Register Allocation – 2
SSA-based Register Allocation

15-411/15-611 Compiler Design

Seth Copen Goldstein

September 8, 2019

Today

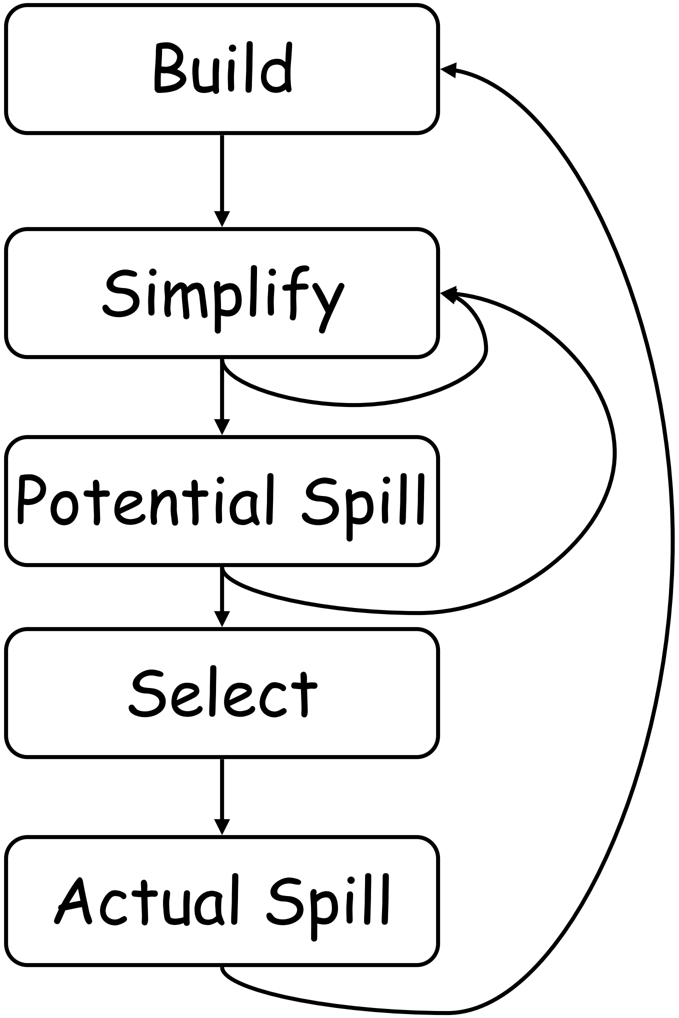
- Iterated Register Allocation
  - Conservative Coalescing
  - Special registers
  - Implementation
- SSA-Based Register Allocation
  - SSA
  - ϕ-functions
  - Chordal Graphs
  - Perfect Elimination Order

Chaitin's allocator

- Build: construct the interference graph
- Simplify: node removal, a la Kempe
- Spill: if necessary, remove a degree≥K node, marking it as a potential spill
- Select: rebuild the graph, coloring as we go
  - if a potential spill can't be colored, mark it as an actual spill
- if there are actual spills

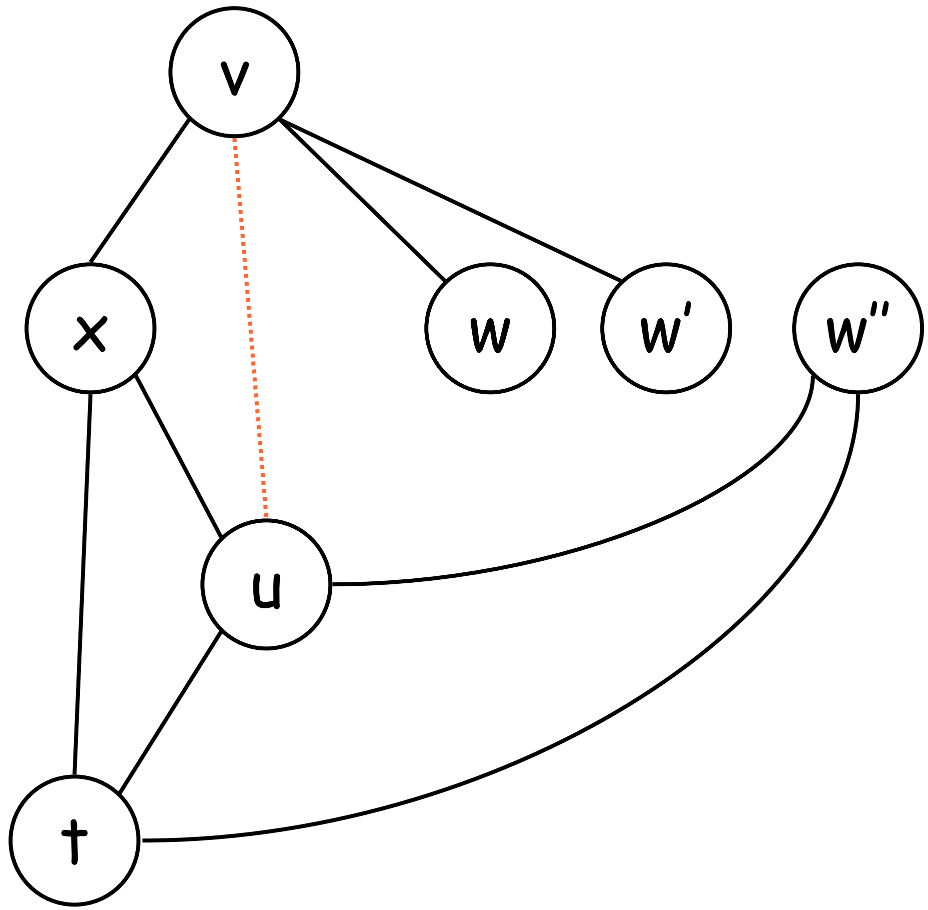
  → generate spill code and start over
- else, done

Where We Are



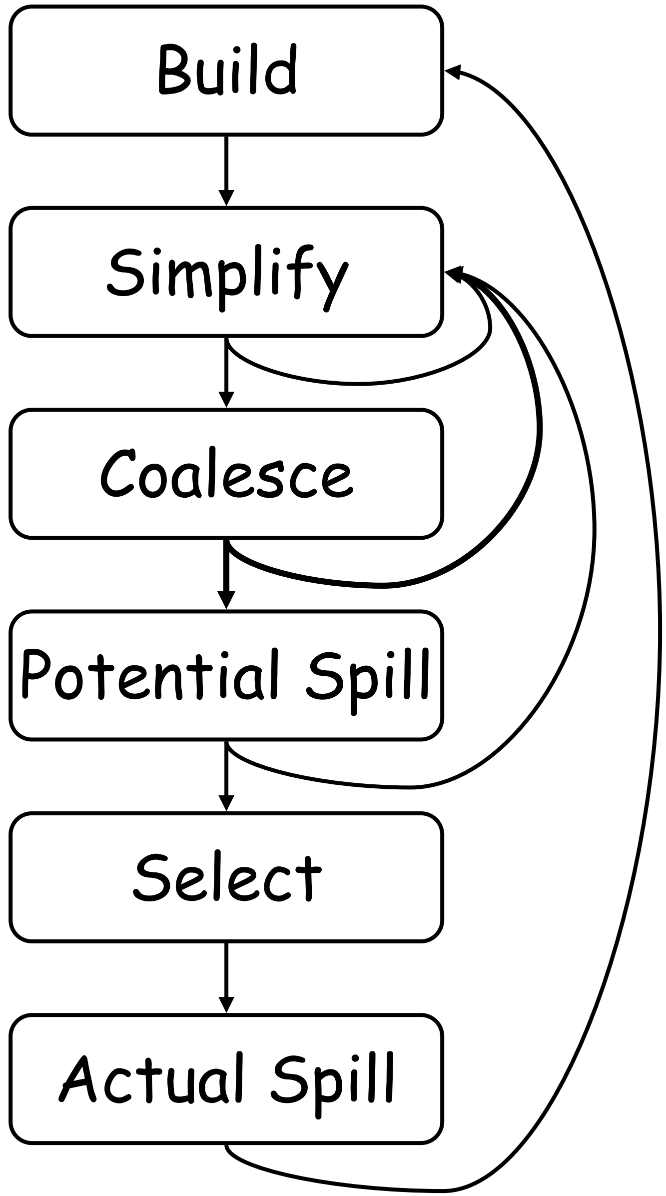
Coalescing

```
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?

Where We Are



Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill

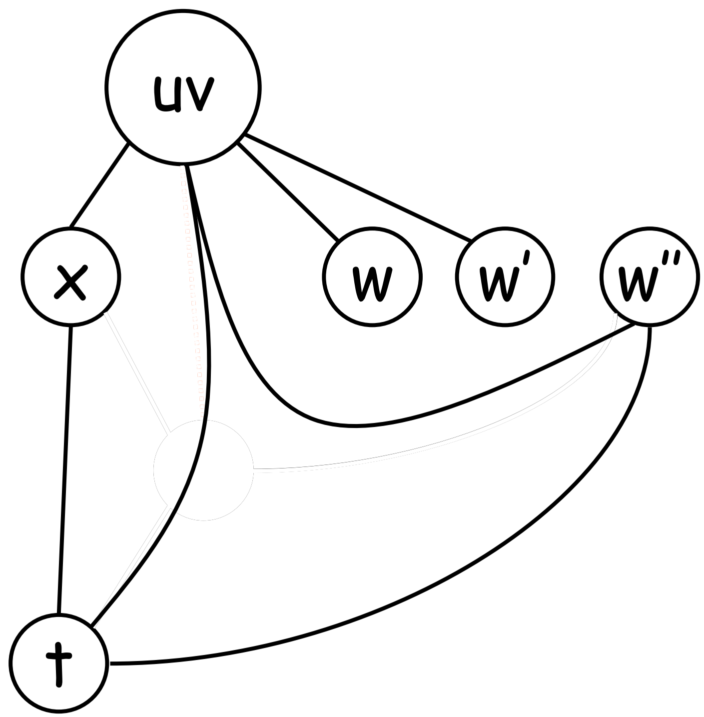
Coalescing

- 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

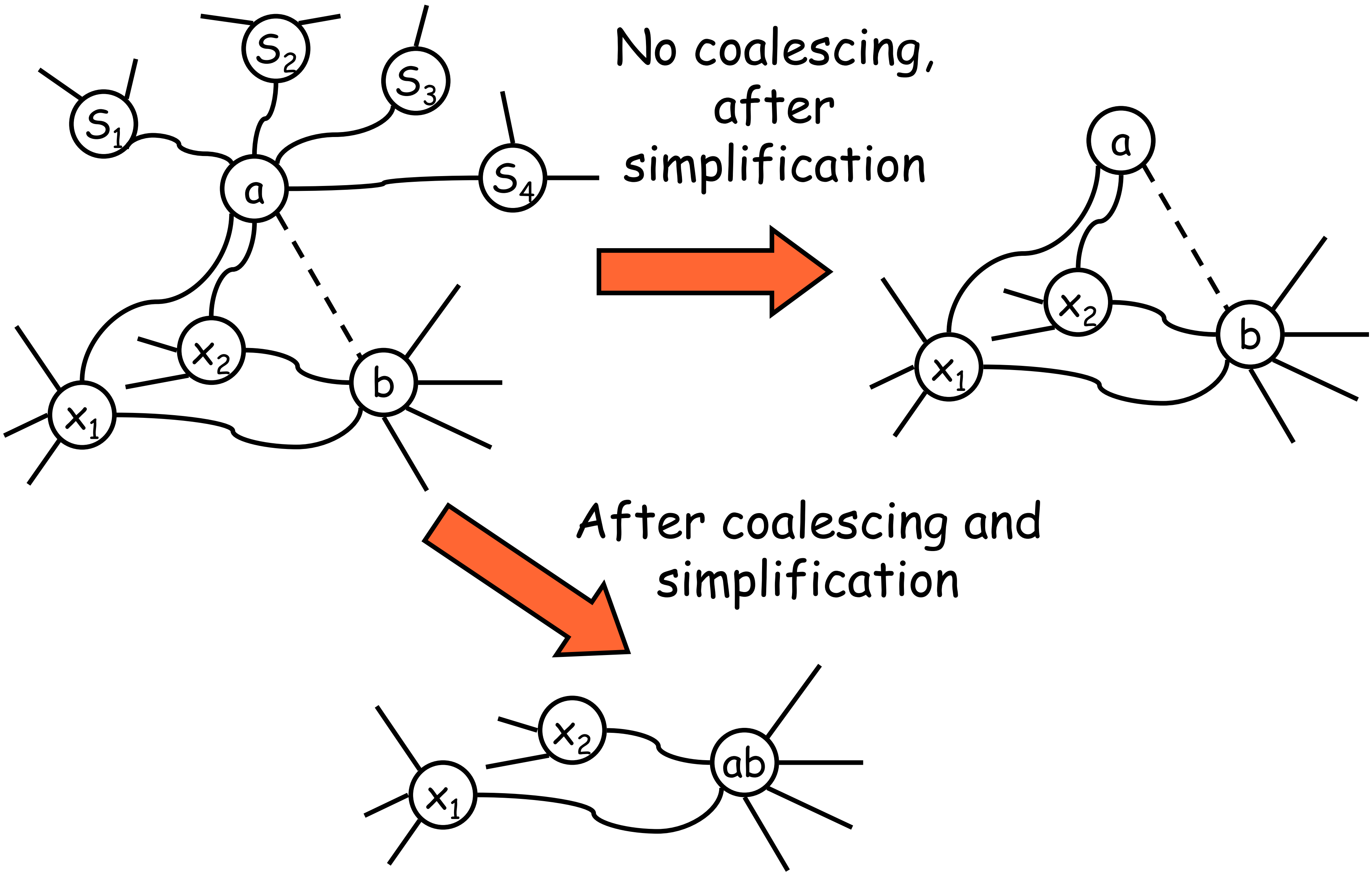
  (# of neighbors of ab with degree < k) < k
- Why?
  - Simplify removes all nodes with degree < k
  - # of remaining nodes < k
  - Thus, ab can be simplified



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



S1
S2
S3
S4
a
x2
x1
b
No coalescing, after simplification
a
x2
x1
b
After coalescing and simplification
x2
x1
ab

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

Spilling

- What should we spill?

Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Something that is NOT used in loops
  - Maybe something that is live across a lot of calls?

Setting Up For Better Spills

- We want temps not-live across procedures to be allocated to caller-save registers. Why?
- We want temps live across many procs to be in callee-save registers
- We prefer to use callee-save registers last.
- We want live ranges of precolored nodes to be short!

What about special registers?

- Instructions with register requirements

```
d ← a * b

ret x
```

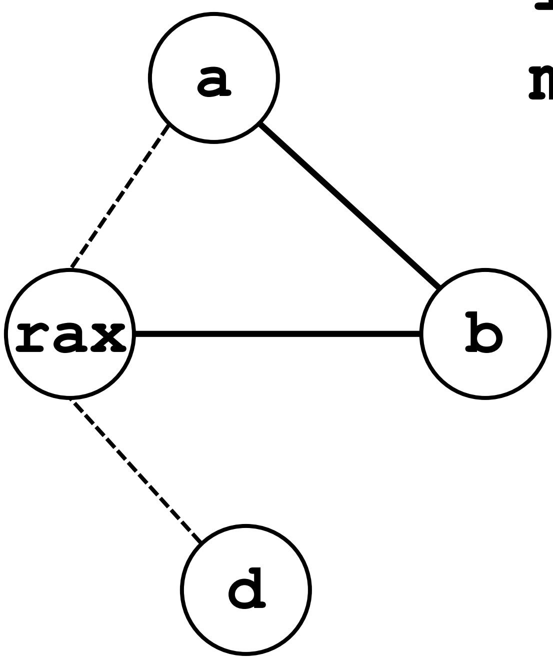
- Callee-save registers
  – x86-64: **RDI**, **RSI**, **RDX**, **RCX**, **R8**, **R9** must be saved by callee if callee wants to use them.

What about special registers?

- Instructions with register requirements

```
d ← a * b
```

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

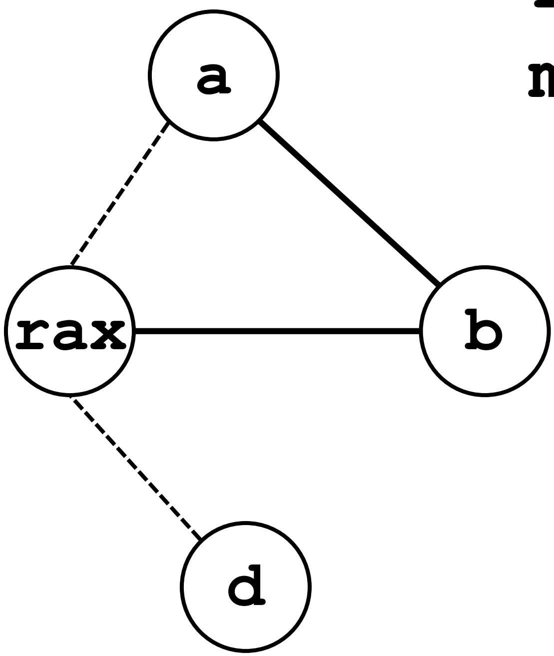


What about special registers?

- 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

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


ret x

        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

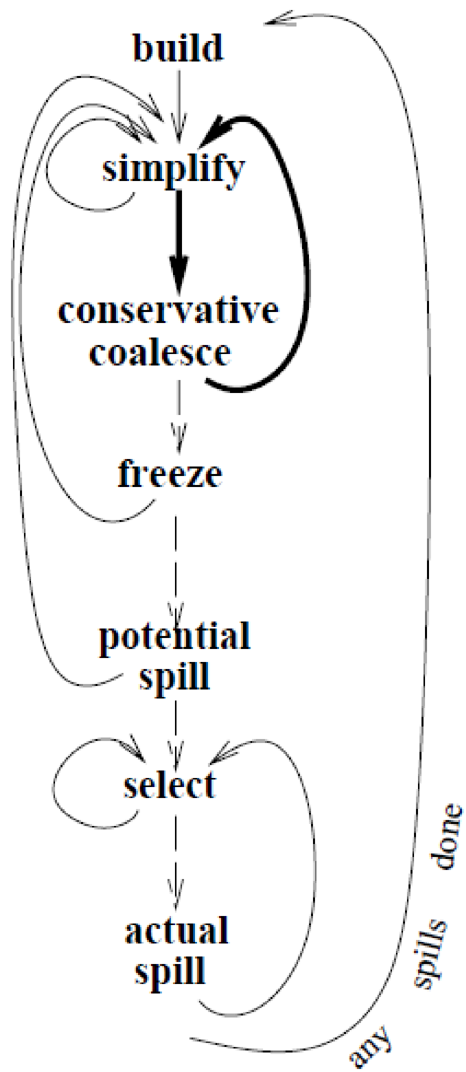
$t1 \leftarrow r$

…

epilogue: $r \leftarrow t1$

use r

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
  - Conservative Coalescing
  - Special registers
  - Implementation
- SSA-Based Register Allocation
  - SSA
  - ϕ-functions
  - 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 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

```
…
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;
   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;
   }
   …
```

In general,

N defs

M uses

$\Rightarrow$ 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;
   }
…
```

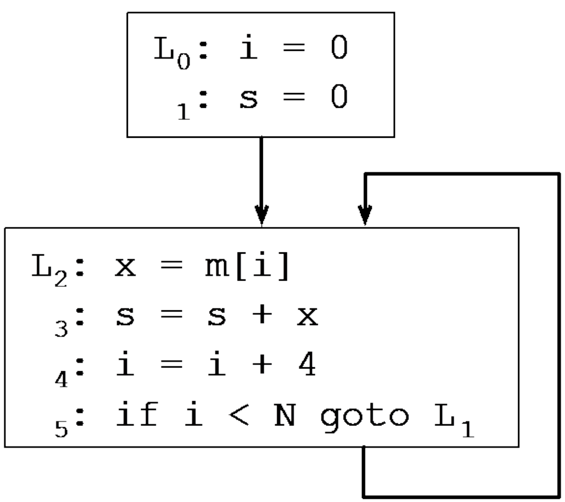
A possible solution is to limit each var to ONE def site

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.)



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 ← 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 ← 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 ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← 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

$a \leftarrow x + y$
$b \leftarrow a + x$
$a \leftarrow b + 2$
$c \leftarrow y + 1$
$a \leftarrow c + a$

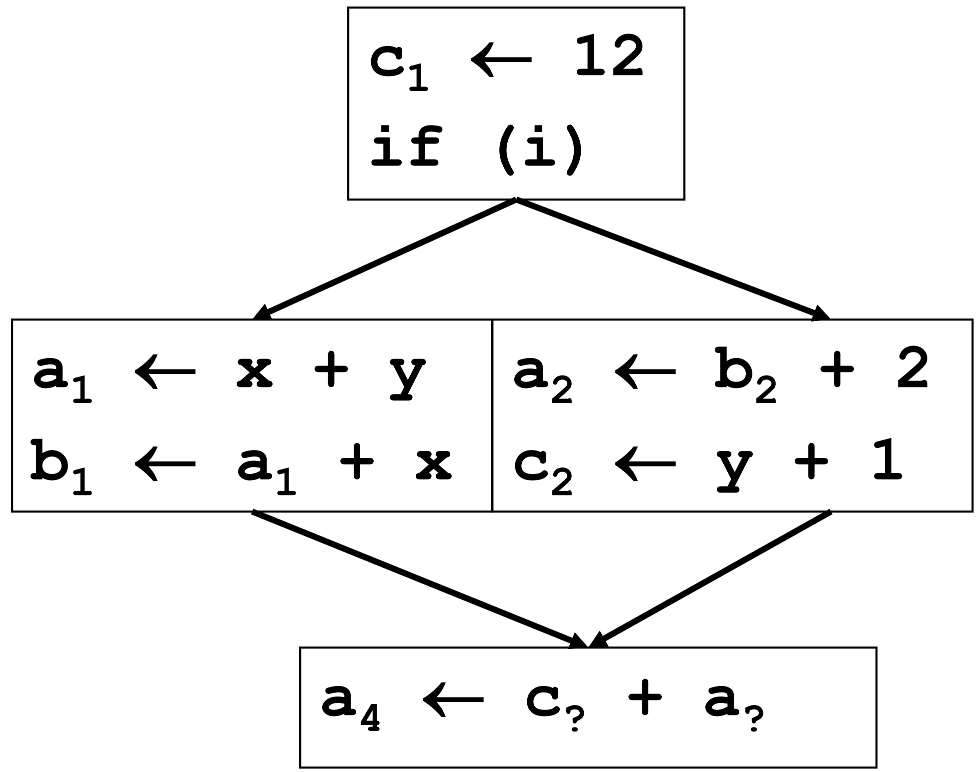
$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_2 \leftarrow b_1 + 2$
$c_1 \leftarrow y + 1$
$a_3 \leftarrow c_1 + a_2$

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: Φ-functions

Merging at Joins

$c_1 \leftarrow 12$

$\texttt{if (i)}$

$a_1 \leftarrow x + y$

$b_1 \leftarrow a_1 + x$

$a_2 \leftarrow b_2 + 2$

$c_2 \leftarrow y + 1$

$a_3 \leftarrow \Phi(a_1, a_2)$

$c_3 \leftarrow \Phi(c_1, c_2)$

$b_2 \leftarrow \Phi(b_1, ?)$

$a_4 \leftarrow c_3 + a_3$

The Φ function

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

  $X_{new} \leftarrow \Phi(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_p)$

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

“Implementing” Φ*

$c_1 \leftarrow 12$
if (i)

$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_3 \leftarrow a_1$
$c_3 \leftarrow c_1$

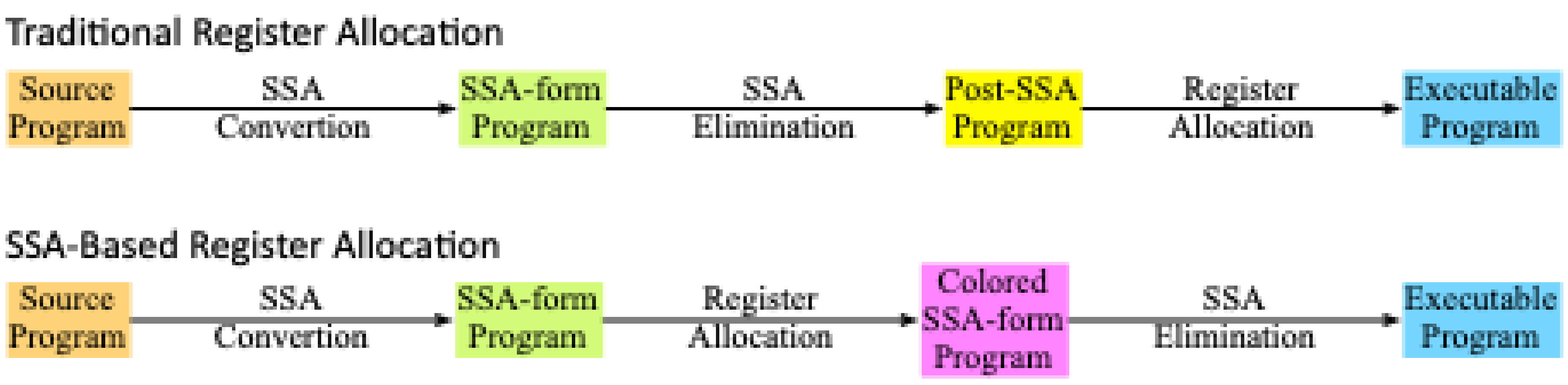
$a_2 \leftarrow b + 2$
$c_2 \leftarrow y + 1$
$a_3 \leftarrow a_2$
$c_3 \leftarrow c_2$

$\cancel{a_3 \leftarrow \Phi(a_1, a_2)}$
$\cancel{c_3 \leftarrow \Phi(c_1, c_2)}$
$a_4 \leftarrow c_3 + a_3$

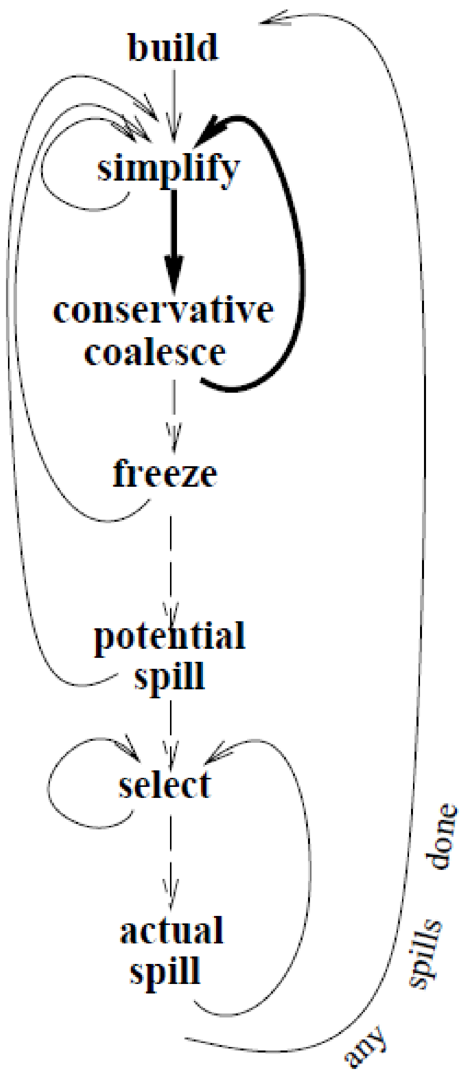
*Huge caveat here, discussed later. (e.g, lost-copy, swap-problem)

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



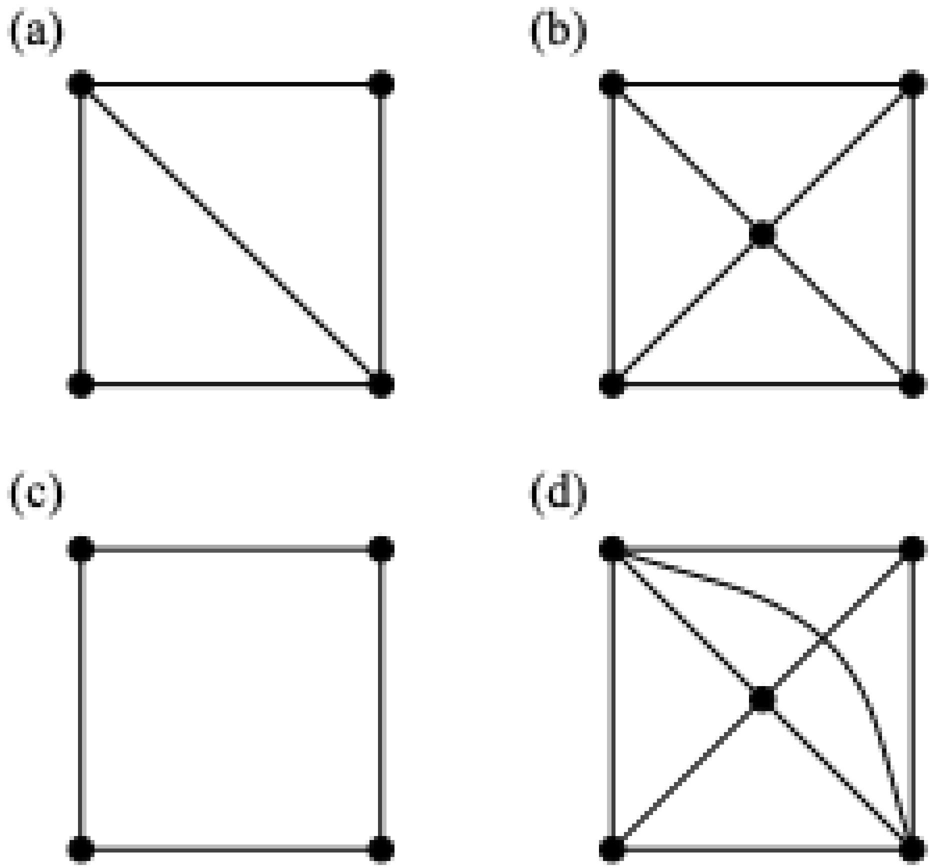
Simplify – creating order in which to color nodes

Select – Uses "simplify" order to color nodes

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

Chordal Graphs

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

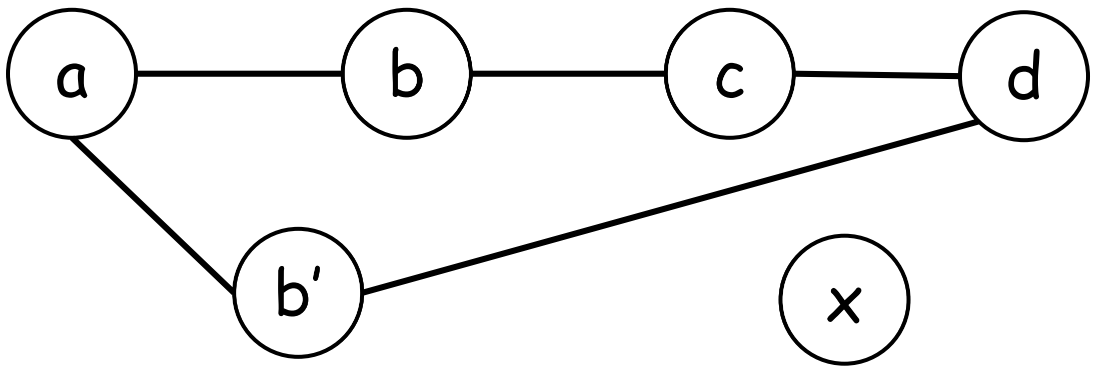


Graph Facts

- Clique: fully connected subgraph
- Chromatic number of graph G: minimal k such that G is k-colorable
- chromatic number of G $\geq$ 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!

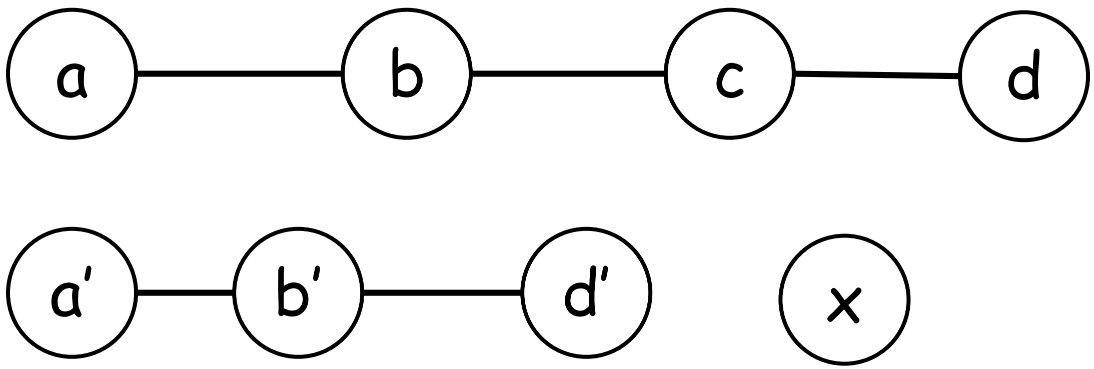
Non-chordal example

```
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  ← 0
b  ← 1
c  ← a + b
d  ← b + c
a′ ← c + d
b′ ← 7
d′ ← a′ + b′
x  ← b′ + d′
     ret x
```

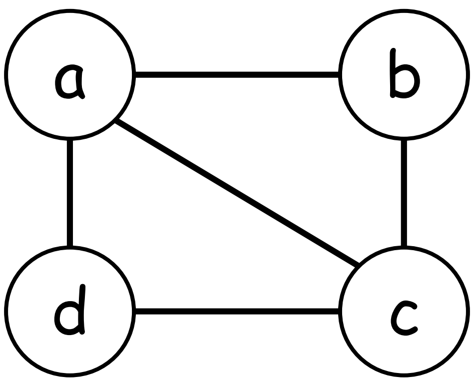


Adding more temps → fewer registers!

BTW: now in SSA-form!

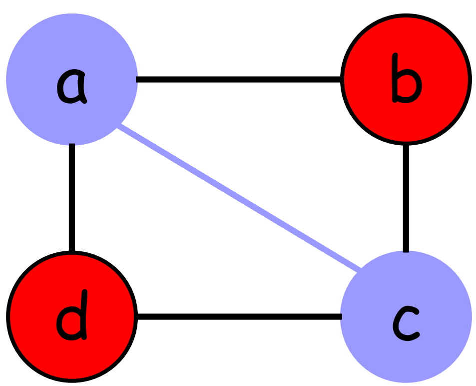
Simplical Elimination Ordering

- 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



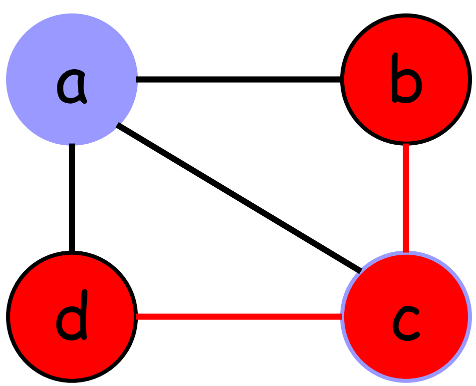
Simplical Elimination Ordering

- 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



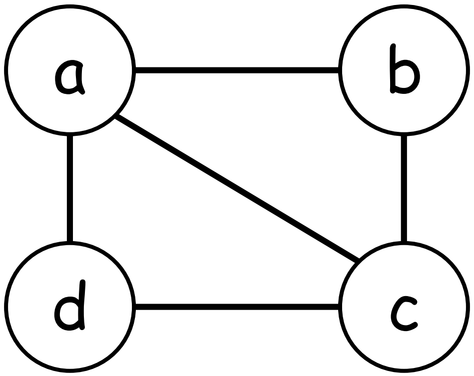
Simplical Elimination Ordering

- 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

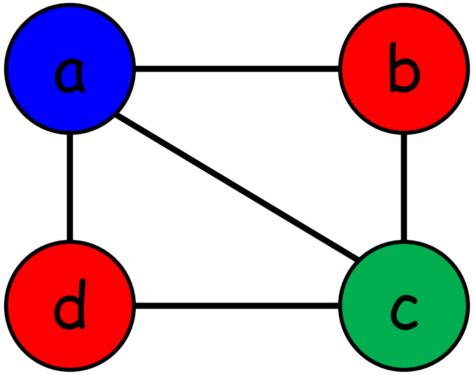
- If $G = (V, E)$ is a graph, then a vertex $v \in V$ is called *simplicial* if, and only if, its neighborhood in G is a clique.
- A *Simplicial Elimination Ordering* of G is a bijection $\sigma: V(G) \rightarrow \{1, \ldots, |V|\}$, such that every vertex v_i is a simplicial vertex in the subgraph induced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



b, a, c, d

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

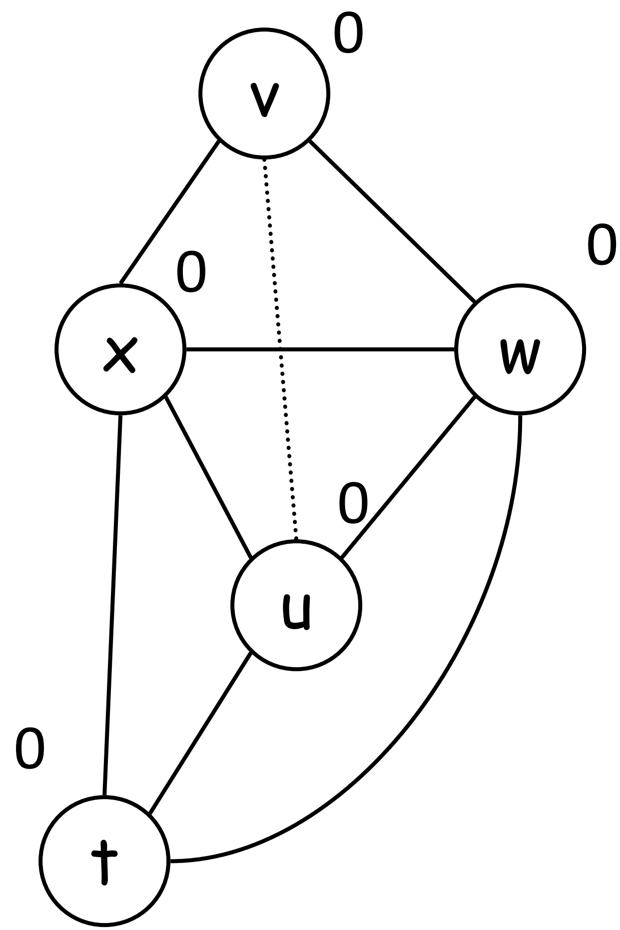
Use Maximum Cardinality Search to generate SEO

Maximum Cardinality Search

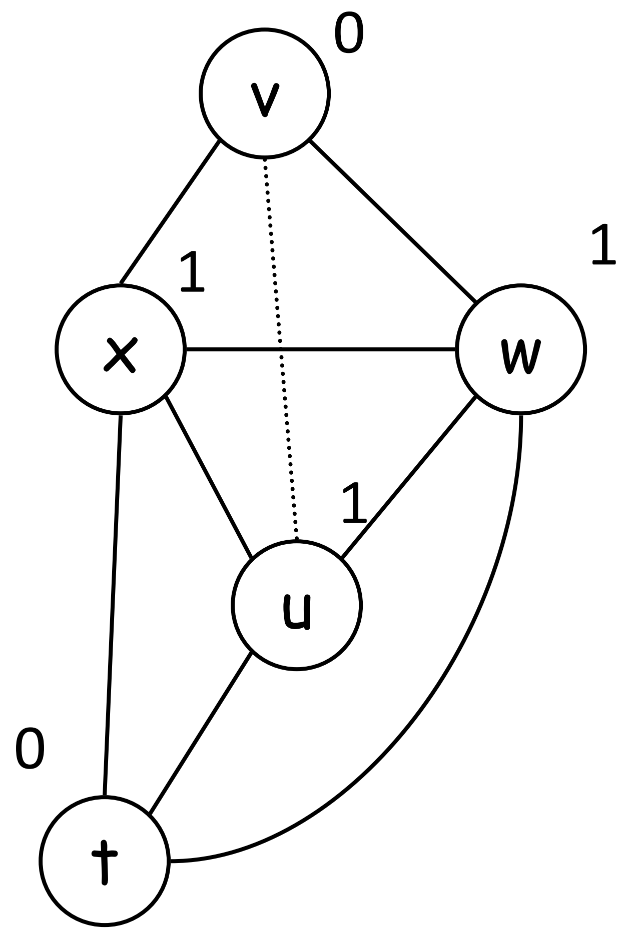
input: $G = (V, E)$ with $|V| = n$
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ **do** $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  let $v \in V$ be a node such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    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 ←  1
w ←  v + 3
x ←  w + v
u ←  v
t ←  u + x
  ←  w
  ←  t
  ←  u
```

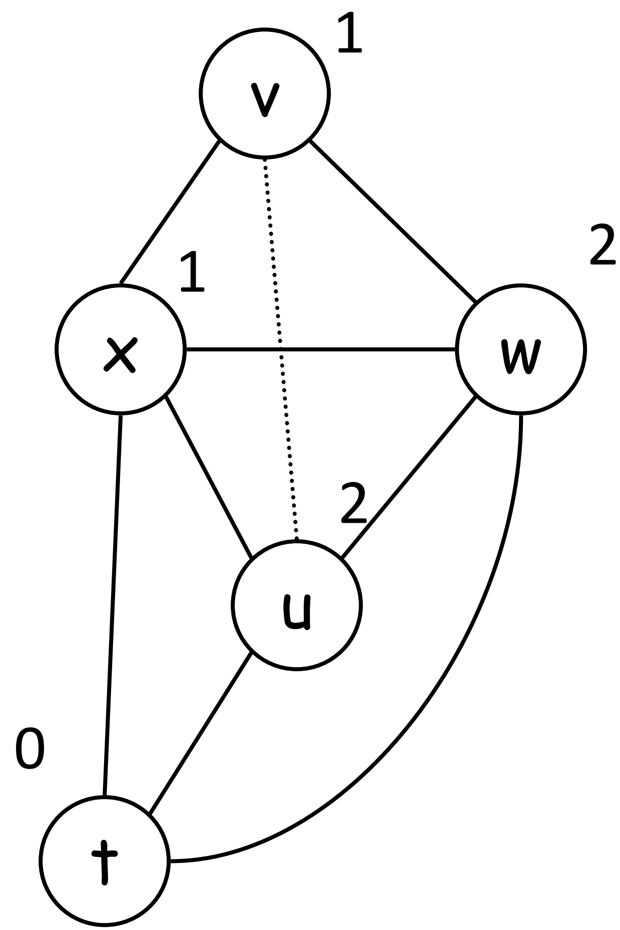


```
v ←  1
w ←  v + 3
x ←  w + v
u ←  v
t ←  u + x
  ←  w
  ←  t
  ←  u
```



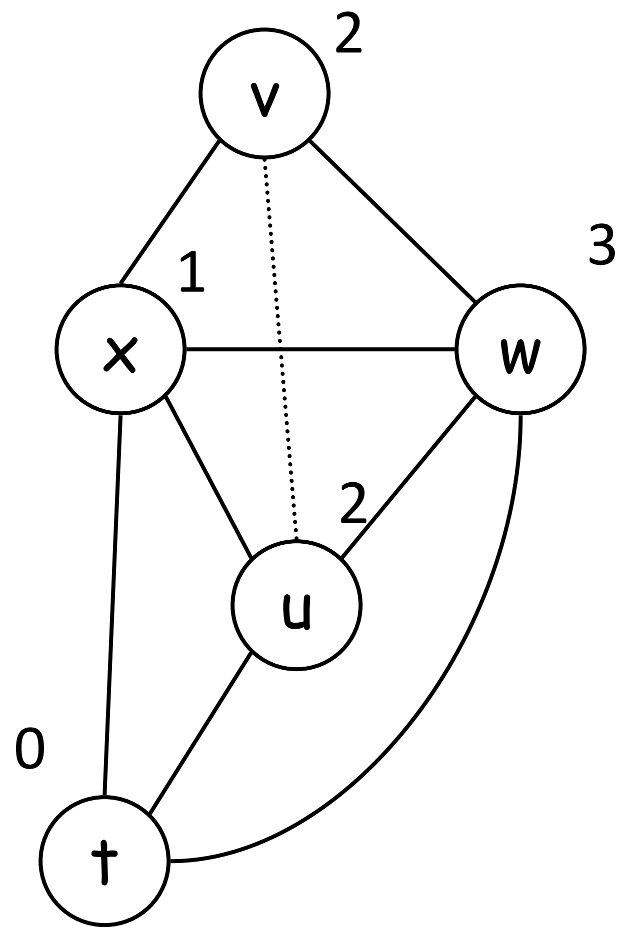
SEO: t

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



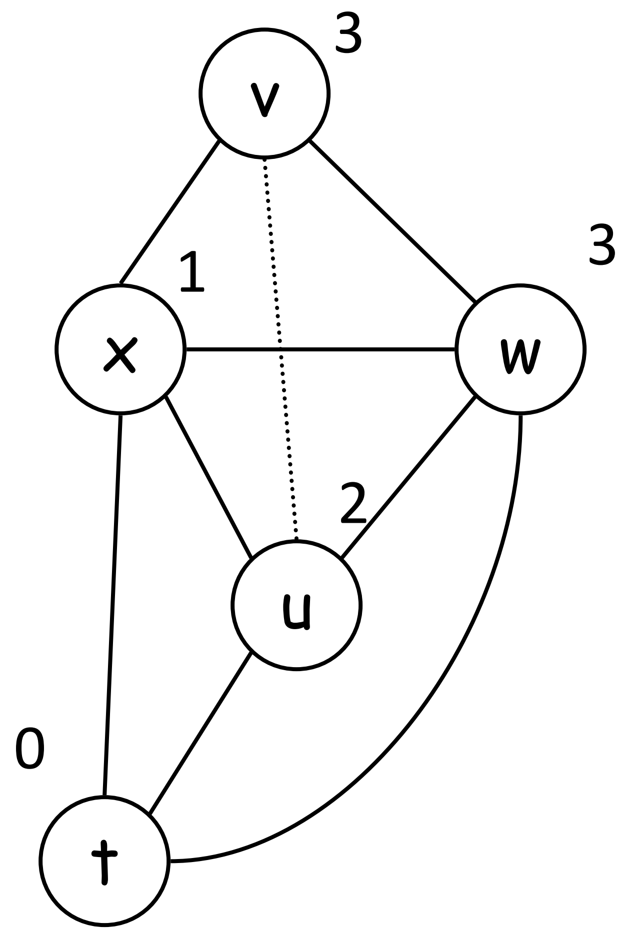
SEO: t, x

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



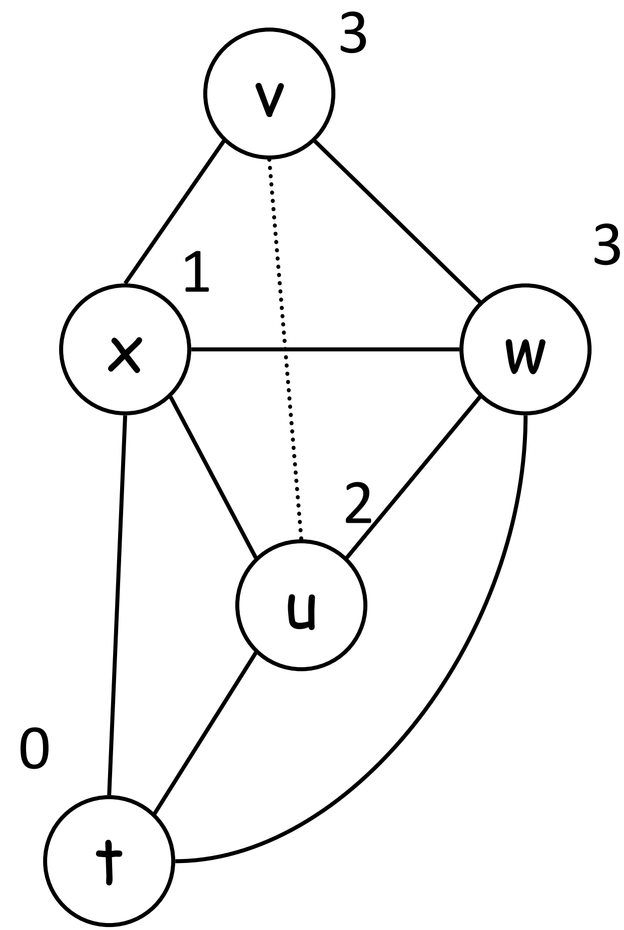
SEO: t, x, u

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



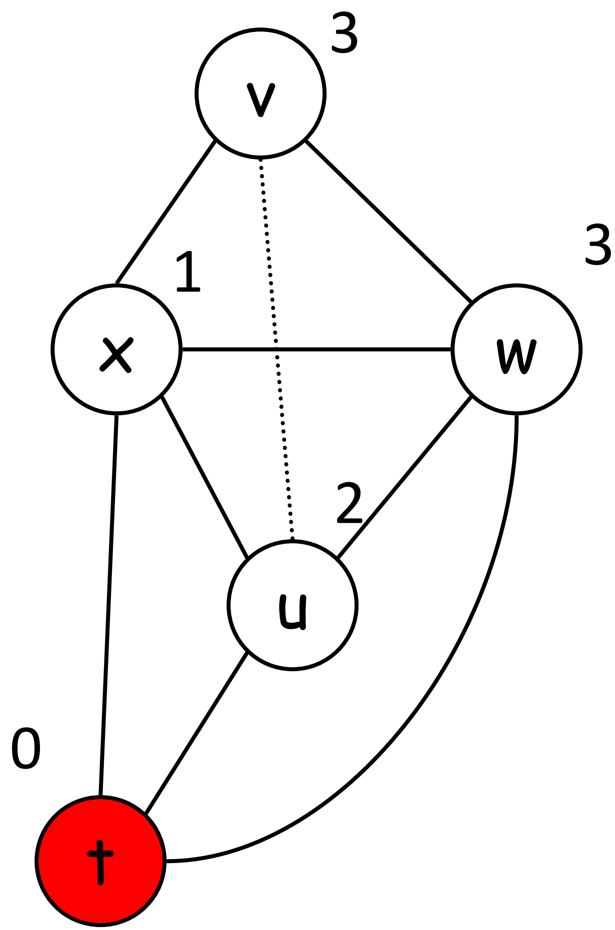
SEO: t, x, u, w

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



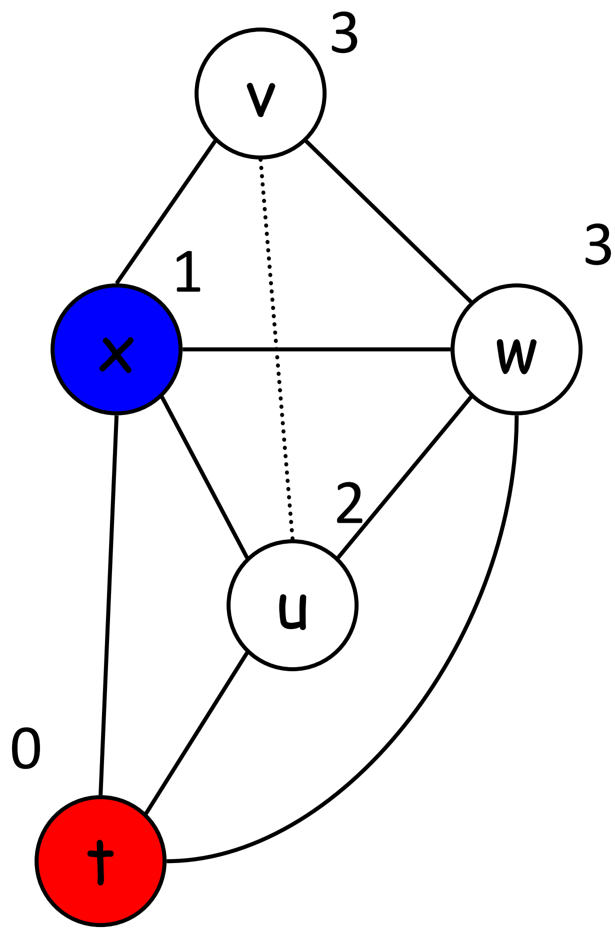
SEO: t, x, u, w, v

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



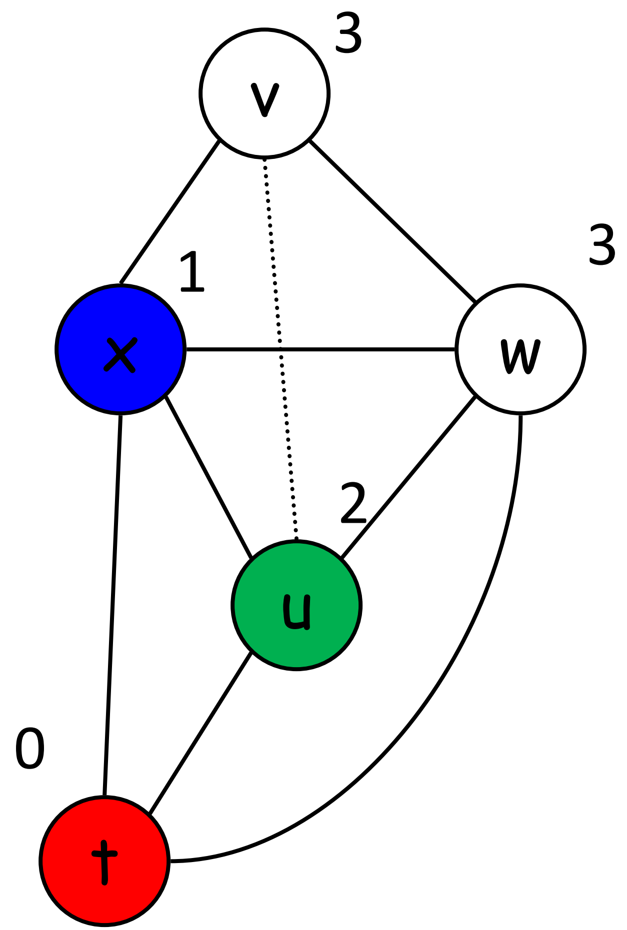
SEO: t, x, u, w, v

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



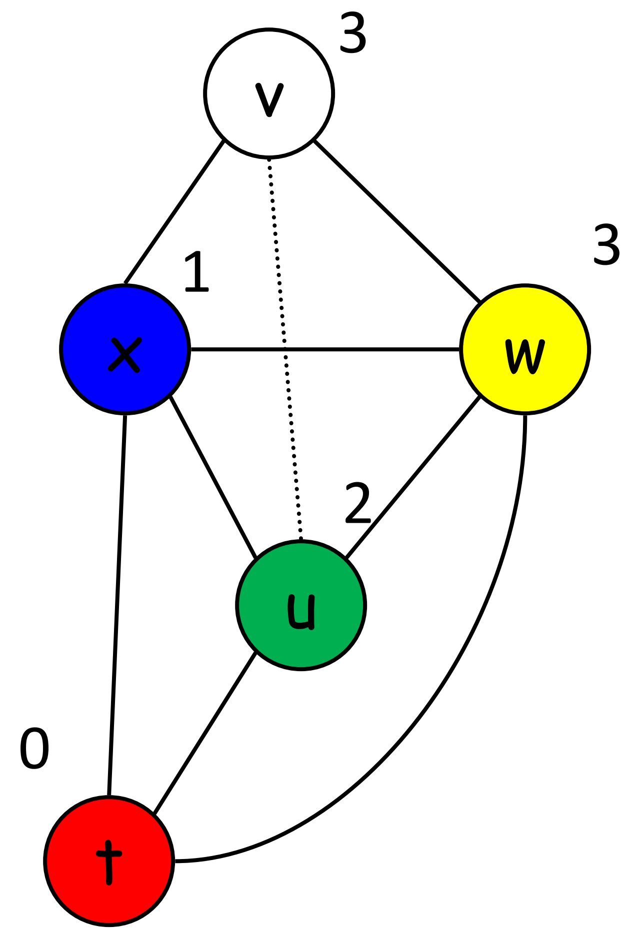
SEO: t, x, u, w, v

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



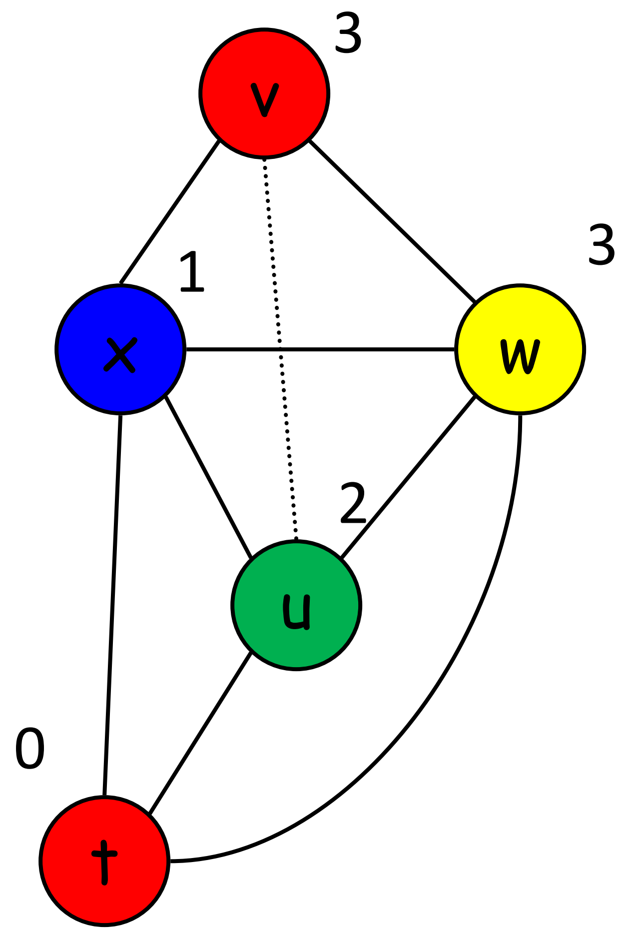
SEO: t, x, u, w, v

```
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

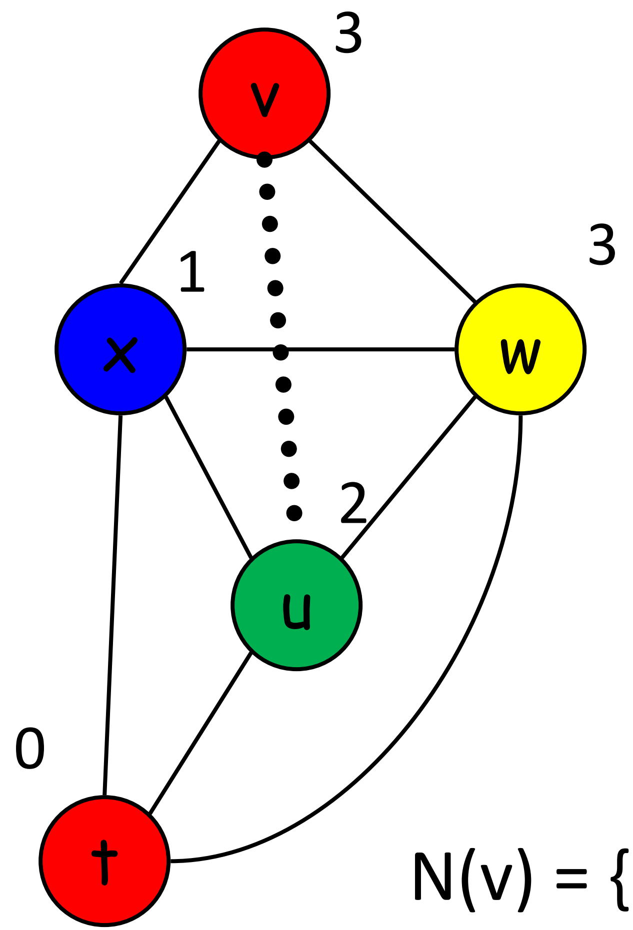
Best Effort Coalescing

input: list L of copy instructions, $G = (V, E)$, K
output: G', the coalesced graph G

$G' = G$
for all $x = y \in L$ **do**
  let S_x be the set of colors in $N(x)$
  let S_y be the set of colors in $N(y)$
  if $\exists c, c < K, c \notin S_x \cup S_y$ **then**
    let xy, $xy \notin V$ be a new node **in**
      add xy to G' with color c
      make xy adjacent to every v, $v \in N(x) \cup N(y)$
      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
```

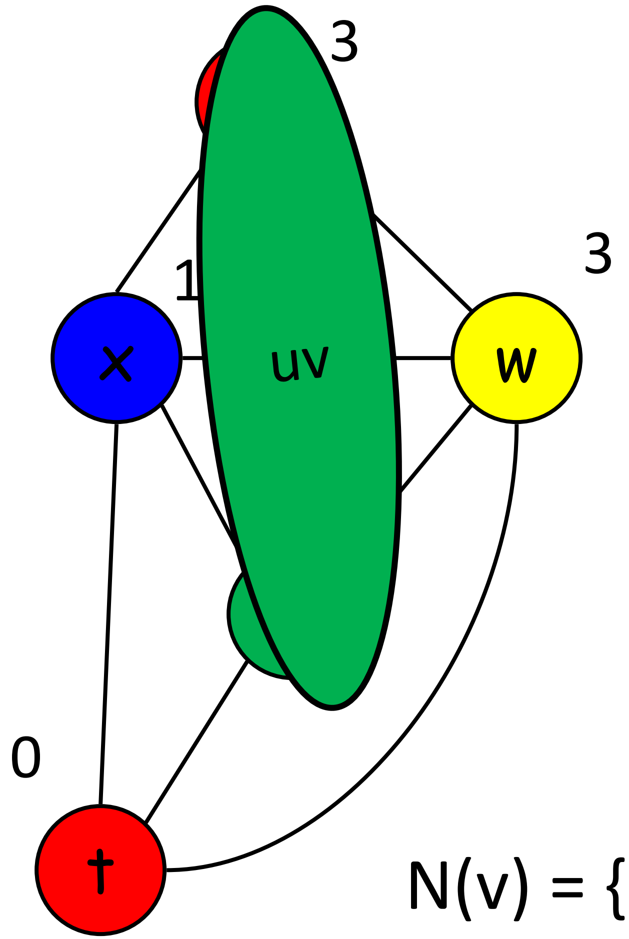


N(v) = { x, w }

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 }

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

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