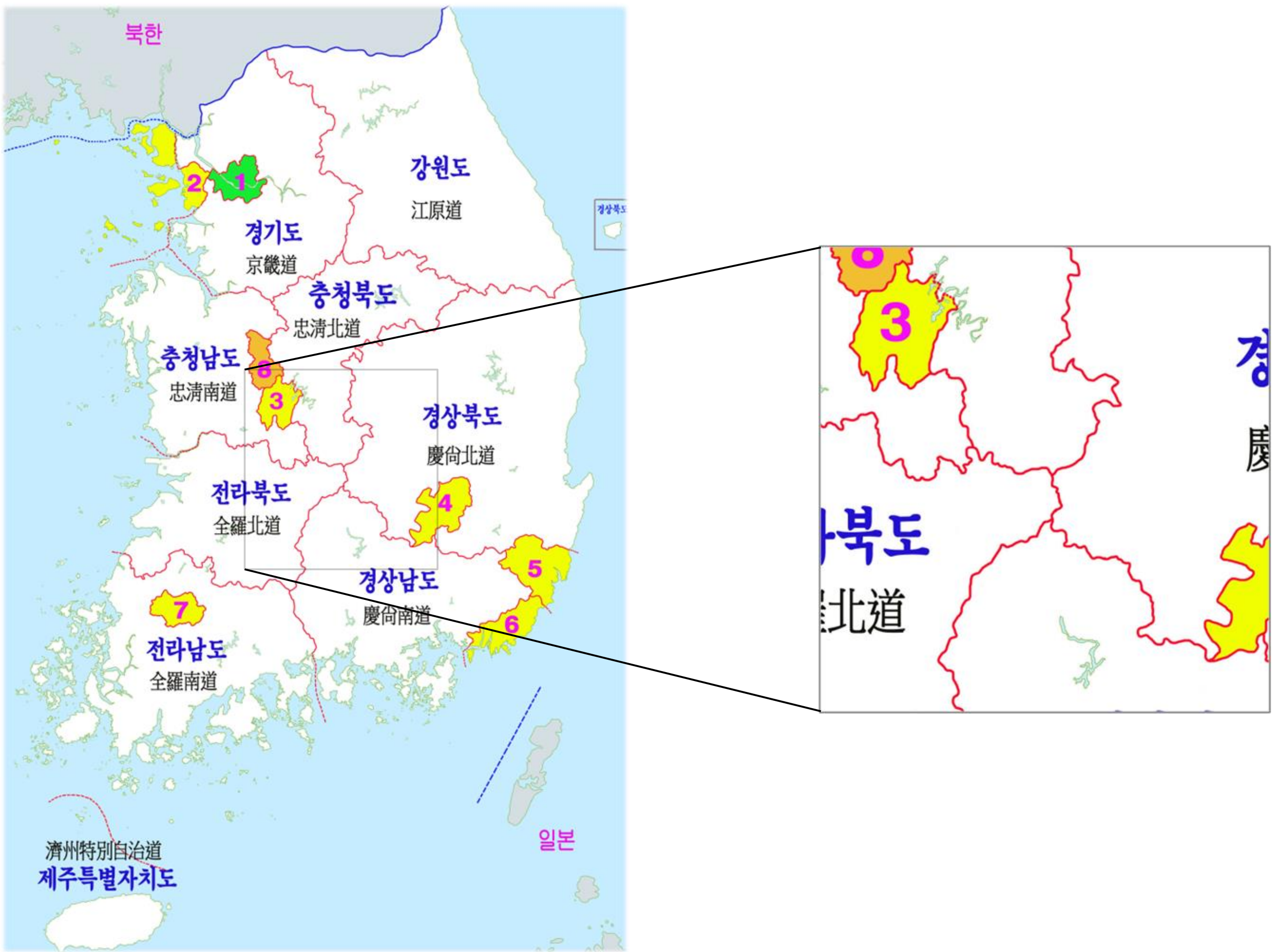
  (Note: the opposite case – requiring a specific machine register – is handled later in the register allocator)

- **x86 Machine Registers**
  The IG for x86 architectures must reflect that accesses to different parts of the same physical register are possible:



Graph Coloring

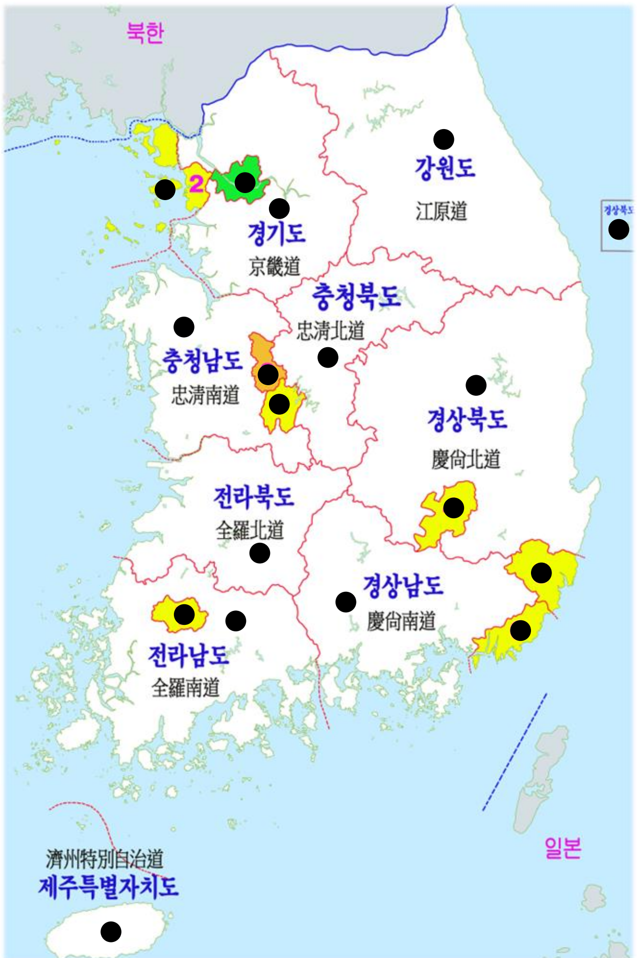
- Coloring a map with as few colors as possible in such a way that no two areas that share a common border have the same color

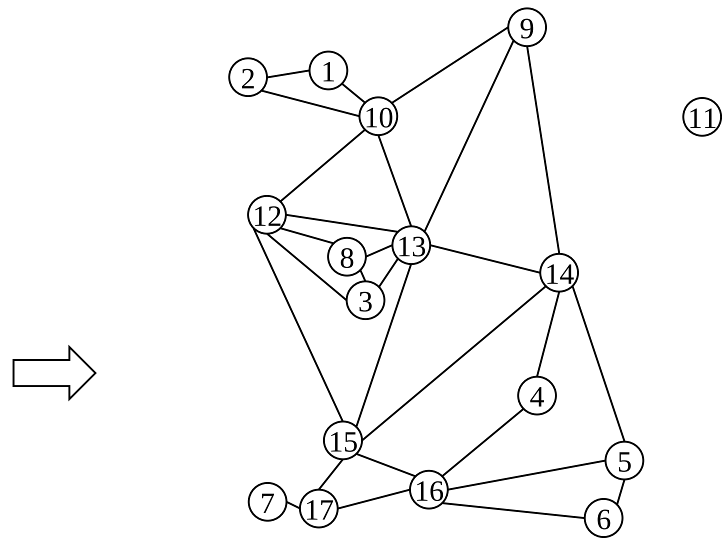


CSE 컴퓨터공학부
Department of Computer Science & Engineering

Graph Coloring

- Coloring a map with as few colors as possible in such a way that no two areas that share a common border have the same color

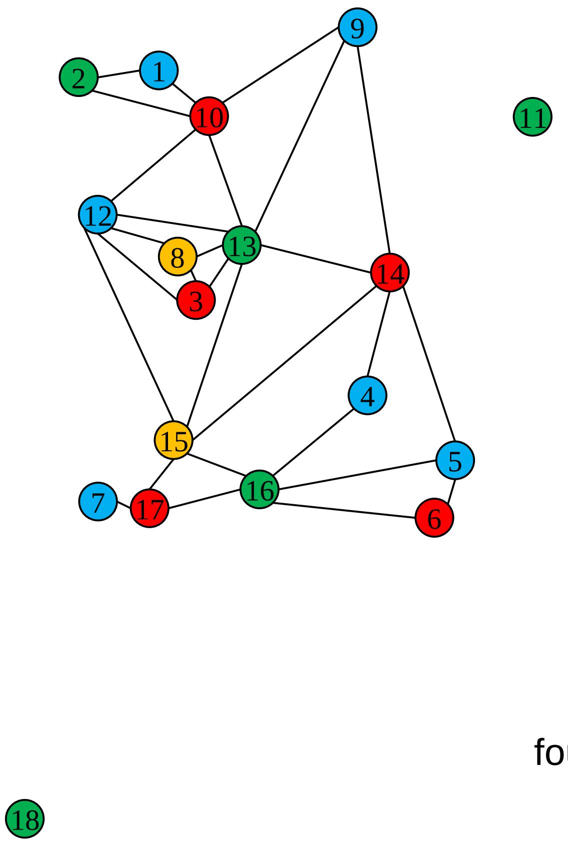




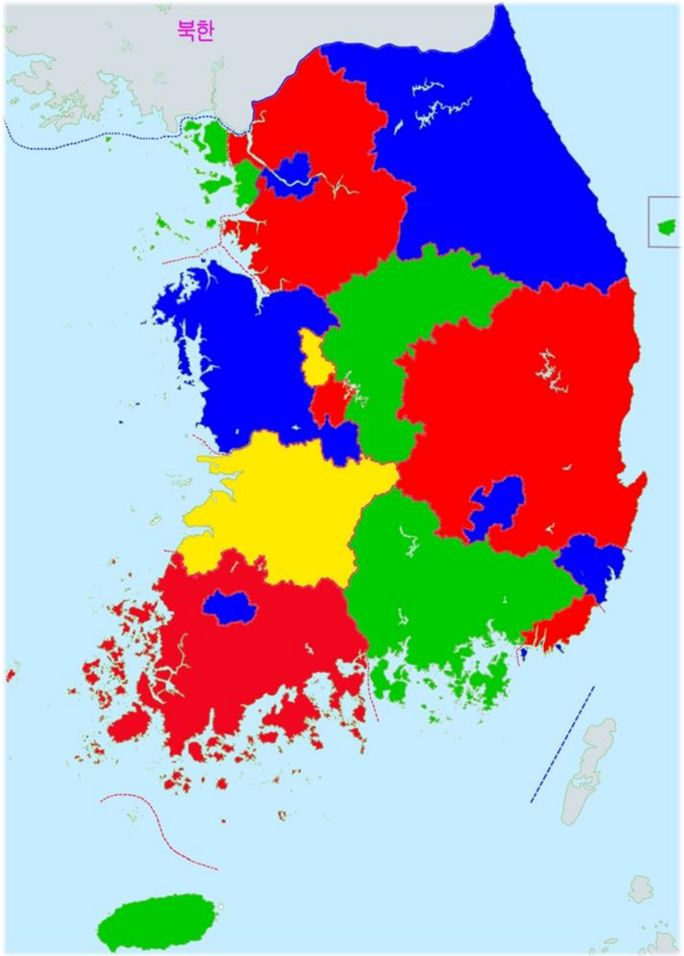
how many colors do we need?

Graph Coloring

- Coloring a map with as few colors as possible in such a way that no two areas that share a common border have the same color



four colors!



Register Allocation via Graph Coloring

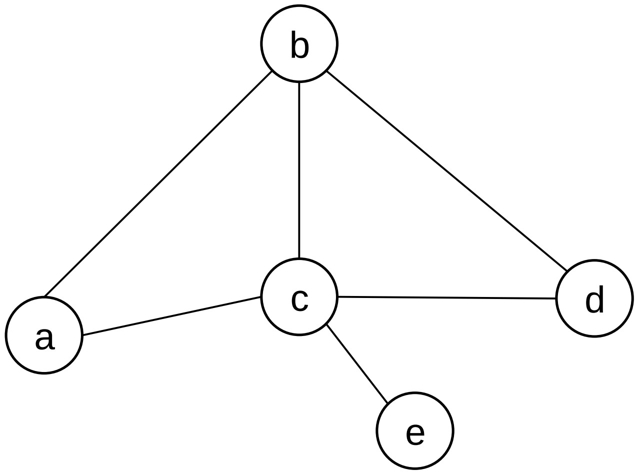
- **Main Observation**
  (~1970s by Cocke, Yershov, Schwartz, first workable implementation published by Chaitin in 1981)

  Coloring the interference graph in such a way that no two interfering nodes are assigned the same color with no more than k colors corresponds to a valid register allocation for a machine with k registers.

- Ok, so how do we color the interference graph?

Nomenclature

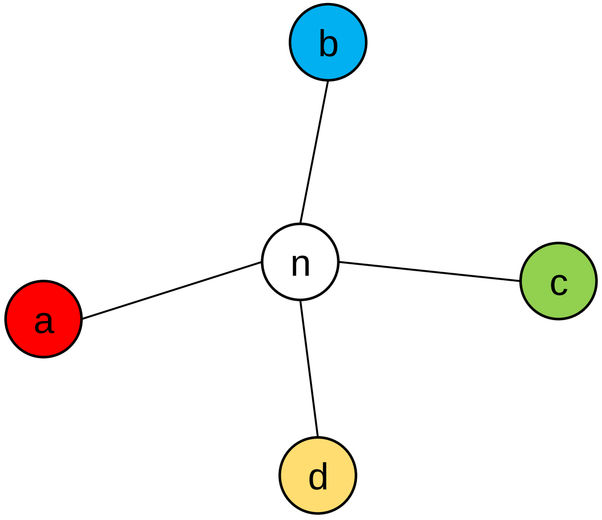
- Given k (the number of hardware registers) we classify nodes in the interference graph into
  - nodes of **significant degree**
    iff the node has more or equal to k neighbors
  - nodes of **insignificant degree**
    iff the node has less than k neighbors
  - Example for k = 3:
    nodes of significant degree = { b, c }
    nodes of insignificant degree = { a, d, e }



Observation

- **A node *n* of insignificant degree can always be assigned a color**
  The node has, by definition, less than *k* neighbors. Even in the worst case if all <*k* neighbors are colored with a different color there is a unused color left for node *n*.

k = 5

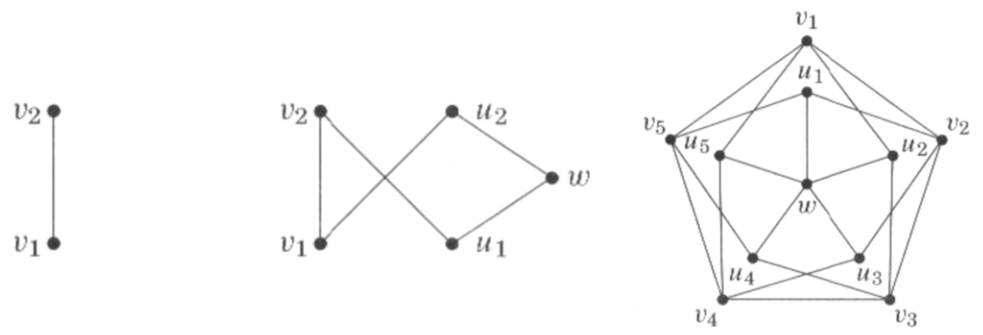


A Tiny Bit of Theory And Some Bad News

- the **chromatic number** $\chi(G)$ is the smallest number of colors needed to color the vertices of G so that no two connected vertices share the same color
- i.e., the smallest value k to obtain a k-coloring is denoted $\chi(G)$
- the **clique number** $\omega(G)$ is the number of vertices in a maximum clique of G
- **Proposition:** $\chi(G) \geq \omega(G)$
- finding the clique number for a given graph G is an NP-complete problem
- hence the bad news:
  register allocation via graph coloring is an NP-complete problem

A Tiny Bit of Theory And Some Bad News

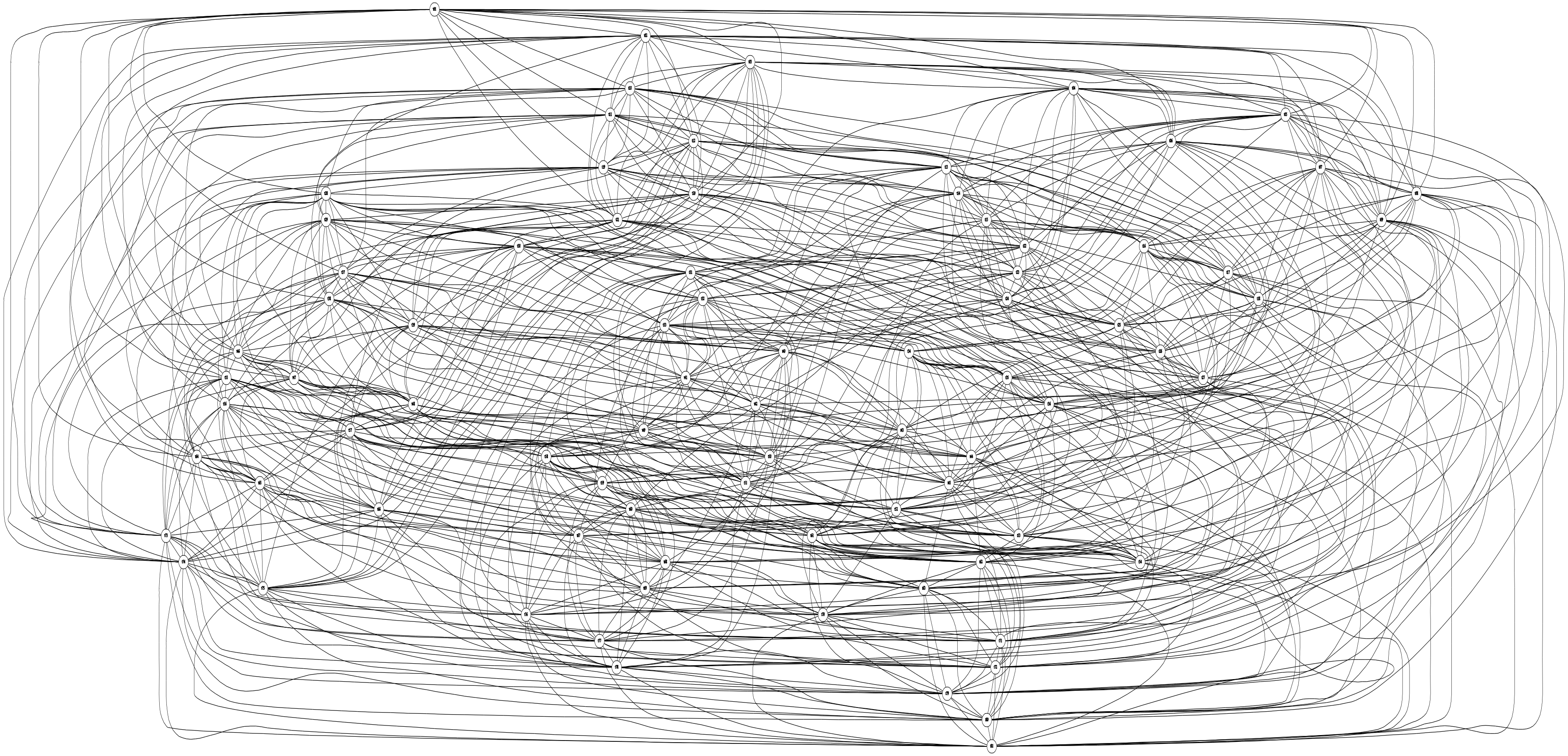
- Is $\chi(G) \geq \omega(G)$ a tight or a loose bound?
  - for most practical interference graphs $\chi(G) \geq \omega(G)$ is a tight bound
  - in theory, the bound can be arbitrarily large
    - example:
      Mycielski construction produces triangle free graphs ($\omega(G) = 2$) with an arbitrarily large $\chi(G)$



$\chi(G) = 2$ $\chi(G) = 3$ $\chi(G) = 4$

Before We Begin

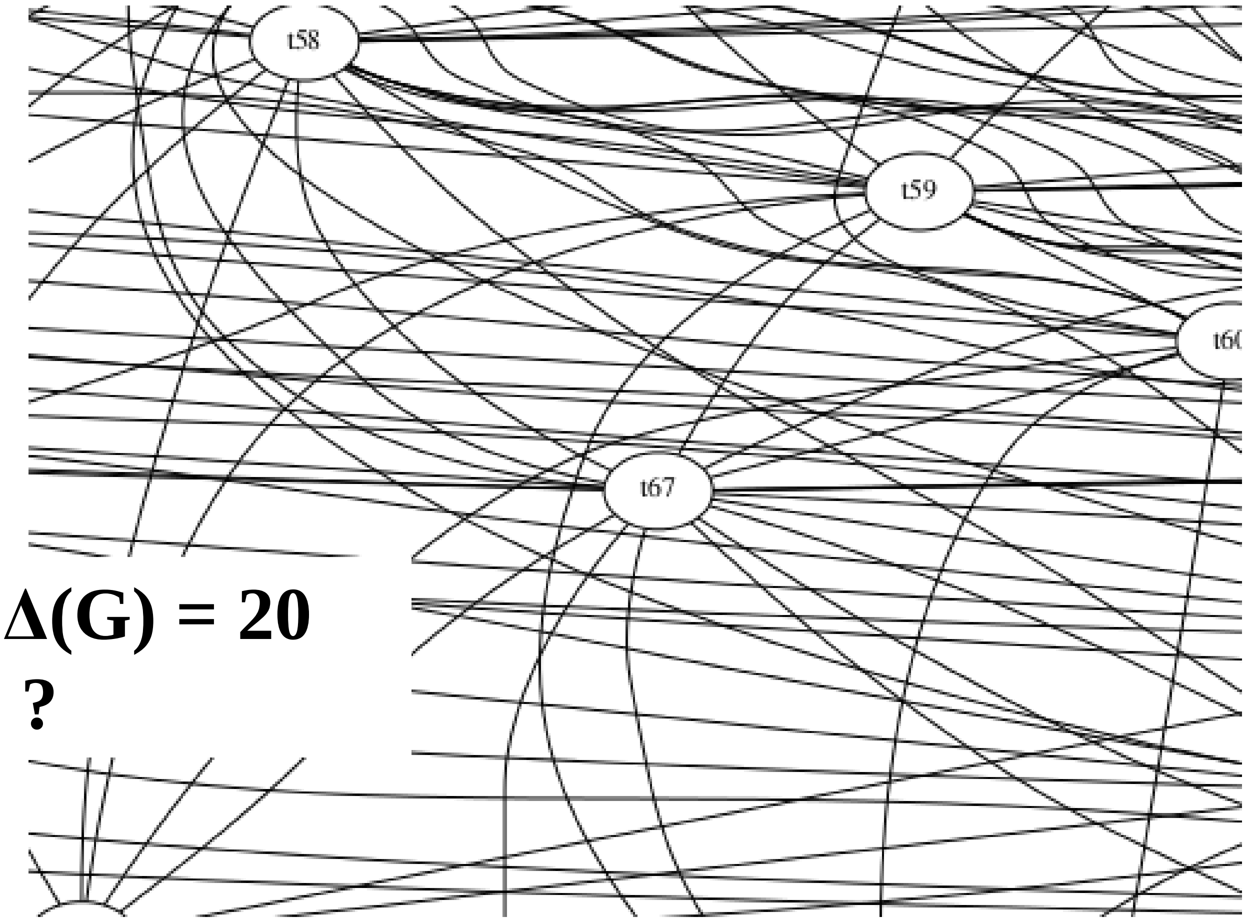
- There are lots of people who manually try to solve register allocation for rather crazy interference graphs:



Before We Begin

- There are lots of people who manually try to solve register allocation problems for crazy interference graphs:

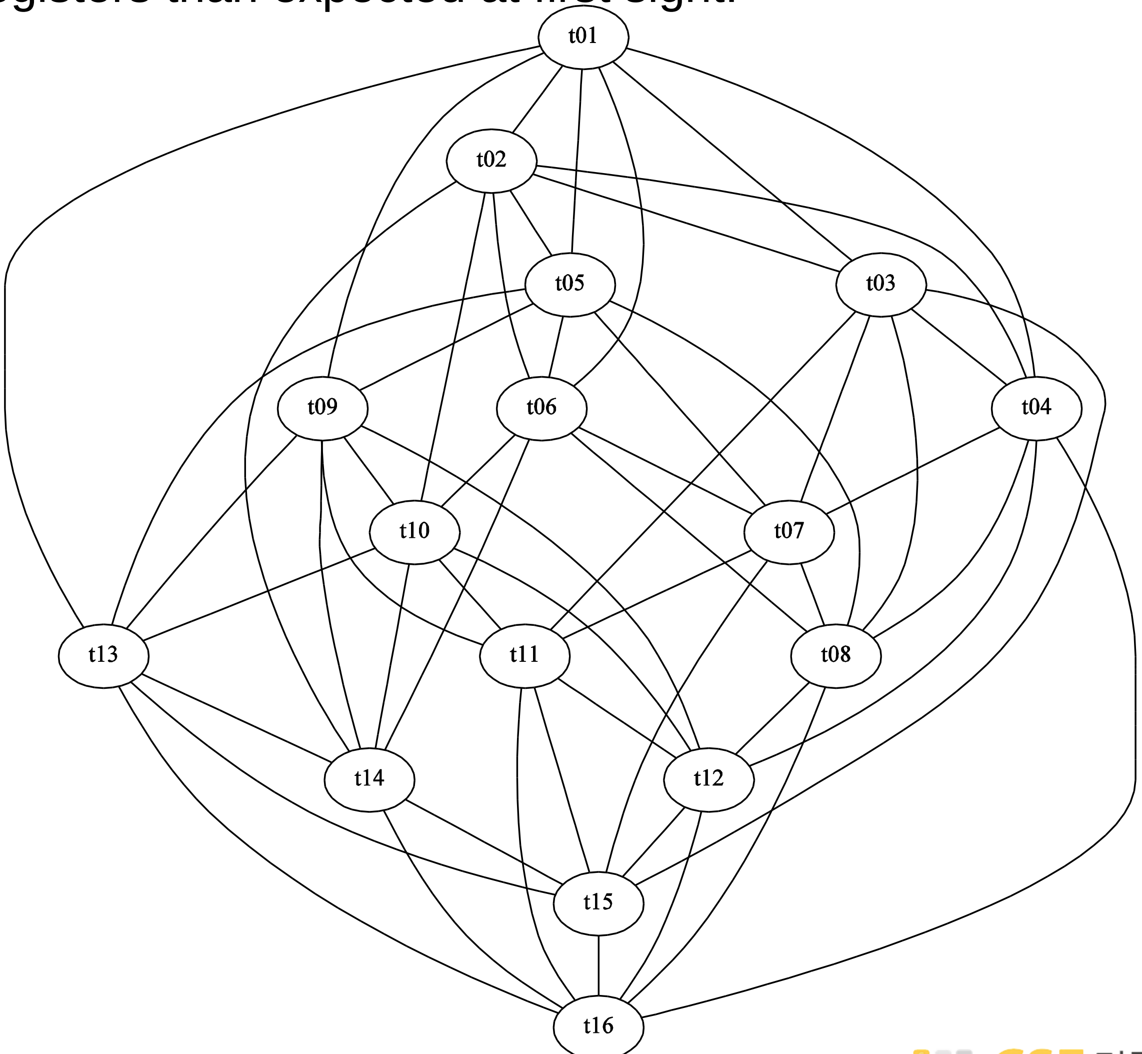
Zoom-in:



$$\delta(G) = \Delta(G) = 20$$
$$\chi(G) = ?$$

Before We Begin

- We may need fewer registers than expected at first sight:



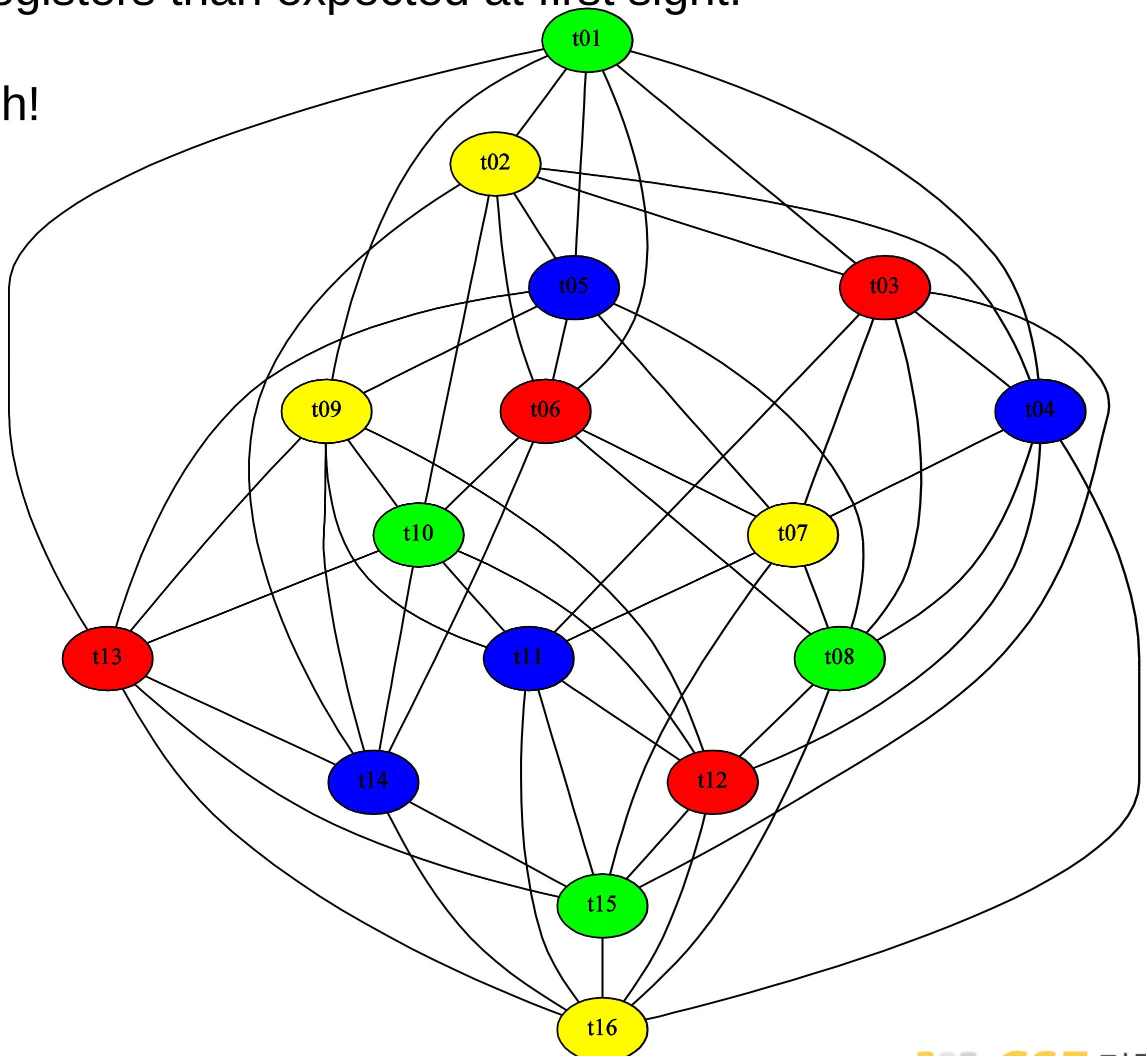
$\delta(G) = \Delta(G) = 7$

$\chi(G) = ?$

Before We Begin

- We may need fewer registers than expected at first sight:

  Four colors are enough!



$$\delta(G) = \Delta(G) = 7$$

$$\chi(G) = ?$$

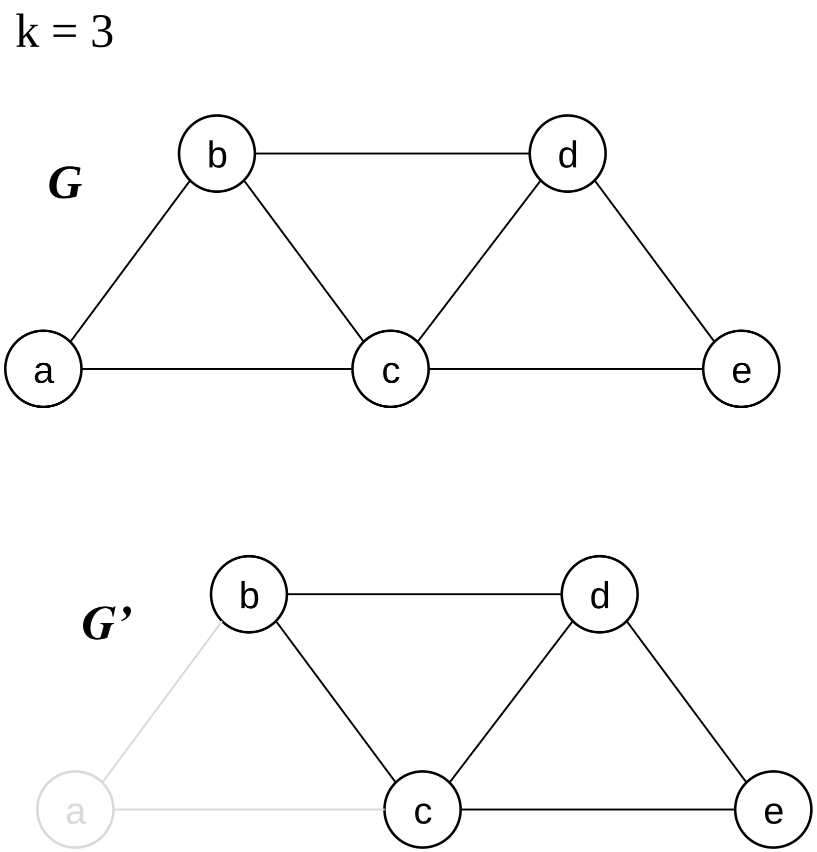
A Simple Heuristic to the Rescue

- **Graph Coloring Heuristic**

  Assume graph G contains a node n of insignificant degree. Then, any valid coloring obtained for $G' = G - \{n\}$ leads to a valid coloring for G because there is always a free color available for n with $<k$ neighbors.

  Repeating this process for G' leads to G'' if there is at least one node of insignificant degree, and so on, until all nodes have been removed.

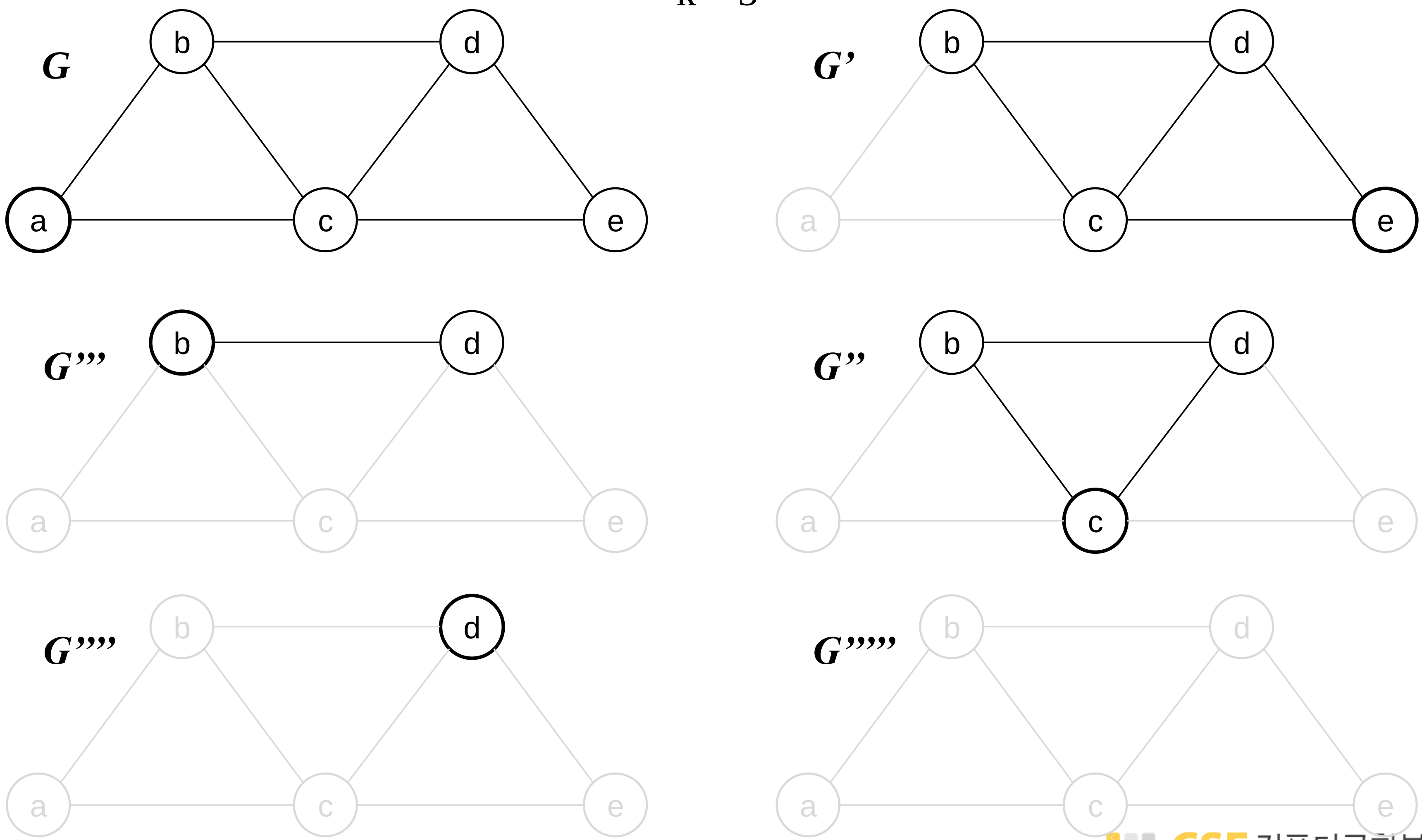
  Colors are then assigned in reverse order of removal.



A Simple Heuristic to the Rescue

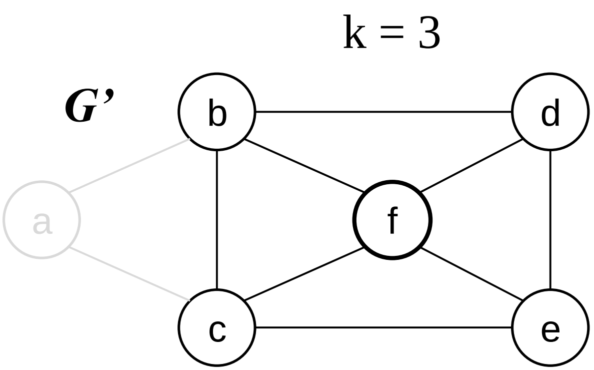
- **Graph Coloring Heuristic**

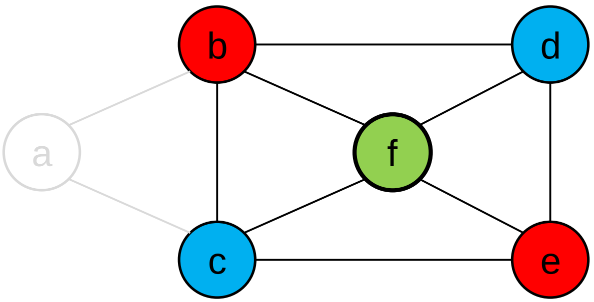
k = 3

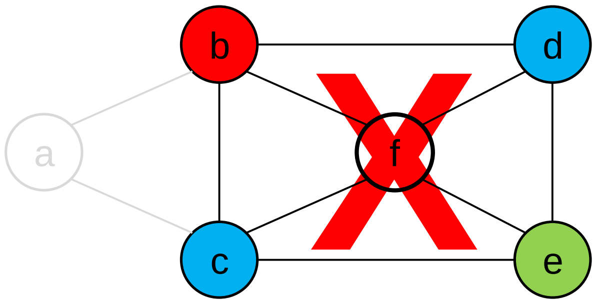


Problems to Solve

- **What if the graph only contains nodes of significant degree?**
  - pick “any” node, remove it from the graph and mark it as a *potential spill*, and continue with removing nodes
  - when assigning colors in reverse order, a color *may* be available for the potential spill (aka “optimistic coloring” by Briggs et al.)

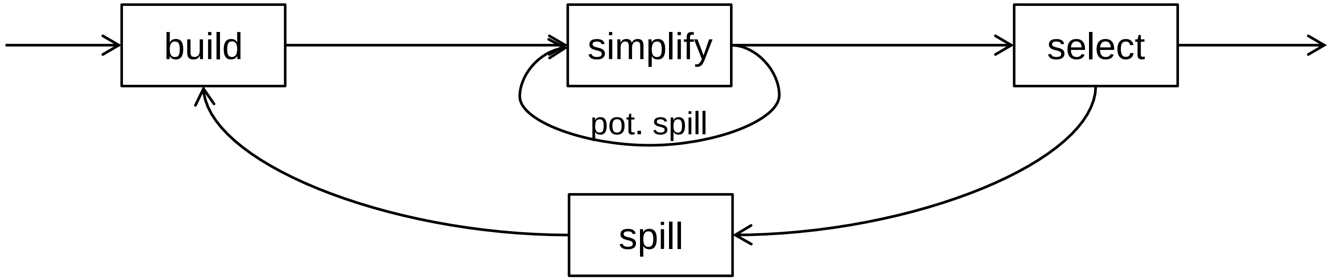






RA - Optimistic Coloring

- **Optimistic Coloring**



- **build**: construct interference graph
  based on liveness analysis (traditional / SSA-based)
- **simplify**: simply interference graph by pushing nodes with fewer than k neighbors, one by one.
- **potential spill**: if simplify fails to push all nodes, mark one of the remaining nodes for spilling and push it. Go back to simplify until all nodes have been removed.
- **select**: pop nodes and assign a color. For potential spill nodes, if there is a free color, assign it. Otherwise, mark the node as an actual spill.
- **spill**: as long as actual spill nodes are present, spill and repeat

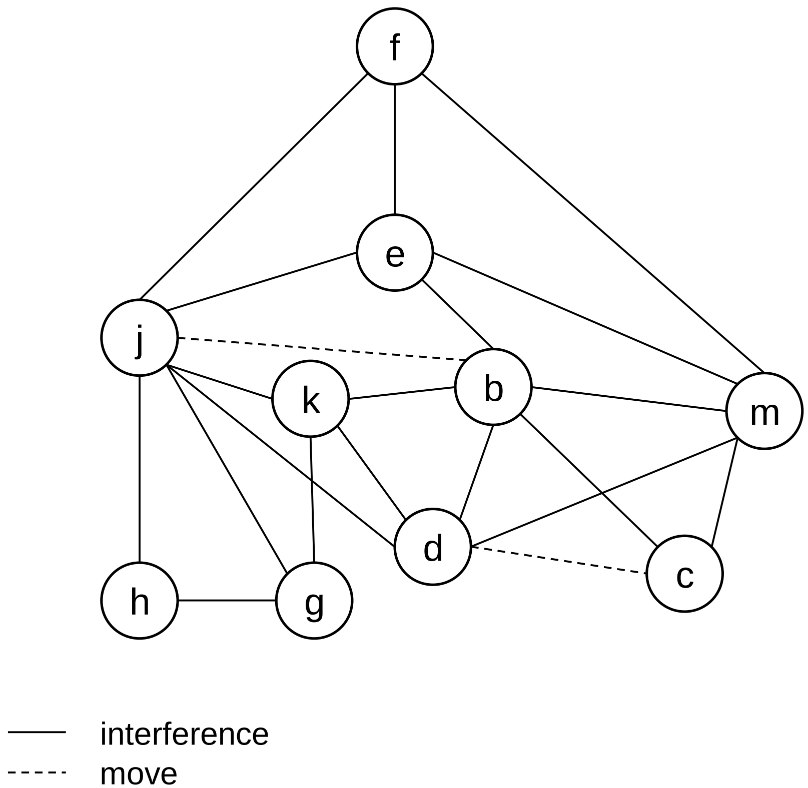
RA - Optimistic Coloring

- Example

live-in: `k, j`

```
g := mem[j+12]
h := k - 1
f := g * h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
```

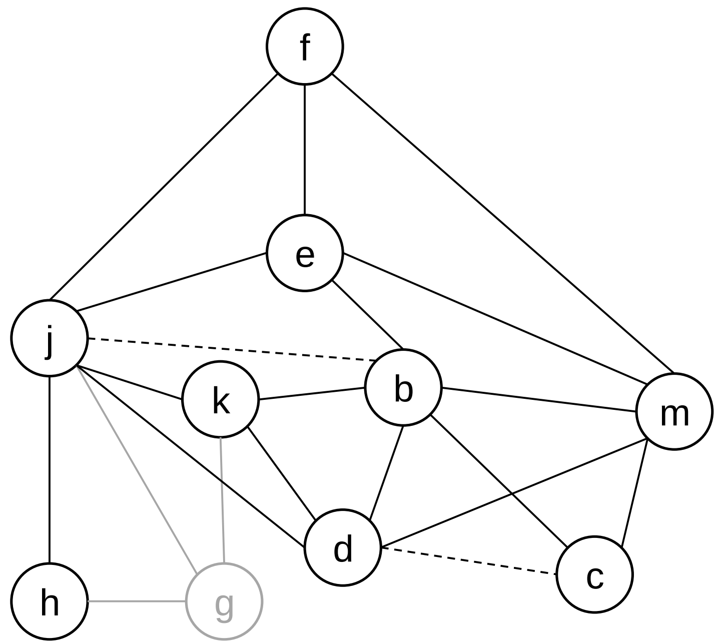
live-out: `d, k, j`



RA - Optimistic Coloring

- Example – k = 4

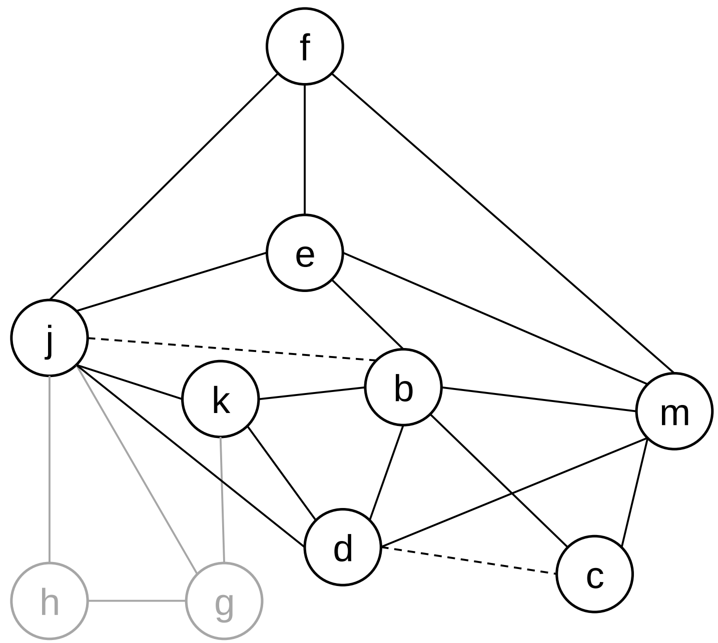
stack:
g



RA - Optimistic Coloring

- Example – k = 4

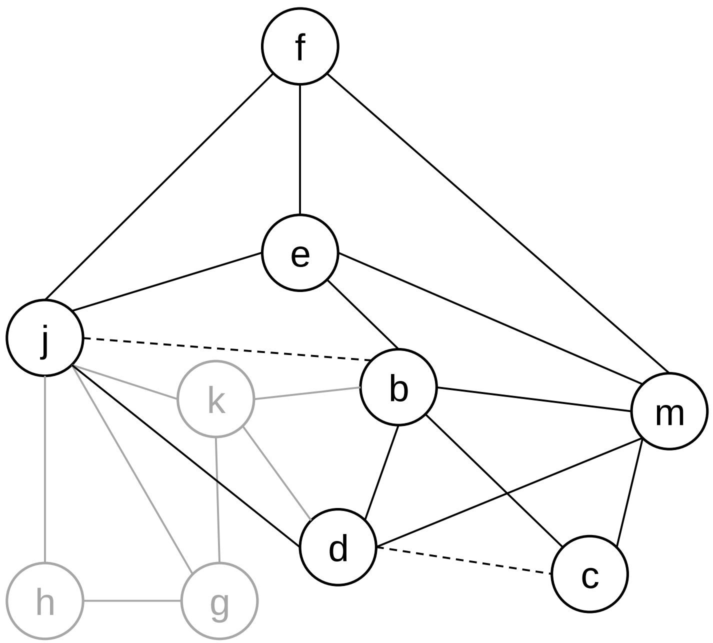
stack:
g
h



RA - Optimistic Coloring

- Example – k = 4

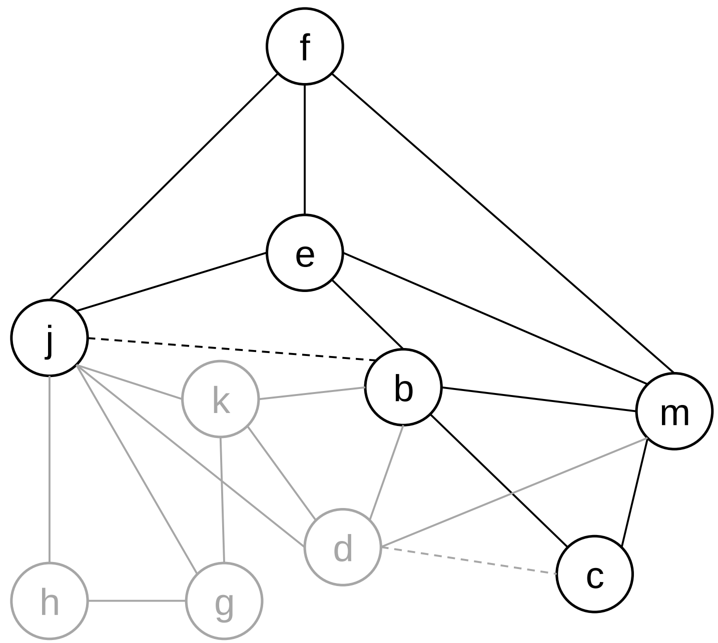
stack:
g
h
k



RA - Optimistic Coloring

- Example – k = 4

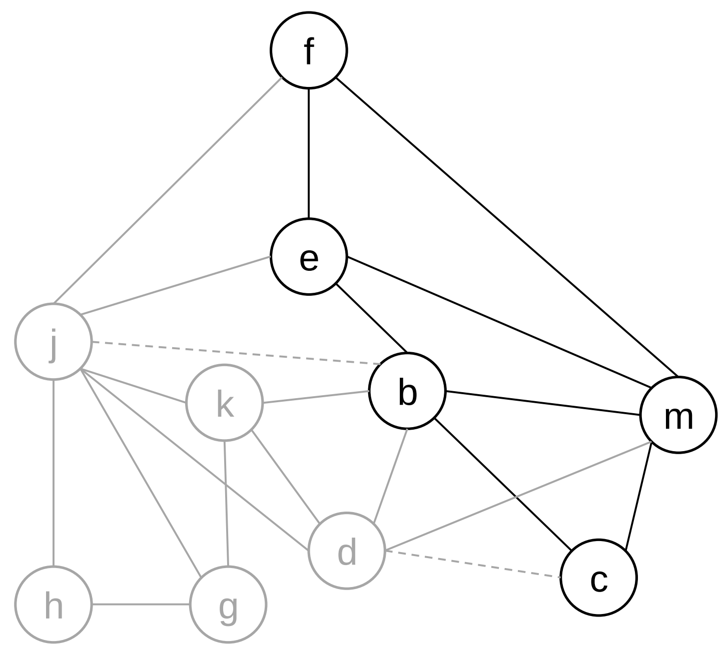
stack:
g
h
k
d



RA - Optimistic Coloring

- Example – k = 4

stack:
g
h
k
d
j

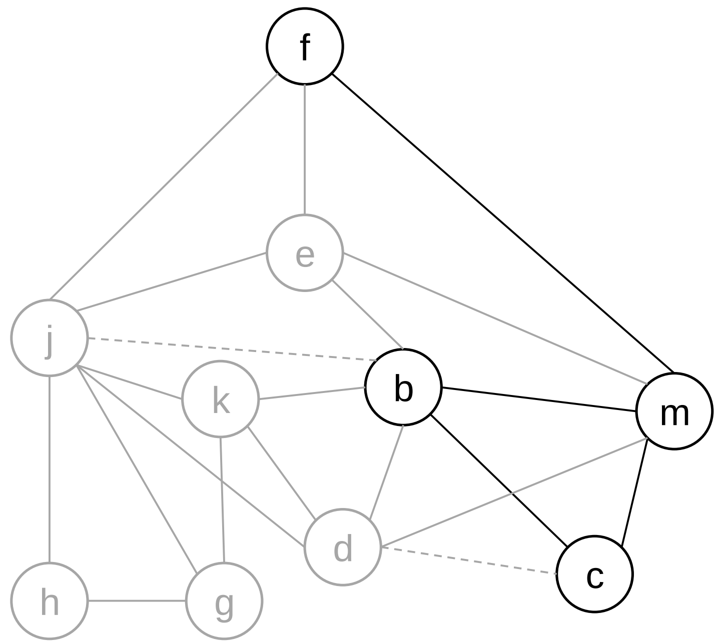


CSE 컴퓨터공학부 Department of Computer Science & Engineering

RA - Optimistic Coloring

- Example – k = 4

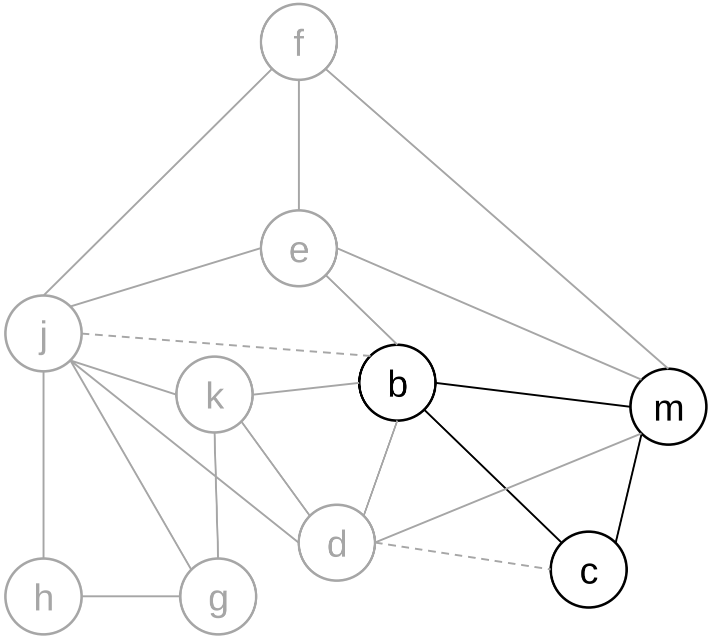
stack:
g
h
k
d
j
e



RA - Optimistic Coloring

- Example – k = 4

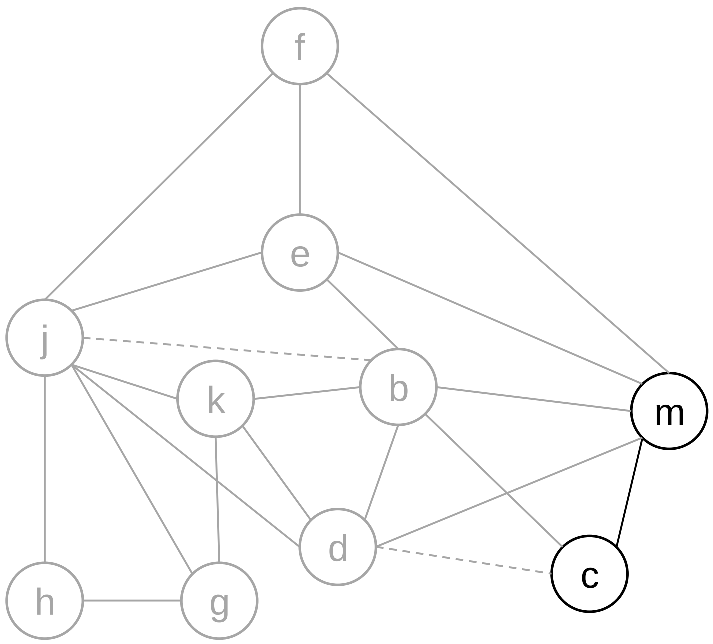
stack:
g
h
k
d
j
e
f



RA - Optimistic Coloring

- Example – k = 4

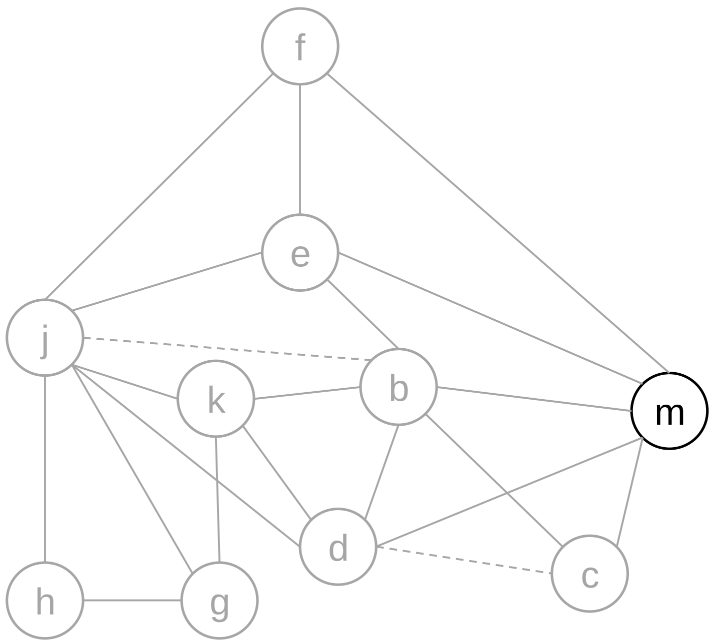
stack:
g
h
k
d
j
e
f
b



RA - Optimistic Coloring

- Example – k = 4

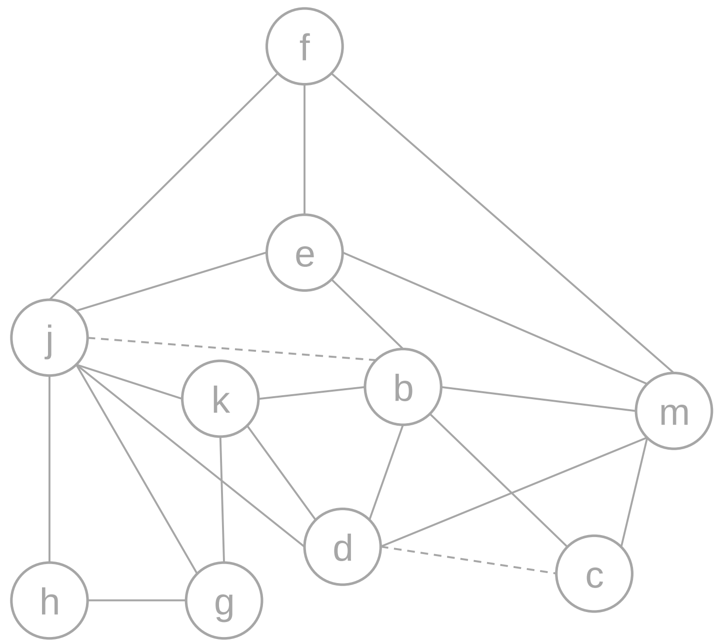
stack:
g
h
k
d
j
e
f
b
c



RA - Optimistic Coloring

- Example – k = 4

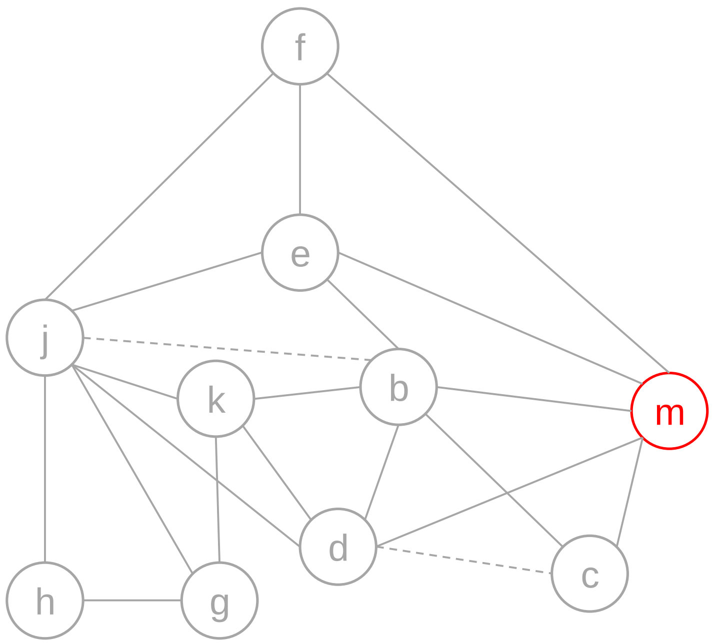
stack:
g
h
k
d
j
e
f
b
c
m



RA - Optimistic Coloring

- Example – k = 4

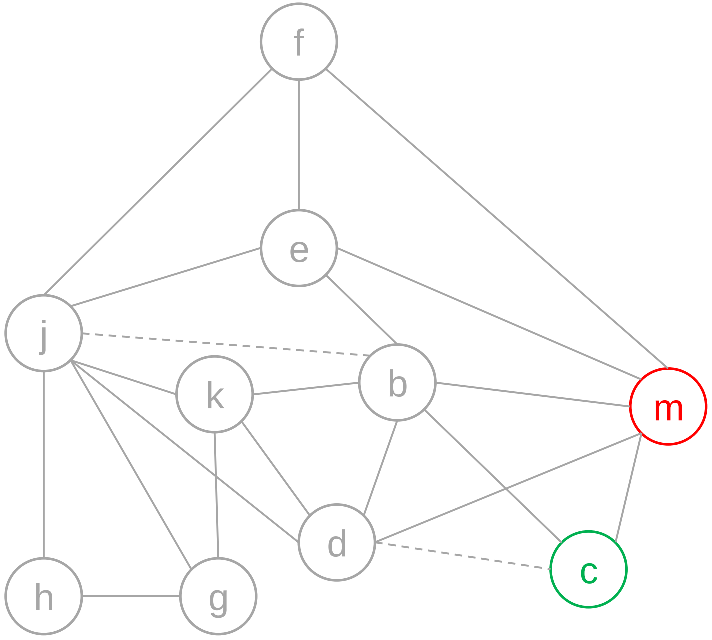
stack:
g
h
k
d
j
e
f
b
c



RA - Optimistic Coloring

- Example – k = 4

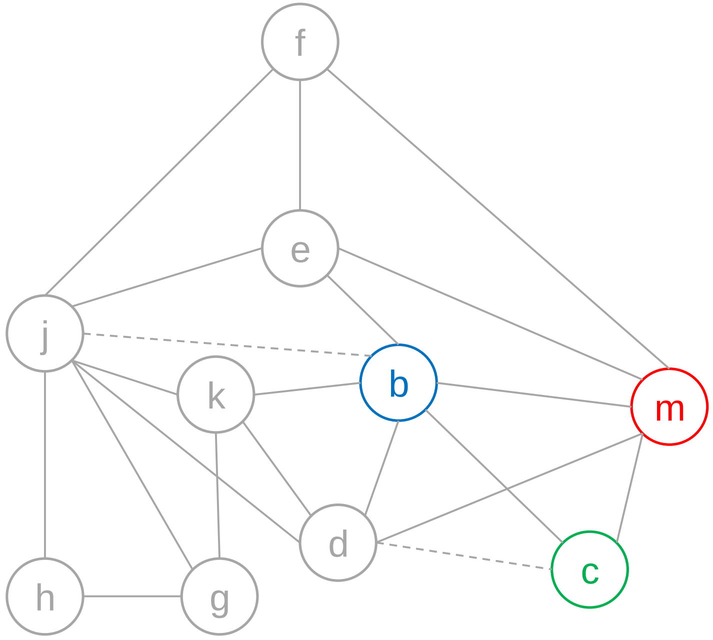
stack:
g
h
k
d
j
e
f
b



RA - Optimistic Coloring

- Example – k = 4

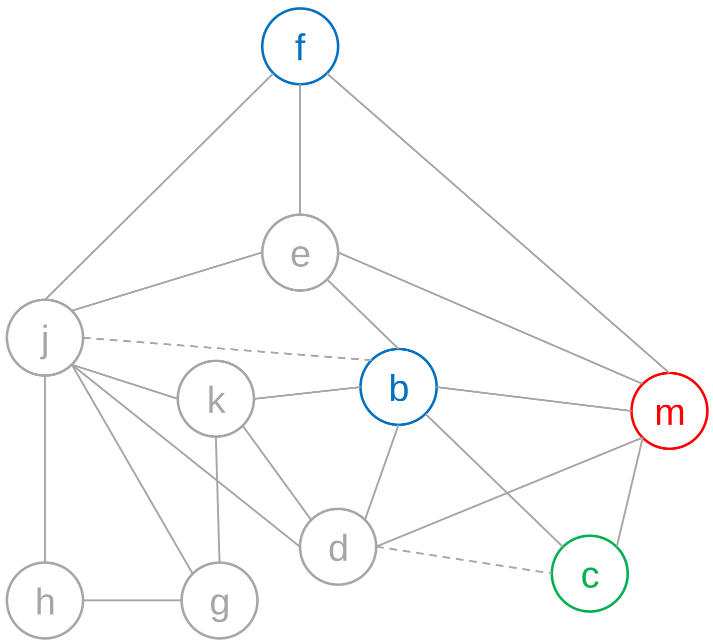
stack:
g
h
k
d
j
e
f



RA - Optimistic Coloring

- Example – k = 4

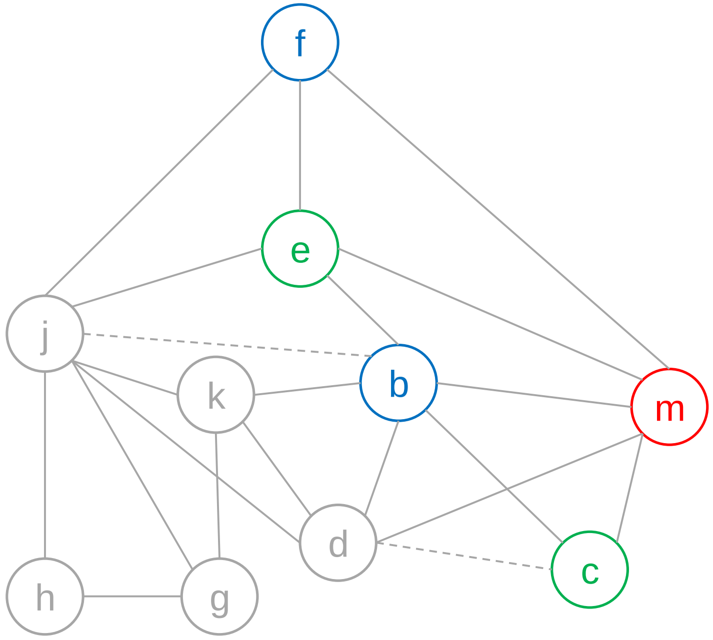
stack:
g
h
k
d
j
e



RA - Optimistic Coloring

- Example – k = 4

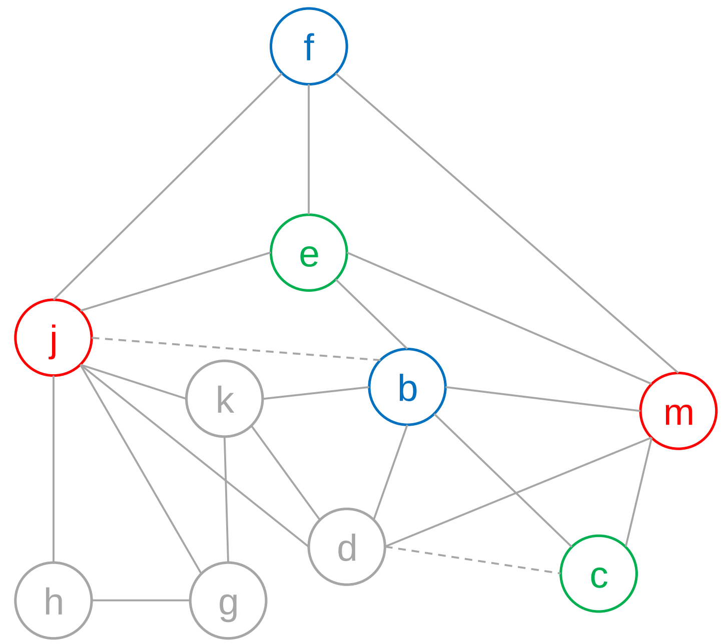
stack:
g
h
k
d
j



RA - Optimistic Coloring

- Example – k = 4

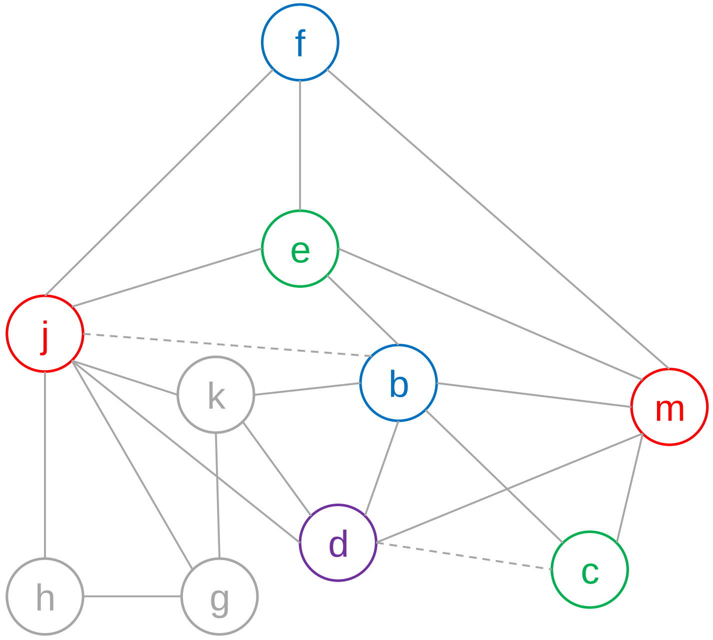
stack:
g
h
k
d



RA - Optimistic Coloring

- Example – k = 4

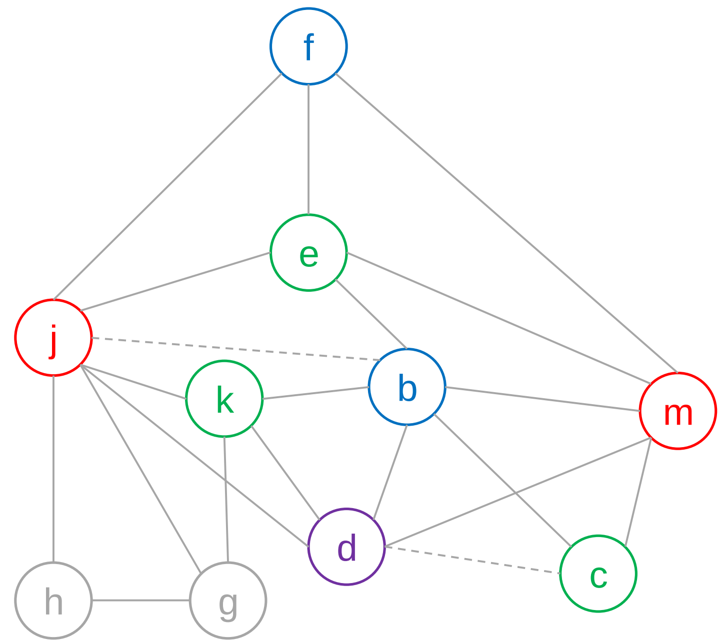
stack:
g
h
k



RA - Optimistic Coloring

- Example – k = 4

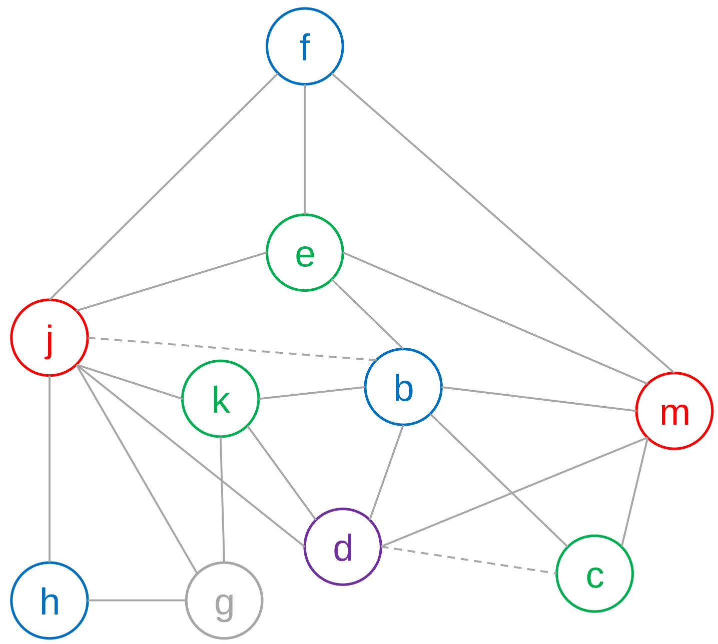
stack:
g
h



RA - Optimistic Coloring

- Example – k = 4

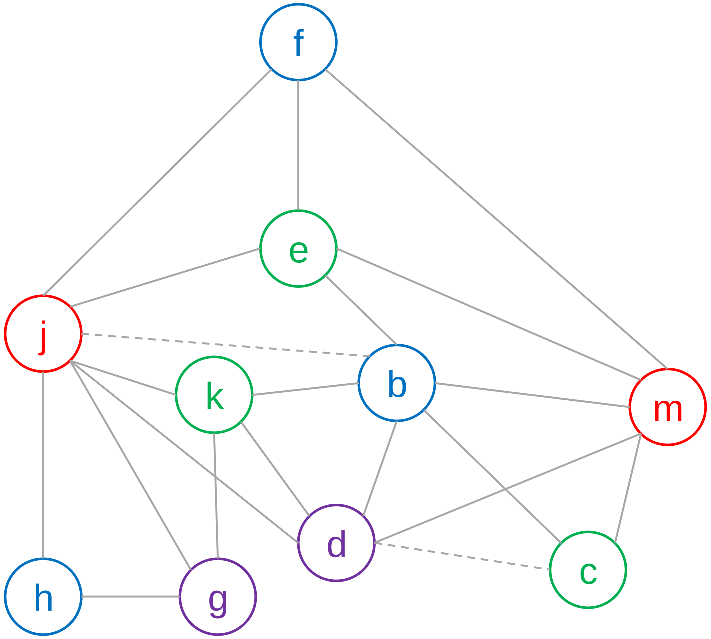
stack:
g



RA - Optimistic Coloring

- Example – k = 4

stack:

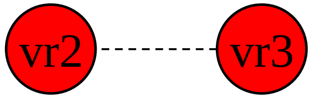


Problems to Solve (cont'd)

- **How can we assign the same color to MOVE nodes?**

  If we can assign the same color to the involved nodes, we can delete the MOVE instruction entirely.

```
vr3 ← vr2
     ...
     ← ...vr2...
     ...
     ← ...vr3...
```



~~r5 ← r5~~
...
← ...r5...
...
← ...r5...

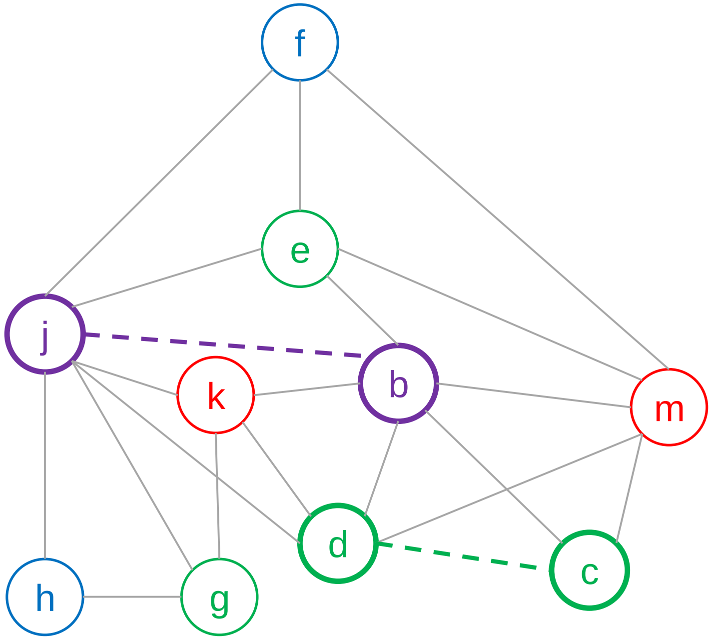
RA with Coalescing (Copy Propagation)

- **Coalescing**: eliminating redundant move instructions
- **Goal**: assign same color for move-related instructions and remove copy node

live-in: `k, j`

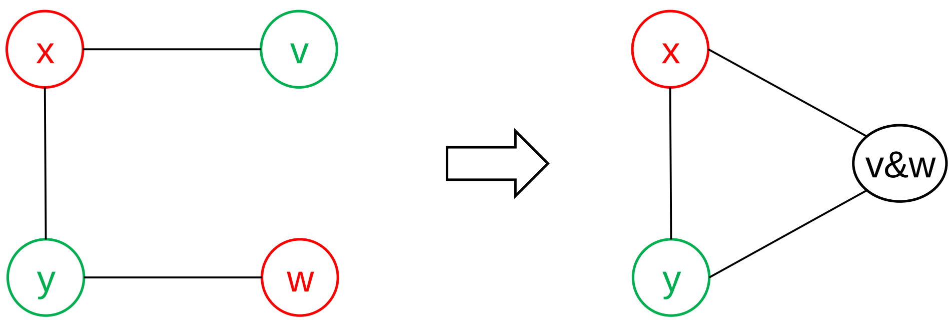
```
g := mem[j+12]
h := k - 1
f := g * h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
```

live-out: `d, k, j`



RA - Coalescing

- Coalescing two nodes into a new node → edges = union of original nodes
- Any pair of non-interfering nodes can be coalesced
  - aggressive coalescing: coalesce all coalescable nodes
    - k-colorable graph may no longer be colorable
    - need to spill (expensive)



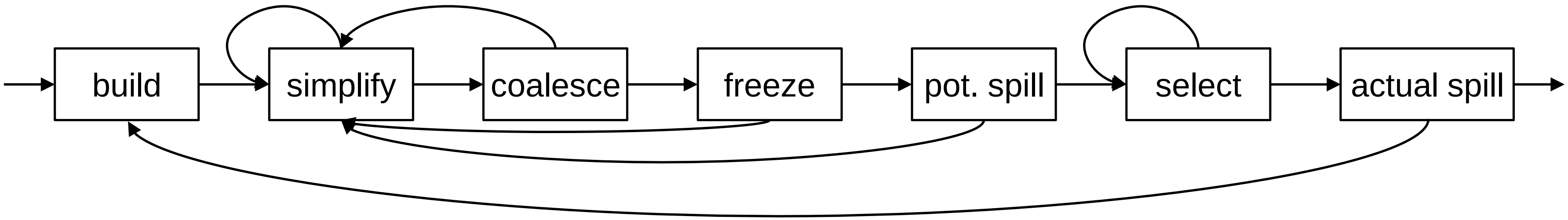
RA - Coalescing

- Safe coalescing
  - safe = k-colorable graph remains k-colorable

- Conservative variants
  - Briggs
    coalesce only if coalesced node has less than k neighbors of significant degree
  - George
    coalesce a & b if, for every neighbor t of a, t already interferes with b or is of insignificant degree

RA - Coalescing

- RA with Conservative Coalescing



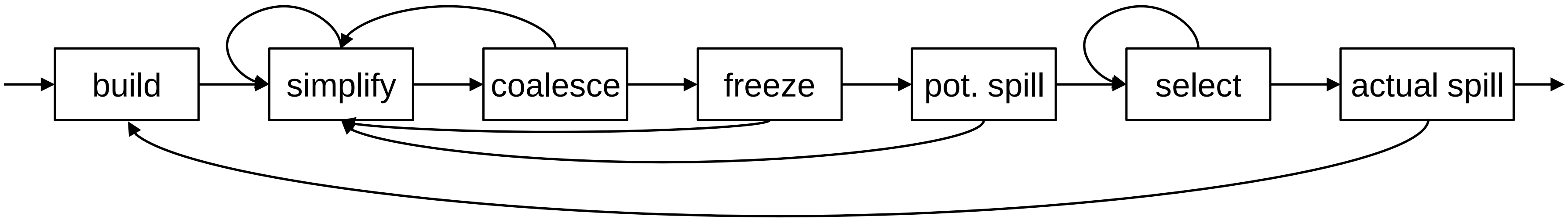
- **build**: construct interference graph
  mark move-related nodes (source or destination)

- **simplify**: simply interference graph by pushing *non-move-related* nodes with fewer than k neighbors, one by one.

- **coalesce**: conservative coalescing. Classify coalesced node (move-related or not).
  Repeat simplify-coalesce until only the following nodes remain
  a) significant-degree nodes or
  b) non-coalescable move-related nodes

RA - Coalescing

- RA with Conservative Coalescing



- **freeze**: classify move-related nodes of low degree as non-move related, i.e., give up on coalescing them.
  Resume simplify-coalesce

- **pot. spill**: if all nodes are significant, select one for potential spilling and push it on the stack.
  Resume simplify-coalesce-freeze

- **select**: standard color selection

- **actual spill**: spill node, start over

RA - Coalescing

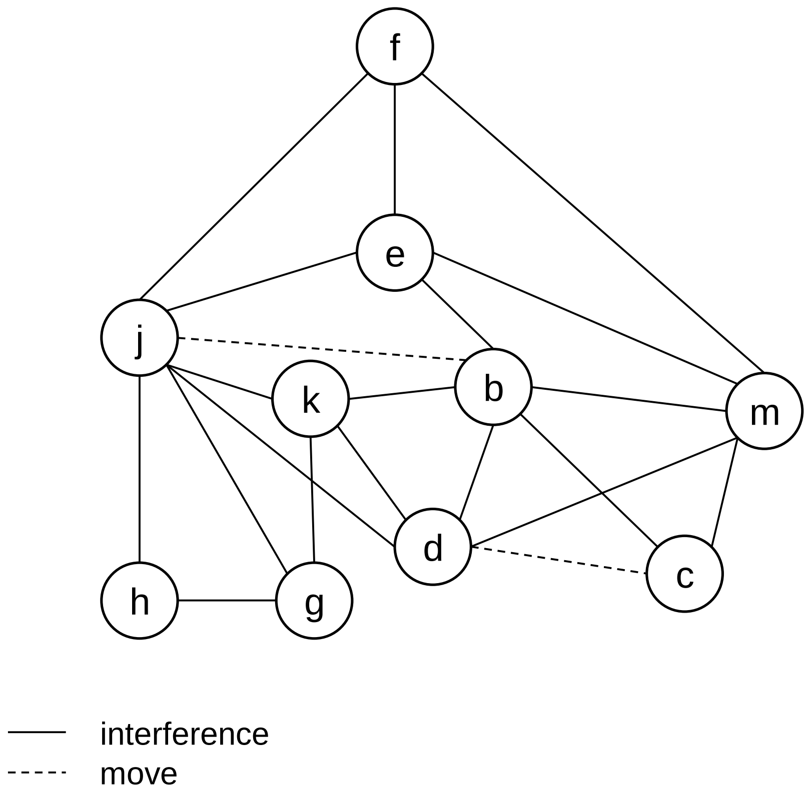
- Example – k = 4

  move-related nodes: b, c, d, j

live-in: `k, j`

```
g := mem[j+12]
h := k - 1
f := g * h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
```

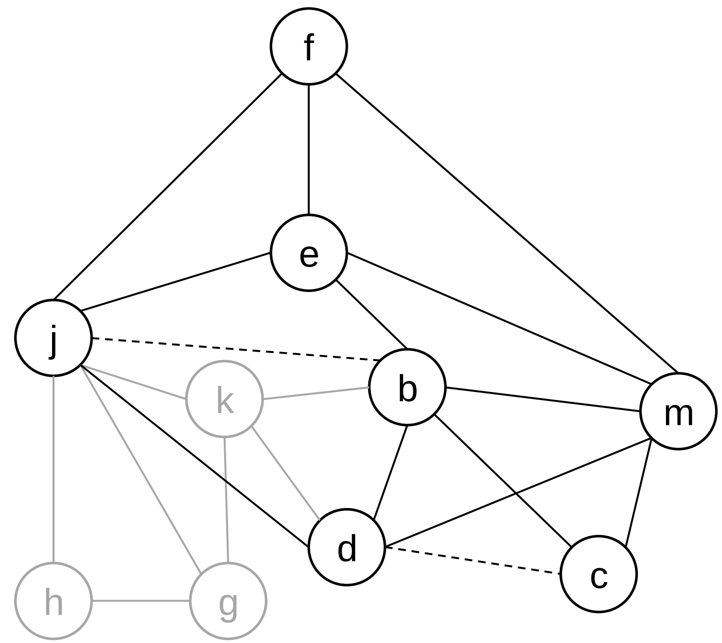
live-out: `d, k, j`



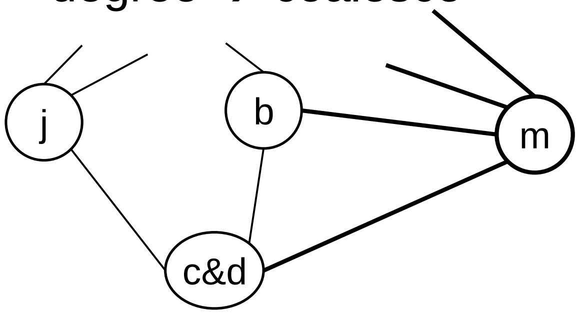
RA - Coalescing

- Example – after removing g, h, k

stack:
g
h
k



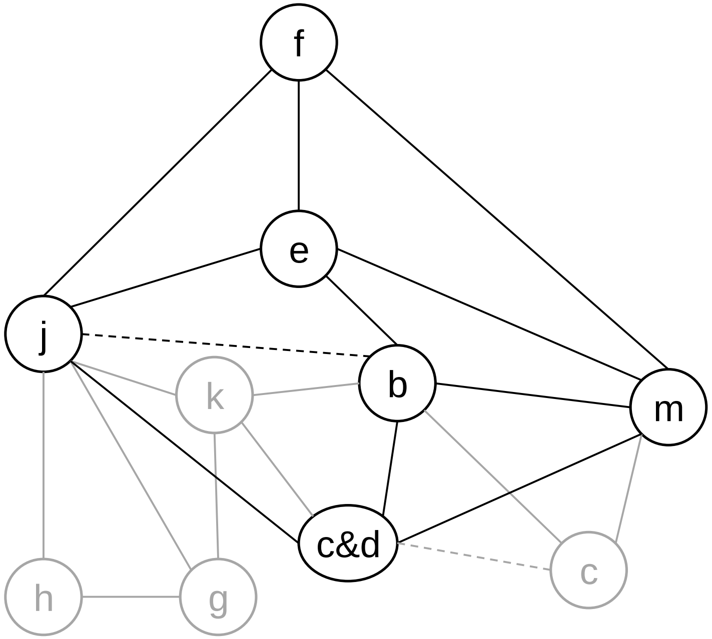
- consider coalescing c, d: only one node of significant degree → coalesce



RA - Coalescing

- Example – after coalescing c&d

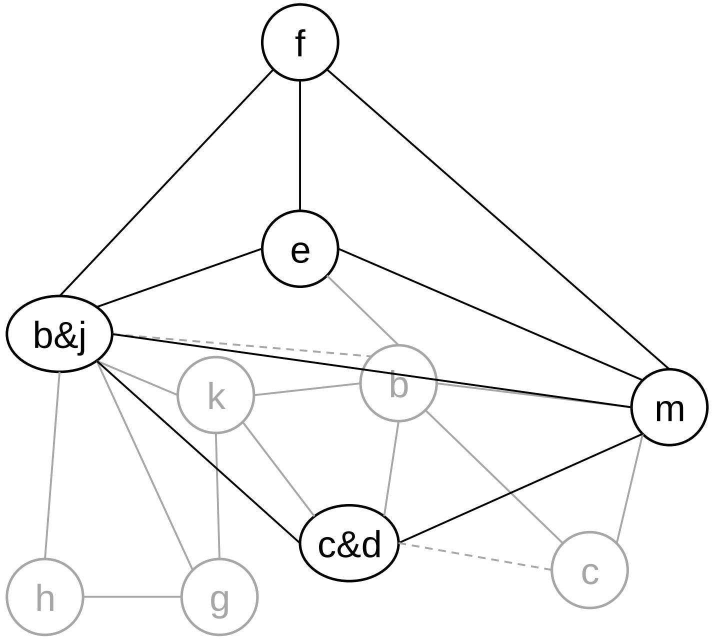
stack:
g
h
k
c&d



RA - Coalescing

- Example – after coalescing b&j

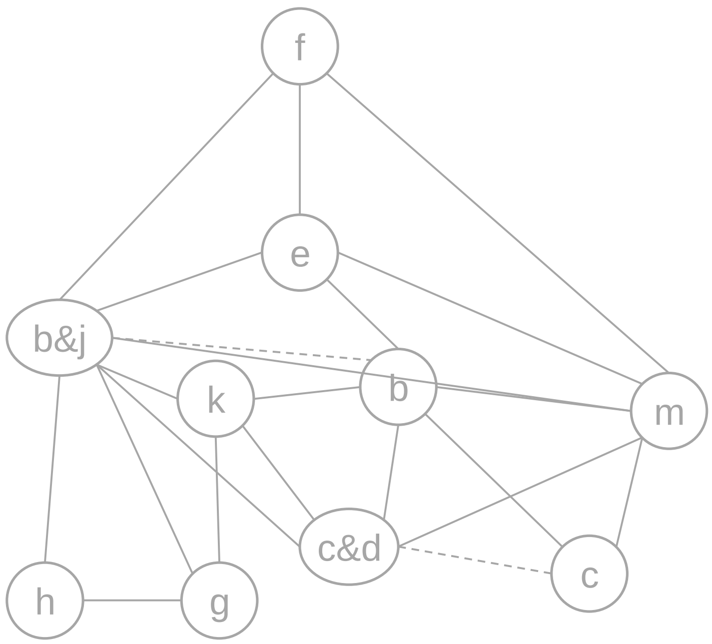
stack:
g
h
k
c&d
b&j



RA - Coalescing

- Example – before selection

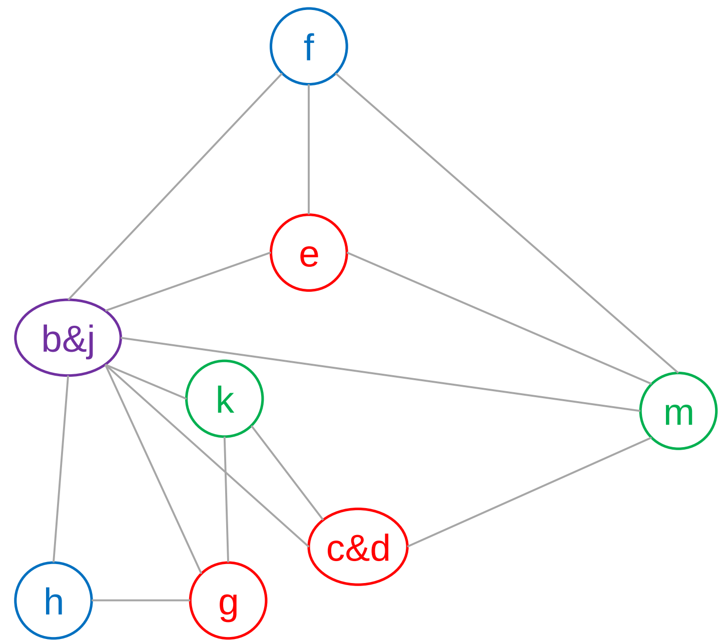
stack:
g
h
k
c&d
b&j
f
m
e



RA - Coalescing

- Example – after selection

stack:
g
h
k
c&d
b&j
f
m
e



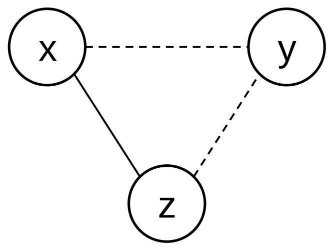
RA - Coalescing

- **Constrained nodes**
  - move-related nodes that are neither coalesced nor frozen

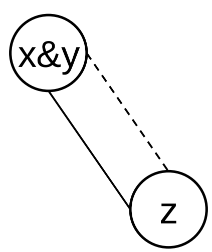
live-in: `y, z`

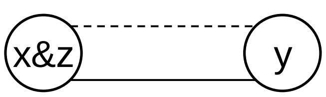
```
x := y
y := z
```

live-out: `x`



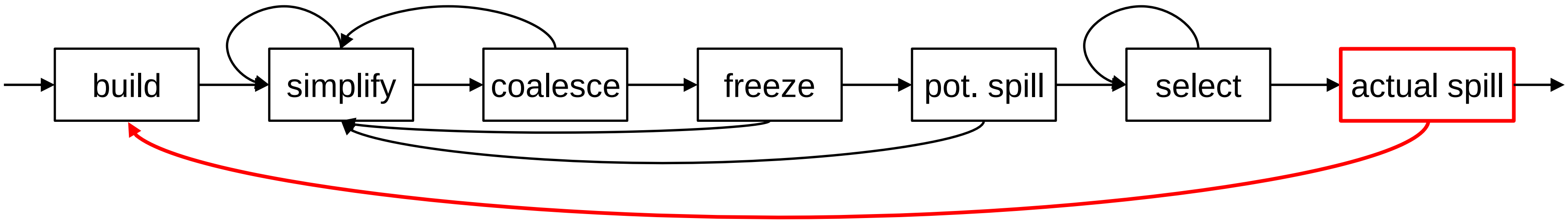
  - after selecting either x&y or y&z for coalescing, the remaining node cannot be coalesced anymore → remove move-related flag





RA – Spilling and Coalescing

- **Actual spills require re-running the entire graph coloring**



- optimistic coloring:
  simply go back to the **build** phase

- graph coloring with coalescing
  - simple: discard any coalesced nodes and start over
  - more efficient: preserve coalesced nodes done before the first potential spill was discovered, uncoalesce any coalescences done after that point

RA – Spilling

- **Optimal spilling**
  - NP complete
  - lots and lots of papers on it

  - basic intuition
    - spill nodes that do not cause too much overhead when spilled
    - cost model based on
      - profile information
      - static estimation (i.e., if-else: 50:50, all loops executed 10 times)

  - spilling may introduce temporaries and thus not decrease register pressure
    - can use a dedicated register for spilled values
    - x86 architectures support memory operands
      → no temporary register needed

RA – Coalescing of Spills

- **Problems with a large number of spills**
  - each spill requires a location on the stack
    $\rightarrow$ stack frame may grow very large
  - spilled move instructions:

…	…
$a \leftarrow b$	$t \leftarrow M[b_{loc}]$
…	$M[a_{loc}] \leftarrow t$
	…

  - coalesce spills (with $k = \infty$)
    - interference graph shows live ranges for spilled nodes
    - coalesce *all* pairs of non-interfering spilled nodes
    - run simplify & select
    - # of colors = # of locations for spills

RA – Precolored Nodes

- **Precolored nodes**
  - represent fixed registers (i.e., certain instructions on IA32 require the source to be in a specific register)
  - model calling convention
    - parameters, return value
    - caller/callee-saved registers

  - handling precolored nodes
    - add registers as precolored nodes to interference graph
      - all registers interfere with each other
      - registers only interfere with normal nodes if explicitly used (i.e. CC)
    - simplify: cannot simplify precolored nodes
    - coalesce: can coalesce precolored nodes with normal nodes
    - spill: cannot spill precolored nodes
      - implementation: spill cost = infinite

RA – Precolored Nodes

- **Hints for handling precolored nodes**
  - callee-saved registers
    - def in ENTRY, use in EXIT
    - long live range hampers ability to color
    - introduce temporary copies in the code generator

ENTRY: def(ebx)	ENTRY: def(ebx) $t \leftarrow ebx$
…	…
EXIT: use(ebx)	EXIT: $ebx \leftarrow t$ use(ebx)

  - same technique can be used for calling conventions

References and Further Reading

- A. Aho et al. “Compilers: Principles, Techniques, and Tools,” 2nd edition, *Addison Wesley*, 2006, ISBN 978-0321486813
- A. Appel. “Modern Compiler Implementation in Java,*”* 2nd edition, *Cambridge University Press*, 2002, ISBN 978-0521820608
- G. Chaitin et al. "Register Allocation via Coloring," *Computer Languages,* vol. 6,issue 1, 1981
- P. Briggs et al. “Improvements to Graph Coloring Register Allocation,” *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 16, issue 3, 1994
- G. Lueh et al. “Fusion-based Register Allocation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, issue 3, 2000