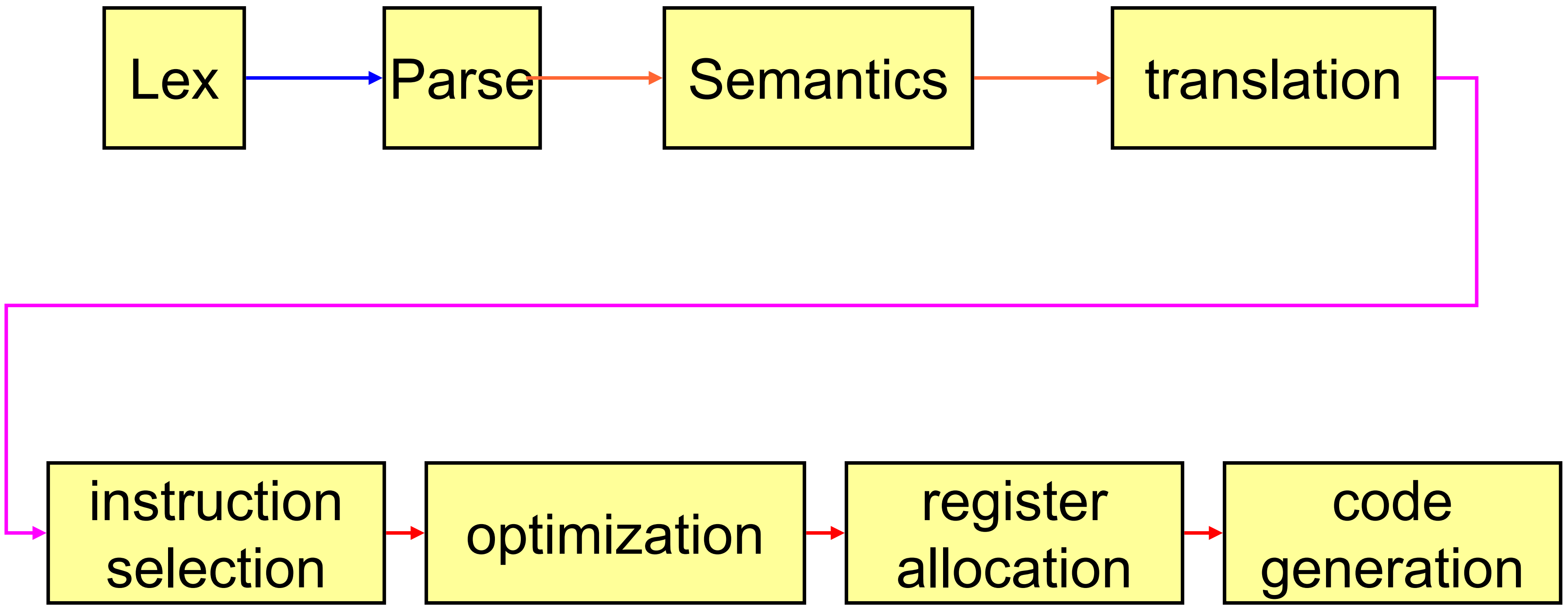
Register Allocation

15-411/15-611 Compiler Design

Seth Copen Goldstein

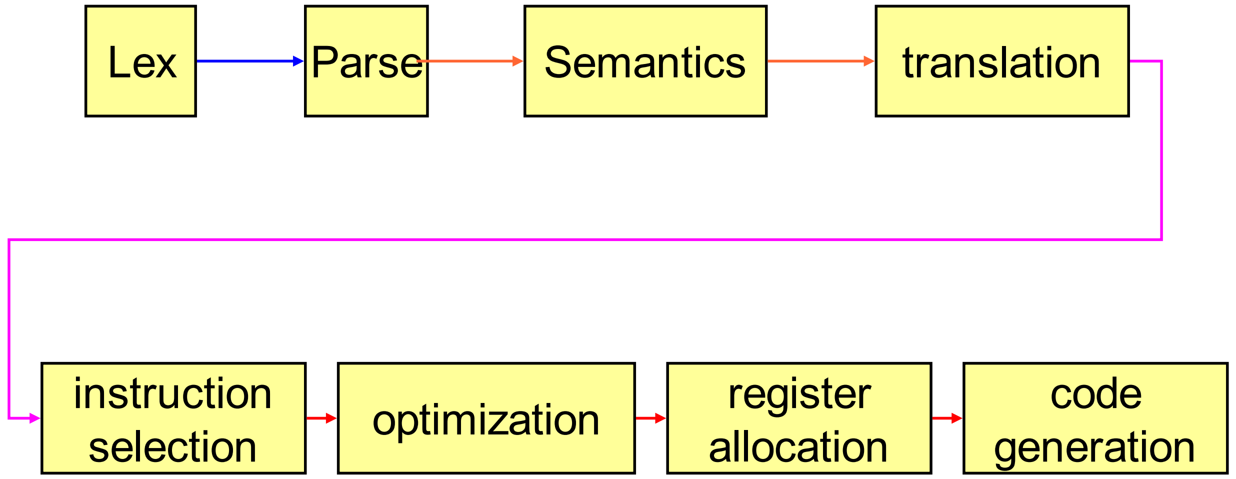
September 3, 2019

Cartoon Compiler



Unusual Order

- Standard is to start at the start and proceed down the passes: lexing, parsing, …
- We start with Register Allocation, then do Instruction Selection!



Today

- Intro to language of L1
- briefly: AST, Abstract assembly, Temps
- Register Allocation Overview
- Interference Graph
- Iterated Register Allocation
  - Simplify/Select
  - Coalescing
  - Spilling
- Special Registers
- Start Chordal Graphs, SSA-coloring

Simple Source Language

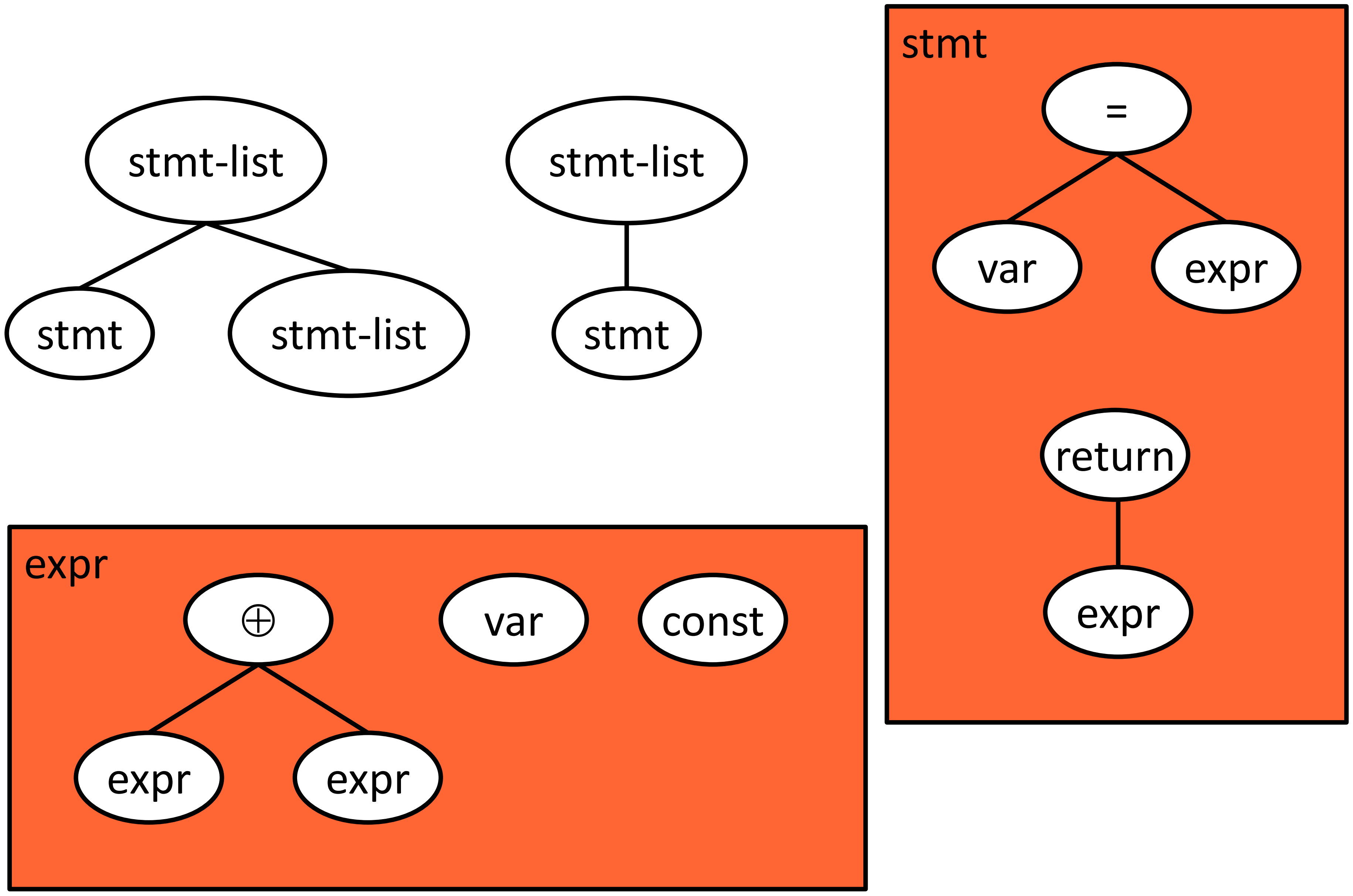
- A language of assignments, expressions, and a return statement.
- Straight-line code
- Basically lab1 subset of C0

Simple Source Language

program	:=	s_1 **;** s_2 **;** … s_n **;**	sequence of statements
s	:=	v **=** e	assignment
	|	**return** e	return
e	:=	c	constant
	|	v	variable
	|	$e_1 \oplus e_2$	binary operation
$\oplus$	:=	**+ | - | * | / | %**	

Ambiguity?
Semantics?

Abstract Syntax Tree

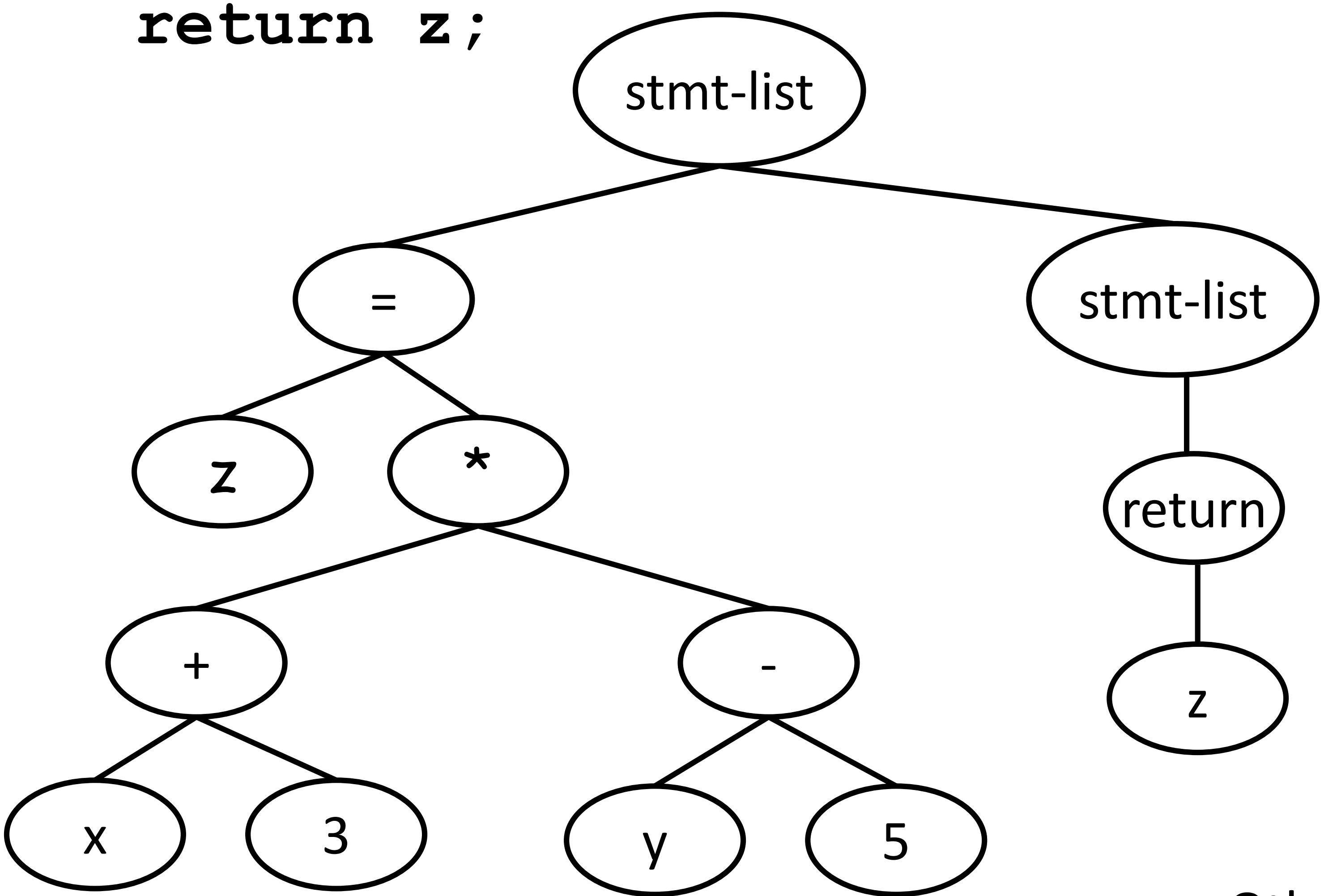


Example

```
z = x + 3 * y – 5;
return z;
```

Possible parse tree

```
z = x + 3 * y - 5;
return z;
```



Other possibilites?

Abstract Assembly as IR

- Lowering of AST
- Facilitate
  - Analysis & optimizations
  - Translation to actual assembly
- Features:
  - Unlimited number of “temporaries”
  - May not restrict how memory is used
  - Simple operations
  - May not restrict how constants are used
  - May specify certain “special registers”

In today’s world aka registers

Abstract Assembly as IR

- Features:
  - Unlimited number of "registers" (aka "temps")
  - May (or may not) restrict how memory is used
  - Simple operations
  - May not restrict how constants are used
  - May specify certain "special registers"
- Form:

  dest $\leftarrow$ src_1 operator src_2

  dest $\leftarrow$ operator src_1

  operator

src can be:

- constant
- temp
- special register
- memory

Abstract Assembly Language

program	:=	$i_1\ i_2\ \ldots\ i_n$	seq of instructions	
i	:=	$d \leftarrow s$	move	
	|	$d \leftarrow s_1 \oplus s_2$	binop	
	|	**return** s_1	return	
s	:=	c	intermediate	
	|	t	temporary	values
	|	r	register	
d	:=	t		locations
	|	r		
$\oplus$	:=	**+ | - | * | / | %**		

What is right "level"?

Closer to the machine

program	:=	$i_1\ i_2\ \ldots\ i_n$	seq of instructions
i	:=	$d \leftarrow s$	move
	|	$d \leftarrow s_1 \oplus s_2$	binop
	|	**return**	return what is in **`rax`**
s	:=	c	intermediate
	|	t	temporary
	|	r	register
d	:=	t	
	|	r	
$\oplus$	:=	**+** | **-** | ***** | **/** | **%**	

Register Allocation

- Until register allocation we assume an infinite set of registers (aka “temps” or “pseudo-registers”).
- But real machines have a fixed set of registers.
- The register allocator must assign each temp to a machine register.

Register Allocation

- Map the variables & temps in the abstract assembly to actual locations in the machine
- The locations are either
  - physical registers
  - slots in the activation frame
- Essential for modern architectures
  - registers are much faster, consume less power, etc.
  - Some operations require registers
  - Goal: Try and allocate as many of the important variables/temps to registers.
- However, there are only a few registers

Sub-tasks of Register Allocation

- **Assignment:** map temps to particular registers
- **Spilling:** If we can't assign to a register, assign to a slot in the stack frame and add code to save and restore temp.
- **Coalescing:** If possible eliminate moves, **`a←b`**, and map both a & b to the same location.
- Ensure special cases are handled properly.
  - instructions, e.g., **`imul`**, **`ret`**, …
  - ABI, e.g., callee/caller save registers, function arguments.

Interference

- Consider two temps, t0 and t1.
- If the live ranges for t0 and t1 overlap, we say that they *interfere.*

- *First rule of register allocation*:
  - Temps with interfering live ranges may not be assigned to the same machine register.

Running Example

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

- Two variables, e.g., x & v, need to be in different registers if at some point in the program they hold different values.

Running Example

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

- Two variables, e.g., x & v, need to be in different registers if at some point in the program they hold different values.

Running Example

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

- Two variables, e.g., x & v, need to be in different registers if at some point in the program they hold different values.
- Use **liveness** information
- A variable is live at a given point in the program if it can be used at some later point in the program.

Liveness in straight line code

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

- Work backwards and at each instruction:
- If variable is used on right hand side, it is live-in
- if variable was live before it is still live-in (unless defined on left-hand side)

Liveness in straight line code

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

- Work backwards and at each instruction:
- If variable is used on right hand side, it is live-in
- if variable was live before it is still live-in (unless defined on left-hand side)

Liveness in straight line code

```
                    { }
v ←   1
                    { v }
w ←   v + 3
                    { w, v }
x ←   w + v
                    { w, x, v }
u ←   v
                    { w, u, x, v }
t ←   u + v
                    { w, t, u, x }
  ←   w + x
                    { u, t }
  ←   t
                    { u }
  ←   u
```

live-in sets

- Work backwards and at each instruction:
- If variable is used on right hand side, it is live-in
- if variable was live before it is still live-in (unless defined on left-hand side)

Live-out more useful

```
v ←   1
                  { v }
w ←   v + 3
                  { w, v }
x ←   w + v
                  { w, x, v }
u ←   v
                  { w, u, x, v }
t ←   u + v
                  { w, t, u, x }
  ←   w + x
                  { u, t }
  ←   t
                  { u }
  ←   u
                  { }
```

Interference and Liveness

```
v ←  1
                    { v }
w ←  v + 3
                    { w, v }
x ←  w + v
                    { w, x, v }
u ←  v
                    { w, u, x, v }
t ←  u + v
                    { w, t, u, x }
  ←  w + x
                    { u, t }
  ←  t
                    { u }
  ←  u
                    { }
```

- Two variables that are live at the same point in the program interfere with each other and need to be assigned to different registers.

General Plan

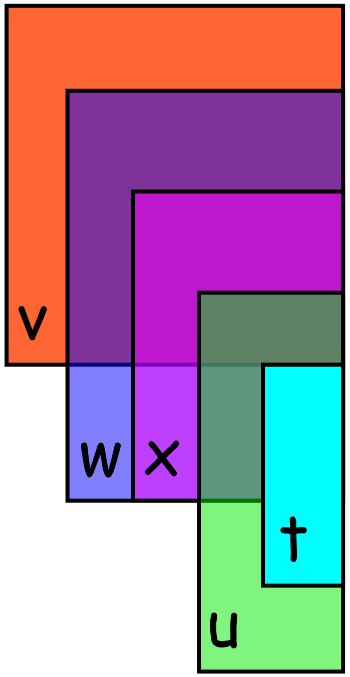
- Construct an interference graph
- Map temps to registers
- Deal with spills
- Generate code to save & restore
- Respect special registers
  - avoid reserved registers
  - Use registers properly
  - respect distinction between callee/caller save registers

Optimistic Graph Coloring

- Construct Interference Graph
  - Use liveness information
  - Each node in the interference graph is a temp
  - (u,v) $\in$ G iff u & v can't be in the same hard register, i.e., they interfere
- Color Graph
  - Assign to each node a color from a set of k colors, k = | register set |
- Spill
  - If can't color graph with k colors then spill some temps into memory. Regenerate asm code and start over.

An Example, k=4

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

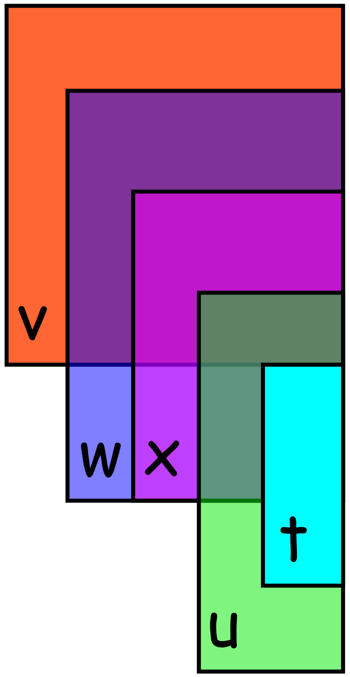


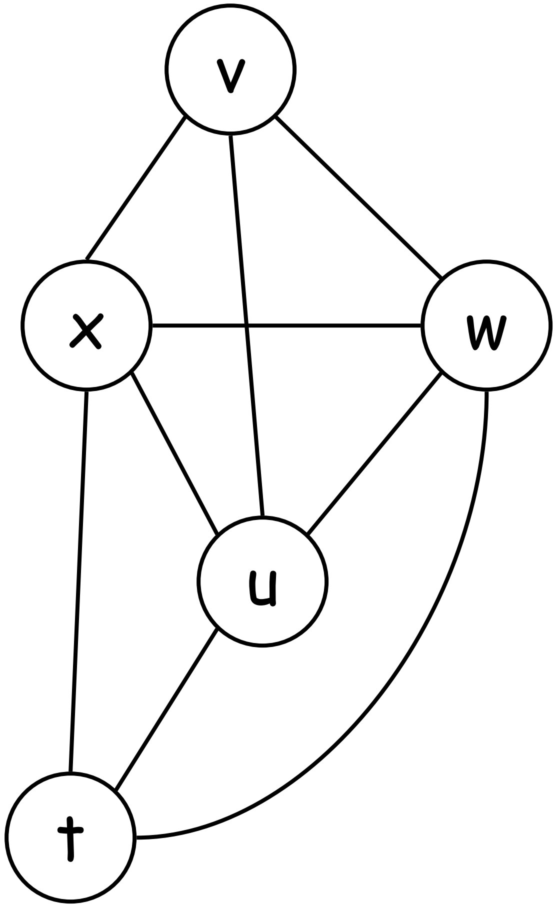
```
{ v }
{ w, v }
{ w, x, v }
{ w, u, x }
{ w, t, u }
{ u, t }
{ u }
{ }
```

Compute live ranges

An Example, k=4

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

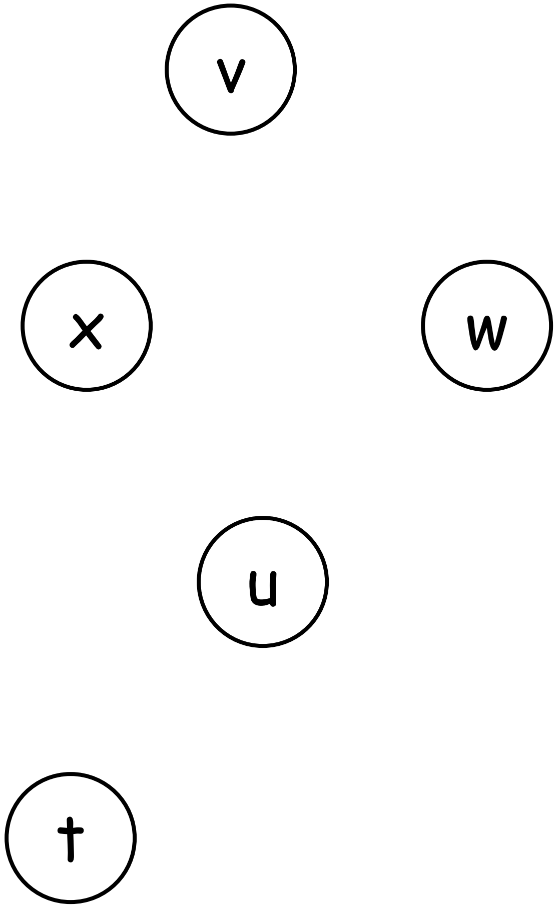




Construct the interference graph

In Practice

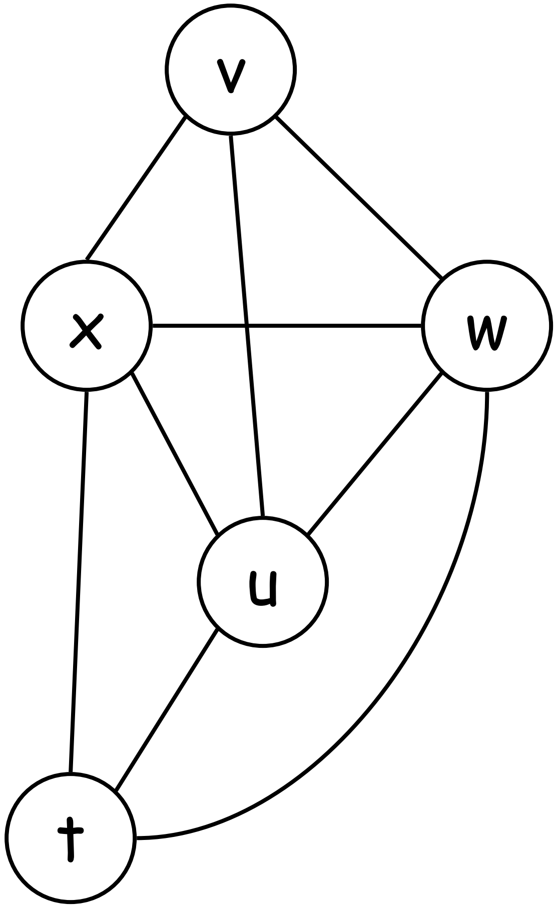
```
v ←  1          { v }
w ←  v + 3      { w, v }
x ←  w + v      { w, x, v }
u ←  v          { w, u, x }
t ←  u + v      { w, t, u }
  ←  w + x      { u, t }
  ←  t          { u }
  ←  u          { }
```



- At point of definition of t, add edges between t and all u ∈ live-out, t≠u

In Practice

```
v ←  1          { v }
w ←  v + 3      { w, v }
x ←  w + v      { w, x, v }
u ←  v          { w, u, x }
t ←  u + v      { w, t, u }
  ←  w + x      { u, t }
  ←  t          { u }
  ←  u          { }
```

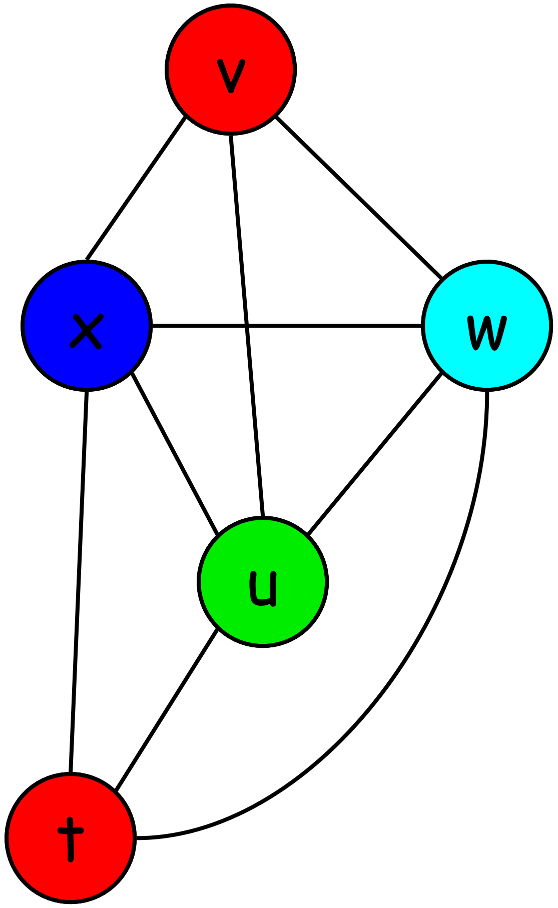


- At point of definition of t, add edges between t and all u ∈ live-out, t≠u

An Example, k=4

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

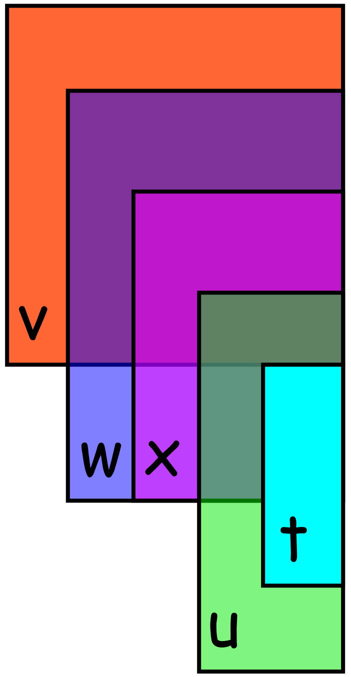
Voila, registers are assigned!

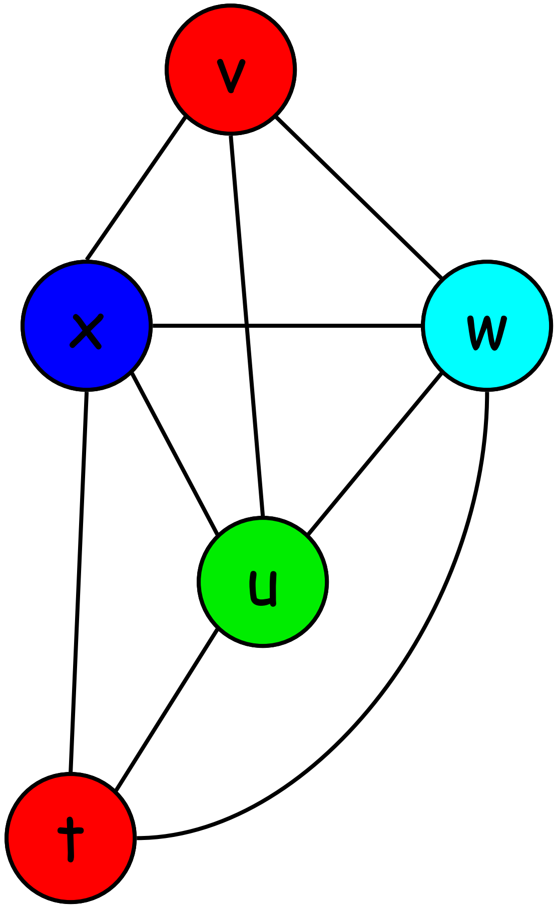


A greedy Coloring

An Example, k=4

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





u & v are special. They interfere, but only through a move!

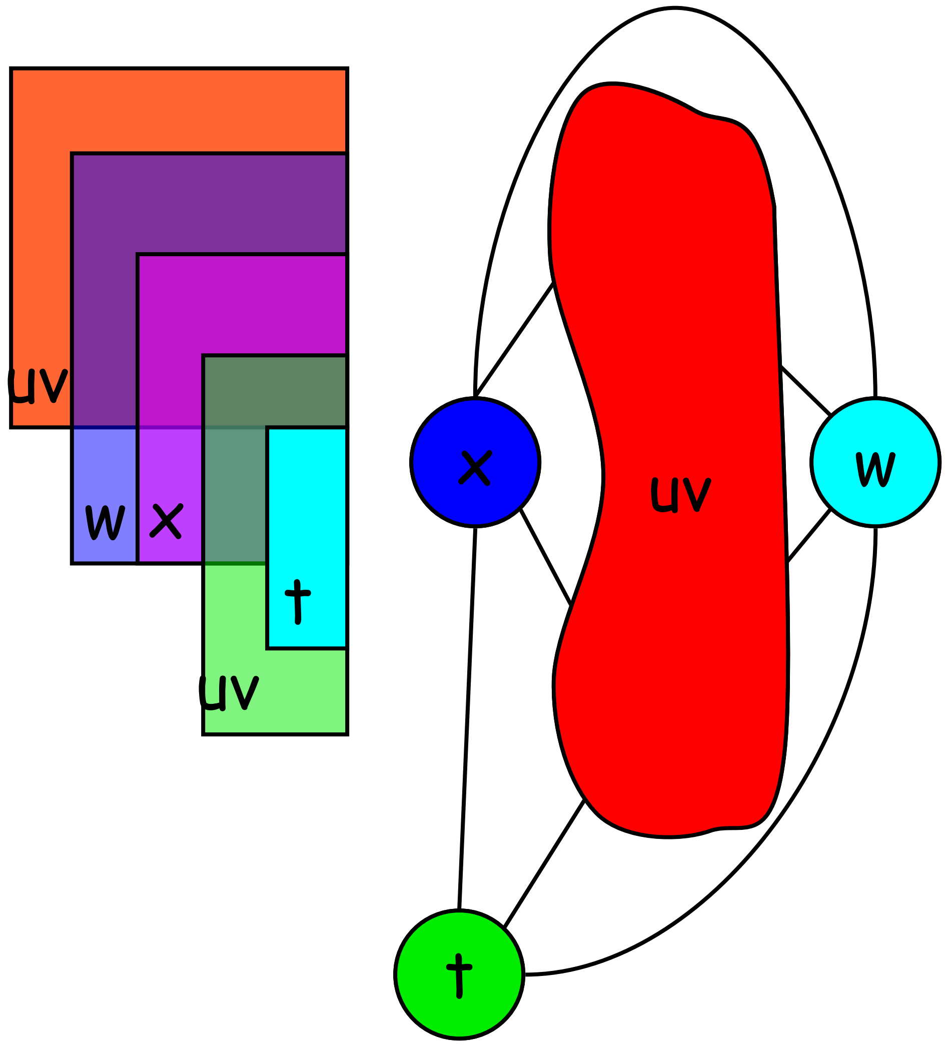
Interference and Coalescing

```
v ←   1
                    { v }
w ←   v + 3
                    { w, v }
x ←   w + v
                    { w, x, v }
u ←   v
                    { w, u, x, v }
t ←   u + v
                    { w, t, u, x }
  ←   w + x
                    { u, t }
  ←   t
                    { u }
  ←   u
                    { }
```

- We would like to eliminate the move **`u ← v`** by having `u` and `v` share a register (i.e, coalescing)

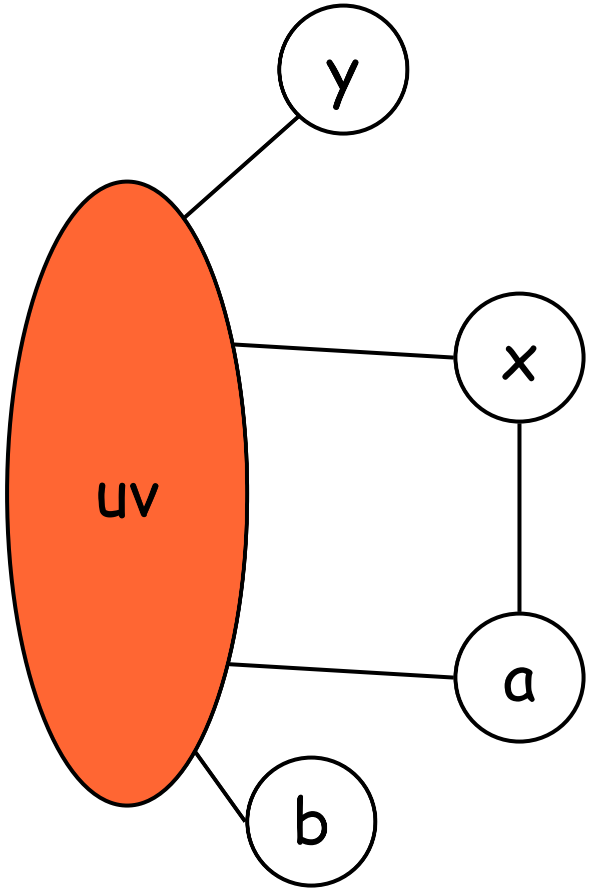
An Example, k=4

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



Rewrite the code to coalesce u & v

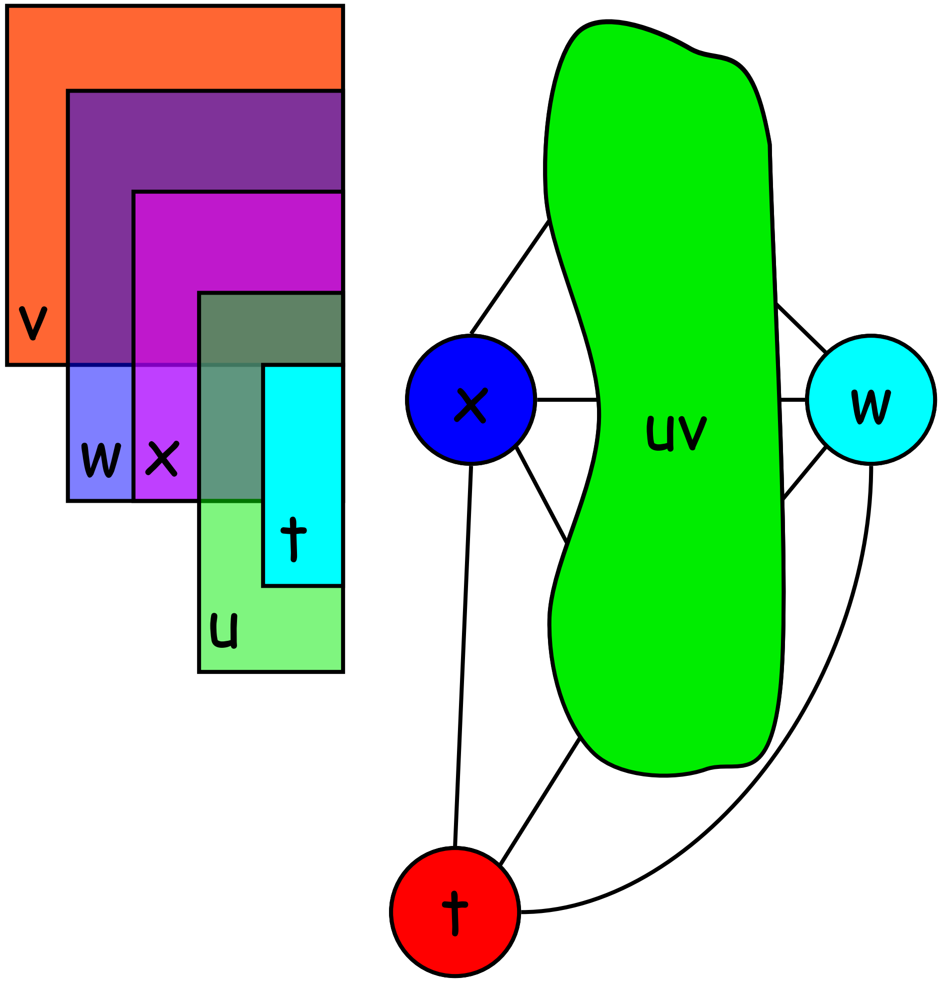
Is Coalescing always good?



Was 2-colorable,
now it needs 3 colors

So, we treat moves specially.

An Example, k=4



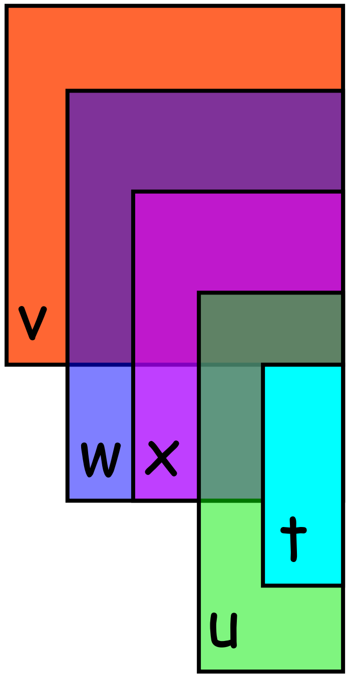
Interference from moves become "move edges."

An Example, k=3

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

An Example, k=3

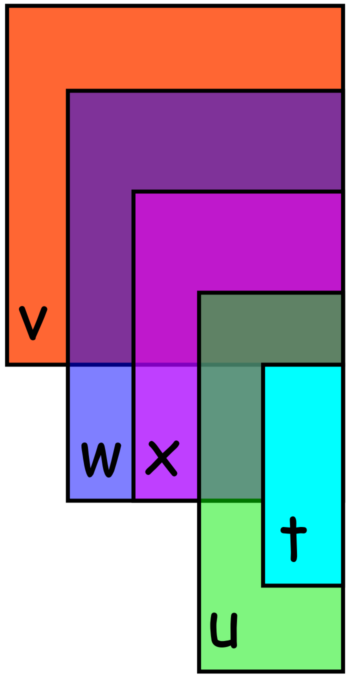
```
v ←  1
w ←  v + 3
x ←  w + v
u ←  v
t ←  u + v
  ←  w + x
  ←  t
  ←  u
```

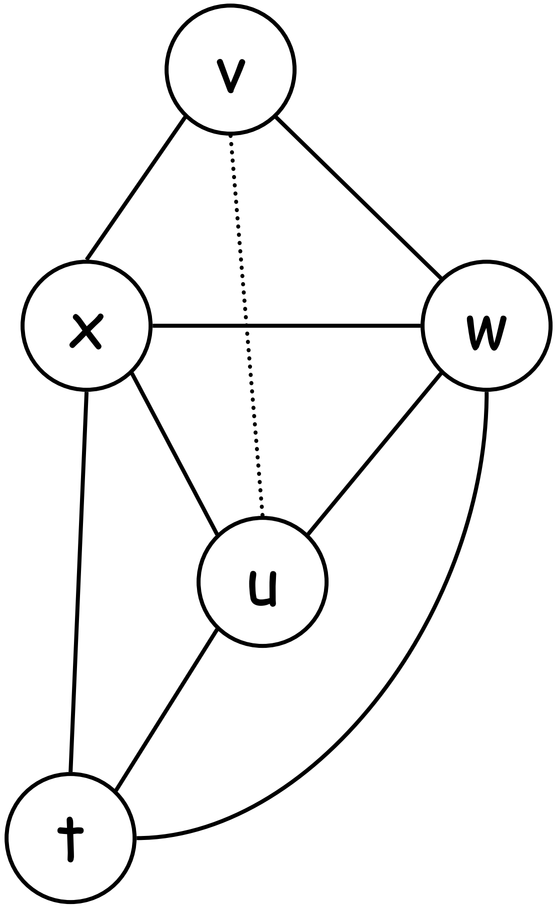


Compute live ranges

An Example, k=3

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

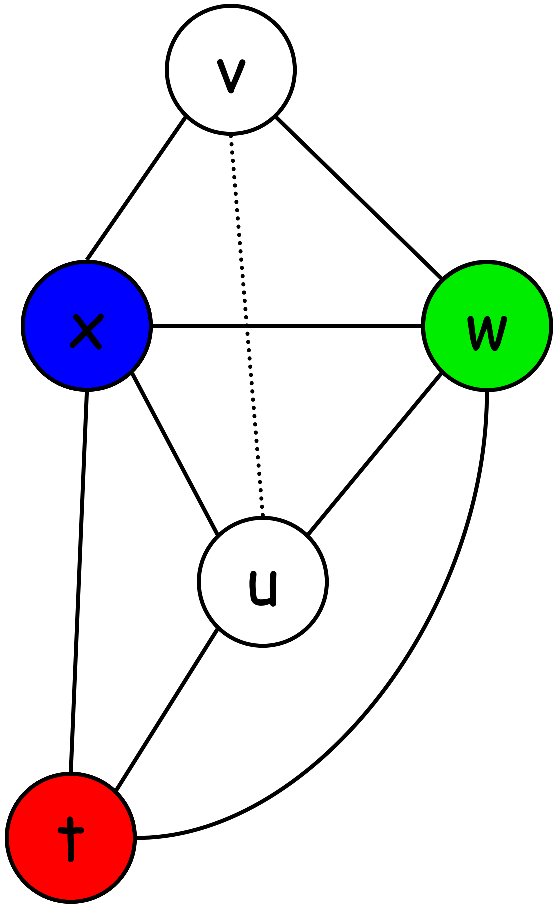




Construct the interference graph

An Example, k=3

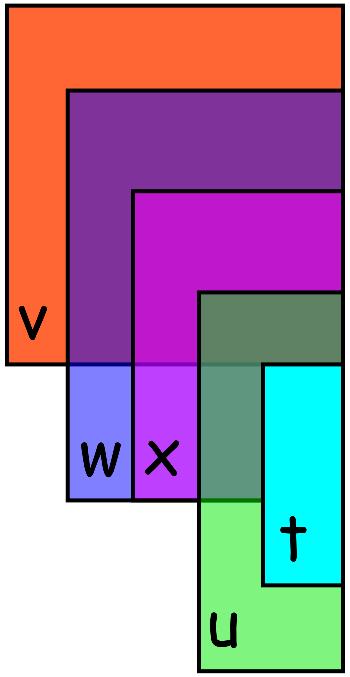
```
v ←  1
w ←  v + 3
x ←  w + v
u ←  v
t ←  u + v
  ←  w + x
  ←  t
  ←  u
```

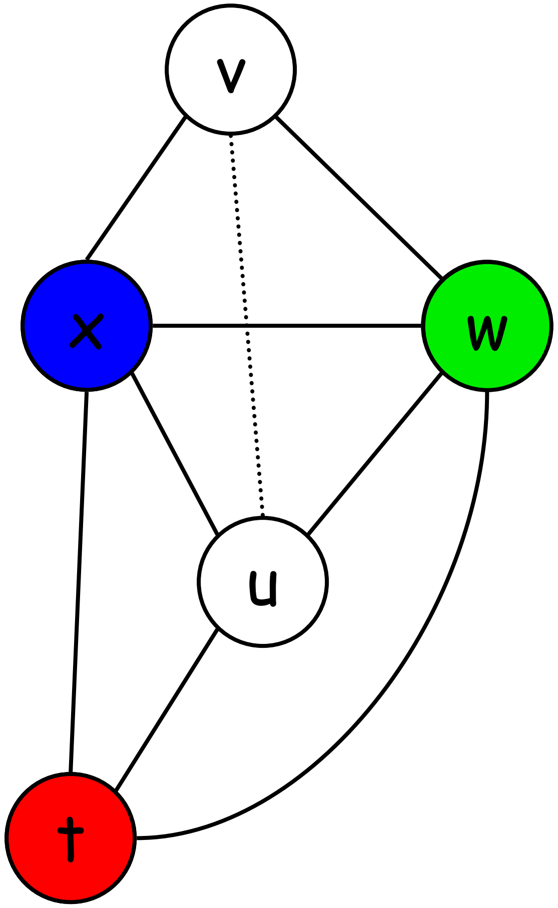


So, we need to spill

An Example, k=3

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





What to spill? Why?

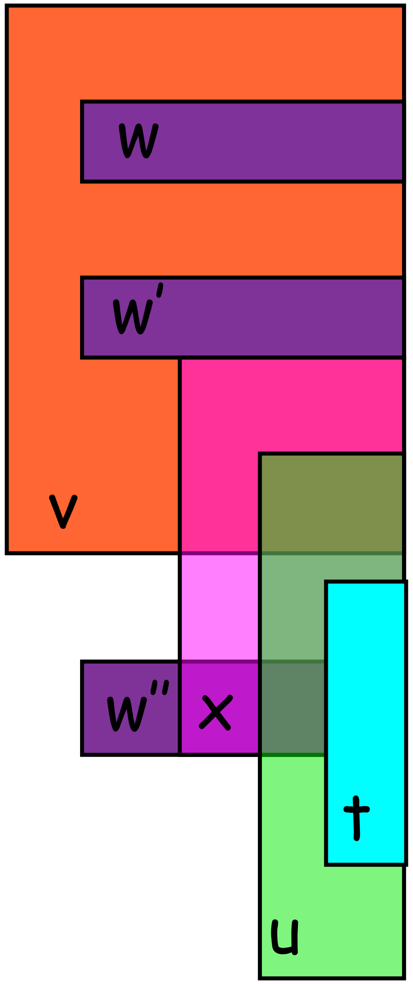
An Example, k=3

Choose w and Rewrite program

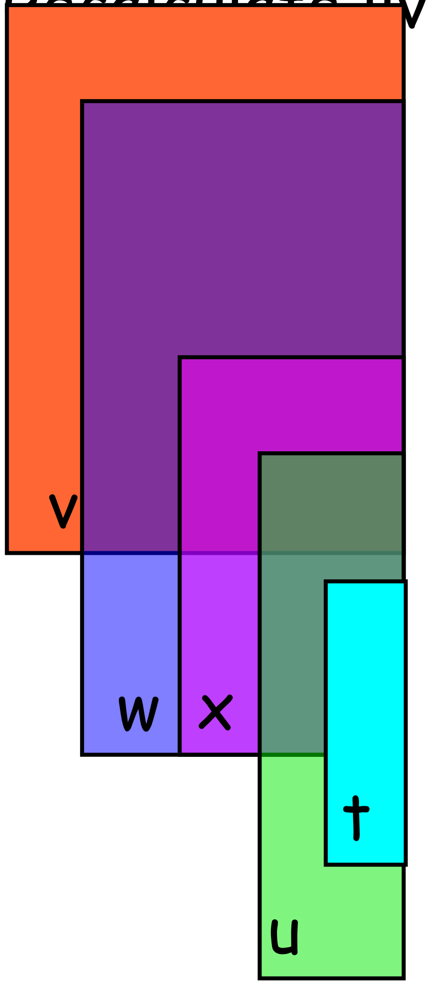
```
v   ←  1
w   ←  v + 3
M[] ←  w
w′  ←  M[]
x   ←  w′ + v
u   ←  v
t   ←  u + v
w″  ←  M[]
    ←  w″ + x
    ←  t
    ←  u
```

An Example, k=3

```
v   ←   1
w   ←   v + 3
M[] ←  w
w′  ←  M[]
x   ←   w′ + v
u   ←   v
t   ←   u + v
w″  ←  M[]
    ←   w″ + x
    ←   t
    ←   u
```



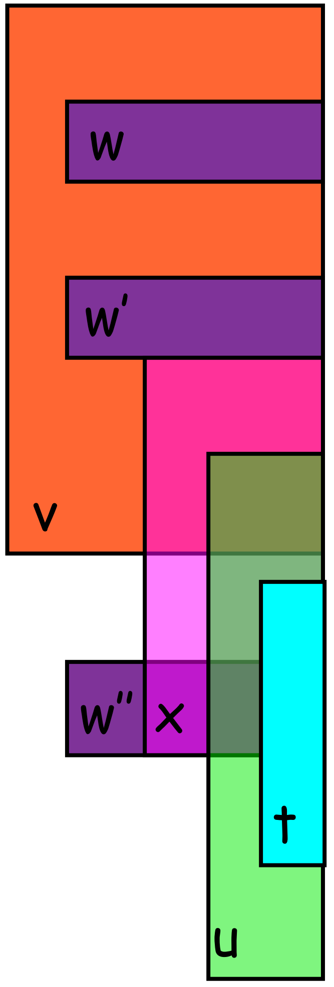
Recalculate live ranges

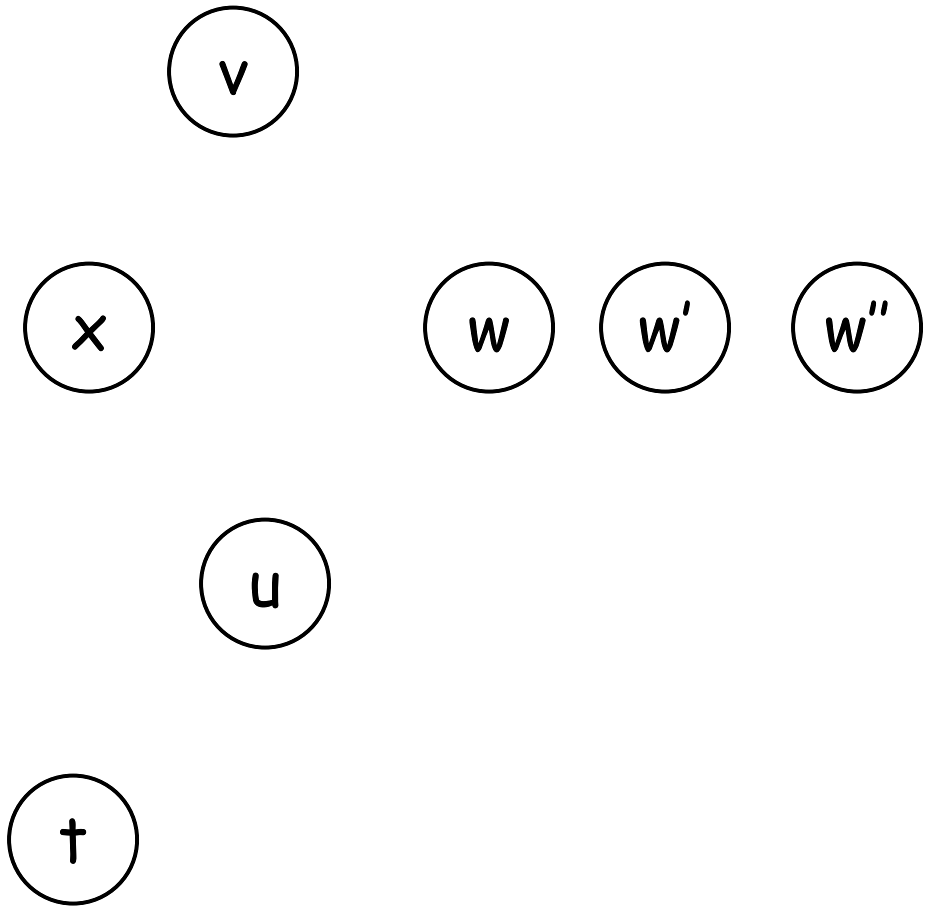


Spilling reduces live ranges, which decreases register pressure.

An Example, k=3

```
v   ←  1
w   ←  v + 3
M[] ←  w
w′  ←  M[]
x   ←  w′ + v
u   ←  v
t   ←  u + v
w″  ←  M[]
    ←  w″ + x
    ←  t
    ←  u
```

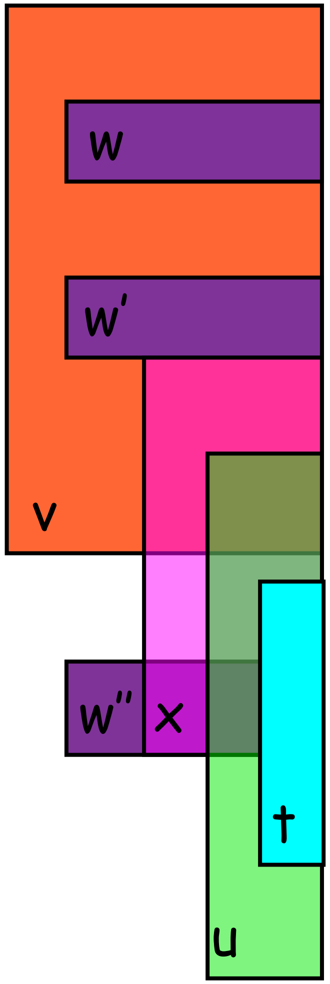


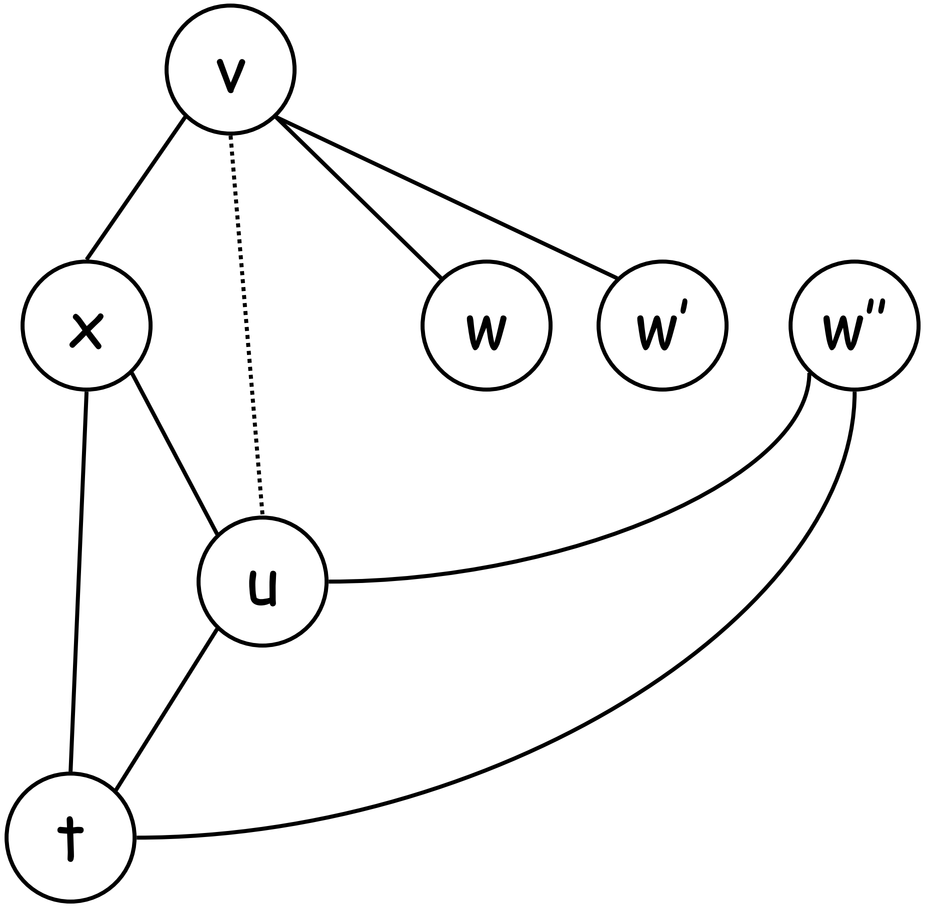


Recalculate interference graph

An Example, k=3

```
v   ←  1
w   ←  v + 3
M[] ← w
w′  ← M[]
x   ←  w′ + v
u   ←  v
t   ←  u + v
w″  ← M[]
    ←  w″ + x
    ←  t
    ←  u
```

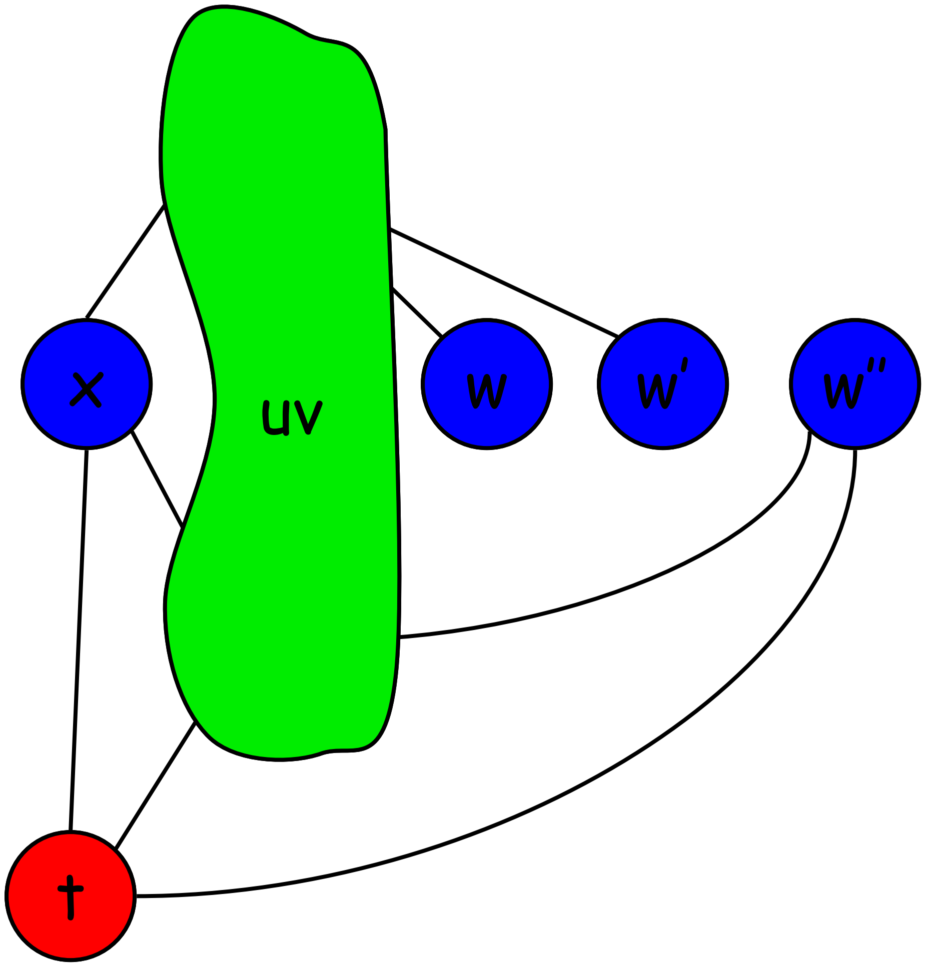




Recalculate interference graph

An Example, k=3

```
v   ←   1
w   ←   v + 3
M[] ← w
w′  ←  M[]
x   ←   w′ + v
u   ←   v
t   ←   u + v
w″  ←  M[]
    ←   w″ + x
    ←   t
    ←   u
```



Recolor

Graph coloring

- Once we have an interference graph, we can attempt register allocation by searching for a K-coloring
- This is an NP-complete problem (for K>2)
- But a linear-time simplification algorithm (by Kempe, 1879) tends to work well in practice

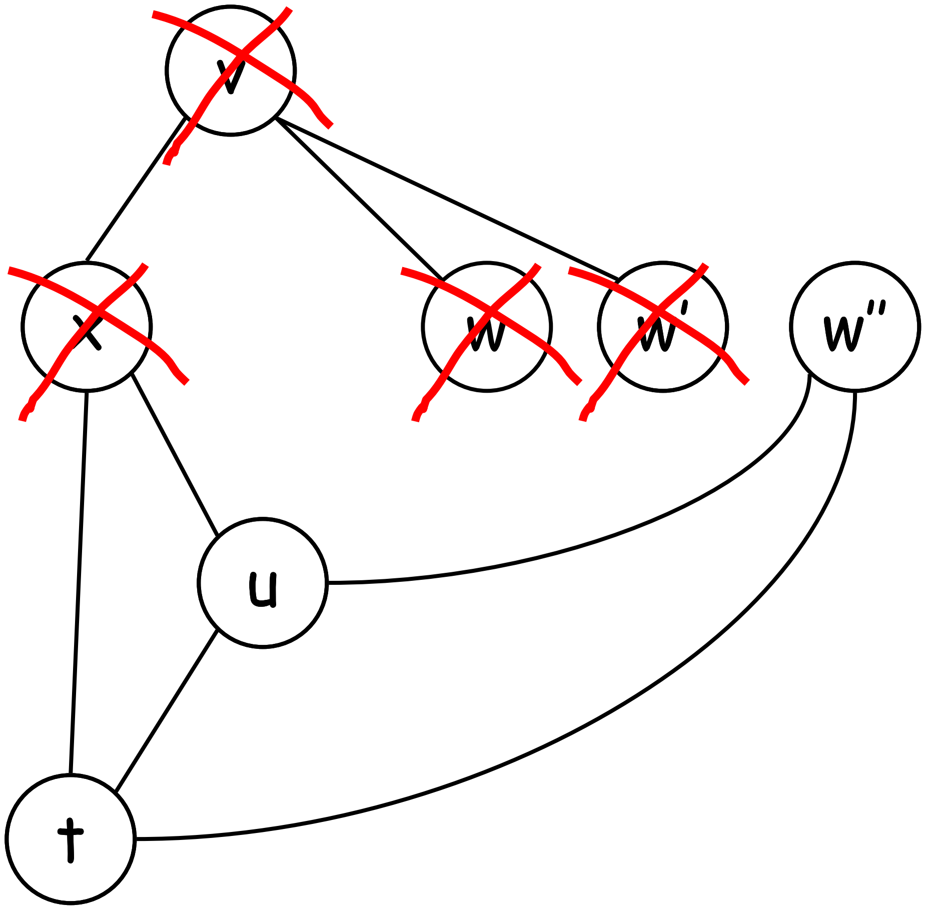
Kempe’s observation

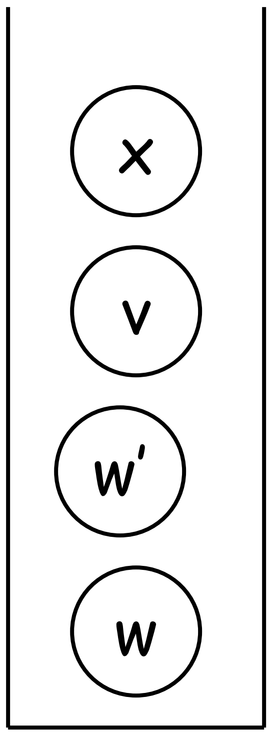
- Given a graph G that contains a node n with degree less than K, the graph is K-colorable iff G with n removed is K-colorable
  - This is called the “degree<K” rule
- So, let’s try iteratively removing nodes with degree<K
- If all nodes are removed, then G is definitely K-colorable

Kempe’s algorithm

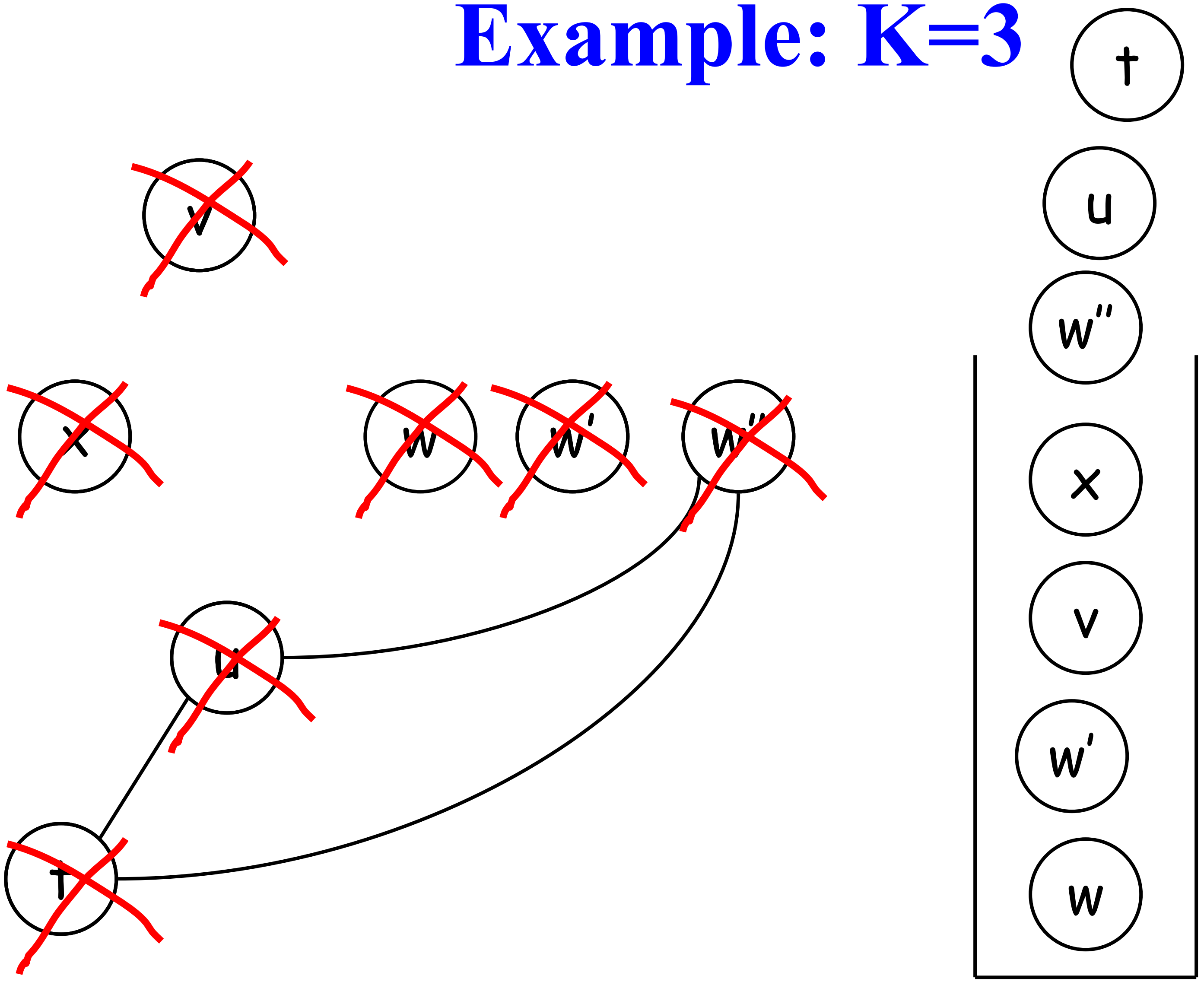
- First, iteratively remove degree<K nodes, pushing each onto a stack
- If all get removed, then pop each node and rebuild the graph, coloring as we go
- If we get stuck (i.e., no degree<K nodes), then remove any node and continue

Example: K=3

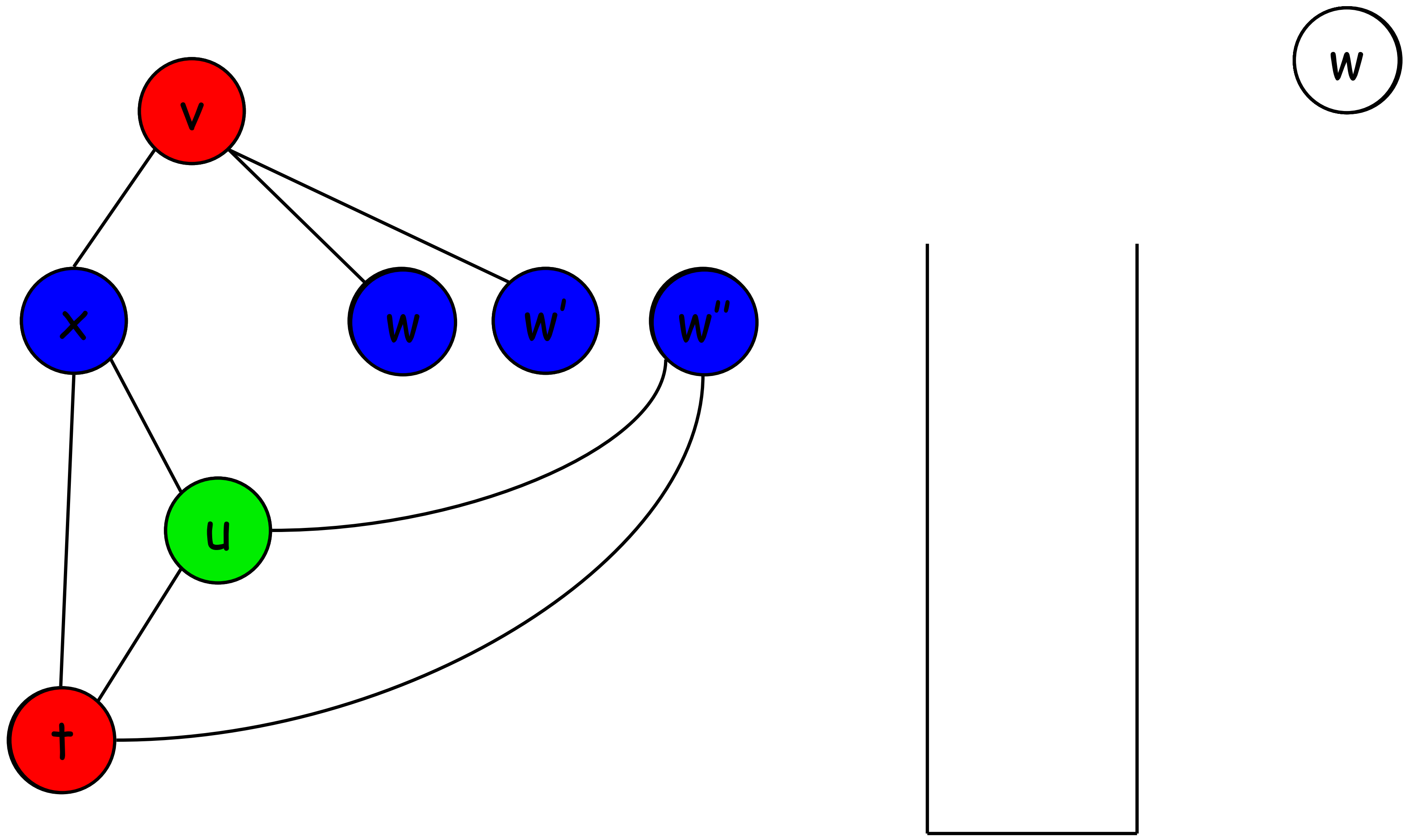




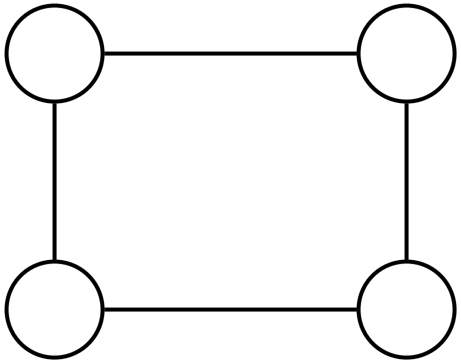
Example: K=3

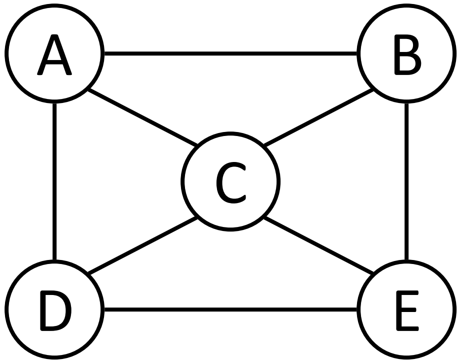


Example: K=3



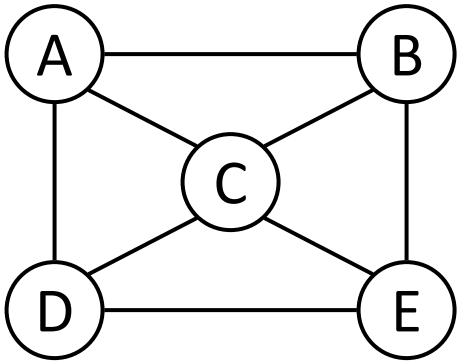
Alg not perfect





What should we do when there is no node of degree < k?

Optimisitic Coloring



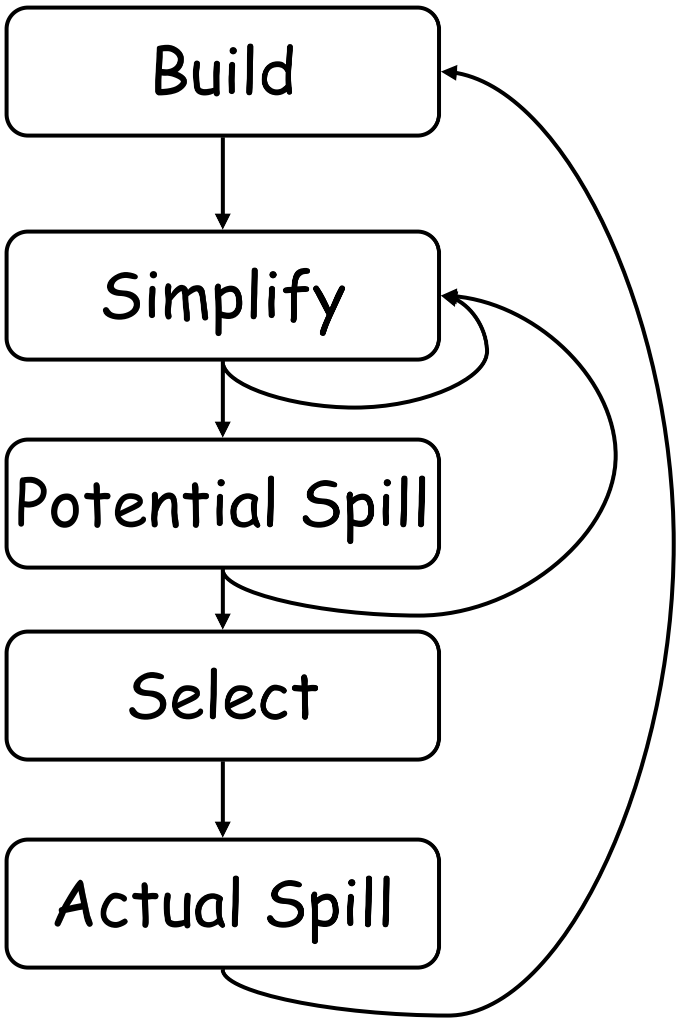
Chaitin's allocator

- 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 and continue
- Start over: if there are actual spills, generate spill code and then start over

Choosing potential spills

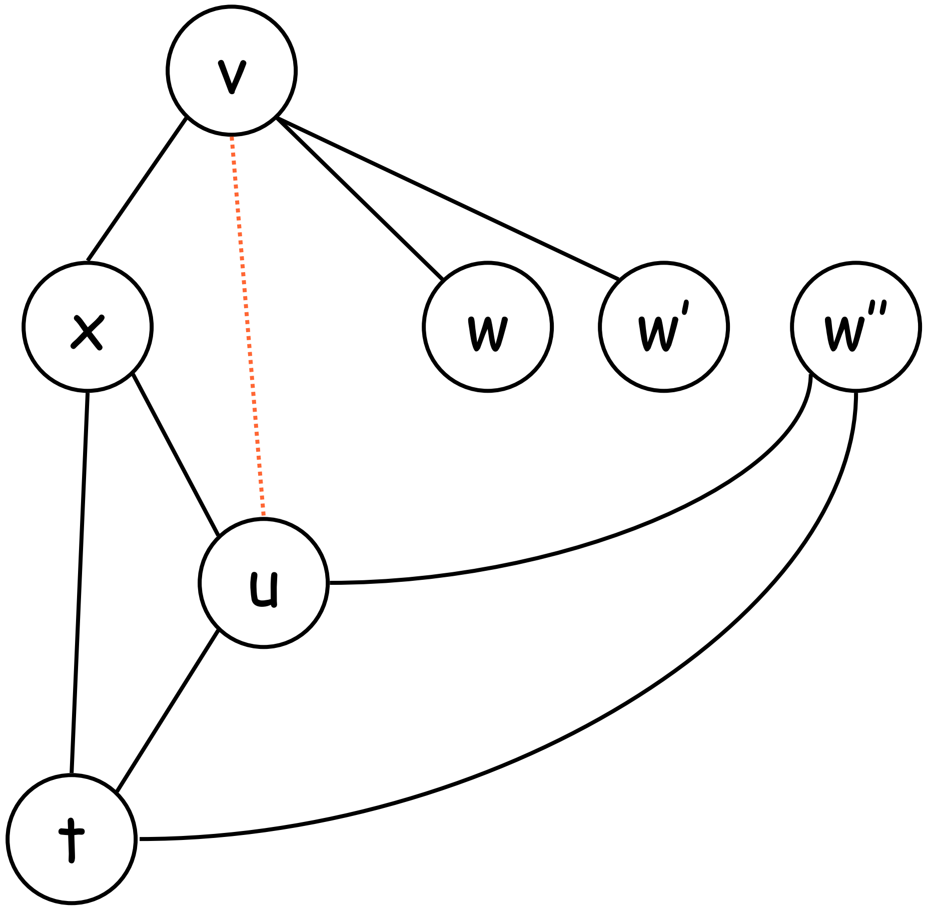
- When choosing a node to be a potential spill, we want to minimize its performance impact
- Can attempt to compute a spill cost for each temp
  - by estimating performance cost
  - or by using actual profile information
- More on this later...

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
```

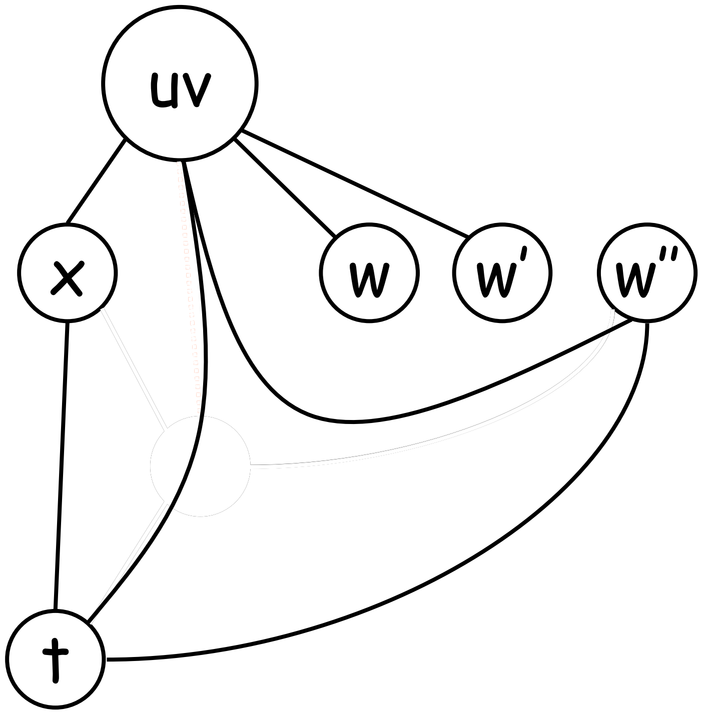


Can u & v be coalesced?
Should u & v be coalesced?

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



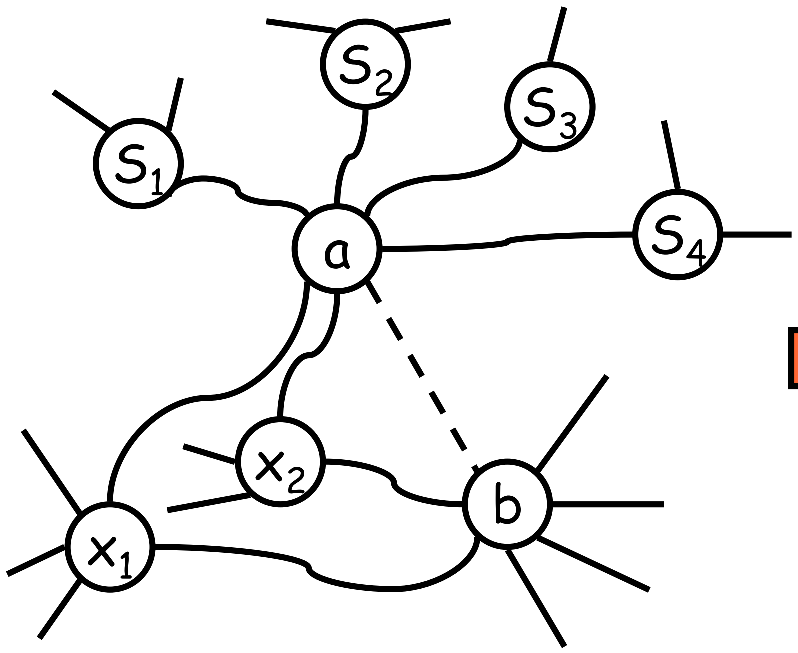
Preston

- Can coalesce a and b if

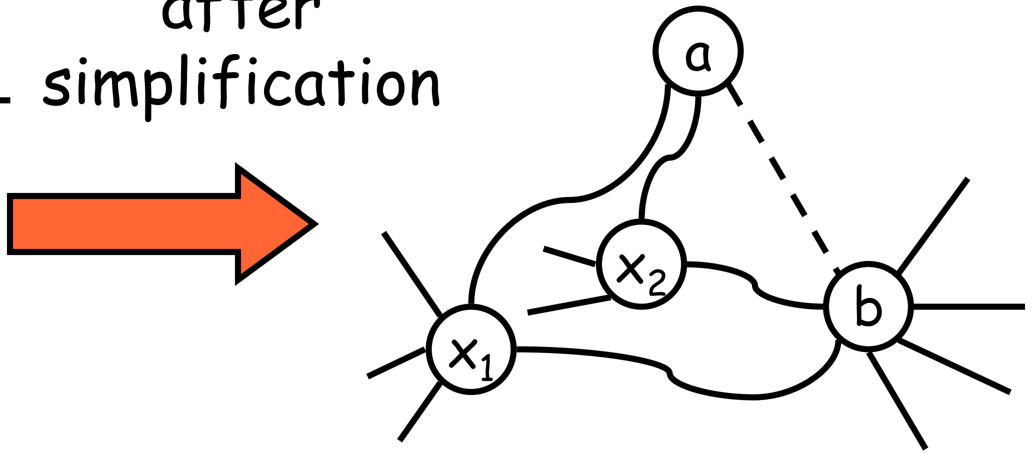
  foreach neighbor t of a
  - t interferes with b, or,
  - degree of t < k

- 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

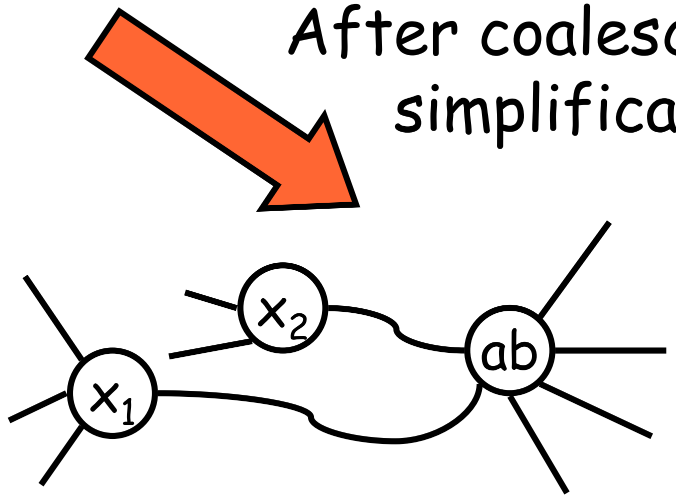
Preston



No coalescing, after simplification



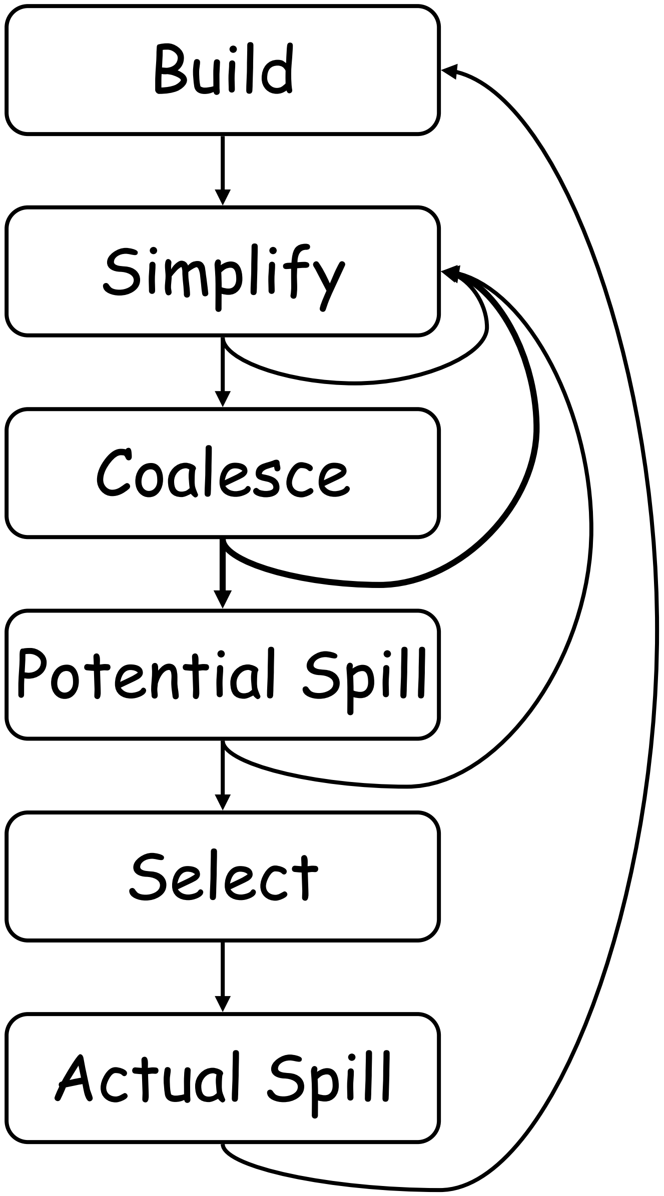
After coalescing and simplification



Why Two Methods?

- With Briggs one needs to look at all neighbors of a & b
- With Preston, only need to look at neighbors of a.
- We need to insert hard registers in graph and they will have LARGE adjacency lists.
- So
  - Precolored nodes have infinite degree
  - No other precolored nodes in adj list
  - Use Preston if one of a & b is precolored
  - Use Briggs if both are temps

Where We Are



Actually, one more step: Freeze

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

What about special registers?

- Instructions with register requirements

```
d ← a * b

          ret x
```

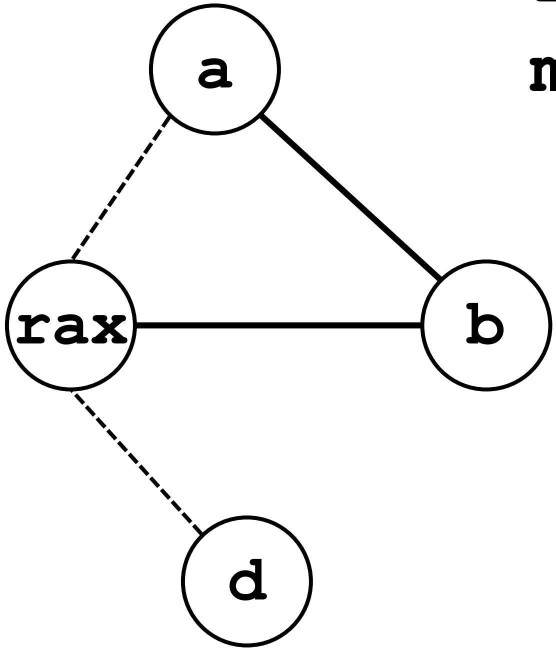
- Callee-save registers
  - x86-64: **`RDI`**, **`RSI`**, **`RDX`**, **`RCX`**, **`R8`**, **`R9`** must not be saved by callee if callee wants to use them.

What about special registers?

- Instructions with register requirements

```
d ← a * b
```

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

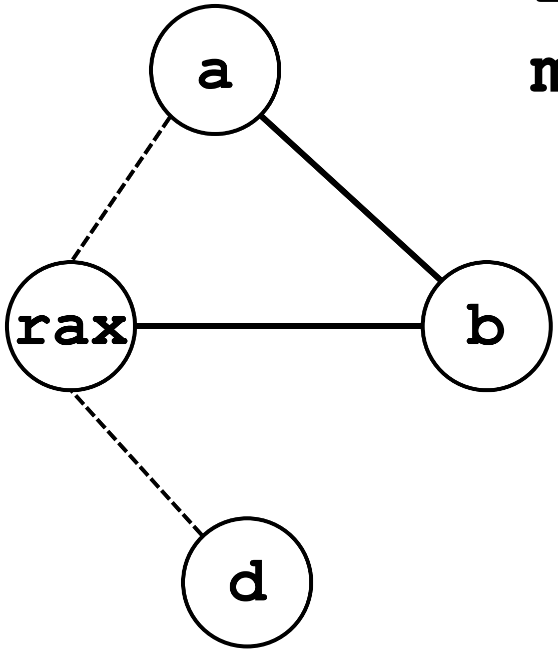


What about special registers?

- 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

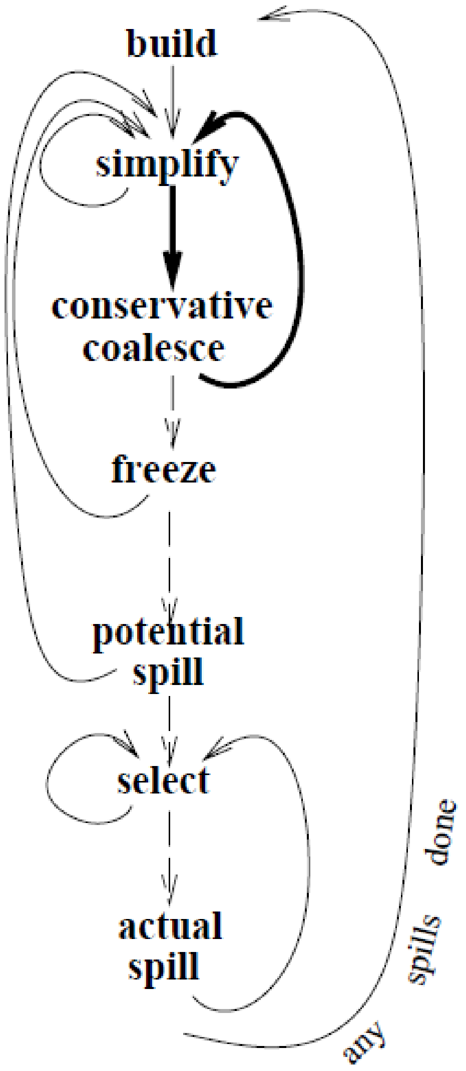
```
d ← a * b

      ⇨  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?

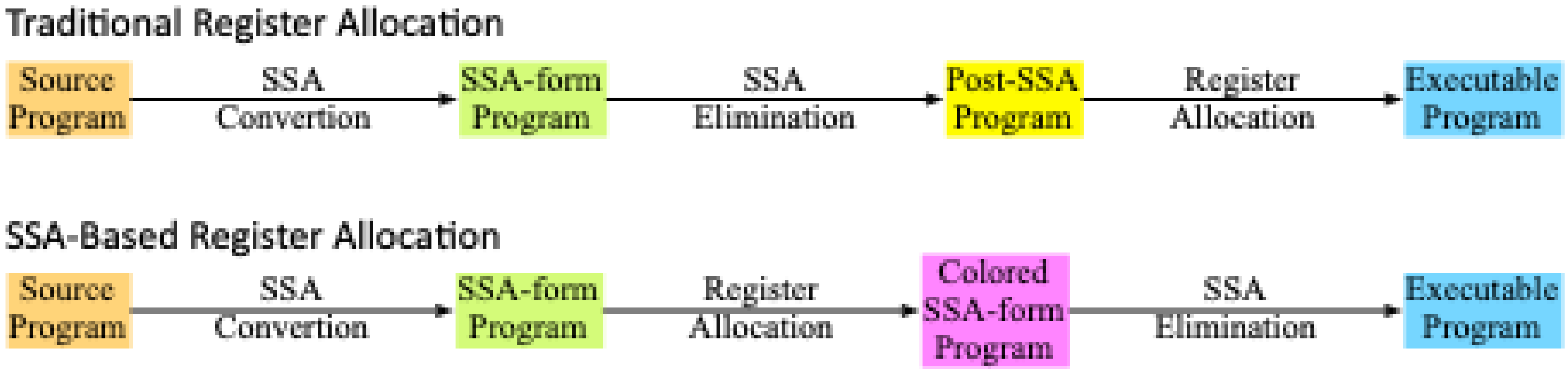
```
prologue:    define r
             t1 ← r
             …
epilogue:    r ← t1
             use r
```

Iterated Register Coloring



SSA-based Register Allocation

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



Simplify/Select: A particular order

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

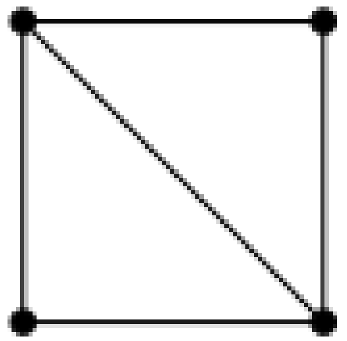
- Greedy Coloring:
  - input: G=(V,E)
    an **ordered sequence $v_1, \ldots, v_n$**
  - output: Assignment col:$V \rightarrow \{0, \ldots, \Delta(G)\}$

```
for i ← 1 to n do
    let c be lowest color not used in $N(v_i)$
    set $col(v_i) \leftarrow c$
```

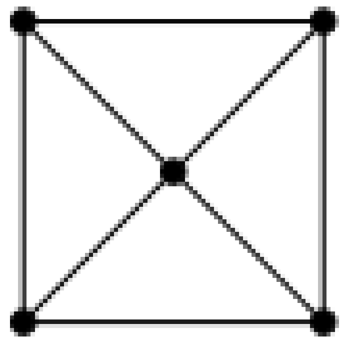
Chordal Graphs

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

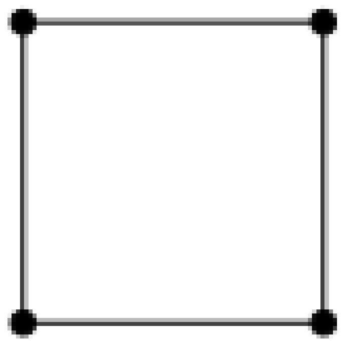
(a)



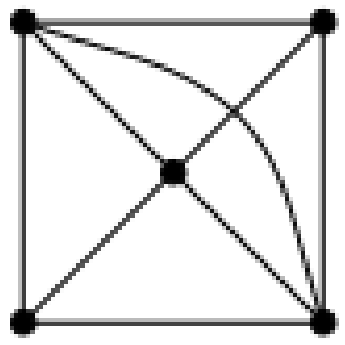
(b)



(c)

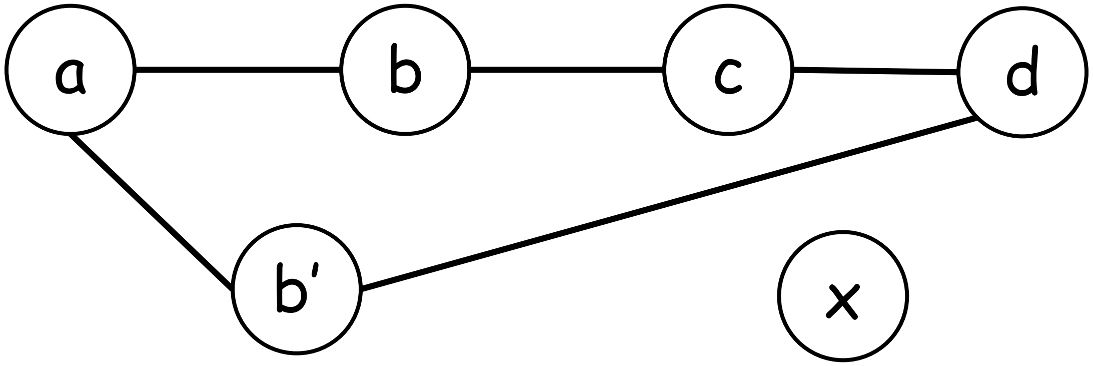


(d)



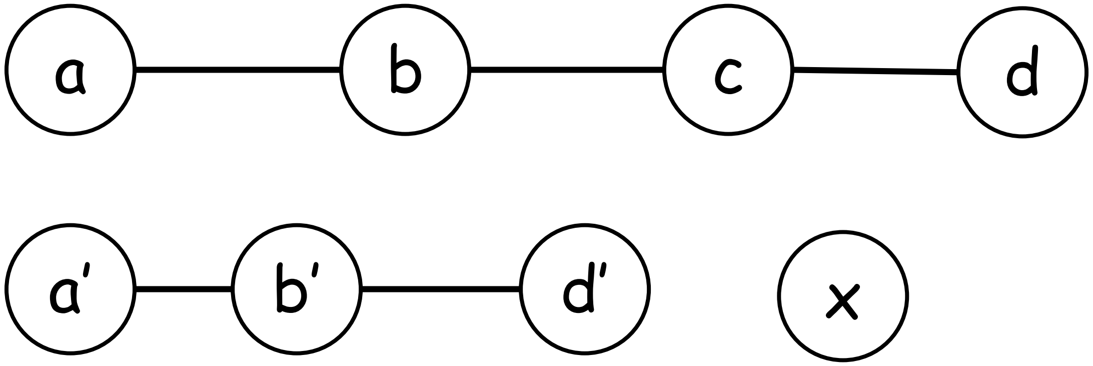
Non-chordal example

```
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
```

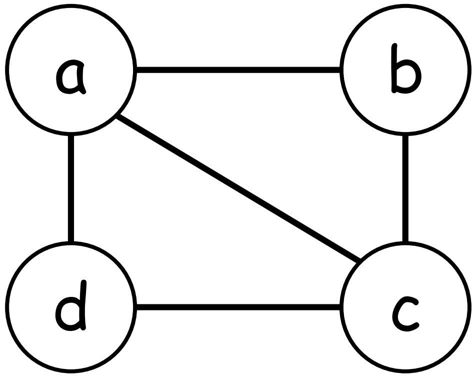


Adding more temps → fewer registers!

BTW: now in SSA-form!

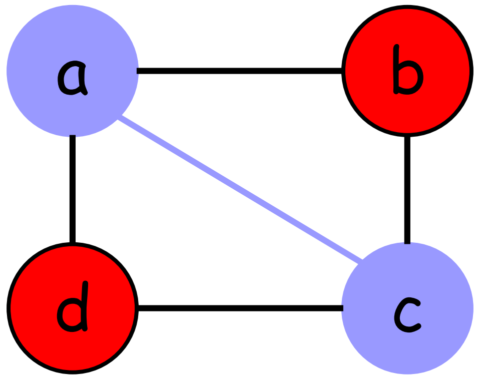
Simplical Elimination Ordering

- 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



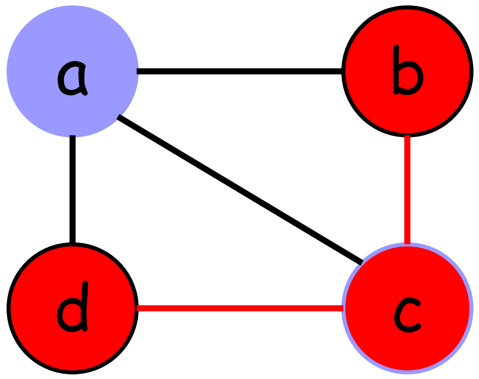
Simplical Elimination Ordering

- 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



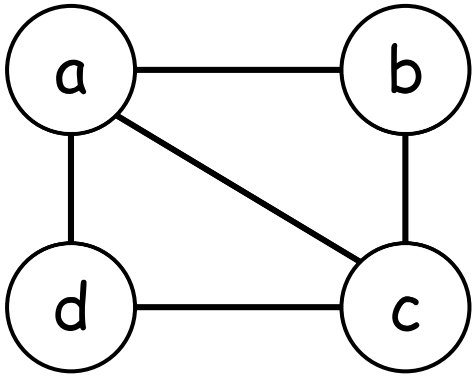
Simplical Elimination Ordering

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



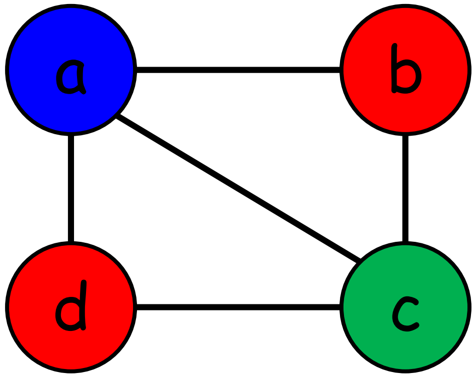
b, a, c, d

Next Time

- Optimality of SEO
- Creating an SEO
- Chordal Coloring and
  - Spilling
  - Coalescing
- Liveness Analysis
- SSA
- Phi-nodes
- Finish Chordal Register Allocation

Greedy Coloring using SEO is optimal

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



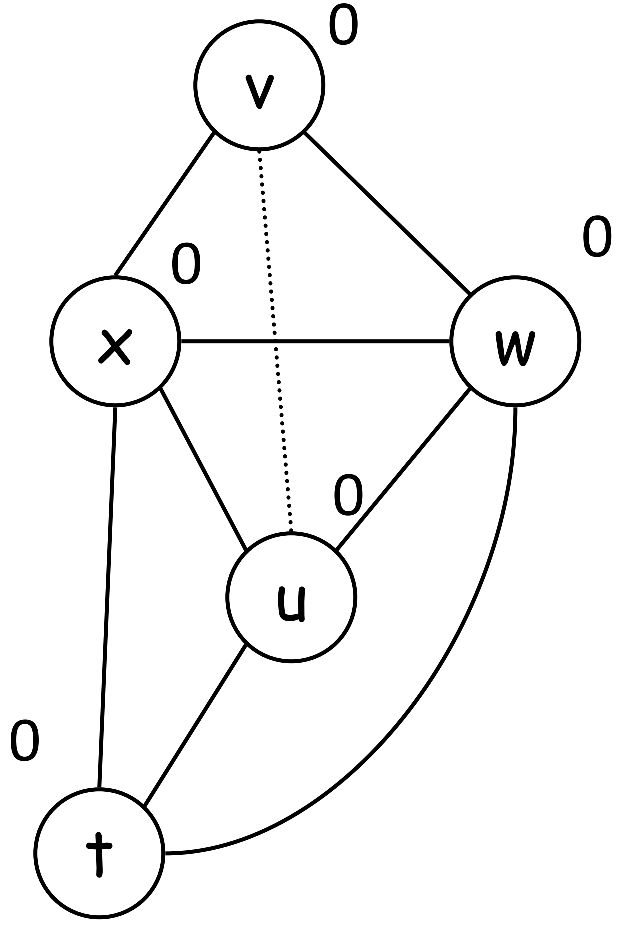
b, a, c, d

Maximal Cardinality Search

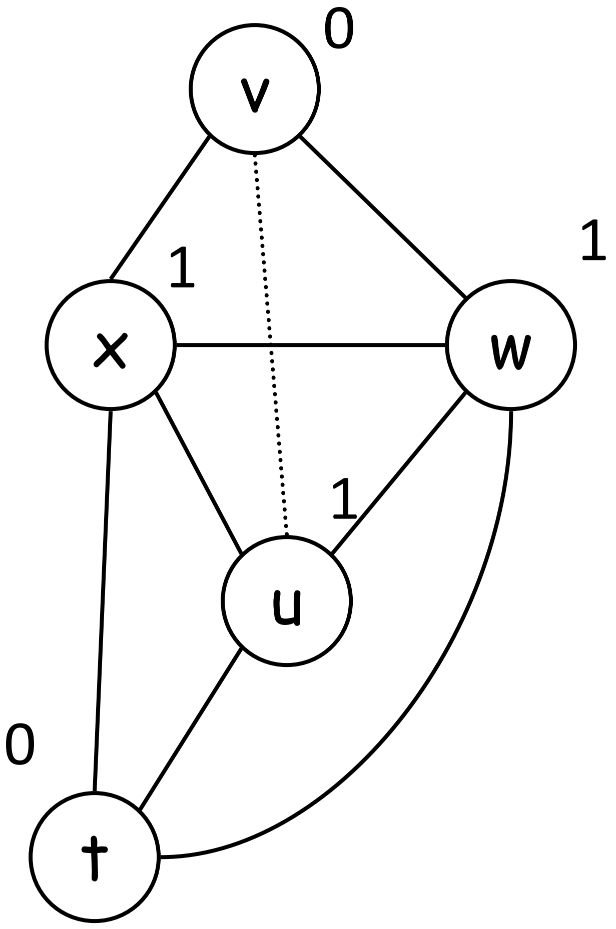
```
Maximum Cardinality Search
  input: G = (V, E) with |V| = n
  output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
  for all $v \in V$ do $\lambda(v) \leftarrow 0$
  for $i \leftarrow 1$ to n do
    let $v \in V$ be a node such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ in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

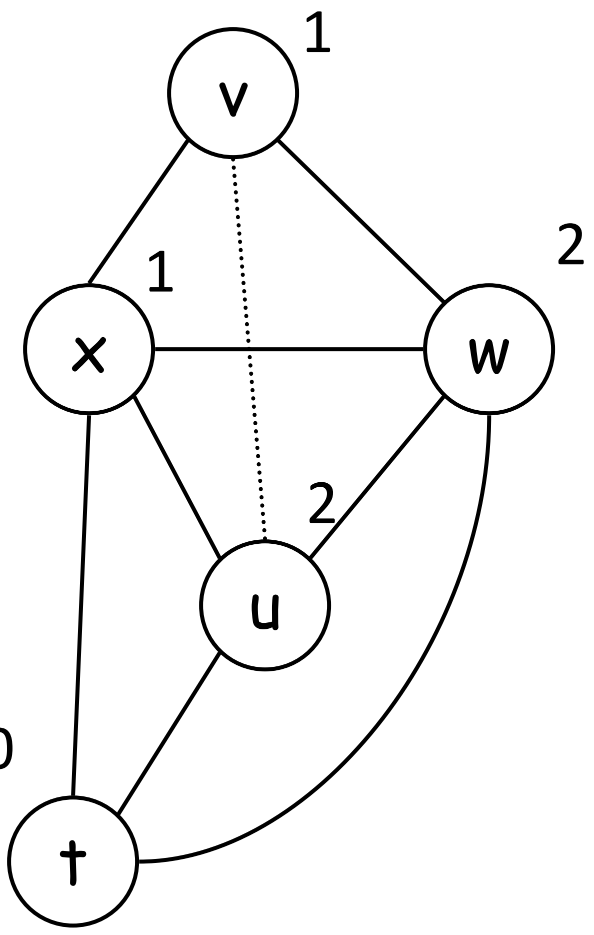


```
v ←  1
w ←  v + 3
x ←  w + v
u ←  v
t ←  u + x
  ←  w
  ←  t
  ←  u
```



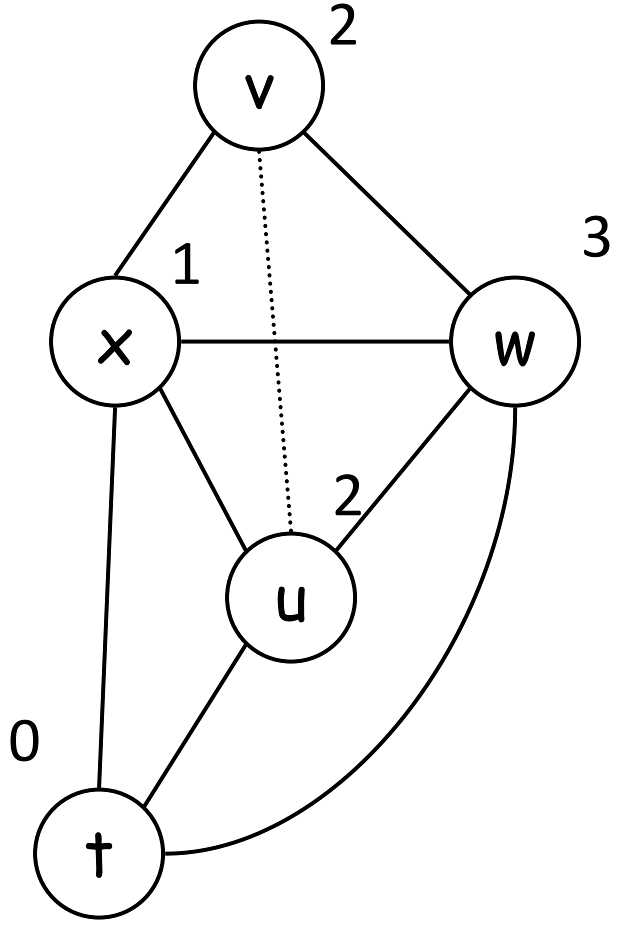
SEO: t

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



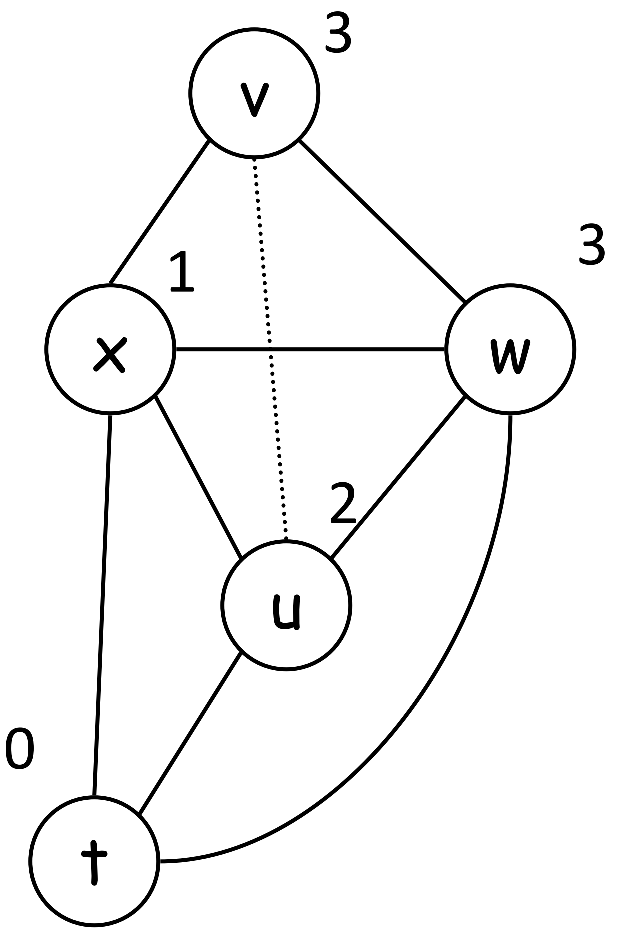
SEO: t, x

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



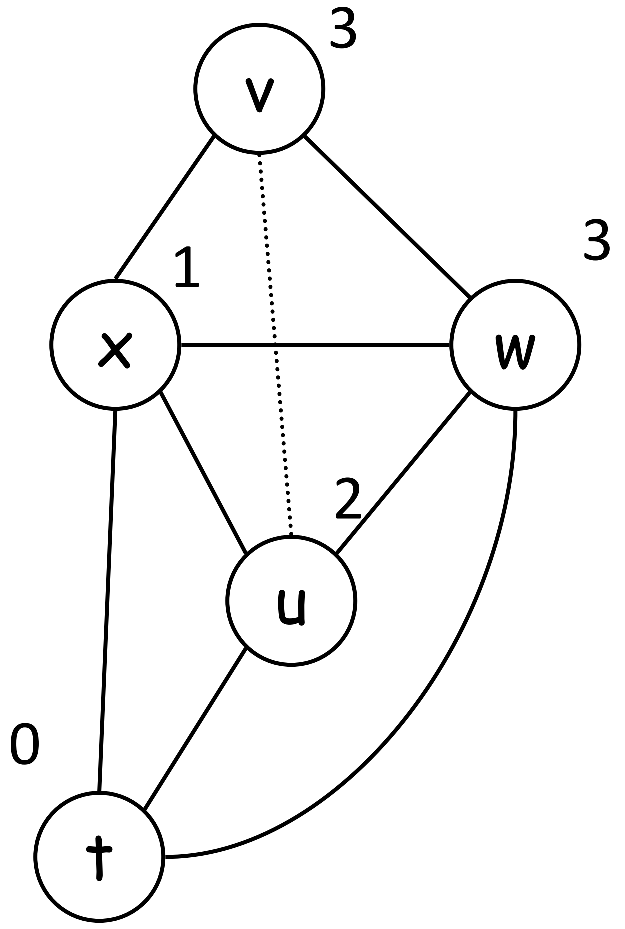
SEO: t, x, u

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



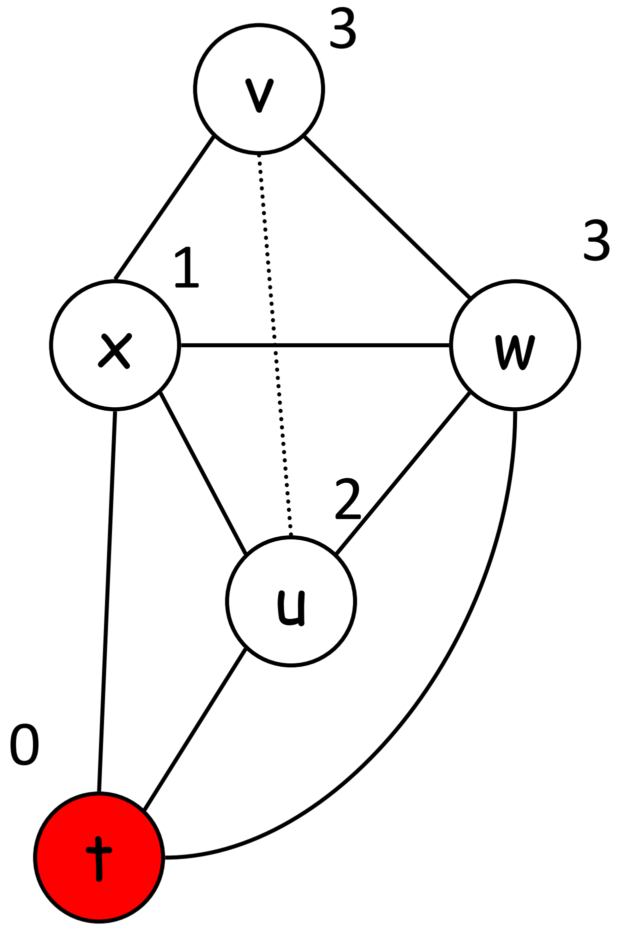
SEO: t, x, u, w

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



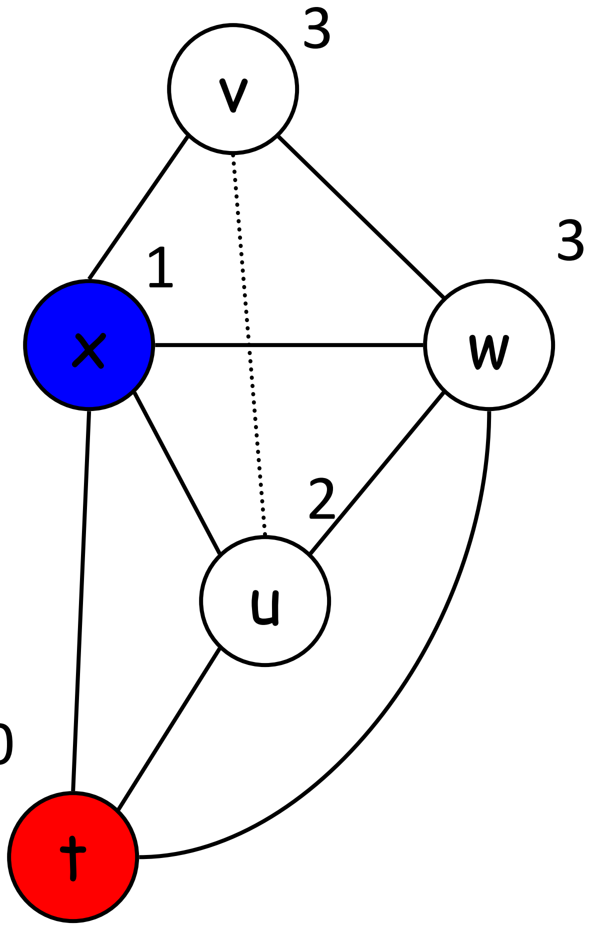
SEO: t, x, u, w, v

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



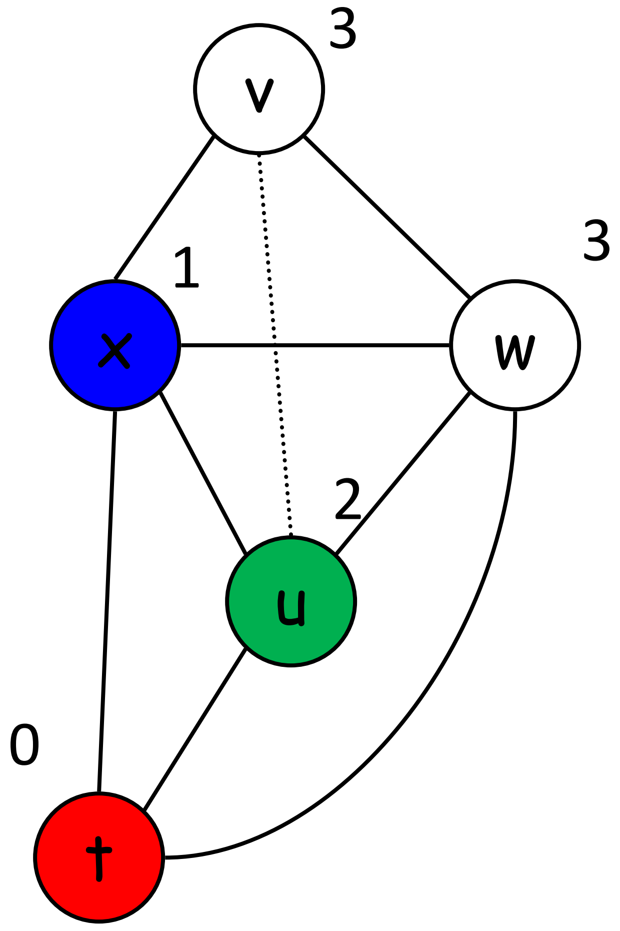
SEO: t, x, u, w, v

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



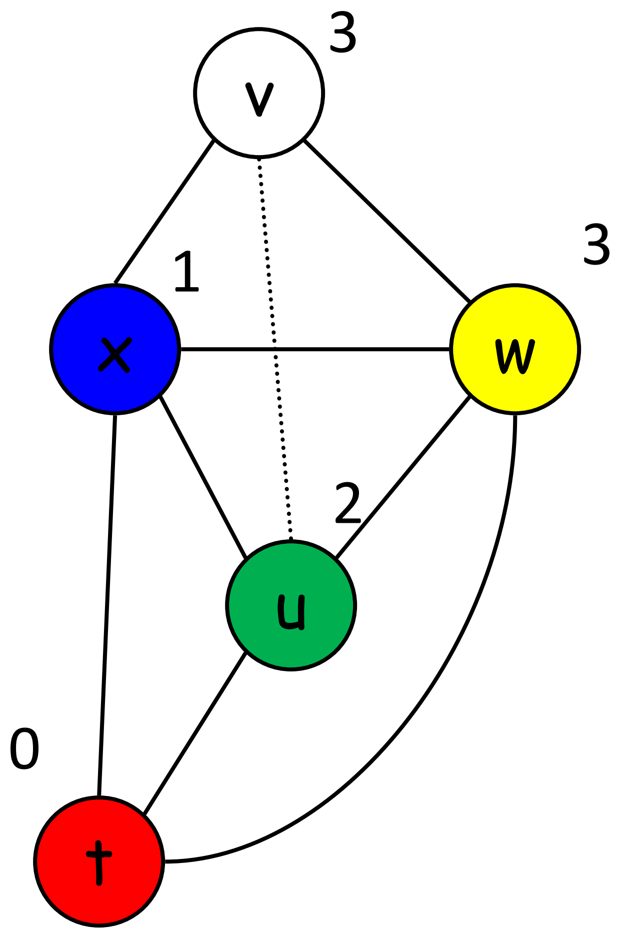
SEO: t, x, u, w, v

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



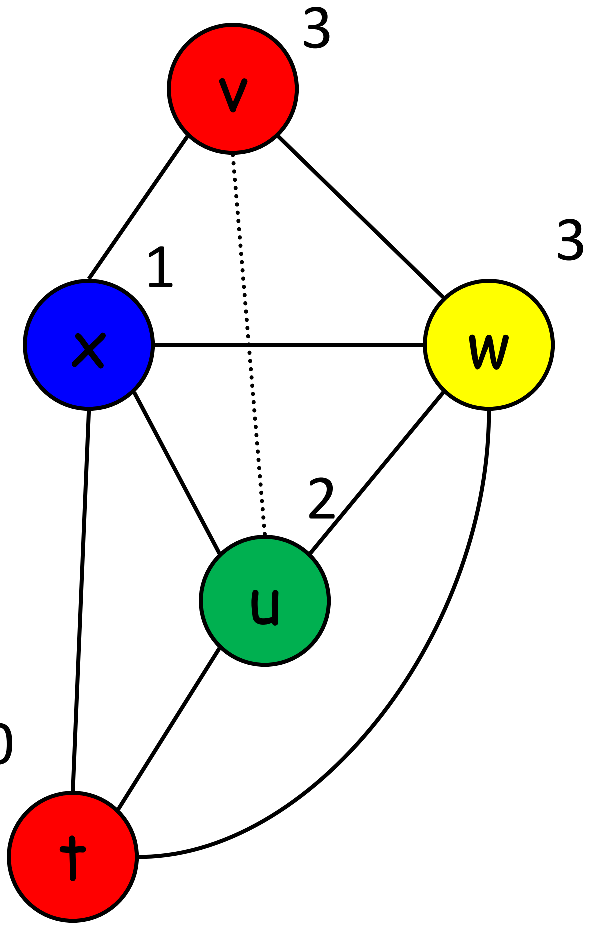
SEO: t, x, u, w, v

```
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 ith 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.

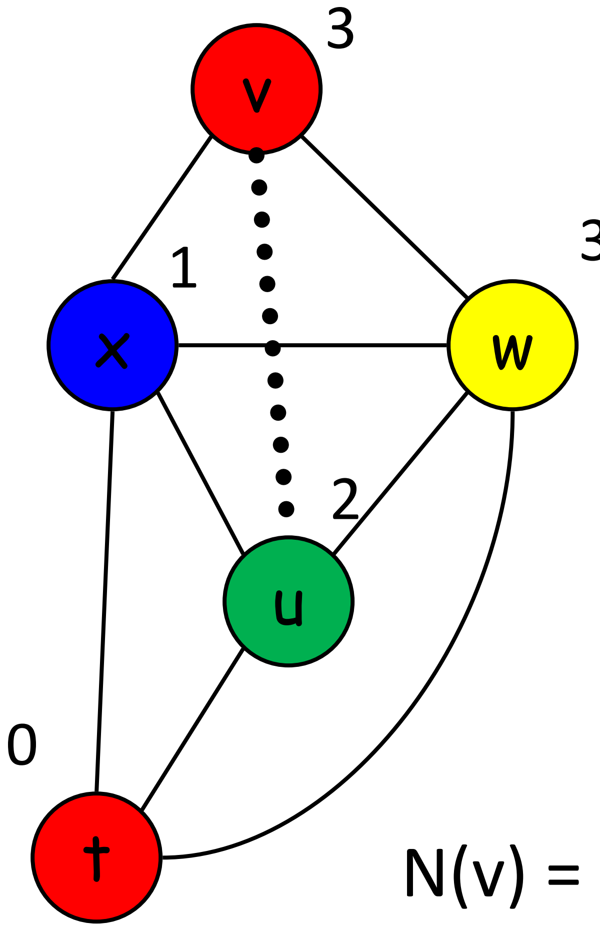
I.e., The chromatic number of a chordal graph is the size of largest clique

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 ∈ 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
```

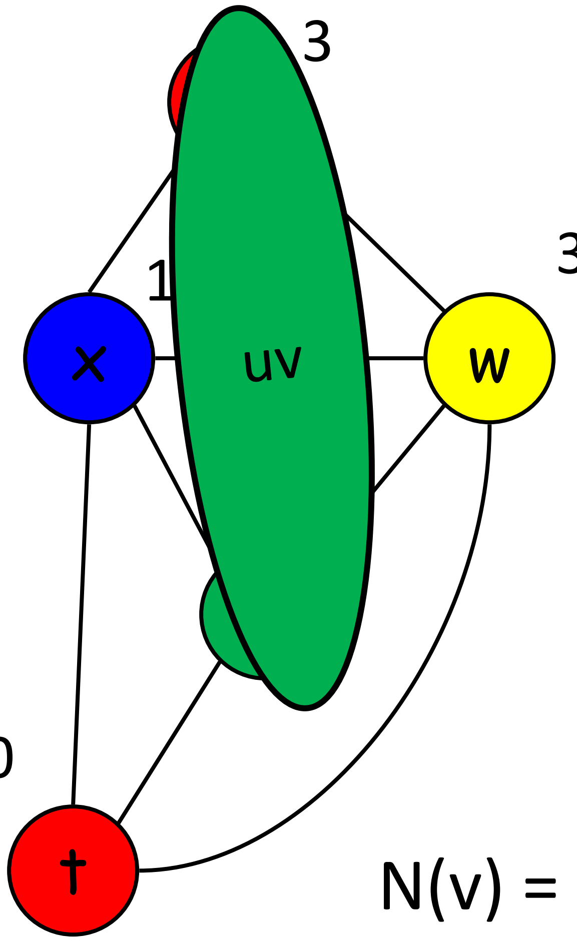


N(v) = { x, w }

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 }

Next Time

- Liveness Analysis
- SSA
- Phi-nodes
- Finish Chordal Register Allocation

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
  - Preform BestEffortCoalescing

Next Time

- Liveness Analysis
- SSA
- Phi-nodes
- Finish Chordal Register Allocation