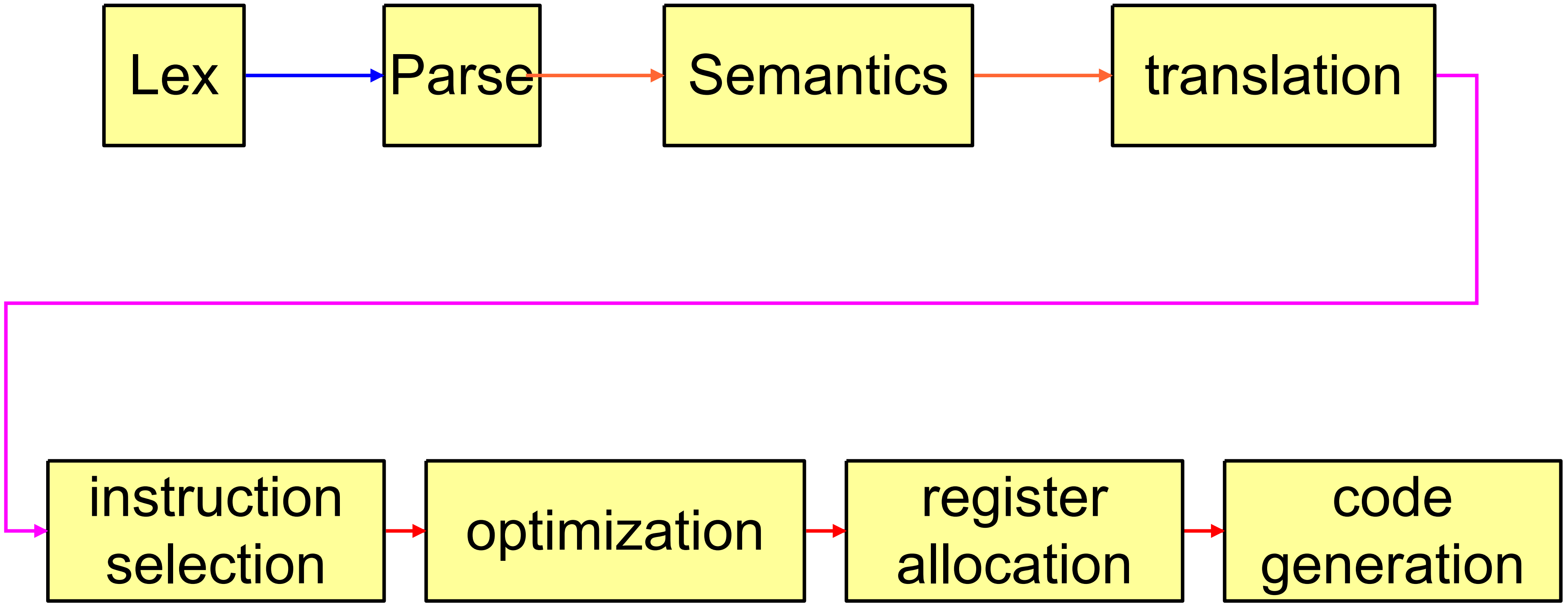
Register Allocation

15-411/15-611 Compiler Design

Seth Copen Goldstein

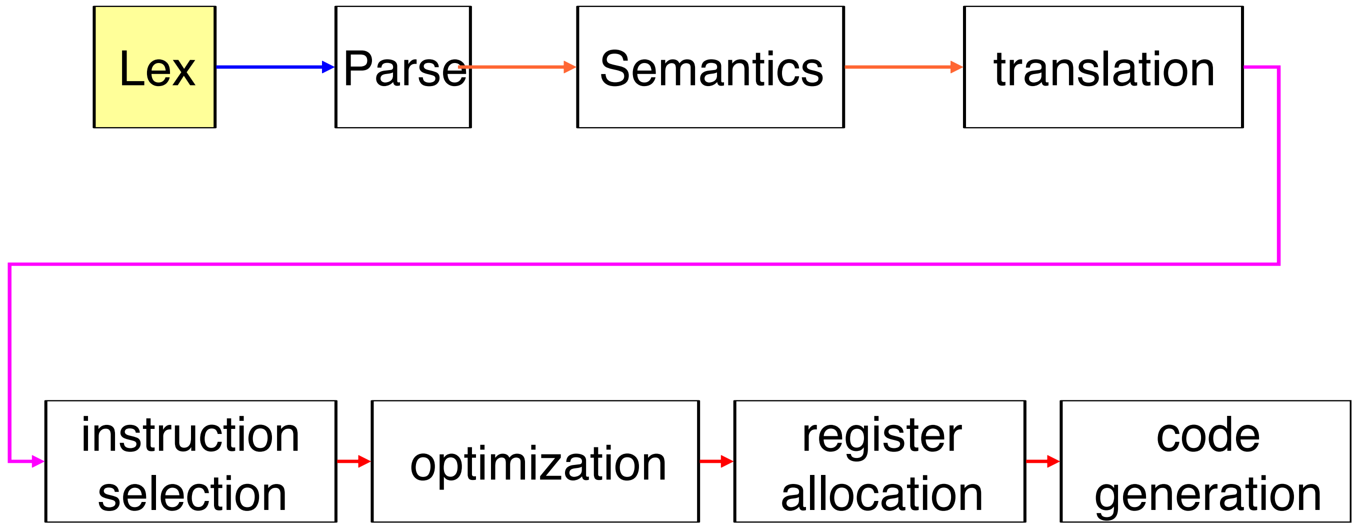
January 16, 2025

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

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** ereturn

e := c constant

| v variable

| e_1 ? e_2 binary operation

? := **+ | - | * | / | %**

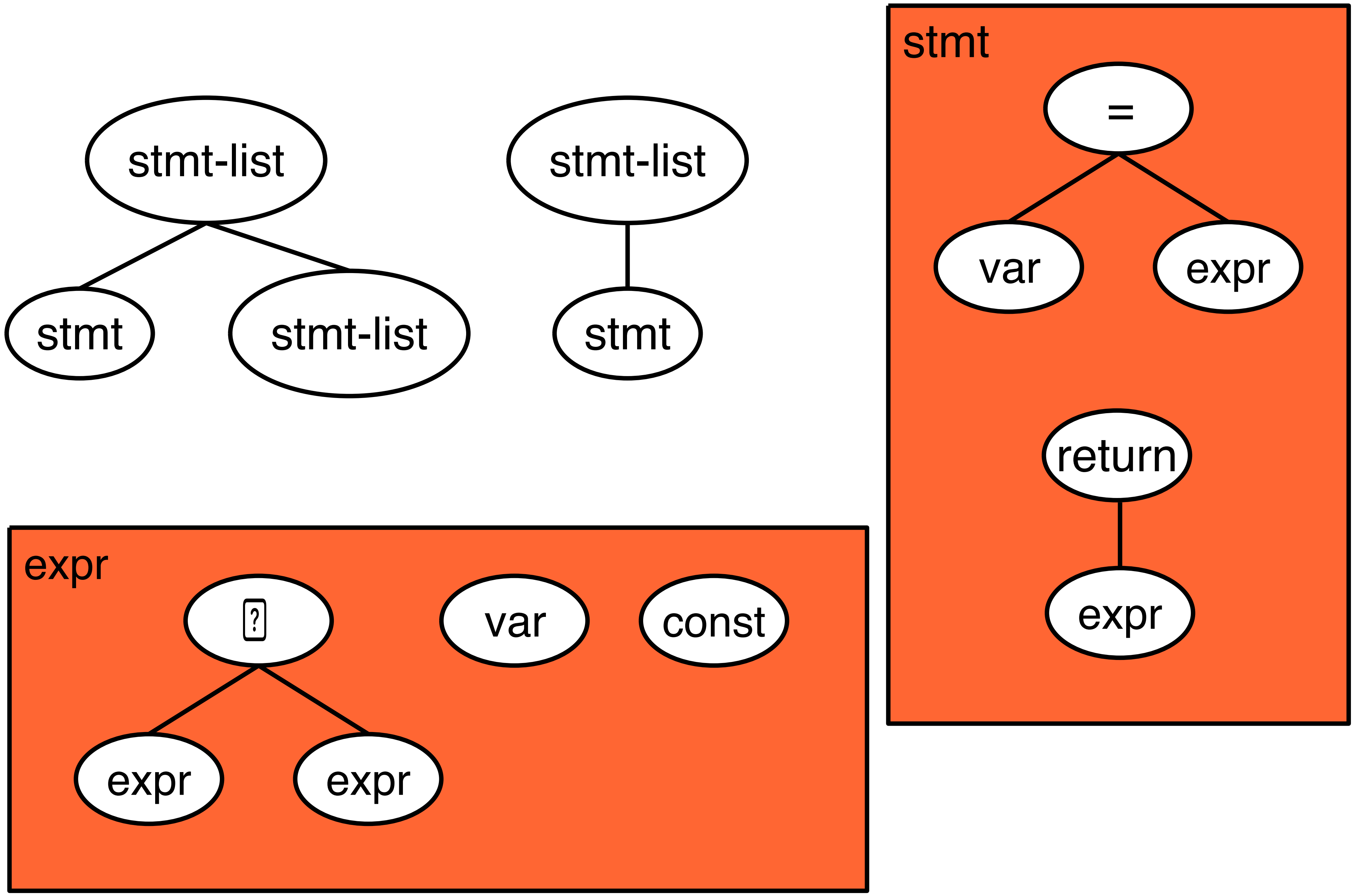
x=8

x=y+1

Ambiguity?

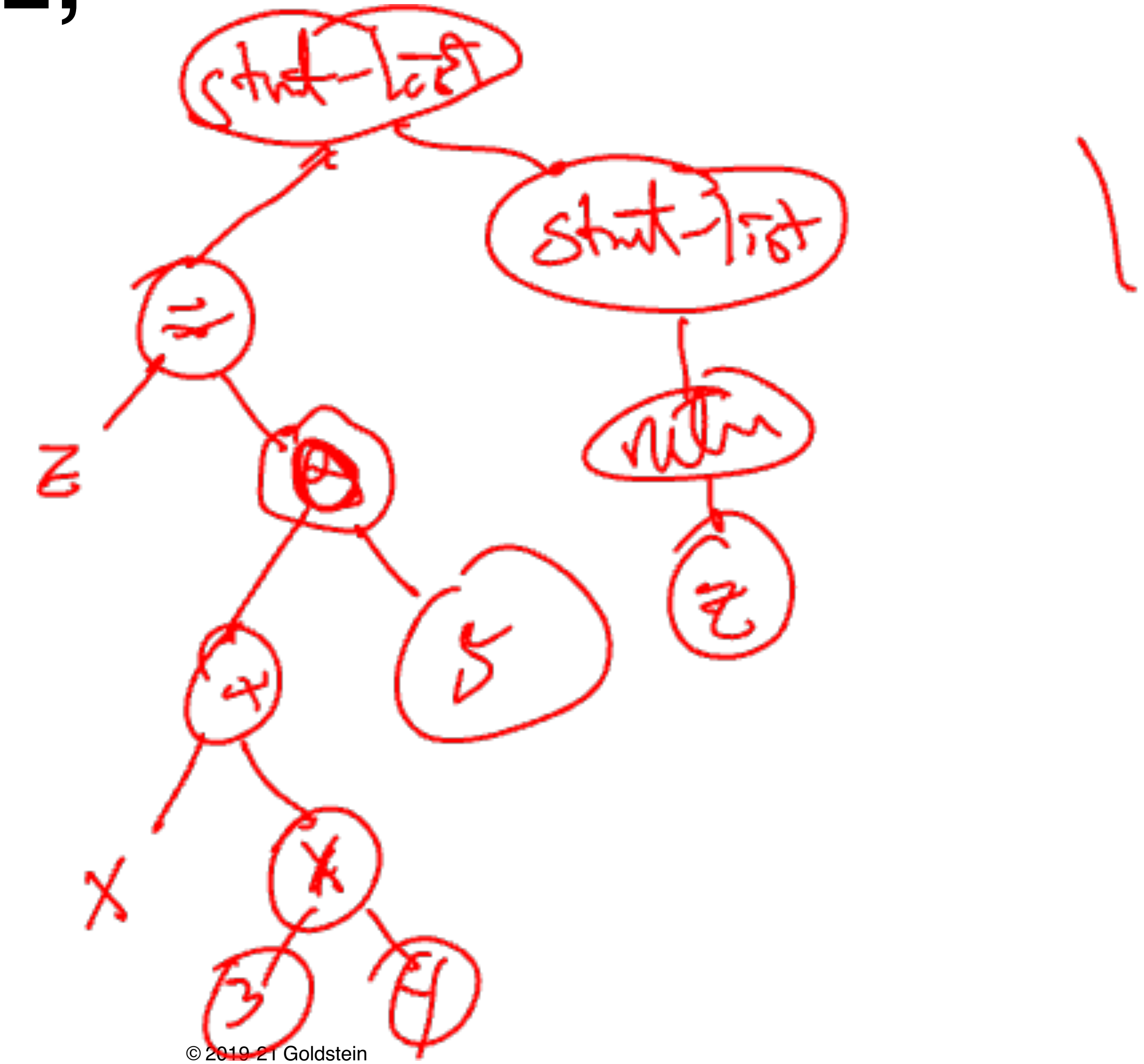
Semantics?

Abstract Syntax Tree



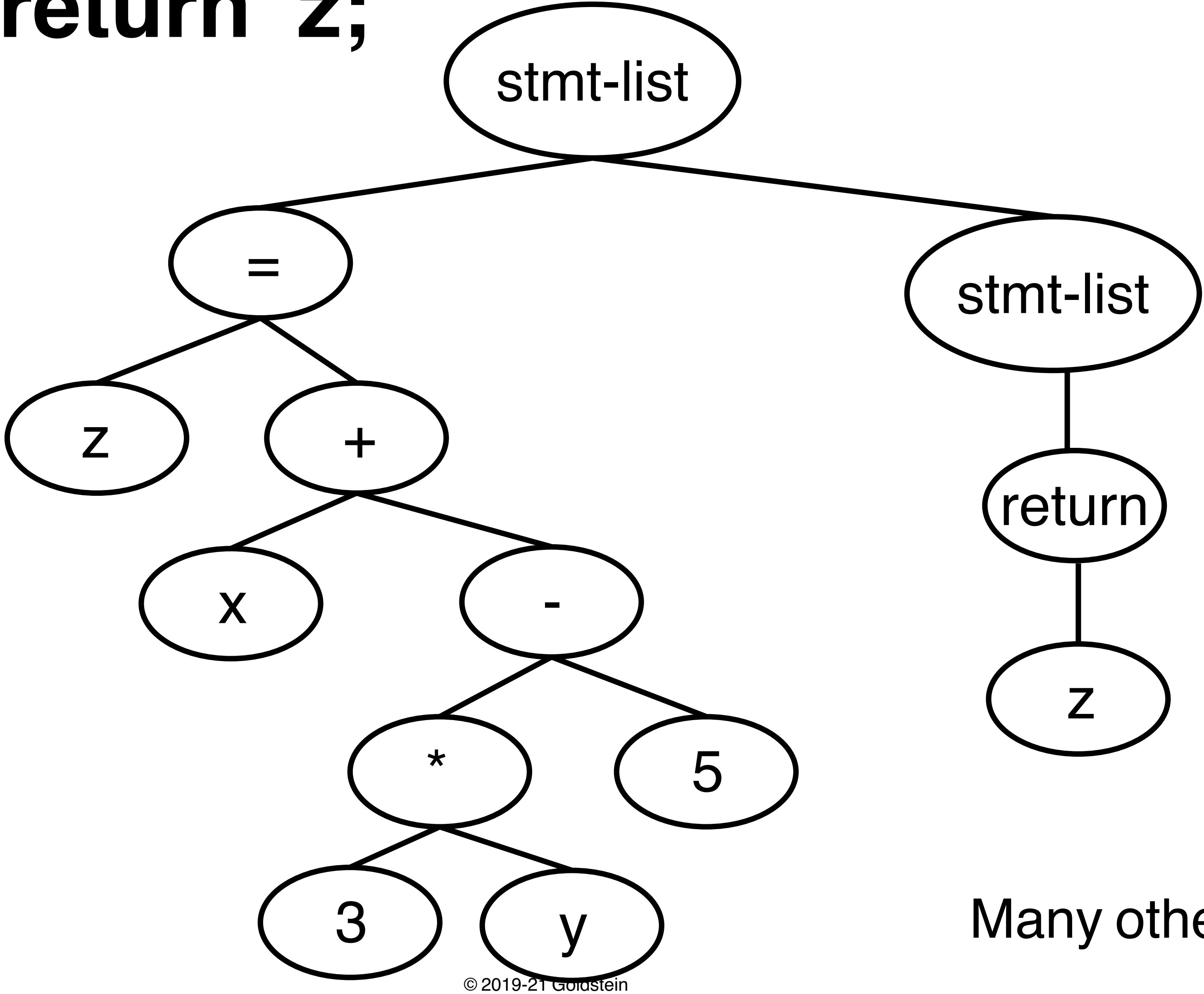
Example

```
z = x + 3 * y – 5;
return z;
```



Possible parse tree

```
z = x + 3 * y – 5;
return z;
```



Many other possibilites

Abstract Assembly as IR

- Lowering of AST

$M[a] \leftarrow M[y]\ a\ M[B]$

- Facilitate
  - Analysis & optimizations
  - Translation to actual assembly
- Features:
  - Unlimited number of “temporaries” $\longrightarrow$ In today’s world aka registers
  - May (or may not) restrict how memory is used
  - Simple operations
  - May (or may not) restrict how constants are used
  - May specify certain “special registers”

Abstract Assembly as IR

- Features:
  - Unlimited number of “temporaries”
  - May (or may not) restrict how memory is used
  - Simple operations
  - May (or may not) restrict how constants are used
  - May specify certain “special registers”

dest ⍰ src_1 operator src_2

dest ⍰ operator src_1

~~operator~~

$M[x] \leftarrow src_1$

dest $\leftarrow M[x]$

src can be:
- constant
- ~~temporary~~
- special register
- memory

Abstract Assembly Language

program := i_1 i_2 … i_n seq of instructions

- **intermediate** – constants of some type
- **temporary** – a compiler generated location which holds a value. After compilation it will be mapped to a register or a memory location
- **register** – generally a real register from the target architecture

$\}$ values

$\}$ locations

Abstract Assembly Language

program := i_1 i_2 … i_n seq of instructions

i := d $\boxed{?}$ s move

| d $\boxed{?}$ s_1 $\boxed{?}$ s_2 binop

| **return** s_1 return

s := c intermediate

| t temporary

| r register

d := t

| r

$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ values

$\left.\begin{array}{l}\\ \end{array}\right\}$ locations

$\boxed{?}$:= **+ | - | * | / | %**

What is right "level"?

Closer to the machine

program := i_1 i_2 … i_n seq of instructions

i := d $\boxed{?}$ s move

| d $\boxed{?}$ s_1 $\boxed{?}$ s_2 binop

| **return** return what is in **rax**

s := c intermediate

| t temporary

| r register

d := t

| r

$\boxed{?}$:= **+ | - |* | / | %**

Deep Breath

- Defined source language using BNF
  - Ambiguity
  - Semantics
- AST
- Abstract assembly
  - Operators
  - L-values and R-values
  - Temps, registers, constants

Register Allocation

- Until register allocation we assume an unlimited 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

Locations

- Physical registers
- Slots in the activation frame

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.

Register Allocation

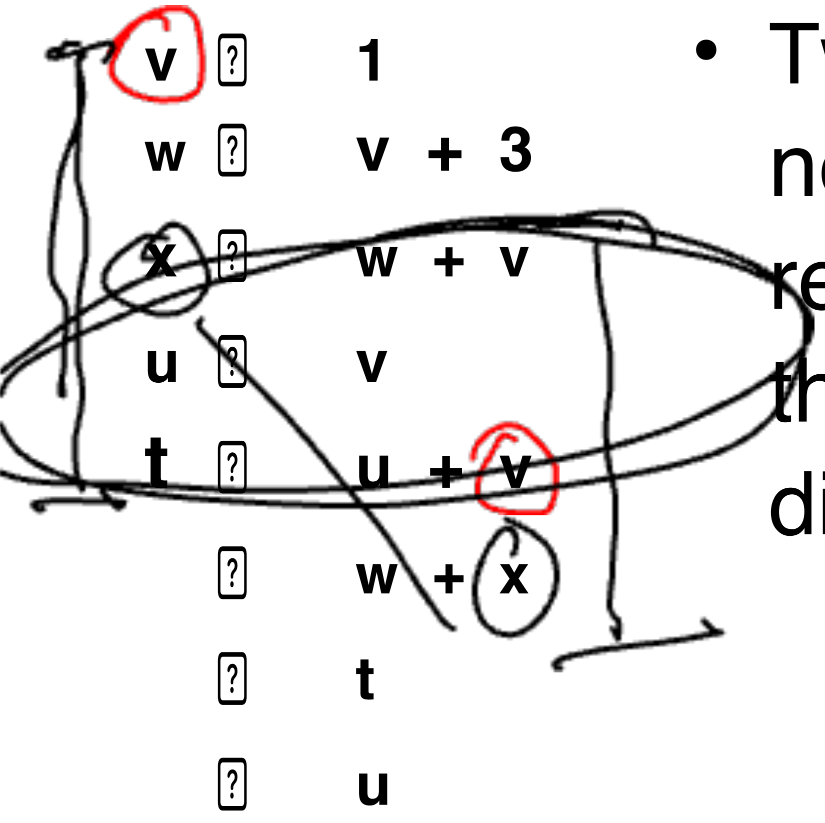
- Given: k registers and code with n registers
  Goal: transform code to use only k registers
- For every instruction we will:
  - Determine which values are in registers
  - Select a register for each value
- Global register allocation is of course NP-complete
- Develop heuristics which minimize running time

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.

What (if any) program points require x & v to be in different registers? (E.g., where do they “interfere”?)

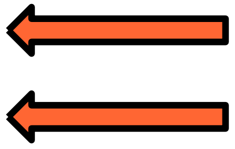
Running Example

```
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 is defined and can be used at some later point in the program.

Liveness in straight line code

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

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

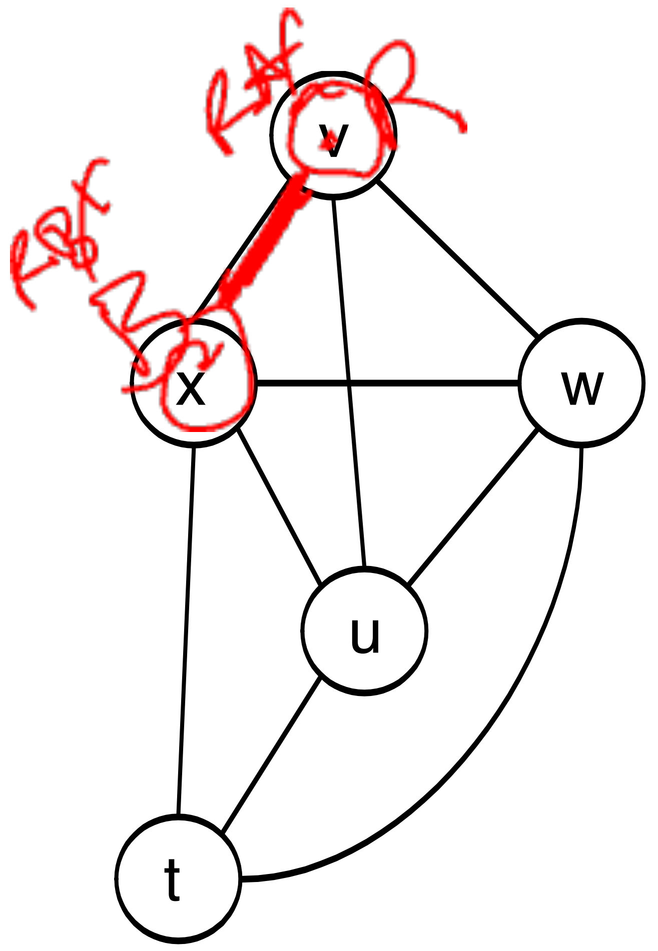
- Two variables that are live at the same point in the program interfere with each other and need to be assigned to different registers.

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

Interference Graph

- Nodes are temps and registers
- Edge (*a*,*b*) indicates *a* and *b* "interfere"
  In other words, *a* and *b* cannot be in the same register.

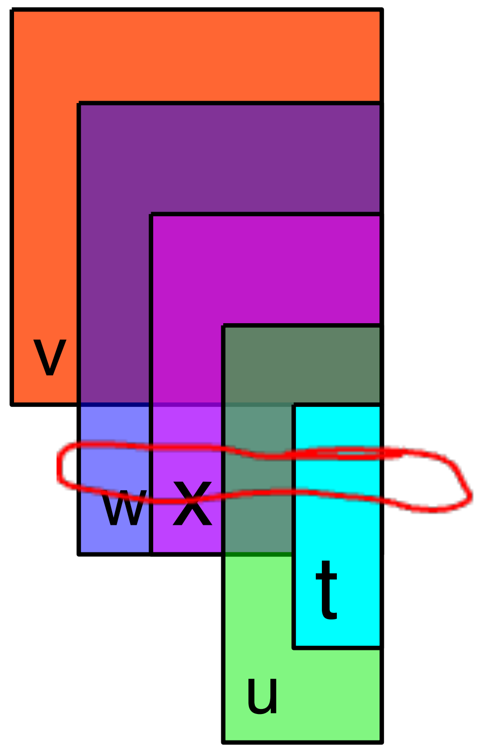


Optimistic Graph Coloring

- Construct Interference Graph
  - Use liveness information
  - Each node in the interference graph is a temp
  - (u,v) ⍰ 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, v }

{ w, t, u, x }

{ u, t }

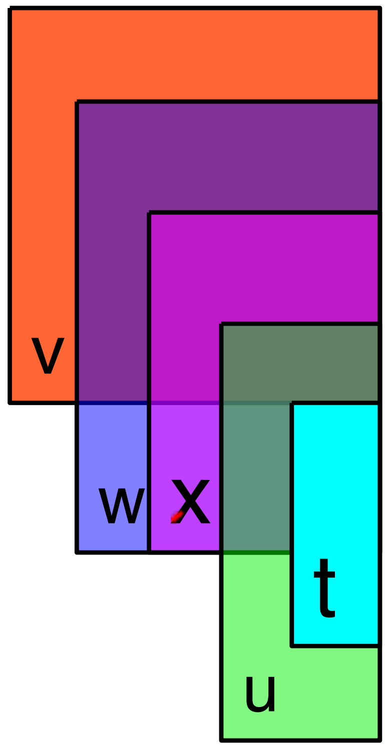
{ u }

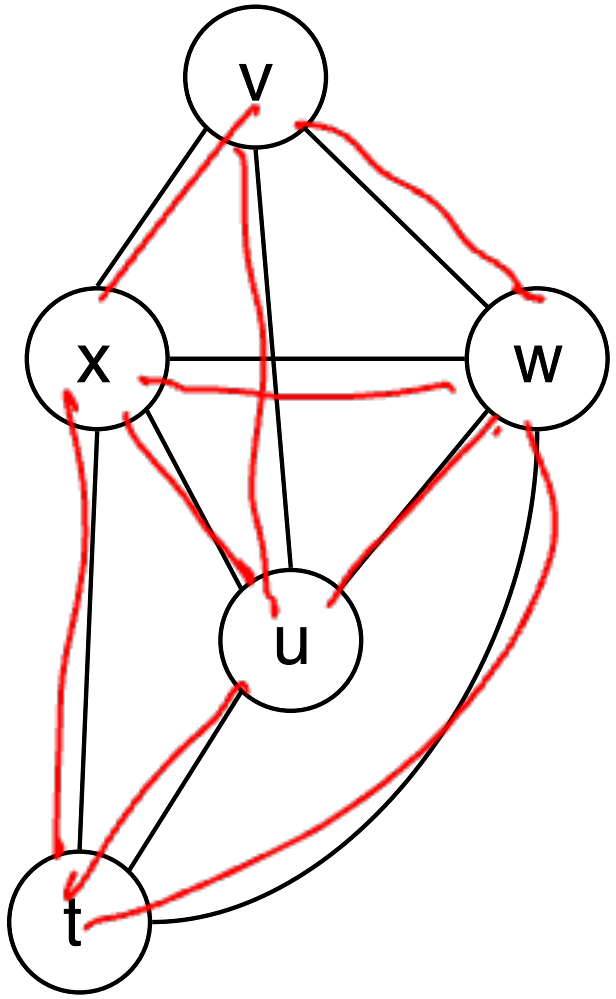
{ }

Compute live ranges

An Example, k=4

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

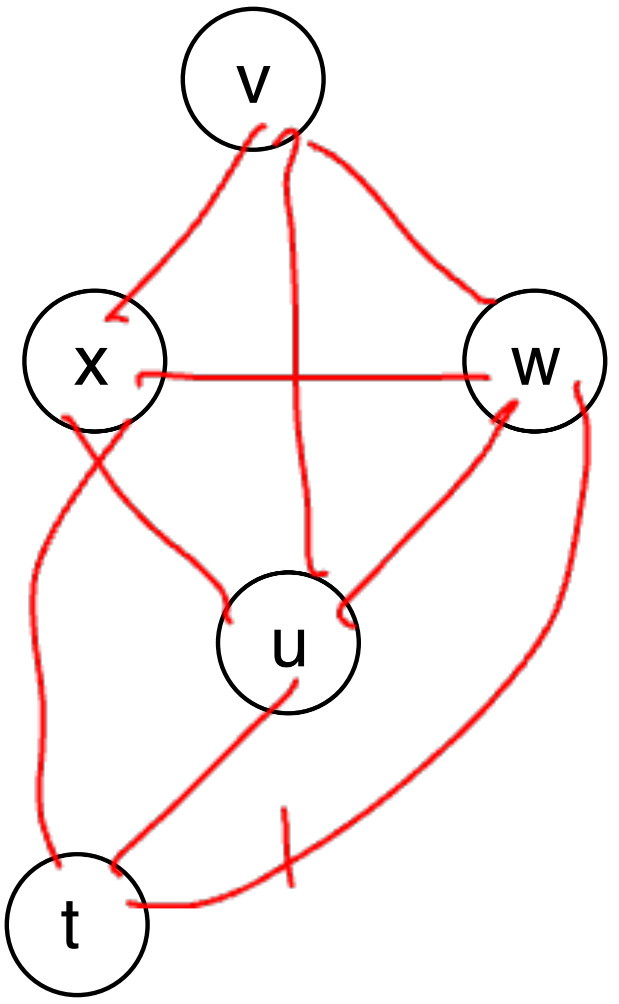




Construct the interference graph

In Practice

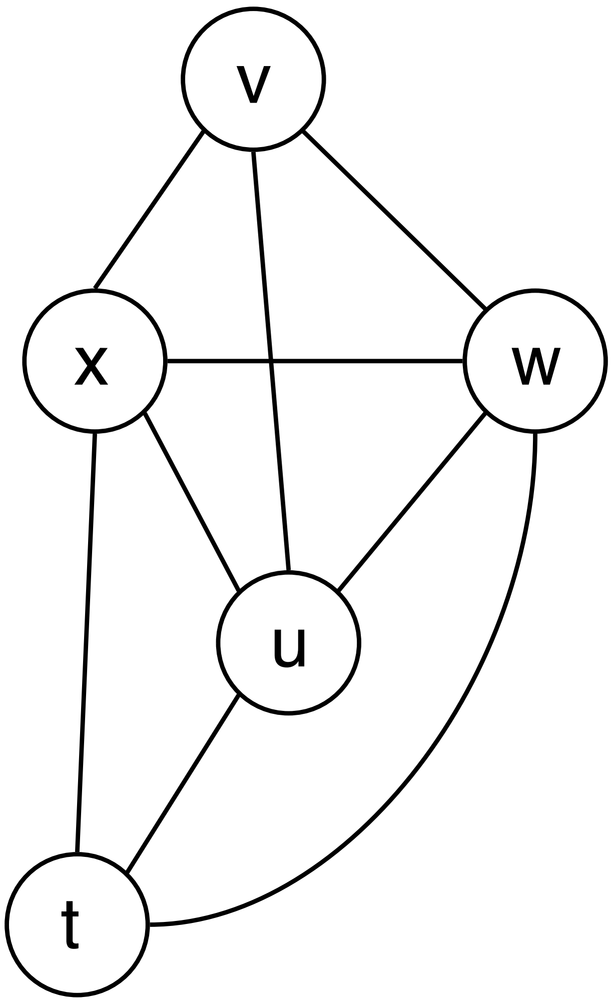
v ←	1	{ v }
w ←	v + 3	{ w, v }
x ←	w + v	{ w, x, v }
u ←	v	{ w, u, x, v }
t ←	u + v	{ w, t, u, x }
←	w + x	{ u, t }
←	t	{ u }
←	u	{ }



- 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, v }
t ⍰    u + v      { w, t, u, x }
  ⍰    w + x      { u, t }
  ⍰    t          { u }
  ⍰    u          { }
```

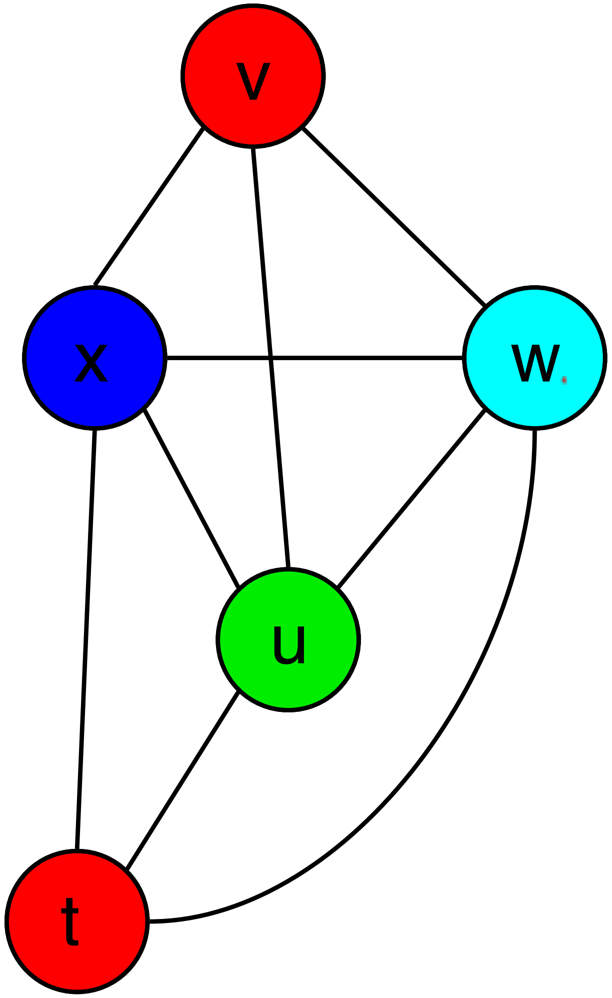


- At point of definition of t, add edges between t and all u ⍰ live-out, t⍰u

An Example, k=4

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

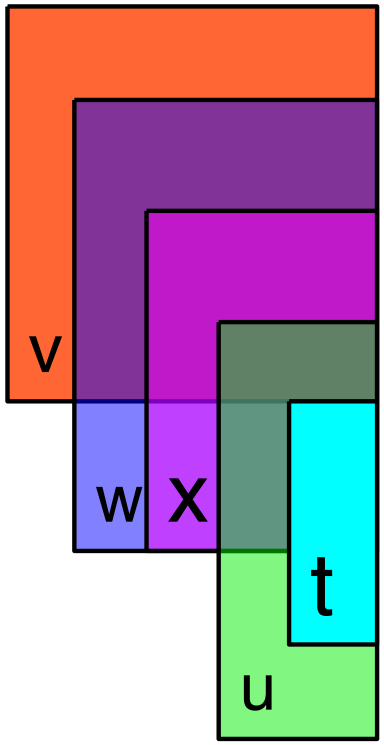
Voila, registers are assigned!

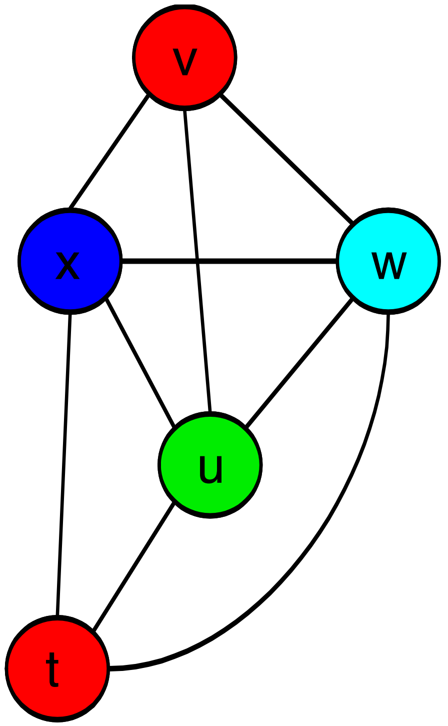


A greedy Coloring

A Special Interference Edge

```
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          { }
```





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

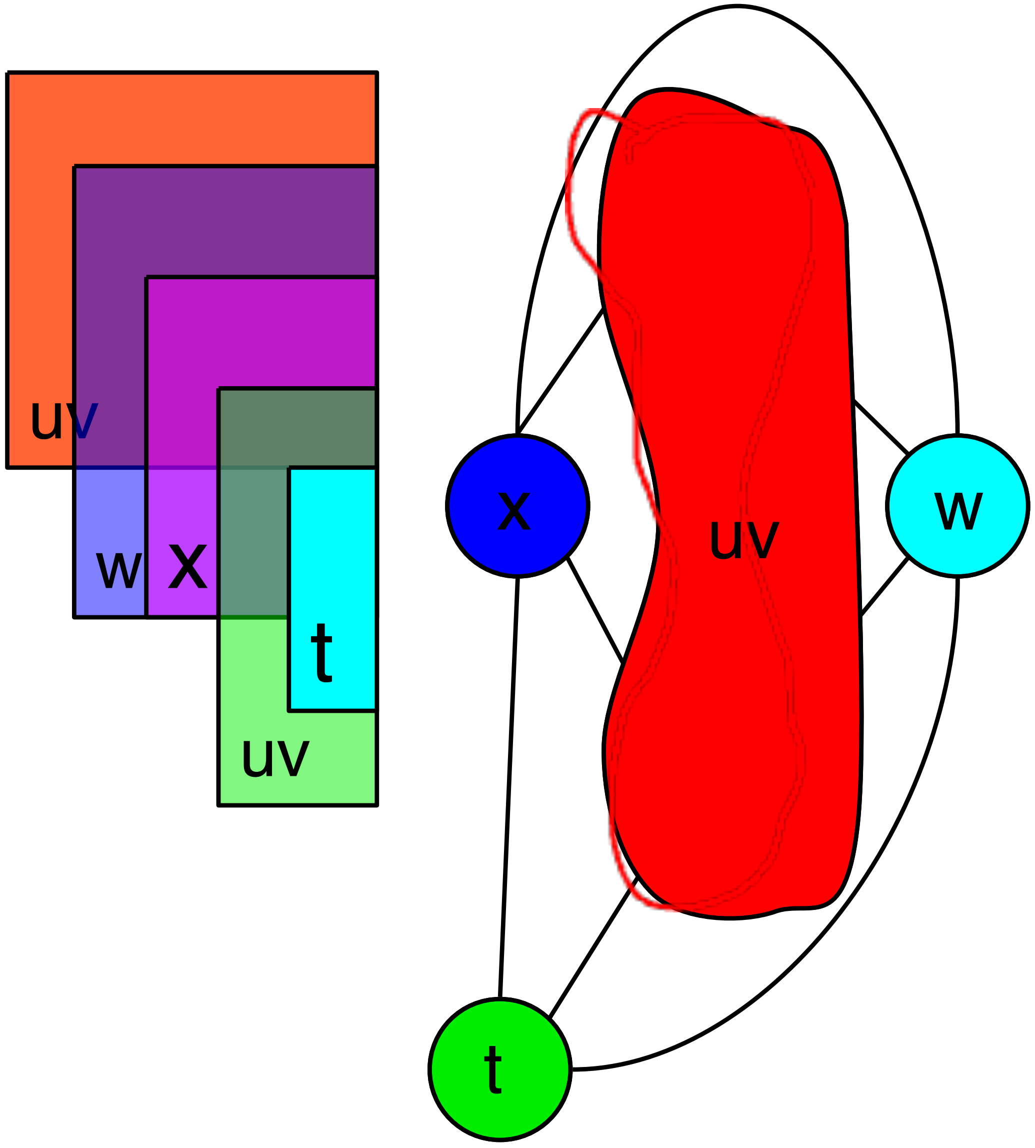
Interference and Coalescing

```
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

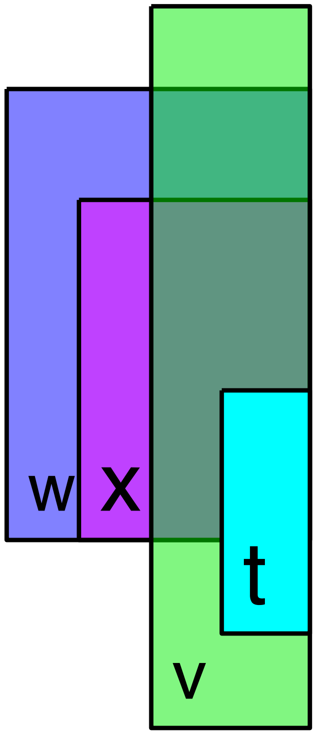
v ? 1
w ? v + 3
x ? w + v
~~u ? v~~
t ? u + v
? w + x
? t
? u

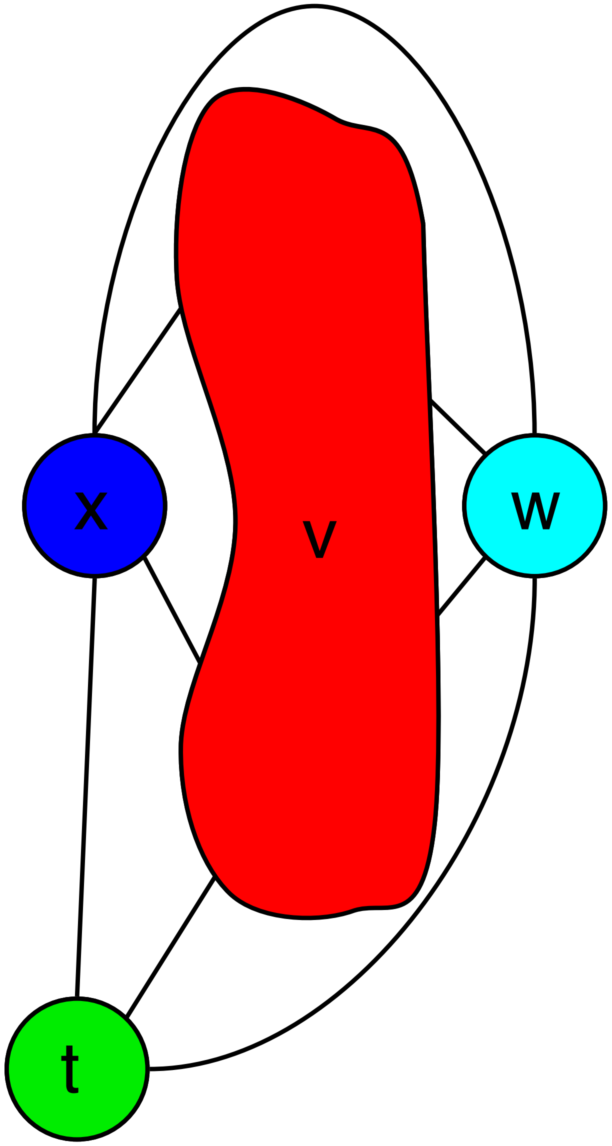


Rewrite the code to coalesce u & v

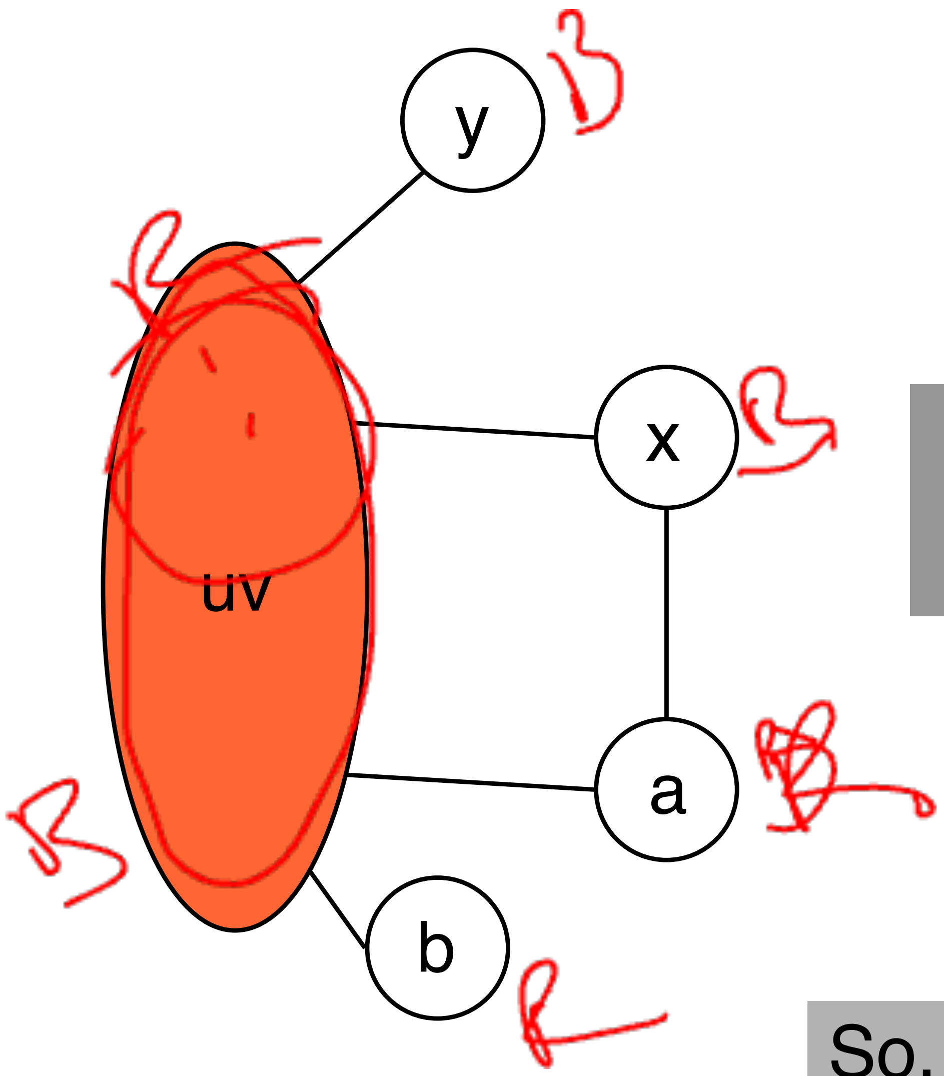
Another way to think about it

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





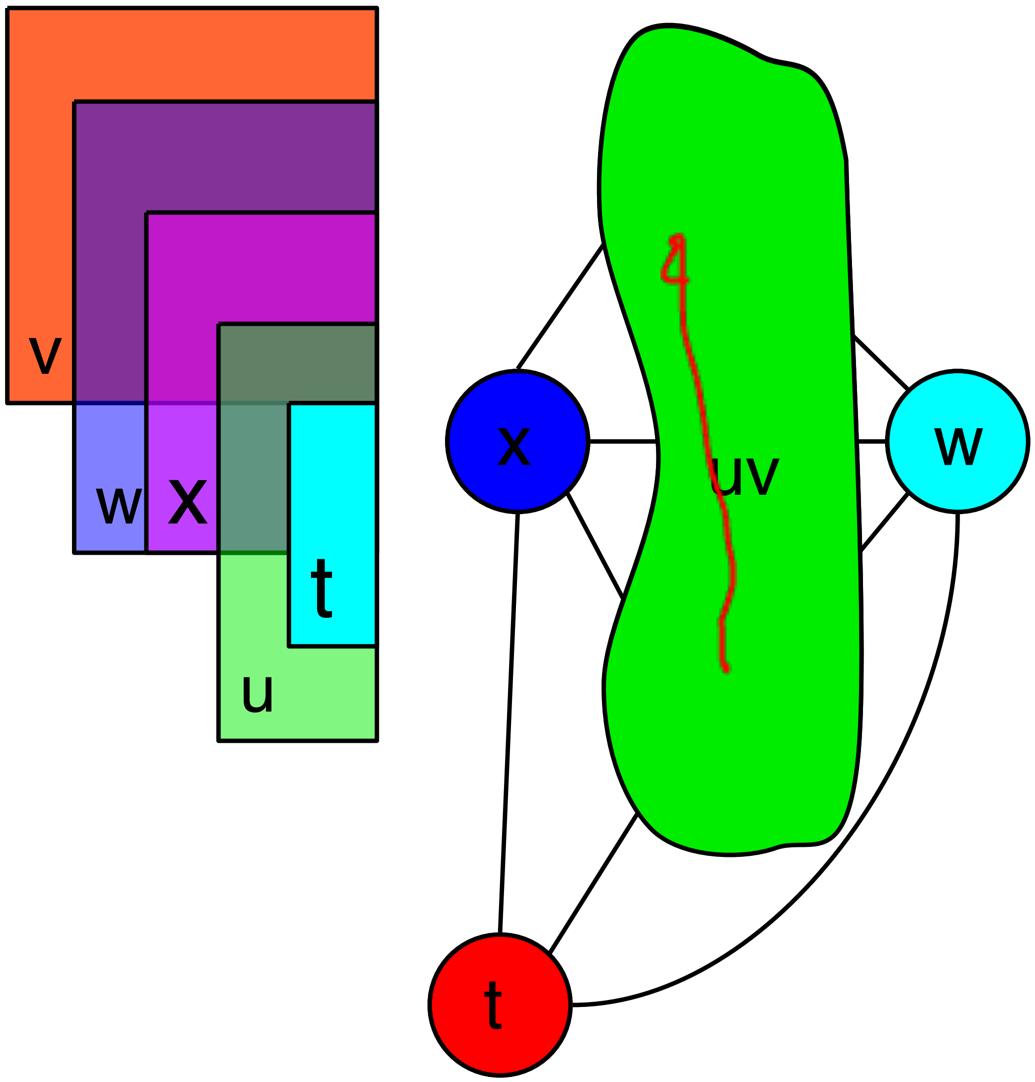
Is Coalescing always good?



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 $\boxed{?}$ **1**

w $\boxed{?}$ **v + 3**

x $\boxed{?}$ **w + v**

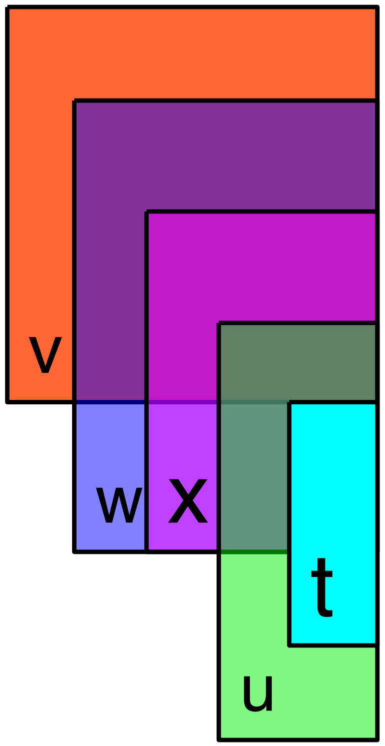
u $\boxed{?}$ **v**

t $\boxed{?}$ **u + v**

$\boxed{?}$ **w + x**

$\boxed{?}$ **t**

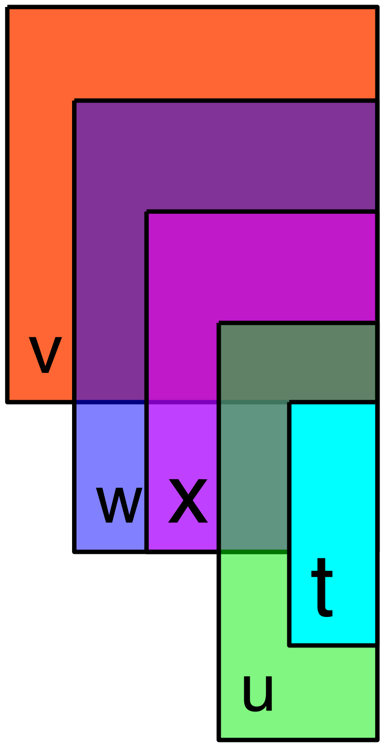
$\boxed{?}$ **u**

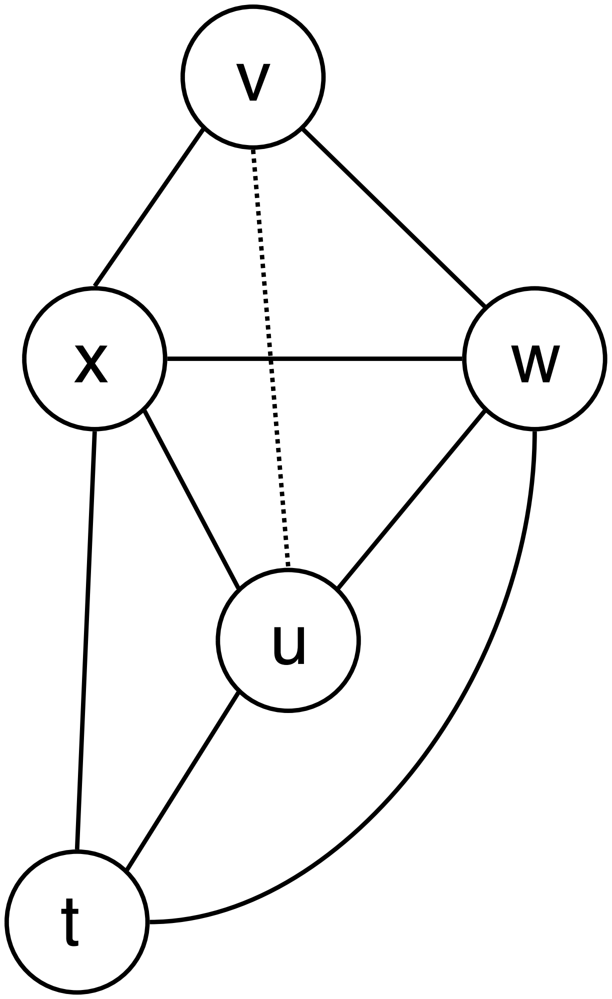


Compute live ranges

An Example, k=3

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

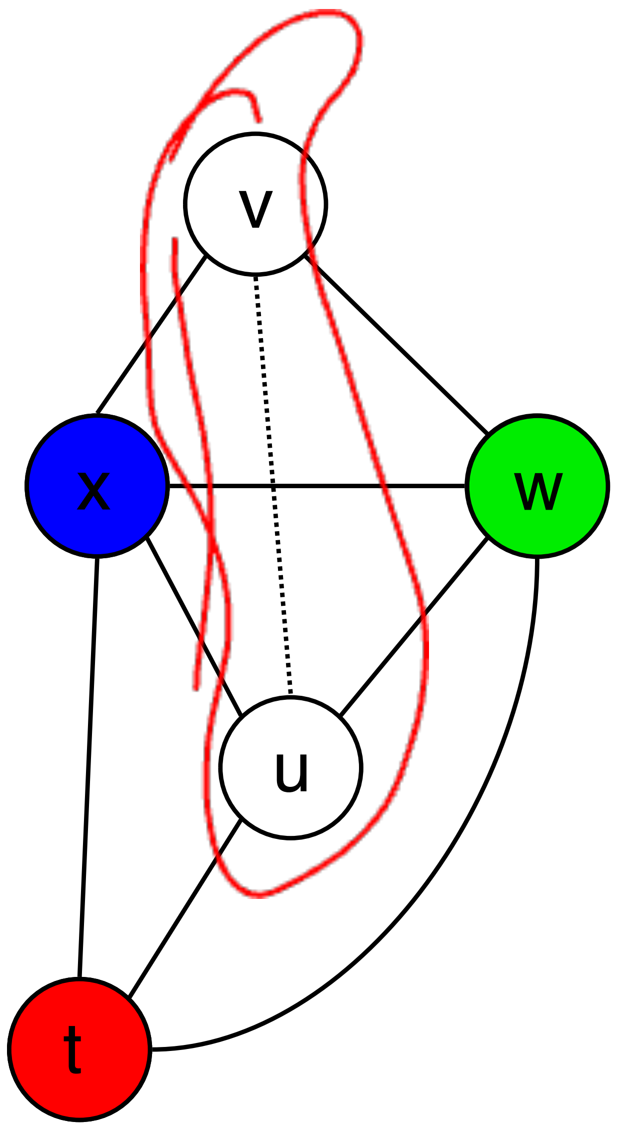




Construct the interference graph

An Example, k=3

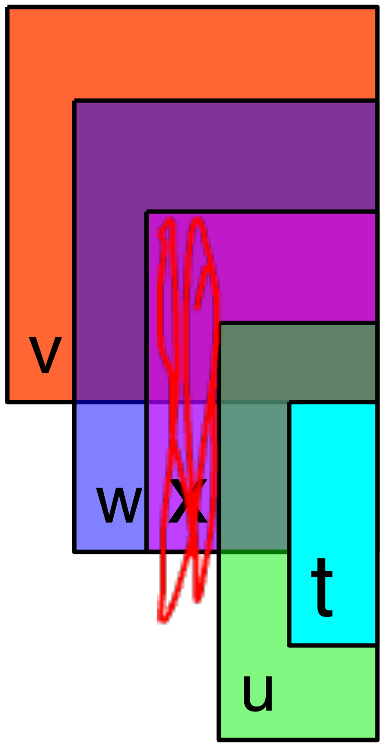
```
v ⍰   1
w ⍰   v + 3
x ⍰   w + v
u ⍰   v
t ⍰   u + v
  ⍰   w + x
  ⍰   t
  ⍰   u
```

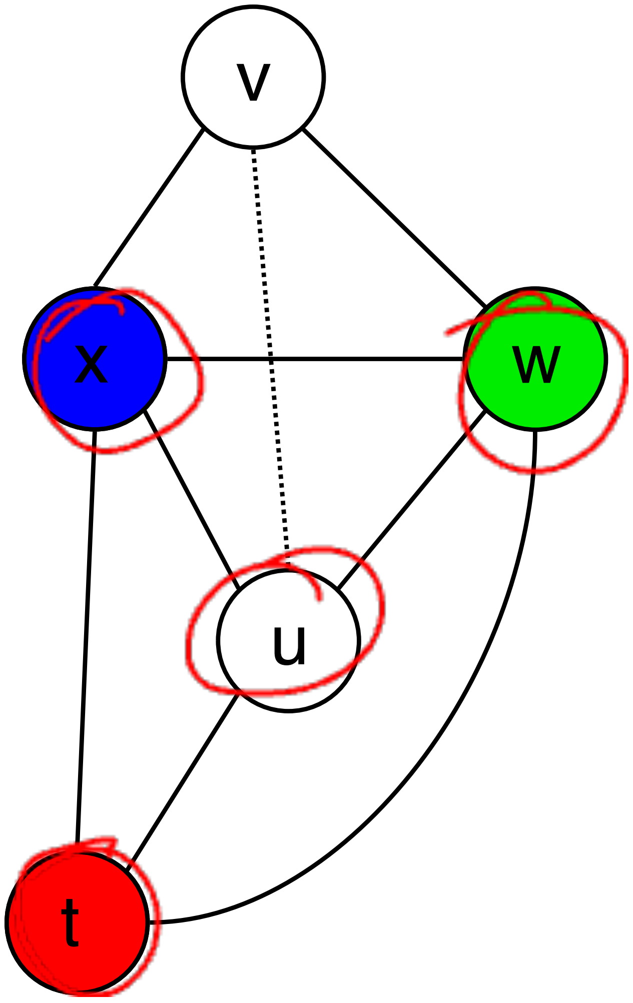


So, we need to spill

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 x and Rewrite program

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

An Example, k=3

recalculate live ranges

```
v   ⍰   1
w   ⍰   v + 3
x   ⍰   w + v
M[] ⍰   x
u   ⍰   v
t   ⍰   u + v
x’  ⍰   M[]
    ⍰   w + x’
    ⍰   t
    ⍰   u            { }
```

An Example, k=3

recalculate live ranges

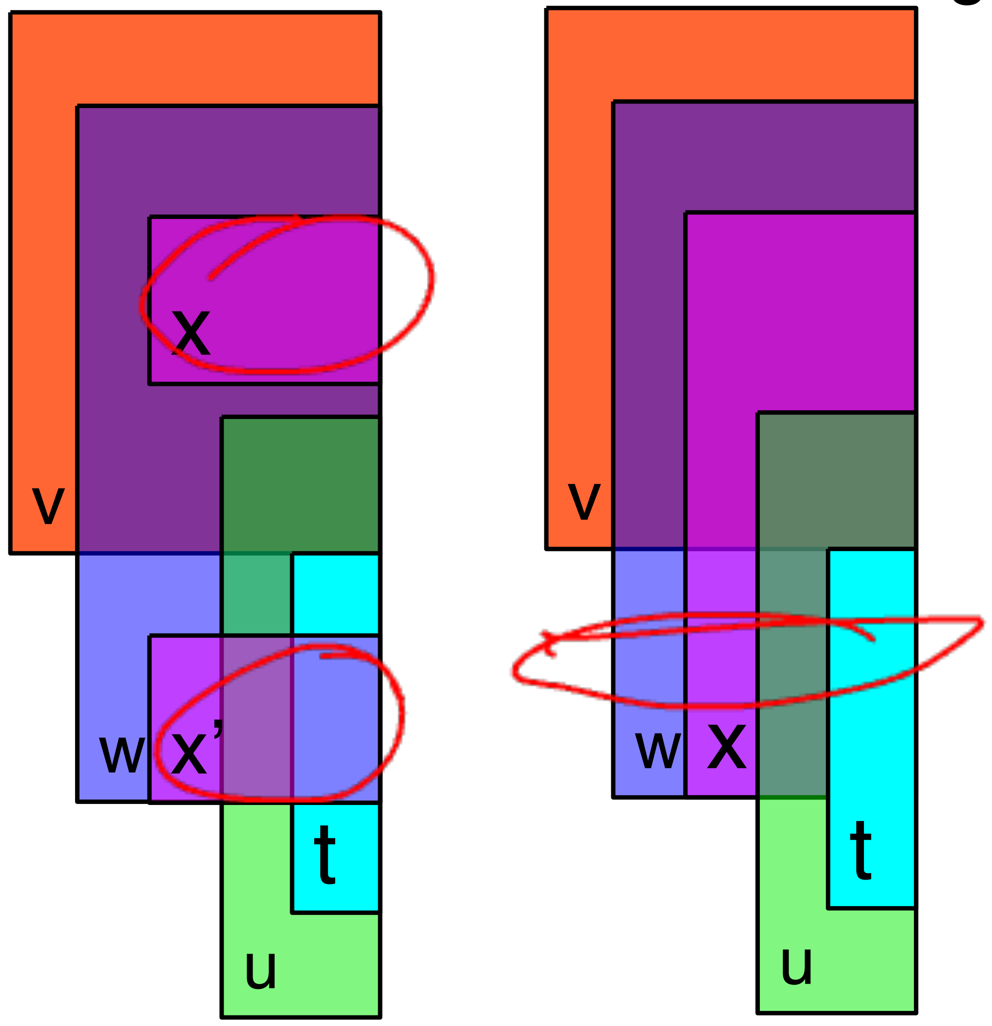
v $\boxed{?}$ 1
w $\boxed{?}$ v + 3
x $\boxed{?}$ w + v
M[] $\boxed{?}$ x
u $\boxed{?}$ v
t $\boxed{?}$ u + v
x' $\boxed{?}$ M[]
$\boxed{?}$ w + x'
$\boxed{?}$ t
$\boxed{?}$ u

{ v }
{ w, v }
{ w, v, x }
{ w, v }
{ w, u, v }
{ w, t, u }
{ w, t, u, x' }
{ u, t }
{ u }
{ }

An Example, k=3

```
v   ?   1
w   ?   v + 3
x   ?   w + v
M[] ?   x
u   ?   v
t   ?   u + v
x’  ?   M[]
    ?   w + x’
    ?   t
    ?   u
```

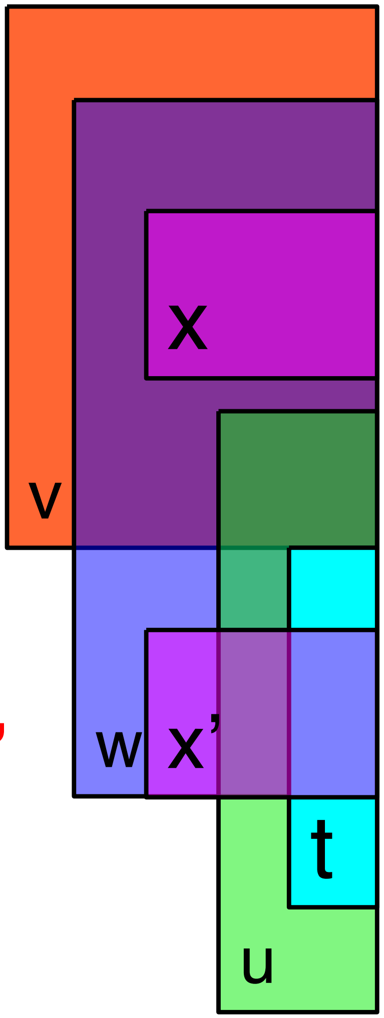
recalculate live ranges

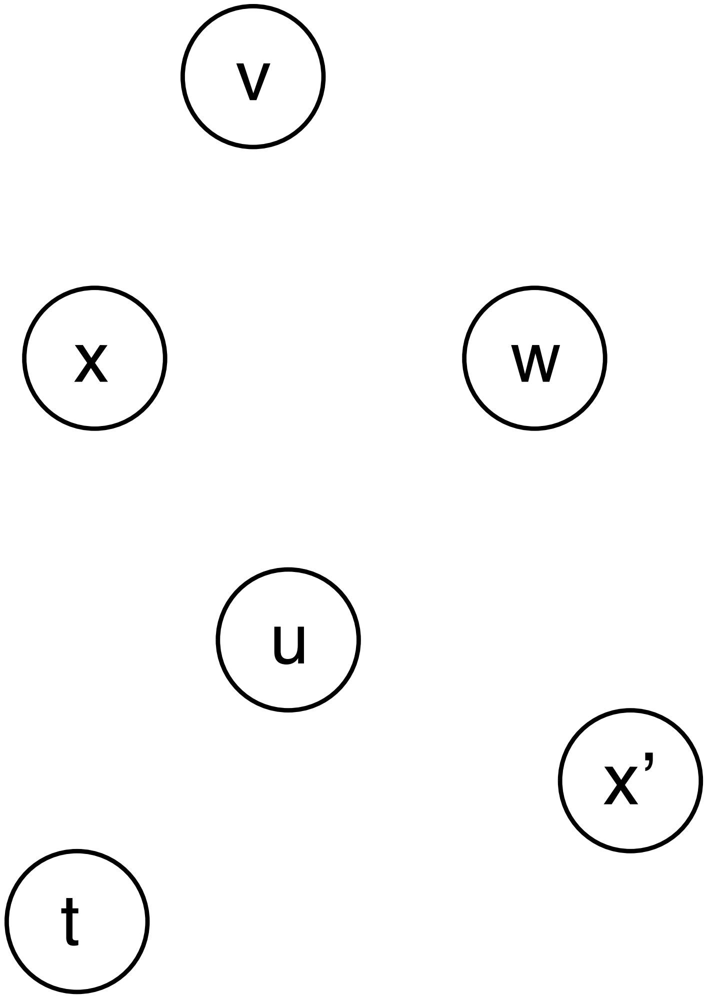


Spilling reduces live ranges, which decreases register pressure.

An Example, k=3

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

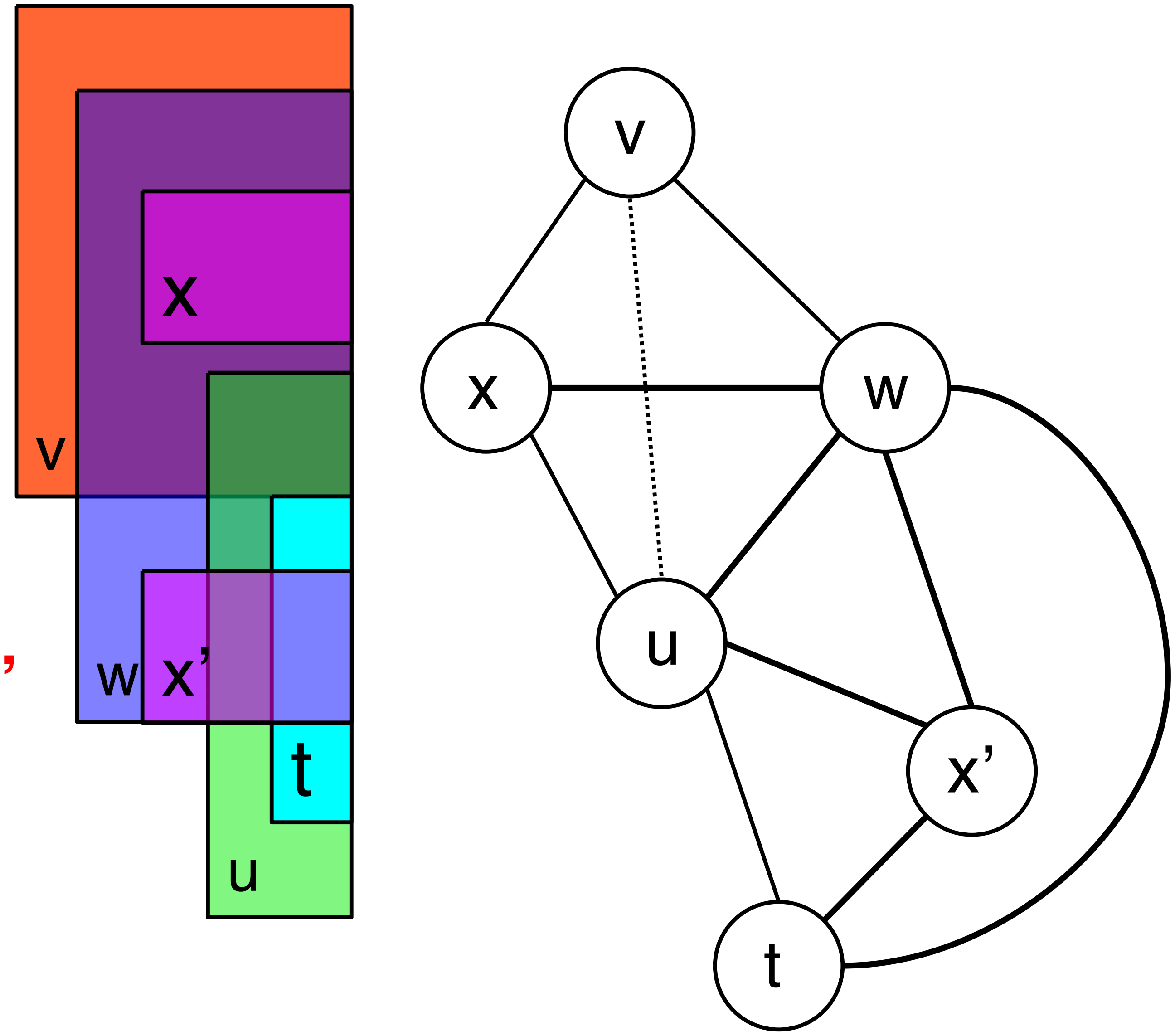




Recalculate interference graph

An Example, k=3

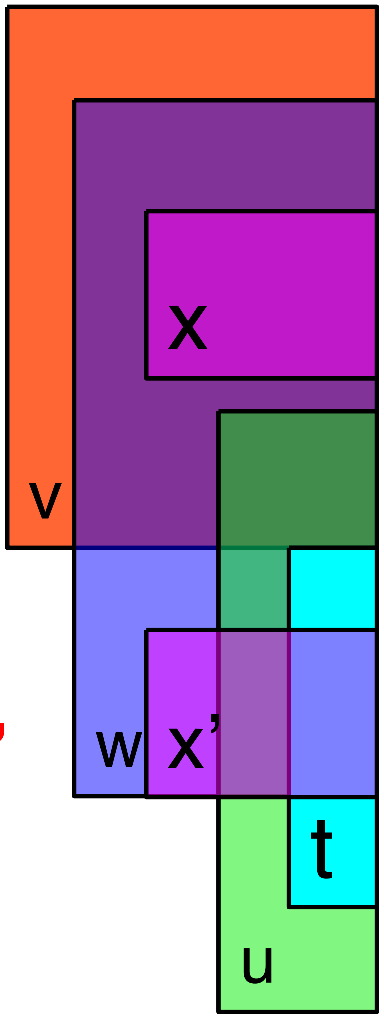
```
v   ?   1
w   ?   v + 3
x   ?   w + v
M[] ?   x
u   ?   v
t   ?   u + v
x’  ?   M[]
    ?   w + x’
    ?   t
    ?   u
```

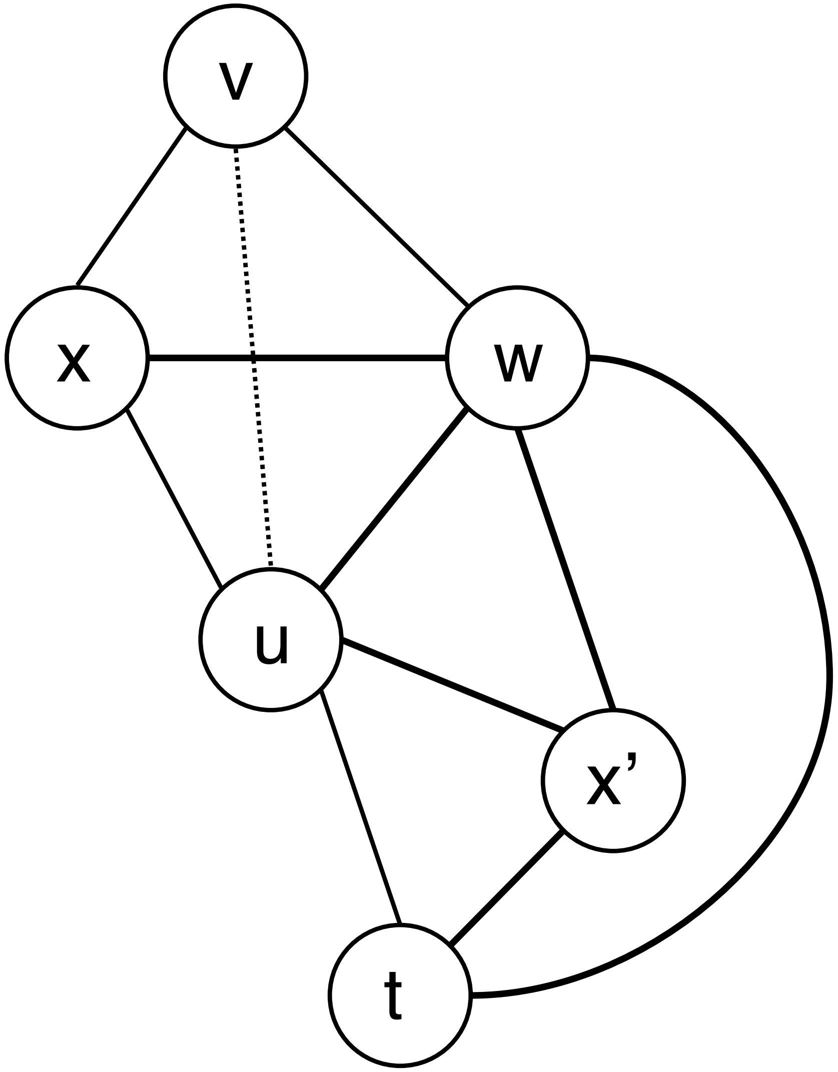


Recalculate interference graph

An Example, k=3

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

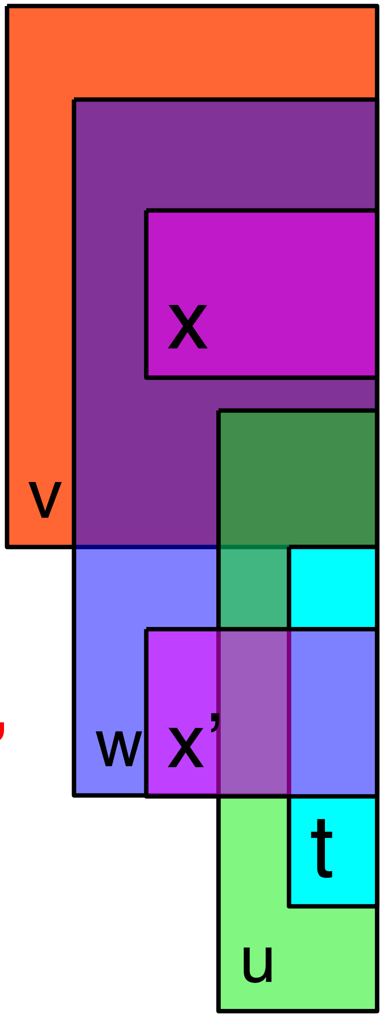


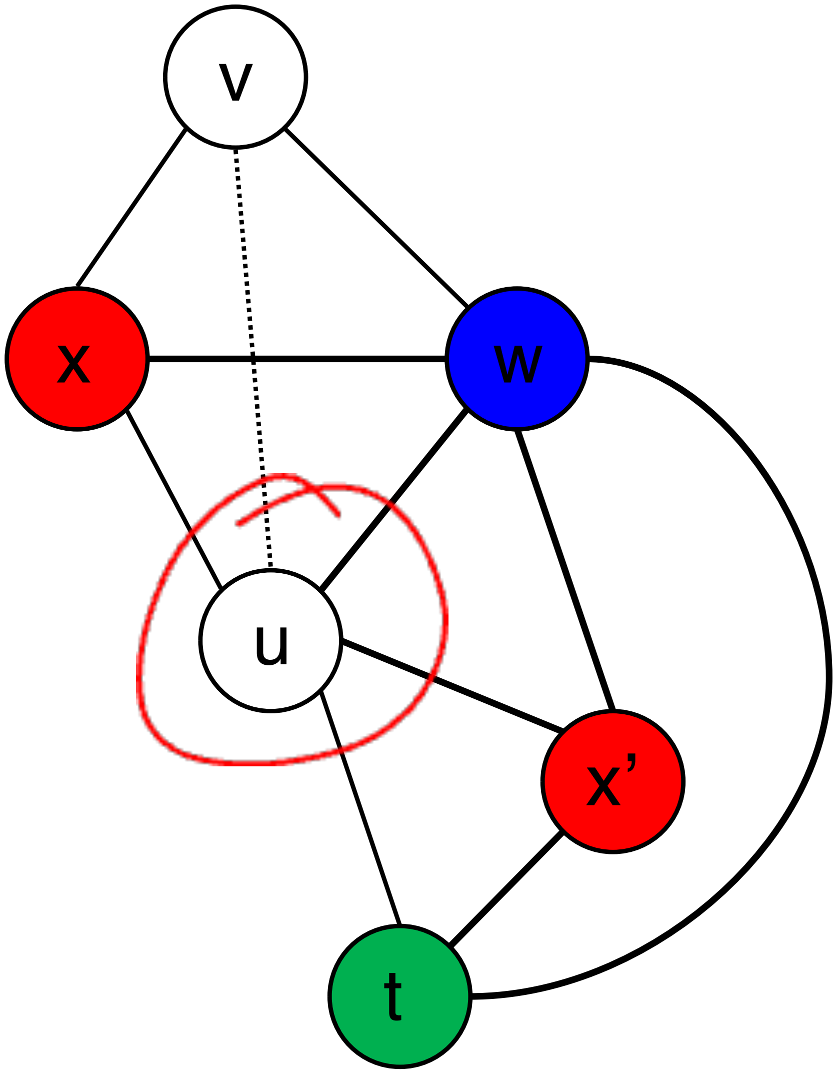


Recolor Graph

An Example, k=3

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





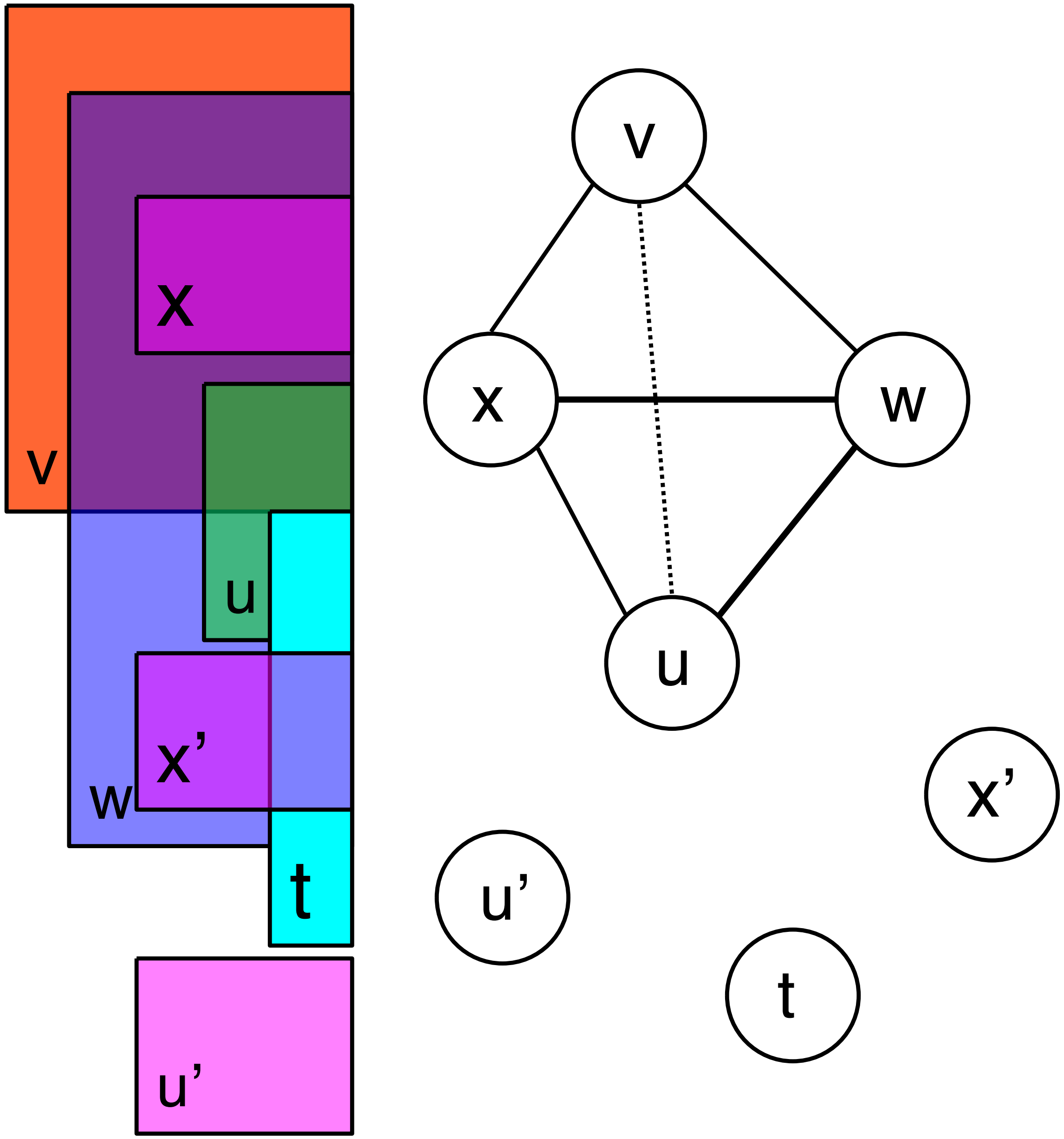
Sigh

An Example, k=3

```
v    ←   1
w    ←   v + 3
x    ←   w + v
M[0] ←   x
u    ←   v
t    ←   u + v
M[1] ←   u
x'   ←   M[0]
     ←   w + x'
     ←   t
u'   ←   M[1]
     ←   u                      respill
```

An Example, k=3

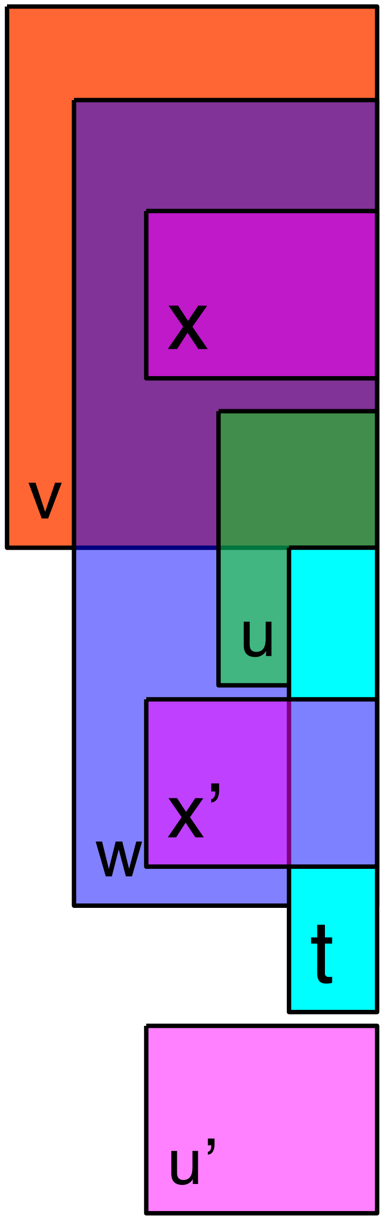
```
v    ?     1
w    ?     v + 3
x    ?     w + v
M[0] ?   x
u    ?     v
t    ?     u + v
M[1] ?   u
x'   ?     M[0]
     ?     w + x'
     ?     t
u'   ?     M[1]
     ?     u
```

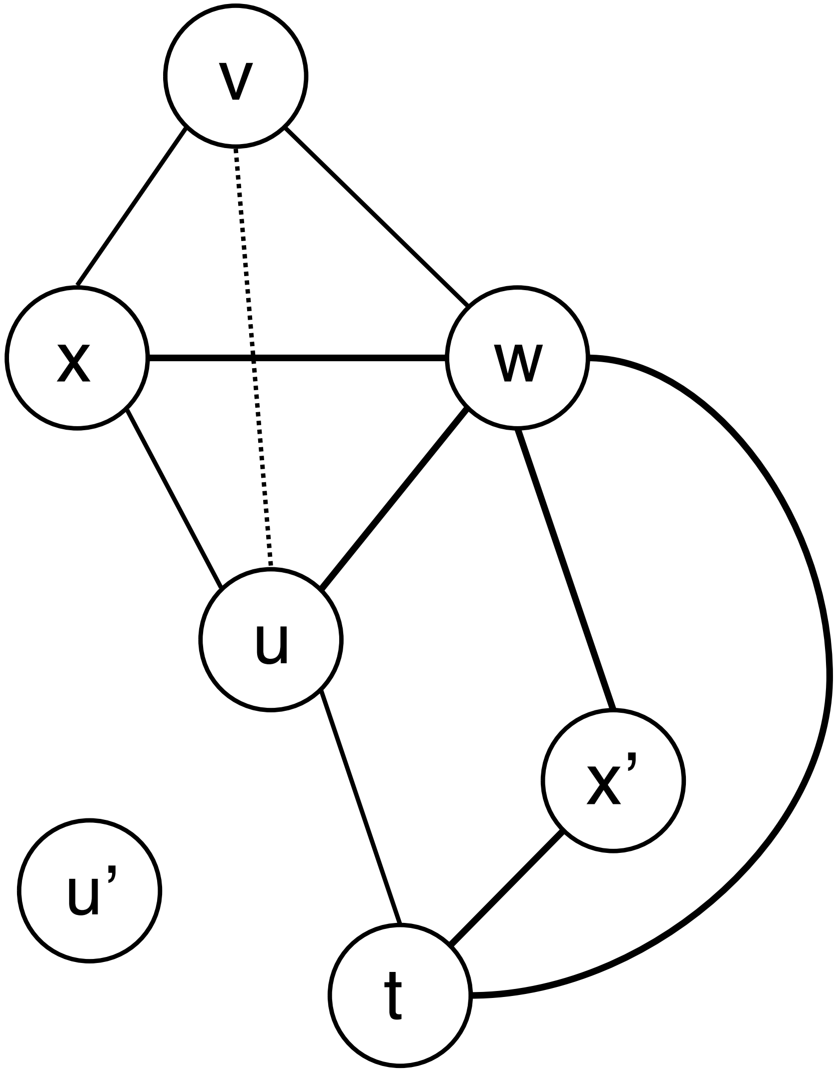


construct new interference graph

An Example, k=3

```
v    ⍰    1
w    ⍰    v + 3
x    ⍰    w + v
M[0] ⍰    x
u    ⍰    v
t    ⍰    u + v
M[1] ⍰    u
x'   ⍰    M[0]
     ⍰    w + x'
     ⍰    t
u'   ⍰    M[1]
     ⍰    u
```

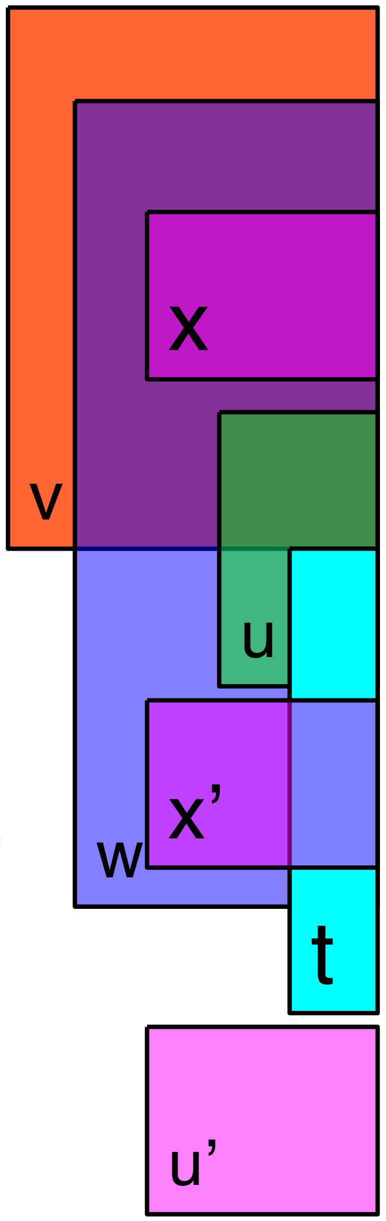


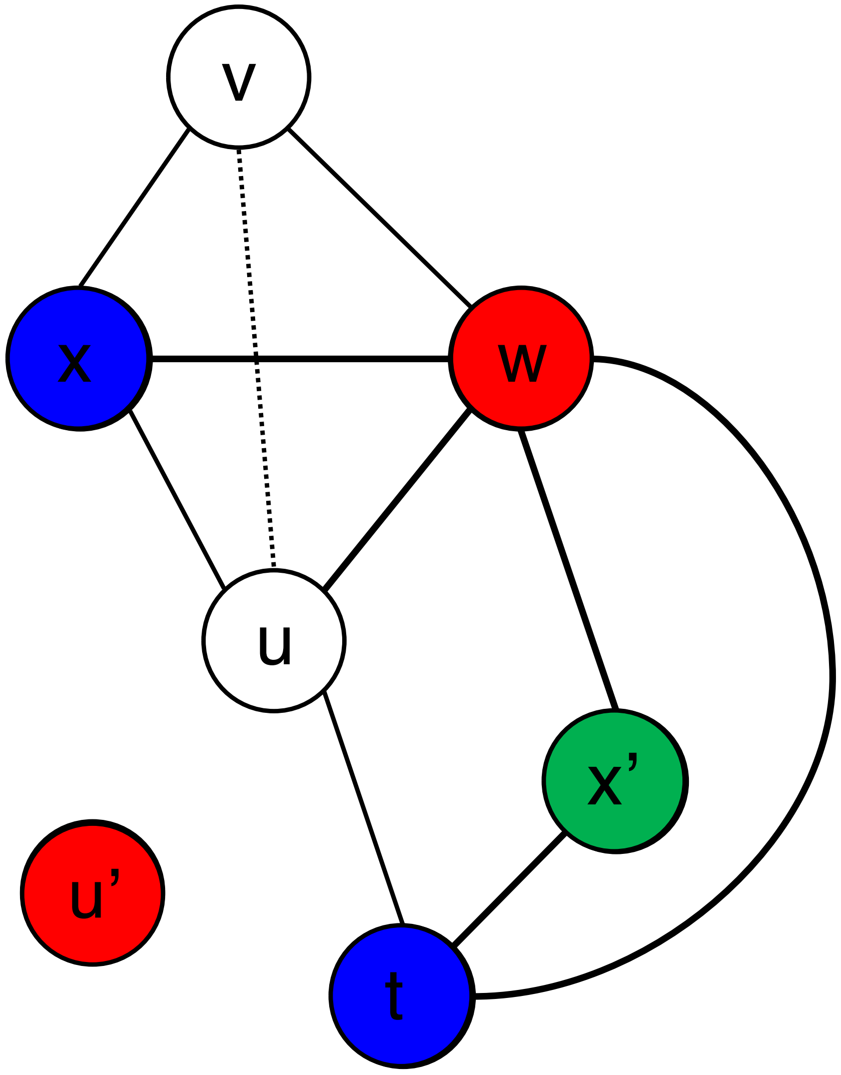


construct new interference graph

An Example, k=3

```
v    ⍰   1
w    ⍰   v + 3
x    ⍰   w + v
M[0] ⍰   x
u    ⍰   v
t    ⍰   u + v
M[1] ⍰   u
x'   ⍰   M[0]
     ⍰   w + x'
     ⍰   t
u'   ⍰   M[1]
     ⍰   u
```

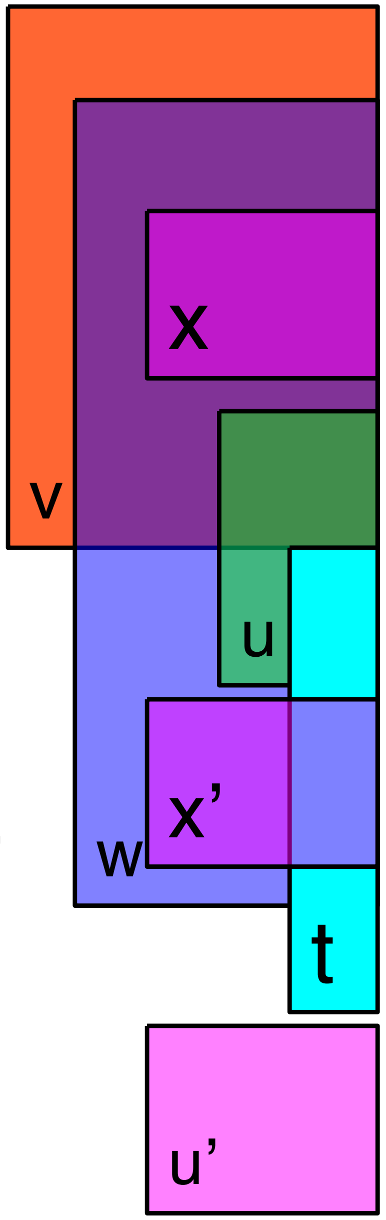


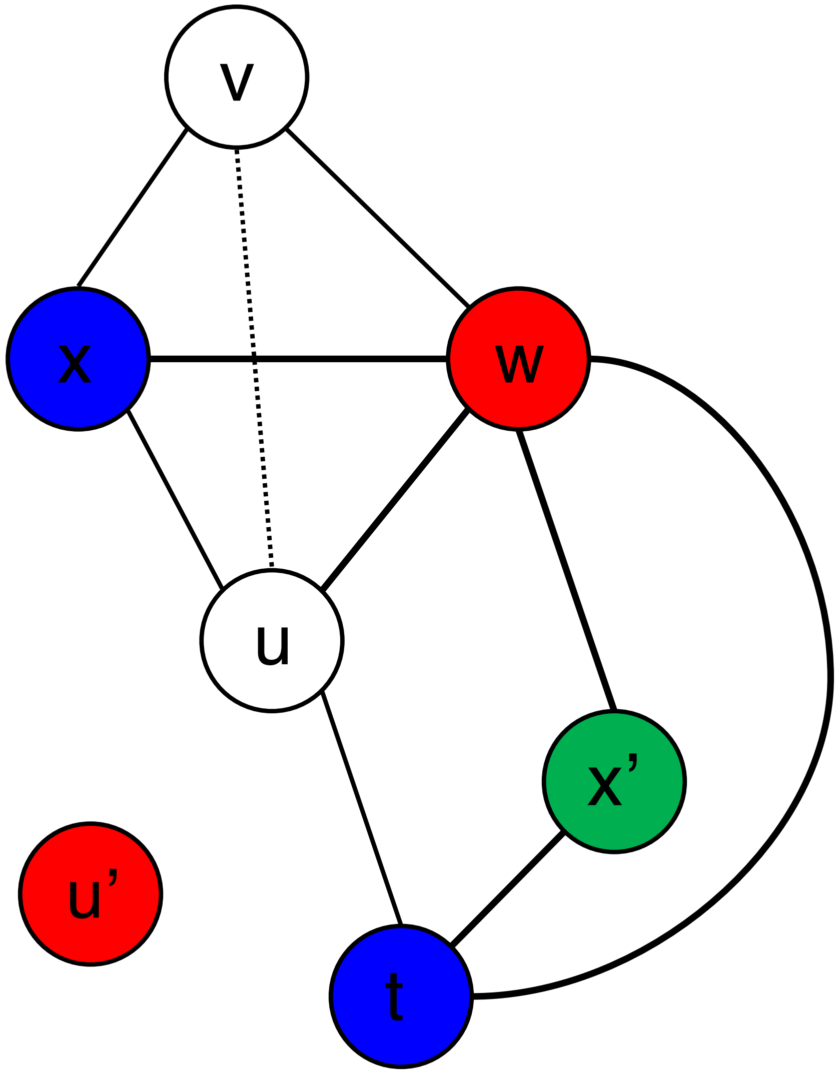


color graph

An Example, k=3

```
v    ⍰   1
w    ⍰   v + 3
x    ⍰   w + v
M[0] ⍰   x
u    ⍰   v
t    ⍰   u + v
M[1] ⍰   u
x’   ⍰   M[0]
     ⍰   w + x’
     ⍰   t
u’   ⍰   M[1]
     ⍰   u
```

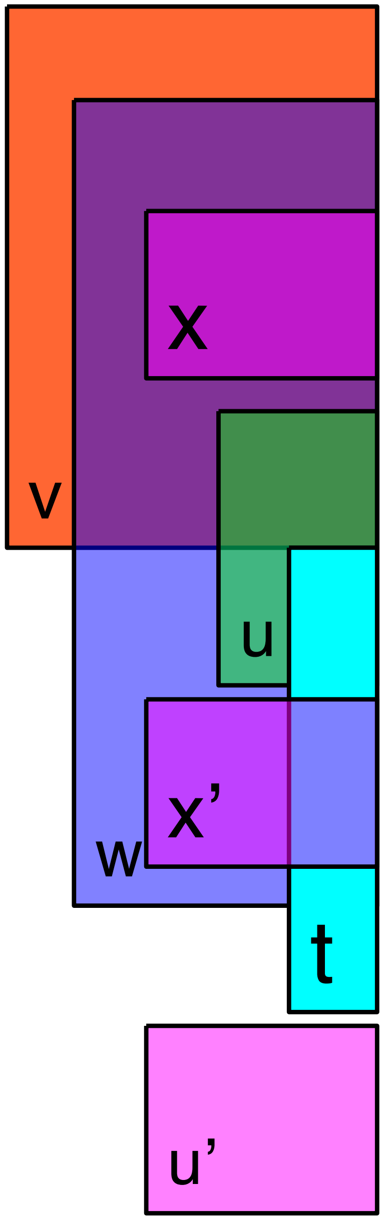


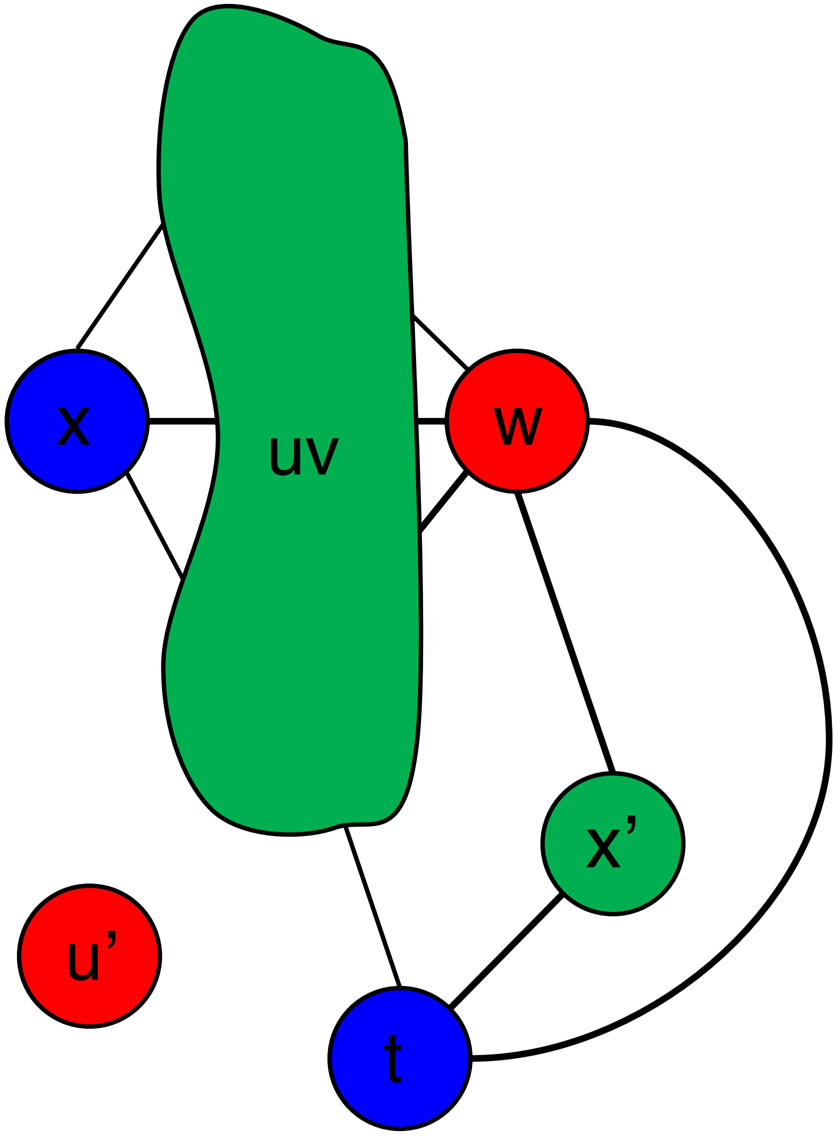


color graph

An Example, k=3

```
v    ⍰   1
w    ⍰   v + 3
x    ⍰   w + v
M[0] ⍰   x
u    ⍰   v
t    ⍰   u + v
M[1] ⍰   u
x’   ⍰   M[0]
     ⍰   w + x’
     ⍰   t
u’   ⍰   M[1]
     ⍰   u
```





color graph

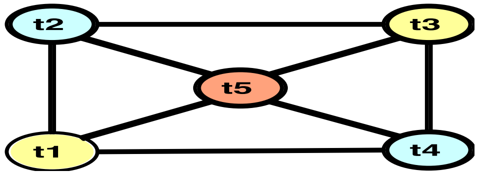
Graph coloring

- 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

Doesn't always help...

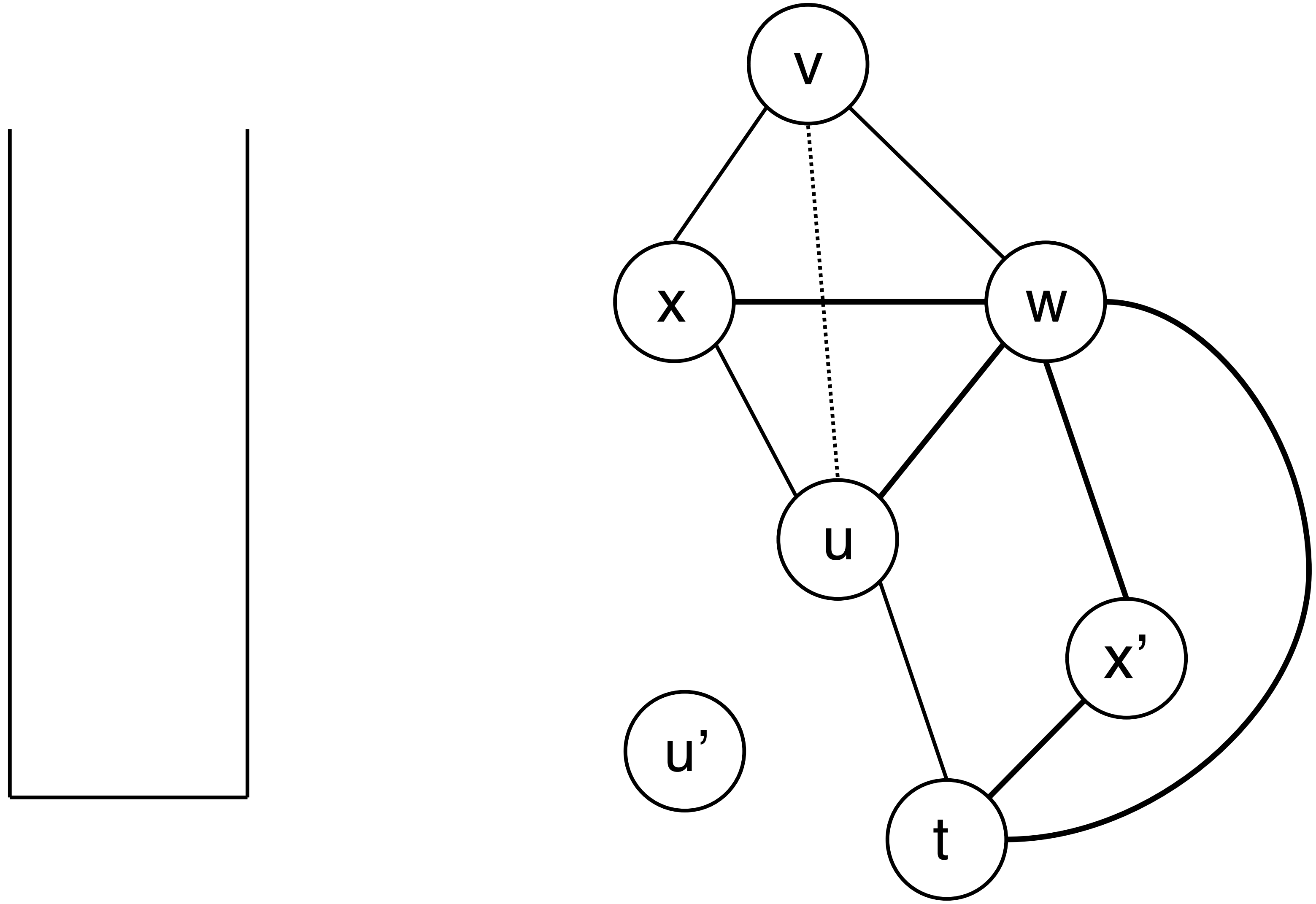


This graph is 3-colorable, but has no nodes with degree < 3

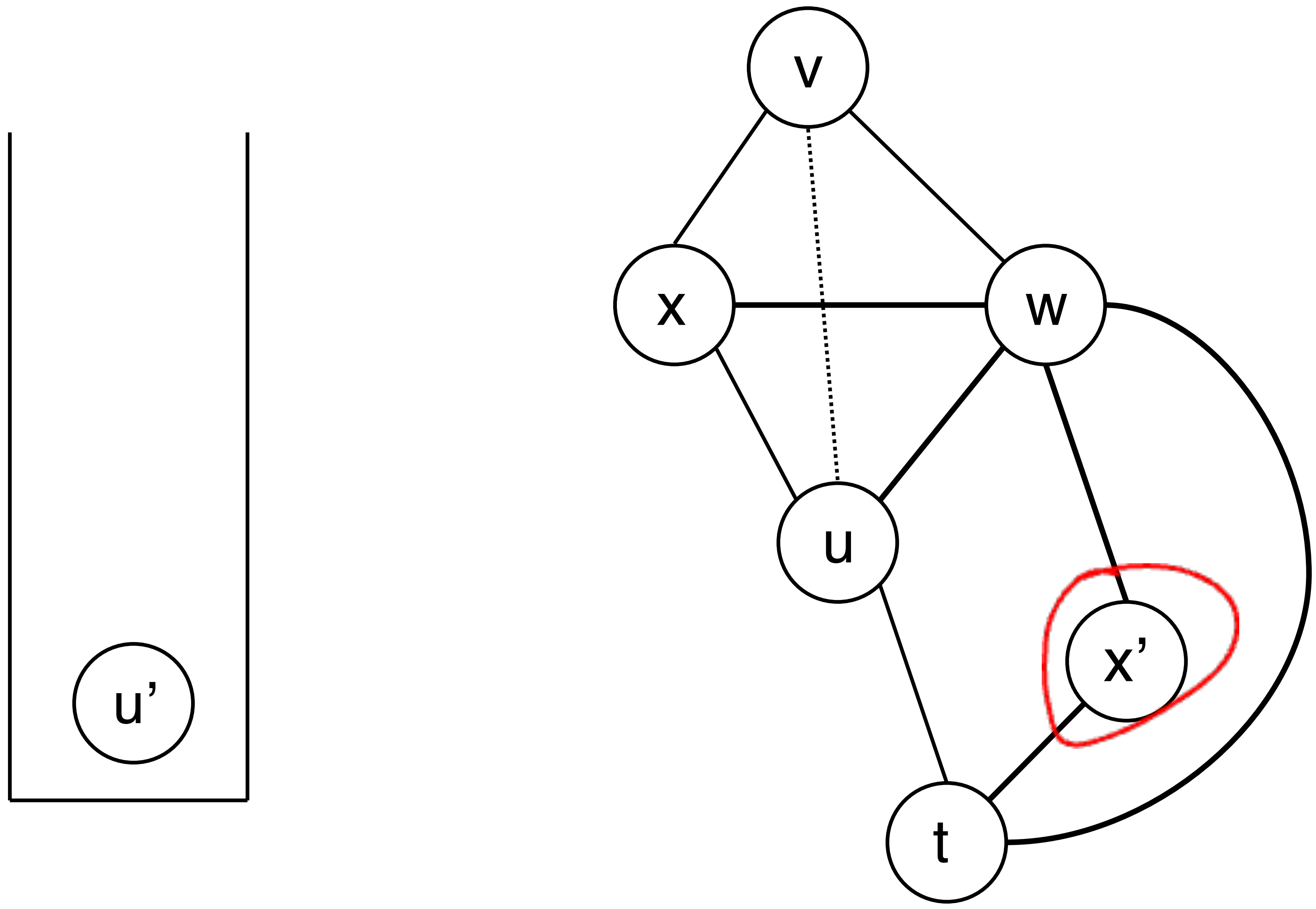
Kempe's algorithm

- 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

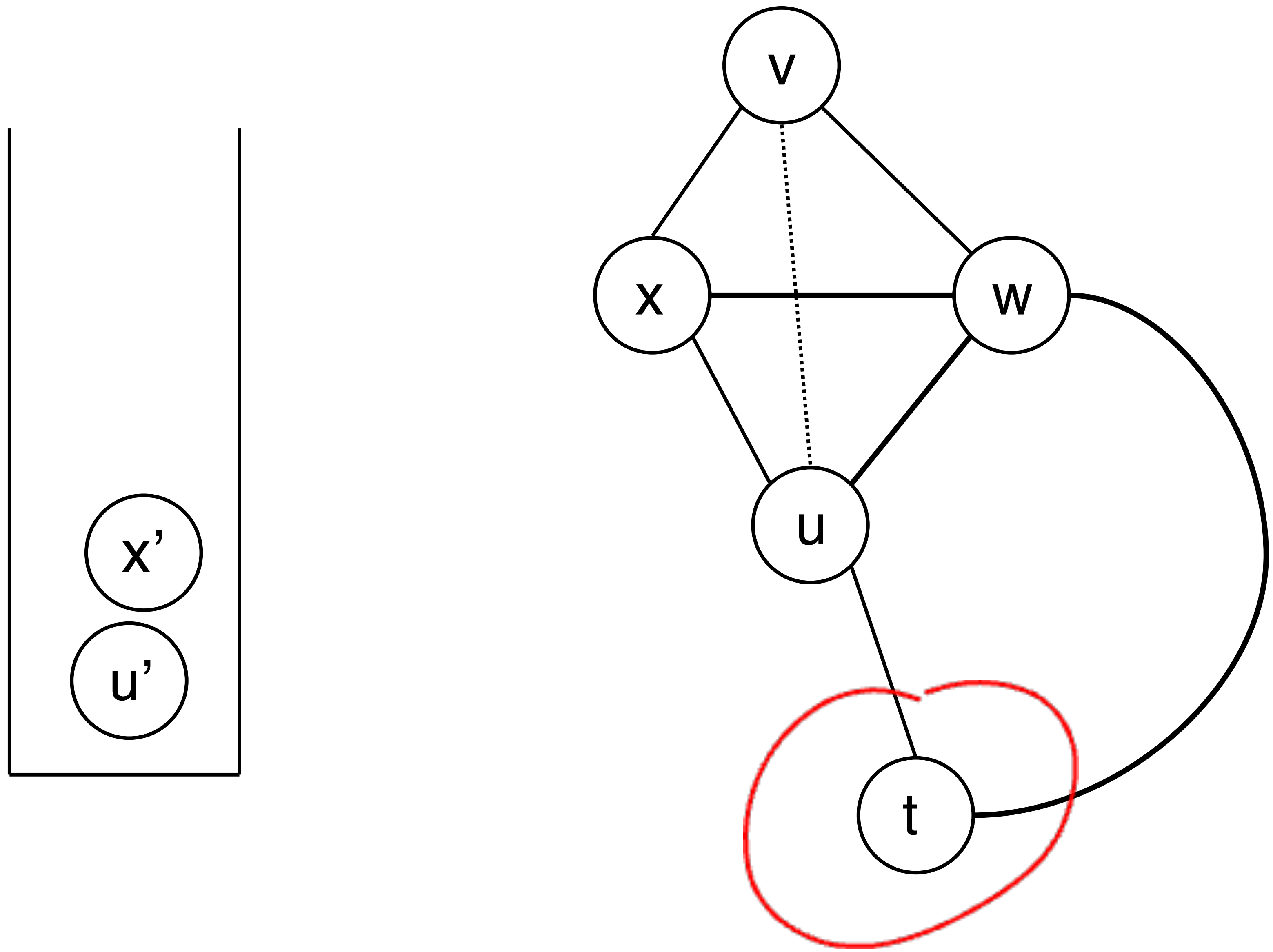
Example, k=3



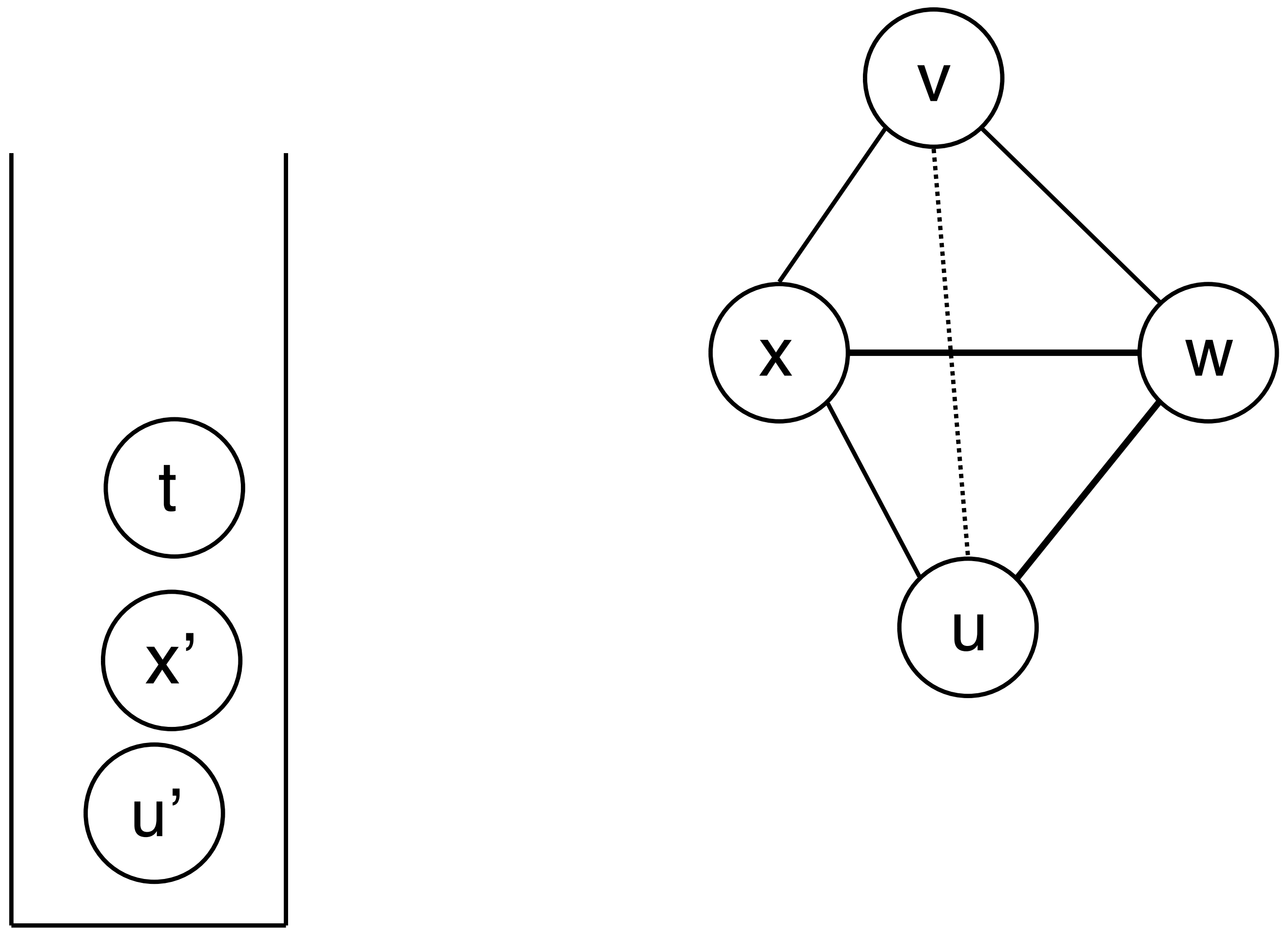
Example, k=3



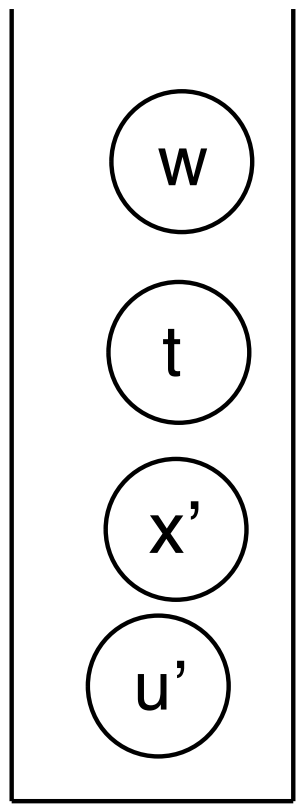
Example, k=3

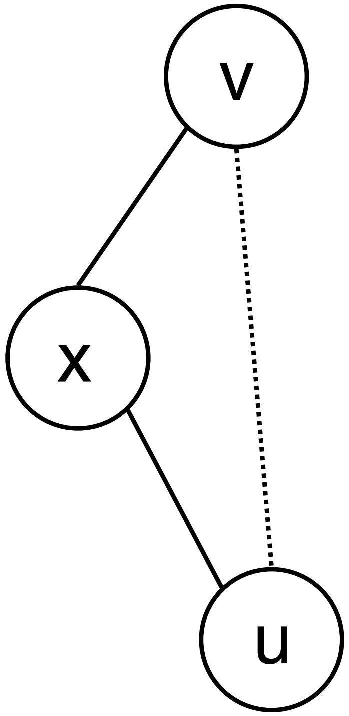


Example, k=3

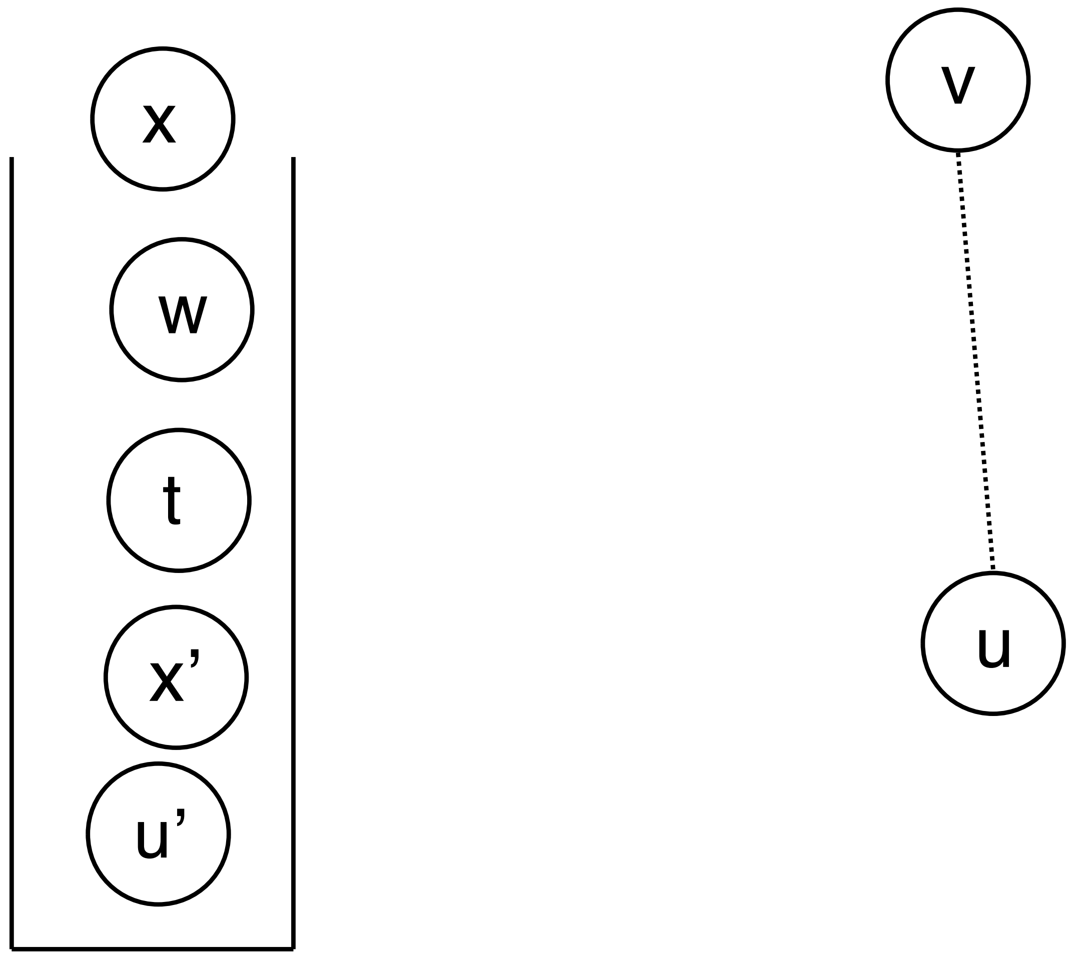


Example, k=3

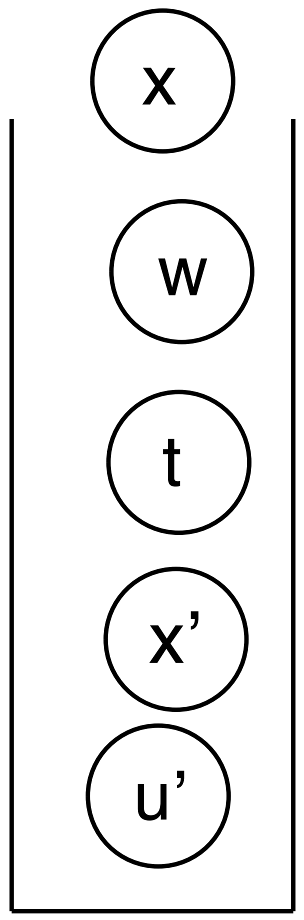


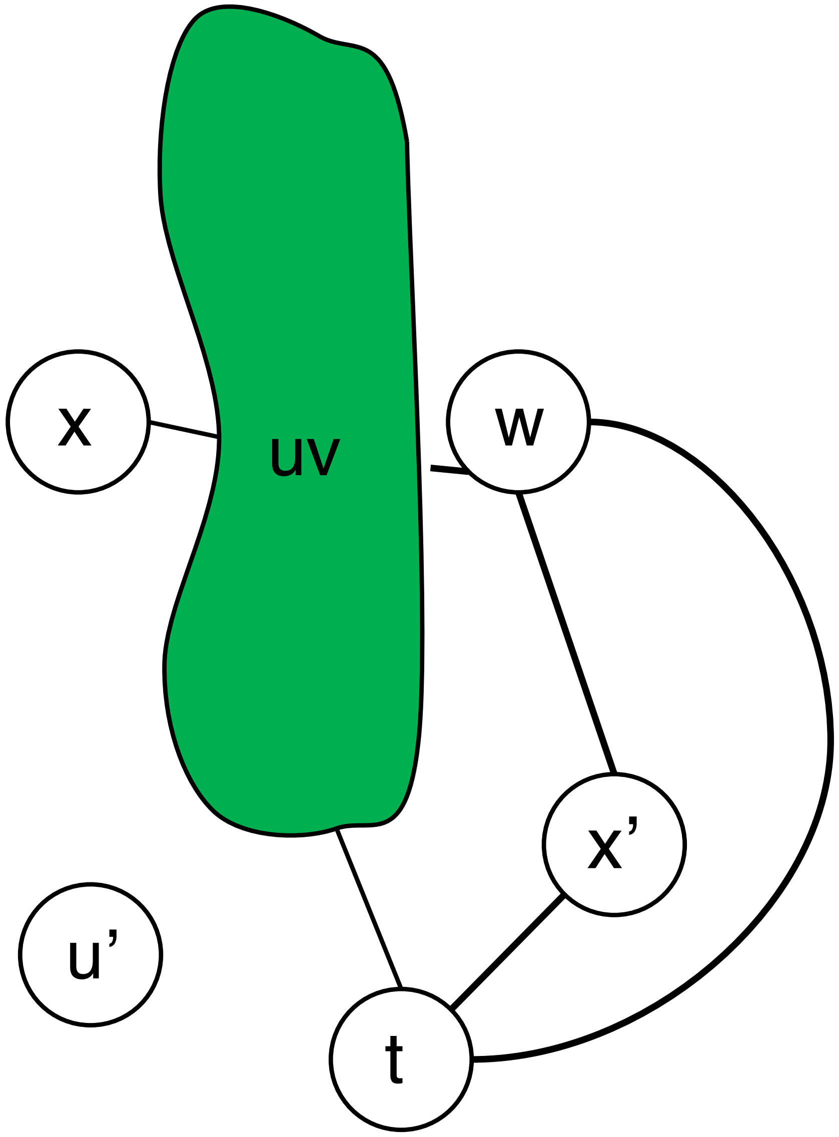


Example, k=3

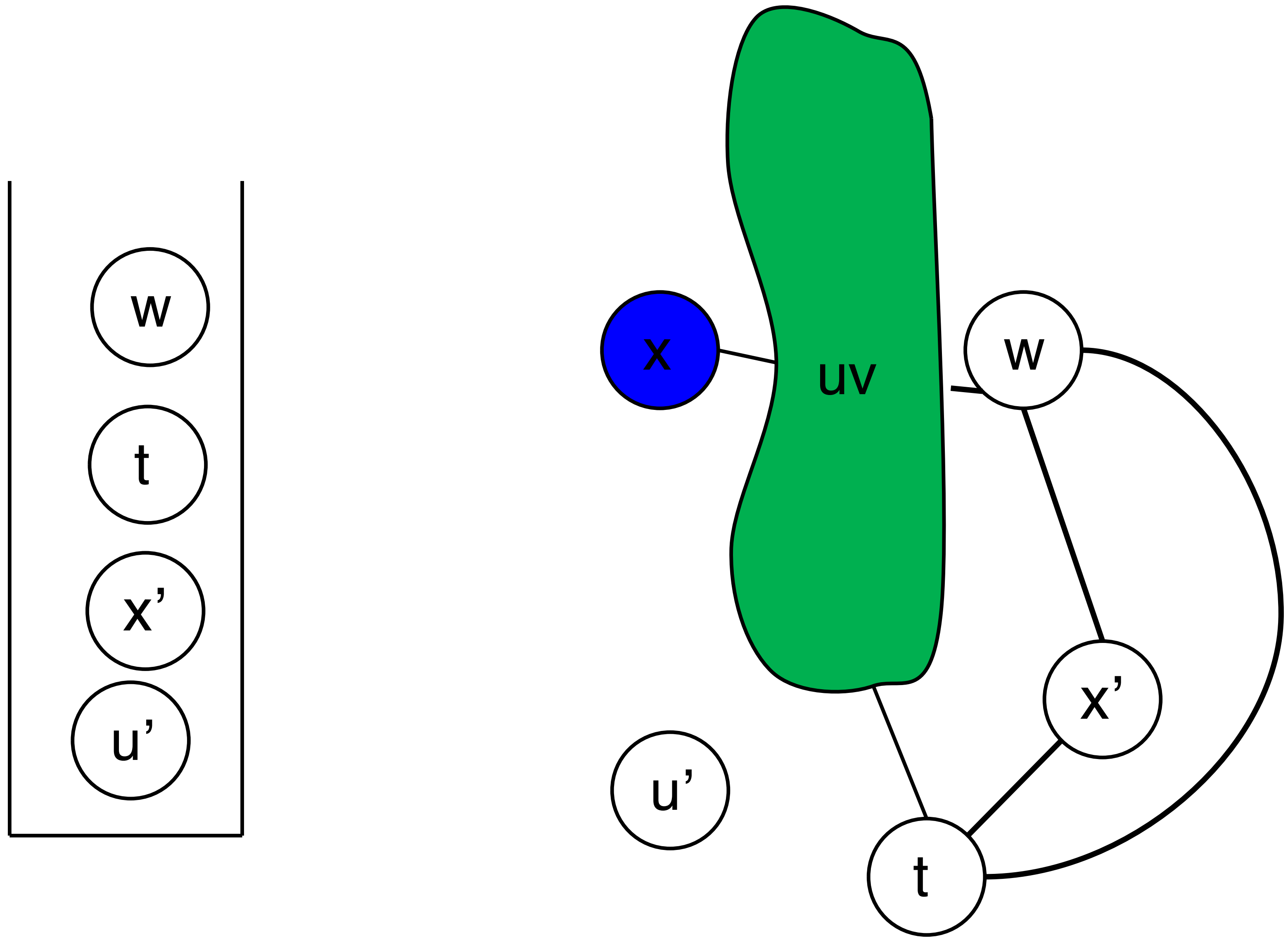


Example, k=3

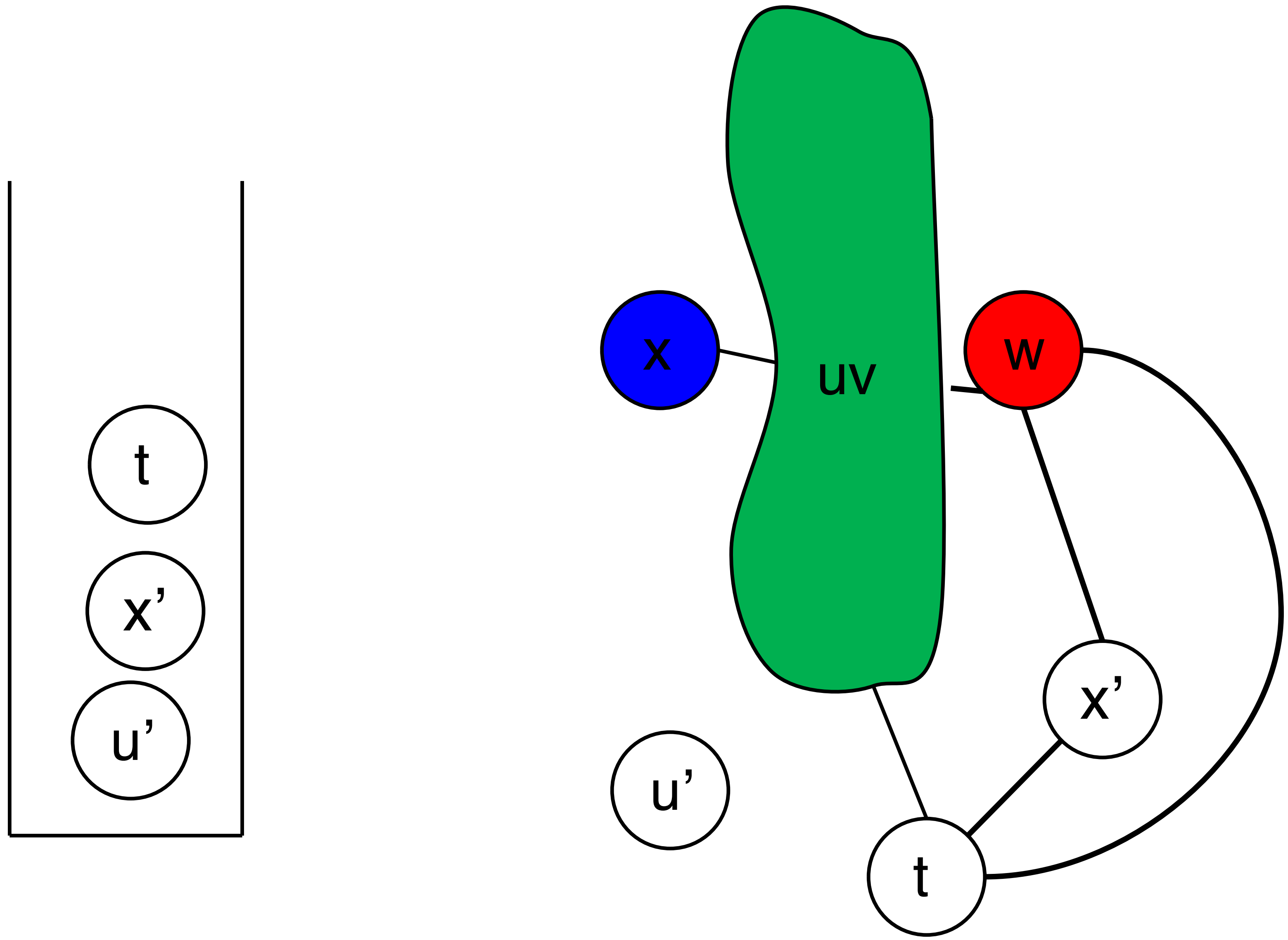




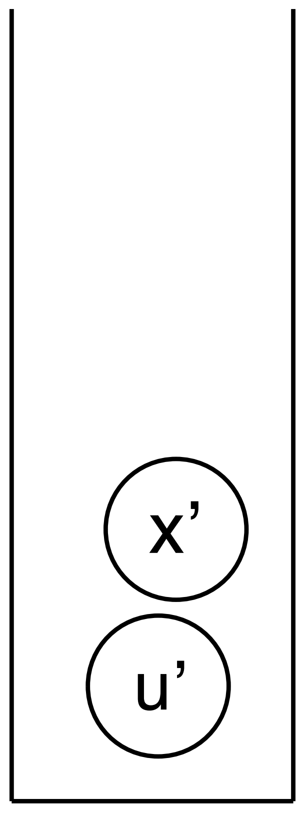
Example, k=3

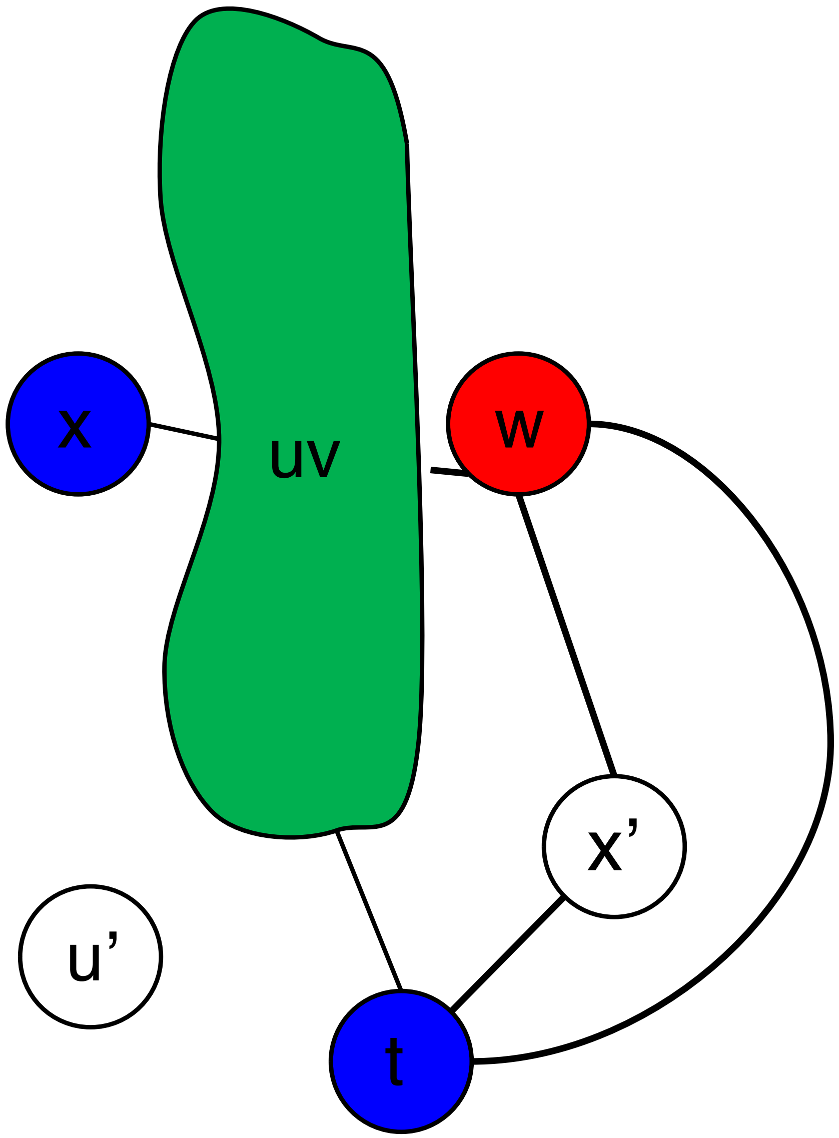


Example, k=3

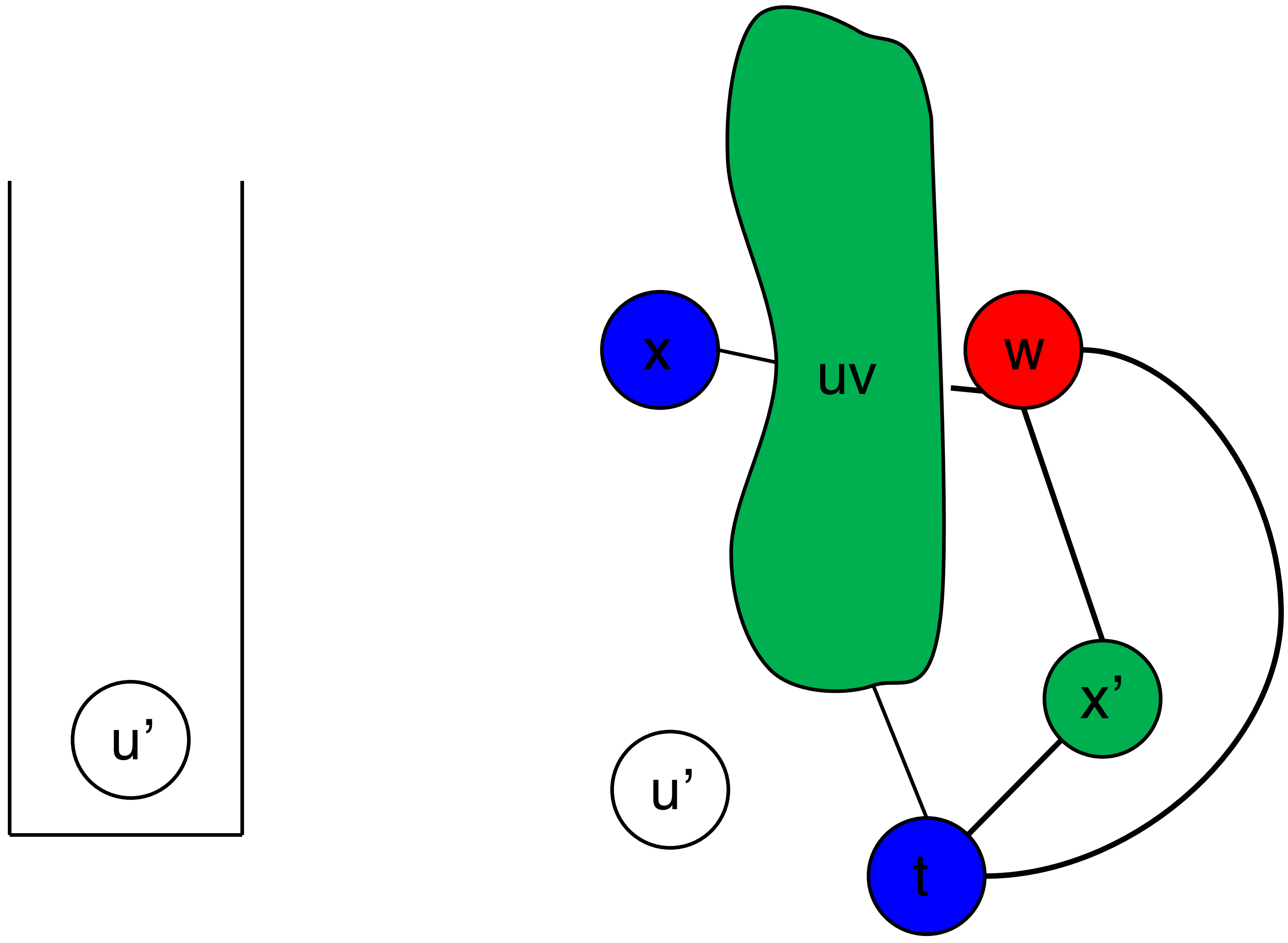


Example, k=3

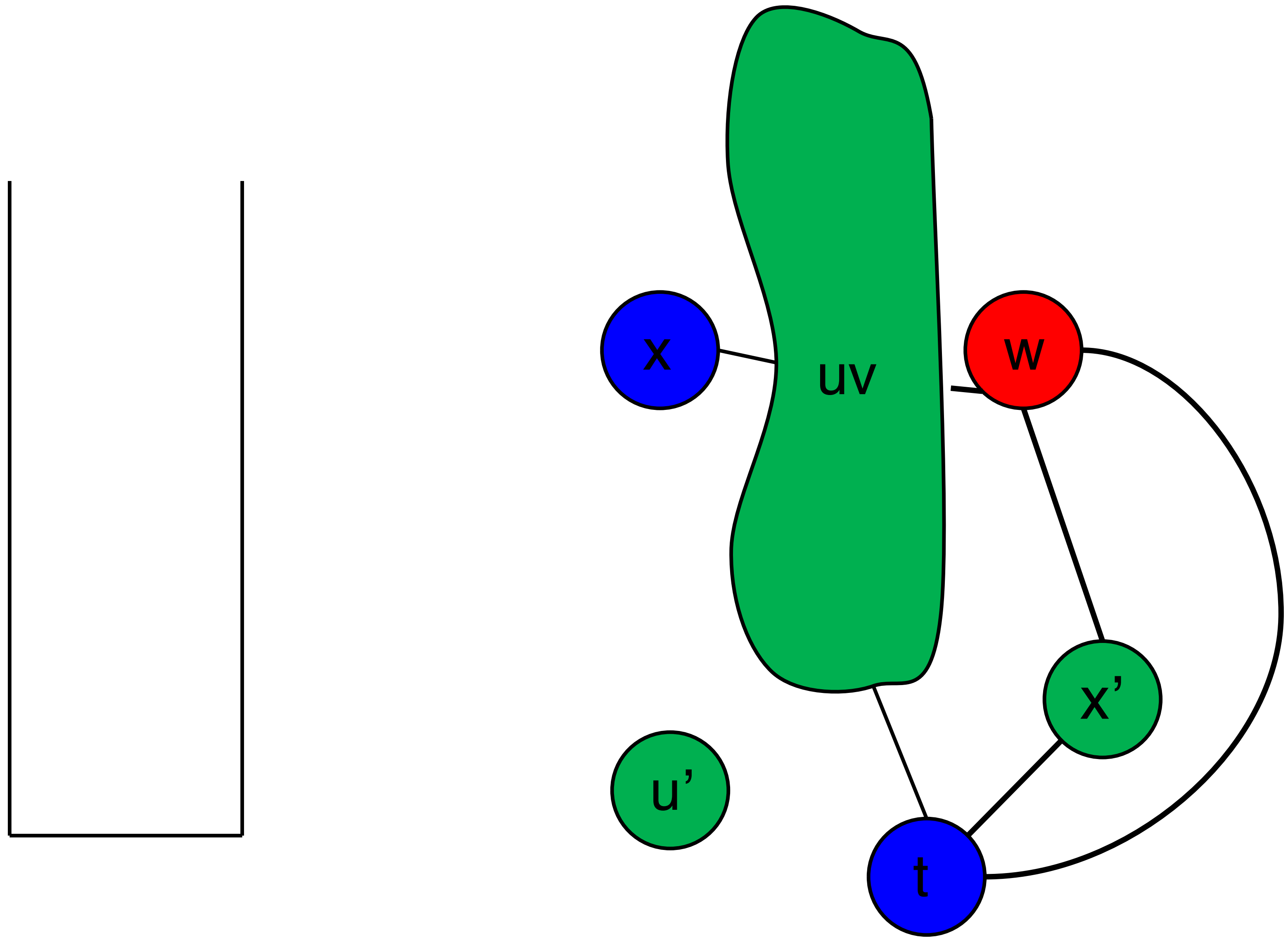




Example, k=3

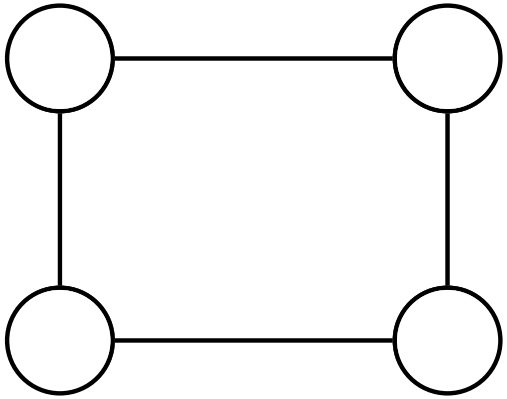


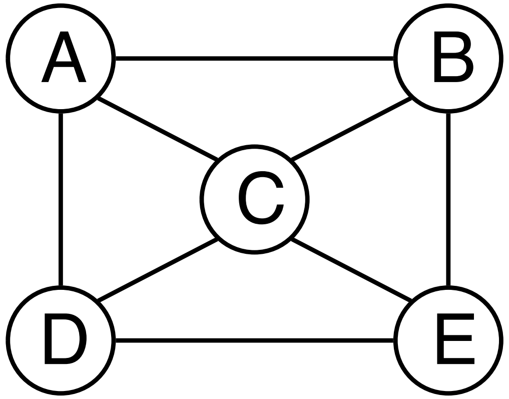
Example, k=3



Voila!

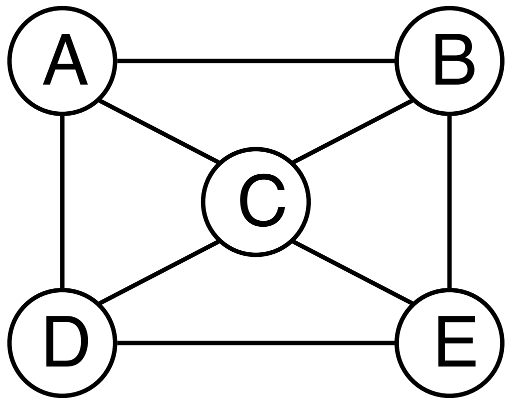
Alg not perfect





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...

Choosing Potential Spills

- When choosing a node to be a potential spill, we want to minimize its performance impact
- What should we choose to 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?

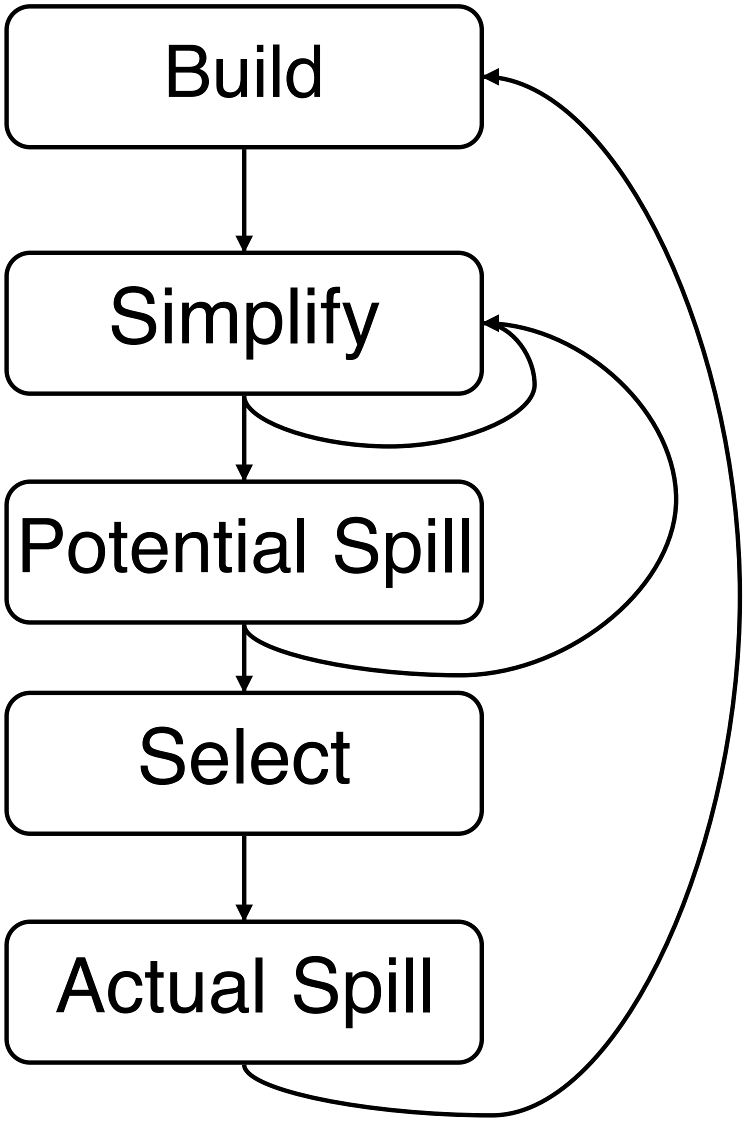
Setting Up For Better Spills

- 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!

K-coloring a graph

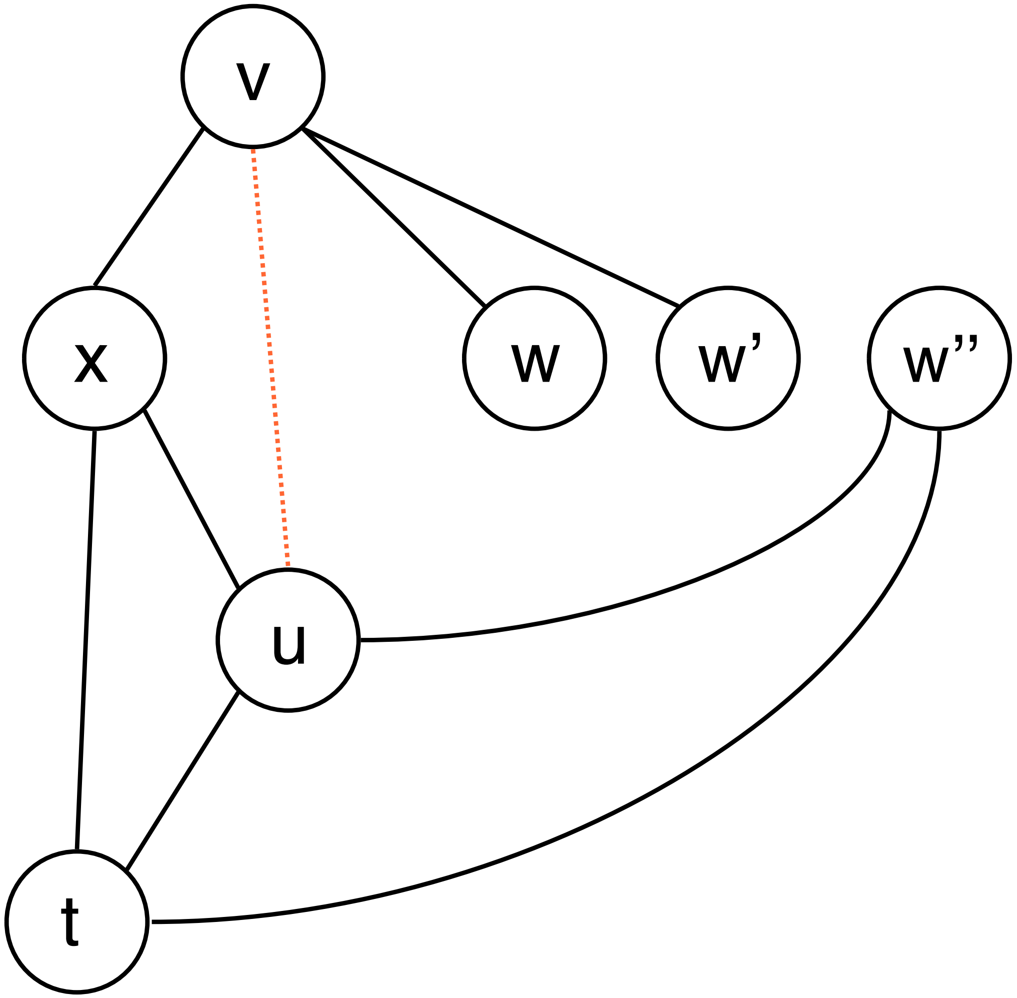
- Lets say we have a node, n, s.t. n $^{\boxed{?}}$ < k and let G' = G – {n}, then
  if G' can be k-colored, then G can be k-colored.
- Proof?
- This suggests the following optimistic heuristic:
- While |G| > 0
  - choose some n with degree < k
  - push n on stack
  - remove n from G
- While |S| > 0
  - pop n from S
  - color with a legal color

Where We Are



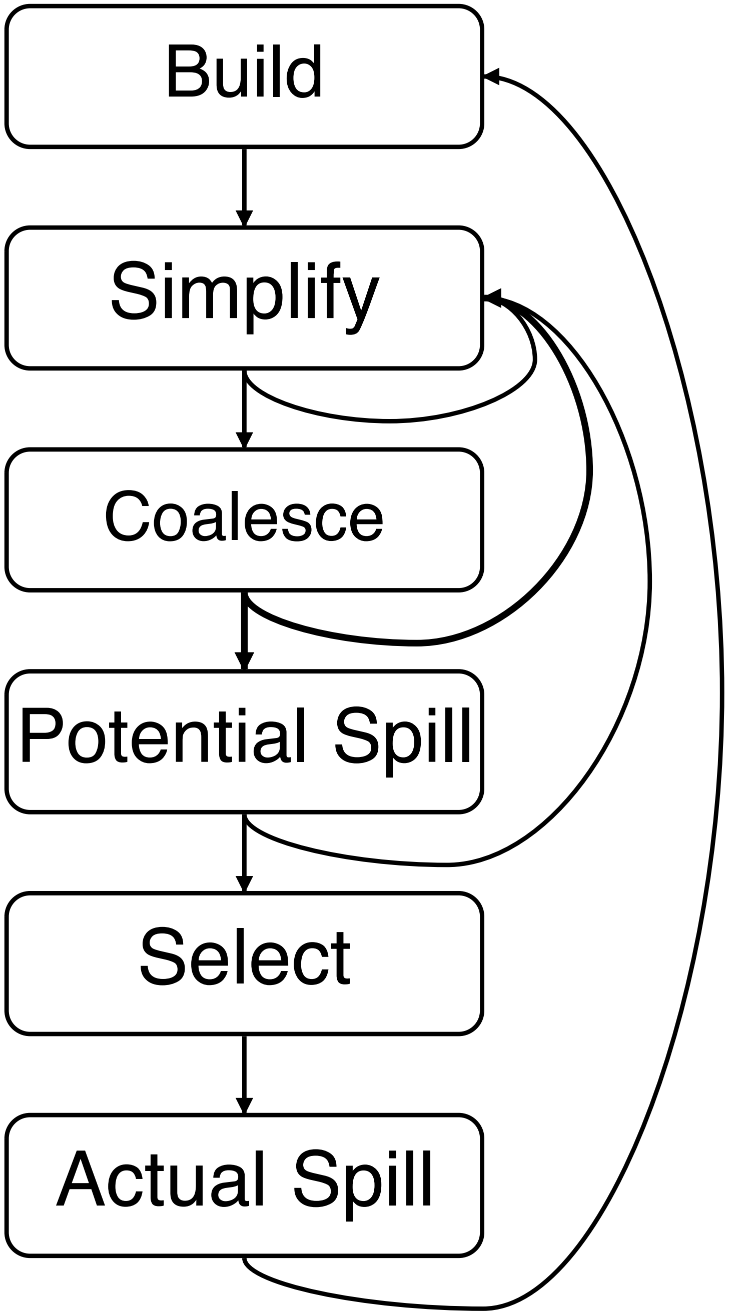
Coalescing

```
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

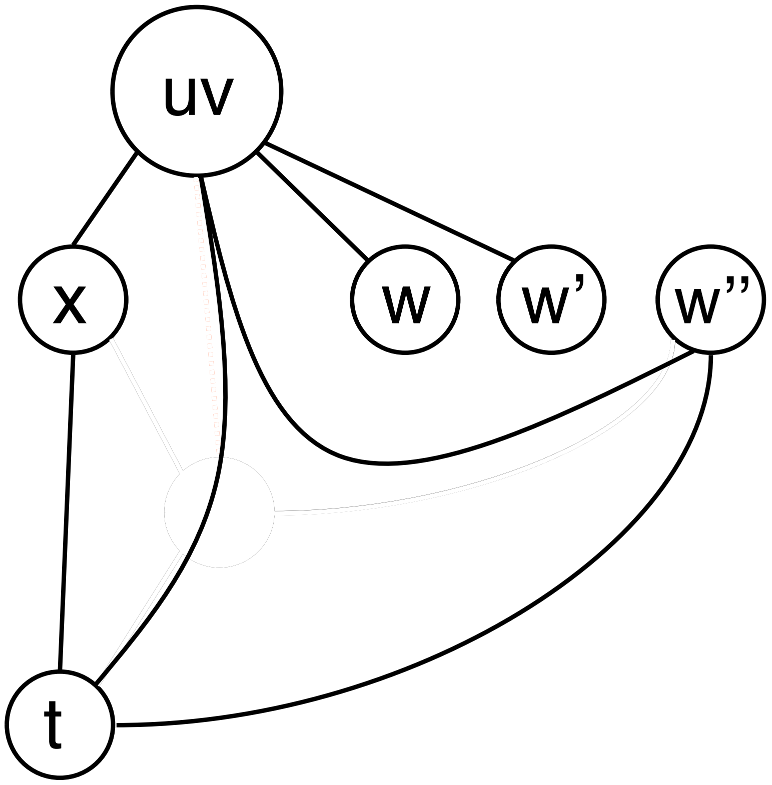

Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill

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

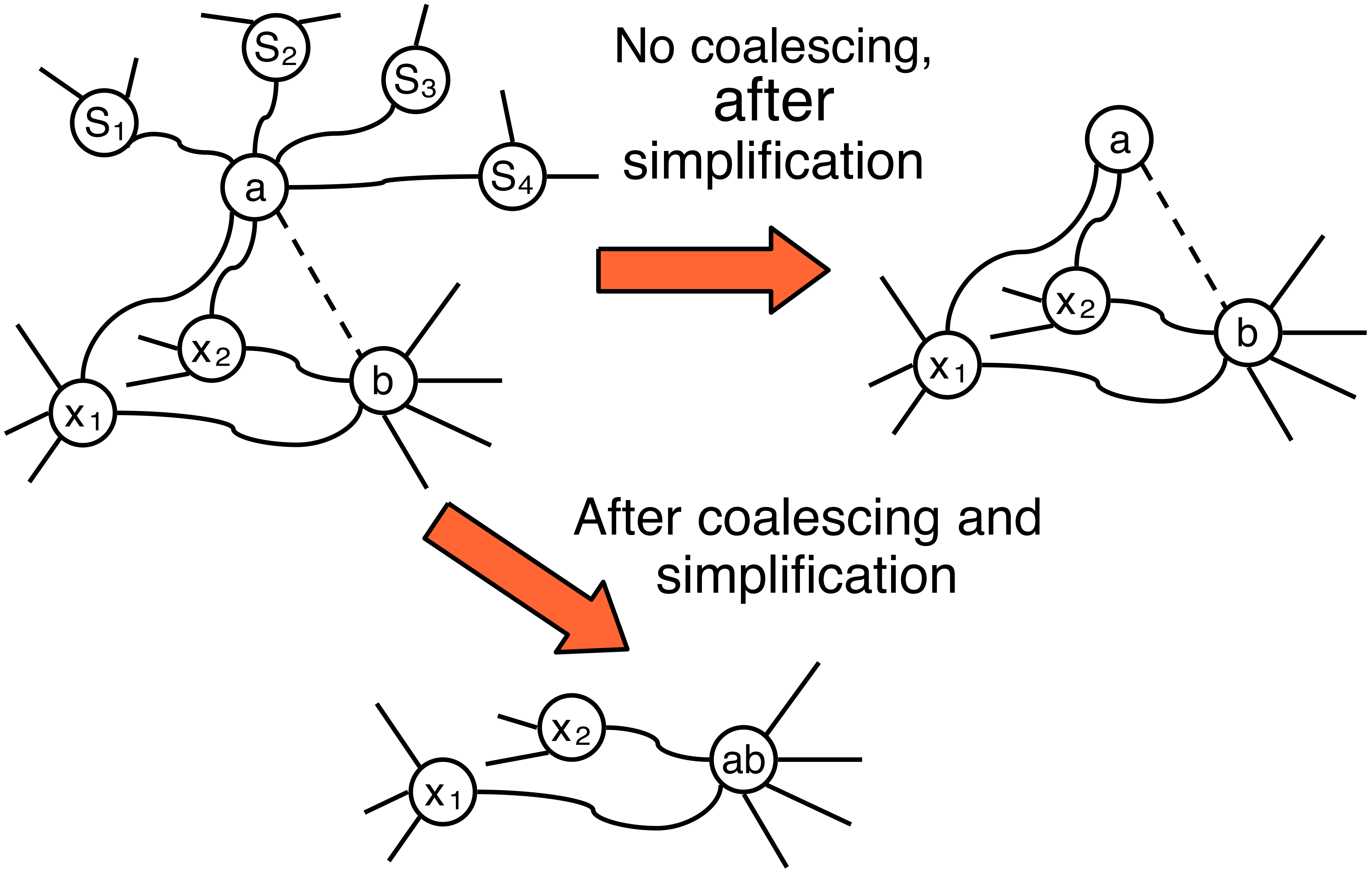


Preston

- 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



S_2
S_3
S_1
S_4
a
x_2
b
x_1
No coalescing, after simplification
a
x_2
b
x_1
After coalescing and simplification
x_2
ab
x_1

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

What about special registers?

- Instructions with register requirements

d ⍰ a * b

ret x

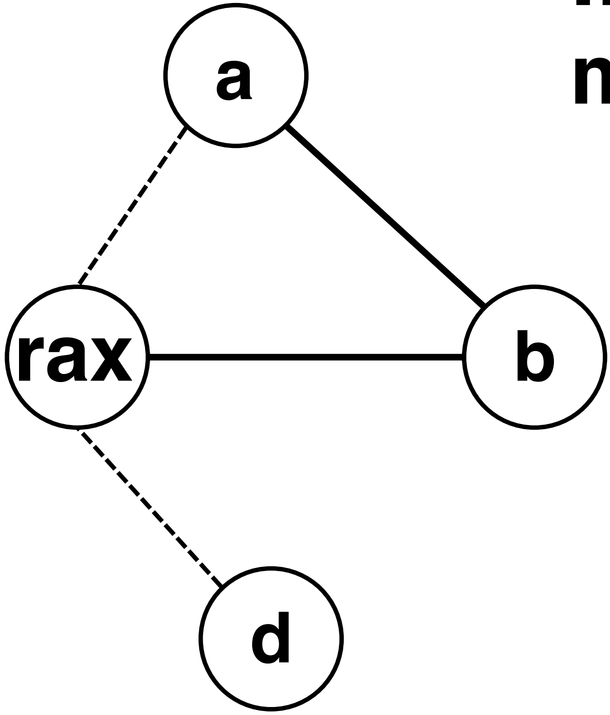
- Callee-save registers
  - x86-64: **RDI**, **RSI**, **RDX**, **RCX**, **R8**, **R9** must be saved by callee if callee wants to use them.

What about special registers?

- Instructions with register requirements

d ⍰ a * b

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

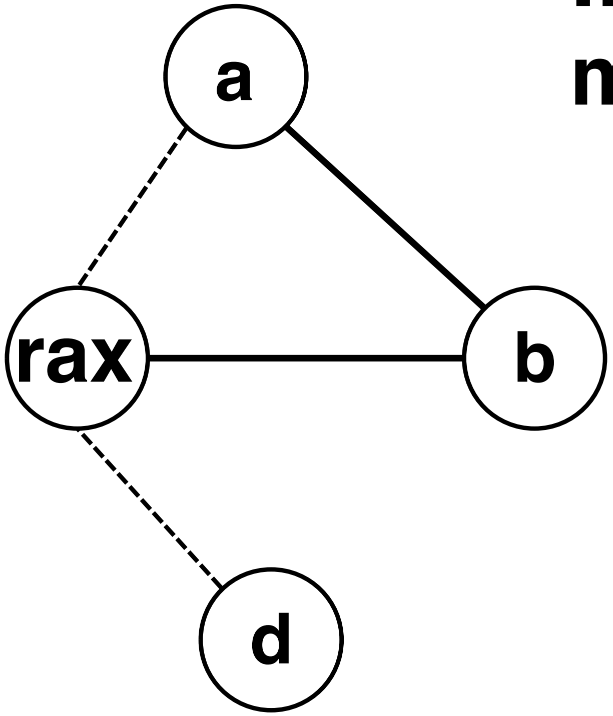


What about special registers?

- 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
```

$\Rightarrow$
```
movl a,   rax
imul b               ;  rdx,rax
movl rax,  d
```

```
ret  x
```

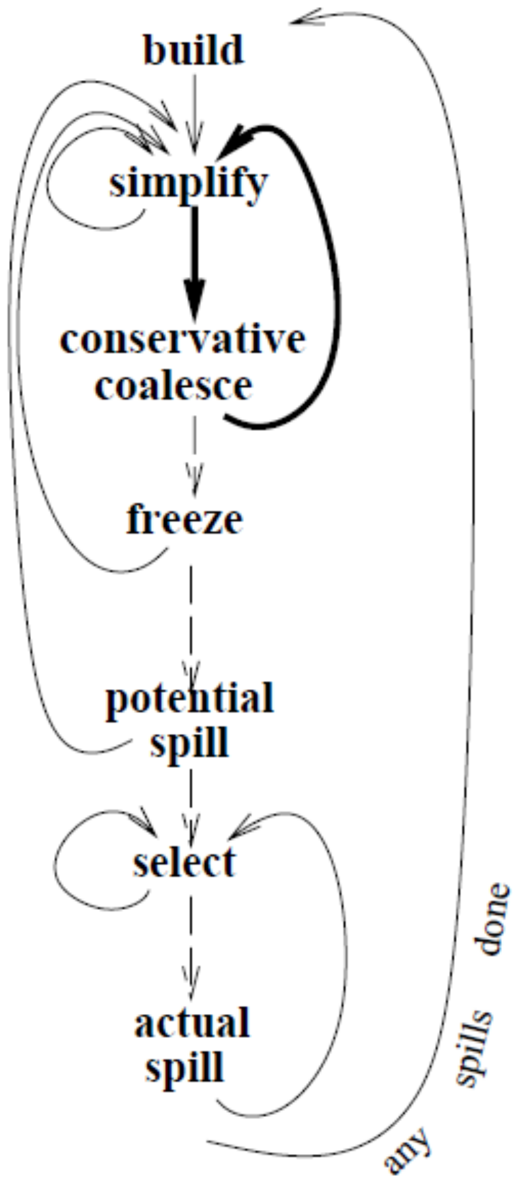
$\Rightarrow$
```
movl x,  rax
ret
```

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
```

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