Register Allocation – 2
SSA-based Register Allocation

15-411/15-611 Compiler Design

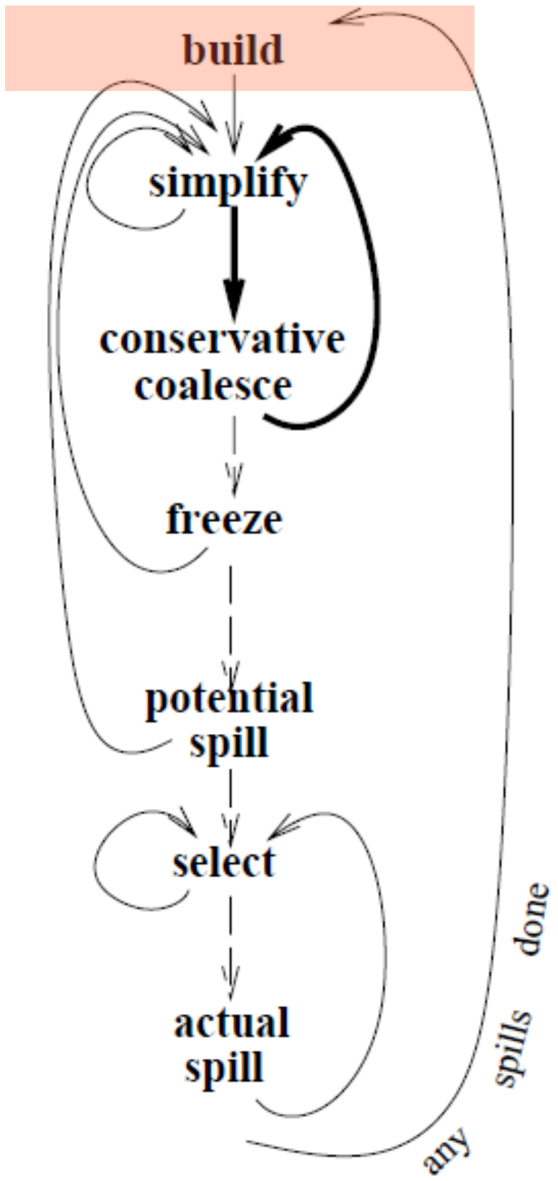
Seth Copen Goldstein

January 21, 2025

Today

- Iterated Register Allocation
  - Coalescing
  - Special registers
  - Spilling
  - Frame slot coalescing
  - Implementation
- SSA-Based Register Allocation
  - SSA
  - φ functions
  - Chordal Graphs
  - Perfect Elimination Order

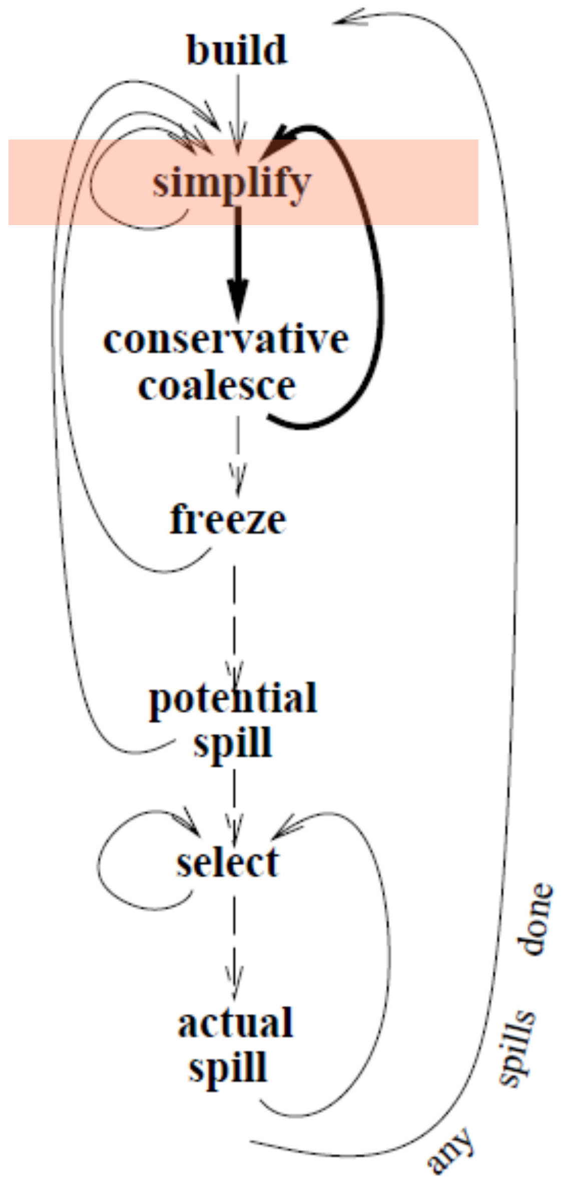
Iterated Register Coloring



Build:

- construct interference graph
  - Construct liveness information
  - Add edge (*u*,*v*) to IG if at point of definition of *u*, *v* is live.

Iterated Register Coloring

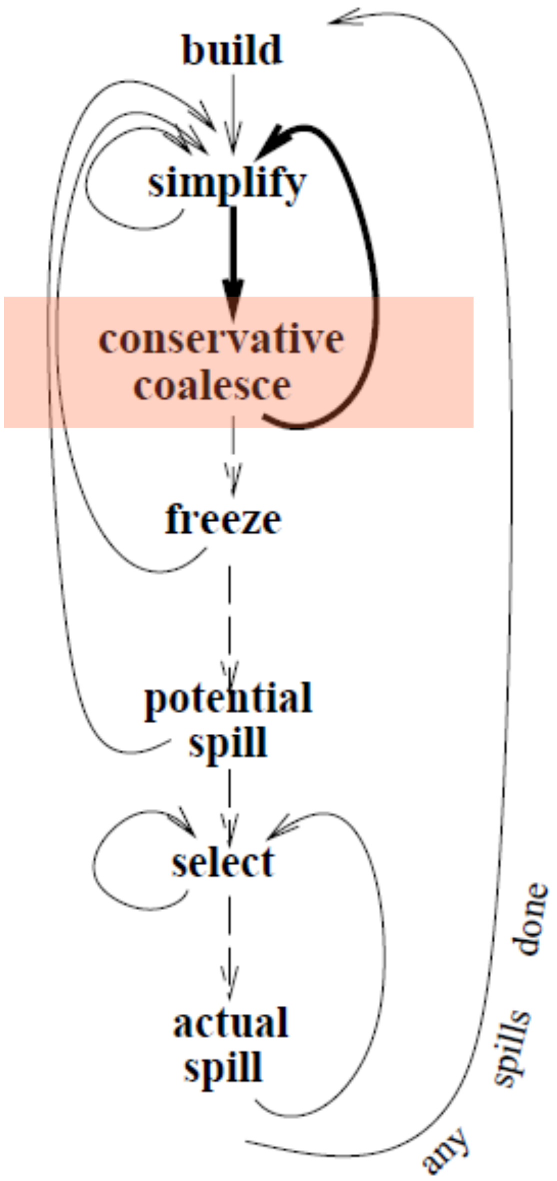


Simplify:

Repeat

- remove nodes with degree < K
- And, which are not “move related”

Iterated Register Coloring

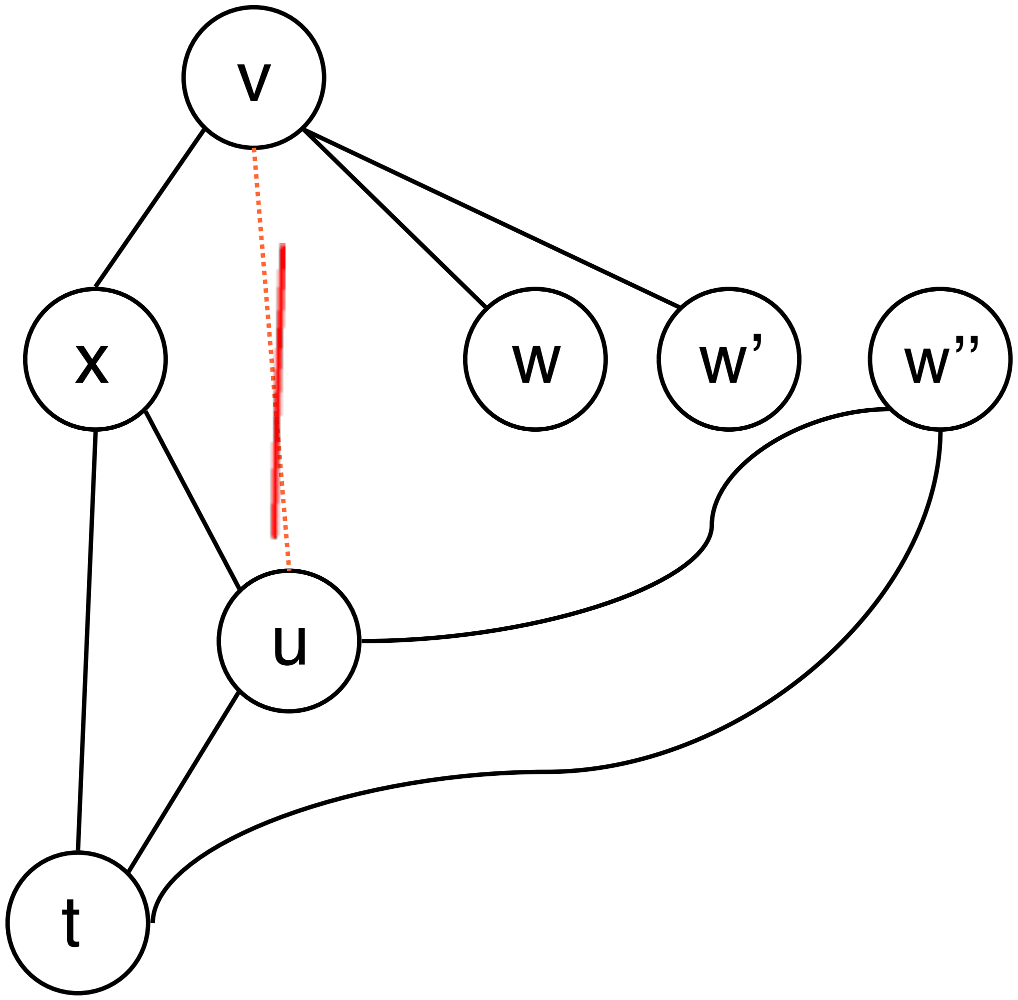


Coalesce:

- For any move related nodes:
    - if they pass conservative test
        - briggs for temp<->temp
        - preston for temp<->hard
    - then, mark move to be deleted
    - merge nodes
    - update degree of neighbors, etc.
    - back to simplify

Coalescing

```
v   ⍰   1
w   ⍰   v + 3
M[] ⍰   w
w'  ⍰   M[]
x   ⍰   w' + v
u   ⍰   v
t   ⍰   u + v
w'' ⍰   M[]
    ⍰   w'' + x
    ⍰   t
    ⍰   u
```

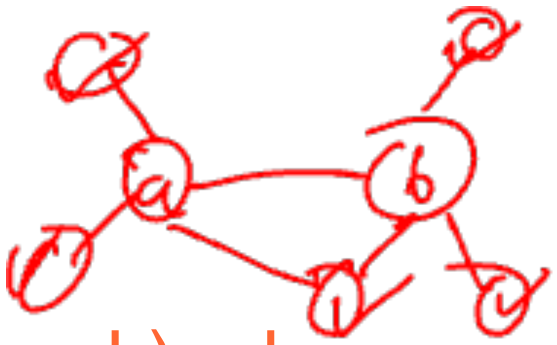


Can u & v be coalesced?
Should u & v be coalesced?

Coalescing

- Conservative or Aggressive?
- Aggressive:
  - coalesce even if potentially causes spill
  - Then, potentially undo
- Conservative:
  - coalesce if it won’t make graph uncolorable
  - How to detect?

Briggs



- Can coalesce a and b if

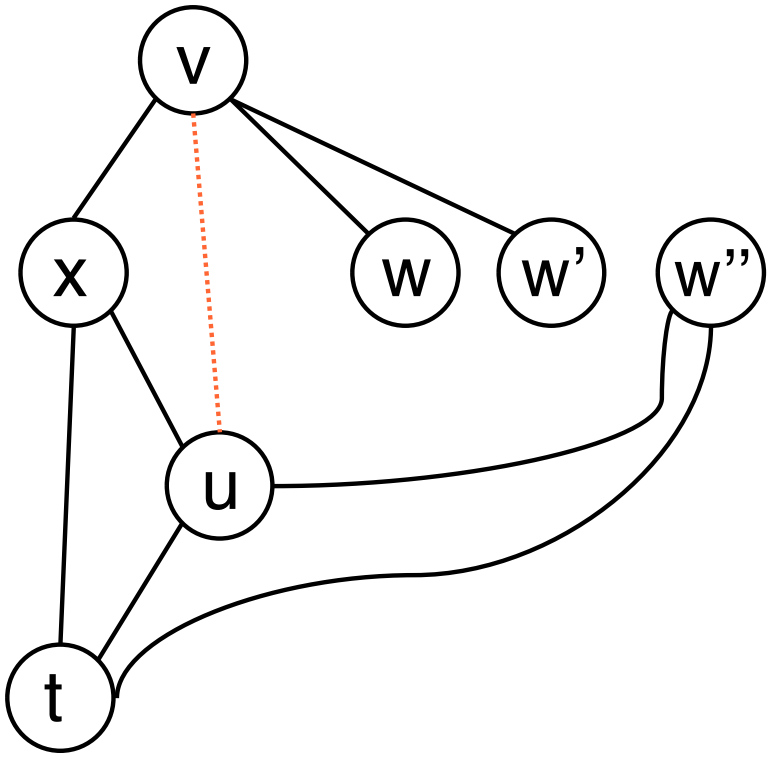
  (# of neighbors of ab with degree k) < k

- Why?

  Simplify removes all nodes with degree < k

  # of remaining nodes < k

  Thus, ab can be simplified



Briggs

- Can coalesce a and b if

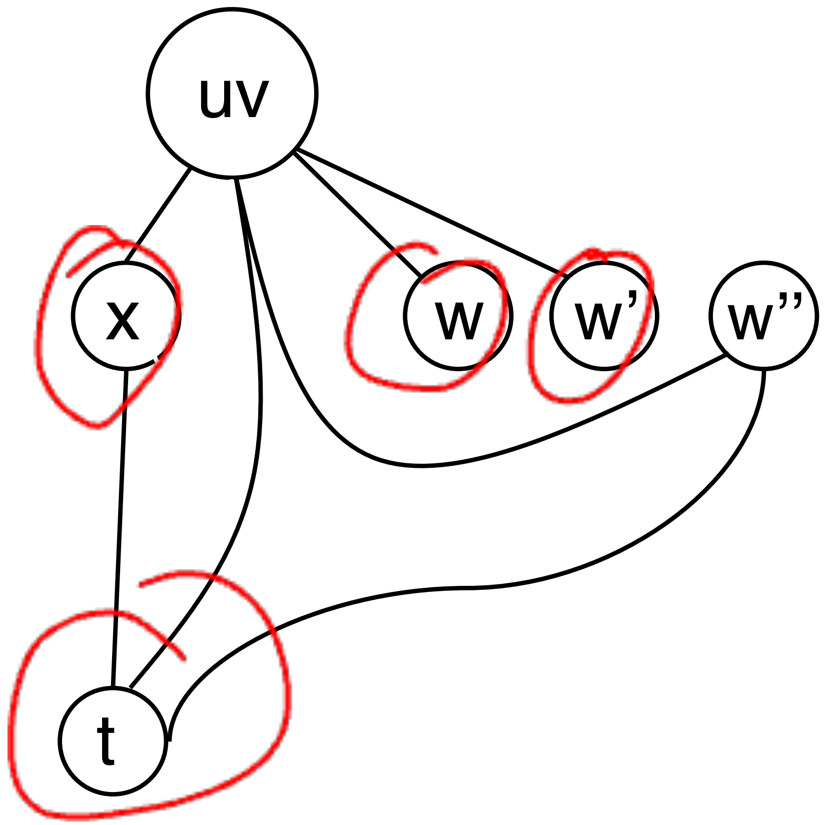
  (# of neighbors of ab with degree   k) < k

- Why?

  Simplify removes all nodes with degree < k

  # of remaining nodes < k

  Thus, ab can be simplified



Preston

- Can coalesce a and b if

  foreach neighbor t of a

    t interferes with b, or,

    degree of t < k

- Why?

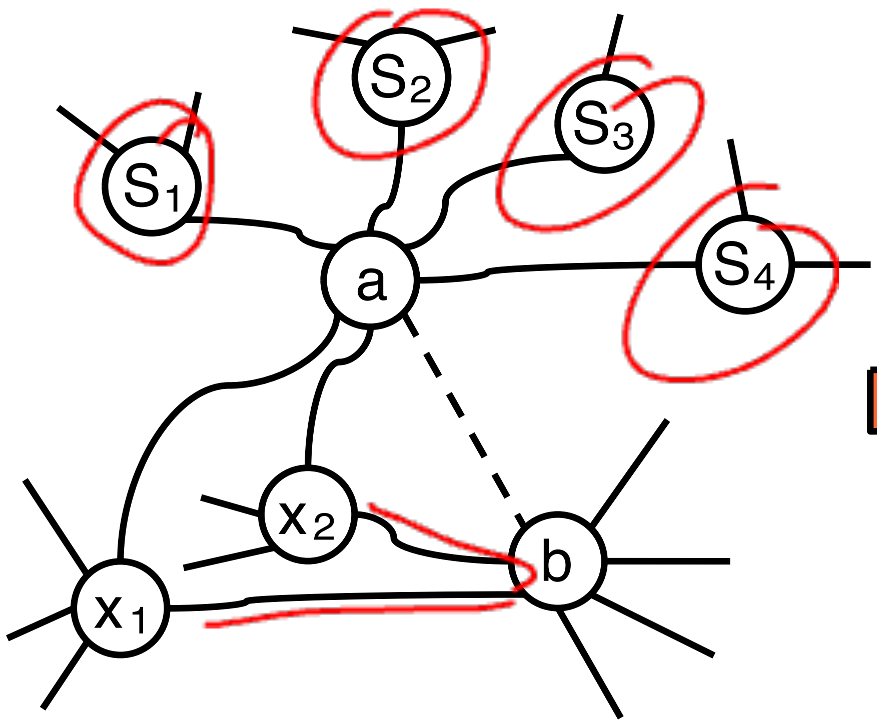
  let S be set of neighbors of a with degree < k

  If no coalescing, simplify removes all nodes in S, call that graph G^1

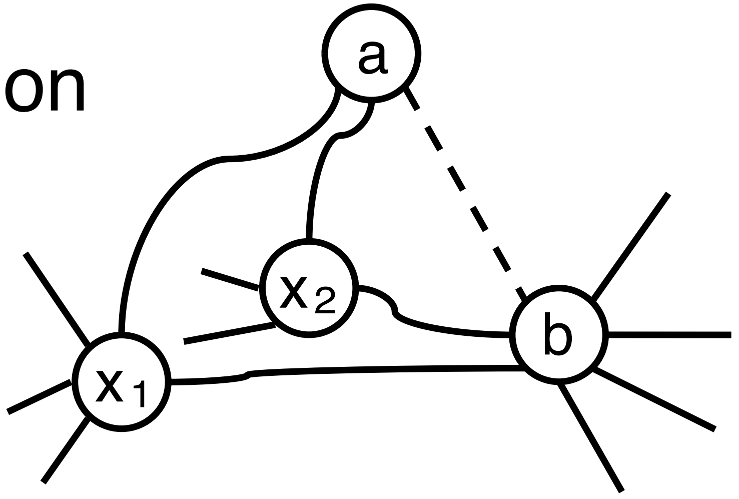
  If we coalesce we can still remove all nodes in S, call that graph G^2

  G^2 is a subgraph of G^1

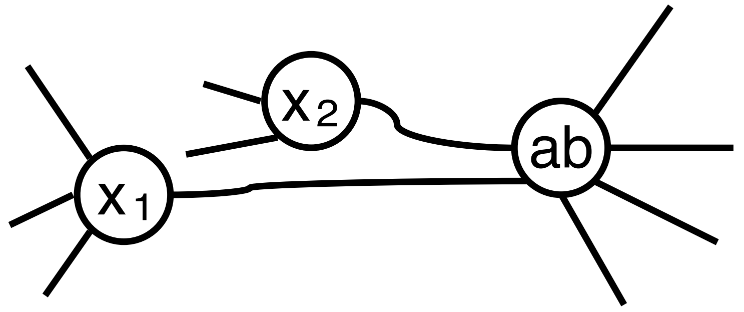
Preston



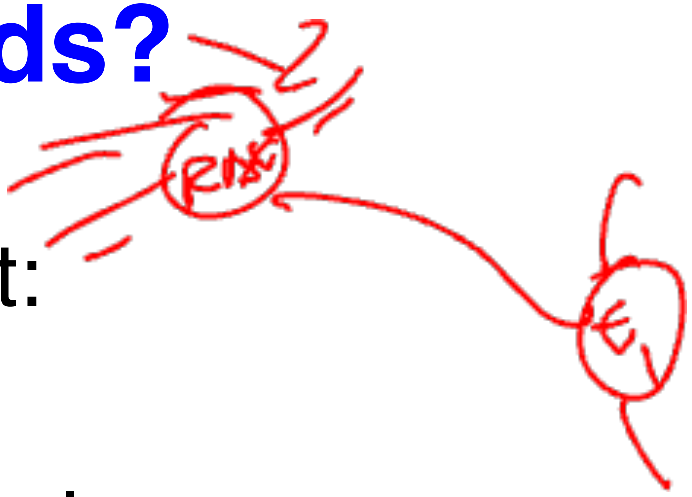
No coalescing, after simplification



After coalescing and simplification



Why Two Methods?

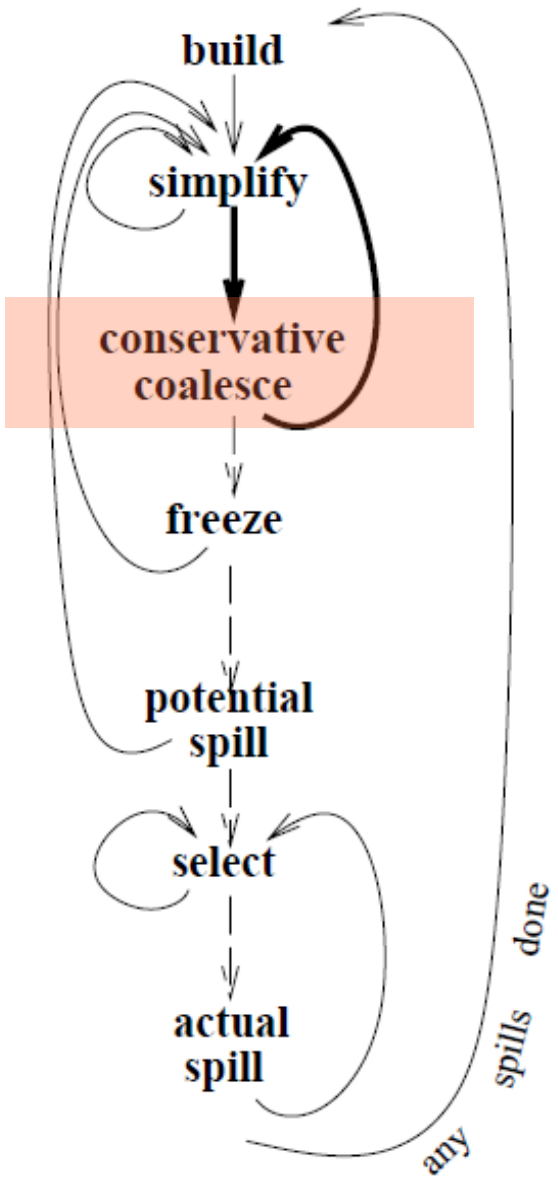


- With Briggs one needs to look at:
  neighbors of **a & b**
- With Preston, only need to look at
  neighbors of **a**.
- As we will see, we will need to insert "hard" registers into graph and they have LOTS of neighbors
  - RAX, RCX, RDI, …
  - Called hard registers
  - aka precolored nodes

Briggs and Preston

- With Briggs one needs to look at:
  neighbors of **a & b**
- With Preston, only need to look at
  neighbors of **a**.
- Briggs
  Used when a and b are both temps
- Preston
  Used when either a or b is precolored

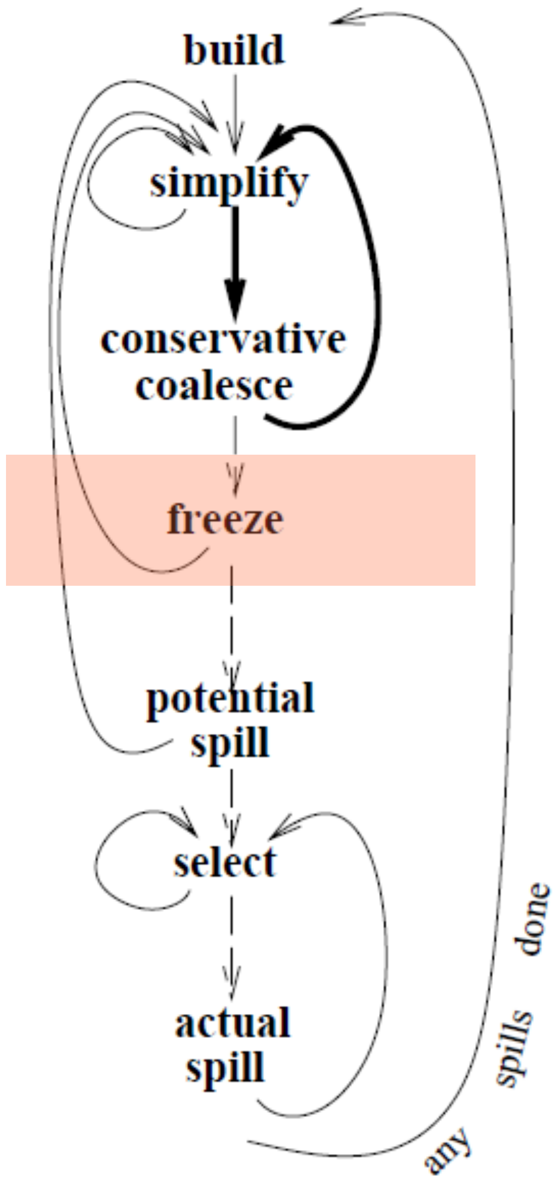
Iterated Register Coloring



Coalesce:

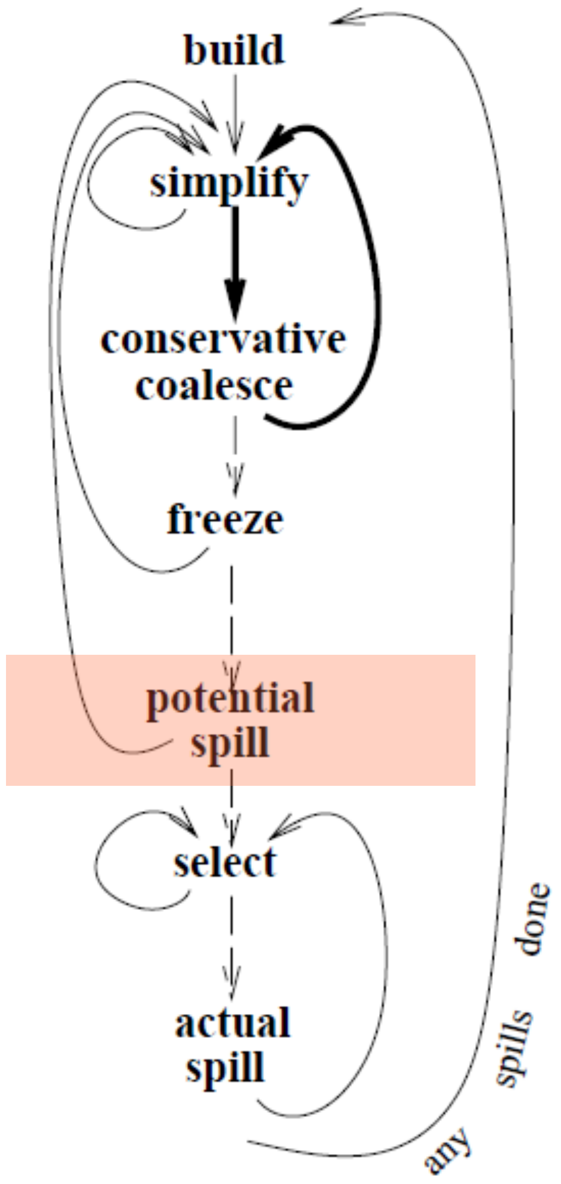
- For any move related nodes:
  - if they pass conservative test
    - briggs for temp<->temp
    - preston for temp<->hard
  - then, mark move to be deleted
  - merge nodes
  - update degree of neighbors, etc.
  - back to simplify

Iterated Register Coloring



Freeze:
- Mark any unremoved "move related" nodes as frozen
- E.g., treat them like regular nodes
- Go back to simplify

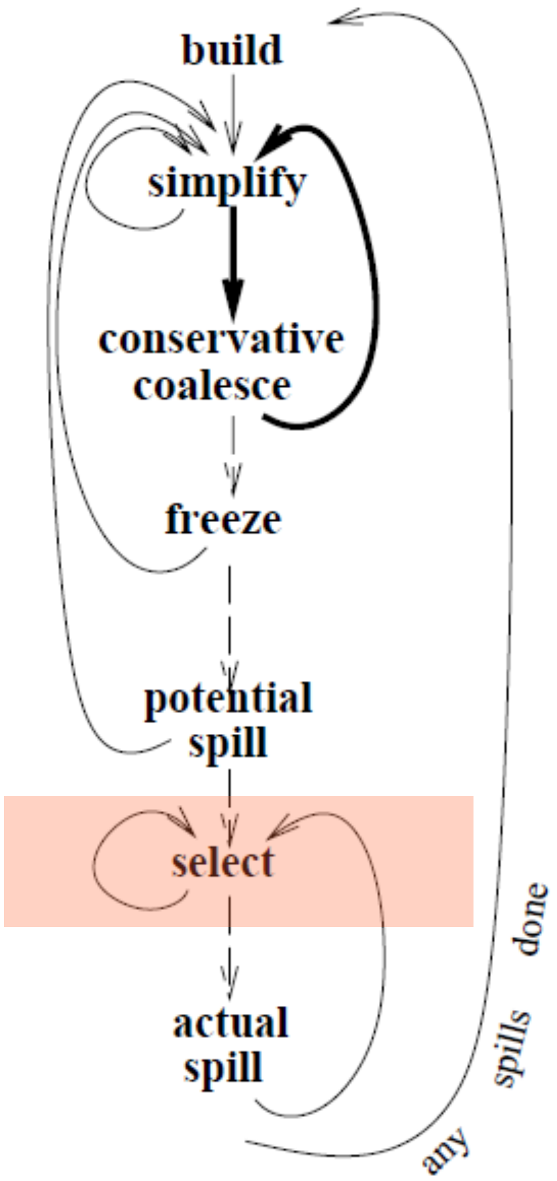
Iterated Register Coloring



Potential Spill:

- Select a node to spill
- remove it and push to stack
- go back to simplify

Iterated Register Coloring



Select:

- Pop nodes, coloring as you go
- If you can't color, then do actual spill
- rewrite code
    - Will have to undo at least some coalescing (can you keep some?)
    - Insert spill code
- go back to build

“Details”

- How to choose a node to spill?
- How to limit size of stack frame?
- What about hard registers?

Spill Heuristics

- Choose a temp to map to stack frame
  - will be used as infrequently as possible
  - will be most likely to make IG colorable
- for each temp evaluate spillCost(t). Choose minimum to potentially spill
- For example:

N ≤ loop depth

```
spillCost(t):
    t.cost = 0
    for every def of t and every use of t
        t.cost += 10^N / t.degree
```

Choosing frame slots

- Want to minimize stack frame.
- if v and u need to be spilled, they could go into same fame slot
- After register allocation is done, can use coloring method (k=?) to color spill slots and use coalescing
  - minimizes frame slots needed
  - can help coalesce spill-spill moves

Choosing frame slots

- Want to minimize stack frame.
- if v and u need to be spilled, they could go into same fame slot
- After register allocation is done, can use coloring method (k=?) to color spill slots and use coalescing
    - minimizes frame slots needed
    - can help coalesce spill-spill moves

```
v ?   ...
w ?   v + 3
...
  ?   w + v
...       //v dead
u ?   x + w
...
  ?   w + u
...
```

Choosing frame slots

- Want to minimize stack frame.
- if v and u need to be spilled,
  they could go into same fame slot
- After register allocation is done,
  can use coloring method (k=?) to
  color spill slots and use coalescing
  - minimizes frame slots needed
  - can help coalesce spill-spill moves

What about special registers?

- Precolored nodes/hard registers
- Instructions with register requirements

d ← a * b

ret x

- Callee-save registers

  x86-64: **RDI**, **RSI**, **RDX**, **RCX**, **R8**, **R9** must be saved by callee if callee wants to use them.

- Special registers: **RSP** or frame pointer

Precolored Nodes

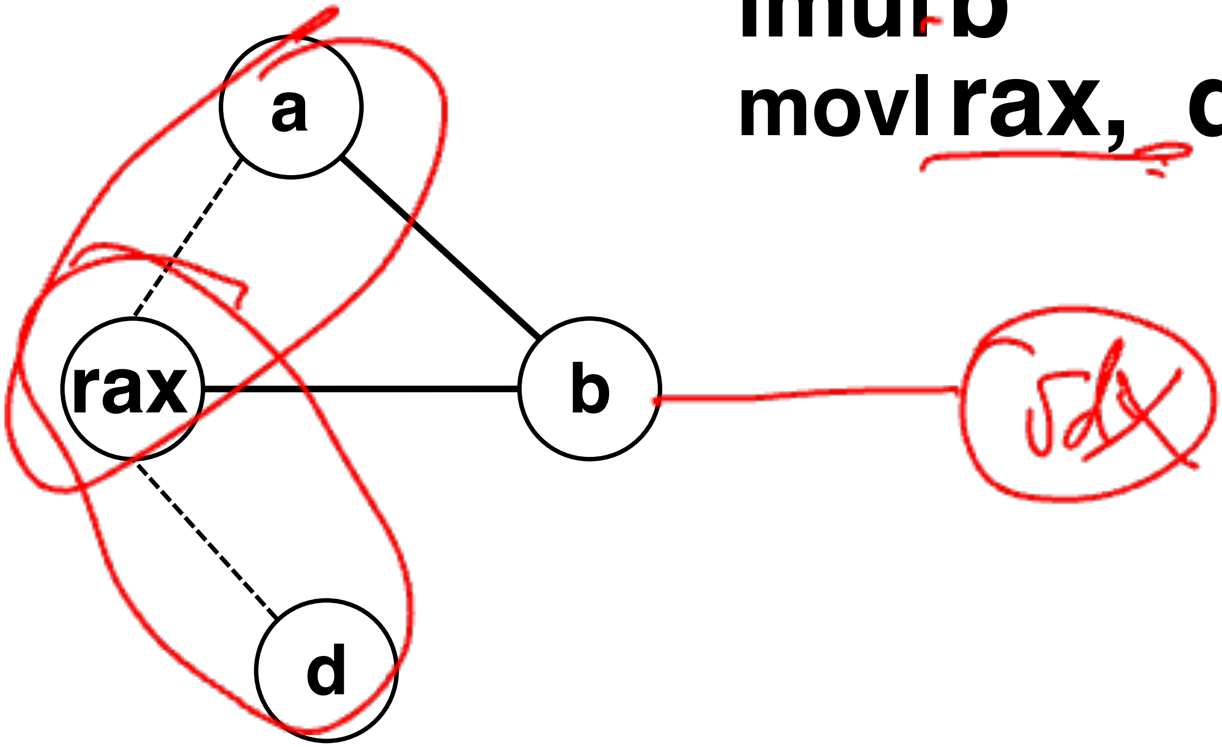
- Some temps are real registers
- Obviously they interfere with each other
    - don’t add edges in IG
    - just set degree to infinity
    - they can’t be spilled.
- Some interfere with all temps (e.g., frame pointer)
- Hope for coalescing
- Start “select” phase when only precolored nodes remain in IG

What about special registers?

- Instructions with register requirements

d ⍰ a * b

```
movl a,  rax
imul b             ;  rdx,rax
movl rax,  d
```

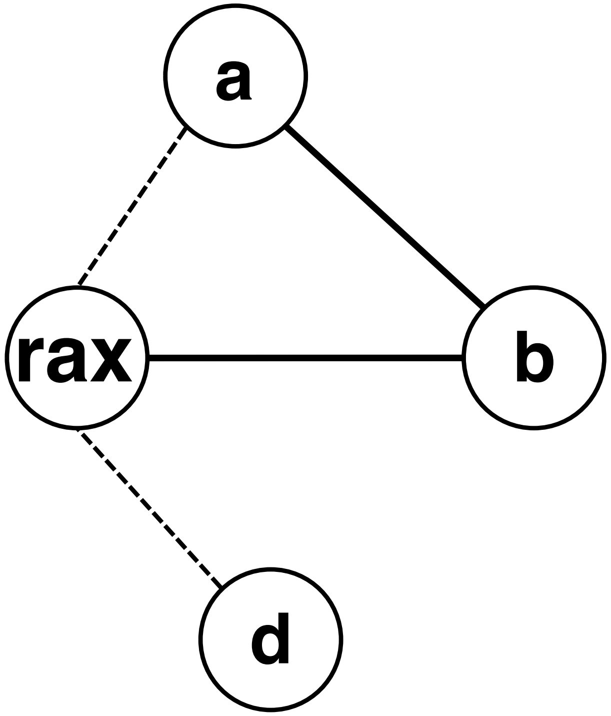


What about special registers?

- Instructions with register requirements

d ⍰ a * b

```
movl a,   rax
imul b              ;  rdx,rax
movl rax,  d
```



If all goes perfectly, then **a** & **d** will end up being coalesced with **rax**

What about special registers?

- Instructions with register requirements

d ⍰ a * b

$\Rightarrow$

```
movl a,  rax
imul b              ;  rdx,rax
movl rax,  d
```

ret x

$\Rightarrow$

```
movl x,  rax
ret
```

Preserving Callee-registers

- Move callee-reg to temp at start of proc
- Move it back at end of proc.
- What happens if there is no register pressure?
- What happens if there is a lot of register pressure?

```
prologue:    define r
             t1 ? r
             …
epilogue:    r ? t1
             use r
```

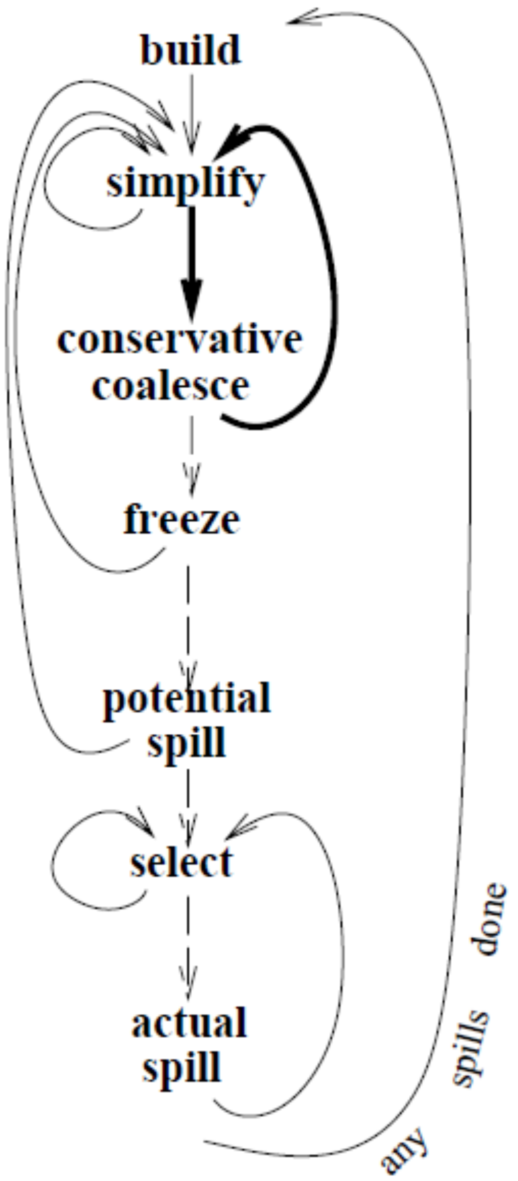
push

pop

Using Caller Save Registers

- Prefer not to use caller save registers across calls
- How can we make this happen with existing machinery?

Iterated Register Coloring



In practice

- Iterated Register Coloring does a good job
- Building Interference Graph is Expensive
  - Calculating live ranges
  - graph is $O(n^2)$
  - Need quick test for interference
  - Need quick test for neighbors
- Coalescing is important
  - Many passes generate extra temps and moves
  - Aggressive requires fix-up (e.g., live range splitting)
- Spilling has biggest impact on generated code

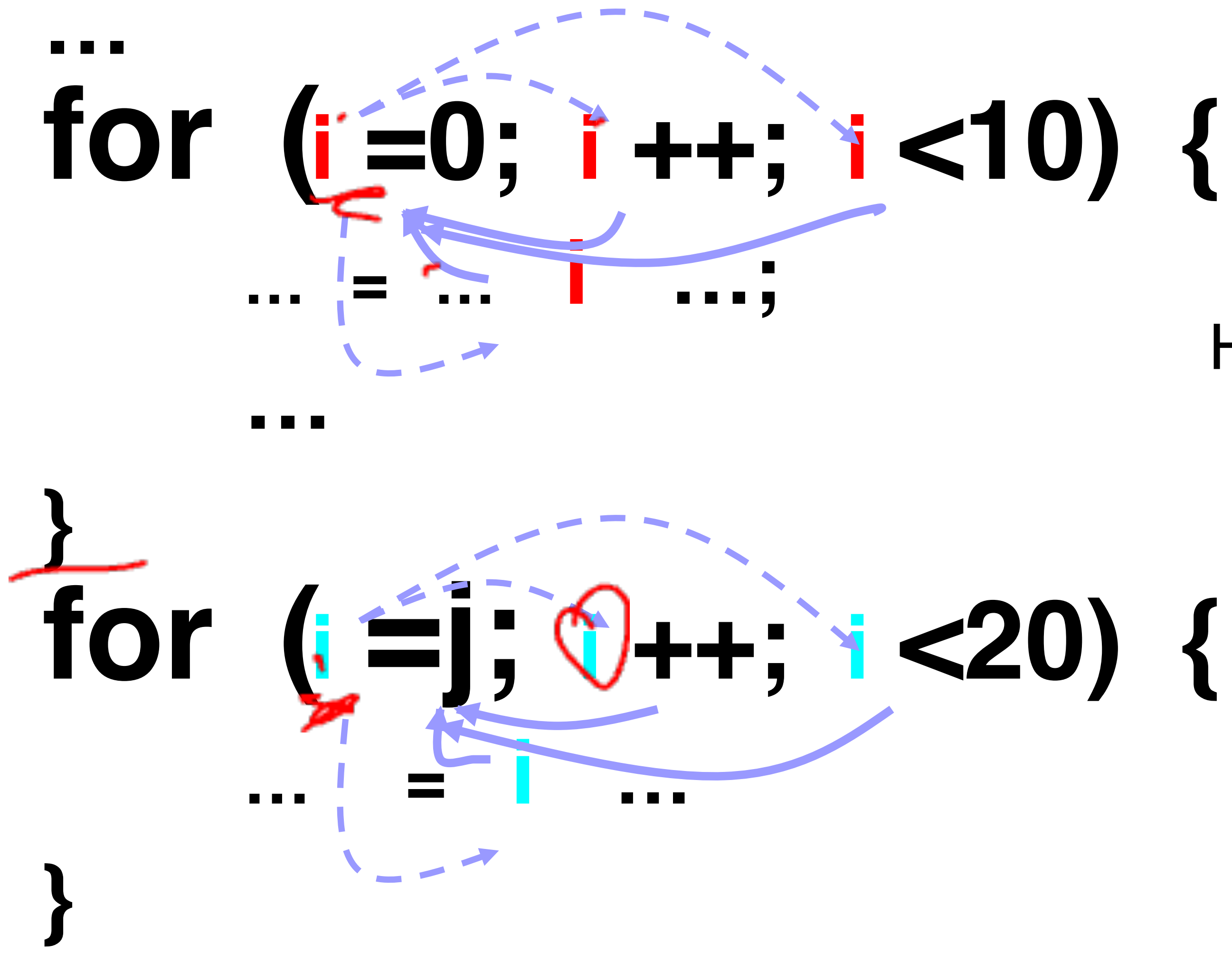
Today

- Iterated Register Allocation
- SSA-Based Register Allocation
  - Def-Use chains
  - SSA
  - ?functions (briefly)
  - Chordal Graphs
  - Perfect Elimination Order

Def-Use Chains

- Common Analysis in support of optimizations, register allocation, etc.
  - Find all the sites where a variable is used
  - Find the definition of a variable in an expression
- Traditional Solution: def-use chains
  - Link each triple defining a variable to all triples that use it
  - Link each use of a variable to its definition

Def-Use Chains



```
…
for (i =0; i ++; i <10) {
      … = … i …;
      …
}
for (i =j; i ++; i <20) {
      … = i …
}
```

How is this related to register allocation?

Unrelated uses of the same variable are mixed together – complicates analysis.

Def-Use chains are expensive

```
foo(int i, int j) {
        ...
        switch (i) {
        case 0: x=3;break;
        case 1: x=1; break;
        case 2: x=6; break;
        case 3: x=7; break;
        default: x = 11;
        }
        switch (j) {
        case 0: y=x+7; break;
        case 1: y=x+4; break;
        case 2: y=x-2; break;
        case 3: y=x+1; break;
        default: y=x+9;
        }
        ...
```

Def-Use chains are expensive

```
foo(int  i,  int  j) {
        ...
        switch  (i)  {
        case  0:  x=3;
        case  1:  x=1;
        case  2:  x=6;
        case  3:  x=7;
        default:  x  =  11;
        }
        switch  (j)  {
        case  0:  y=x+7;
        case  1:  y=x+4;
        case  2:  y=x-2;
        case  3:  y=x+1;
        default:  y=x+9;
        }
        ...
```

x6 ← x

In general,

N defs

M uses

⍰ O(NM) space and time

A solution is to limit each var to ONE def site

Def-Use chains are expensive

```
foo(int i, int j) {
    …
    switch (i) {
    case 0: x=3; break;
    case 1: x=1; break;
    case 2: x=6;
    case 3: x=7;
    default: x = 11;
    }
    x1 is one of the above x's
    switch (j) {
    case 0: y=x1+7;
    case 1: y=x1+4;
    case 2: y=x1-2;
    case 3: y=x1+1;
    default: y=x1+9;
    }
    …
```

A possible solution is to limit each var to ONE def site

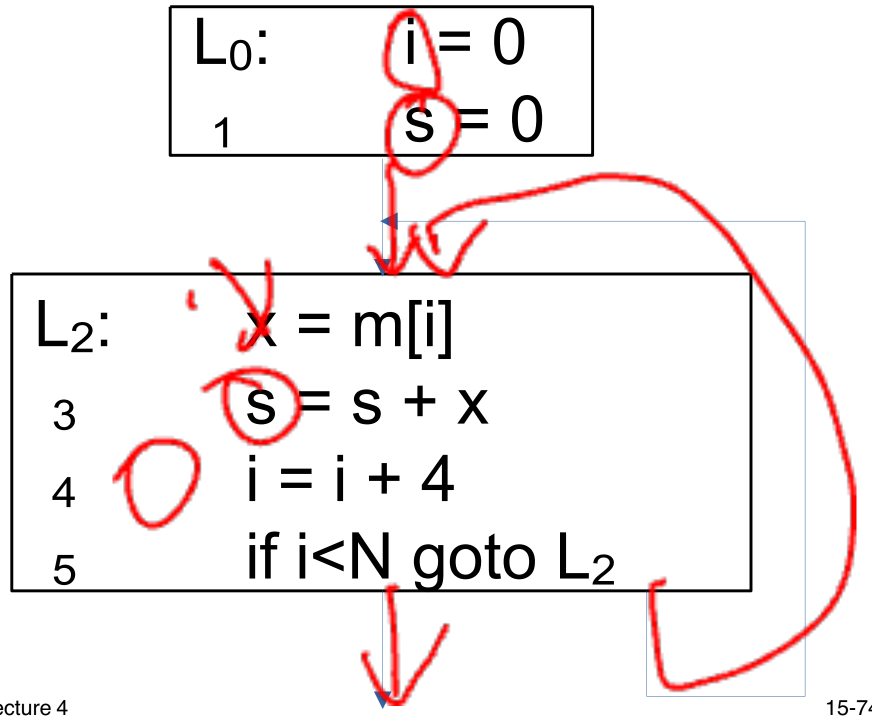
Basic Blocks & Control Flow Graph

- Control Flow
  - what is potential sequence of instructions?
  - Only interested in transfers of control
    - jump
    - conditional jump
    - call
    - label (target of a transfer)

- Group together non-jumps into Basic Block
  - One entry point
  - One point of exit
  - When entered all instructions are executed
- Basic Blocks are nodes in Control Flow Graph

SSA

- Static single assignment is an **IR** where every variable has only ONE definition in the program text
  - single **static** definition
  - (Could be in a loop which is executed dynamically many times.)



```
L0:     i = 0
 1      s = 0

L2:     x = m[i]
 3      s = s + x
 4      i = i + 4
 5      if i<N goto L2
```

Not in SSA form:

- **i** and **s** have two static def sites
- **x** has only one static def site, but may be dynamically defined many times in loop.

SSA

- Static single assignment is an **IR** where every variable has only ONE definition in the program text
  - single **static** definition
  - (Could be in a loop which is executed dynamically many times.)
- Easy for a straight-line code:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow optimizations
  - Easier
  - faster
- Improves register allocation
  - Makes building interference graphs easier
  - Easier register allocation algorithm
  - Decoupling of spill, color, and coalesce
- For most programs reduces space/time requirements

SSA History

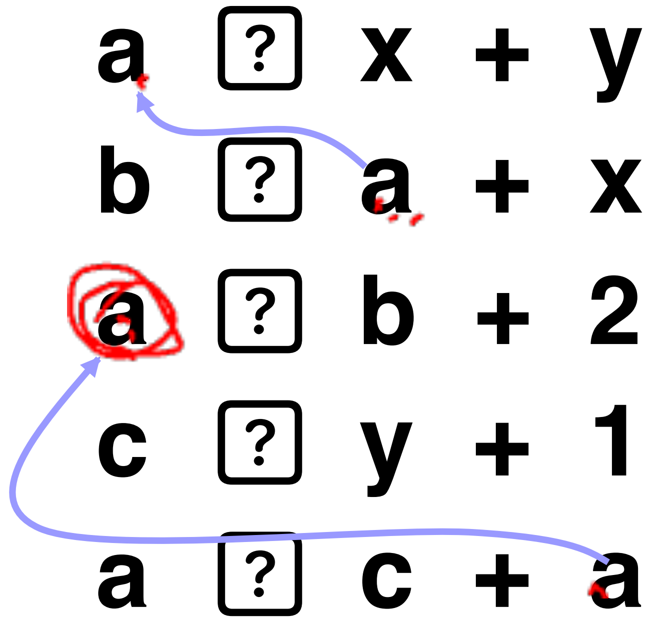
- Developed by Wegman, Zadeck, Alpern, and Rosen in 1988
- Today used in most production compilers, e.g., gcc, llvm, most JIT compilers, …

Straight-line SSA

$$a \ \boxed{?} \ x + y$$
$$b \ \boxed{?} \ a + x$$
$$a \ \boxed{?} \ b + 2$$
$$c \ \boxed{?} \ y + 1$$
$$a \ \boxed{?} \ c + a$$

- Straight forward to convert basic block into SSA
- Connect each use to its most recent definition

Straight-line SSA



```
a ? x + y
b ? a + x
a ? b + 2
c ? y + 1
a ? c + a
```

- Straight forward to convert basic block into SSA
- Connect each use to its most recent definition

Straight-line SSA

a $\boxed{?}$ x + y

b $\boxed{?}$ a + x

a $\boxed{?}$ b + 2

c $\boxed{?}$ y + 1

a $\boxed{?}$ c + a

```
for each variable a:
    count[a] = 0
    Stack[a] = [0]
rename_basic_block(B) =
    for each instruction S in block B:
        for each use of a variable x in S:
            i = top(Stack[x])
            replace the use of x with x_i
        for each variable a that S defines
            count[a] = count[a] + 1
            i = count[a]
            push i onto Stack[a]
            replace definition of a with a_i
```

Straight-line SSA

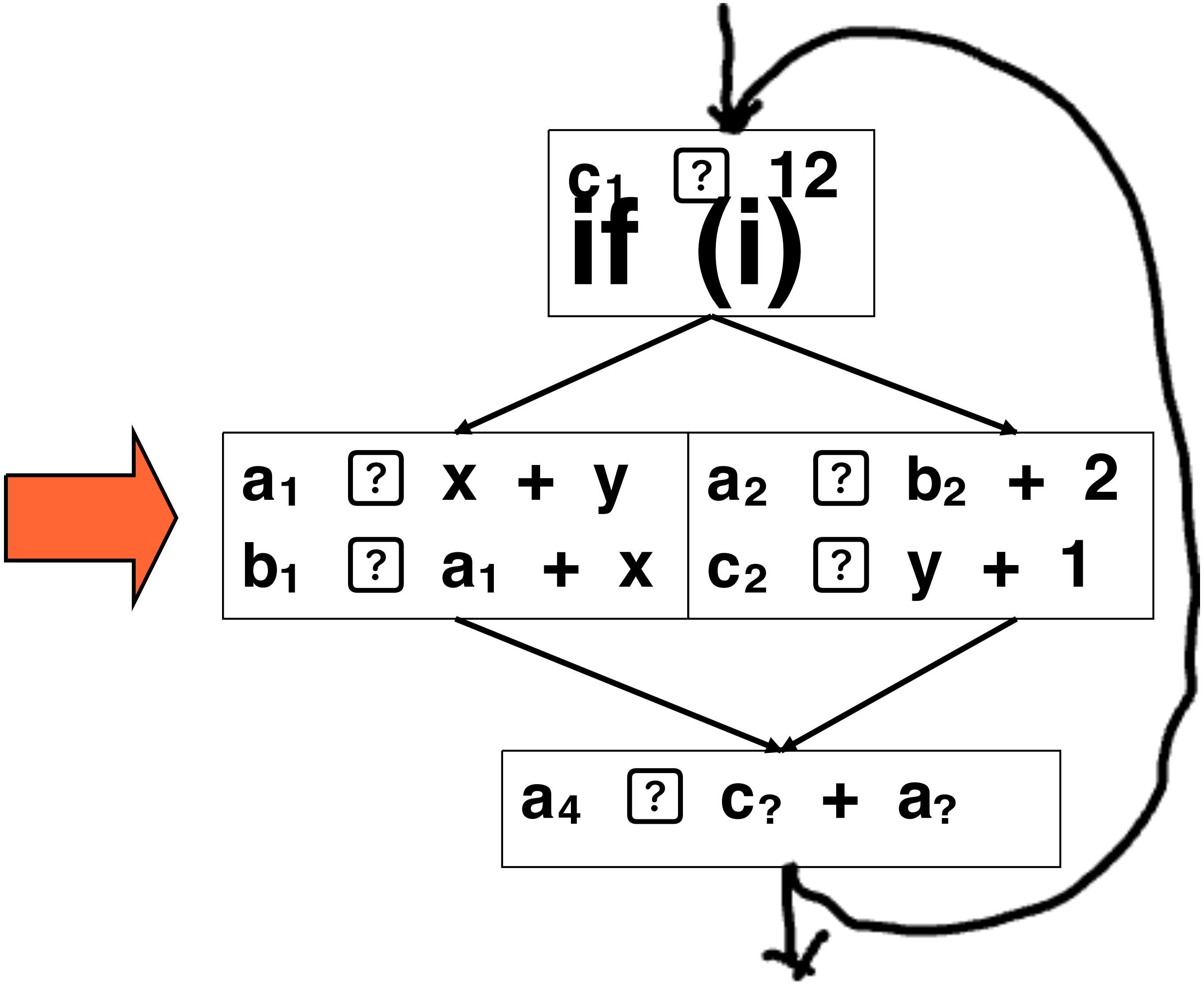
$$
\begin{array}{lllll}
a & \boxed{?} & x & + & y \\
b & \boxed{?} & a & + & x \\
a & \boxed{?} & b & + & 2 \\
c & \boxed{?} & y & + & 1 \\
a & \boxed{?} & c & + & a
\end{array}
\quad \Longrightarrow \quad
\begin{array}{lllll}
\underline{a_1} & \boxed{?} & x & + & y \\
b_1 & \boxed{?} & a_1 & + & x \\
a_2 & \boxed{?} & b_1 & + & 2 \\
c_1 & \boxed{?} & y & + & 1 \\
a_3 & \boxed{?} & c_1 & + & a_2
\end{array}
$$

SSA

- Static single assignment is an **IR** where every variable has only ONE definition in the program text
  - single **static** definition
  - (Could be in a loop which is executed dynamically many times.)
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
- What about at joins in the CFG?

Merging at Joins

```
{
 c ⍰ 12
 if (i) {
        a ⍰ x
    + y
        b ⍰ a
    + x
 } else {
        a ⍰ b
    + 2
        c ⍰ y
}   + 1
 }
 a ⍰ c + a
```



SSA

- Static single assignment is an **IR** where every variable has only ONE definition in the program text

    single **static** definition

    (Could be in a loop which is executed dynamically many times.)

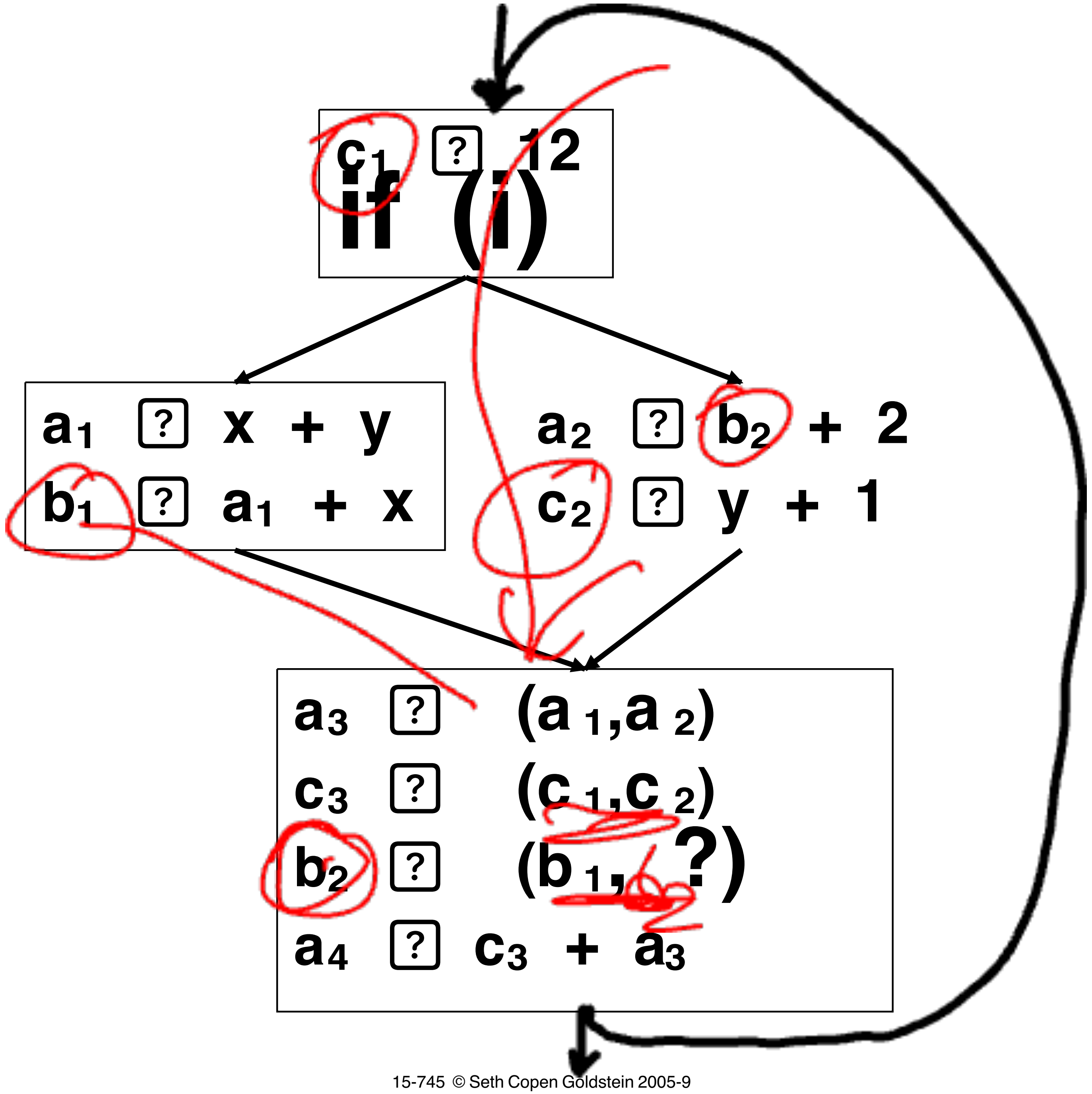
- Easy for a basic block:

    assign to a fresh variable at each stmt.

    Each use uses the most recently defined var.

- What about at joins in the CFG?
- Use notional fiction: -functions

Merging at Joins



```
c1 ? 12
if (i)

a1 ? x + y
b1 ? a1 + x

a2 ? b2 + 2
c2 ? y + 1

a3 ? (a 1,a 2)
c3 ? (c 1,c 2)
b2 ? (b 1, ?)
a4 ? c3 + a3
```

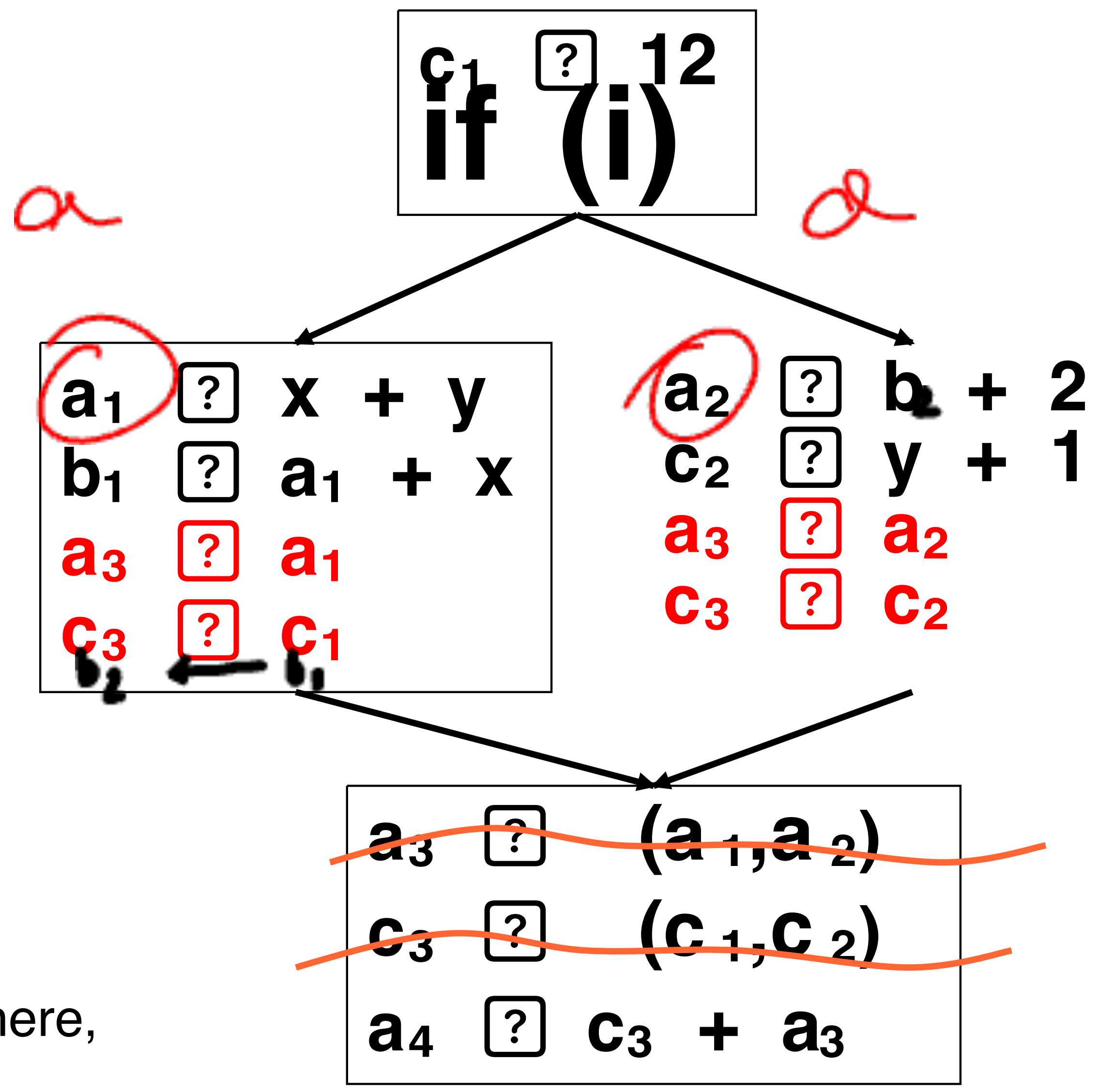
The function

- merges multiple definitions along multiple control paths into a single definition.
- At a BB with p predecessors, there are p arguments to the function.

  x_{new} $\boxed{?}$ $(x_1, x_2, x_3, \ldots, x_p)$

- How do we choose which x_i to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

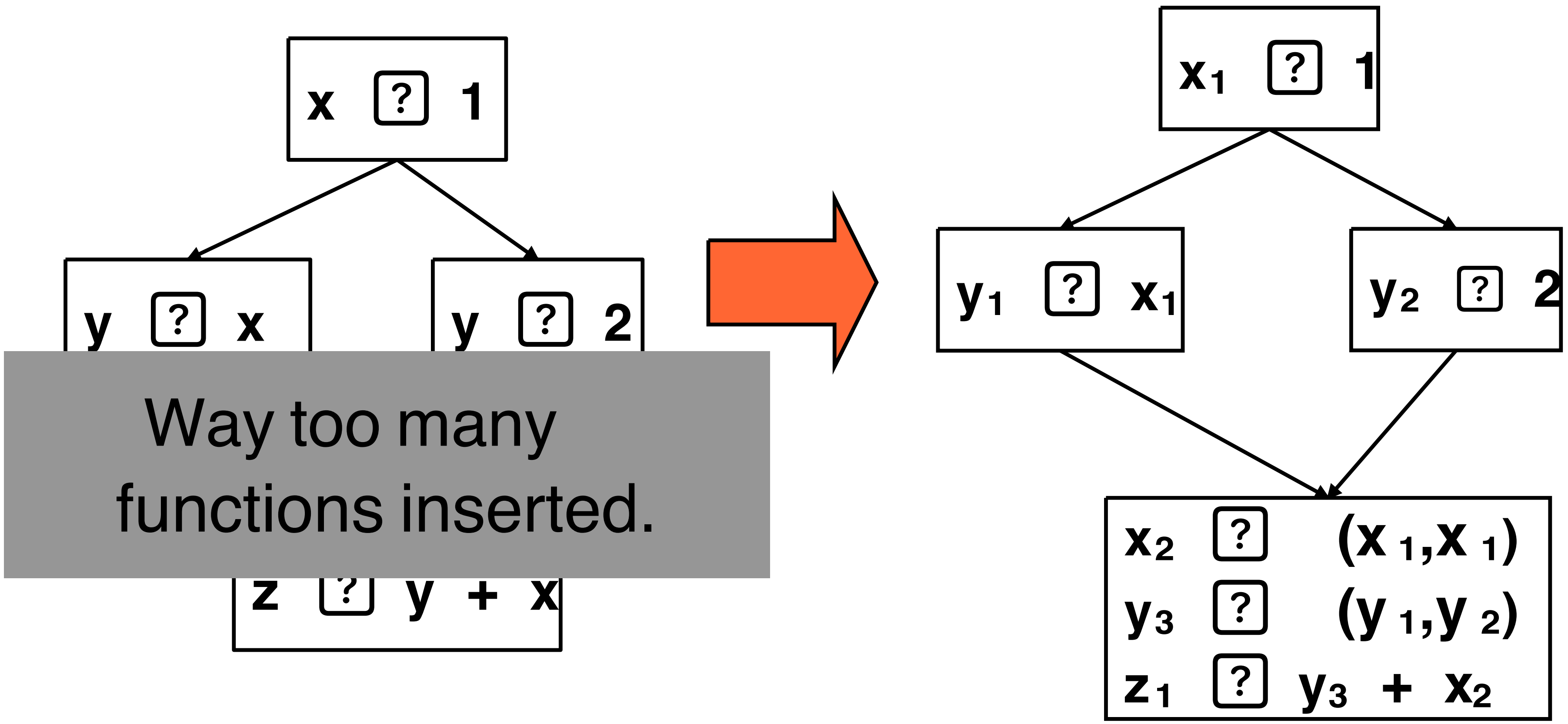
"Implementing" *



*Huge caveat here, discussed later. (e.g, lost-copy, swap-problem)

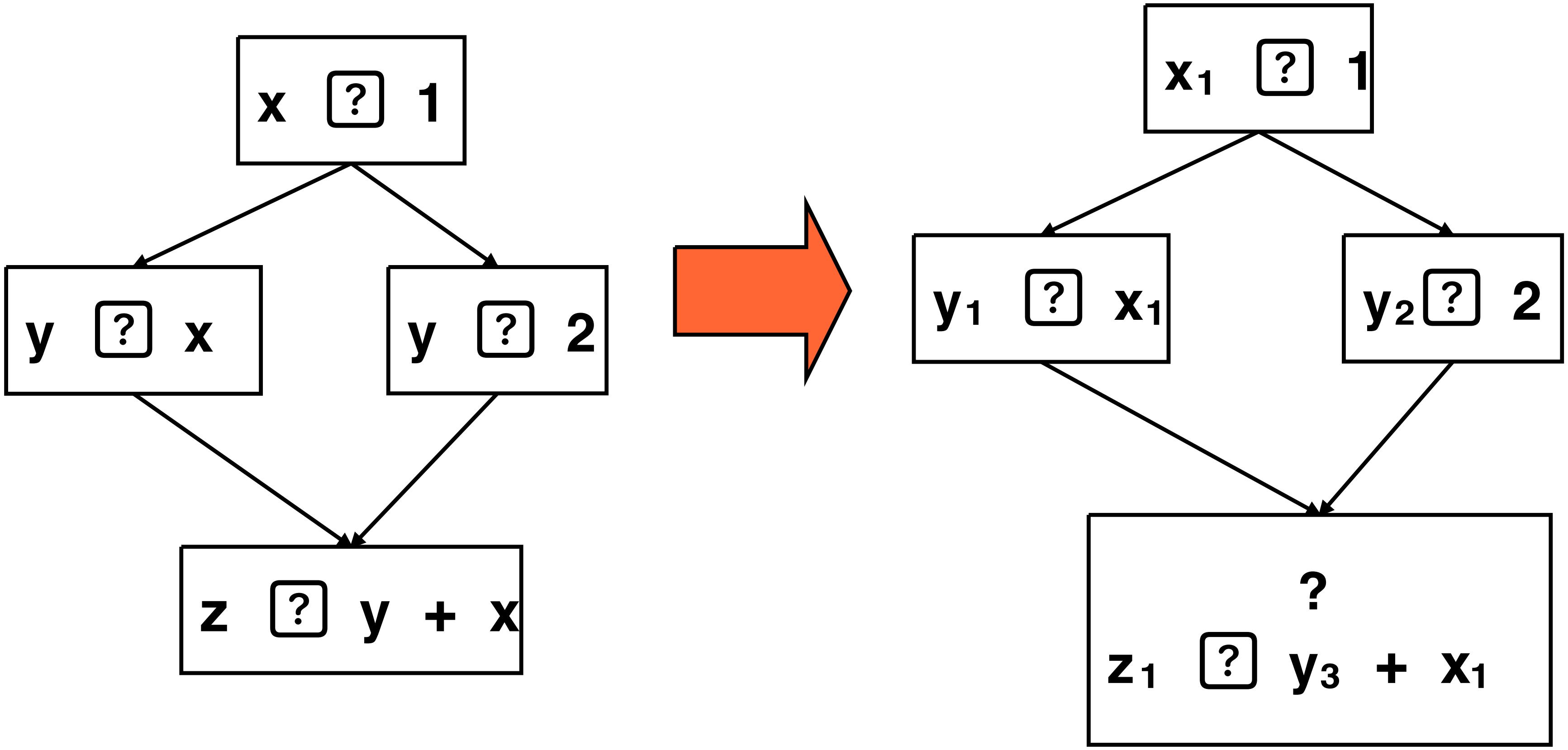
Trivial SSA

- Each assignment generates a fresh variable.
- At each join point insert functions for all live variables.



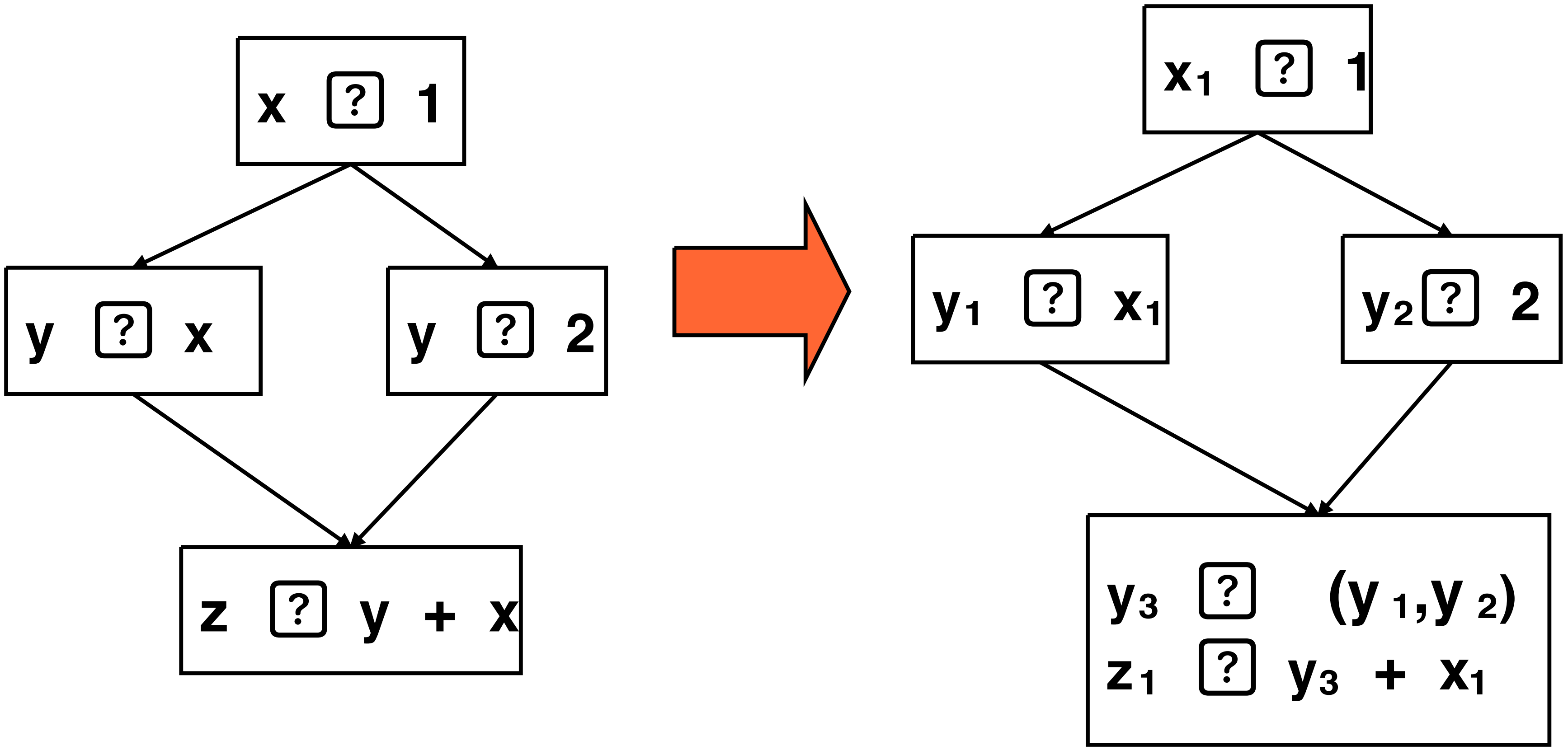
Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert functions for all variables with multiple outstanding defs.



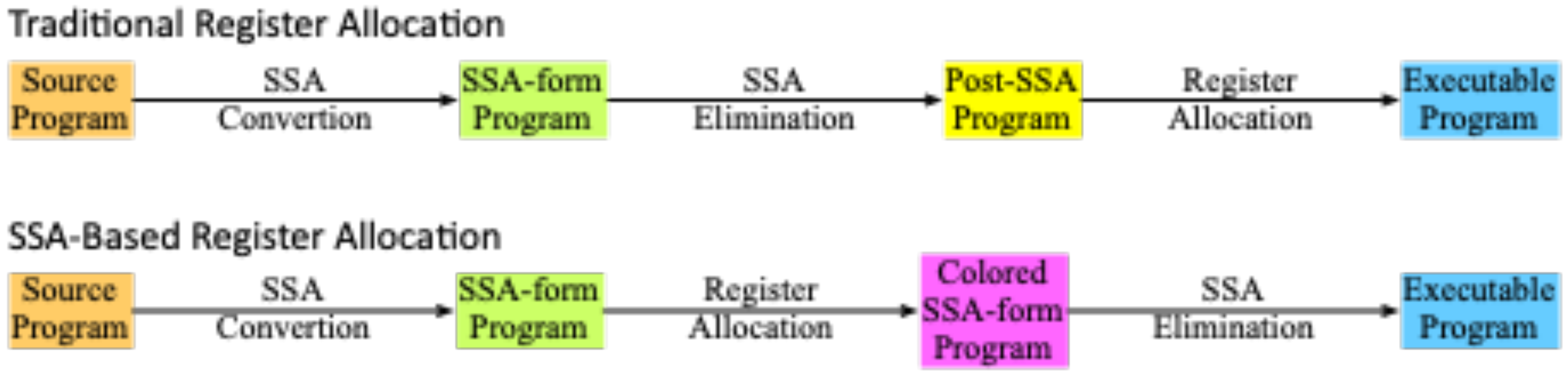
Minimal SSA

- ⍰ Each assignment generates a fresh variable.
- ⍰ At each join point insert   functions for all variables with multiple outstanding defs.

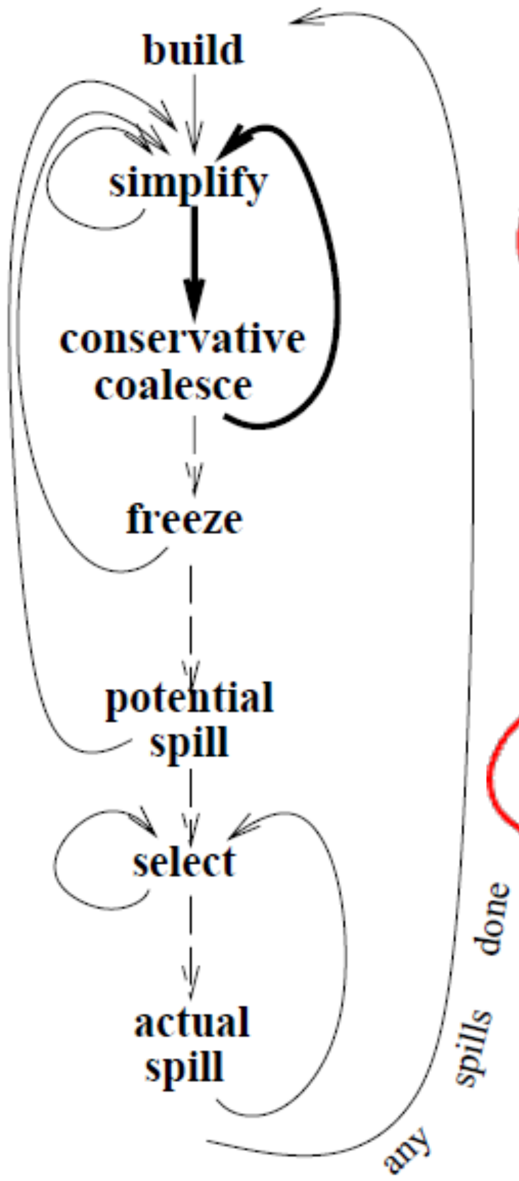


SSA-based Register Allocation

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



Basis for Coloring Approach



Simplify – creating order in which to color nodes

Select – Uses "simplify" order to color nodes

Need heuristic because minimal coloring of general graph is NP-complete

Simplify/Select: A particular order

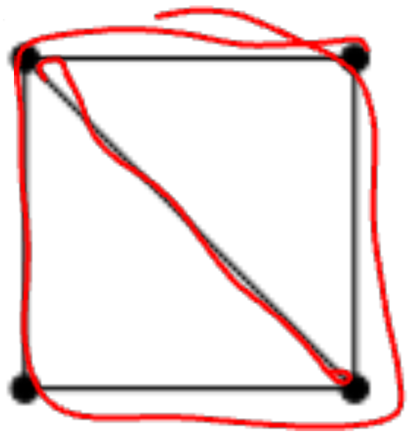
- (G): the number of colors used to color G
- N(v): the neighbors of v

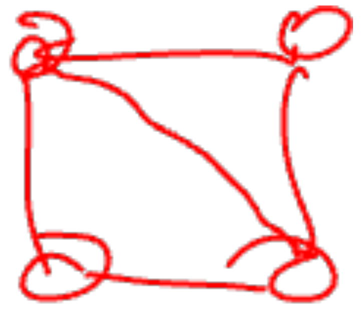
- Greedy Coloring:

```
    input:  G=(V,E)
            an ordered sequence v_1,…, v_n
    output: Assignment  col:V ? {0, …,    (G)}
  for i ? 1 to n do
     let c be lowest color not used in N(v_i)
     set col(v_i) ? c
```

Chordal Graphs

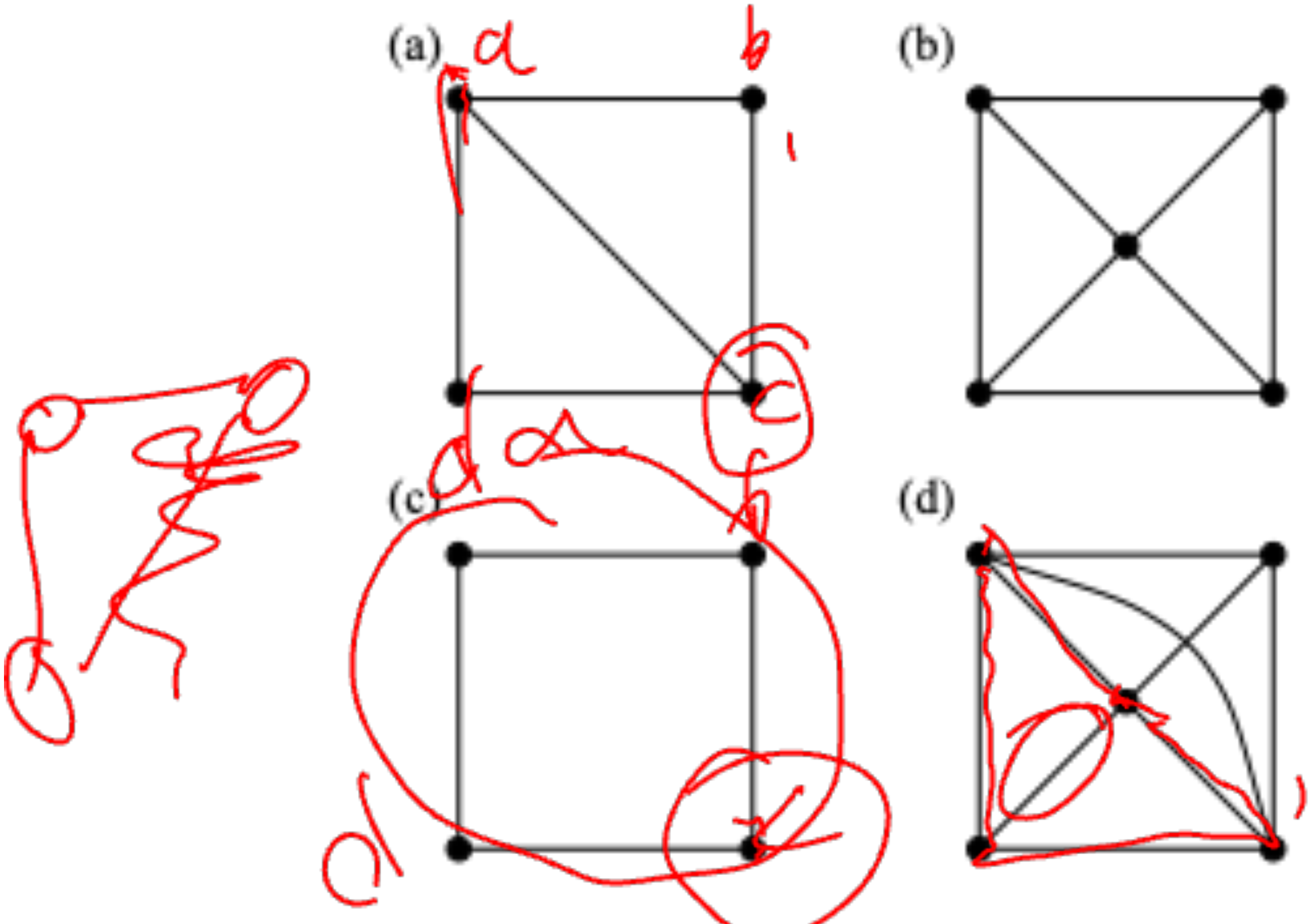
- An undirected graph is chordal if every cycle of 4 or more nodes has a chord.
- A chord is an edge the connects two vertices in the cycle, but is not part of the cycle.





Chordal Graphs

- An undirected graph is chordal if every cycle of 4 or more nodes has a chord.

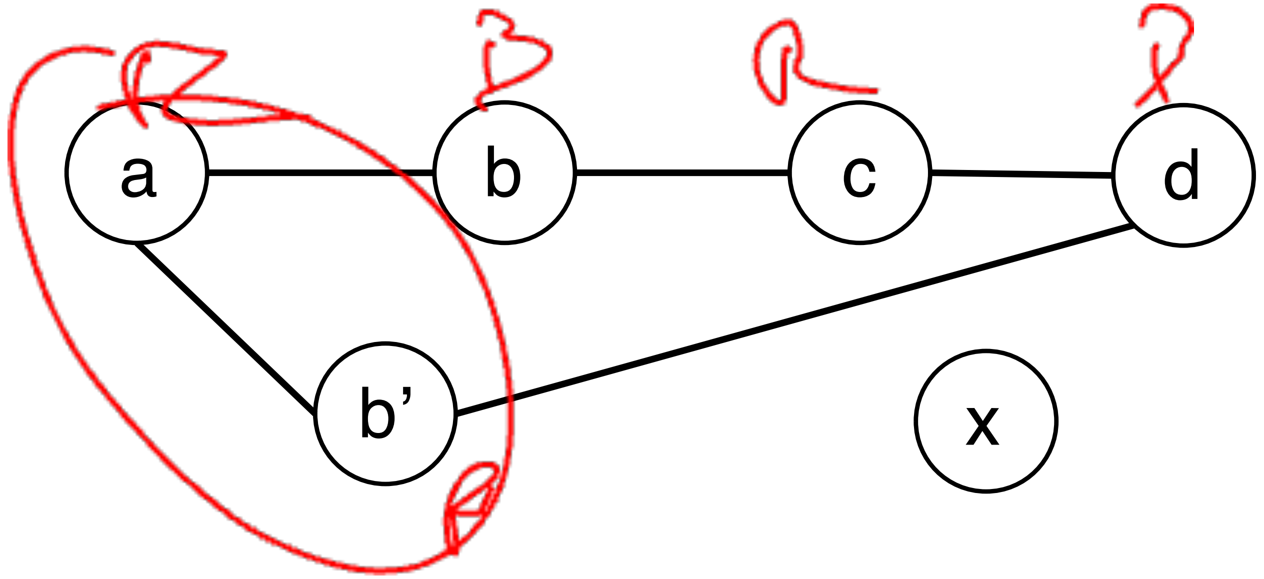


Graph Facts

- Clique: fully connected subgraph
- Chromatic number of graph G: minimal k such that G is k-colorable
- chromatic number of G ⍰ size of largest clique
- Perfect graph: chromatic number = size of largest clique
- All chordal graphs are perfect
- Can color perfect graph in poly-time
- Finally, IG of SSA programs is chordal!

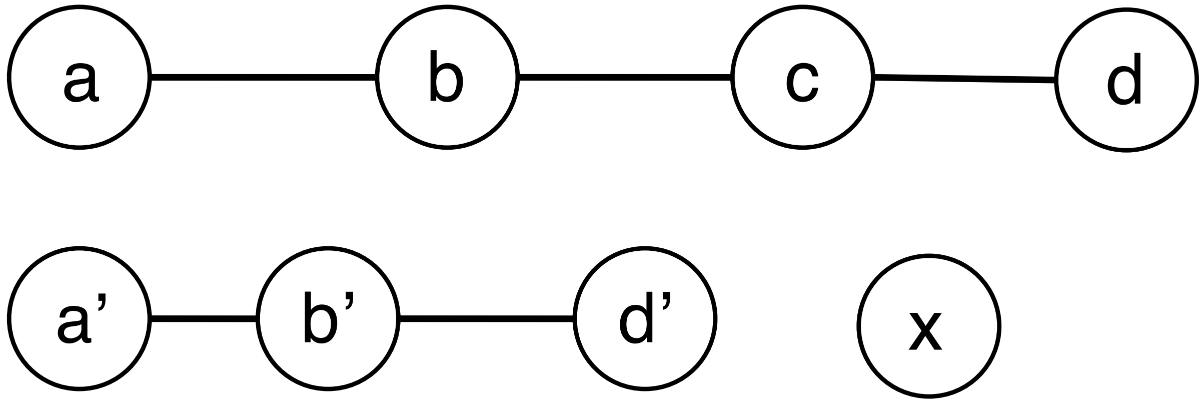
Non-chordal example

```
a  ⍰   0
b  ⍰   1
c  ⍰  a + b
d  ⍰  b + c
a  ⍰  c + d
b' ⍰  7
d  ⍰  a + b'
x  ⍰  b'+ d
      ret x
```



Break up the live ranges

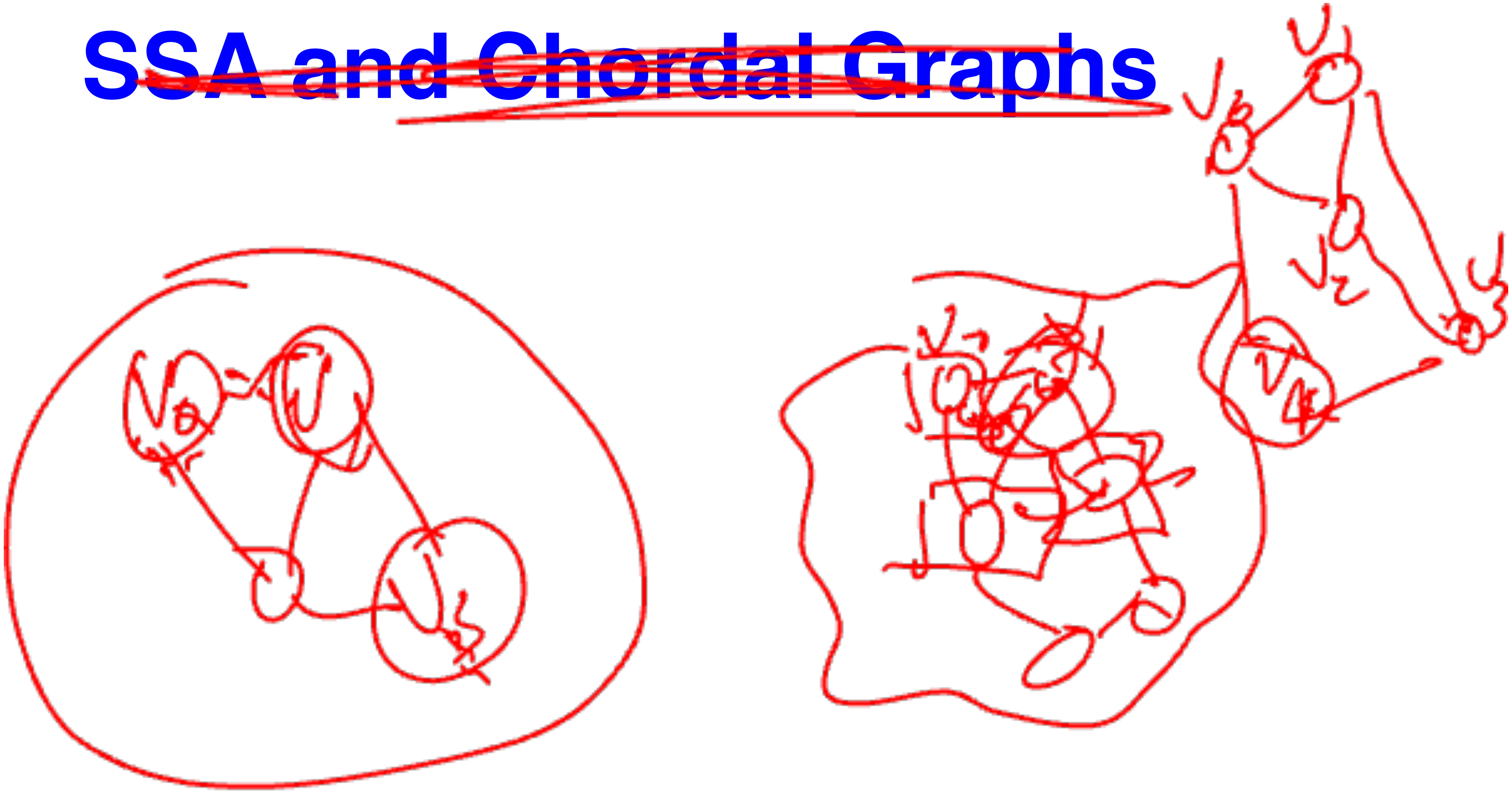
a $\boxed{?}$ 0
b $\boxed{?}$ 1
c $\boxed{?}$ a + b
d $\boxed{?}$ b + c
a’ $\boxed{?}$ c + d
b’ $\boxed{?}$ 7
d’ $\boxed{?}$ a’ + b’
x $\boxed{?}$ b’+ d’
ret x

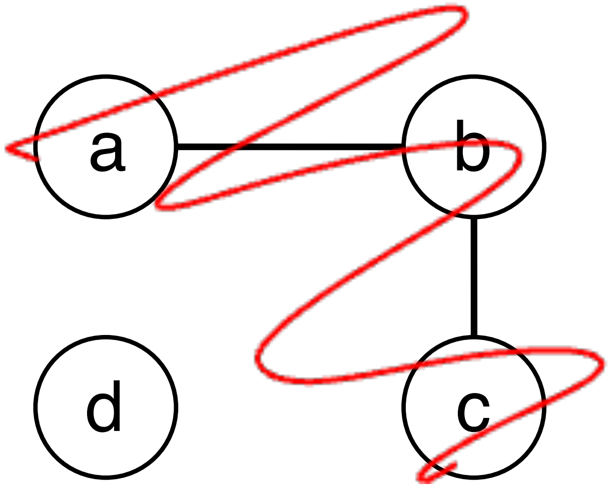


Adding more temps $\boxed{?}$ fewer registers!

BTW: now in SSA-form!

~~SSA and Chordal Graphs~~

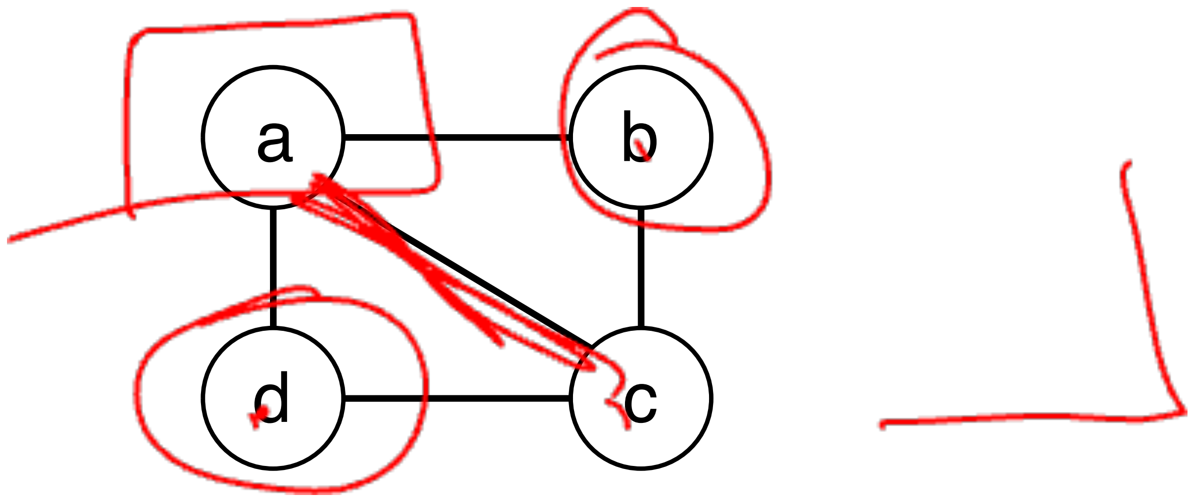




V_0

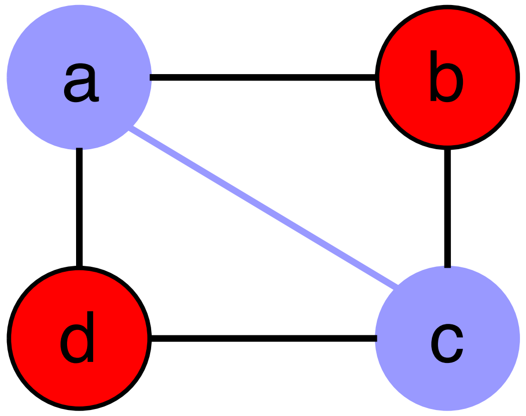
Simplical Elimination Ordering

- 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.
- b & d are simplical



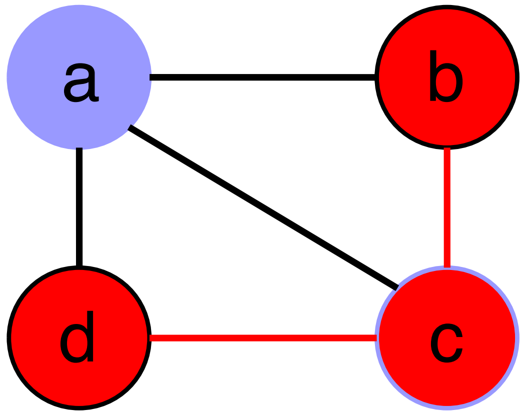
Simplical Elimination Ordering

- 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.
- b & d are simplical



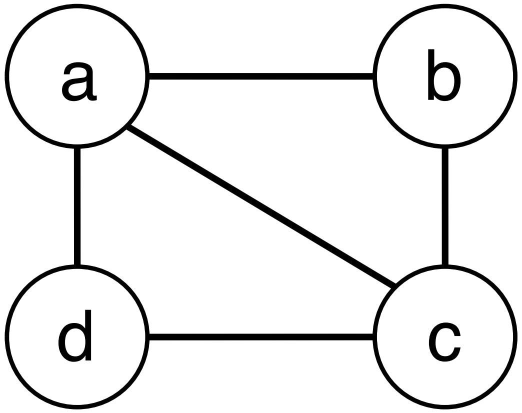
Simplical Elimination Ordering

- 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.
- b & d are simplical
- a & c are not



Simplical Elimination Ordering

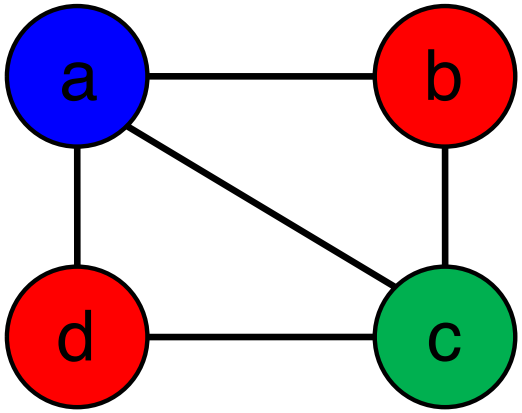
- If G = (V, E) is a graph, then a vertex $v \in V$ is called *simplicial* if, and only if, its neighborhood in G is a clique.
- A *Simplicial Elimination Ordering* of G is a bijection $\sigma: V(G) \rightarrow \{1, \dots, |V|\}$, such that every vertex v_i is a simplicial vertex in the subgraph induced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



b, a, c, d

Greedy Coloring using SEO is optimal

- 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 $\sigma: V(G) \rightarrow \{1, \ldots, |V|\}$, such that every vertex v_i is a simplicial vertex in the subgraph induced by $\{v_1, \ldots, v_i\}$.



b, a, c, d

Maximal Cardinality Search

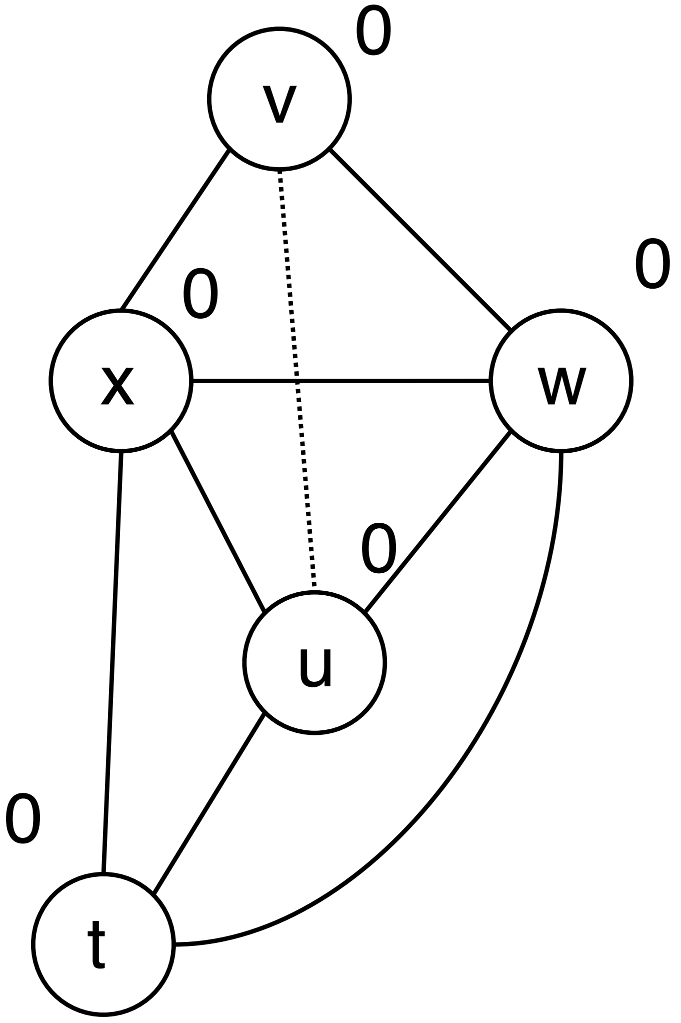
Use Maximum Cardinality Search to generate SEO

Maximum Cardinality Search

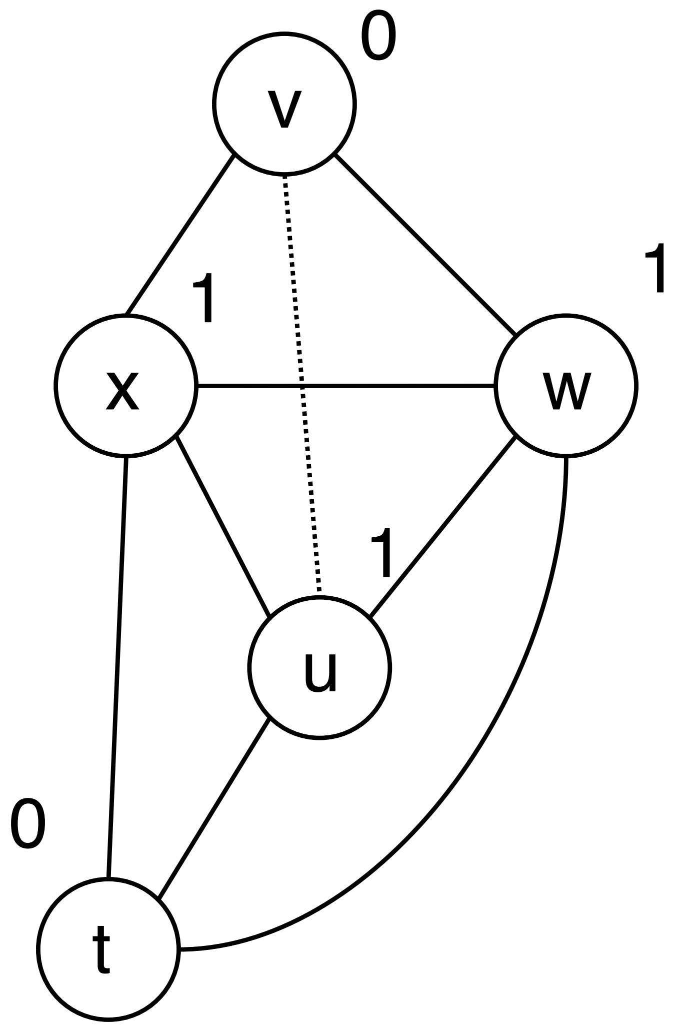
```
input: $G = (V, E)$ with $|V| = n$
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    let $v \in V$ be a node such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ in
        $\sigma(i) \leftarrow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

Running Time: $O(|V|+|E|)$

v	$\boxed{?}$	1
w	$\boxed{?}$	v + 3
x	$\boxed{?}$	w + v
u	$\boxed{?}$	v
t	$\boxed{?}$	u + x
	$\boxed{?}$	w
	$\boxed{?}$	t
	$\boxed{?}$	u

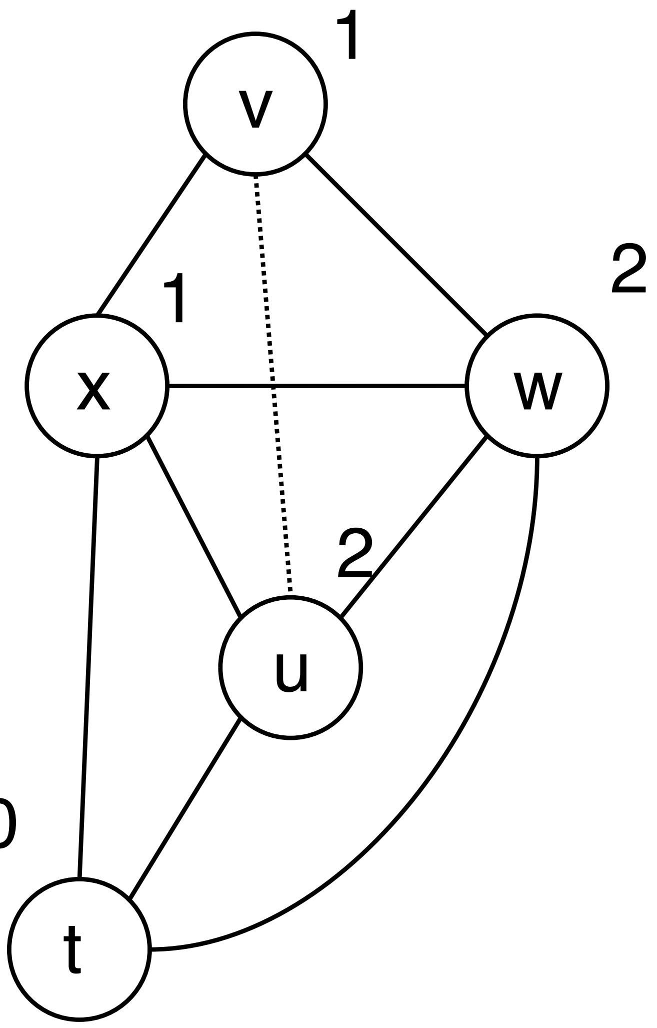


```
v  ⍰  1
w  ⍰  v + 3
x  ⍰  w + v
u  ⍰  v
t  ⍰  u + x
   ⍰  w
   ⍰  t
   ⍰  u
```



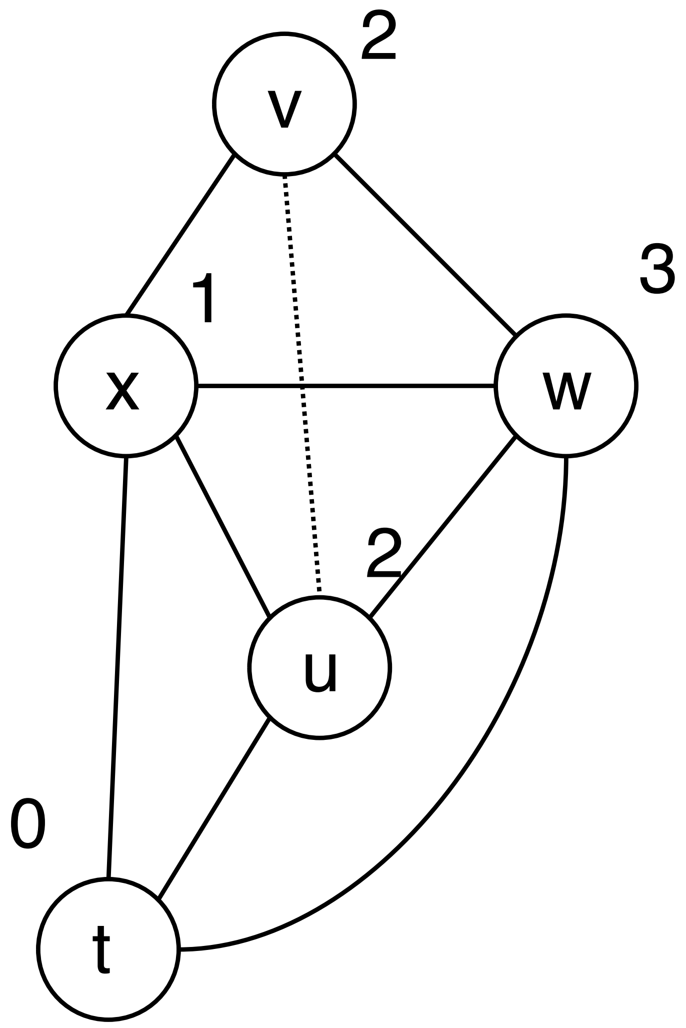
SEO: t

```
v ⍰  1
w ⍰  v + 3
x ⍰  w + v
u ⍰  v
t ⍰  u + x
  ⍰  w
  ⍰  t
  ⍰  u
```



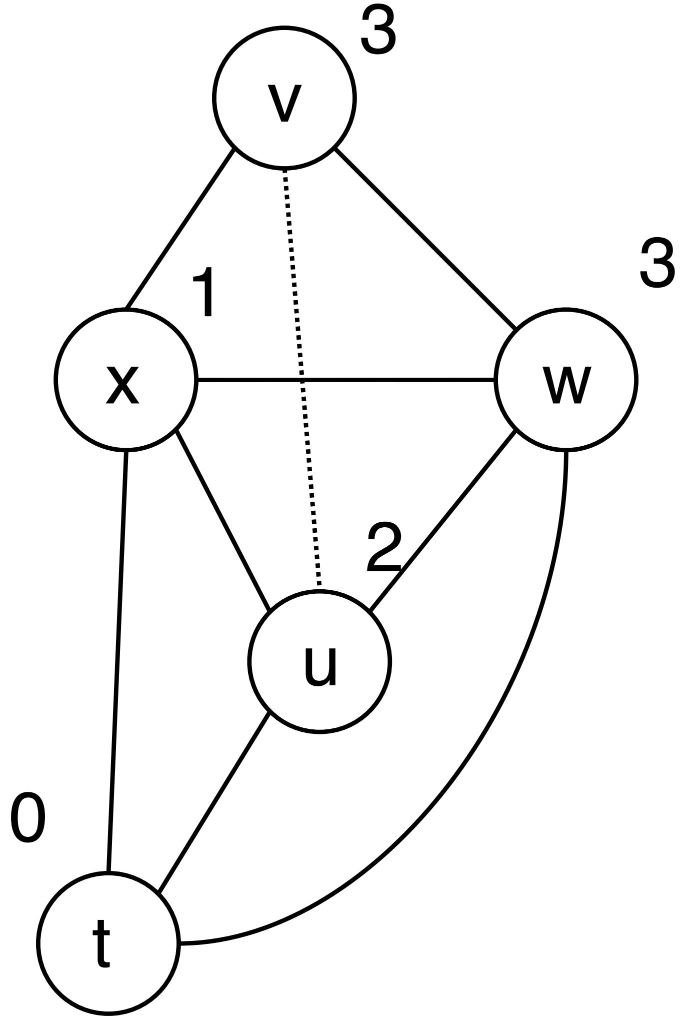
SEO: t, x

```
v ⍰  1
w ⍰  v + 3
x ⍰  w + v
u ⍰  v
t ⍰  u + x
  ⍰  w
  ⍰  t
  ⍰  u
```



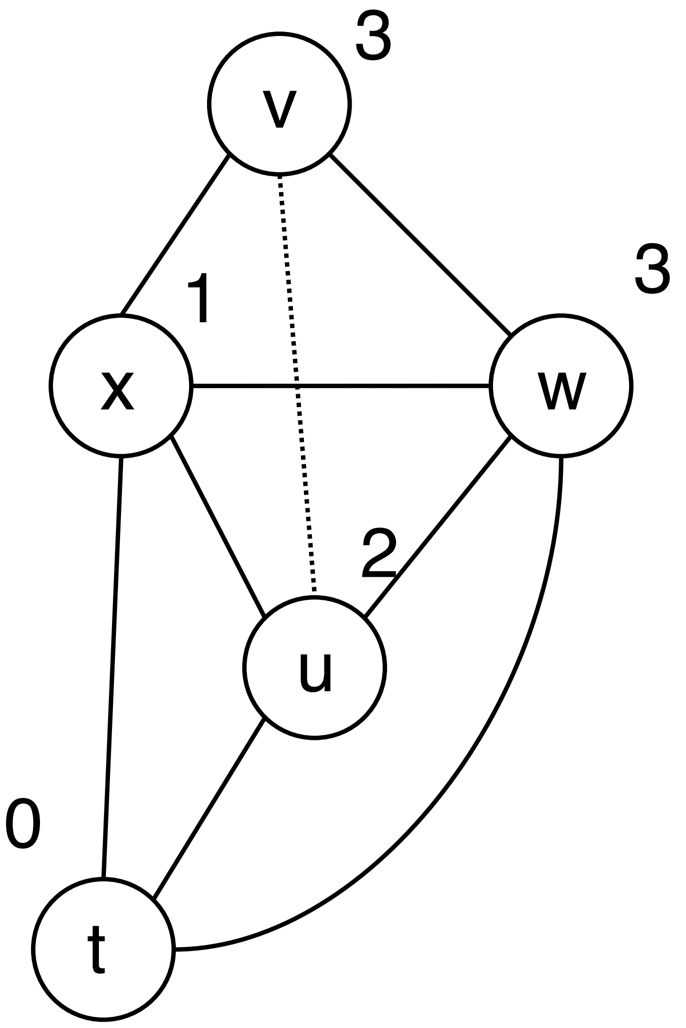
SEO: t, x, u

```
v  ⍰   1
w  ⍰   v + 3
x  ⍰   w + v
u  ⍰   v
t  ⍰   u + x
   ⍰   w
   ⍰   t
   ⍰   u
```



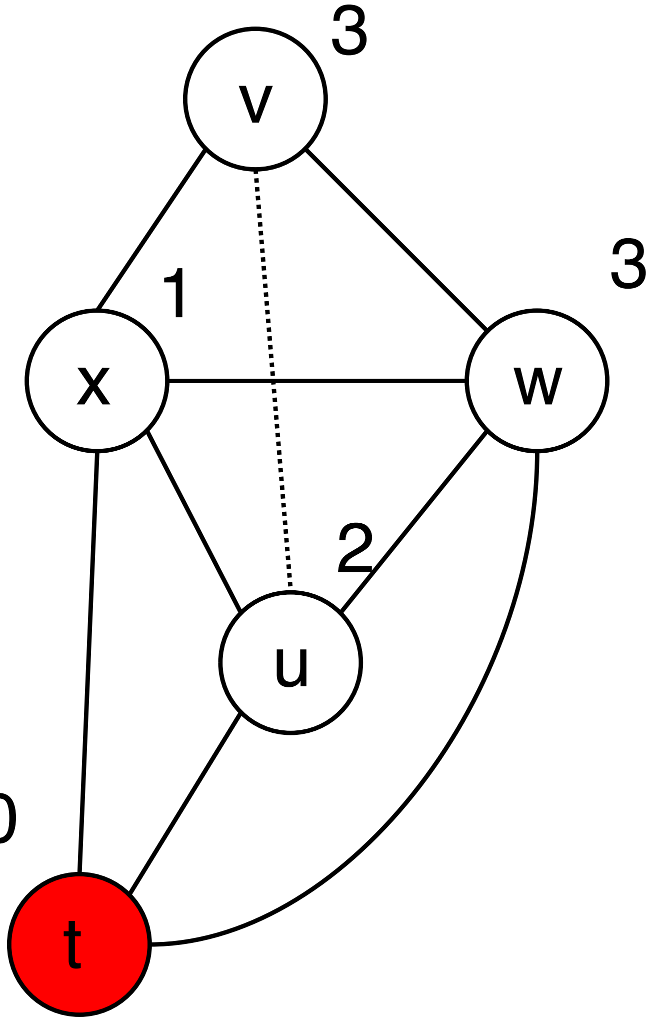
SEO: t, x, u, w

```
v  ⍰   1
w  ⍰   v + 3
x  ⍰   w + v
u  ⍰   v
t  ⍰   u + x
   ⍰   w
   ⍰   t
   ⍰   u
```



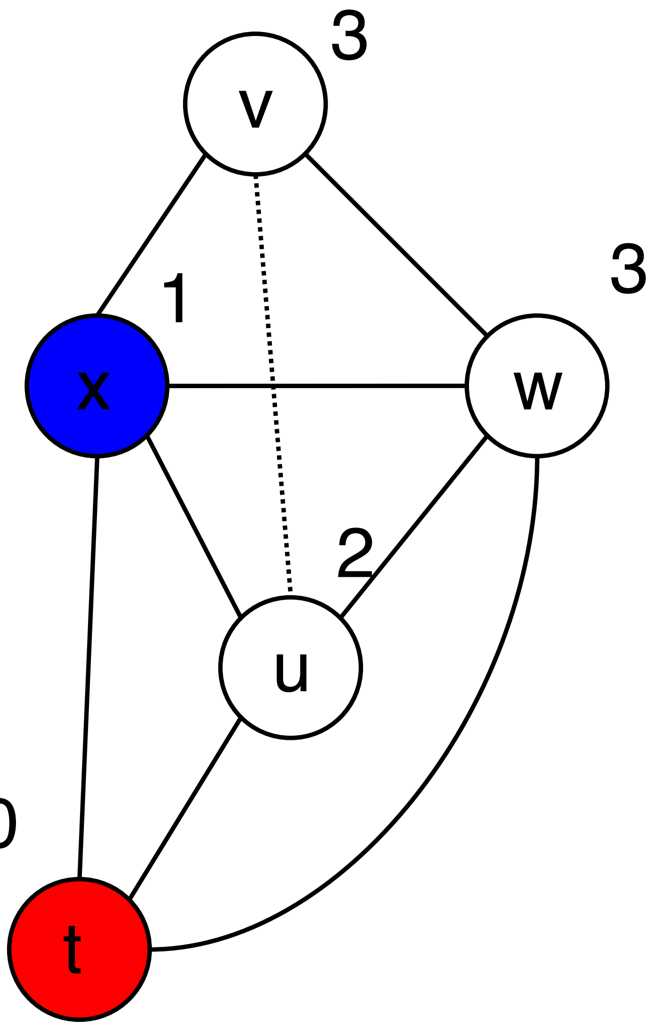
SEO: t, x, u, w, v

```
v  ⍰  1
w  ⍰  v + 3
x  ⍰  w + v
u  ⍰  v
t  ⍰  u + x
   ⍰  w
   ⍰  t
   ⍰  u
```



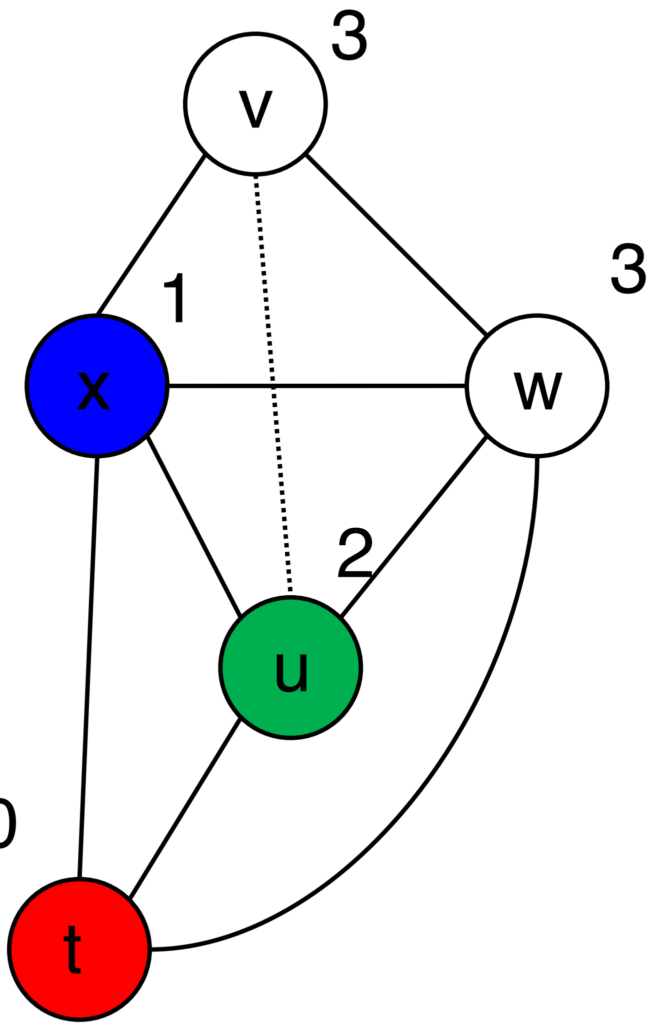
SEO: t, x, u, w, v

```
v ⍰  1
w ⍰  v + 3
x ⍰  w + v
u ⍰  v
t ⍰  u + x
  ⍰  w
  ⍰  t
  ⍰  u
```



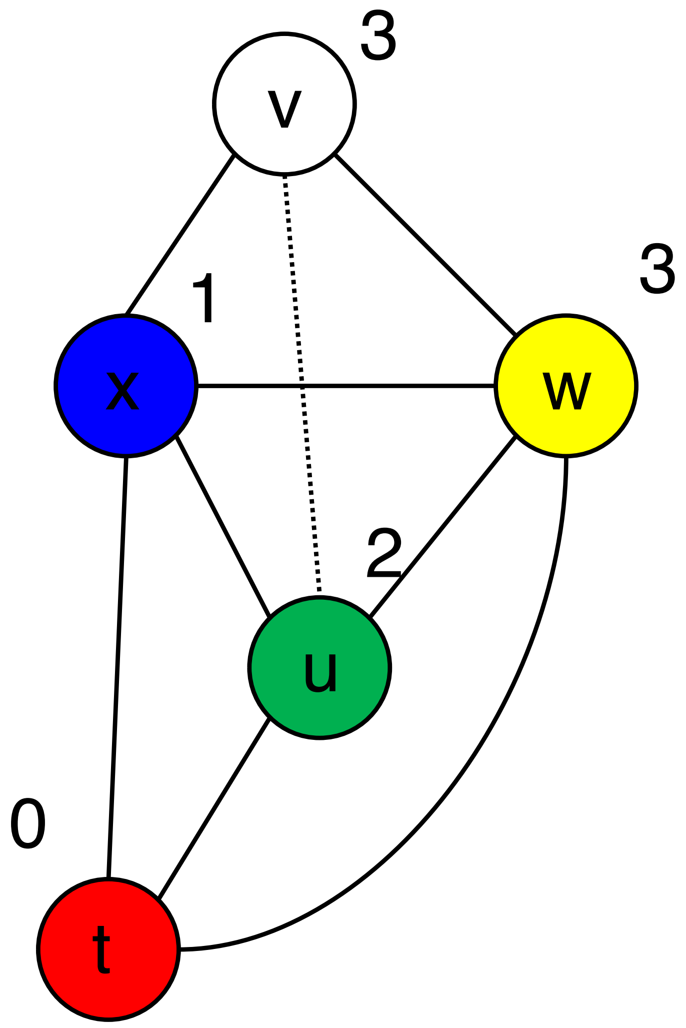
SEO: t, x, u, w, v

```
v  ⍰  1
w  ⍰  v + 3
x  ⍰  w + v
u  ⍰  v
t  ⍰  u + x
   ⍰  w
   ⍰  t
   ⍰  u
```



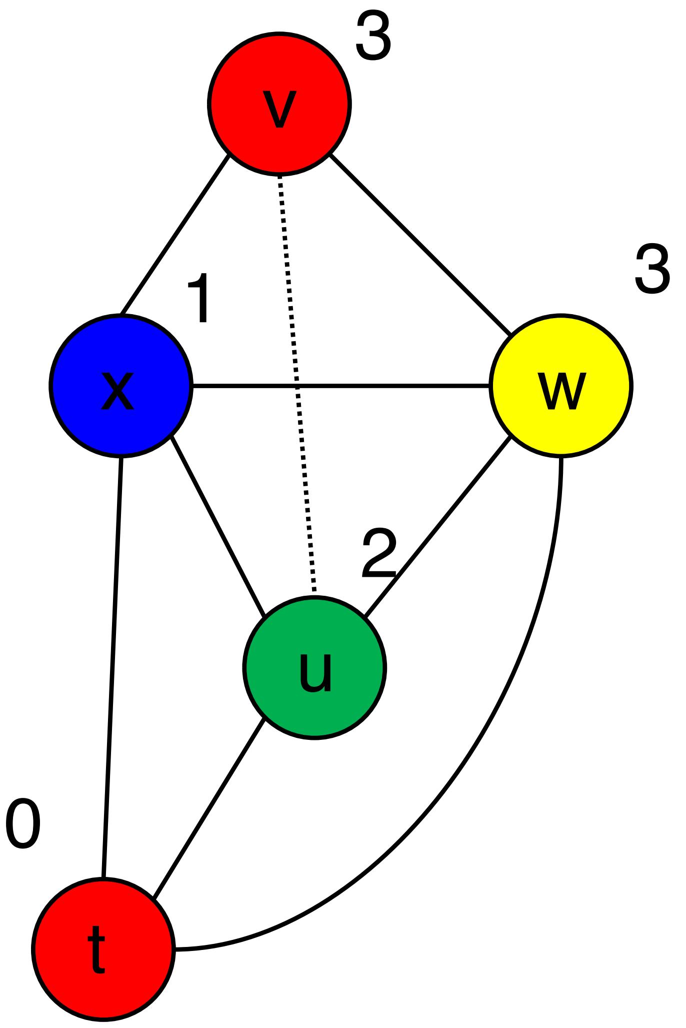
SEO: t, x, u, w, v

```
v ⍰  1
w ⍰  v + 3
x ⍰  w + v
u ⍰  v
t ⍰  u + x
  ⍰  w
  ⍰  t
  ⍰  u
```



SEO: t, x, u, w, v

```
v  ⍰  1
w  ⍰  v + 3
x  ⍰  w + v
u  ⍰  v
t  ⍰  u + x
   ⍰  w
   ⍰  t
   ⍰  u
```



SEO: t, x, u, w, v

Using the SEO is optimal

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, when we color the i th node this ordering, all the neighbors of v_i that have been already colored form a clique.
- All the nodes in a clique must receive different colors.
- Thus, if v_i has M neighbors already colored, we will have to give it color M+1.

I.e., The chromatic number of a chordal graph is the size of largest clique

An advantage of SSA-based RA

- No longer need to iterate
- Instead:
  - Decoupled Spilling
  - Use SEO greedy coloring
  - Do best effort coalescing

Decoupling Coloring and Spilling

- In iterated register coloring we iterate for both coalescing and spilling.
- With chordal register coloring we can use a decoupled approach.

    find maximum clique, C, in IG

    Spill until |C| <= K

    Use MCS to find the SEO

    Color graph greedily

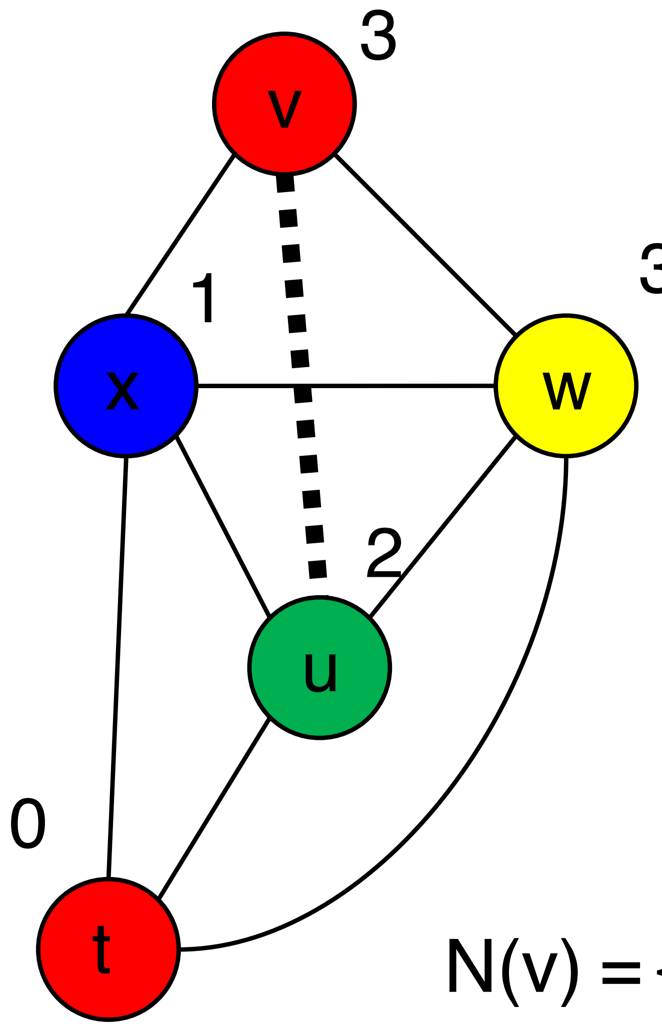
    Perform BestEffortCoalescing

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 ∪ 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'
```

Can we Coalesce?

```
v ⍰ 1
w ⍰ v + 3
x ⍰ w + v
u ⍰ v
t ⍰ u + x
  ⍰ w
  ⍰ t
  ⍰ u
```

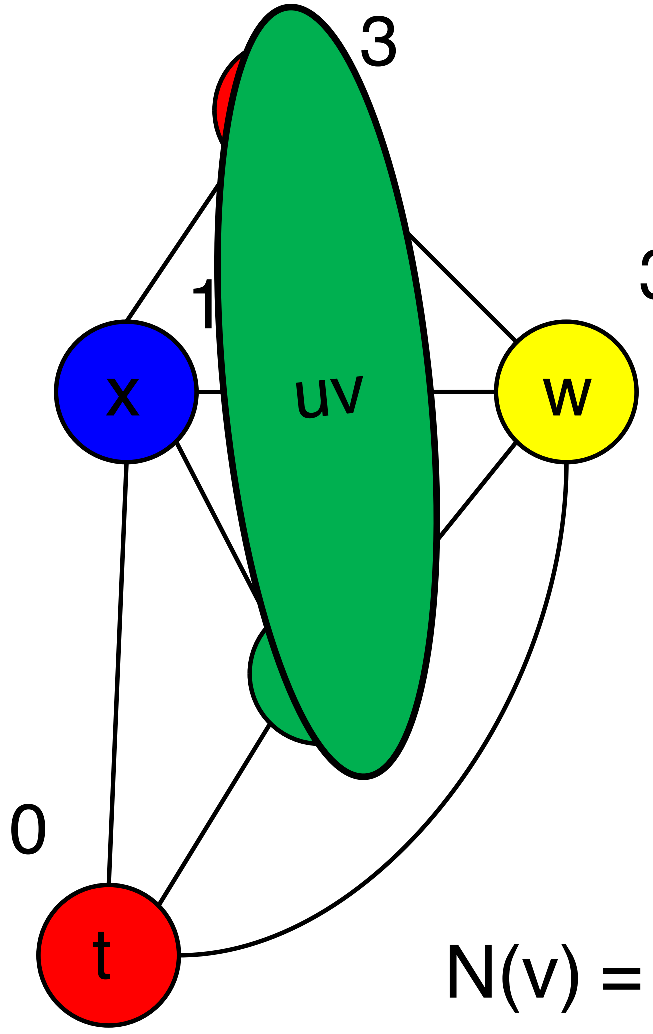


N(v) = { x, w }

N(u) = { x, w, t }

Can we Coalesce?

```
v  ?  1
w  ?  v + 3
x  ?  w + v
u  ?  v
t  ?  v + x
   ?  w
   ?  t
   ?  v
```



N(v) = { x, w }

N(u) = { x, w, t }

In practice

- pre-colored nodes break chordality
- Often assuming chordal is ok
- Have to get out of SSA sometime
- You will use SSA anyway, so register allocation on SSA seems logical
- Will revisit later
- For L1:
  - Can use basic renaming to get into SSA
  - Then, spill, color, coalesce