Coalescing Register Allocation

CS4410: Spring 2013

Recap:

Basic Graph-Coloring Register Allocation

- Build interference graph G
  - use liveness analysis
- Simplify the graph G
  - If x has degree < k, push x and simplify G-{x}
  - if no such x, then we need to *spill* some temp.
  - spilling involves rewriting the code, and then start all over with a new interference graph.
- Once graph is empty, start popping temps and assigning them registers.
  - Always have a free register since sub-graph G-{x} can't have >= k interfering temps.

Spilling…

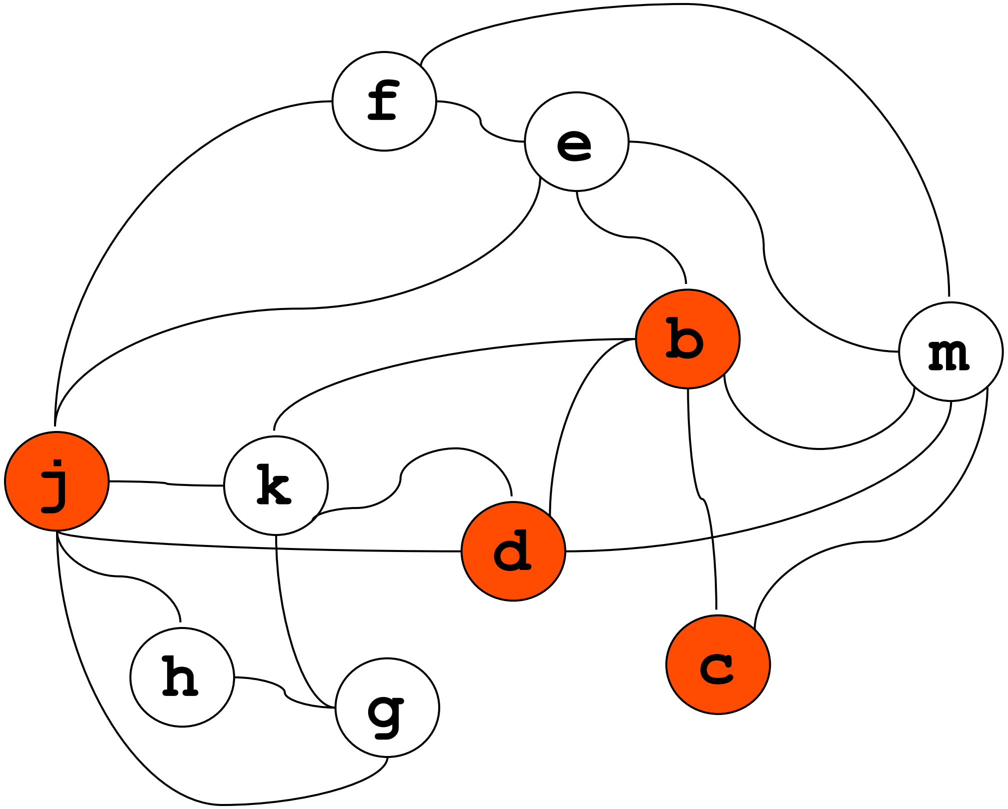
- Pick one of the nodes to spill.
  - Picking a high-degree temp will make it more likely that we can color the rest of the graph.
  - Picking a temp that is used infrequently will likely generate better code.
    - e.g., spilling a register used in a loop when we could spill one accessed outside the loop is a bad idea…
- Rewrite the code:
  - after definition of temp, write it into memory.
  - before use of temp, load it into another temp.
  - simplifies things to reserve a couple of registers.

Coalescing Register Allocation

- If we have "x := y" and x and y have no edge in the interference graph, we *might* be able to assign them the same color.
  - so this would translate to “ti := ti" which we could simplify away.
- One idea is to optimistically *coalesce* nodes in the interference graph.
  - just take the edges to be the union.
  - but of course, this may make a k-colorable graph uncolorable!

Example from book

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



Brigg's Strategy:

It's safe to coalesce x & y if the resulting node will have fewer than k neighbors with degree >= k.

George's strategy:

We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.

New Algorithm:

- **Build**: construct the interference graph.
  - label each node as *move-related* or *not move-related*.
  - move-related: source or destination of a move.
- **Simplify**: remove a non-move-related node of low degree from the graph & push on stack. Continue until all nodes are move related and/or have high degree.

New Algorithm Continued

- **Coalesce:** coalesce nodes on the reduced graph using either Briggs' or George's conservative strategy.
  - Simplifying will hopefully have reduced the degree on many of the nodes.
  - Possibly re-mark the nodes that were coalesced as non-move-related.
  - go back to simplifying non-move-related, low-degree nodes.

New Algorithm Continued

- **Freeze**: if we have some nodes x & y of low degree, but they are move-related and cannot be safely coalesced, we *freeze* the move involving x & y.
  - i.e., we can't coalesce x & y.
  - so go back and treat them as non-move-related.
  - then, hopefully we can remove them with simplify, then do more coalescing, etc.

Algorithm Continued:

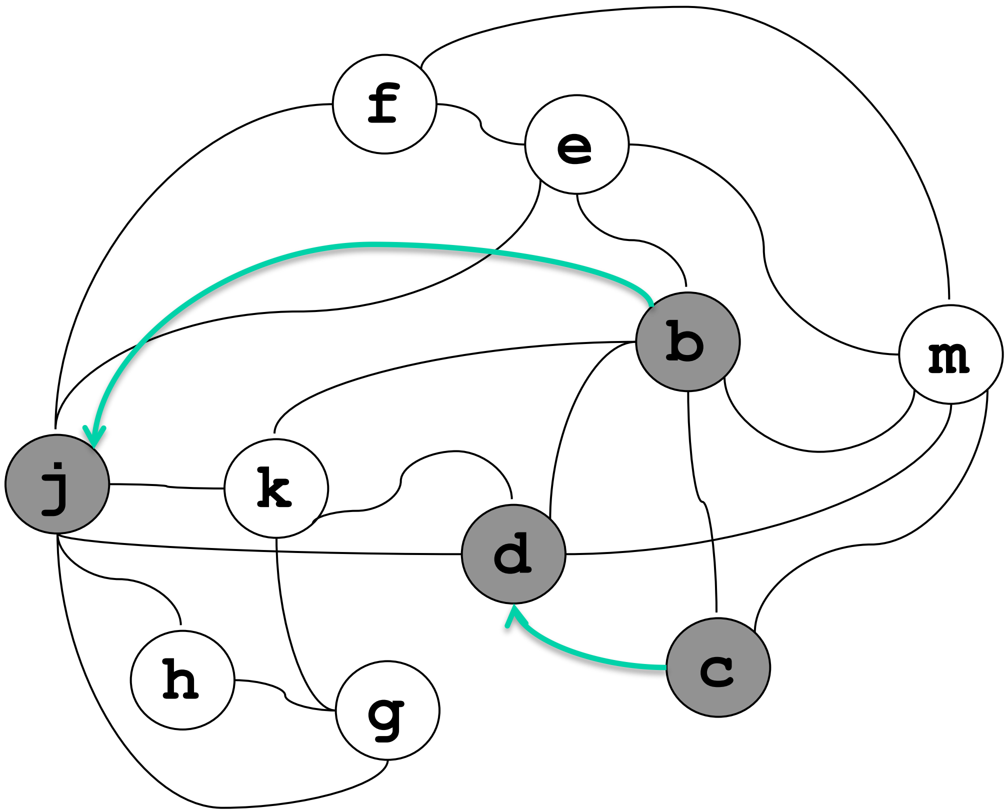
- **Spill:** we've gotten down to only high-degree nodes. Pick a *potential* spill candidate and push it on the stack.
  - We don't actually do the spill yet, but rather record that this node may need to be spilled.
  - Just assume that it will no longer interfere with any other temp, so remove their edges.
  - Go back and try to simplify/coalesce/freeze the graph some more.

Algorithm Continued.

- **Select**: once we get the empty graph, start popping nodes off the stack and assign them colors.
  - we may not have a free color when we run into a potential spill nodes.
  - in this case, record that this node needs to be actually spilled.
  - if we reserve two registers, then we don't have to iterate, but if we re-use fresh temps, then we need to iterate constructing a fresh interference graph, etc.

Example from book

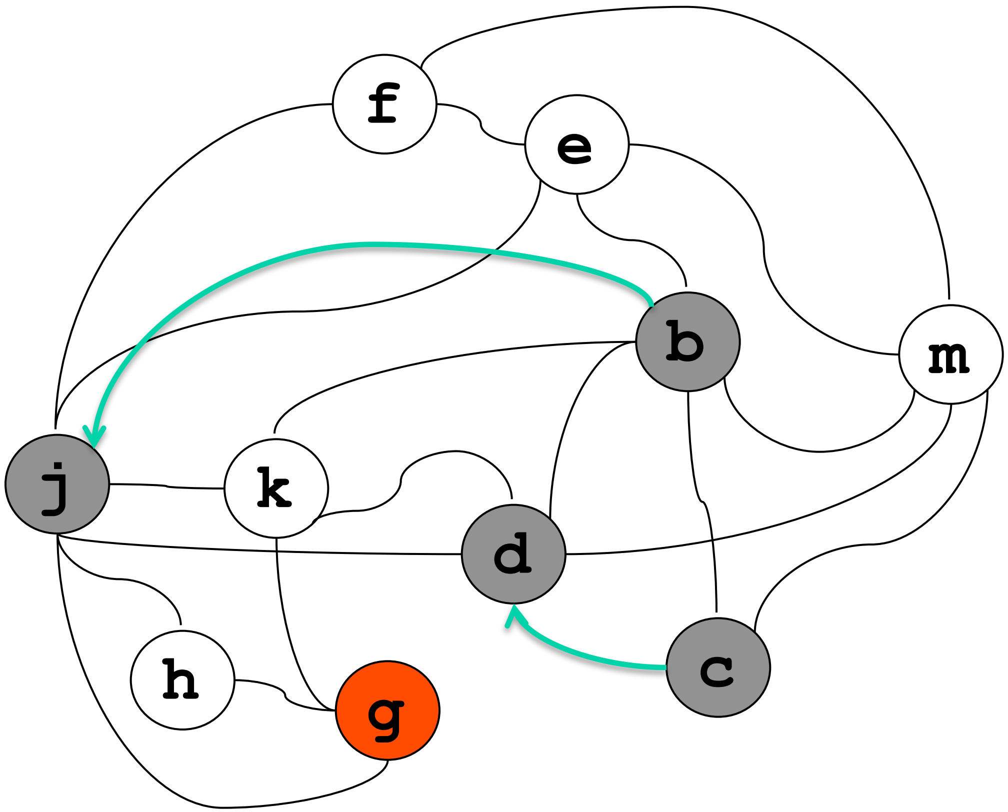
Stack:



j & **b**, **c** & **d** are move-related

Example from book

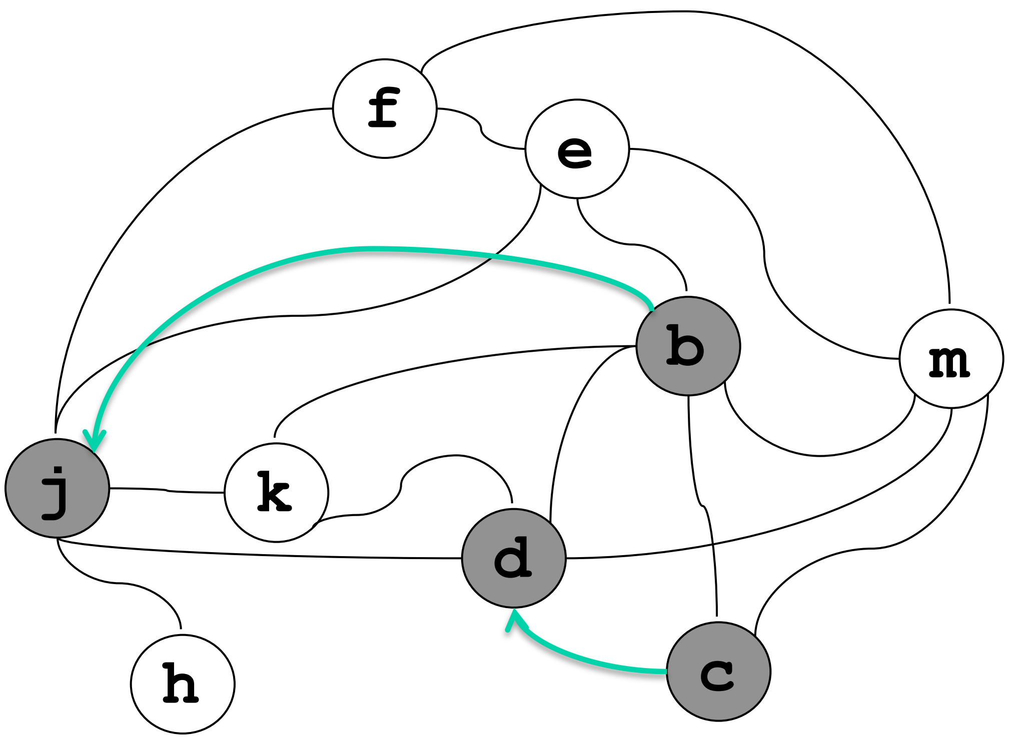
Stack:



Example from book

Stack:

g

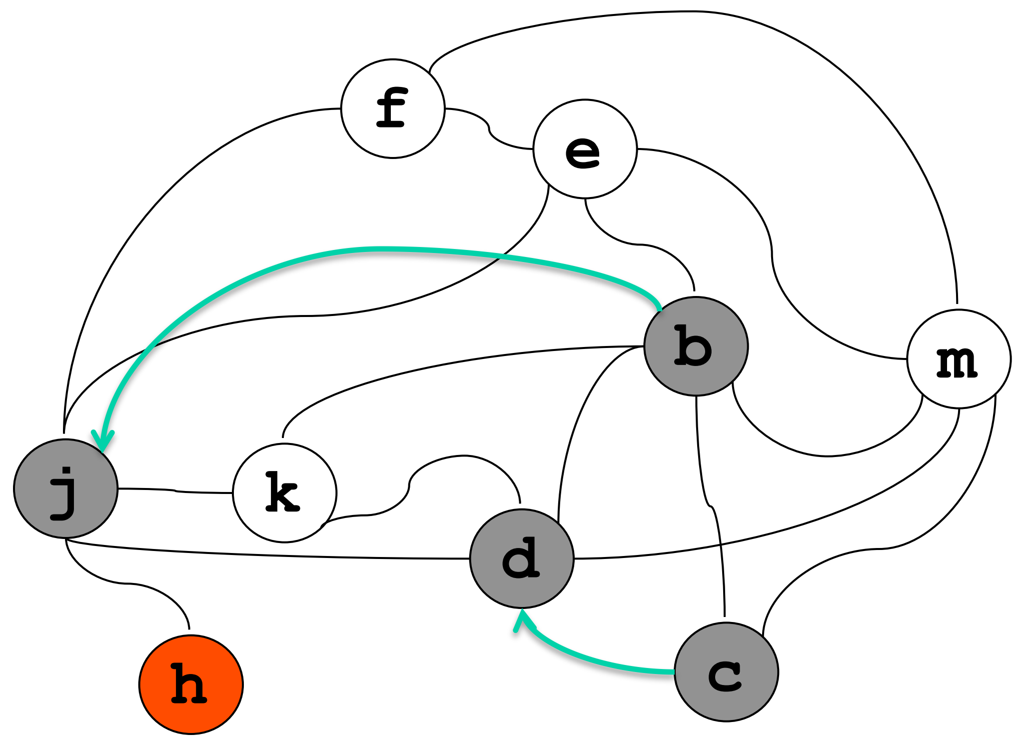


Example from book

Stack:

g

h



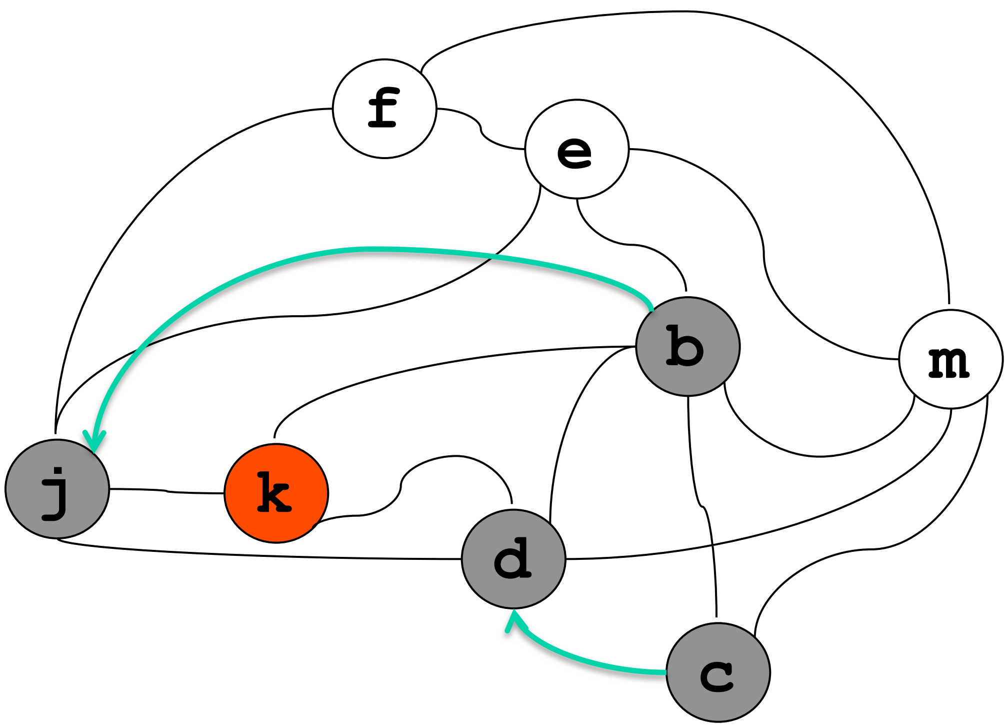
Example from book

Stack:

g

h

k



Example from book

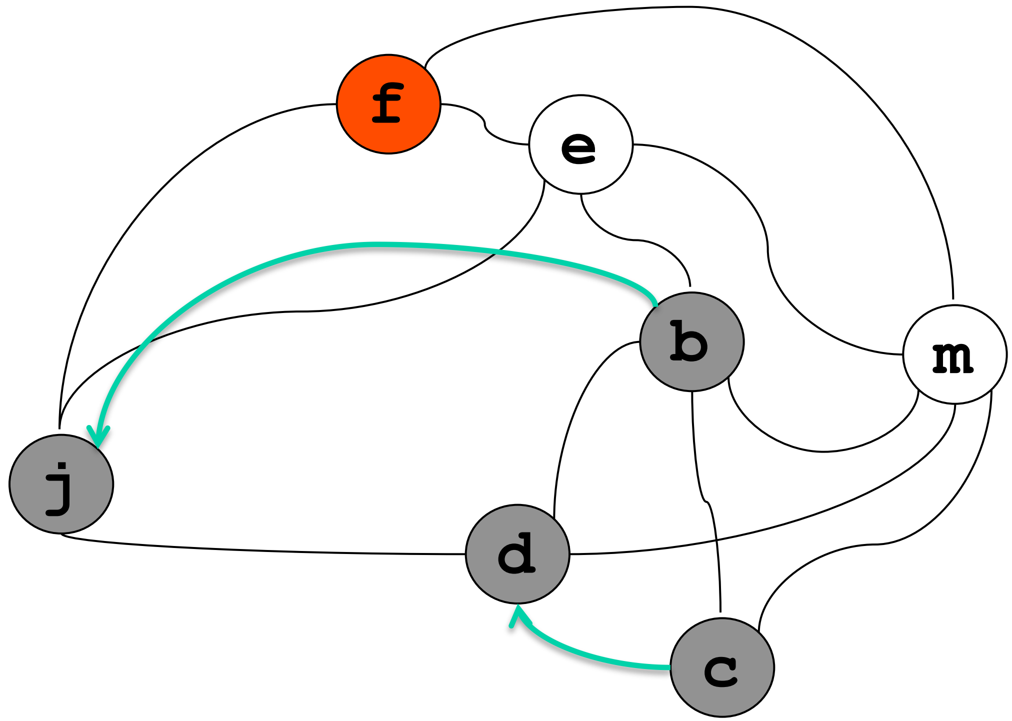
Stack:

g

h

k

f



Example from book

Stack:

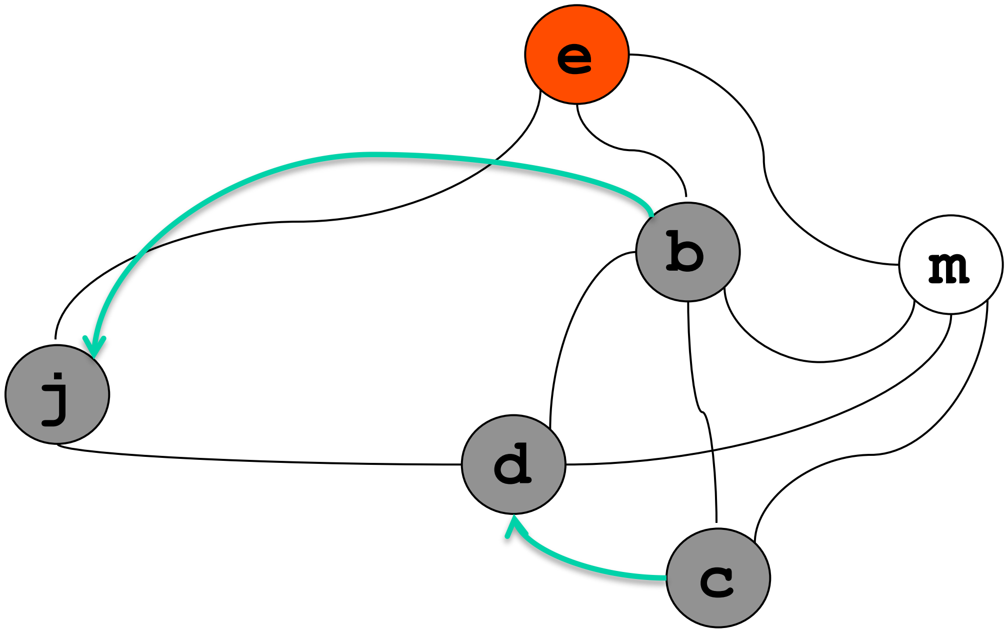
g

h

k

f

e



Example from book

Stack:

g

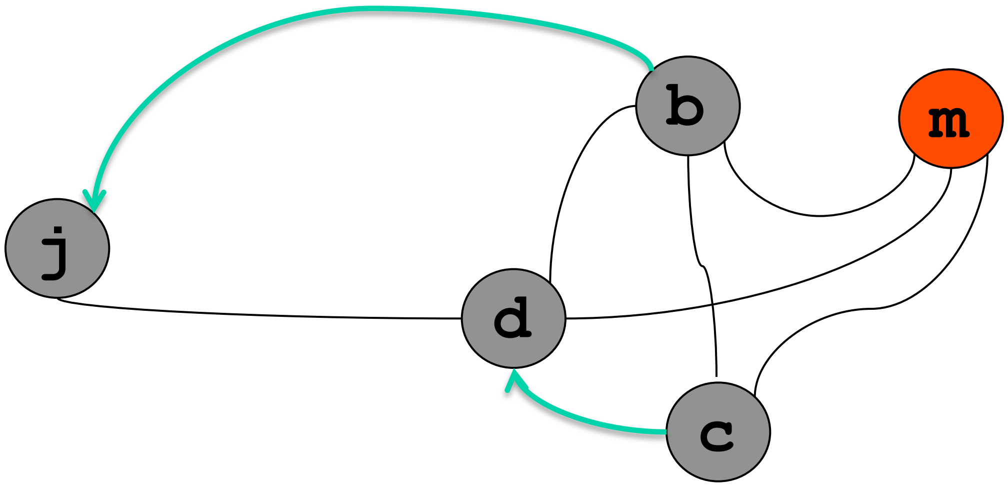
h

k

f

e

m



Example from book

Stack:

g

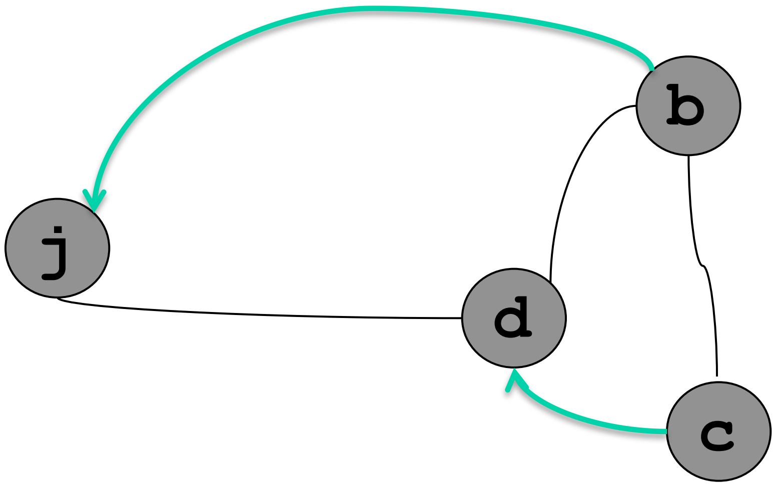
h

k

f

e

m



At this point, all nodes are move-related.
So start coalescing...

Example from book

Stack:

g

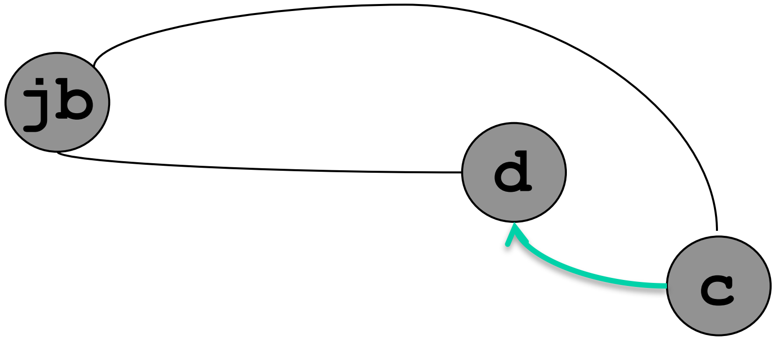
h

k

f

e

m



Example from book

Stack:

g

h

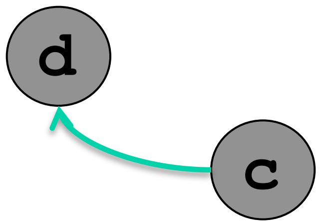
k

f

e

m

jb



Example from book

Stack:

g

h

k

f

e

m

jb

Example from book

Stack:

`g`

`h`

`k`

`f`

`e`

`m`

`jb`

`dc`

Now Select…

Stack:

g

h

k

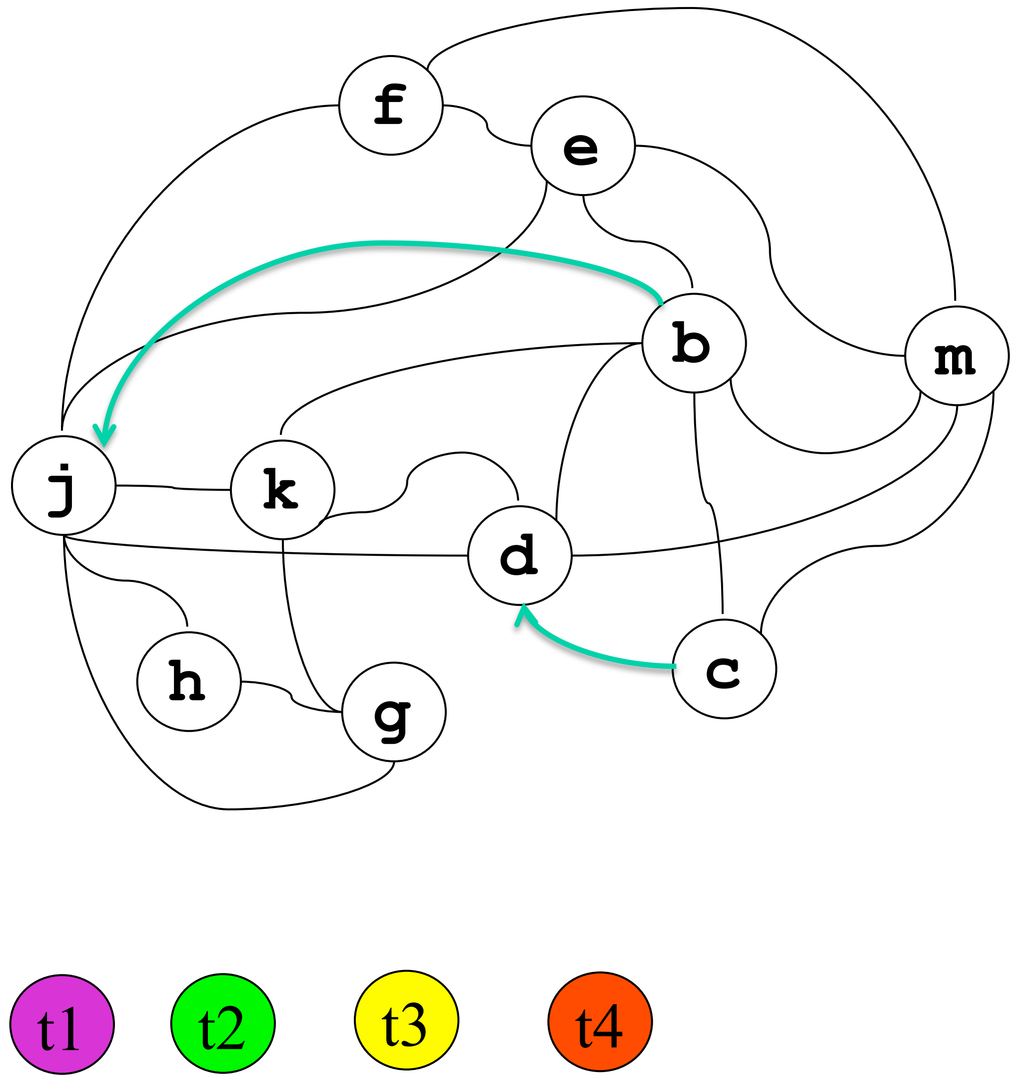
f

e

m

jb

cd



Now Select…

Stack:

g

h

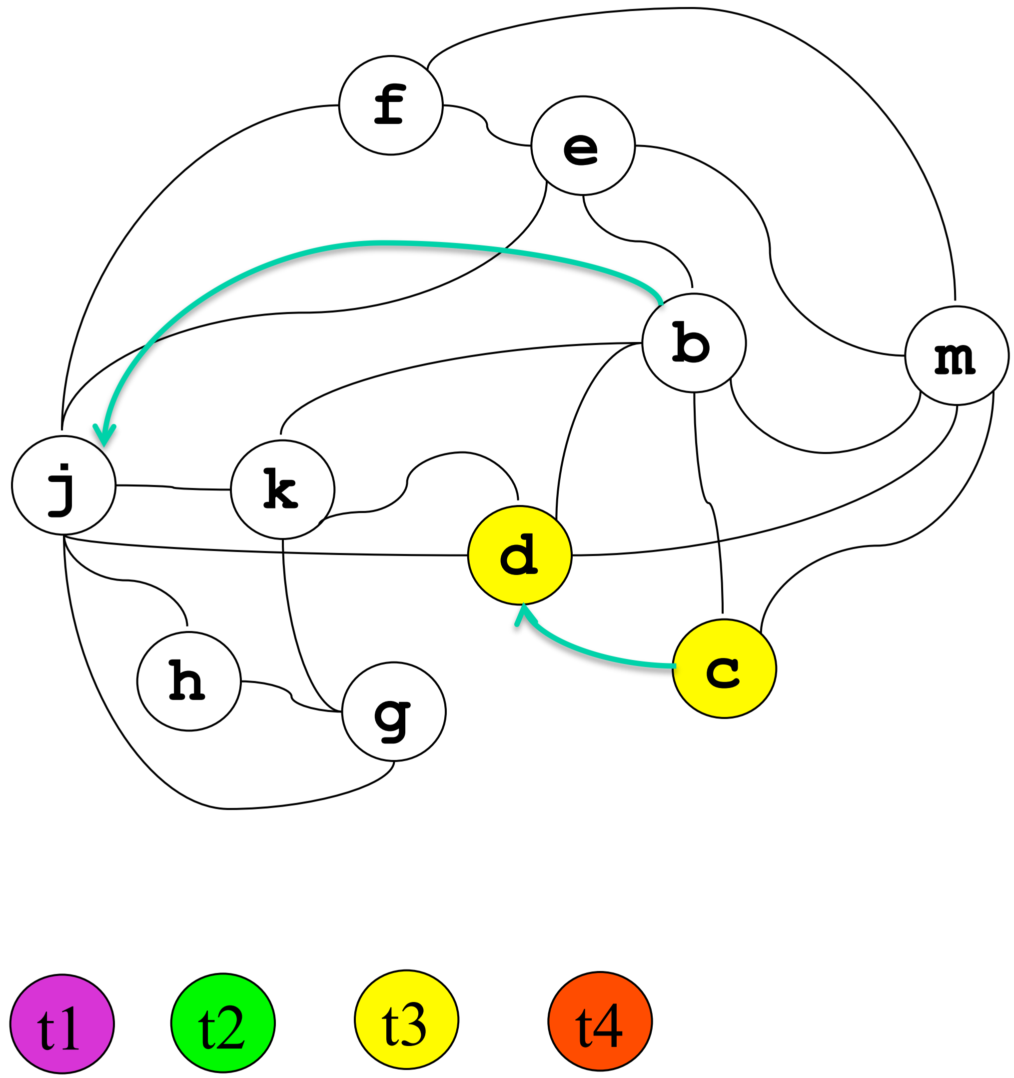
k

f

e

m

jb



Now Select…

Stack:

g

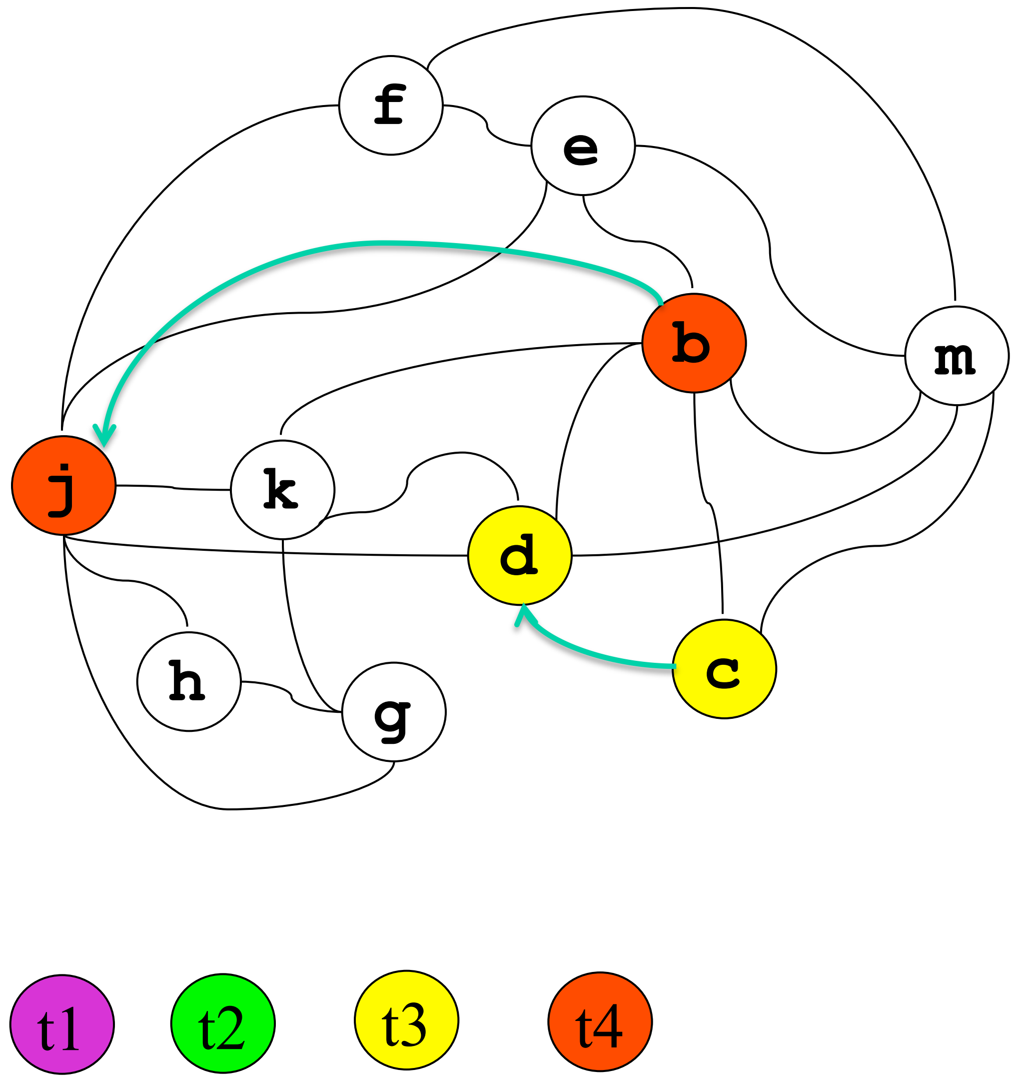
h

k

f

e

m



Now Select…

Stack:

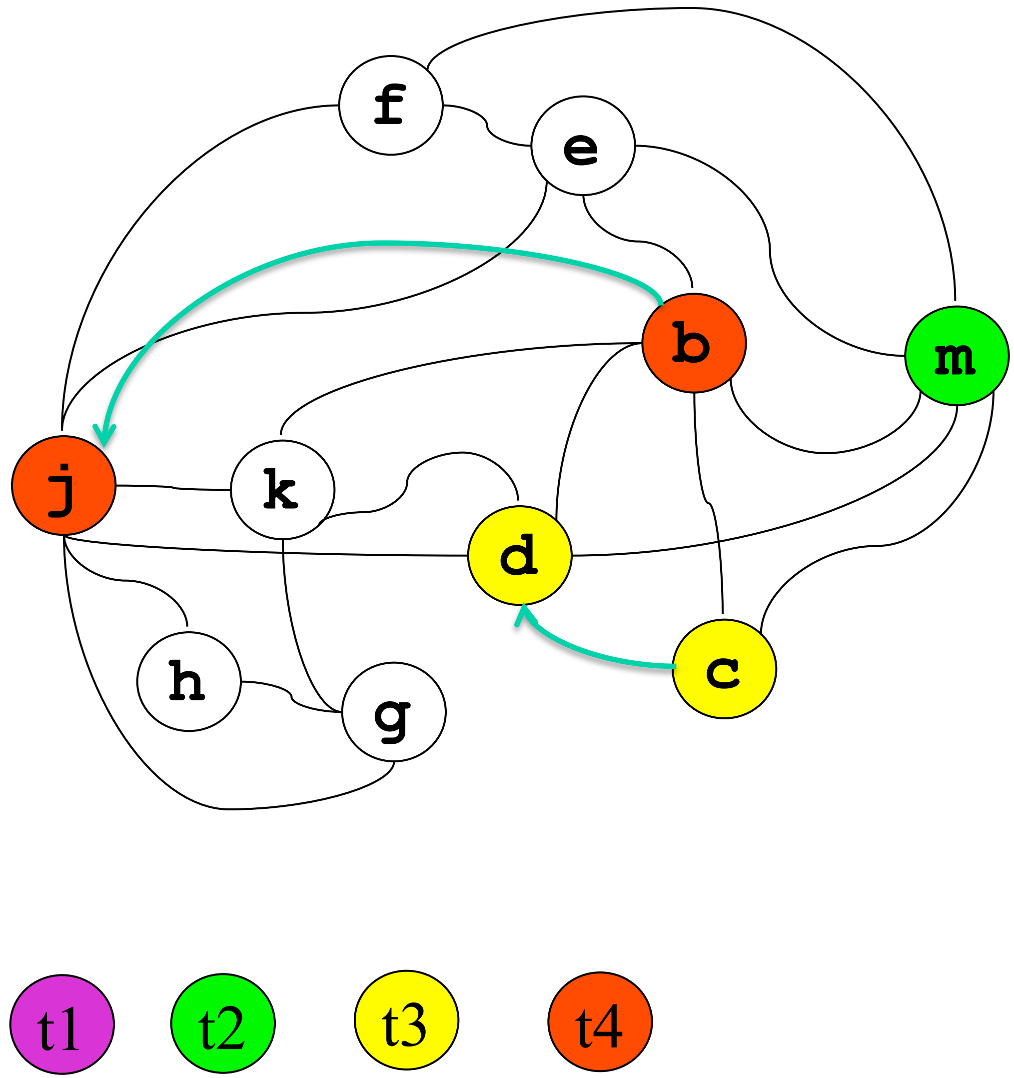
g

h

k

f

e



Now Select…

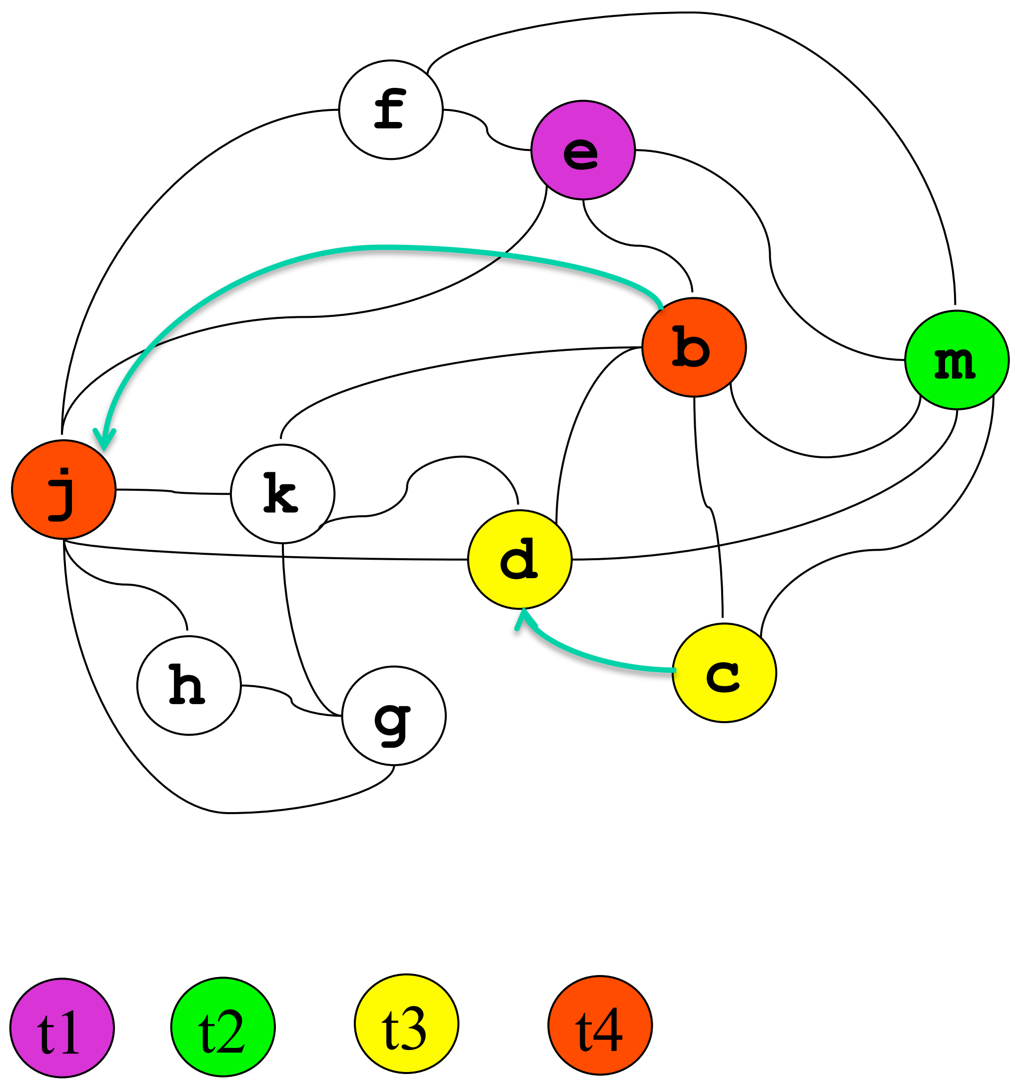
Stack:

g

h

k

f



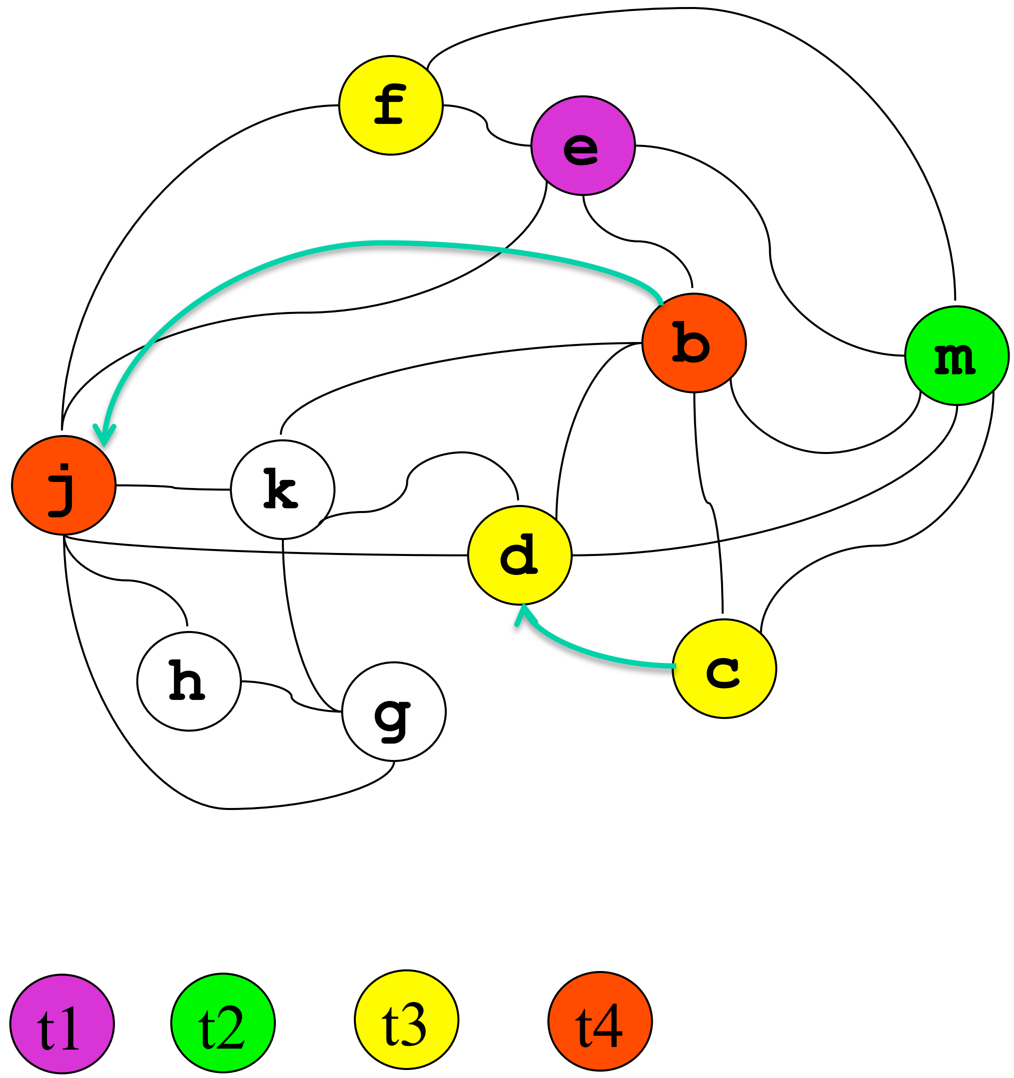
Now Select…

Stack:

g

h

k

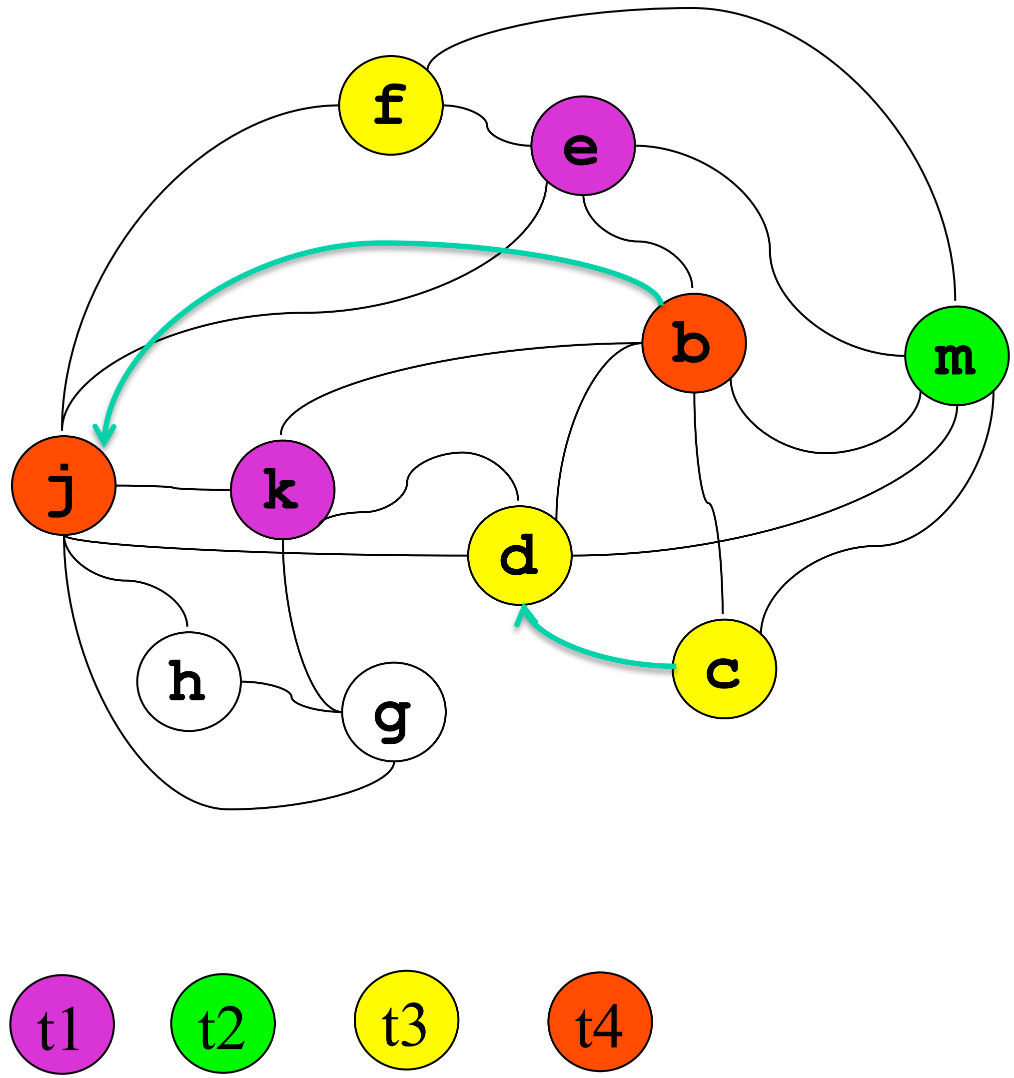


Now Select…

Stack:

g

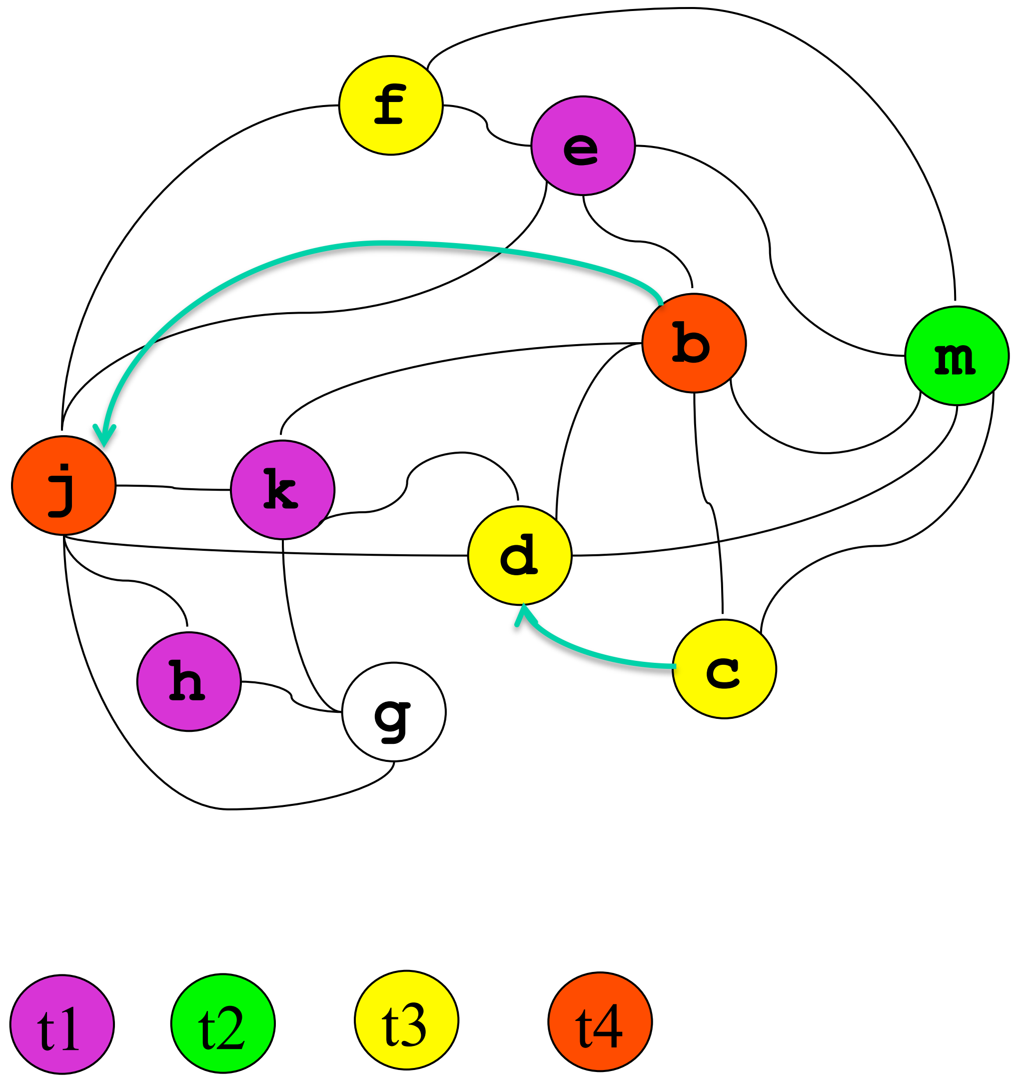
h



Now Select…

Stack:

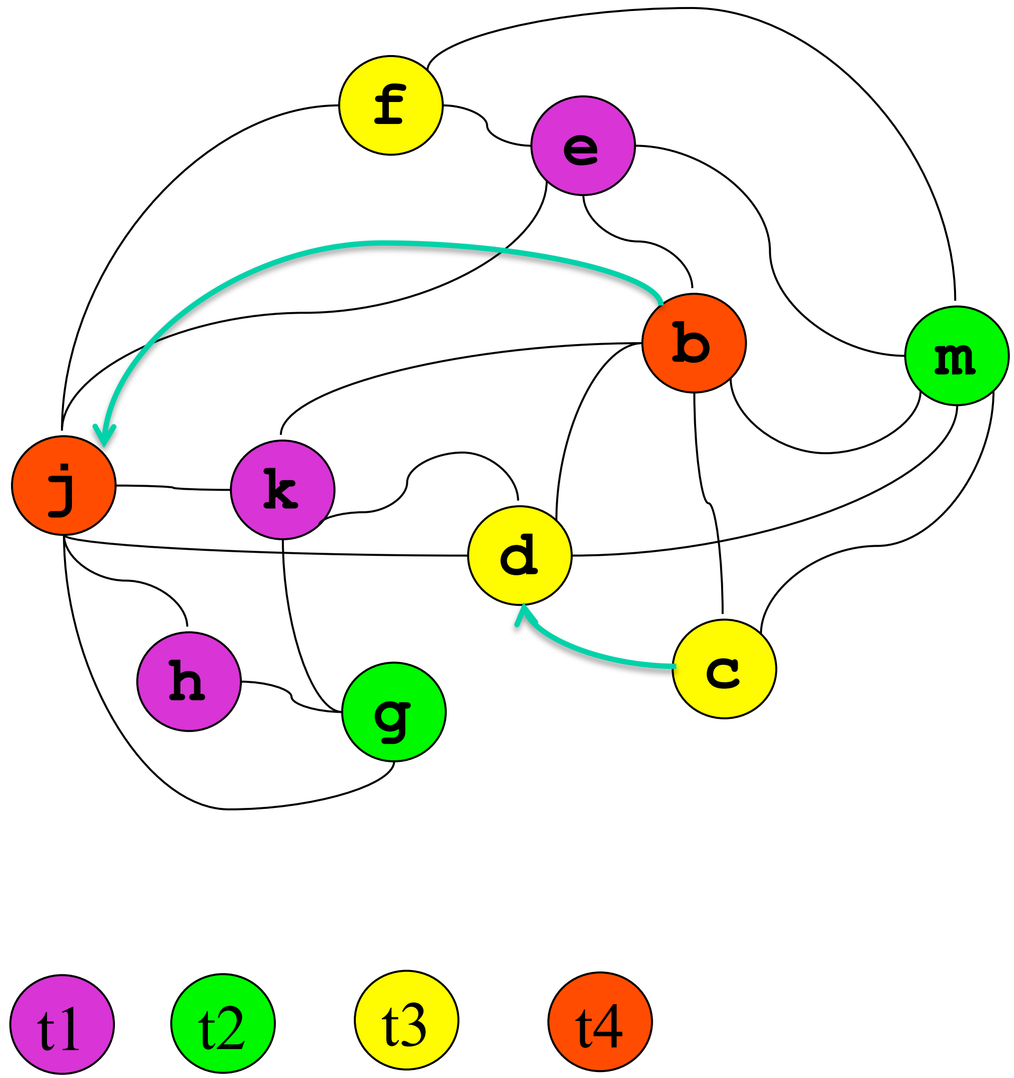
`g`



Now Select…

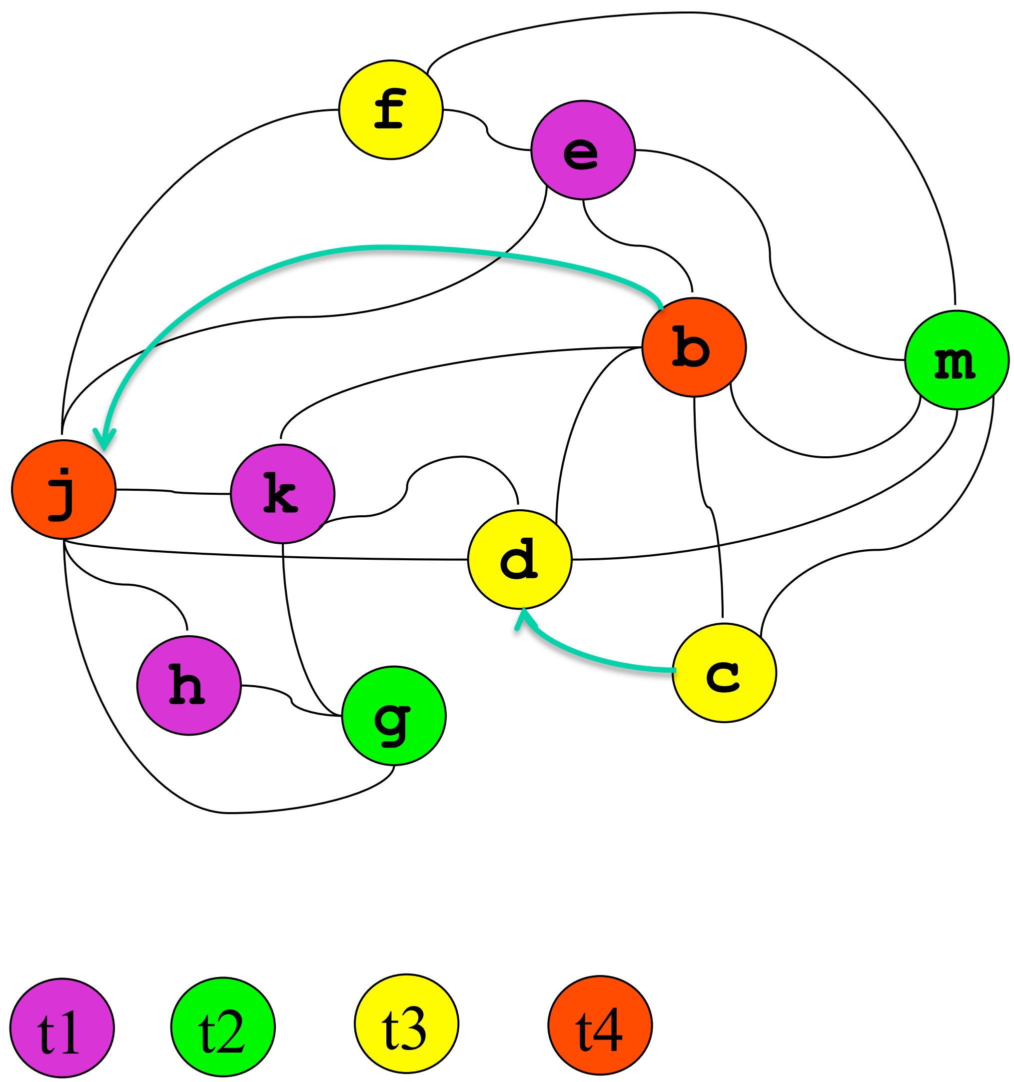
Stack:

`g`



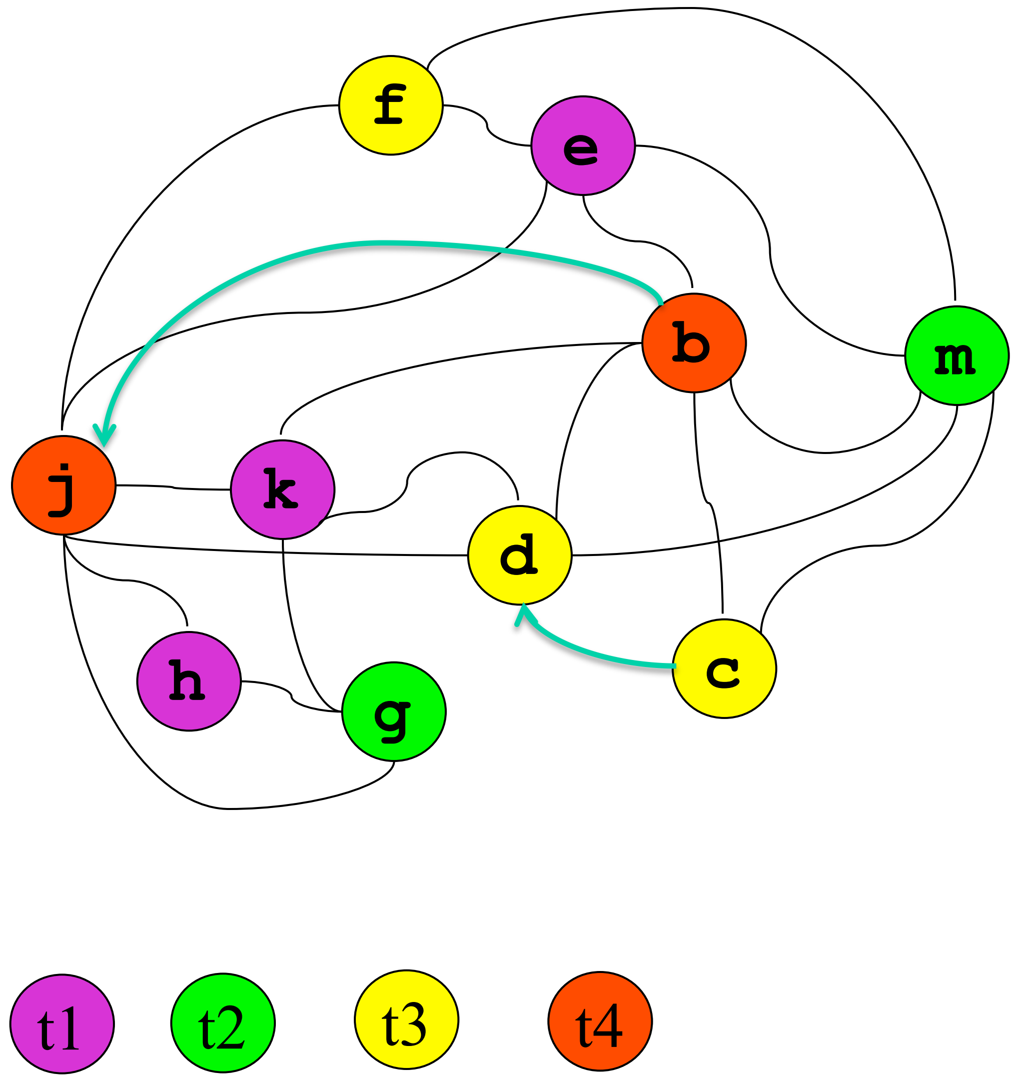
Now Rewrite Code…

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



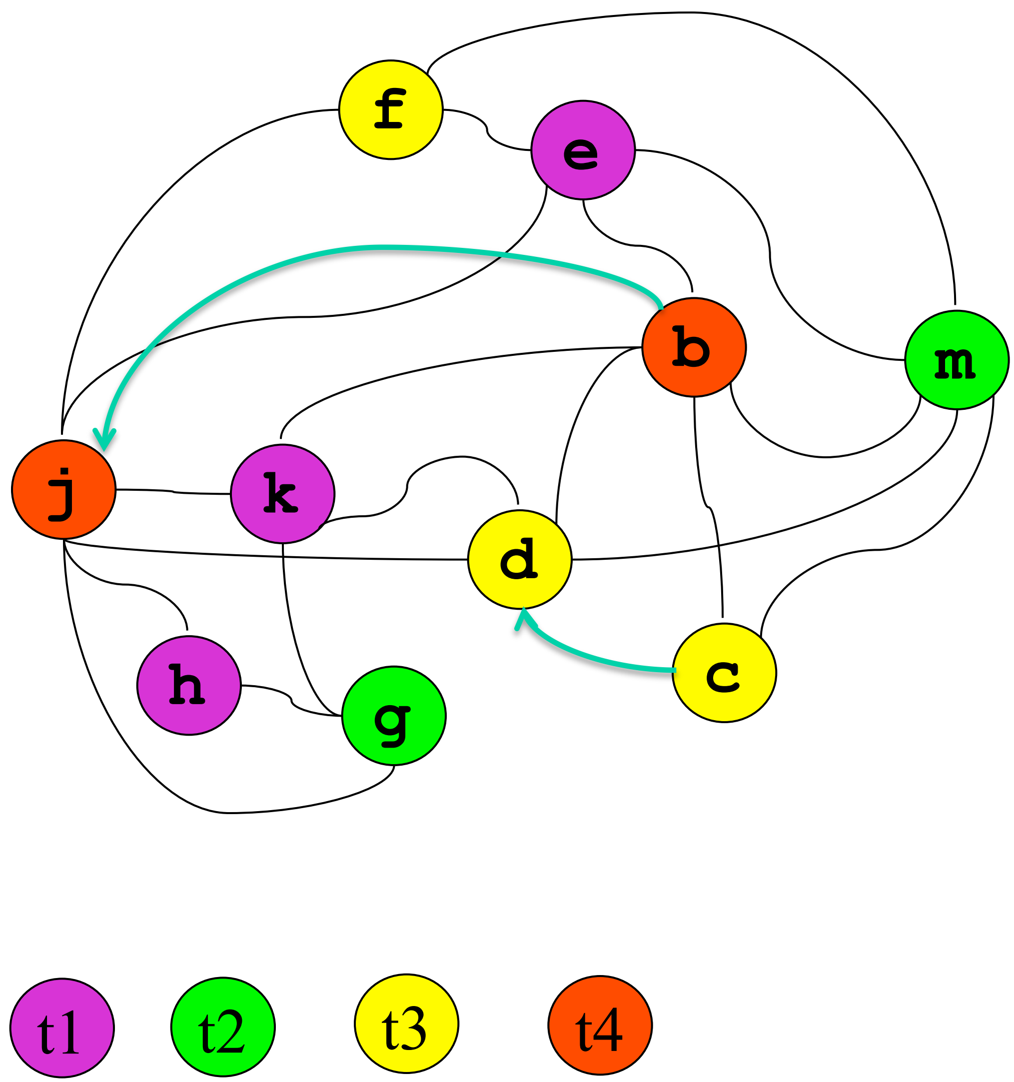
Now Rewrite Code…

```
t2 := mem[t4+12]
t1 := t1 - 1
t3 := t2 * t1
t1 := mem[t4+8]
t2 := mem[t4+16]
t4 := mem[t3]
t3 := t3 + 8
t3 := t3
t1 := t1 + 4
t4 := t4
```



…and simplify moves

```
t2 := mem[t4+12]
t1 := t1 - 1
t3 := t2 * t1
t1 := mem[t4+8]
t2 := mem[t4+16]
t4 := mem[f]
t3 := t1 + 8
t1 := t1 + 4
```



Some Practicalities

- The IL often includes machine registers
  - e.g., FP, $a0-a3, $v0-v1
  - allows us to expose issues of calling convention over which we don't have control.
- We can treat the machine registers as *pre-colored* temps.
  - Their assignment to a physical register is already determined.
  - But note that select & coalesce may put a different temp in the same physical register, as long as it doesn't interfere.

Using Physical Registers

Within a procedure:

- move arguments from $a0-a3 (and Mem[$fp+offset]) into fresh temps, move results into $v0-$v1.
- manipulate the temps directly within the procedure body instead of the physical registers, giving the register allocation maximum freedom in assignment, and minimizing the lifetimes of pre-colored nodes.
- register allocation will hopefully coalesce the argument registers with the temps, eliminating the moves.
- ideally, if we end up spilling a temp corresponding to an argument, we should write it back in the already reserved space on the stack…

Note:

- We cannot *simplify* a pre-colored node:
  - removing a node during simplification happens because we expect to be able to assign it *any* color that doesn't conflict with the neighbors.
  - but we don't have a choice for pre-colored nodes.
  - Trick: treat physical nodes as having "infinite degree" in interference graph.
- Similarly, we cannot *spill* a pre-colored node.

Callee-Saves Registers

- Callee-Saves register r:
  - it's "defined" upon entry to the procedure
  - it's "used" upon exit from the procedure.
  - trick: move it into a fresh temp
  - ideally, the temp will be coalesced with the callee-saves register (getting rid of the move.)
  - otherwise, we have the freedom to spill the temp.

Caller Saves Registers

- Want to assign a temp to a caller-saves register only when it's not live across a function call (for then we have to save/ restore it.)
- So treat a function call as "defining" all of the caller-saves registers.
  - (callee might move values into them.)
  - now any temps that are live across the call will interfere, and assignment will try to find different registers to assign the temps.

Example (p. 238 in book)

We're compiling the following C procedure:

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e > 0);
  return d;
}
```

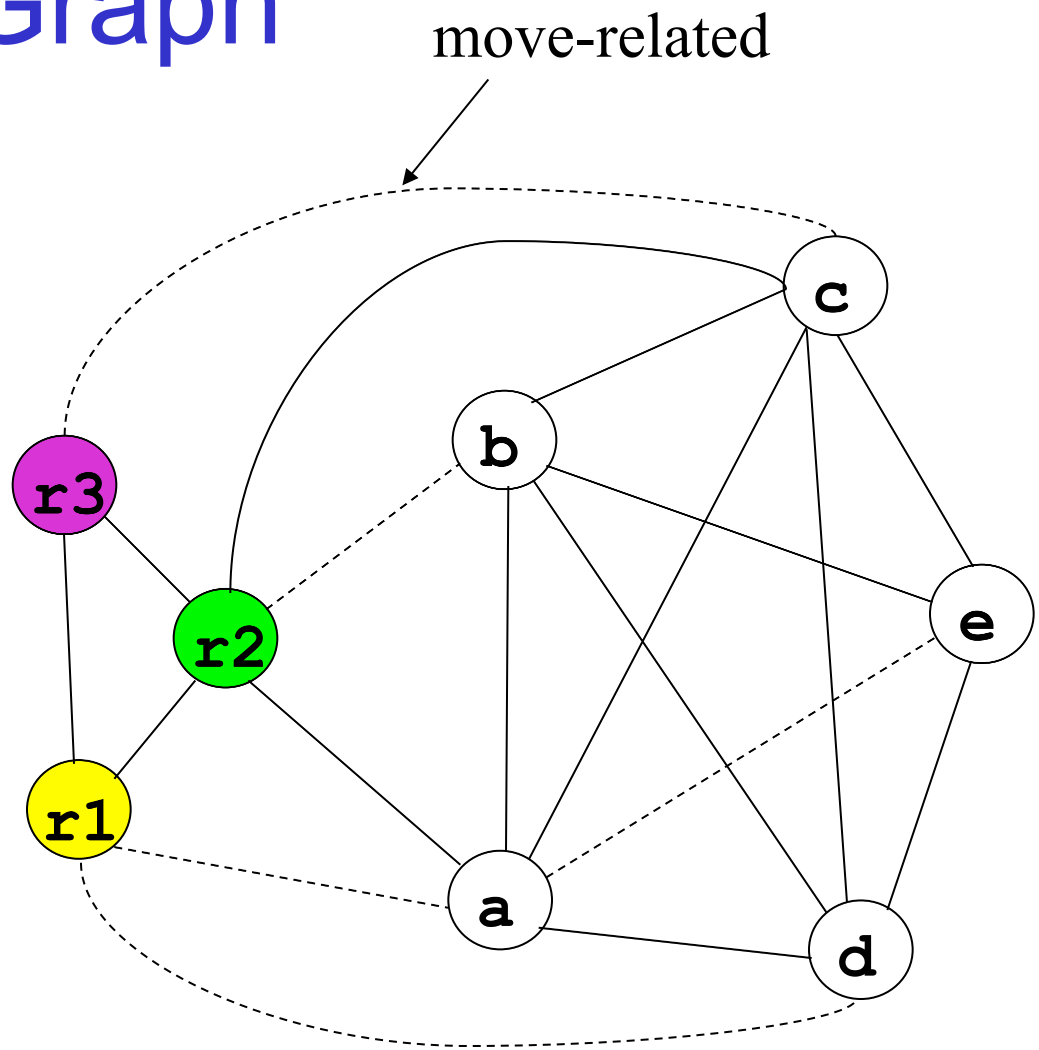
Assume we have a target machine with 3 registers, where r1 and r2 are caller-saves and r3 is callee-saves.

Generated CFG:

```
f:        c := $r3    ; preserve callee
          a := $r1    ; move arg1 into a
          b := $r2    ; move arg2 into b
          d := 0
          e := a
loop:     d := d + b
          e := e - 1
          if e > 0 loop else end
end:      r1 := d     ; return d
          r3 := c     ; restore callee
          return      ; $r3,$r1 live out
```

Interference Graph

```
f:    c := $r3
      a := $r1
      b := $r2
      d := 0
      e := a
L:    d := d + b
      e := e - 1
      if e > 0 L
         else E
E:    r1 := d
      r3 := c
      return
```



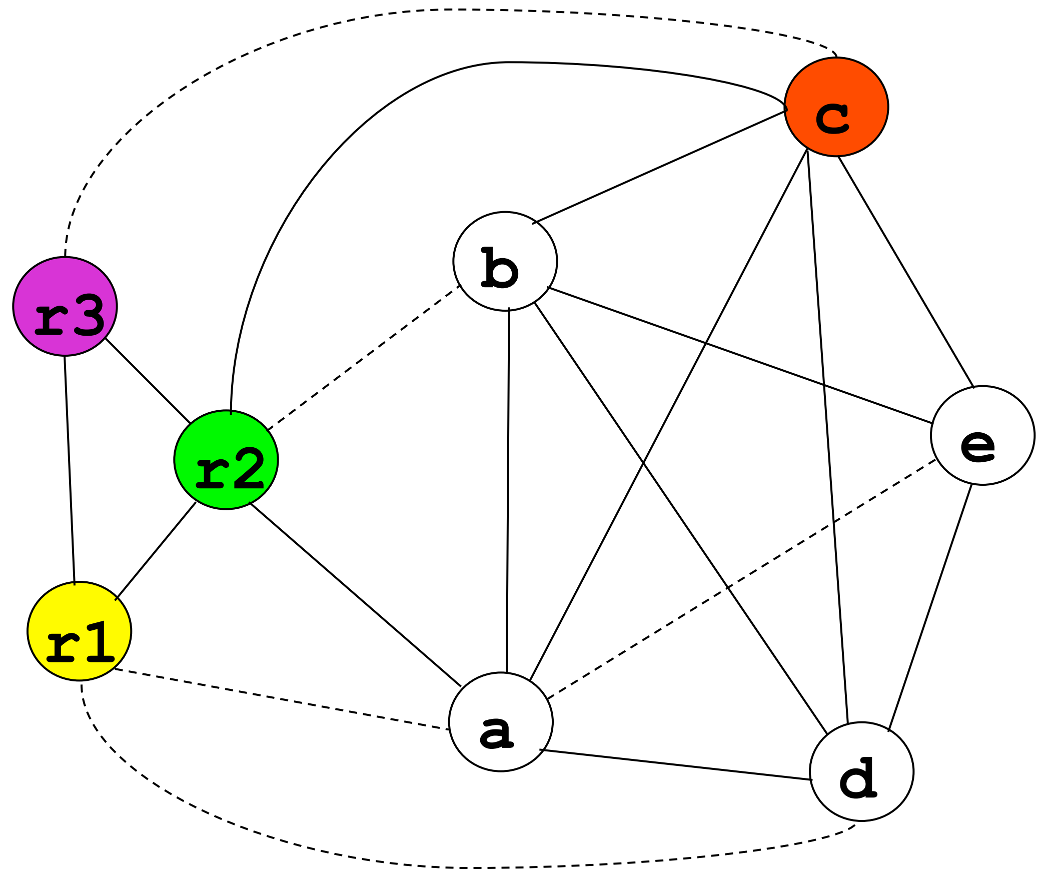
No simplify, freeze, or coalesce is possible…

Spilling:

Node **c** is a good candidate for spilling.

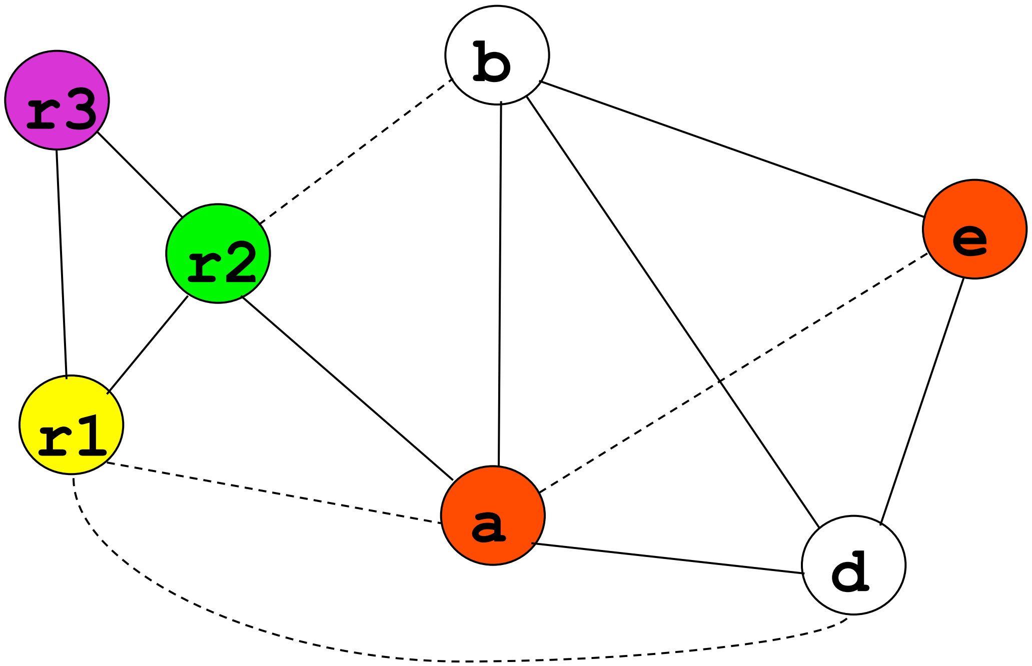
So push it as a potential spill.

Stack: sp(c)



After Spilling c:

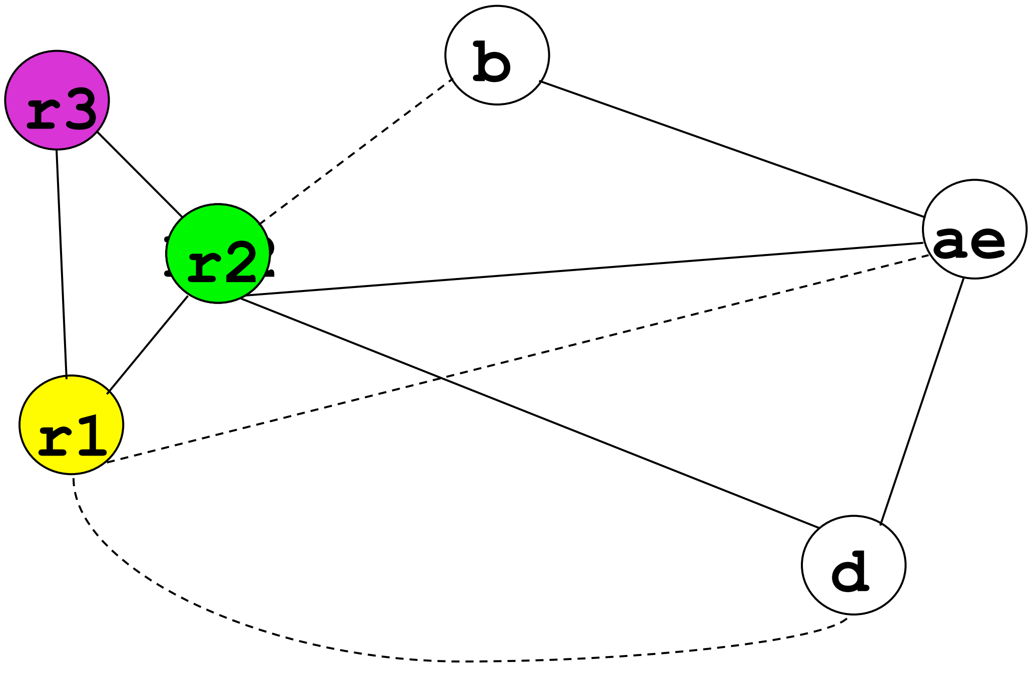
Now we can safely coalesce **a** & **e**.



Stack: sp(c)

After Coalescing a & e:

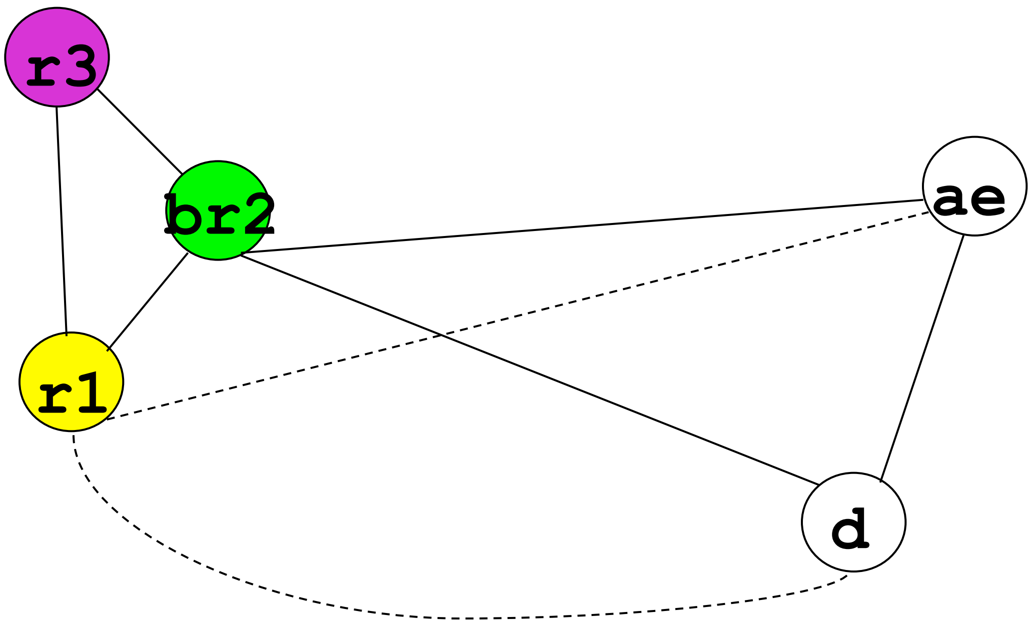
Now we can safely coalesce **`b`** & **`r2`**.



Stack: sp(c)

After Coalescing b & r2:

Now we can safely coalesce **r1** & **ae**.

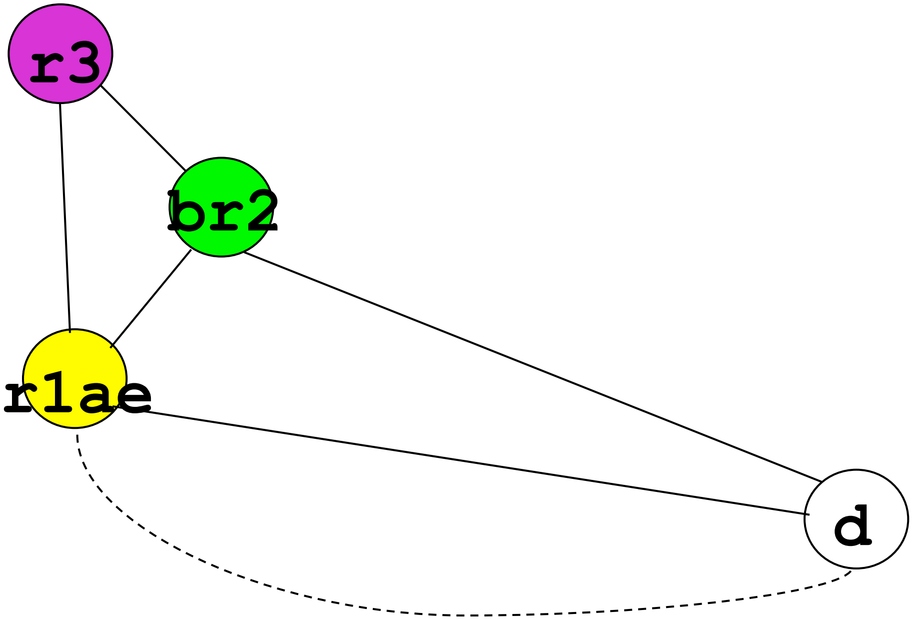


Stack: sp(c)

Constrained Nodes:

We *cannot* safely coalesce **`r1ae`** & **`d`** because they are *constrained.*

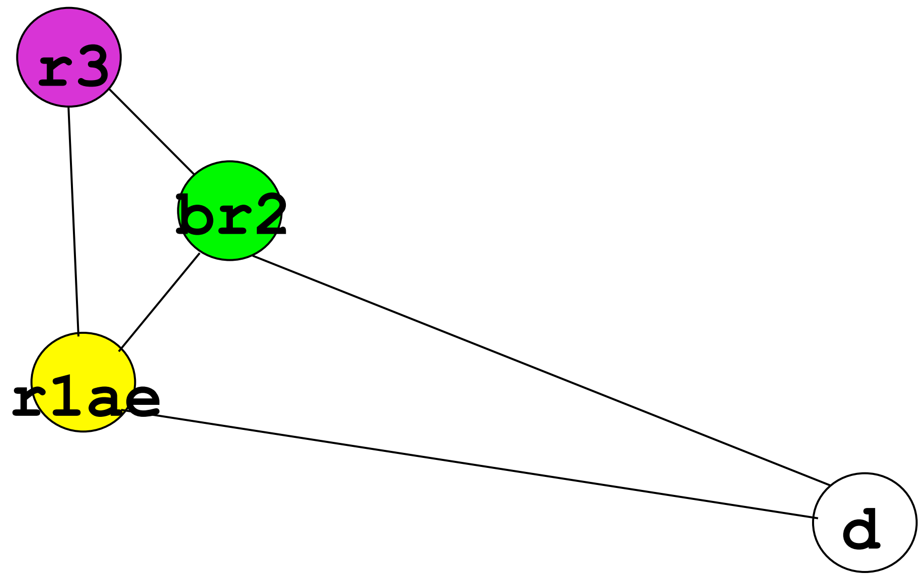
When we coalesce, and we have both a non-move edge and a move-edge, we can't drop the non-move edge…



Simplify:

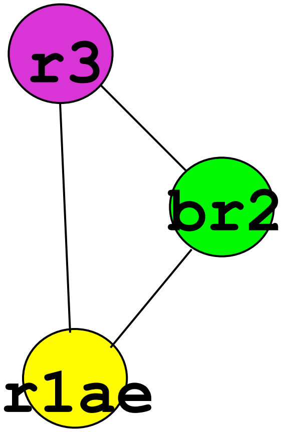
At this point, we can simplify **d**.

Stack: sp(c)



Start Selecting:

Now we only have pre-colored nodes left…

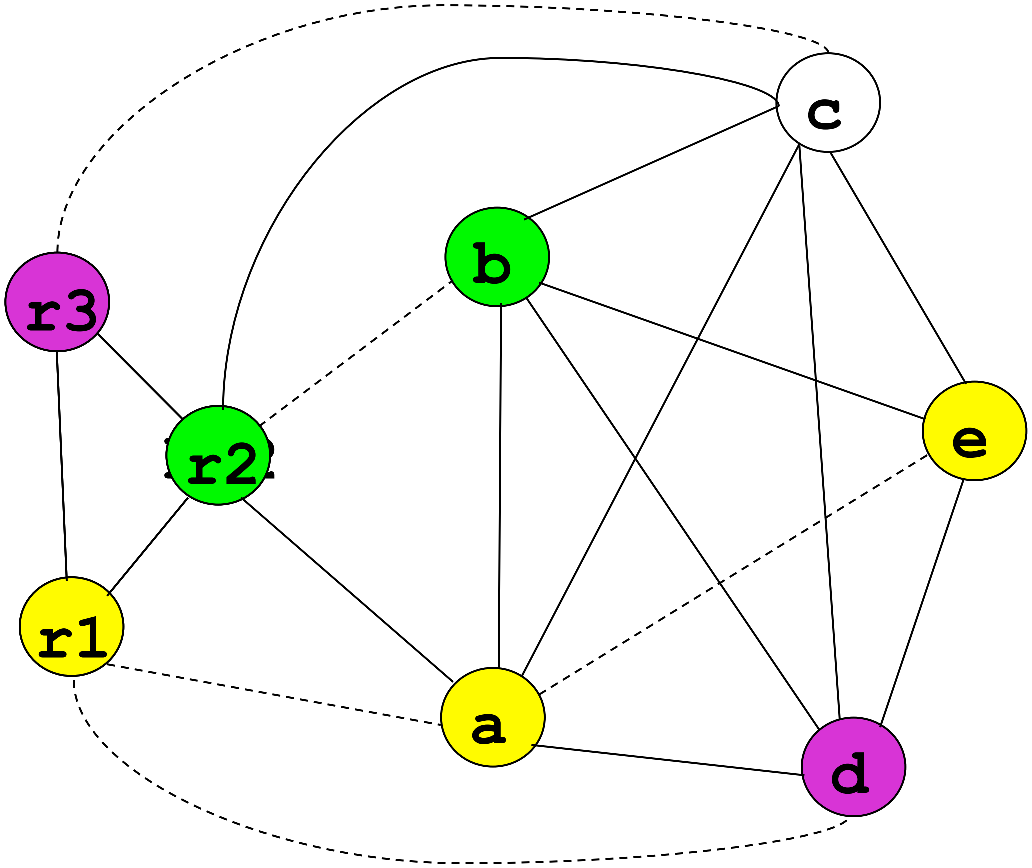


Stack: sp(c), d

Start Selecting:

We pop d and assign it a color.

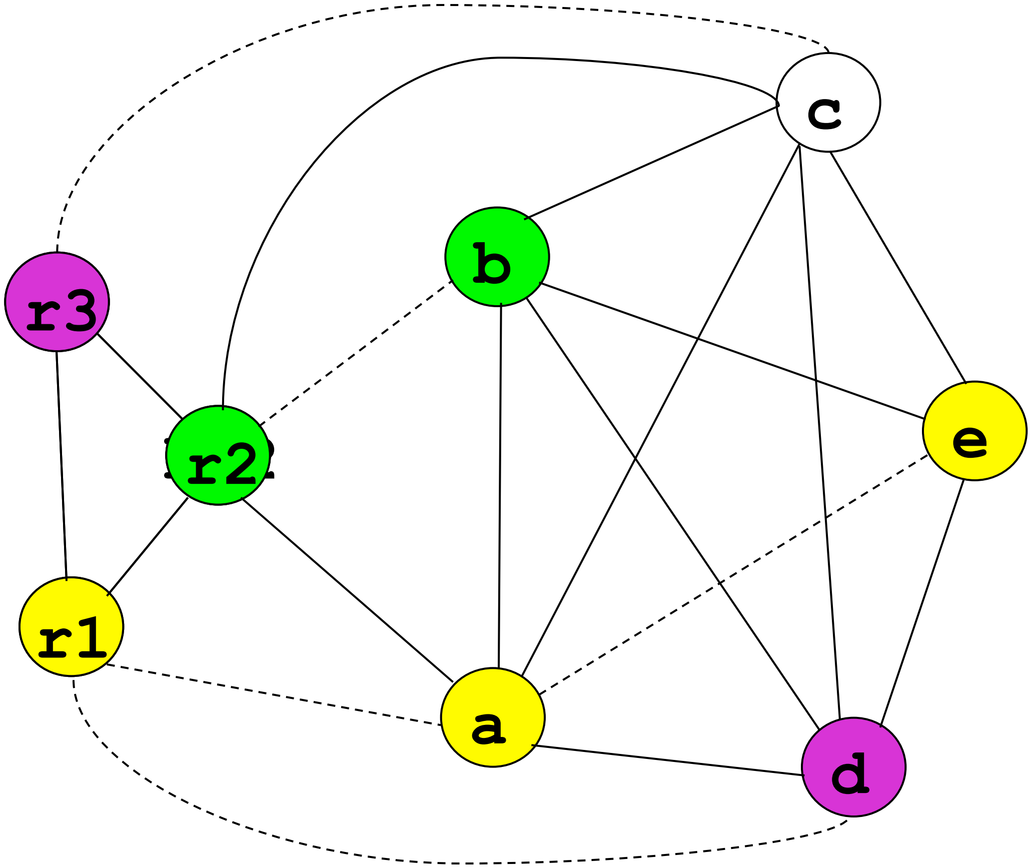
Stack: sp(c)



Optimism Failed

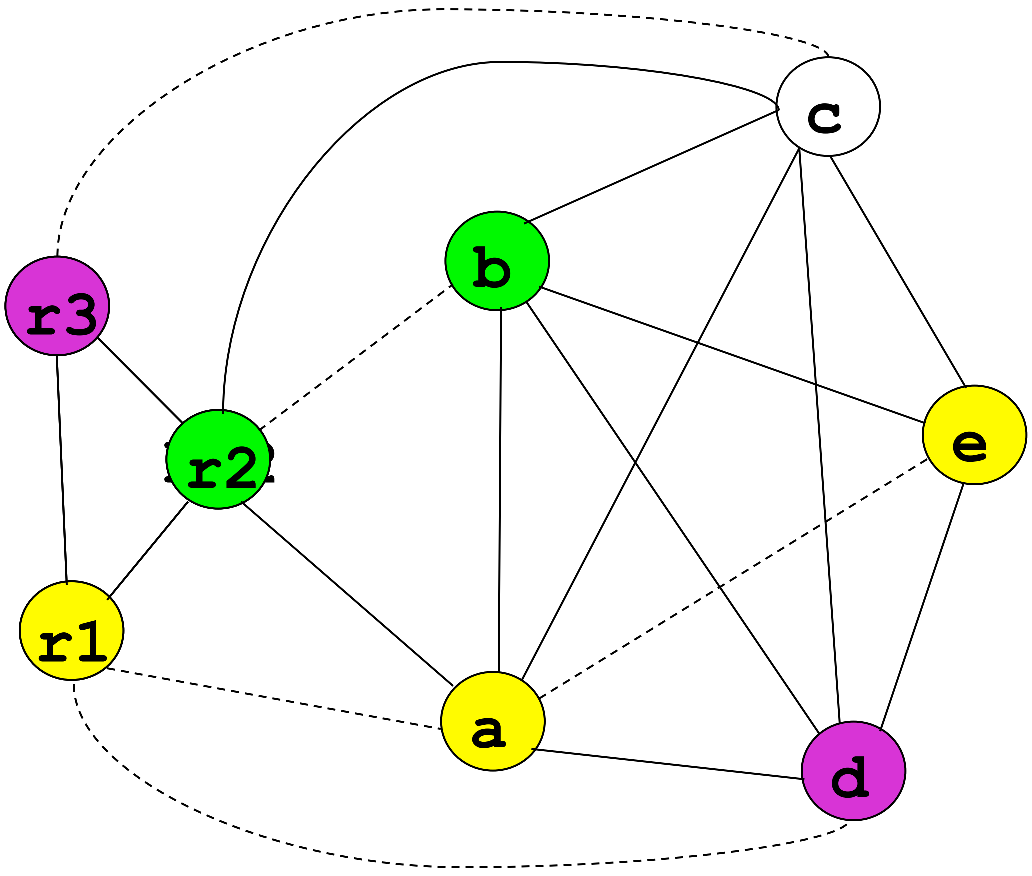
We pop c but find out that we must do an actual spill.

Stack: sp(c)



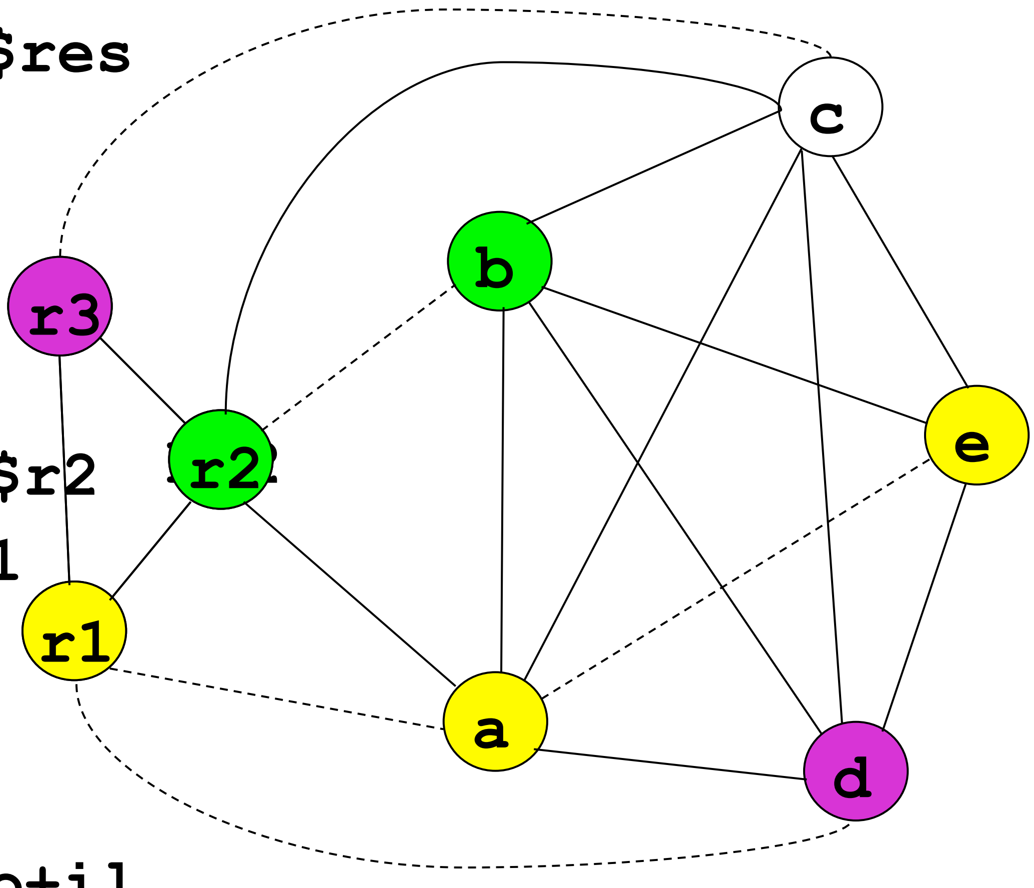
Rewrite Code

```
f:    c := $r3
      a := $r1
      b := $r2
      d := 0
      e := a
L:    d := d + b
      e := e - 1
      if e > 0 L
         else E
E:    r1 := d
      r3 := c
      return
```



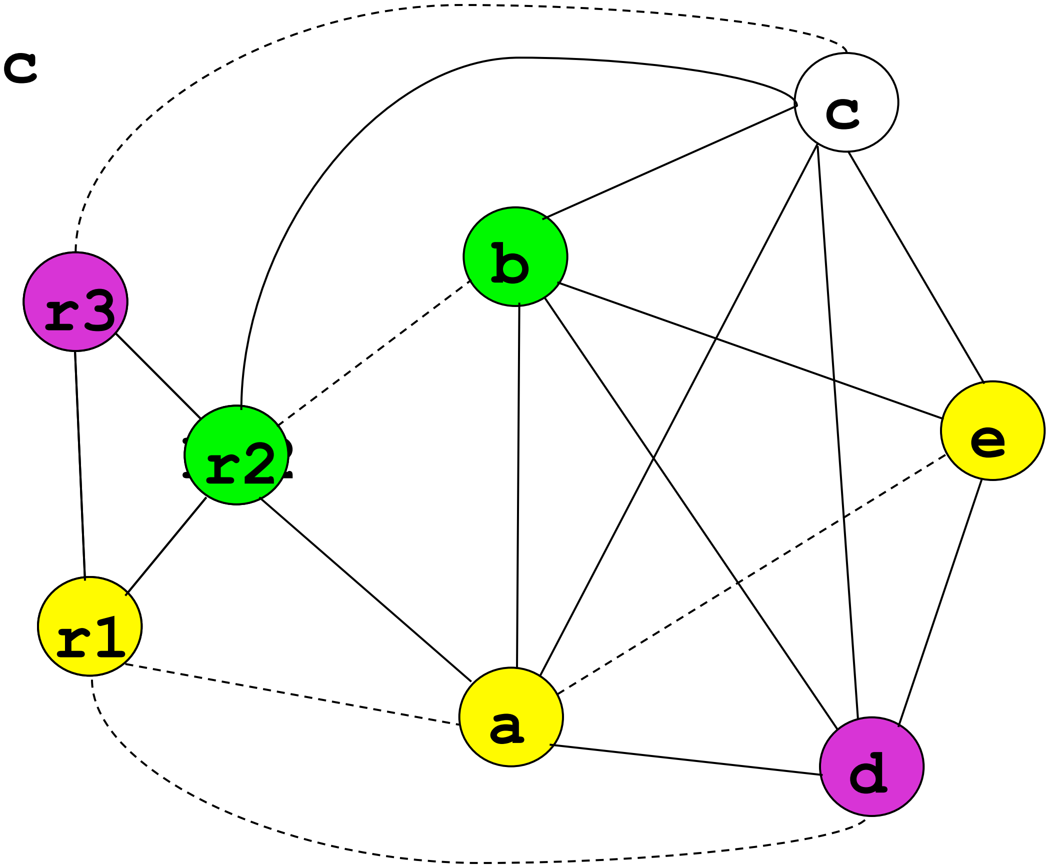
Rewrite Code

```
f:    $res := $r3
      Mem[fp+i] := $res
      $r1 := $r1
      $r2 := $r2
      $r3 := 0
      $r1 := $r1
L:    $r3 := $r3 + $r2
      $r1 := $r1 - 1
      if $r1 > 0 L
         else E
E:    $r1 := $r3
      $res := Mem[fp+i]
      $r3 := $res
      return
```



Alternatively:

```
f:    c := $r3
      Mem[fp+i] := c
      a := $r1
      b := $r2
      d := 0
      e := a
L:    d := d + b
      e := e - 1
      if e > 0 L
         else E
E:    r1 := d
      f := Mem[fp+i]
      r3 := f
      return
```



Get Rid of Stupid Moves:

```
f:    $res := $r3
      Mem[fp+i] := $res
      $r3 := 0
L:    $r3 := $r3 + $r2
      $r1 := $r1 - 1
      if $r1 > 0 L else E
E:    $r1 := $r3
      $res := Mem[fp+i]
      $r3 := $res
      return
```