HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 22:
Register Allocation ctd.

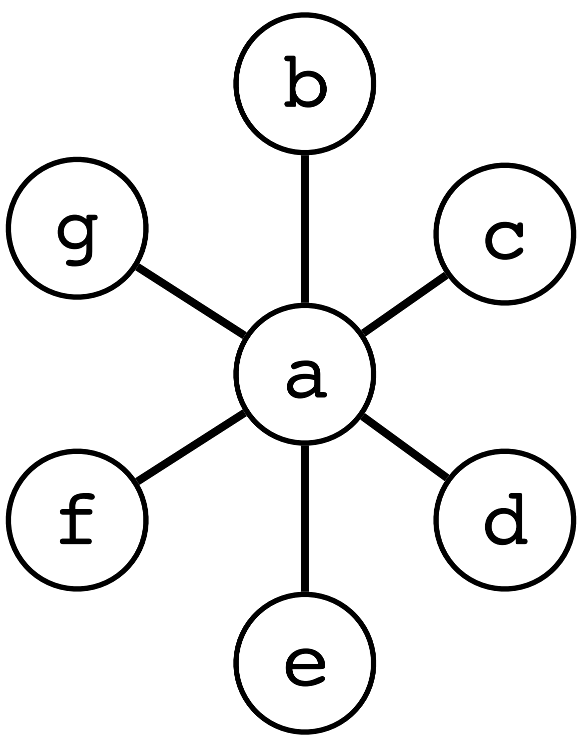
Stephen Chong

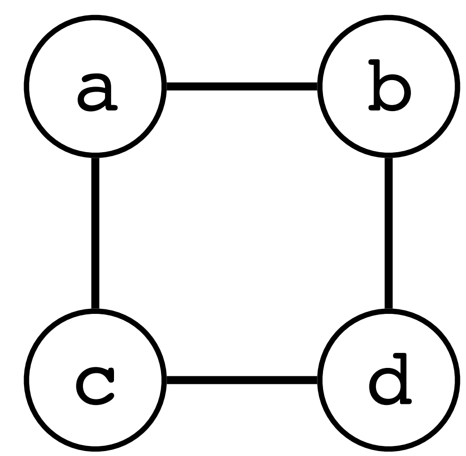
https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

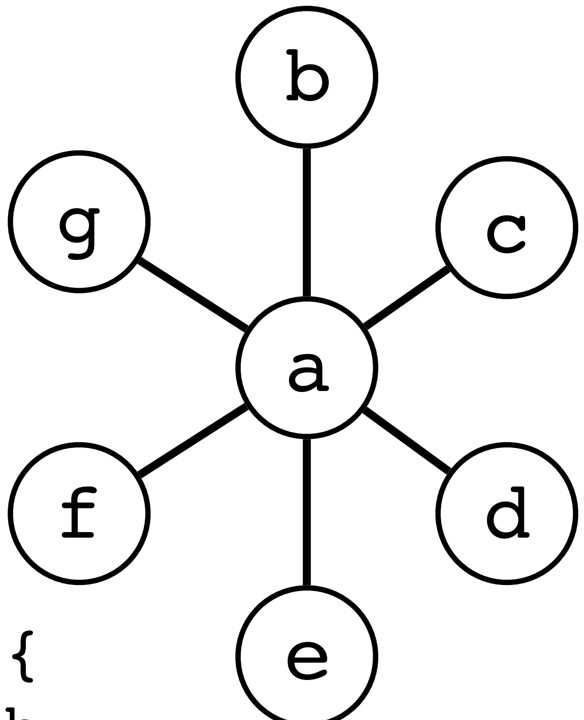
Pre-class Puzzle

- Can you write programs that have the following interference graphs?

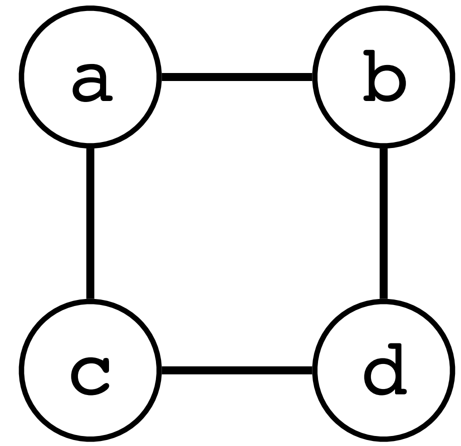




Pre-class Puzzle




```
f(a,b) {
  c := b;
  d := c;
  e := d;
  f := e;
  g := f;
  a := a+g;
  return a;
}
```




```
g(a) {
     if a then goto L1 else goto L2
L1:  c := a;
     a := a + c;
     d := a;
     d := d + c;
     goto L3
L2:  b := a;
     a := a + b;
     d := a;
     d := d + b;
L3:  return d
}
```

Announcements

- HW5: Oat v.2 out
  - Due Tue Nov 19
- HW6: Optimization and Data Analysis
  - Due: Tue Dec 3

Today

- Register allocation ctd
  - Graph coloring by simplification
  - Coalescing
  - Coloring with coalescing
    - Pre-colored nodes to handle callee-save, caller-save, and special purpose registers

Spilling

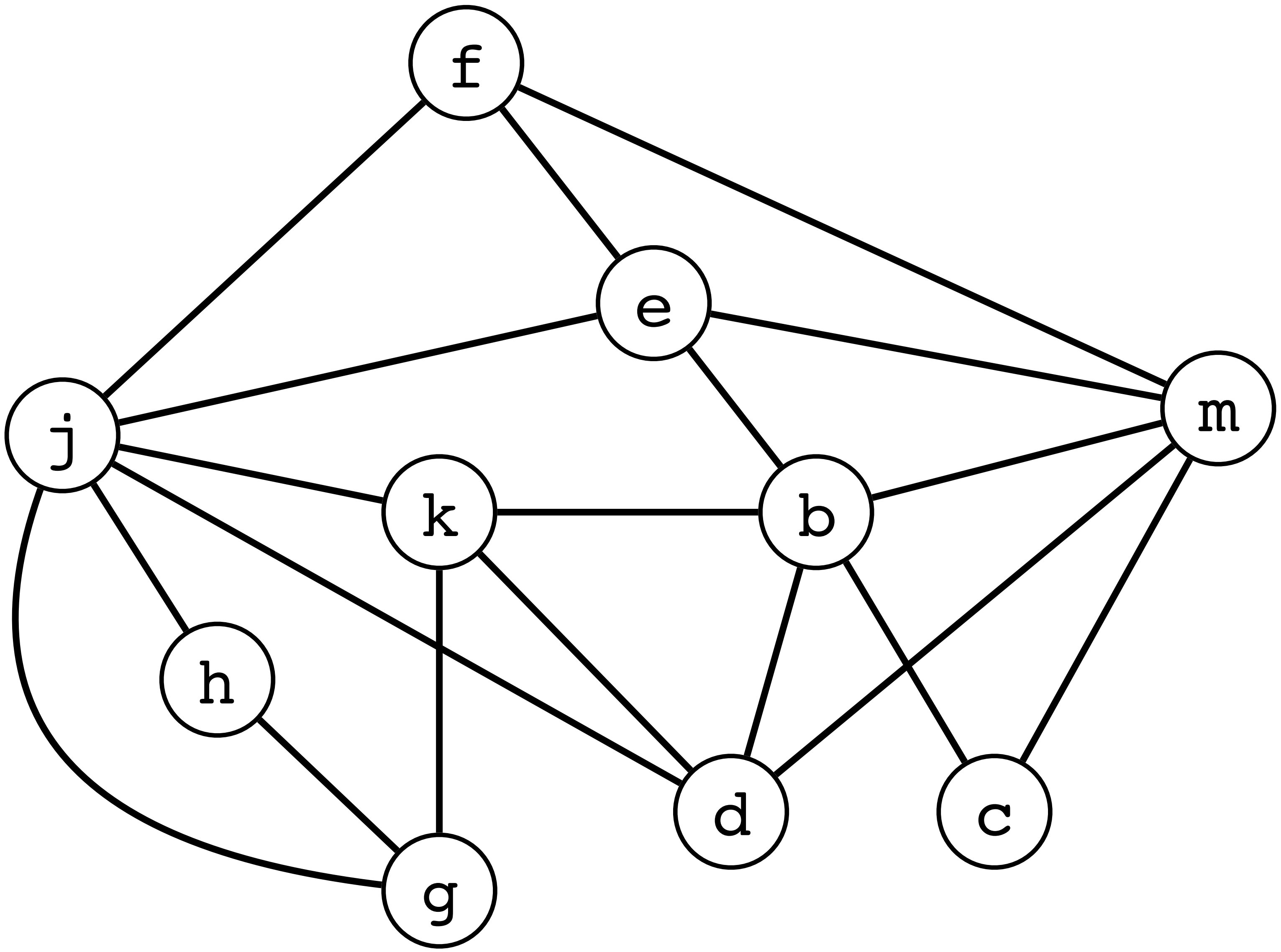
- The previous example worked out nicely!
- Always had nodes with degree $<k$
- Let's try again, but now with only 3 registers...

Example

From Appel

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d,j,k}
```

Interference graph

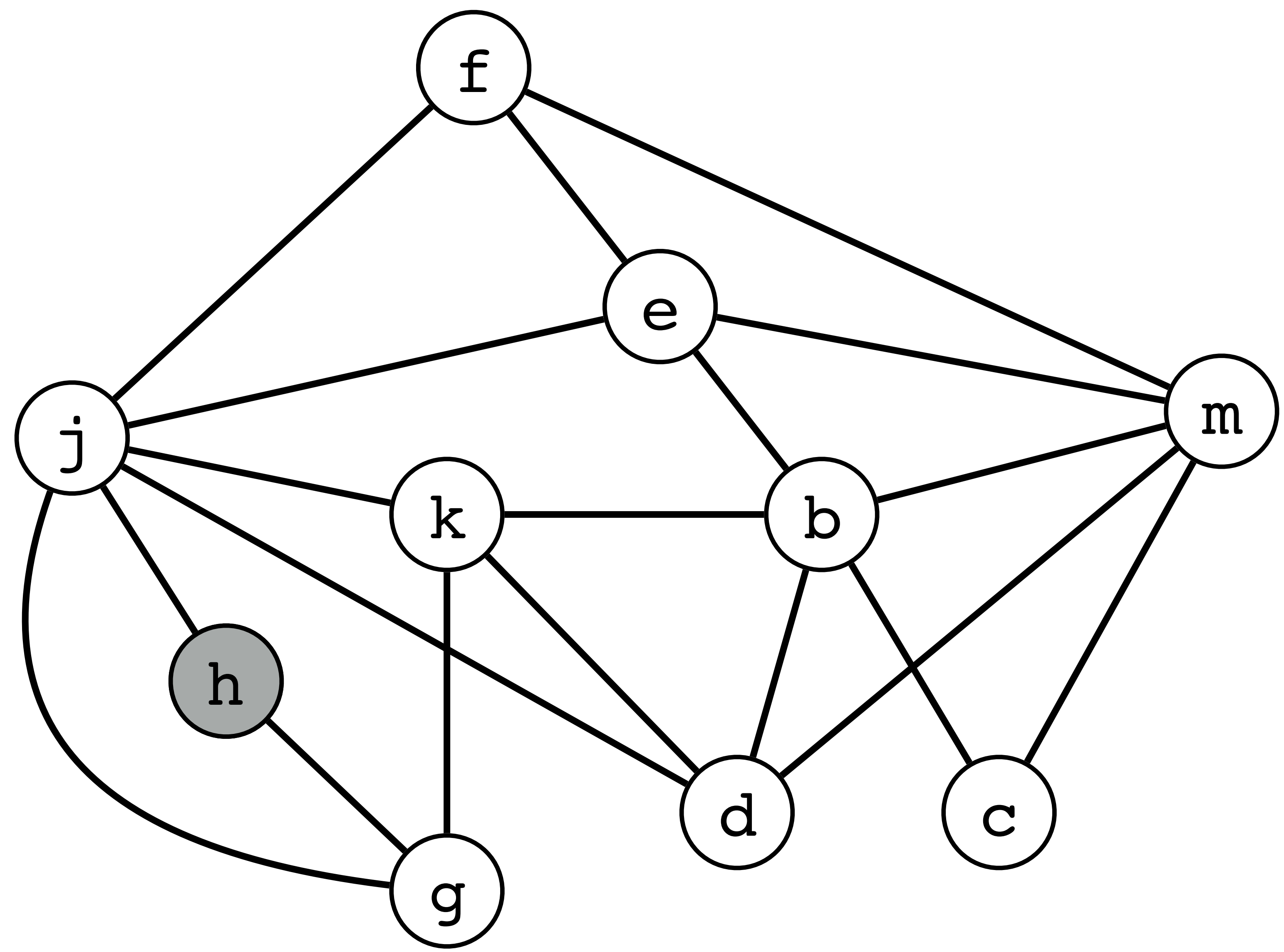


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`

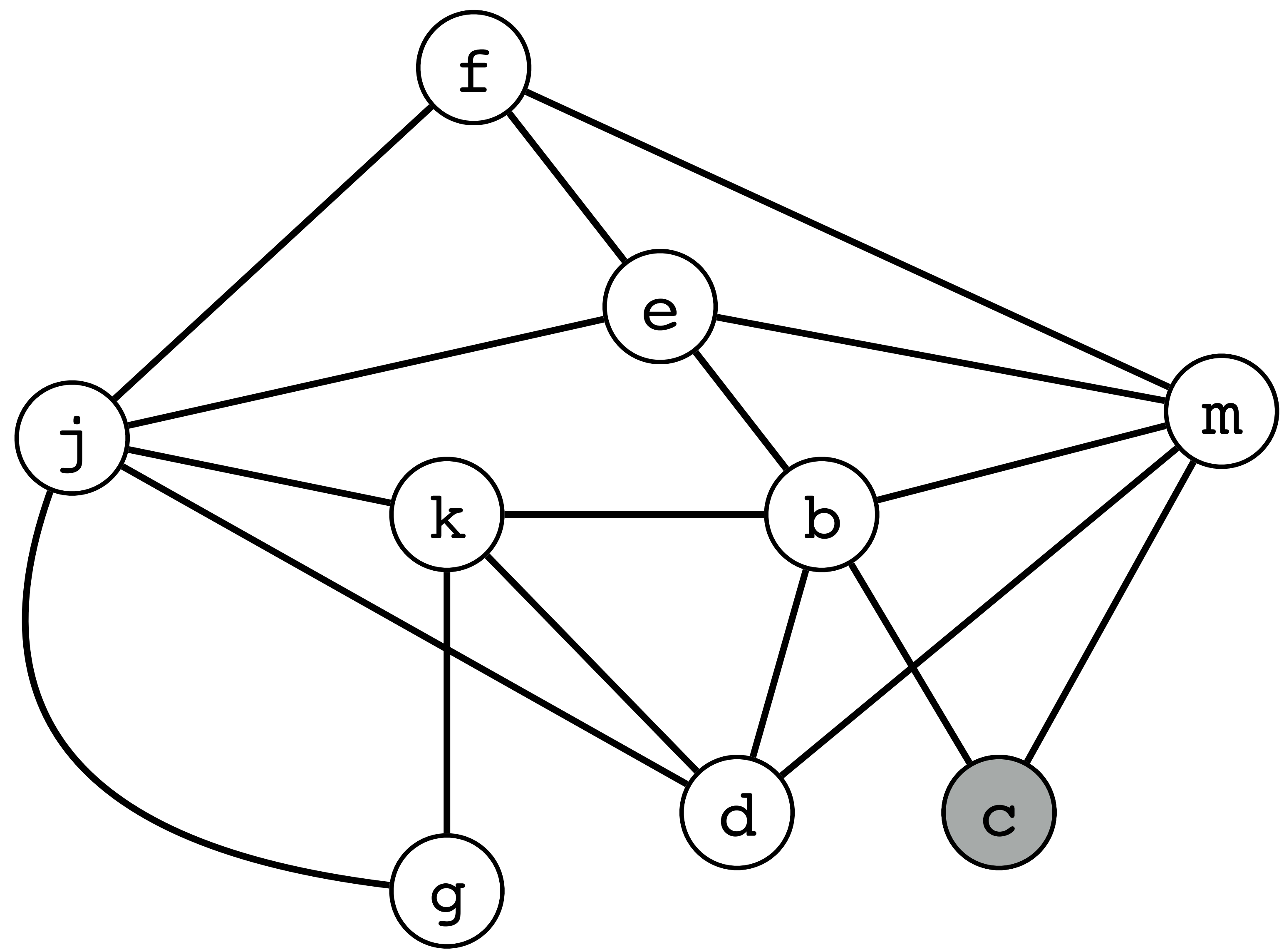


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c

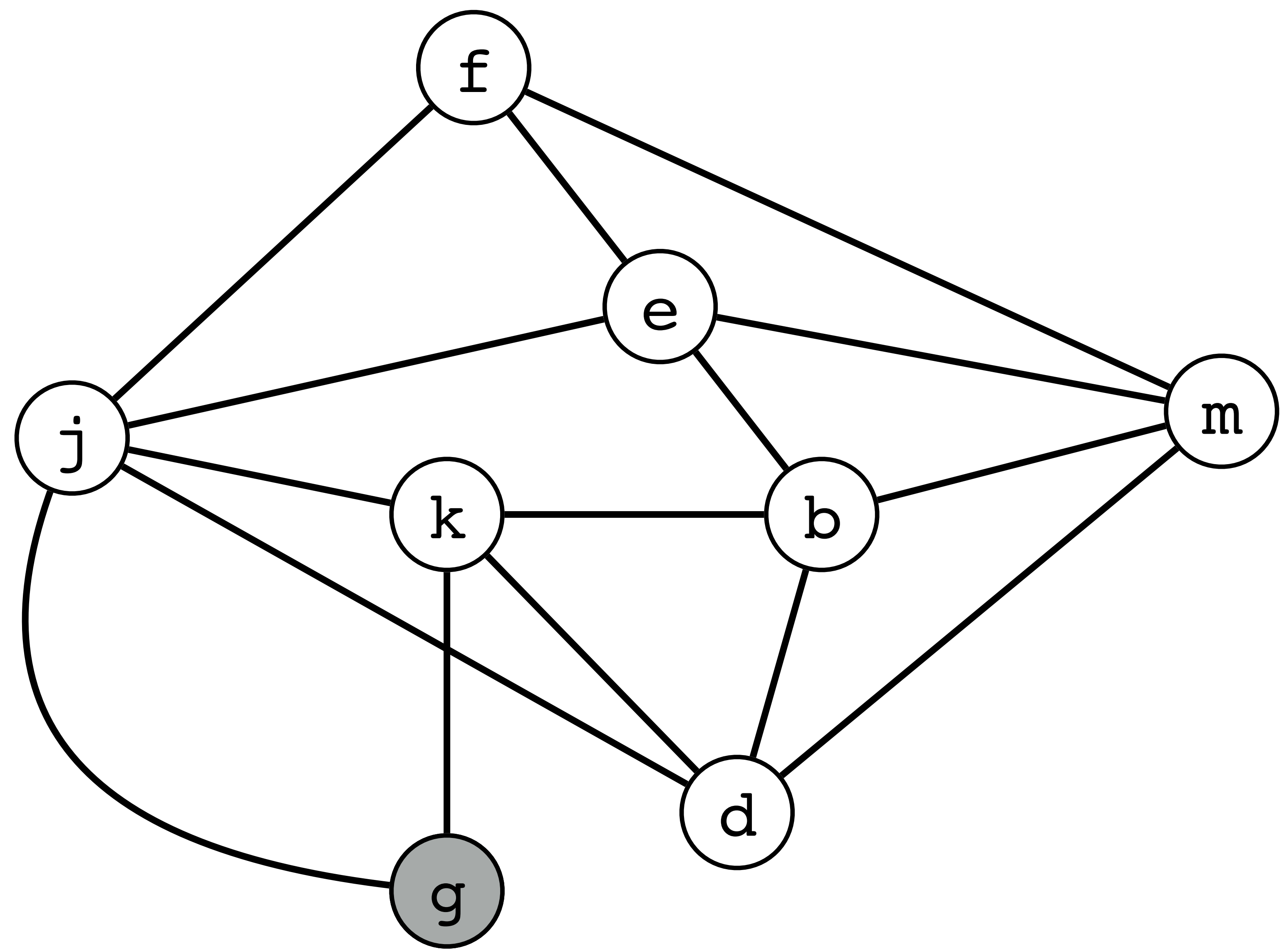


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g

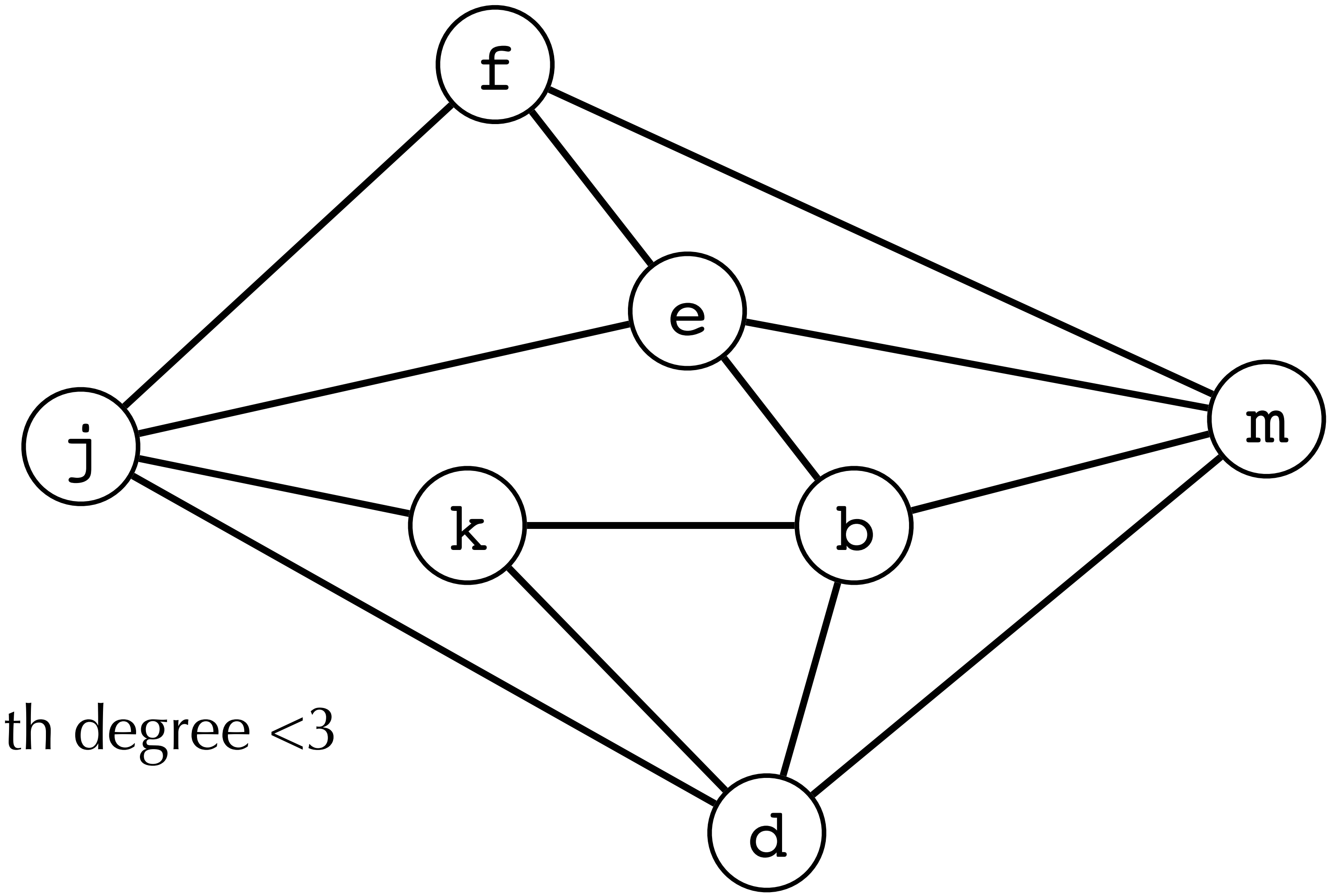


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g

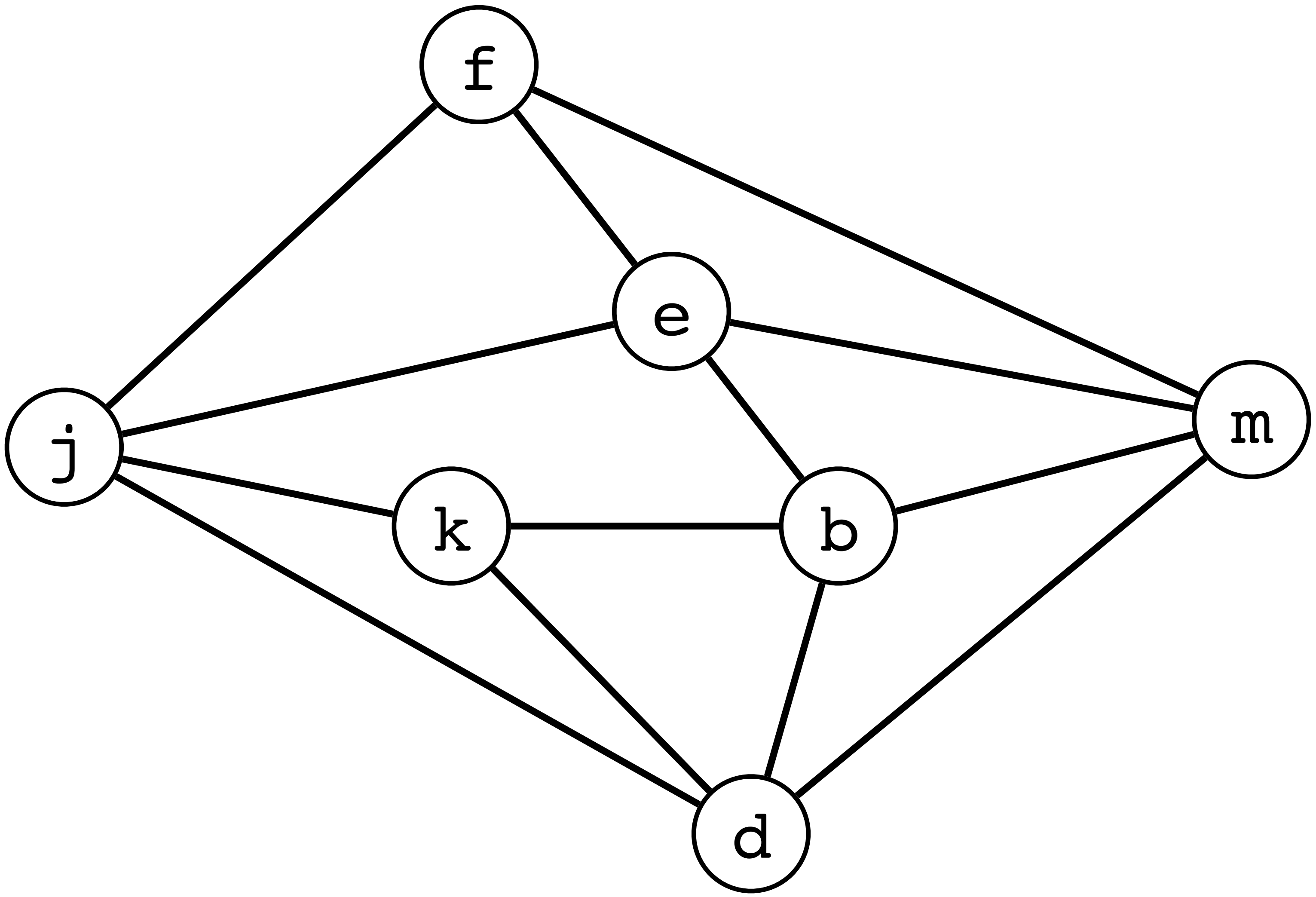


Now we are stuck! No nodes with degree <3

Pick a node to potentially spill

Which Node to Spill?

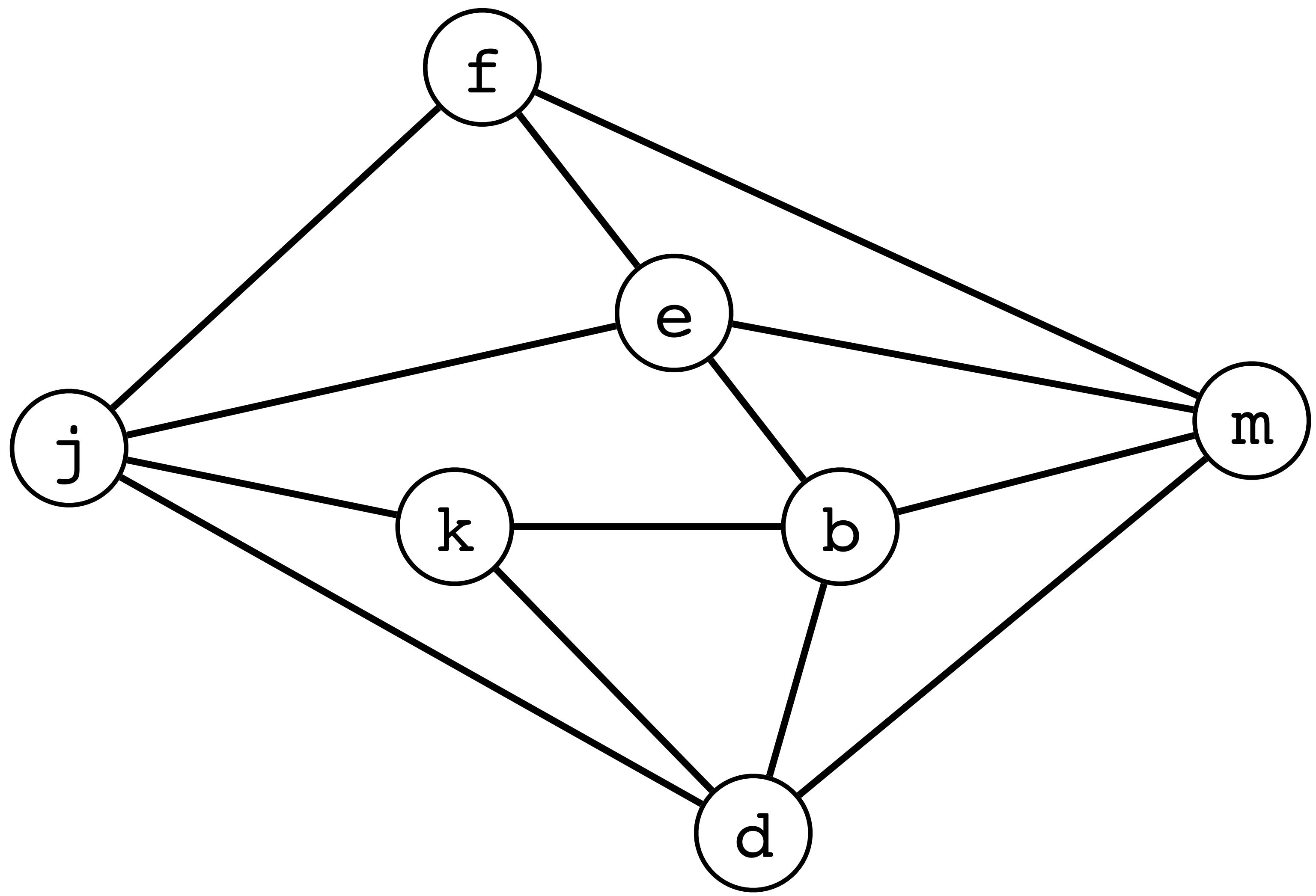
- Want to pick a node (i.e., temp variable) that will make it likely we'll be able to *k* color graph
  - High degree (≈ live at many program points)
  - Not used/defined very often (so we don't need to access stack very often)
- E.g., compute **spill priority** of node



$$\frac{\textit{Uses+defs outside loop} + \textit{Uses+defs in loop} \times 10}{\textit{degree of node}}$$

Which Node to Spill?

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d,j,k}
```



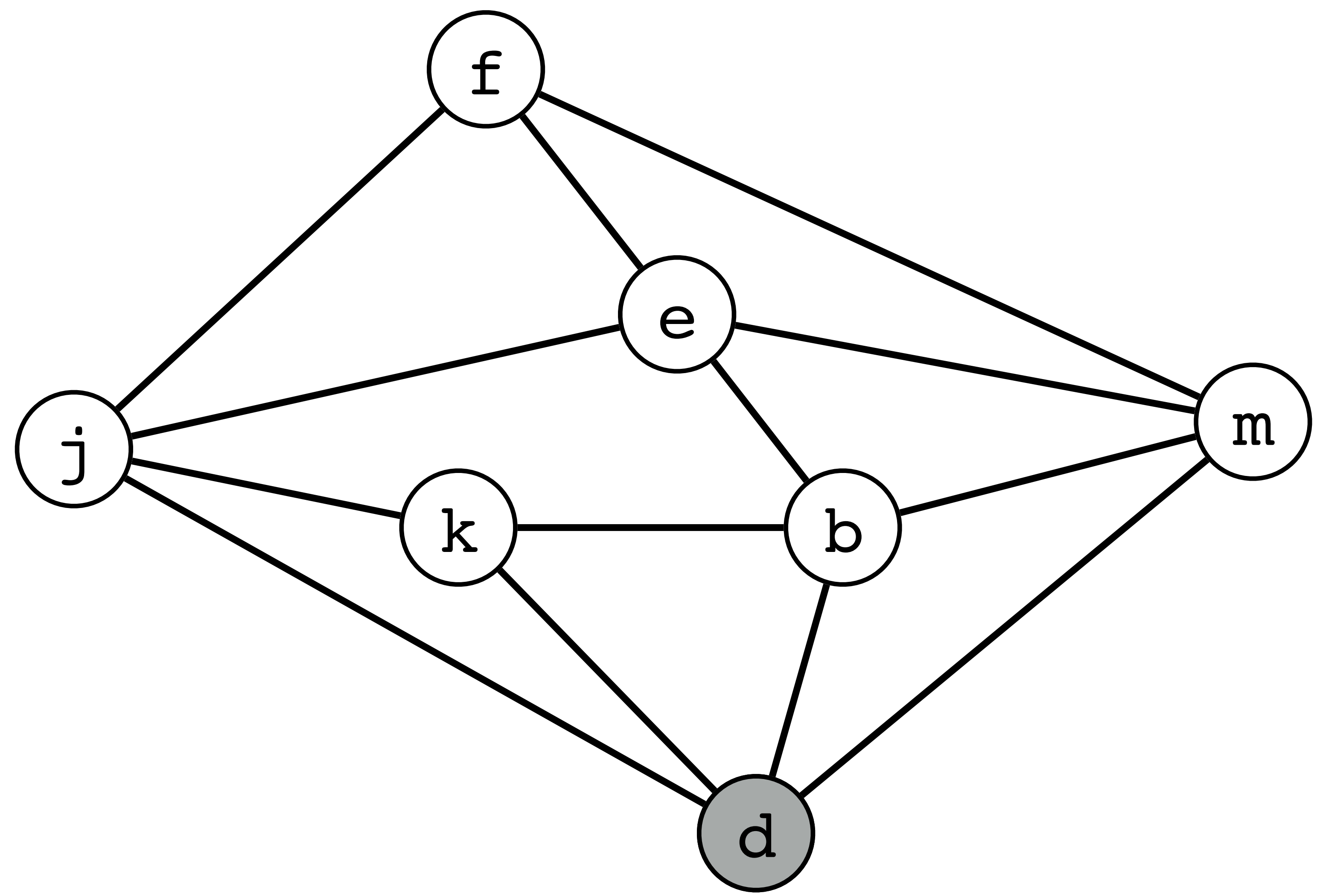
$$\text{Spill priority} = \frac{\textit{Uses+defs outside loop} + \textit{Uses+defs in loop} \times 10}{\textit{degree of node}}$$

Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g
d *spill?*



Pick a node with small spill priority degree to potentially spill

Simplification (3 registers)

Choose any node with degree <3

Stack:

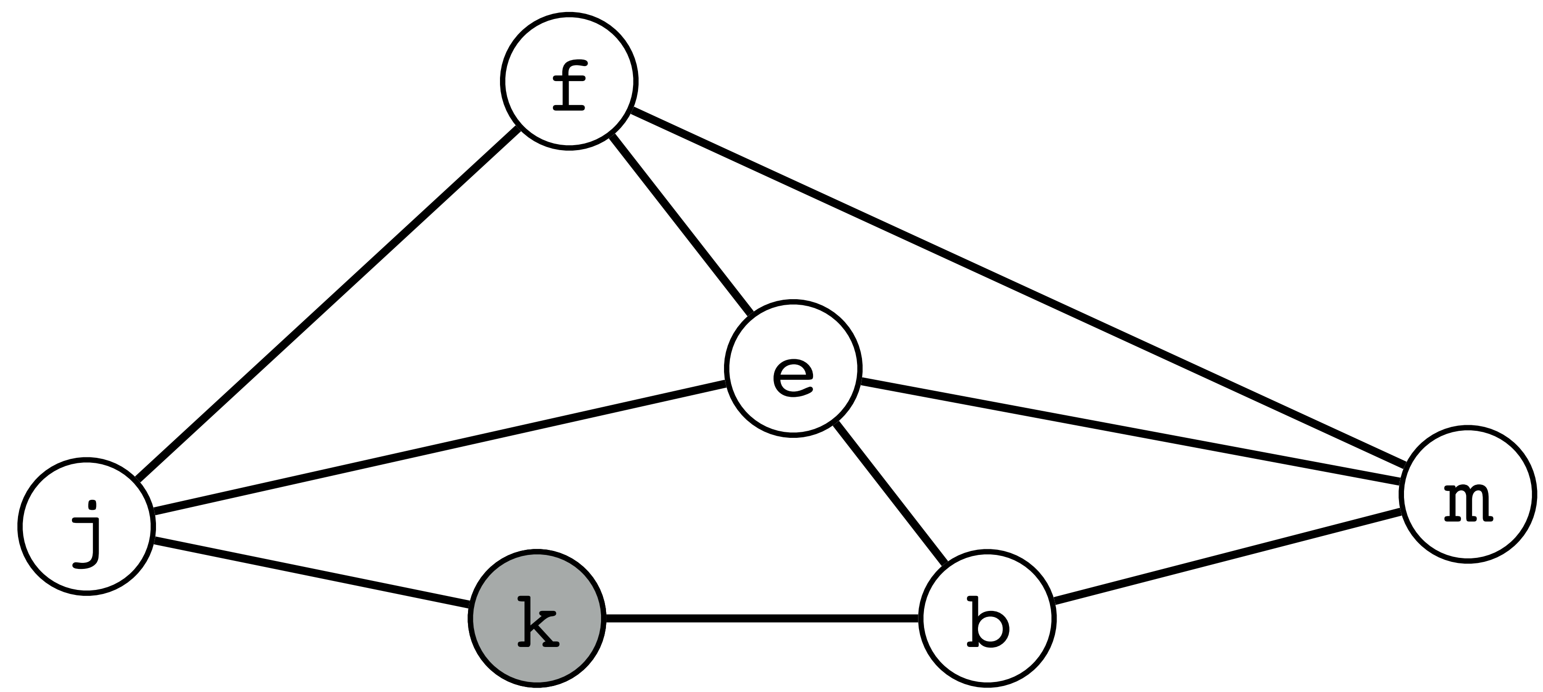
`h`

`c`

`g`

`d` *spill?*

`k`

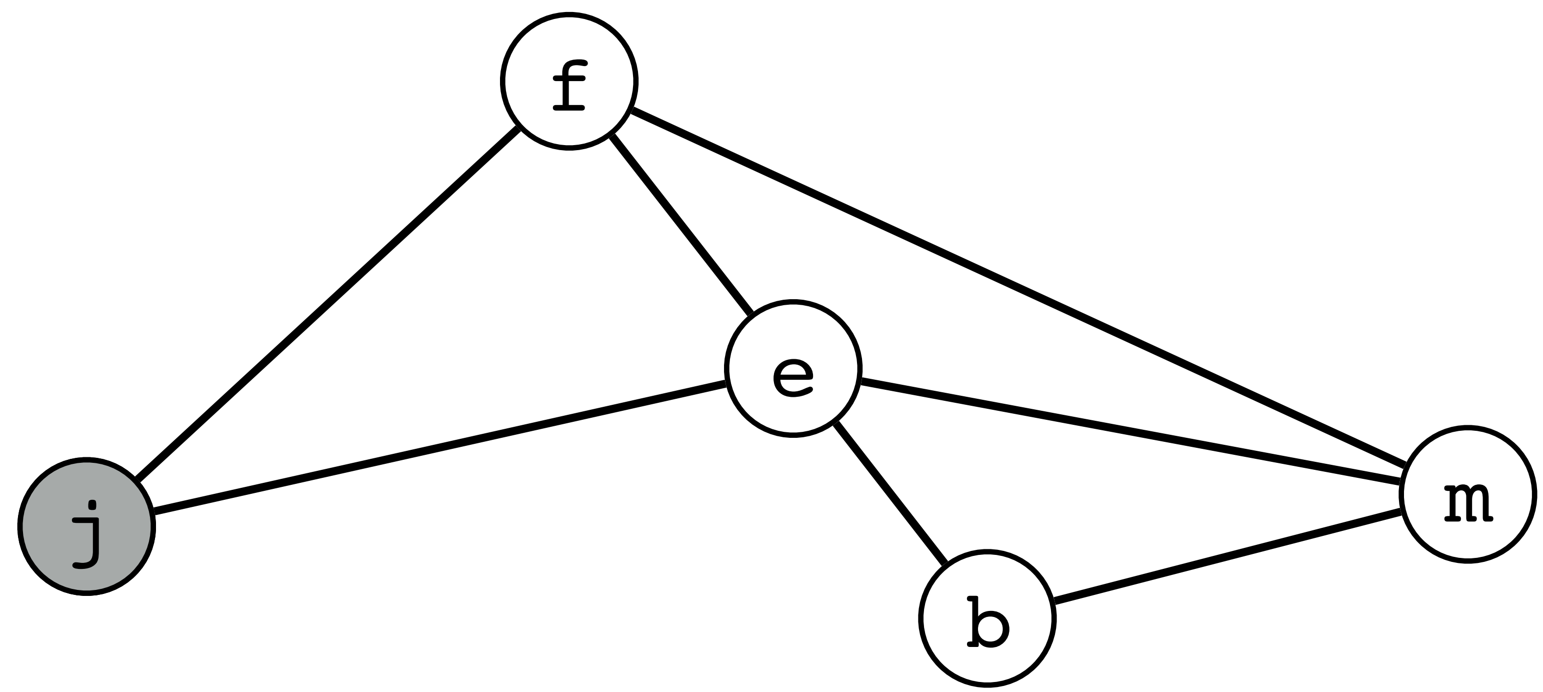


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`
`g`
`d` *spill?*
`k`
`j`

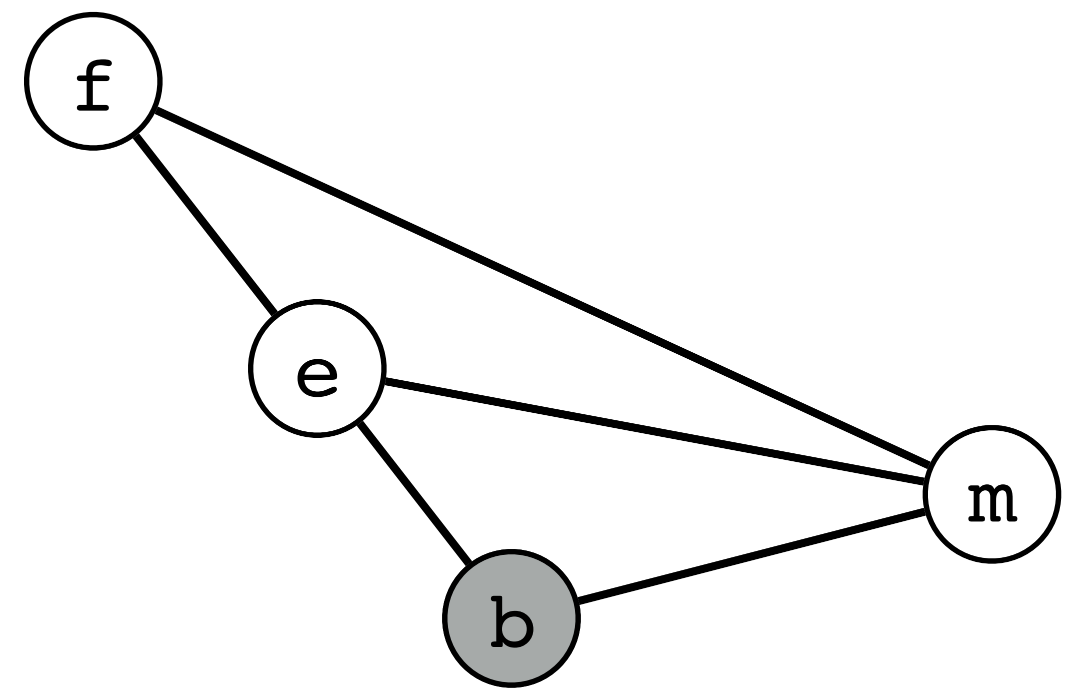


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g
d *spill?*
k
j
b



Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`

`c`

`g`

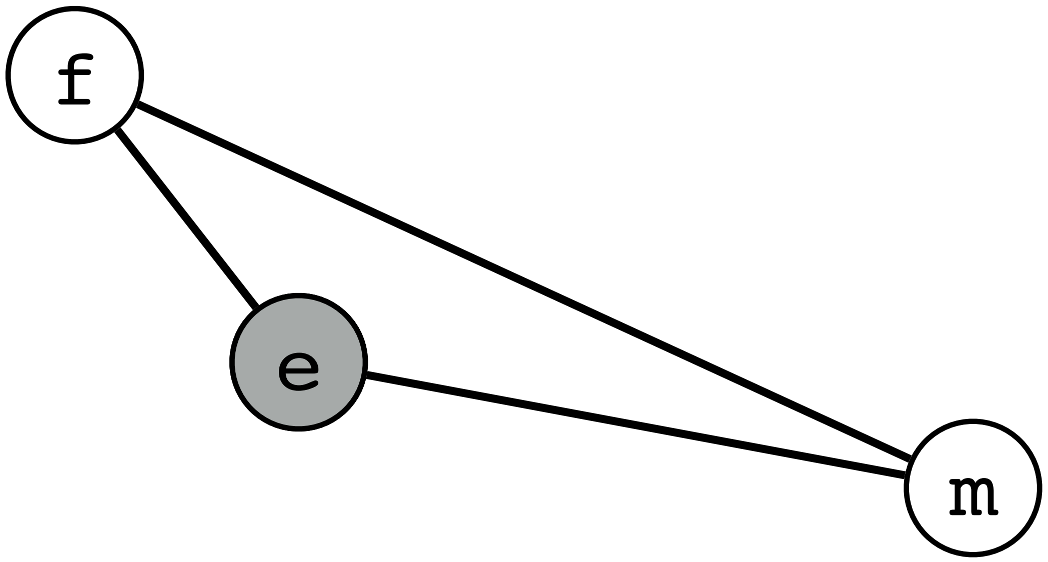
`d` *spill?*

`k`

`j`

`b`

`e`

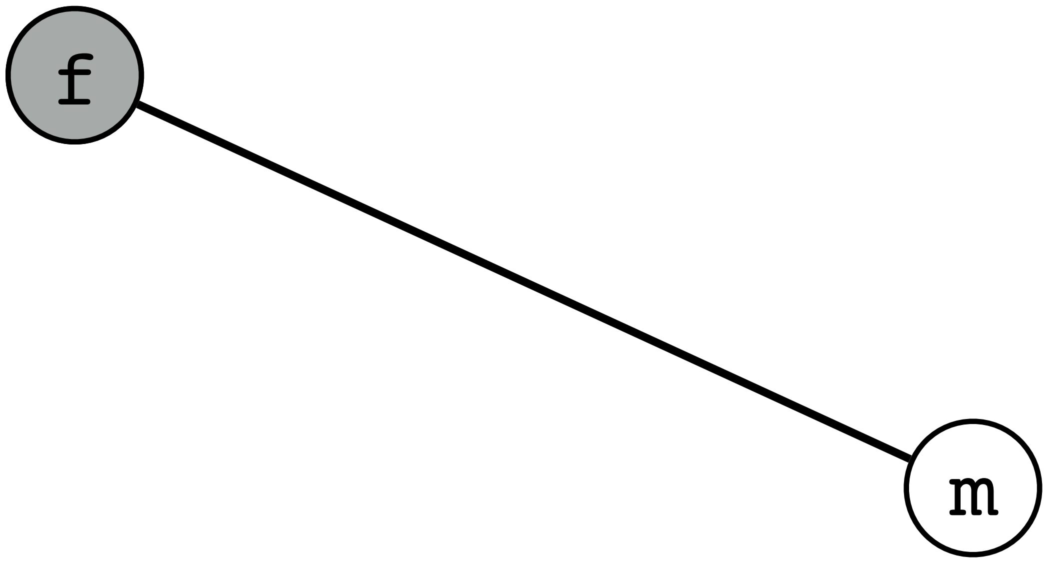


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`
`g`
`d` *spill?*
`k`
`j`
`b`
`e`
`f`



Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`

`c`

`g`

`d` *spill?*

`k`

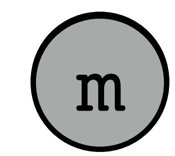
`j`

`b`

`e`

`f`

`m`



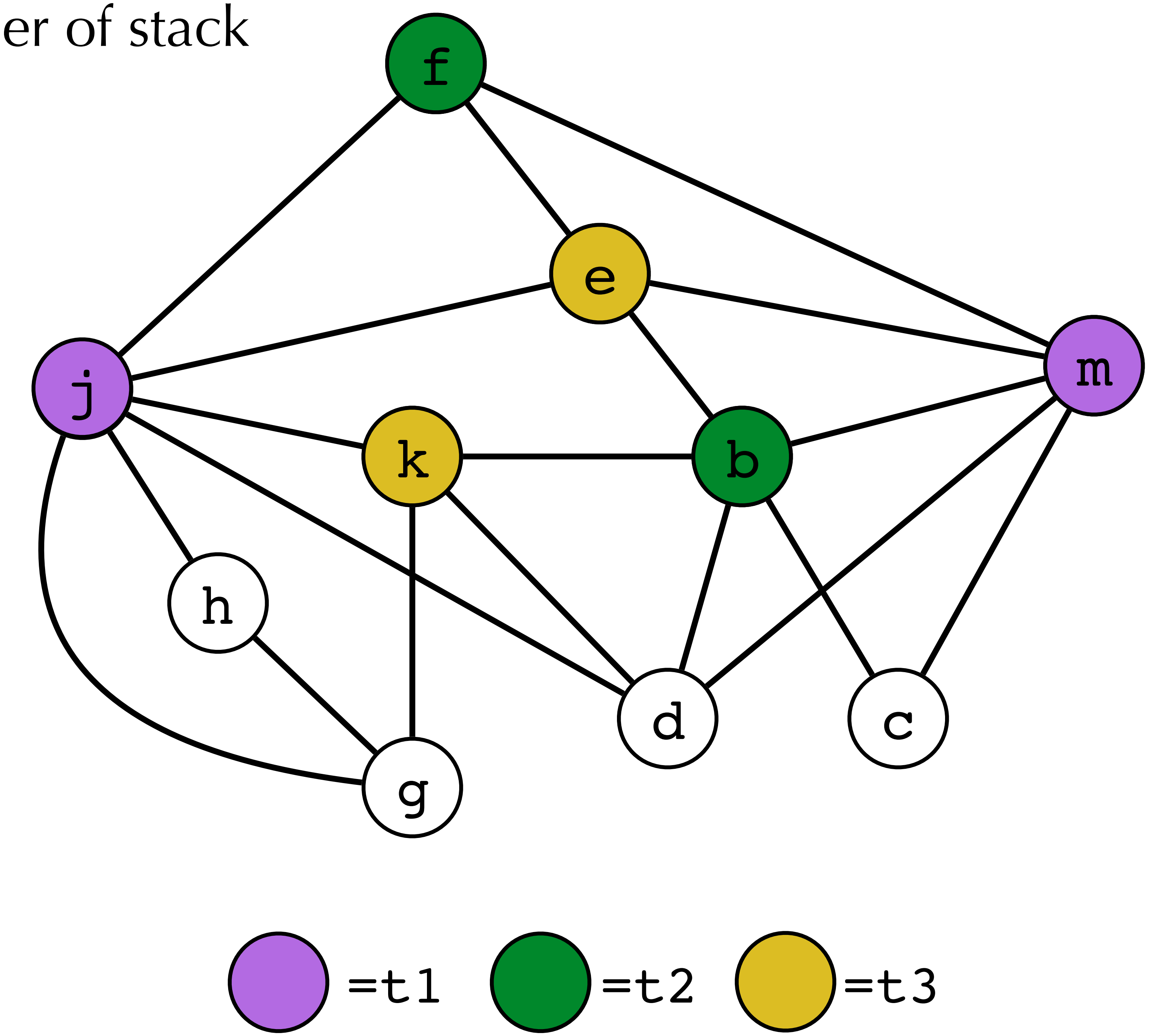
Select (3 registers)

Graph is now empty!

Stack:

Color nodes in order of stack

`h`
`c`
`g`
`d` *spill?*
`k`
`j`
`b`
`e`
`f`
`m`



Select (3 registers)

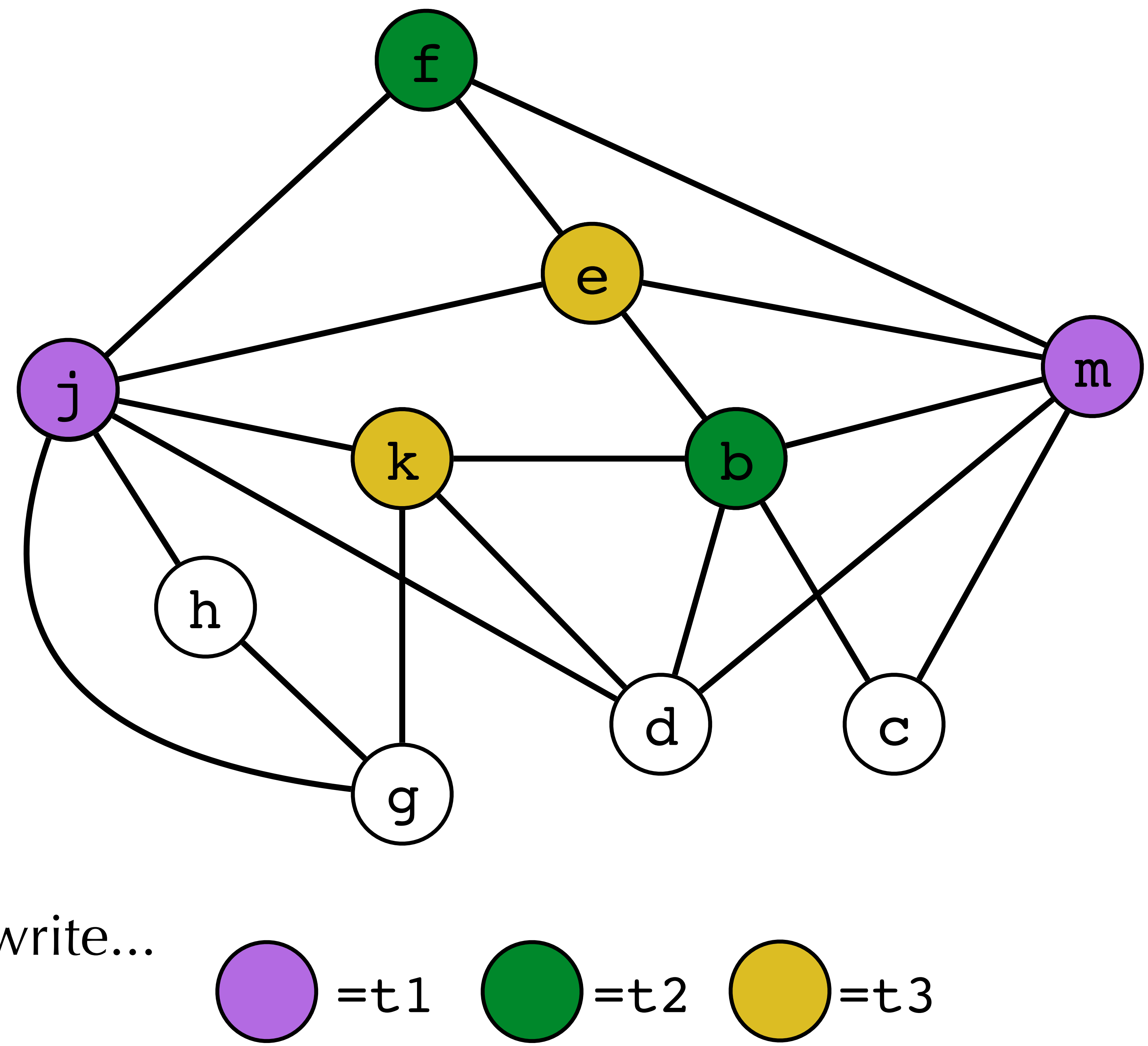
Stack:

```
h
c
g
d
```
spill?

We got unlucky!

In some cases a potential spill node is still colorable, and the Select phase can continue.

But in this case, we need to rewrite...



Select (3 registers)

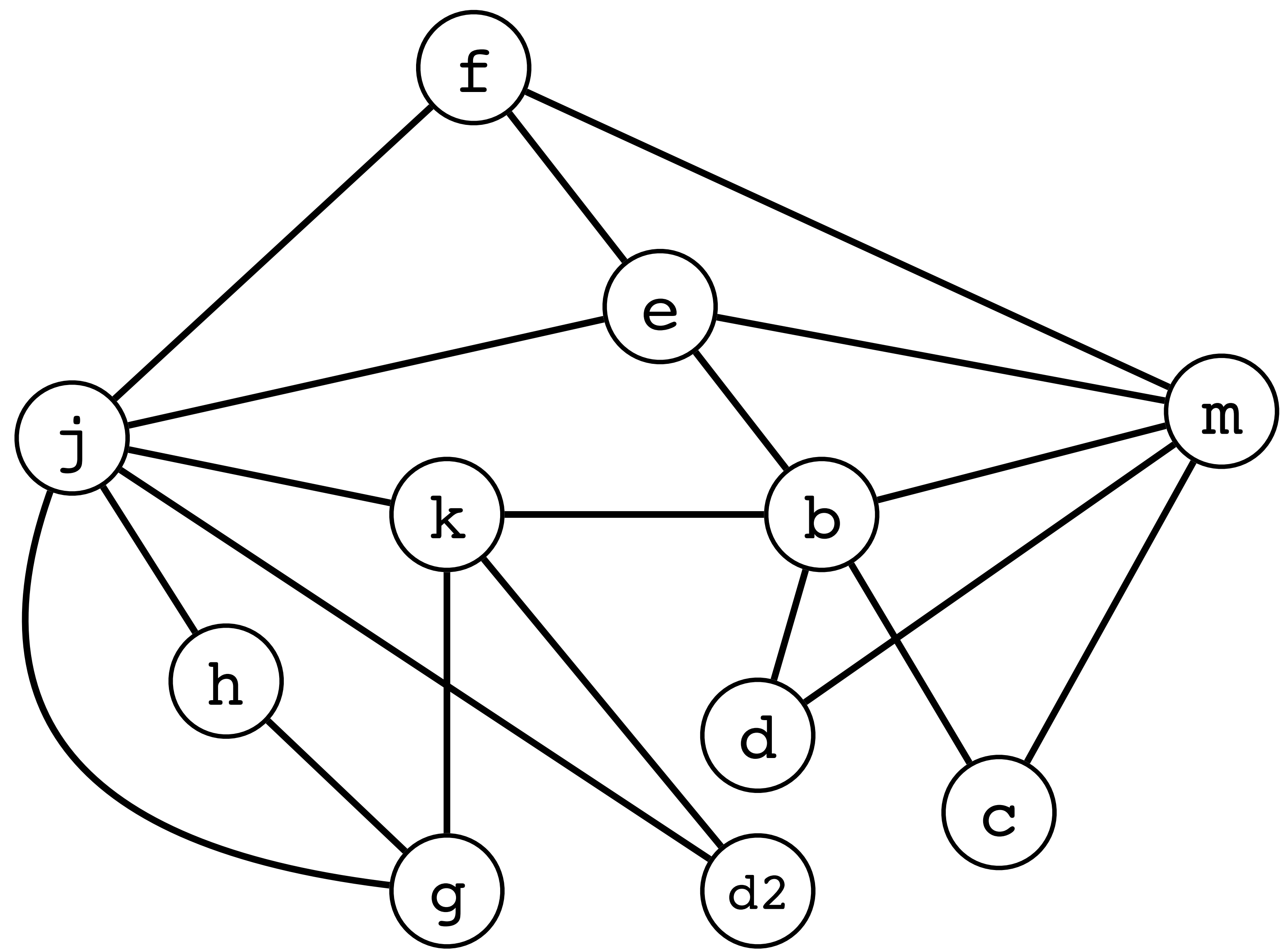
- Spill d

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d,j,k}
```

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
*<fp+doff>:=d
k := m + 4
j := b
d2:=*<fp+doff>
{live-out: d2,j,k}
```

Build

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
*<fp+doff>:=d
k := m + 4
j := b
d2:=*<fp+doff>
{live-out: d2,j,k}
```

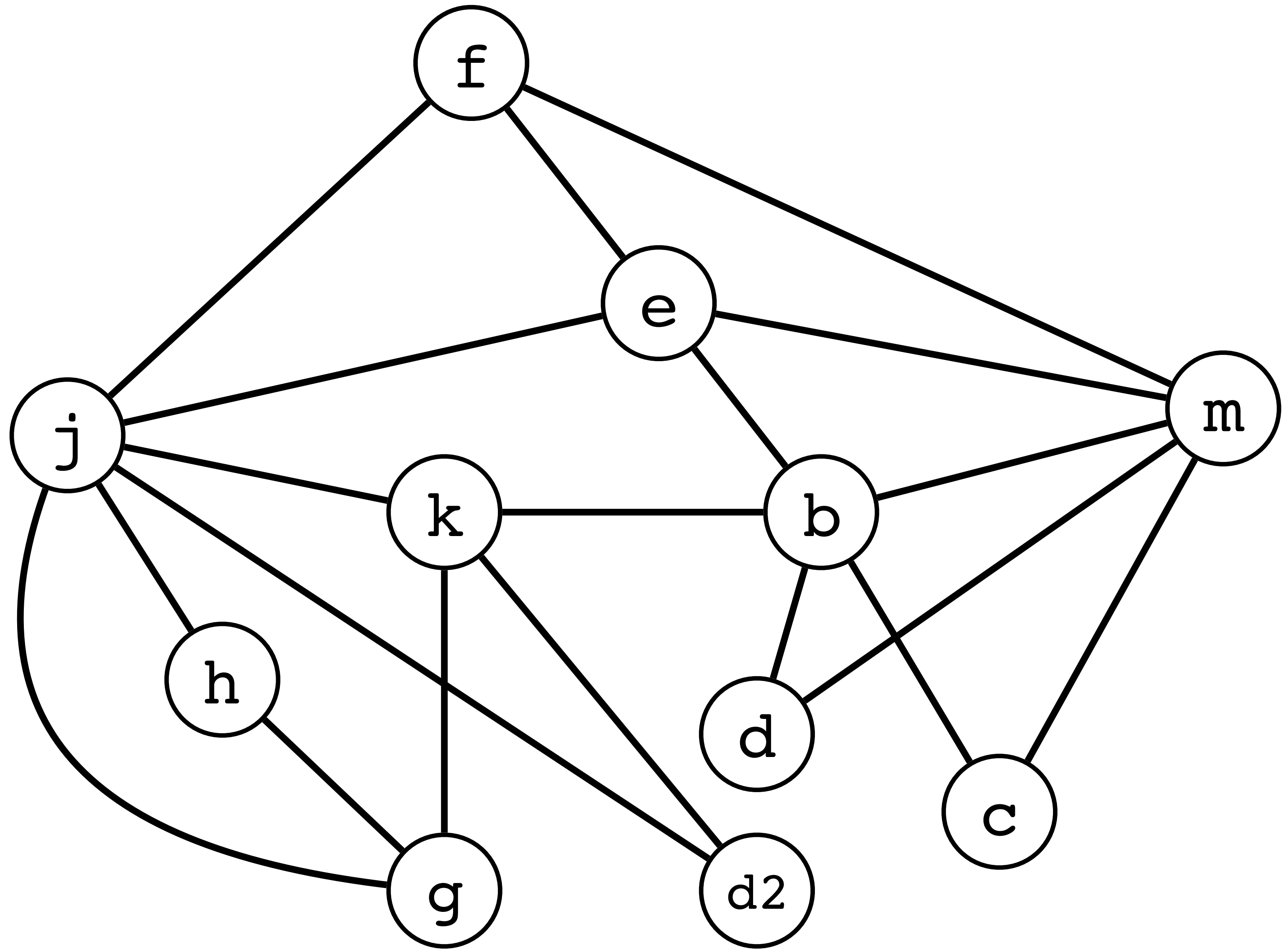


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g
d
d2
k
b
m
e
f
j



This time we succeed and
will be able to complete Select phase successfully!

Register Pressure

- Some optimizations increase live-ranges:
  - Copy propagation
  - Common sub-expression elimination
  - Loop invariant removal
- In turn, that can cause the allocator to spill
- Copy propagation isn't that useful anyway:
  - Let register allocator figure out if it can assign the same register to two temps!
  - Then the copy can go away.
  - And we don't have to worry about register pressure.

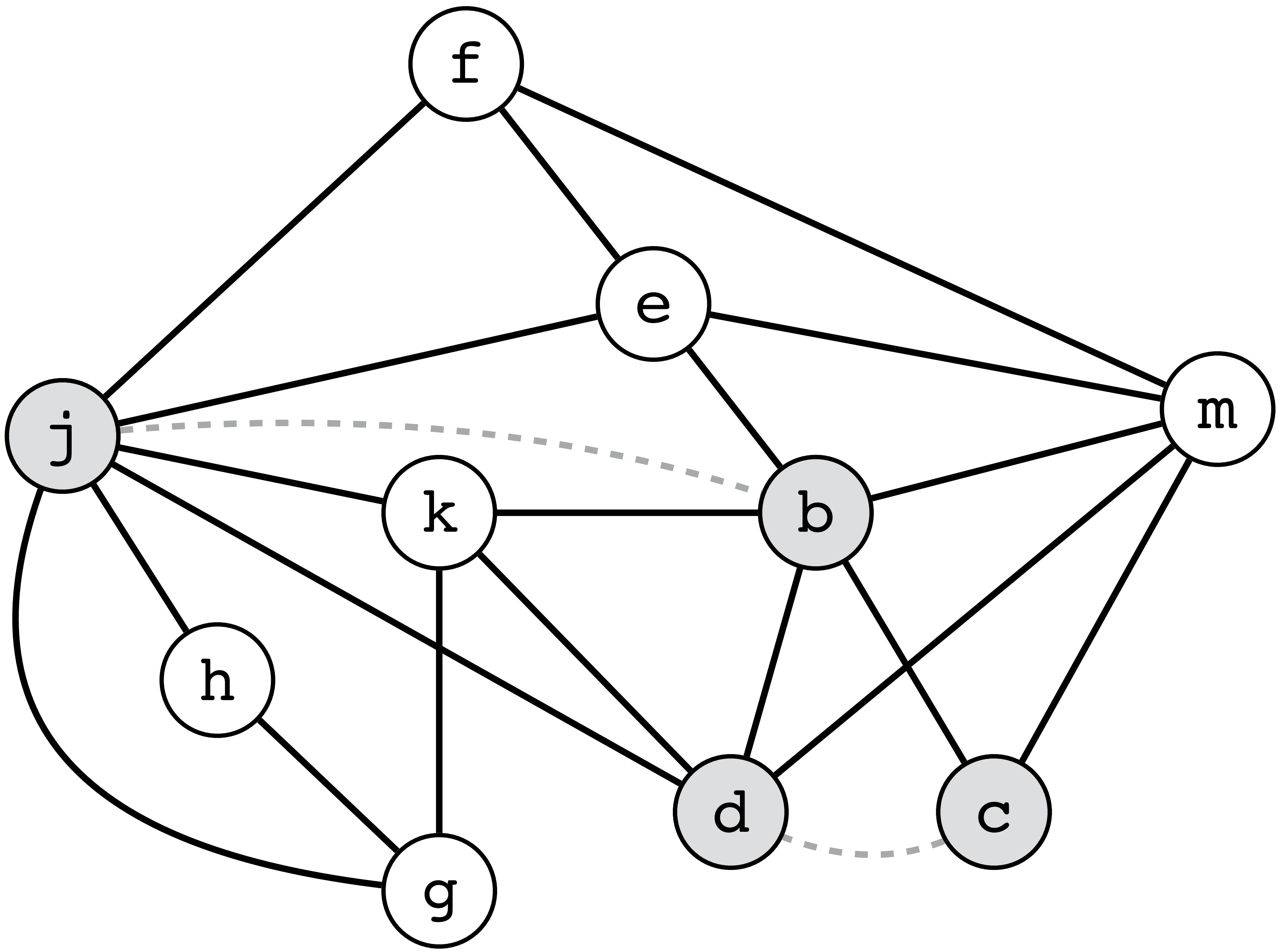
Coalescing Register Allocation

- 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.
  - This would translate to "`ri := ri`" which would then be removed
- One idea is to optimistically coalesce nodes in the interference graph
  - Just take the edges to be the union

Example

- E.g., the following nodes could be coalesced
  - `d` and `c`
  - `j` and `b`

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d,j,k}
```

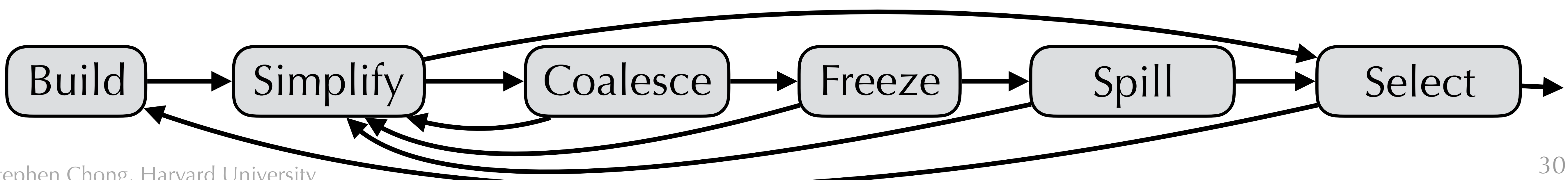


Coalescing Heuristics

- But coalescing may make a *k*-colorable graph uncolorable!
- Briggs: safe to coalesce `x` and `y` if the resulting node will have fewer than *k* neighbors with degree $\geq k$.
- George: safe to coalesce `x` and `y` if for every neighbor `t` of `x`, either `t` already interferes with `y` or `t` has degree $< k$
- These strategies are conservative: will not turn a *k*-colorable graph into a non-*k*-colorable graph
    - Why?

Coloring with Coalescing

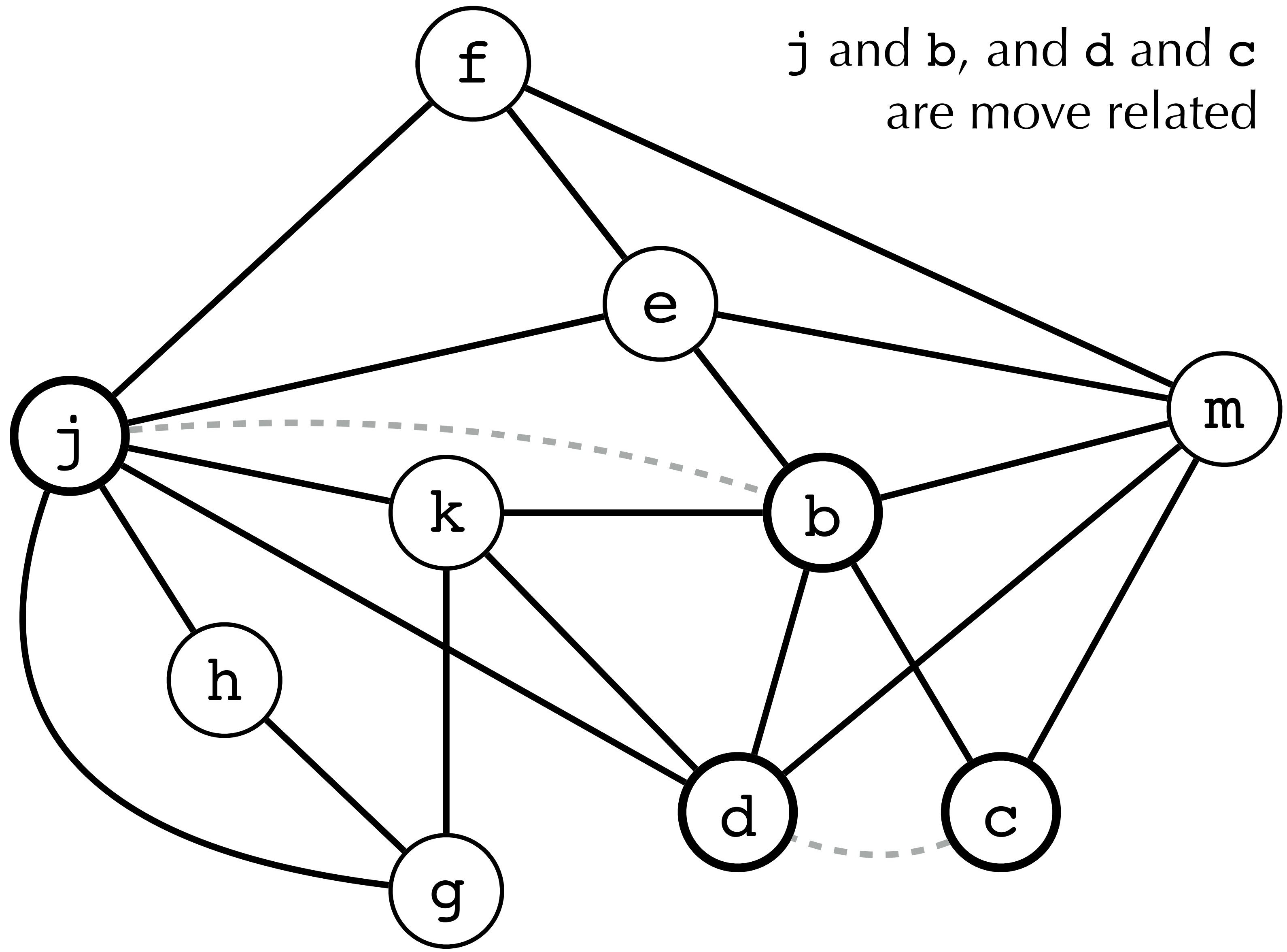
- **Build:** construct interference graph
  - Categorize nodes as *move-related* (if src or dest of move) or *non-move-related*
- **Simplify:** Remove non-move-related nodes with degree $<k$
- **Coalesce:** Coalesce nodes using Briggs' or George's heuristic
  - Possibly re-mark coalesced nodes as *non-move-related*
  - Continue with Simplify if there are nodes with degree $<k$
- **Freeze:** if some low-degree ($<k$) move-related node, **freeze** it
  - i.e., make it non-move-related, i.e., give up on coalescing that node
  - Continue with Simplify
- **Spill:** choose node with degree $\geq k$ to **potentially spill**
  - Then continue with simplify
- **Select:** when graph is empty, start restoring nodes in reverse order and color them
  - Potential spill node: try coloring it; if not rewrite program to use stack and try again!

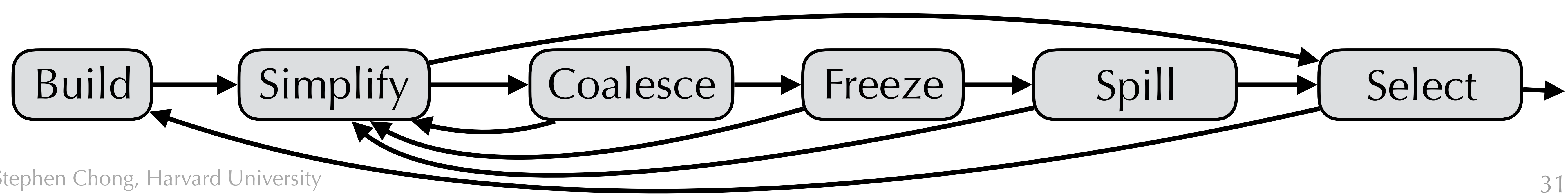


Example (4 registers)

Stack:

`j` and `b`, and `d` and `c` are move related



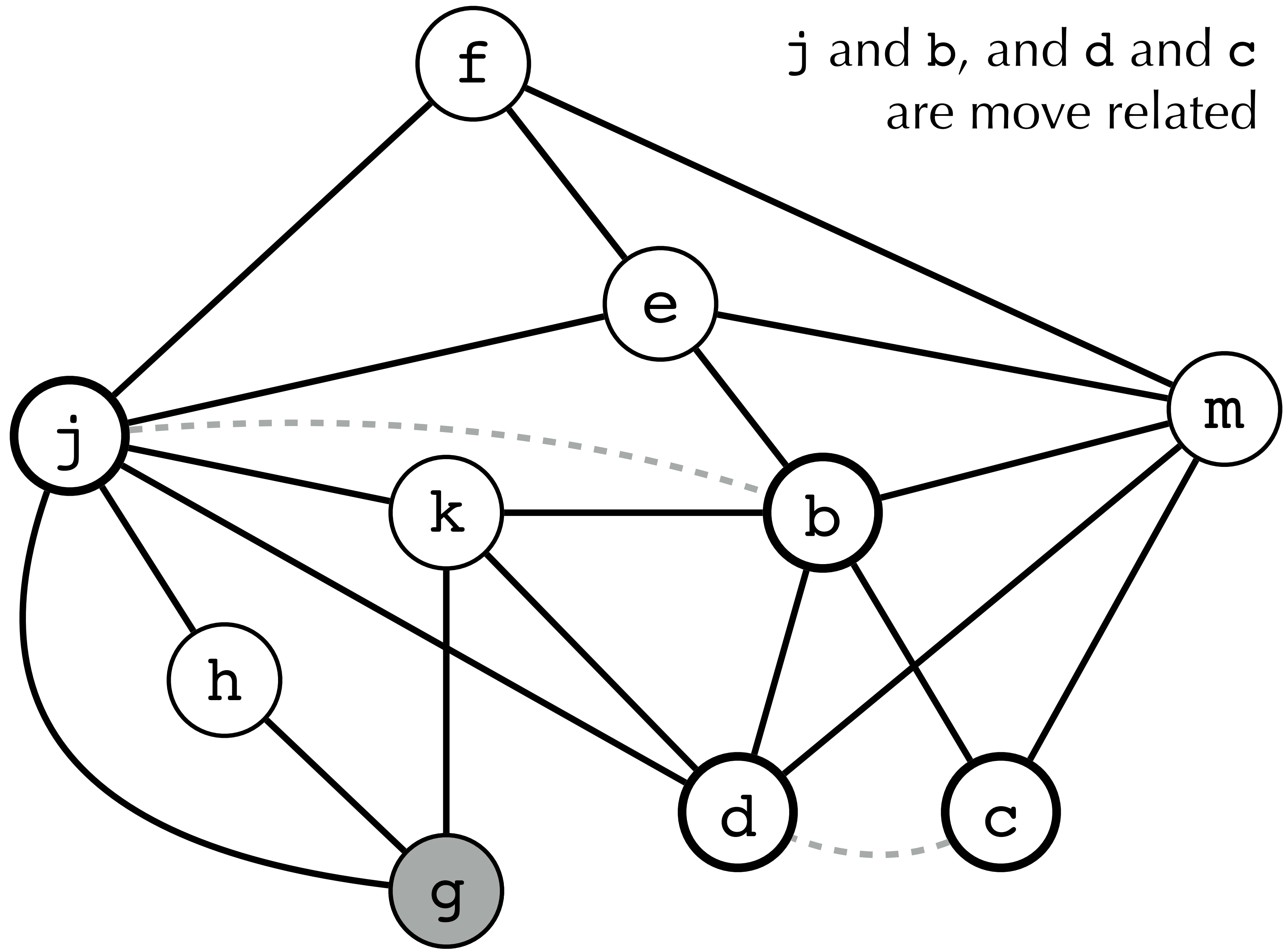


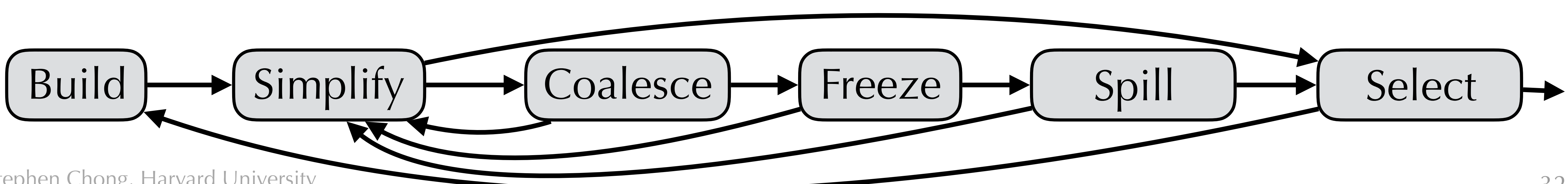
Example (4 registers)

Stack:

g

j and b, and d and c are move related



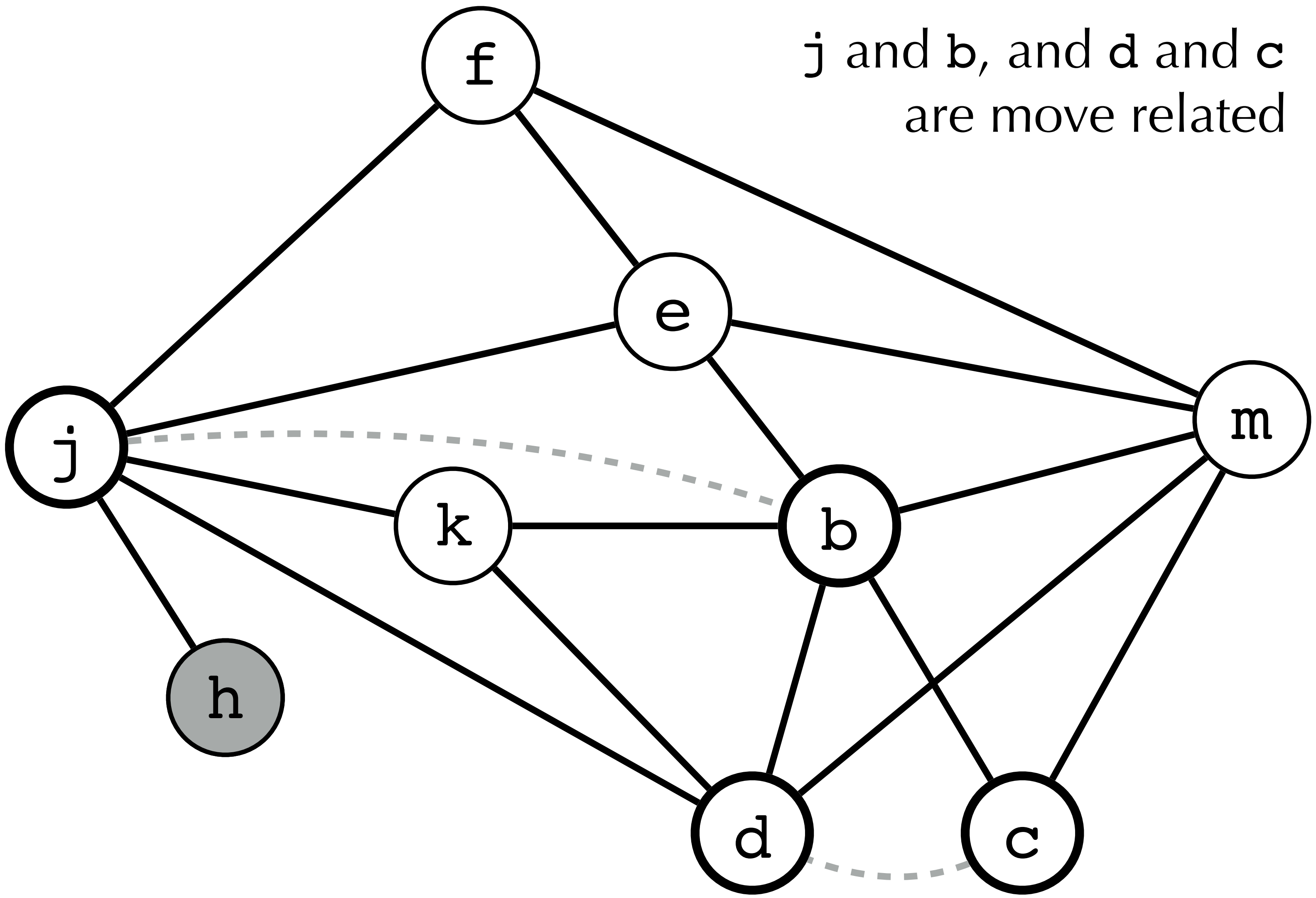


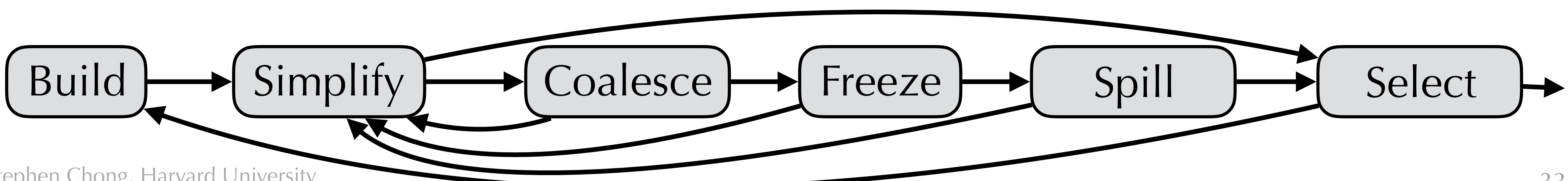
Example (4 registers)

Stack:

g
h

j and b, and d and c are move related

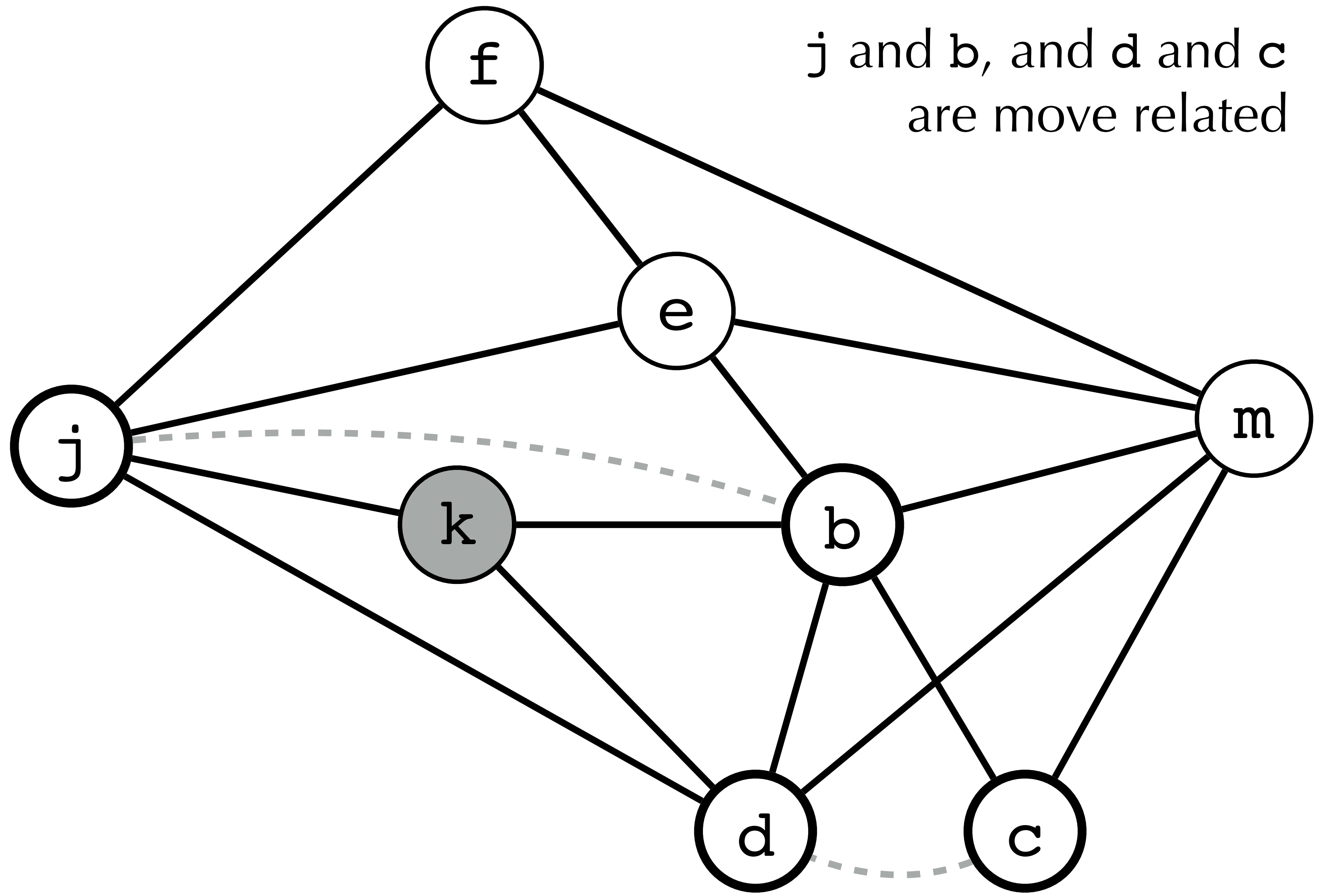




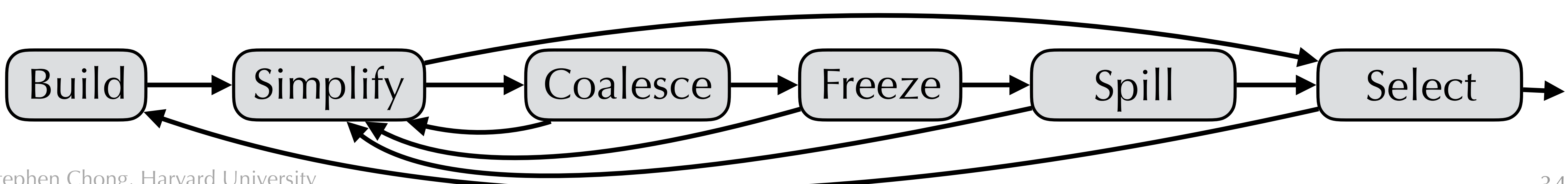
Example (4 registers)

Stack:

g
h
k



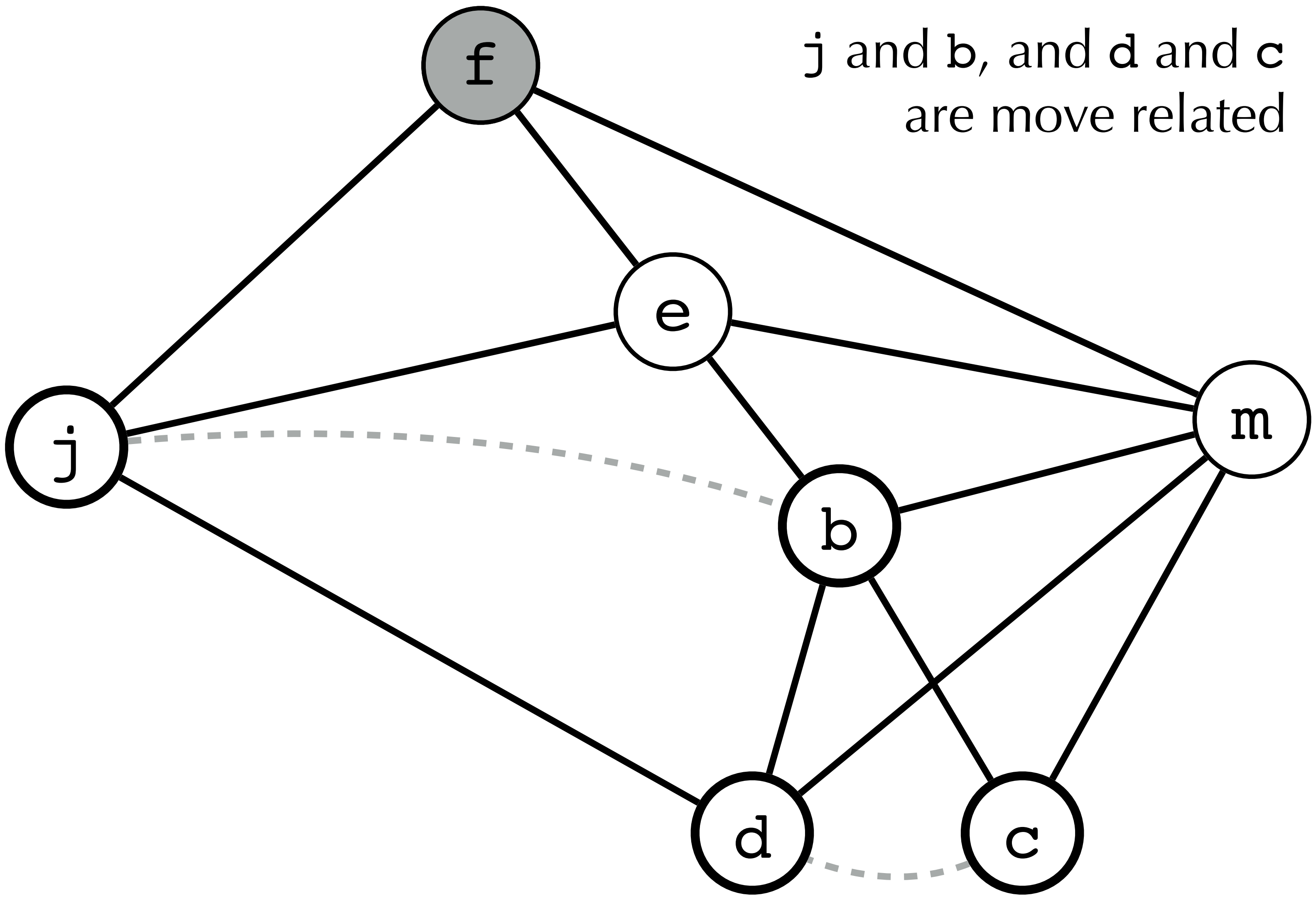
j and b, and d and c are move related



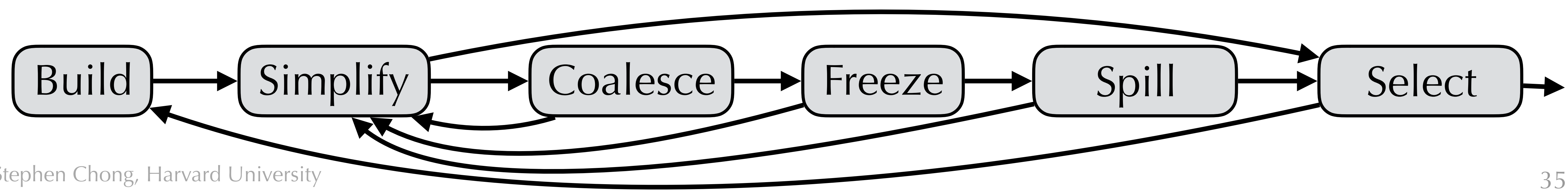
Example (4 registers)

Stack:

g
h
k
f



j and b, and d and c are move related

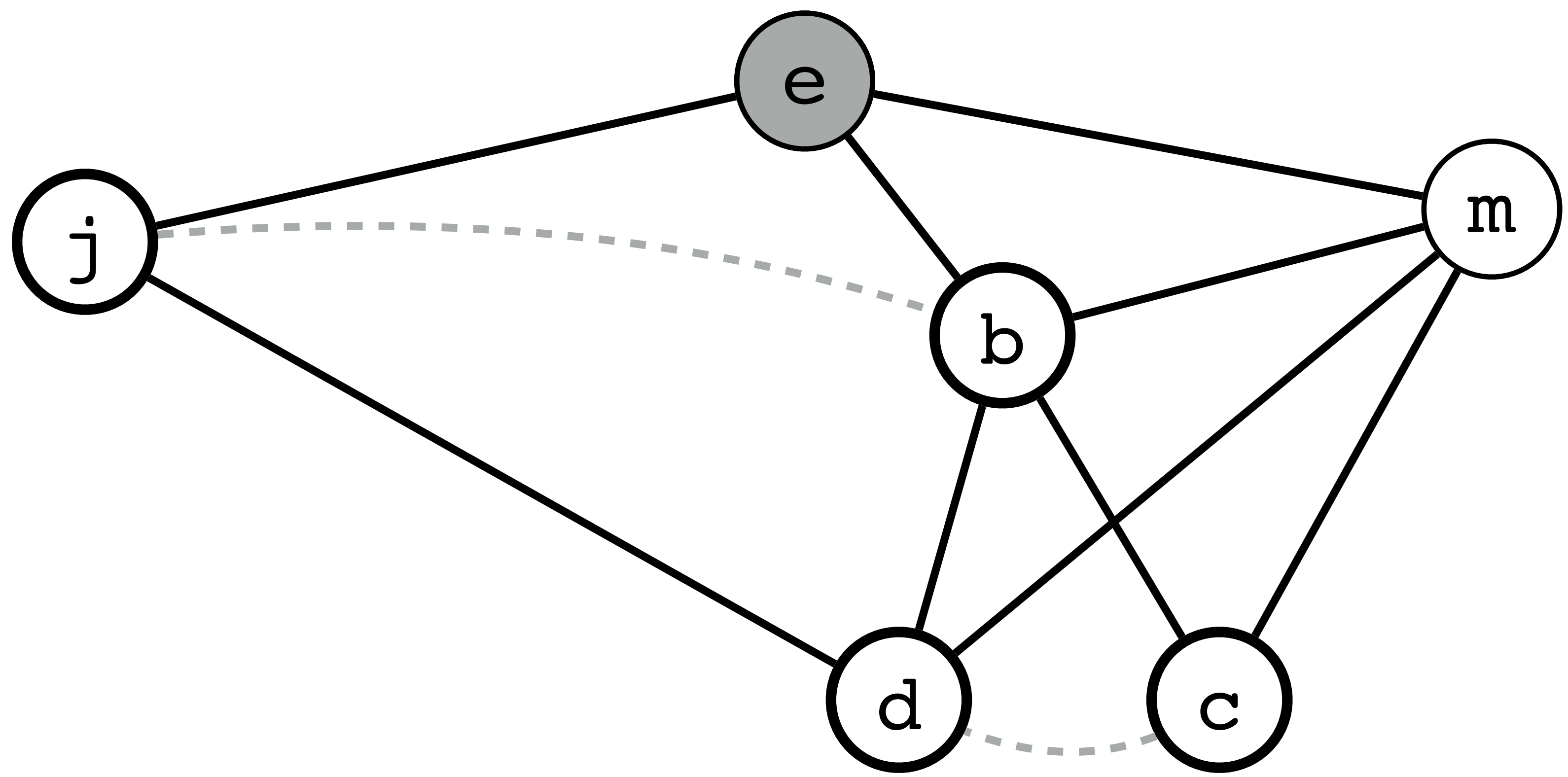


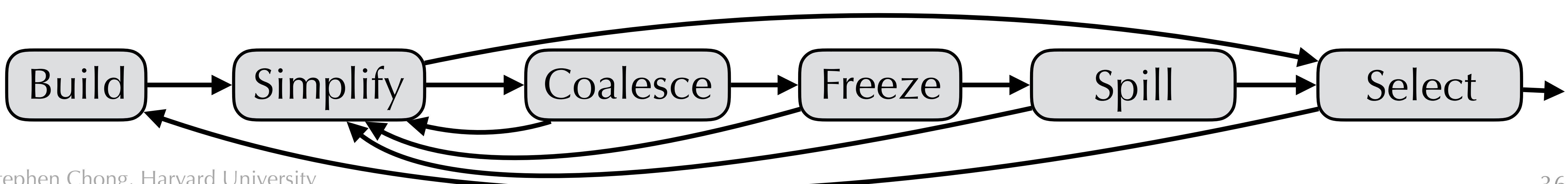
Example (4 registers)

Stack:

g
h
k
f
e

`j` and `b`, and `d` and `c` are move related



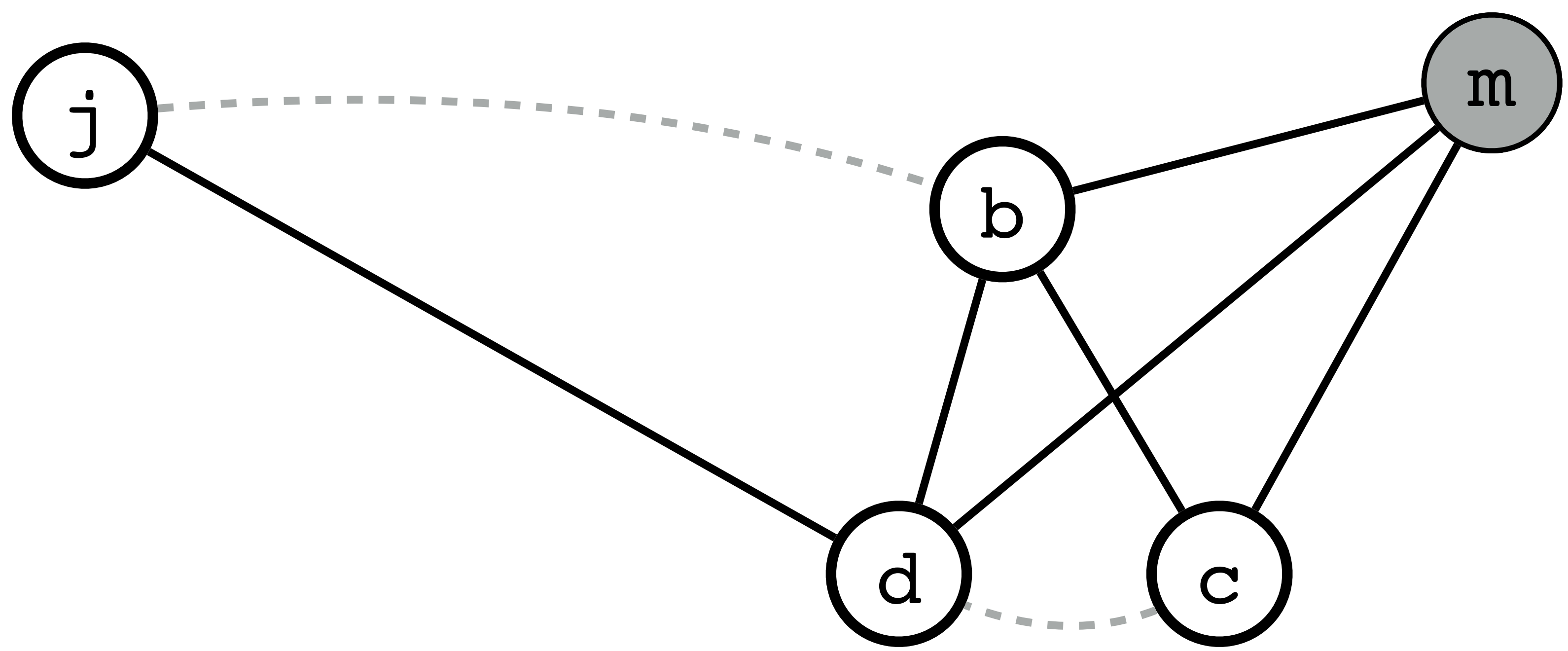


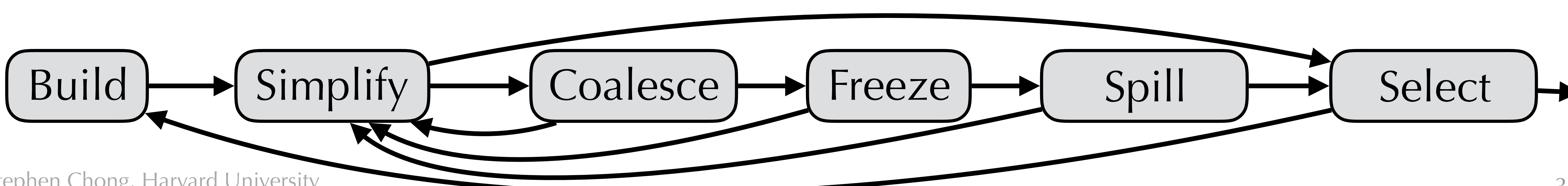
Example (4 registers)

Stack:

g
h
k
f
e
m

`j` and `b`, and `d` and `c` are move related



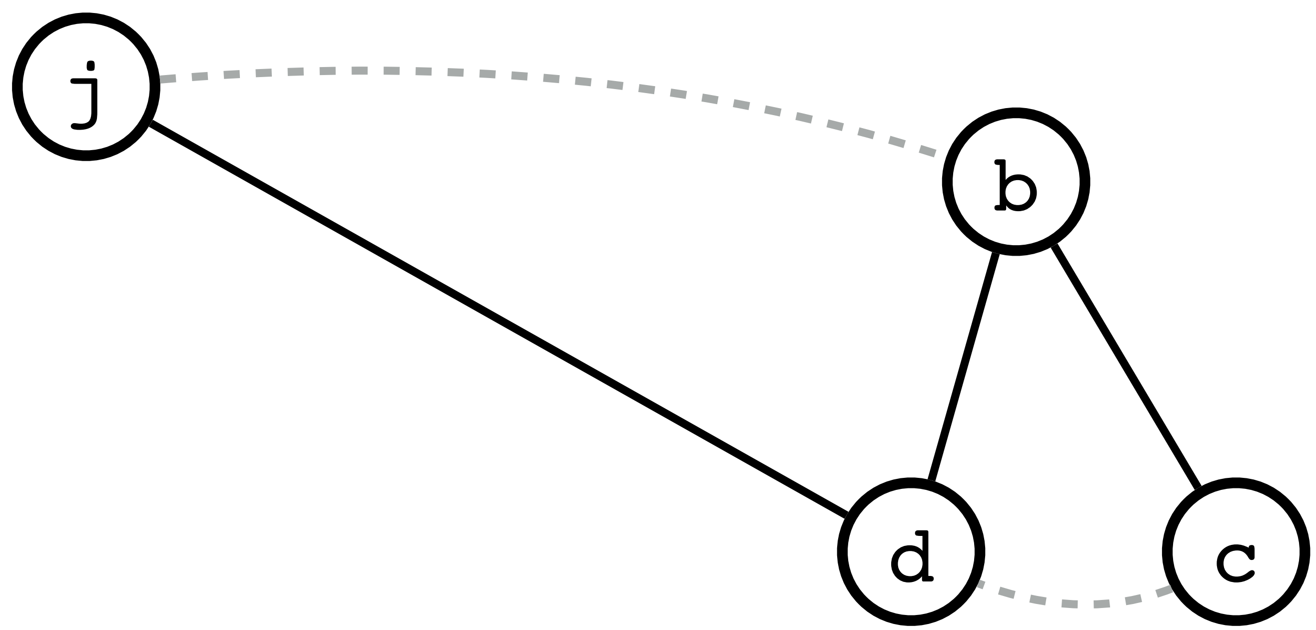


Example (4 registers)

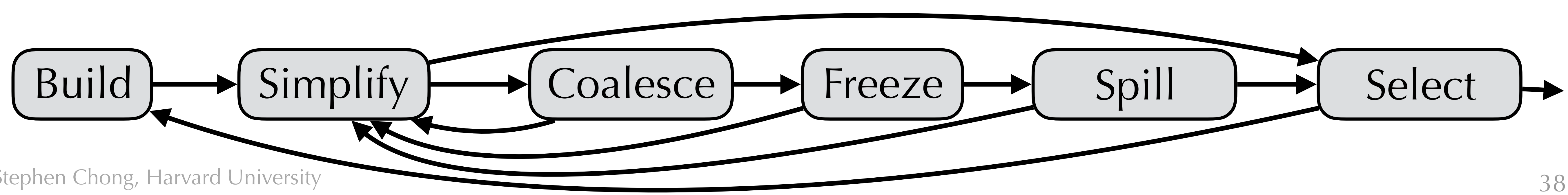
`j` and `b`, and `d` and `c`
are move related

Stack:

g
h
k
f
e
m



Remaining nodes are move related, so coalesce

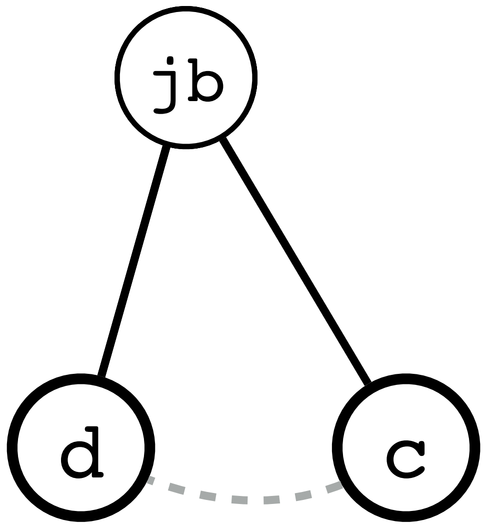


Example (4 registers)

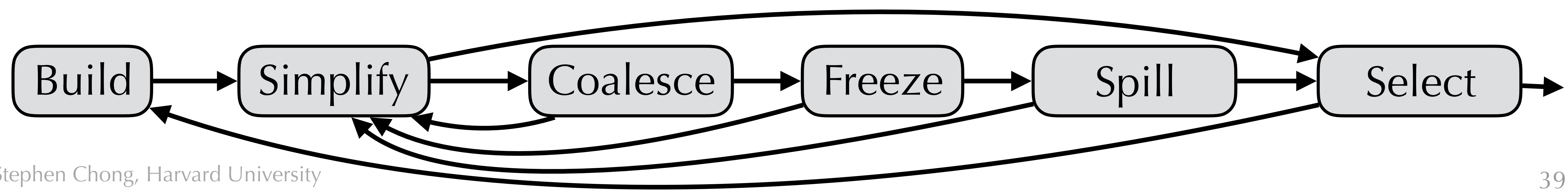
`d` and `c`
are move related

Stack:

g
h
k
f
e
m



Remaining nodes are move related, so coalesce

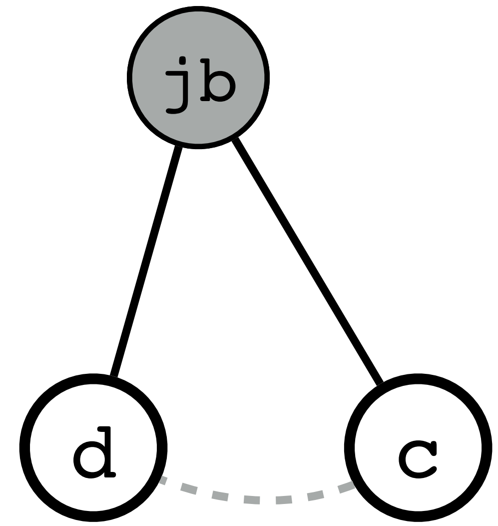


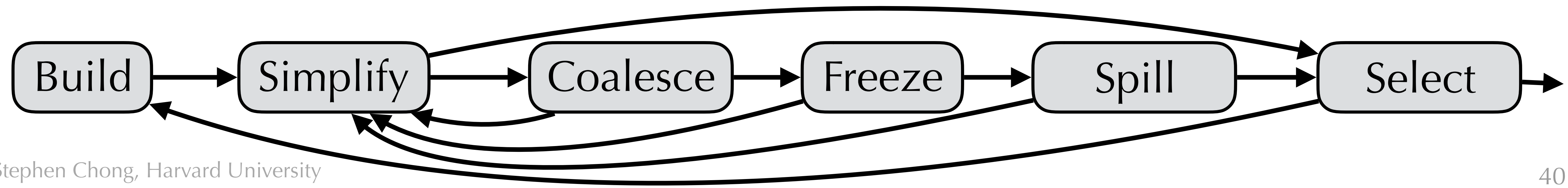
Example (4 registers)

`d` and `c`
are move related

Stack:

g
h
k
f
e
m
jb



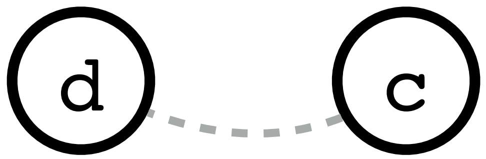


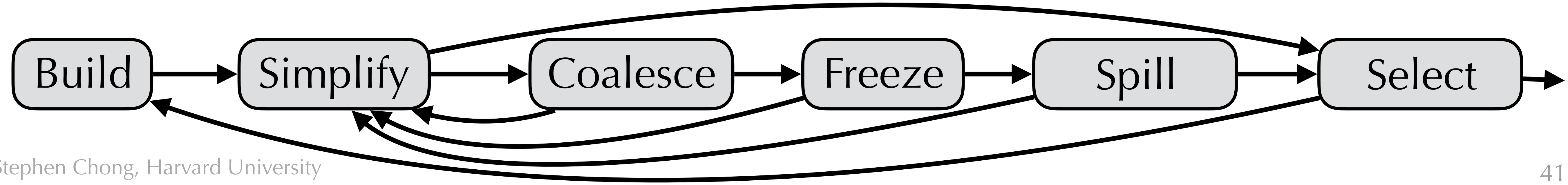
Example (4 registers)

Stack:

g
h
k
f
e
m
jb

`d` and `c` are move related

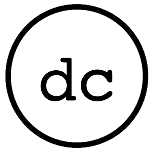


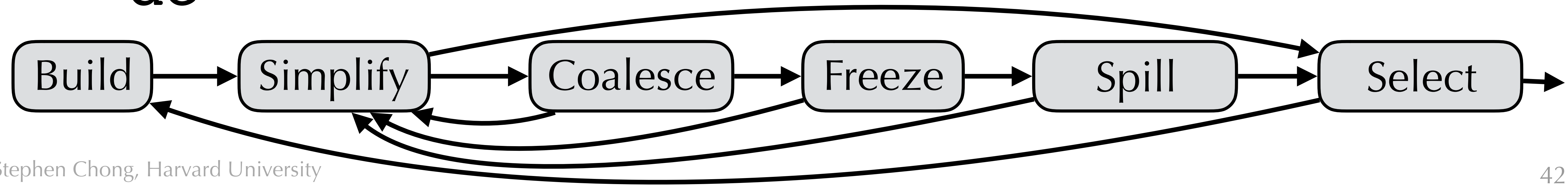


Example (4 registers)

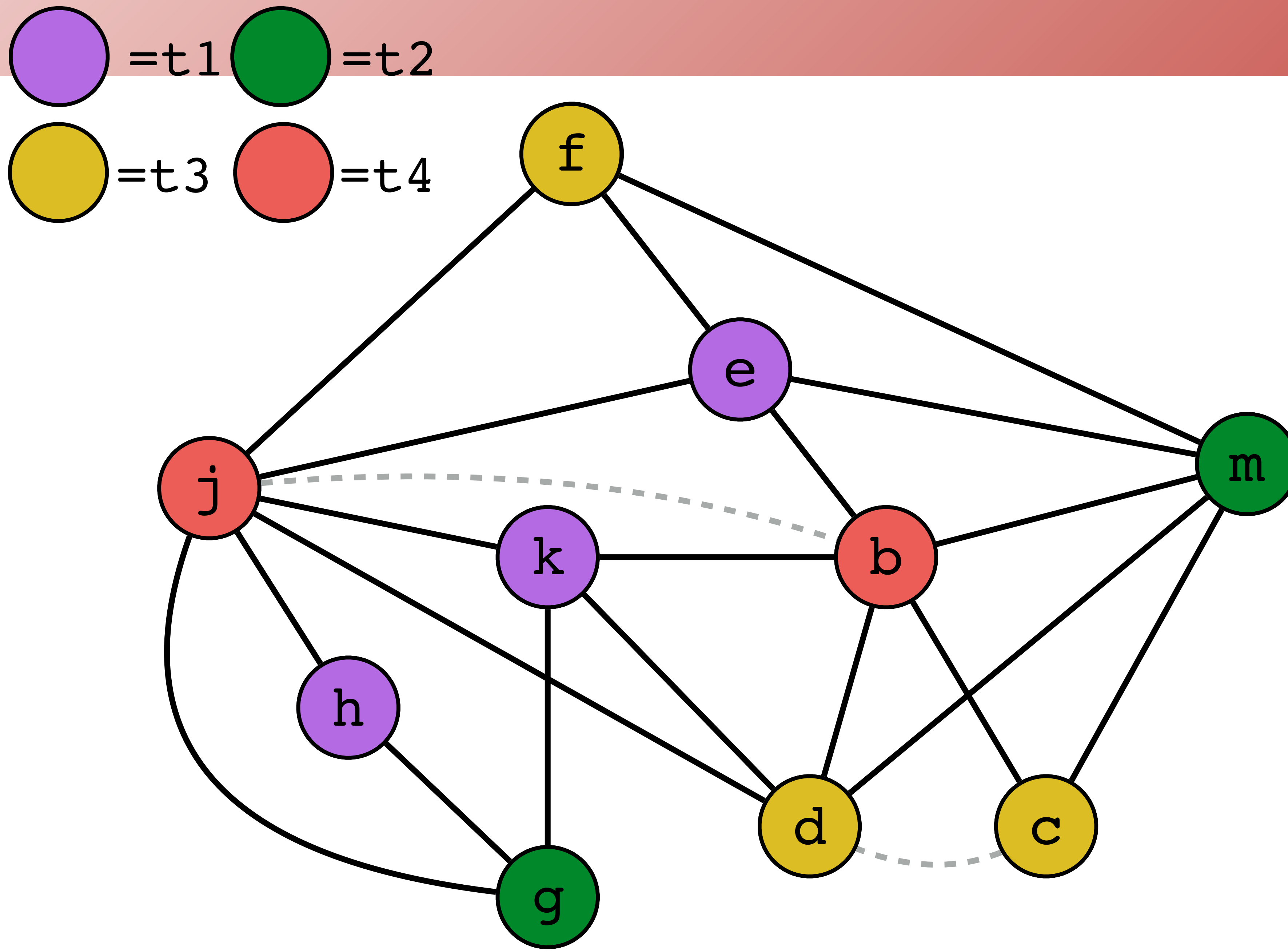
Stack:

```
g
h
k
f
e
m
jb
dc
```



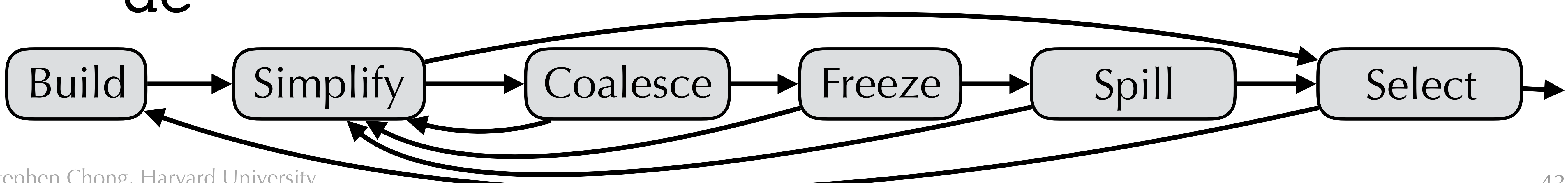


Example (4 registers)

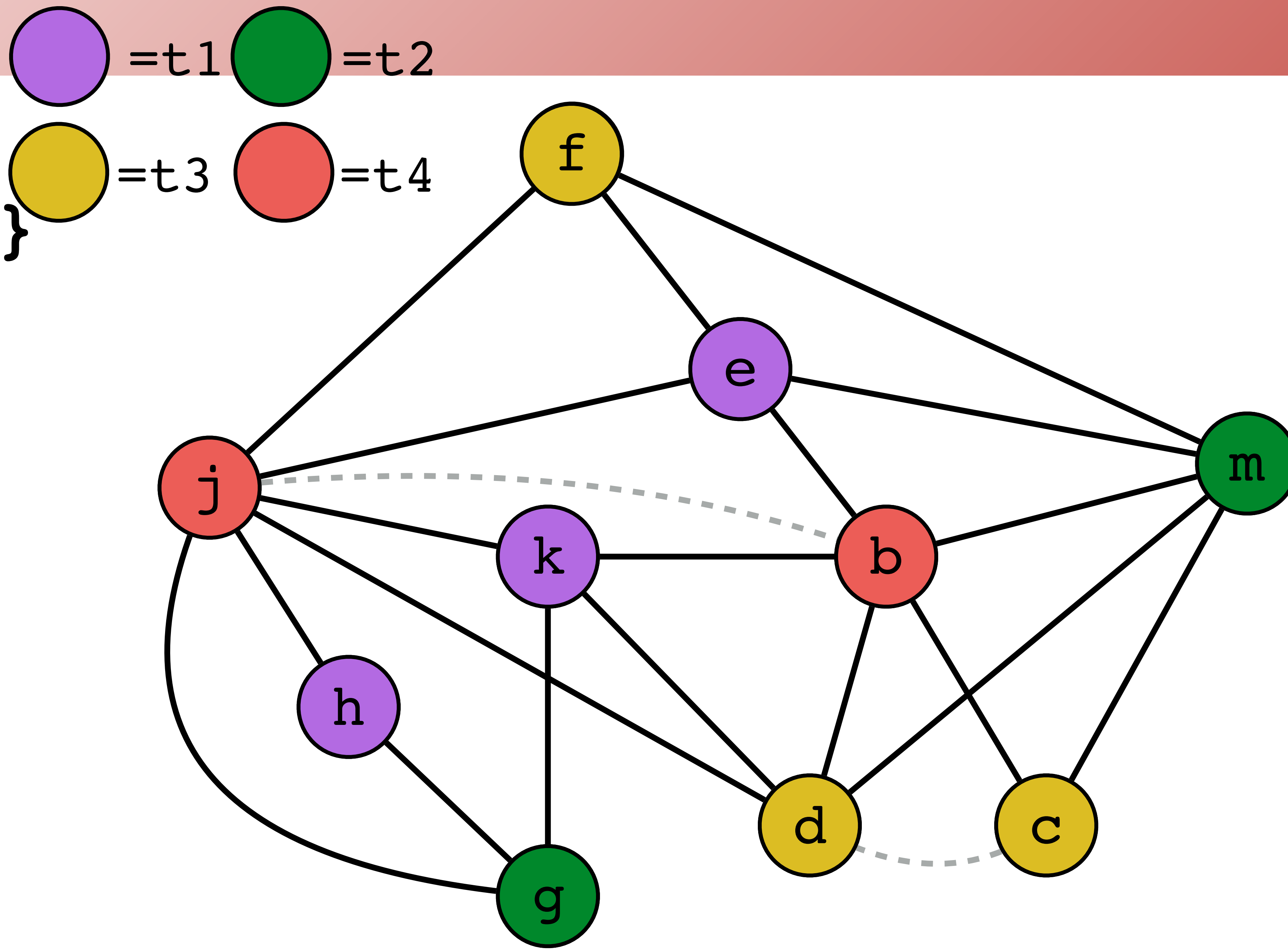


Stack:

g
h
k
f
e
m
jb
dc

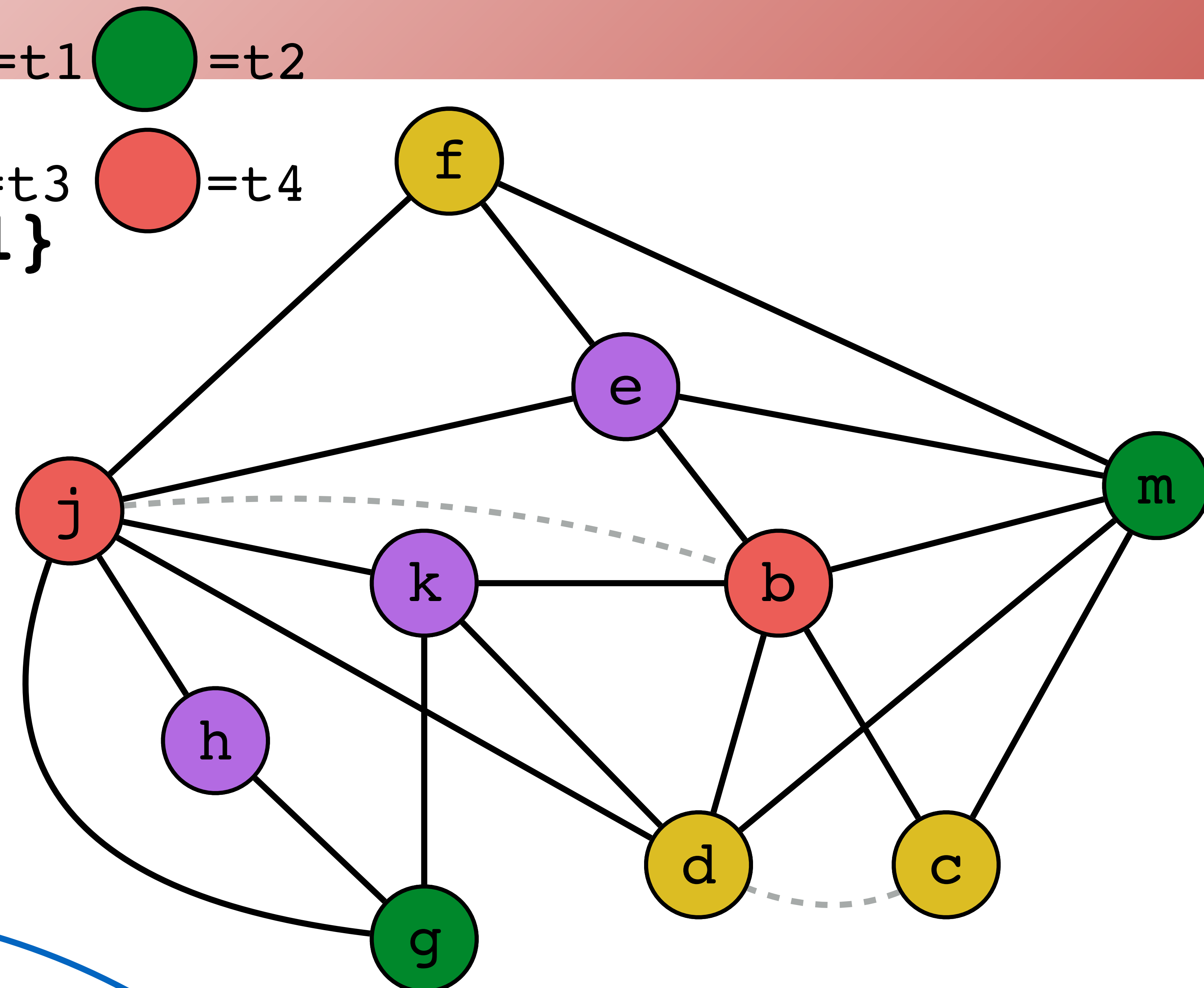


Example (4 registers)




```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d,j,k}
```

Example (4 registers)




```
{live-in: $t4, $t1}
$t2 := *($t4+12)
$t1 := $t1 - 1
$t3 := $t2 * $t1
$t1 := *($t4+8)
$t2 := *($t4+16)
$t4 := *(f+0)
$t3 := $t1 + 8
$t3 := $t3
$t1 := $t2 + 4
$t4 := $t4
{live-out: $t3,$t4,$t1}
```

This is the result of coalescing!

Pre-colored Temps

- The IR often includes machine registers
  - e.g., `$rbp`, `$rsp`, `$rcx`, `$rdx`, ...
  - 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 and Coalesce phases 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 `$rdi`, `$rsi`, `$rdx`, `$rcx`, `$r8`, `$r9` (and `Mem[$rbp+offset]`) into fresh temps, move result into `$rax`
  - 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
- Similarly, we cannot spill a pre-colored node

Callee-Save Registers

- Callee-Save register `r`:
  - Is "defined" upon entry to the procedure
  - Is "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.
- (Example of this soon)

Caller-Save Registers

- Want to assign a temp to a caller-save register only when it's not live across a function call
  - Since then we have to save/restore it
- So treat a function call as "defining" all caller-save registers.
  - Callee might move values into them
  - Now any temps that are live across the call will interfere, and register assignment will find different registers to assign the temps
- **Note:** When constructing interference graph, also need to make sure that any variable defined by a statement S interferes with any variable that is live-out for S. So if a function call "defines" all caller-save registers, all live-out variables live after the function call will interfere with all caller-save registers

Example

p238 in Appel

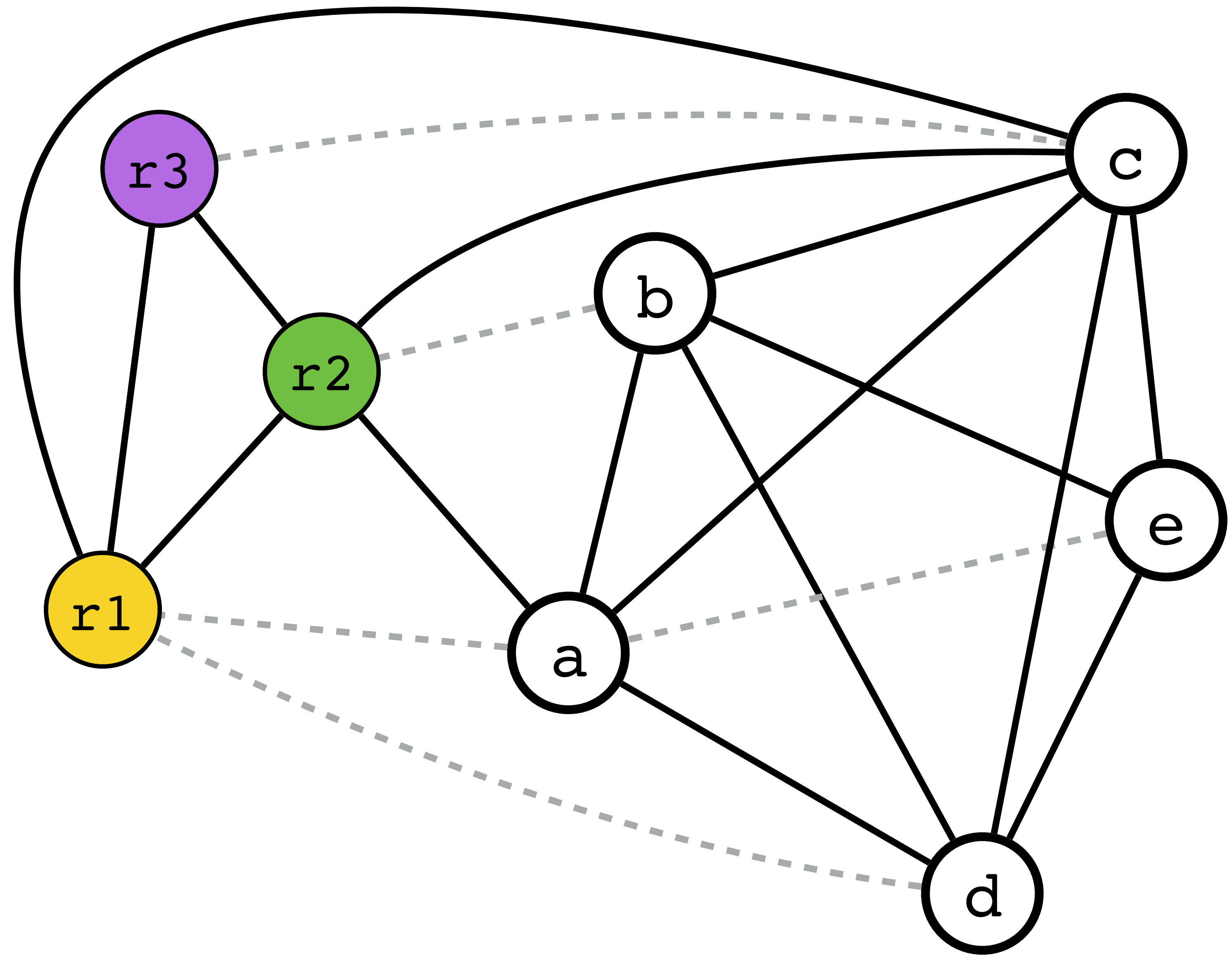
- Compile the following C function
  - Assume target machine has 3 registers
  - `$r1` and `$r2` are caller-save
  - `$r3` is callee-save

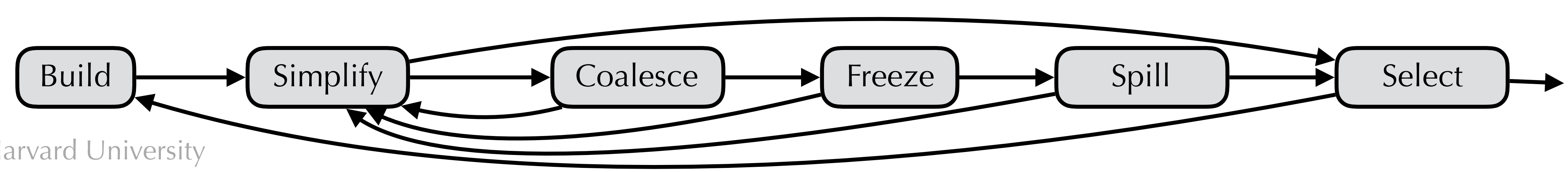
```
int f(int a, int b) {
   int d = 0;
   int e = a;
   do {
      d = d+b;
      e = e-1;
   } while (e > 0);
   return d;
}
```

```
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
```

Example

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

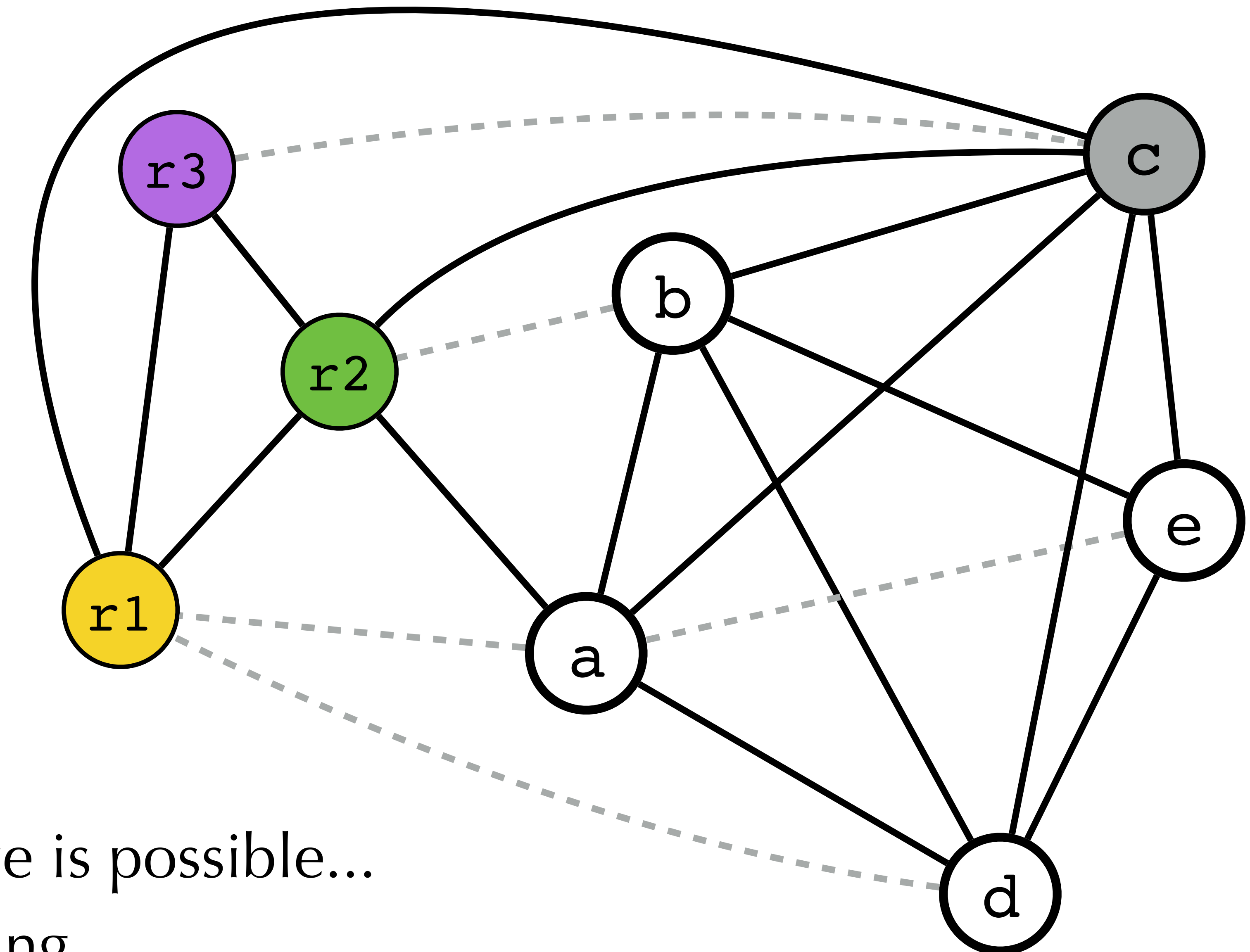




Example

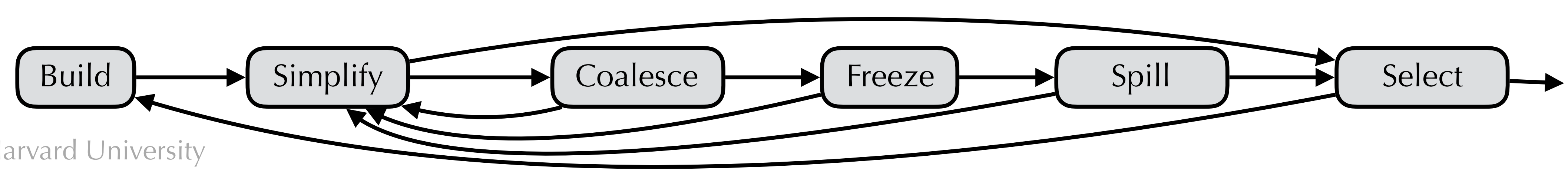
Stack:

c *spill?*



No simplify, coalesce, or freeze is possible...

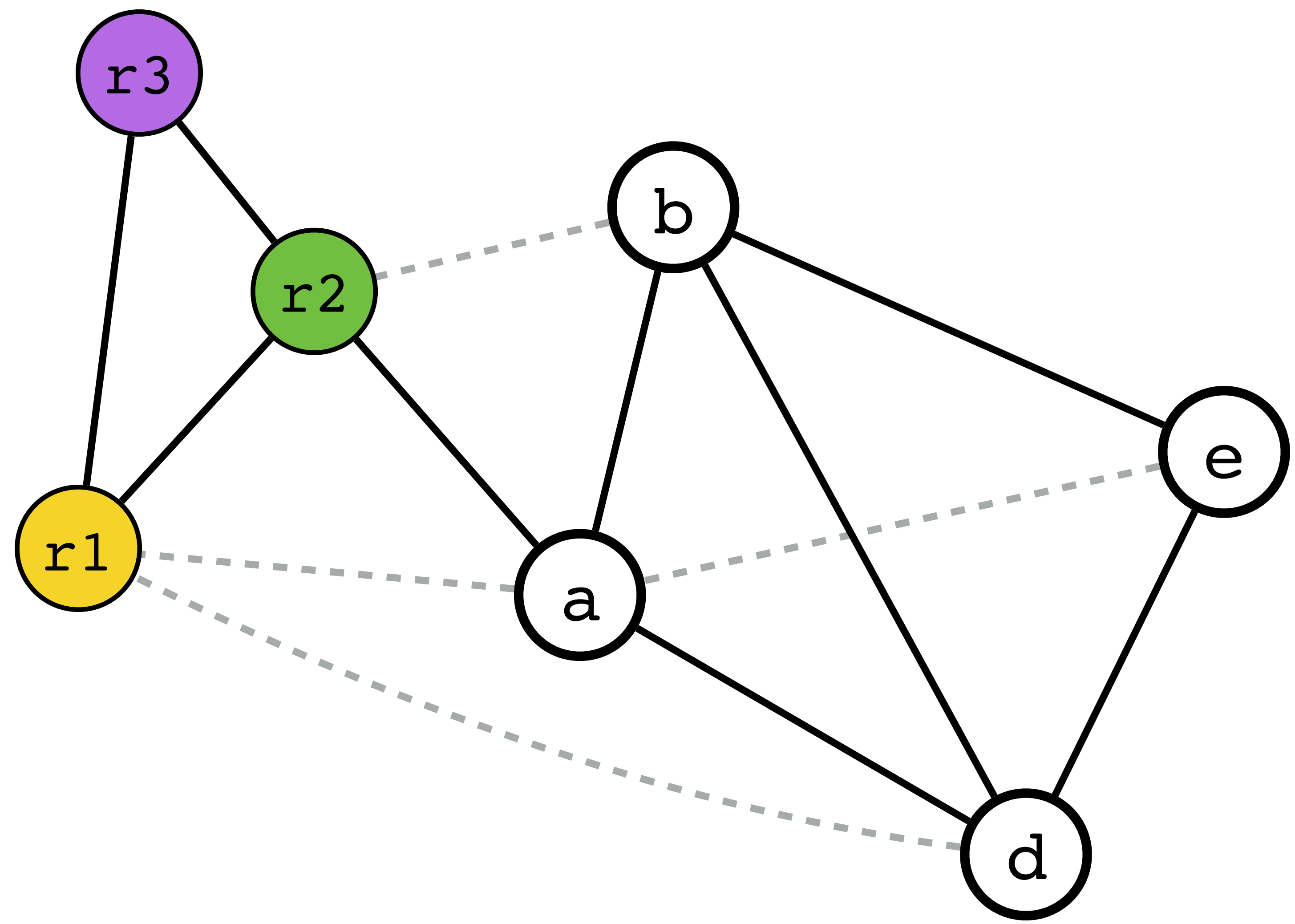
c is a good candidate for spilling...



Example

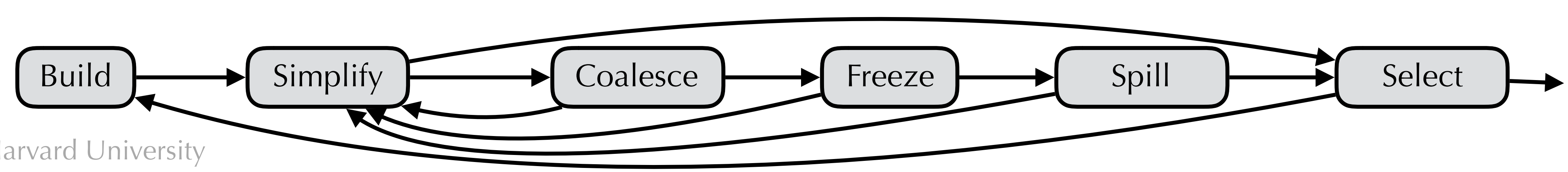
Stack:

c *spill?*



No simplify is possible...

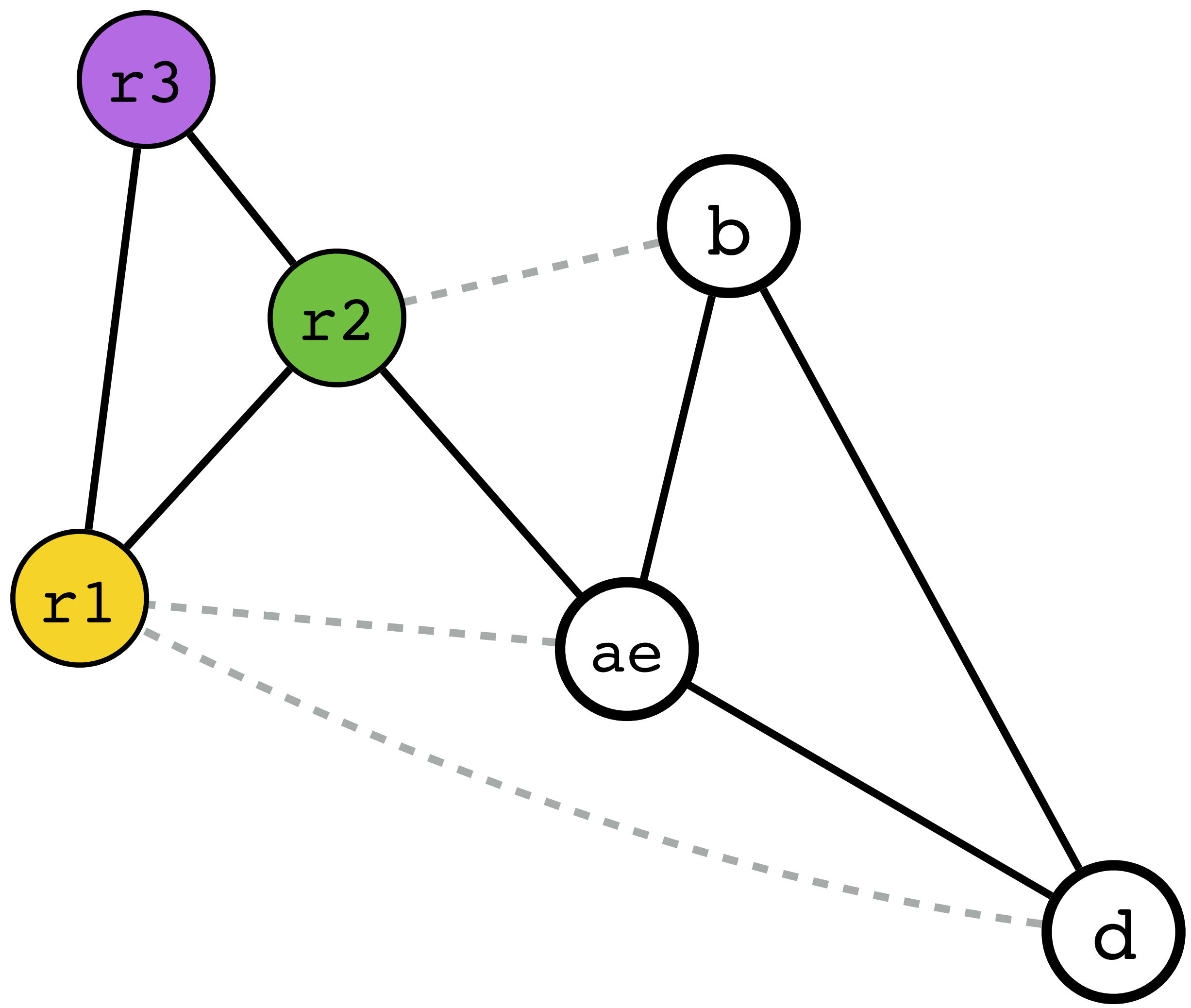
Coalesce **a** and **e**

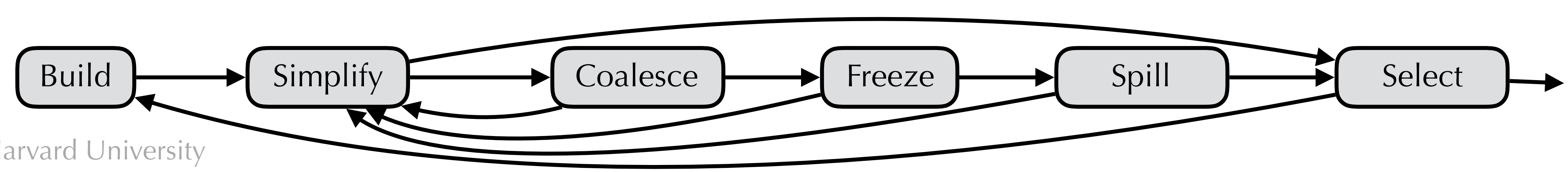


Example

Stack:

c *spill?*

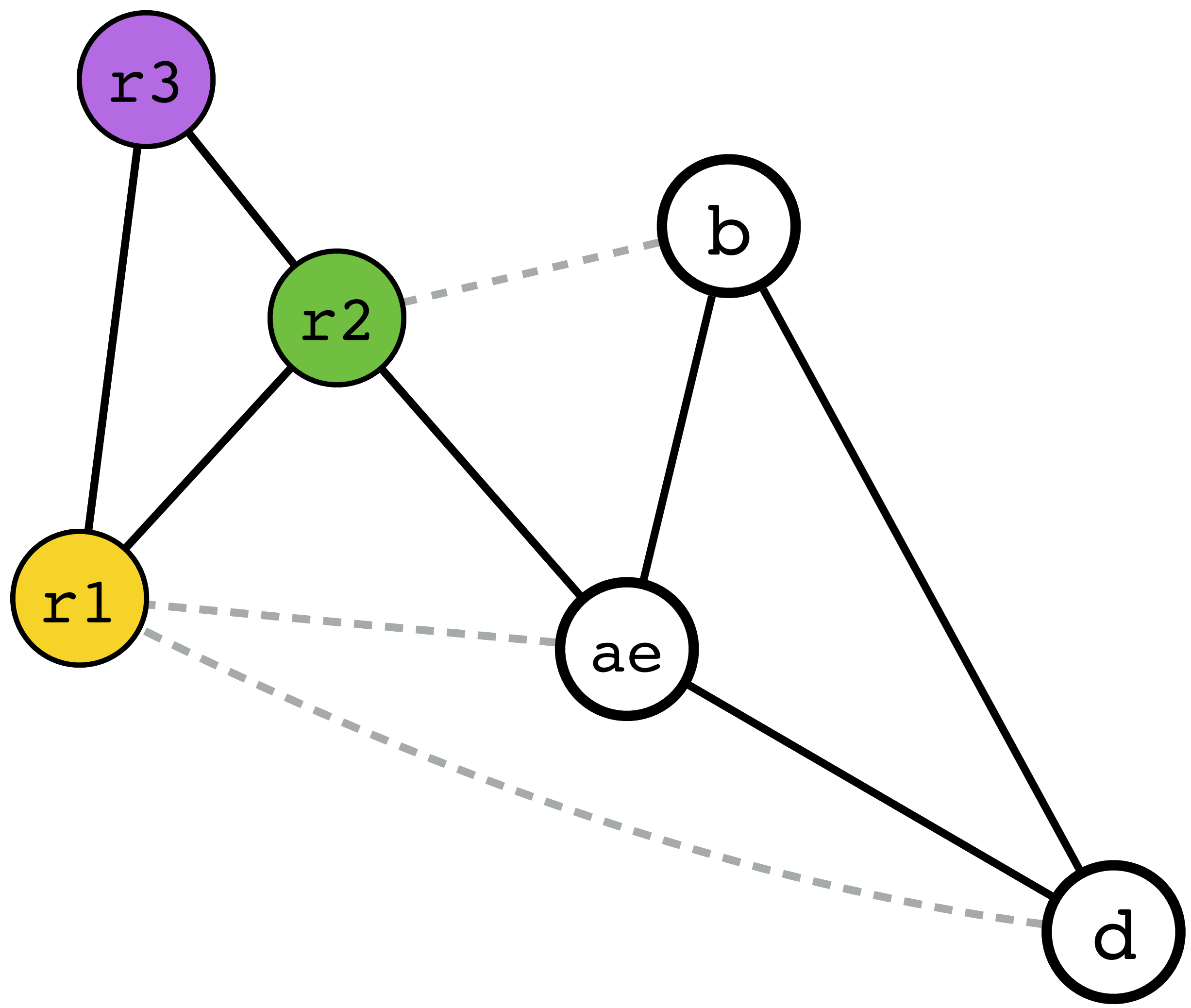




Example

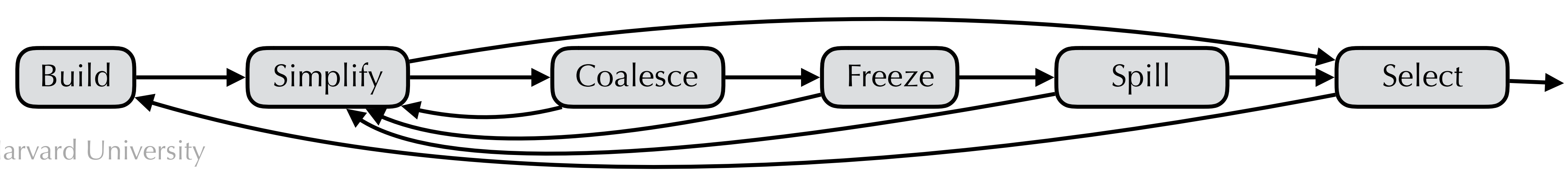
Stack:

`c` *spill?*



No simplify is possible...

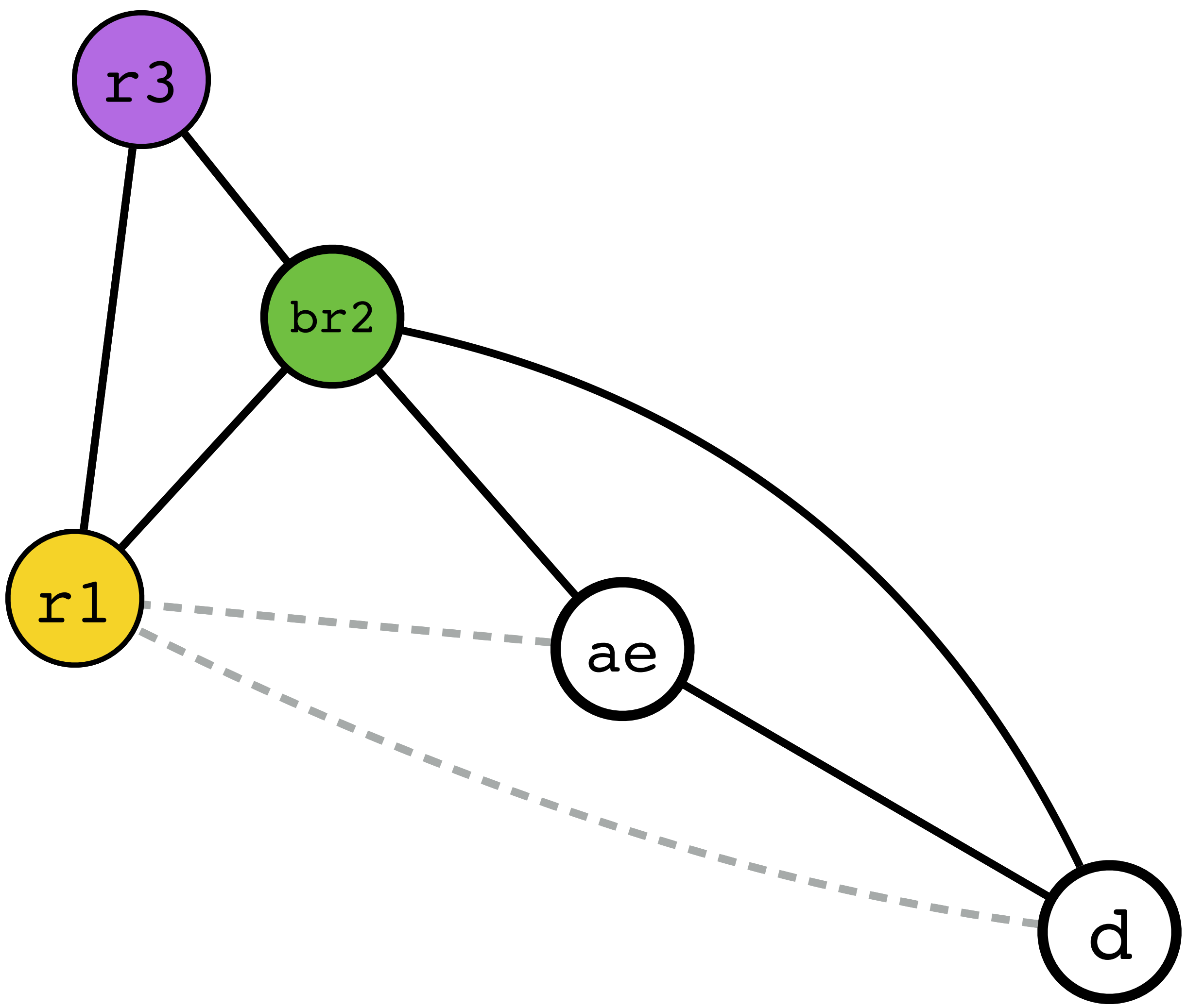
Coalesce `b` and `r2`



Example

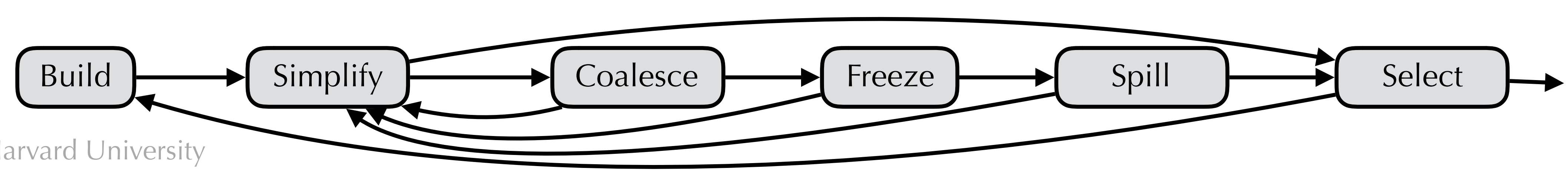
Stack:

c *spill?*



No simplify is possible...

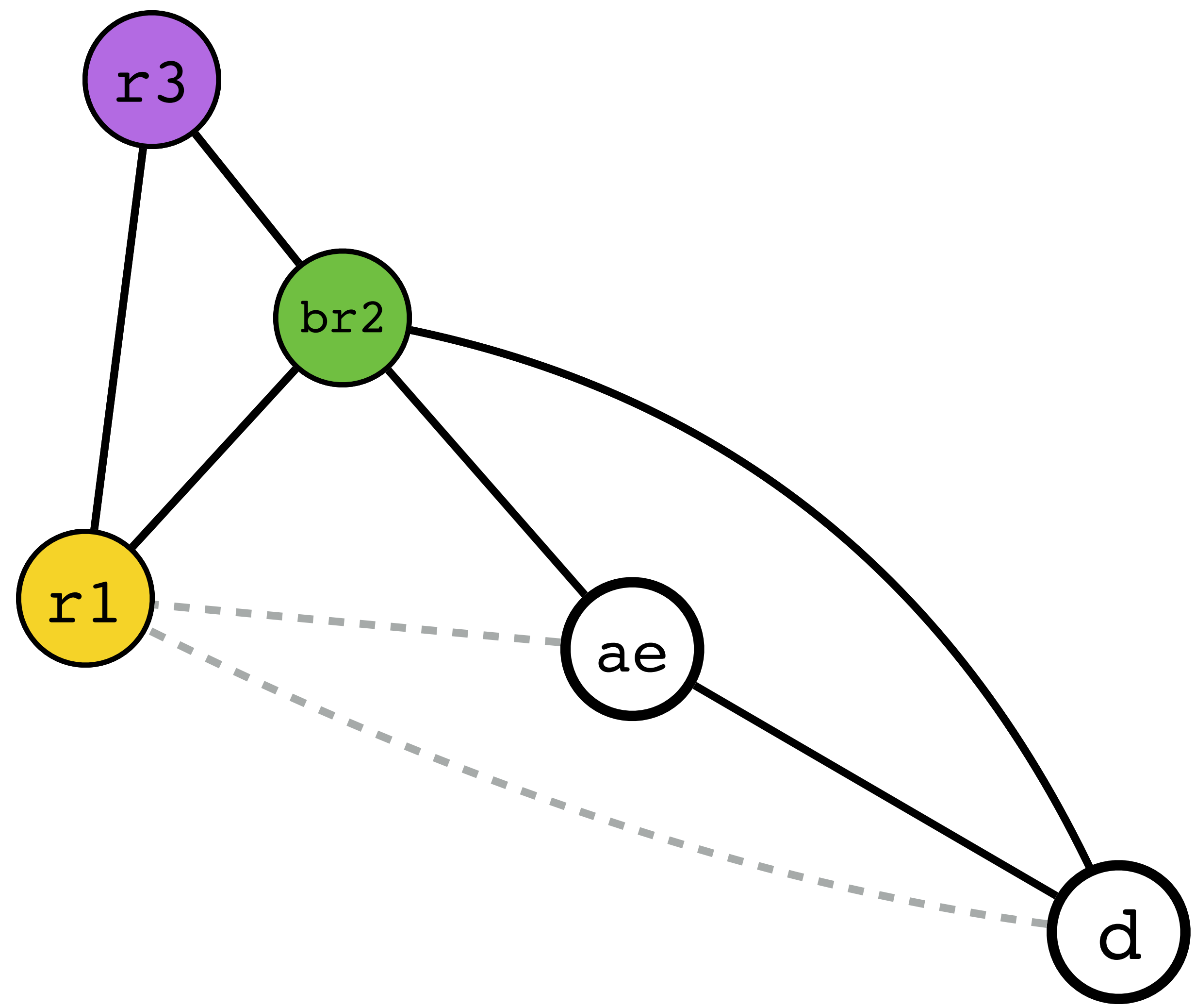
Coalesce **b** and **r2**



Example

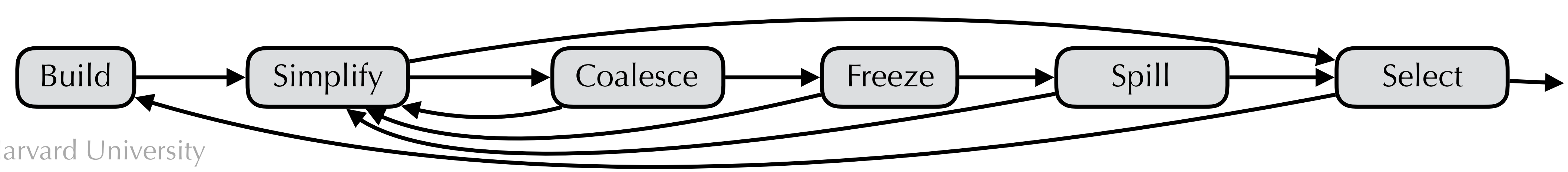
Stack:

c *spill?*



No simplify is possible...

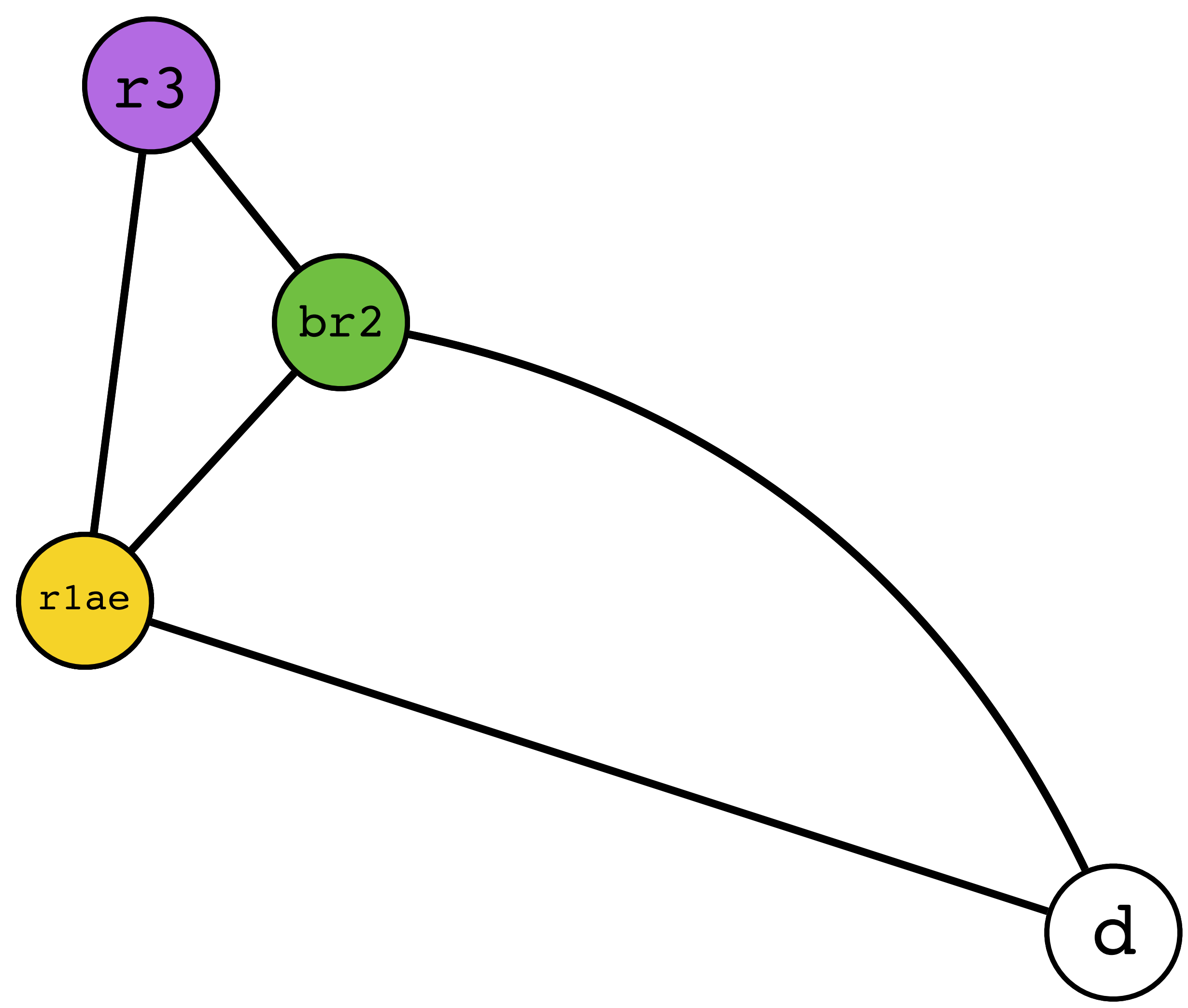
Coalesce `r1` and `ae`



Example

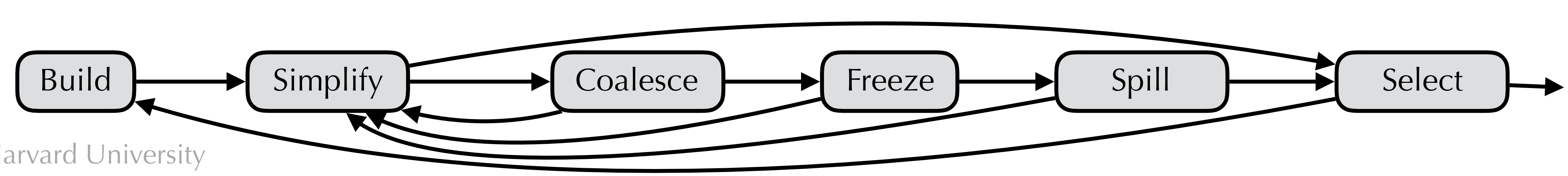
Stack:

c *spill?*



No simplify is possible...

Coalesce `r1` and `ae`

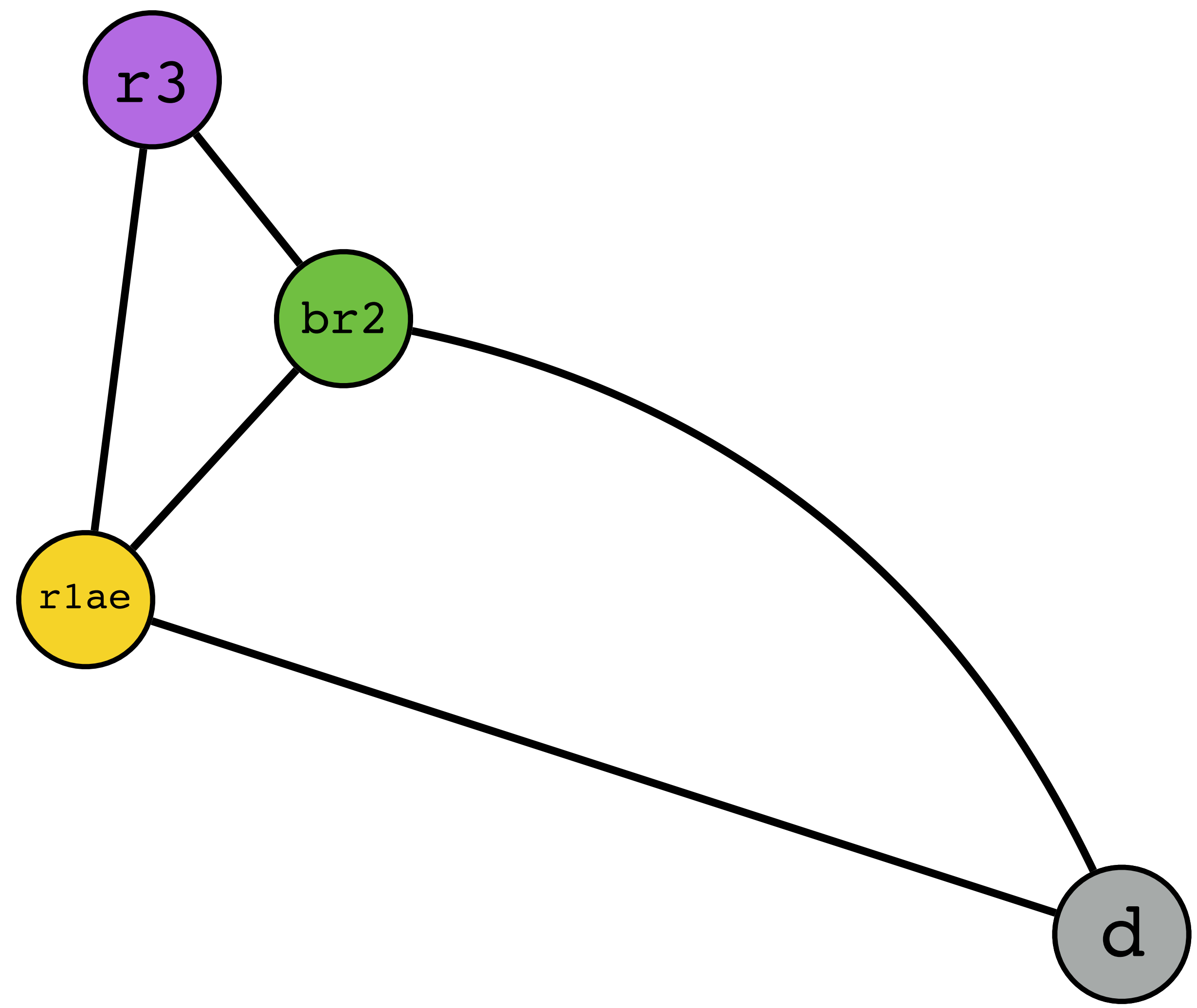


Example

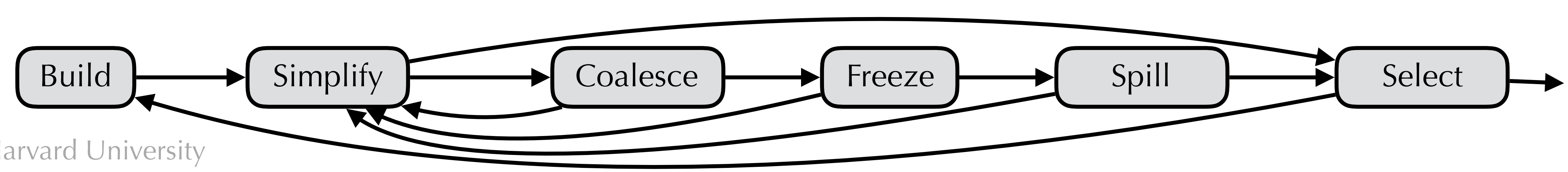
Stack:

`c` *spill?*

`d`



Simplify `d`

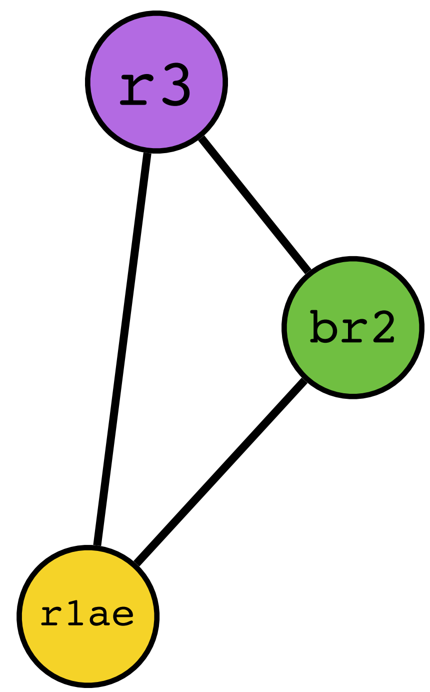


Example

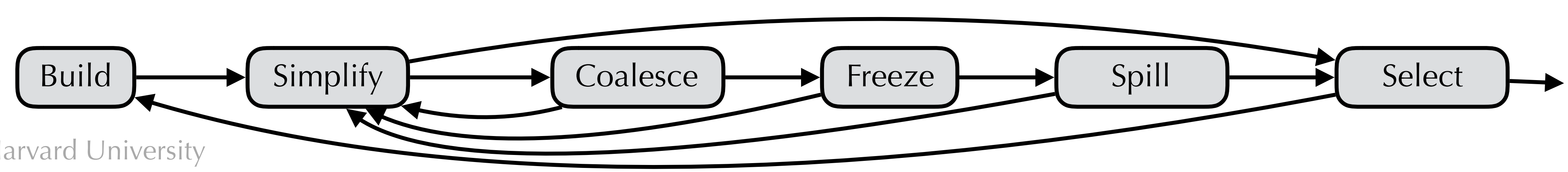
Stack:

`c` *spill?*

`d`



Only pre-colored nodes left, so start Select phase...

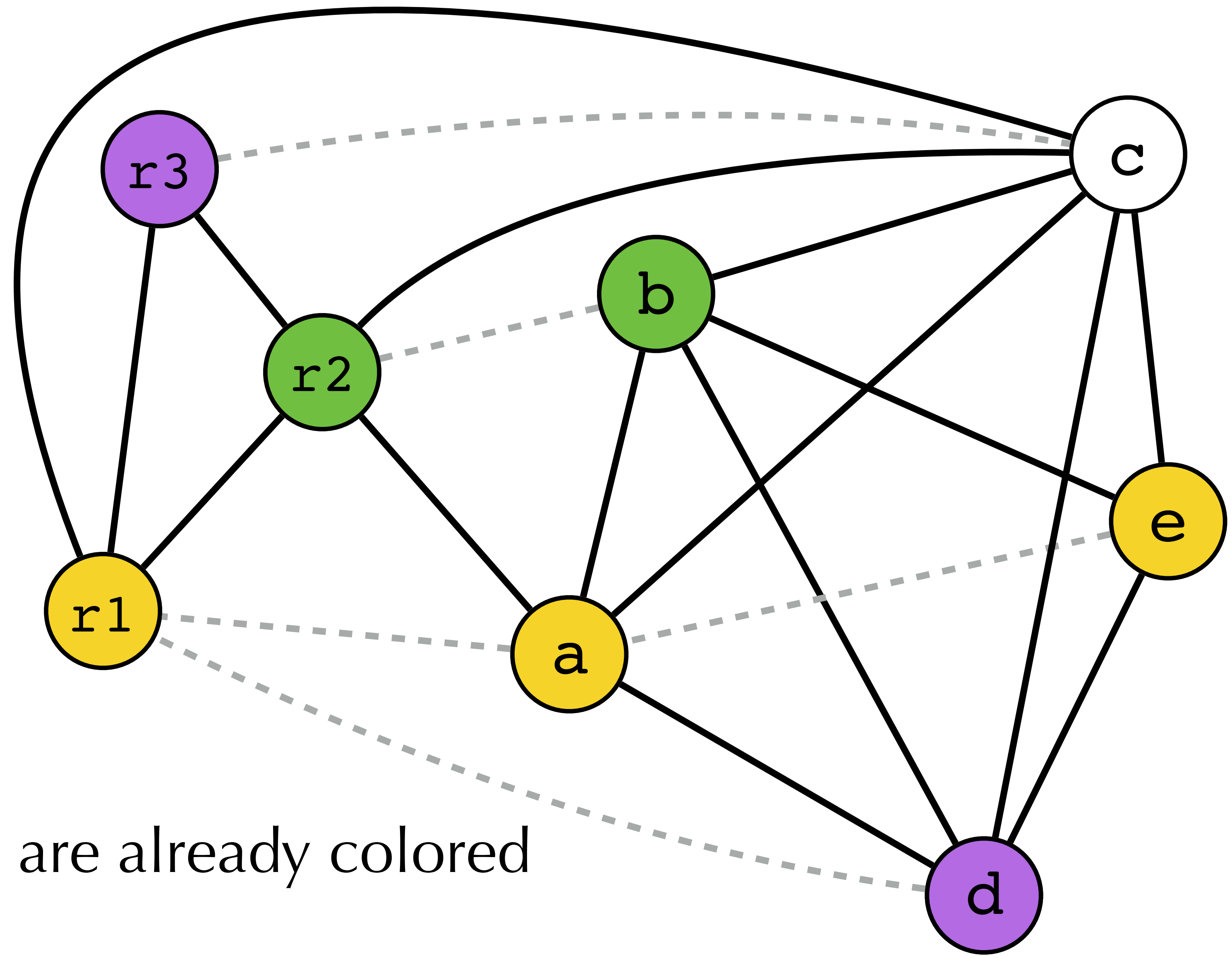


Example

Stack:

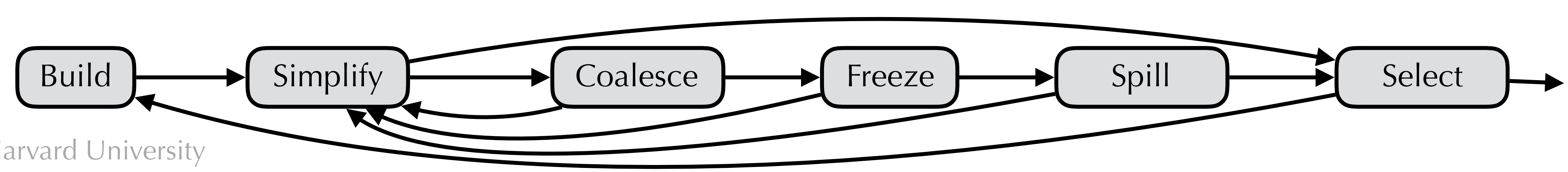
`c` *spill?*

`d`



Due to coalescing, `b`, `a`, and `e` are already colored

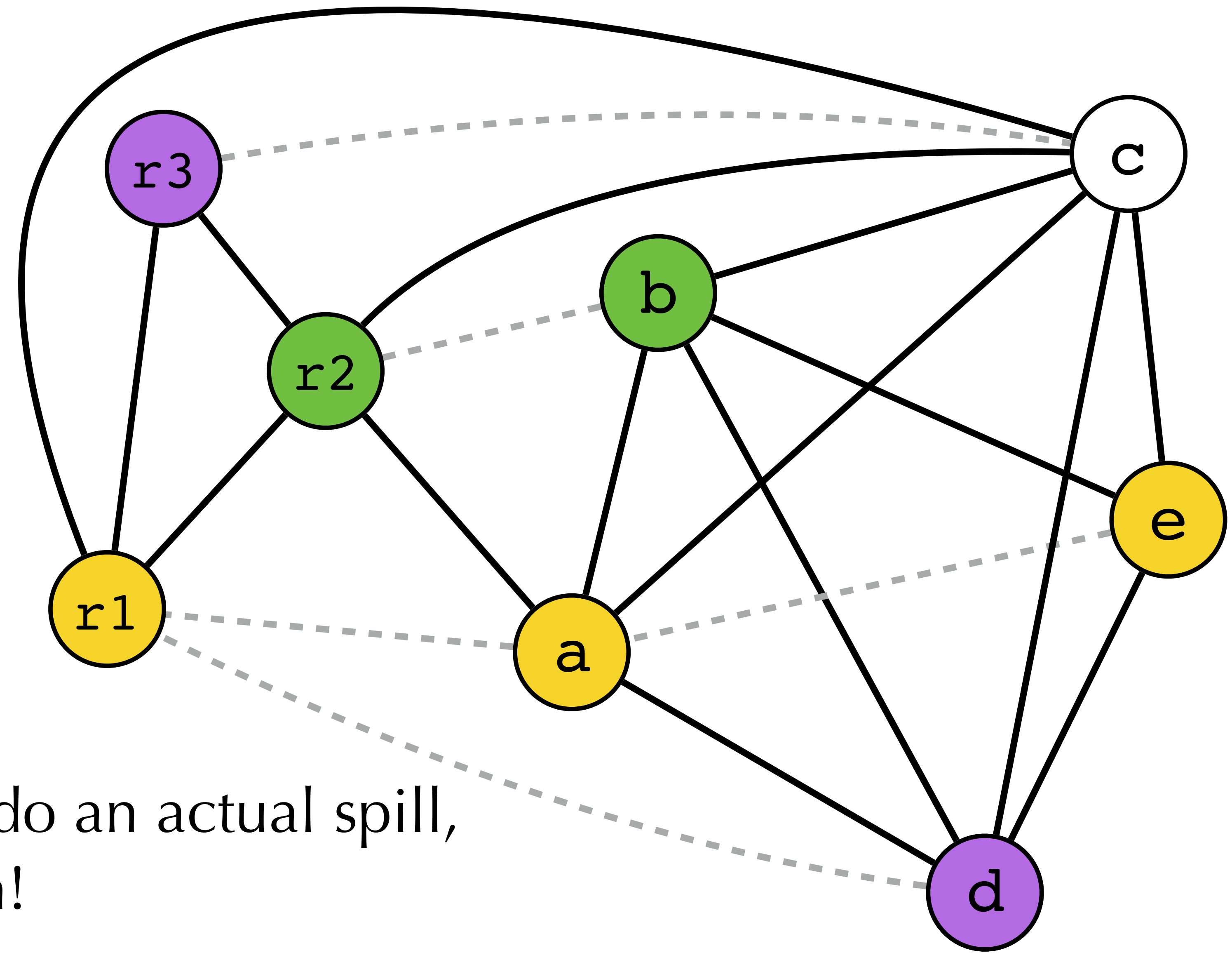
Pop `d` and color it



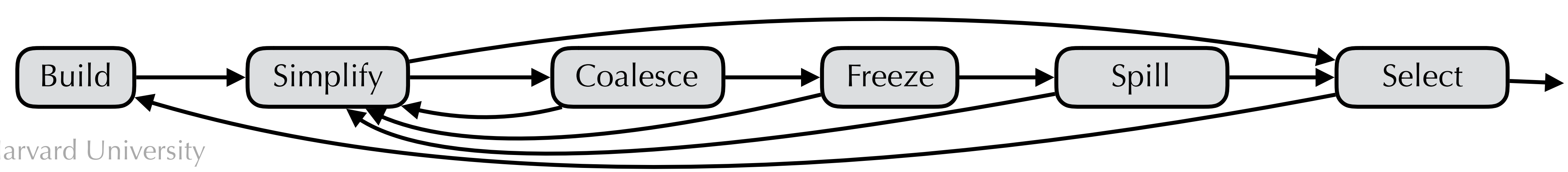
Example

Stack:

c *spill?*



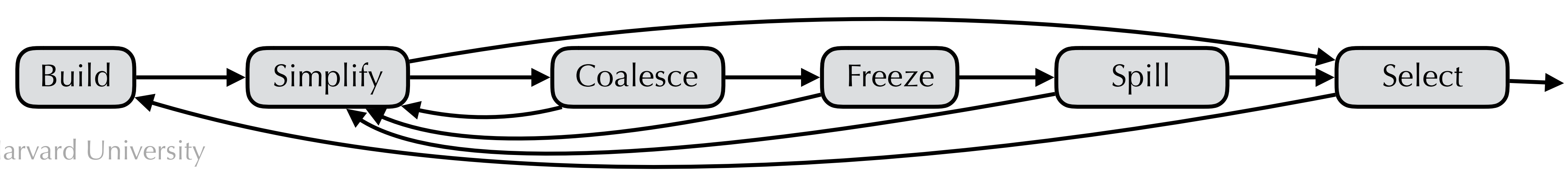
We can't color **c**, so we must do an actual spill, i.e., rewrite code and try again!



Example

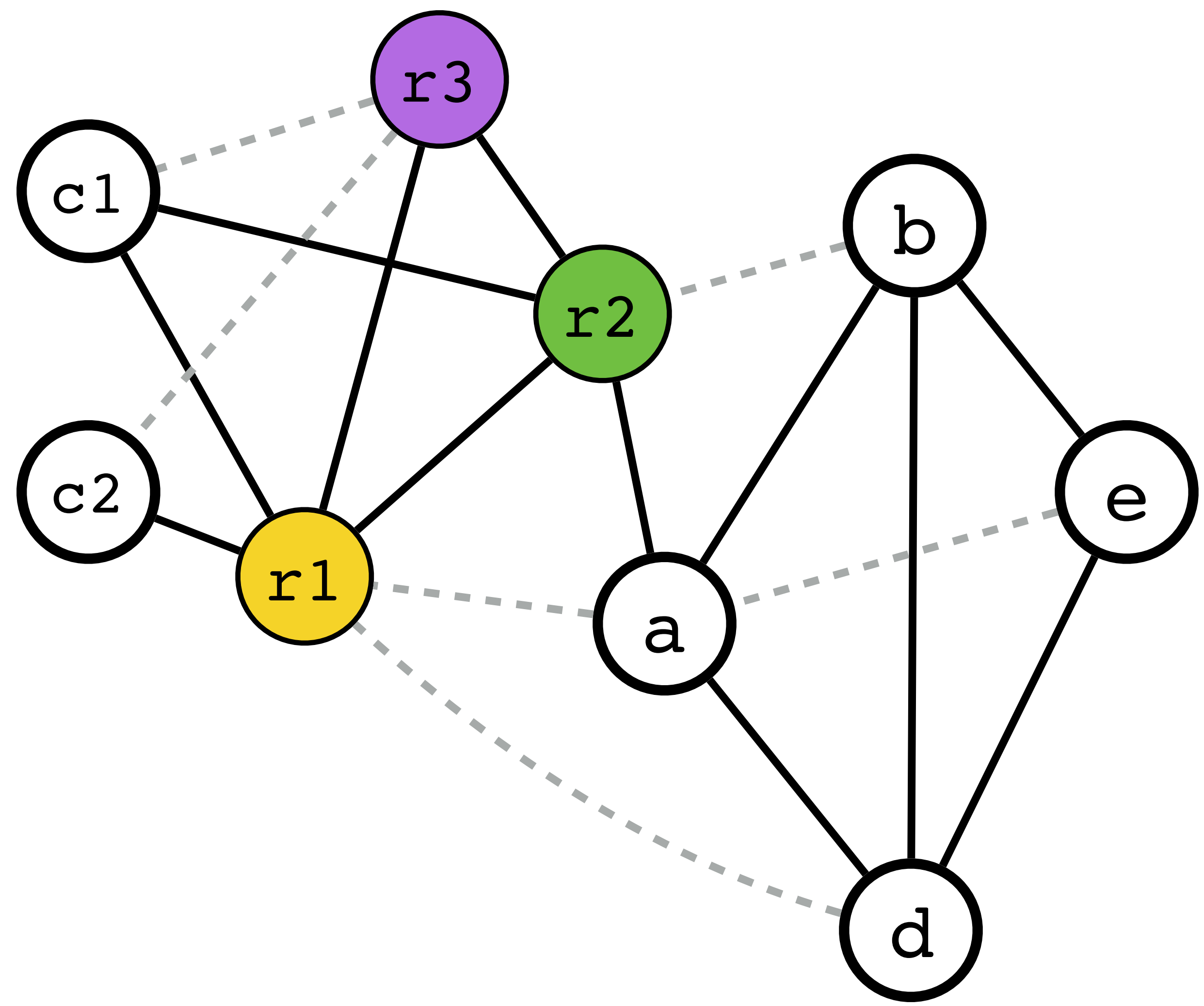
```
f: c := $r3
   a := $r1
   b := $r2
   d := 0
   e := a
loop:
   d := d + b
   e := e - 1
   if e > 0 loop else end
end:
  r1 := d
  r3 := c
  return
```

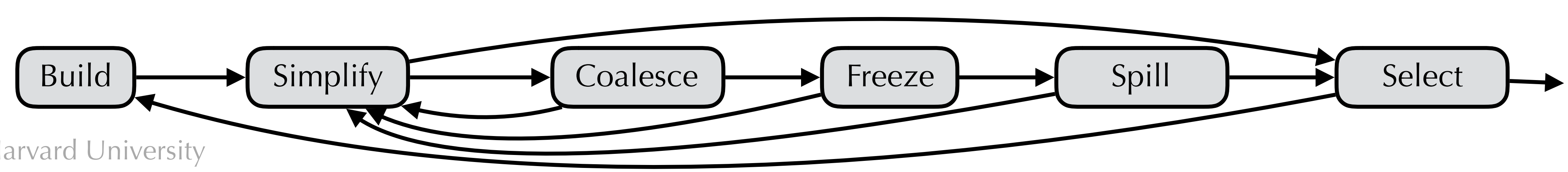
```
f: c1 := $r3
   Mem[fp+i] := c1
   a := $r1
   b := $r2
   d := 0
   e := a
loop:
   d := d + b
   e := e - 1
   if e > 0 loop else end
end:
  r1 := d
  c2 := Mem[fp+i]
  r3 := c2
  return
```



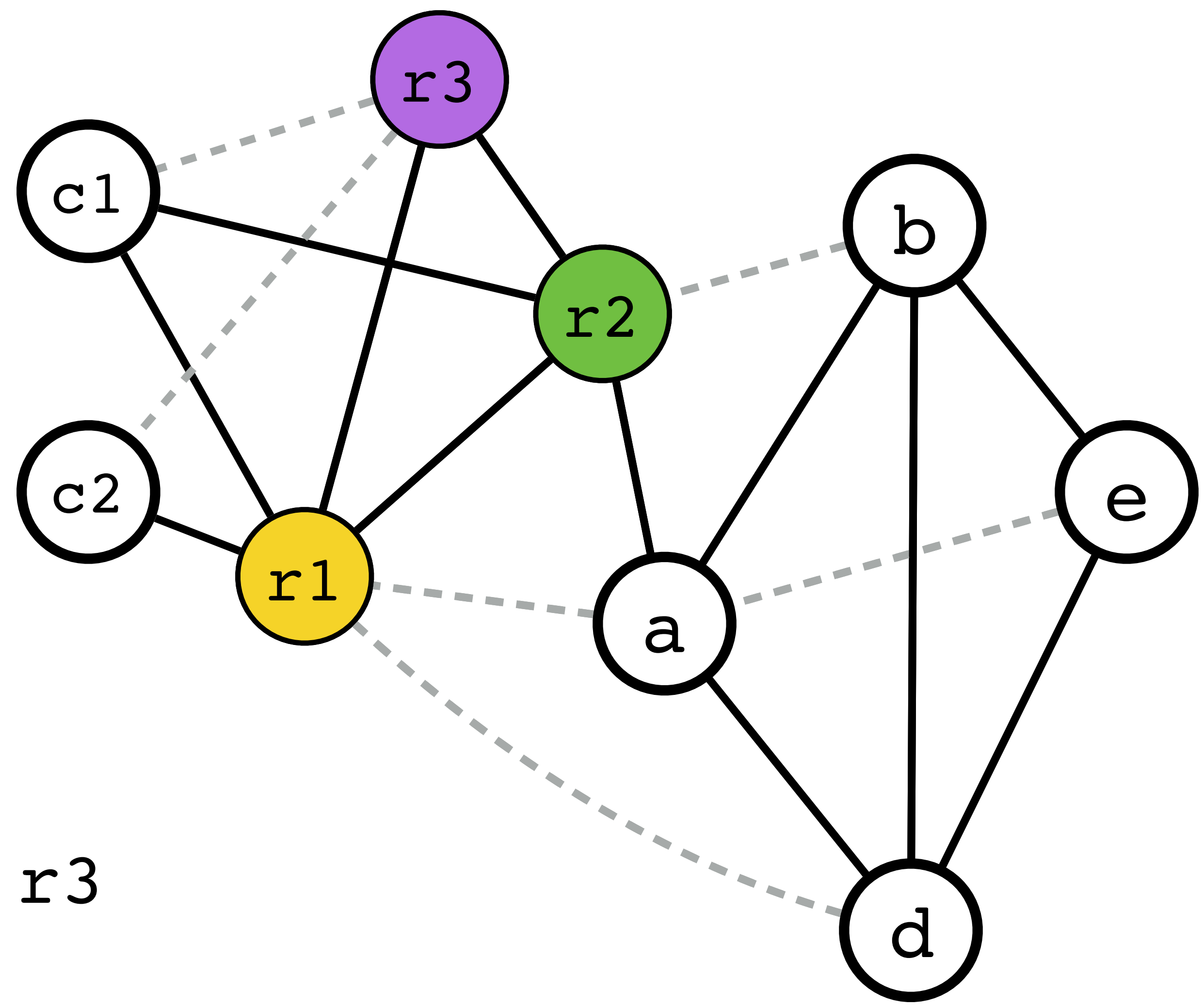
Example

```
f: c1 := $r3
   Mem[fp+i] := c1
   a := $r1
   b := $r2
   d := 0
   e := a
loop:
   d := d + b
   e := e - 1
   if e > 0 loop else end
end:
  r1 := d
  c2 := Mem[fp+i]
  r3 := c2
  return
```

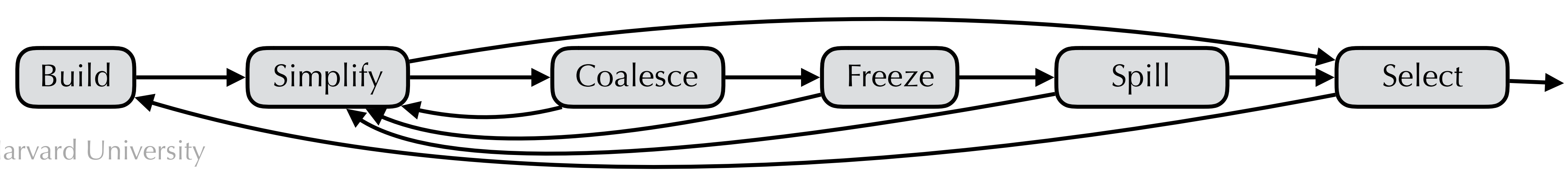




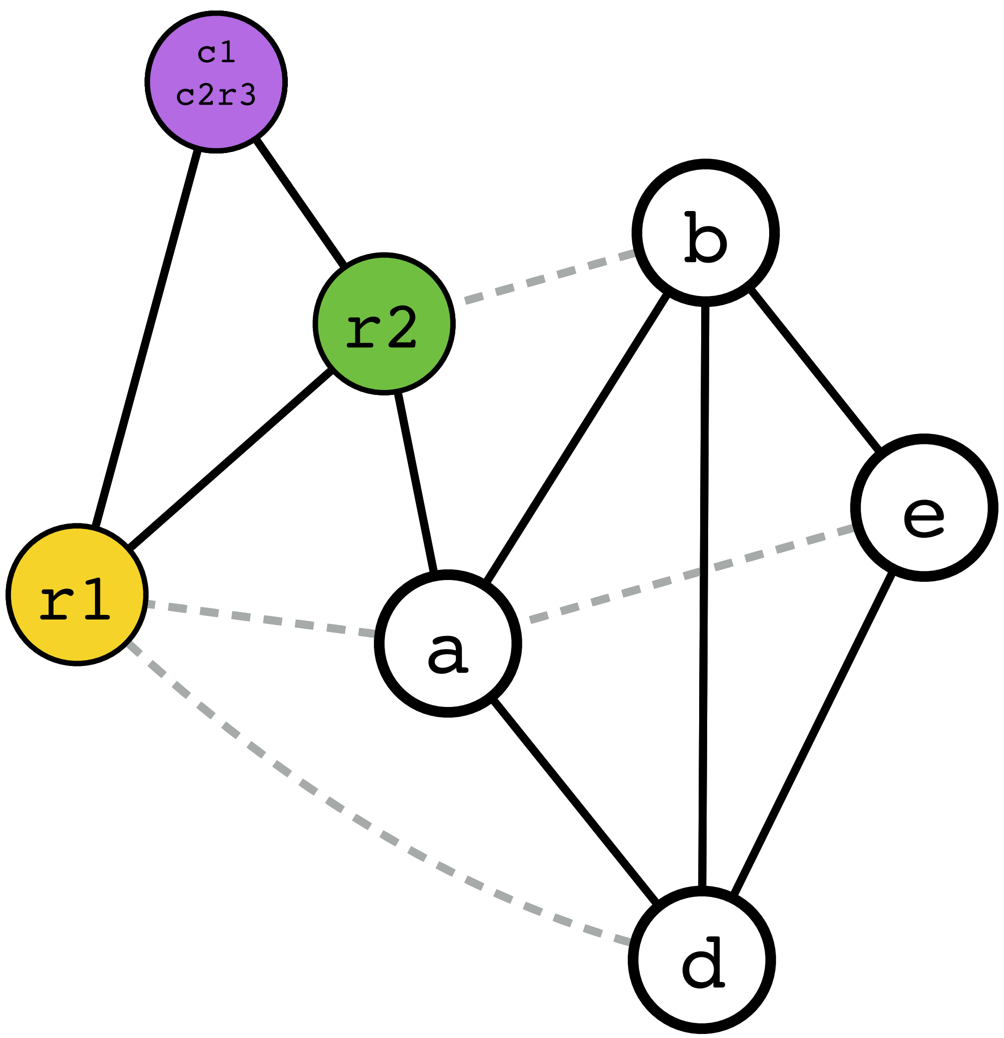
Example



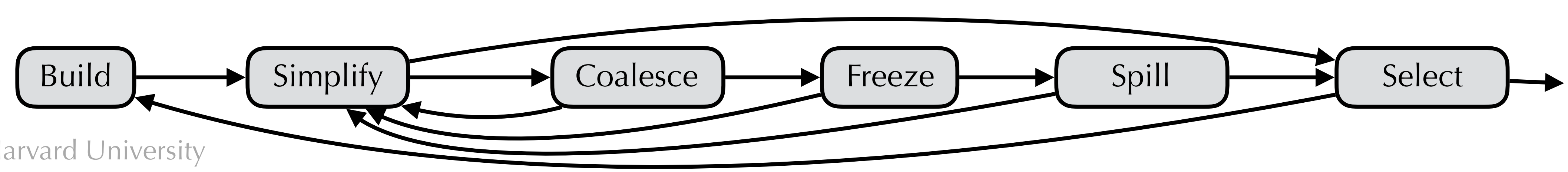
Coalesce `c1` and `r3`, and then `c2` and `r3`



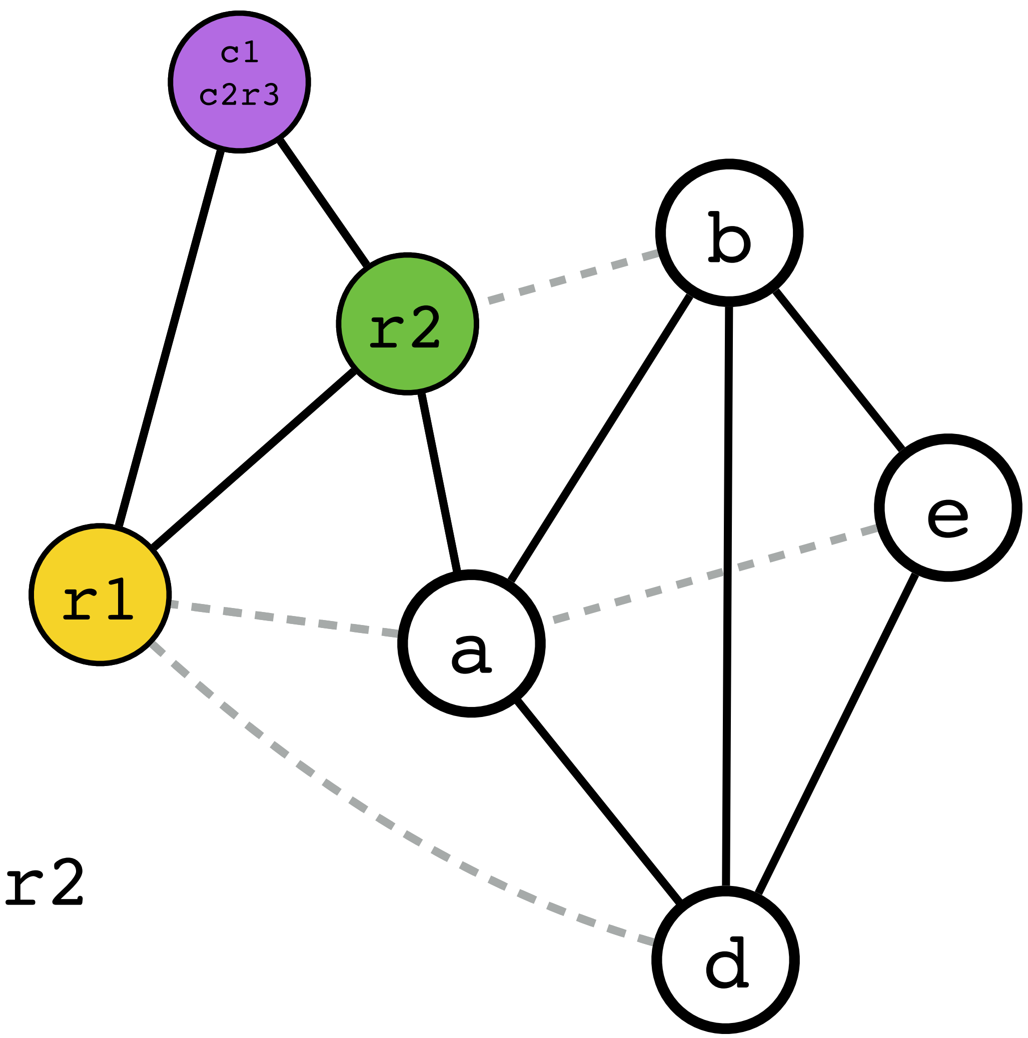
Example



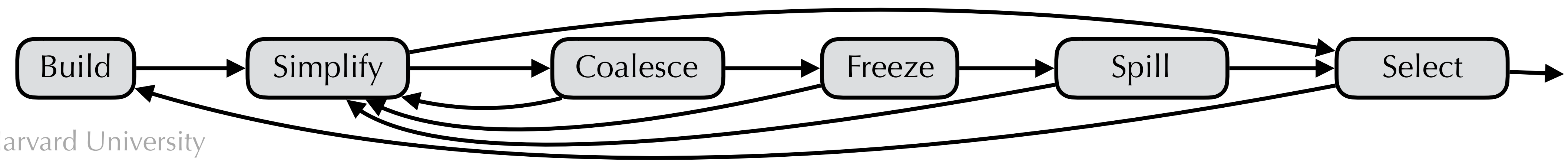
Coalesce `c1` and `r3`, and then `c2` and `r3`



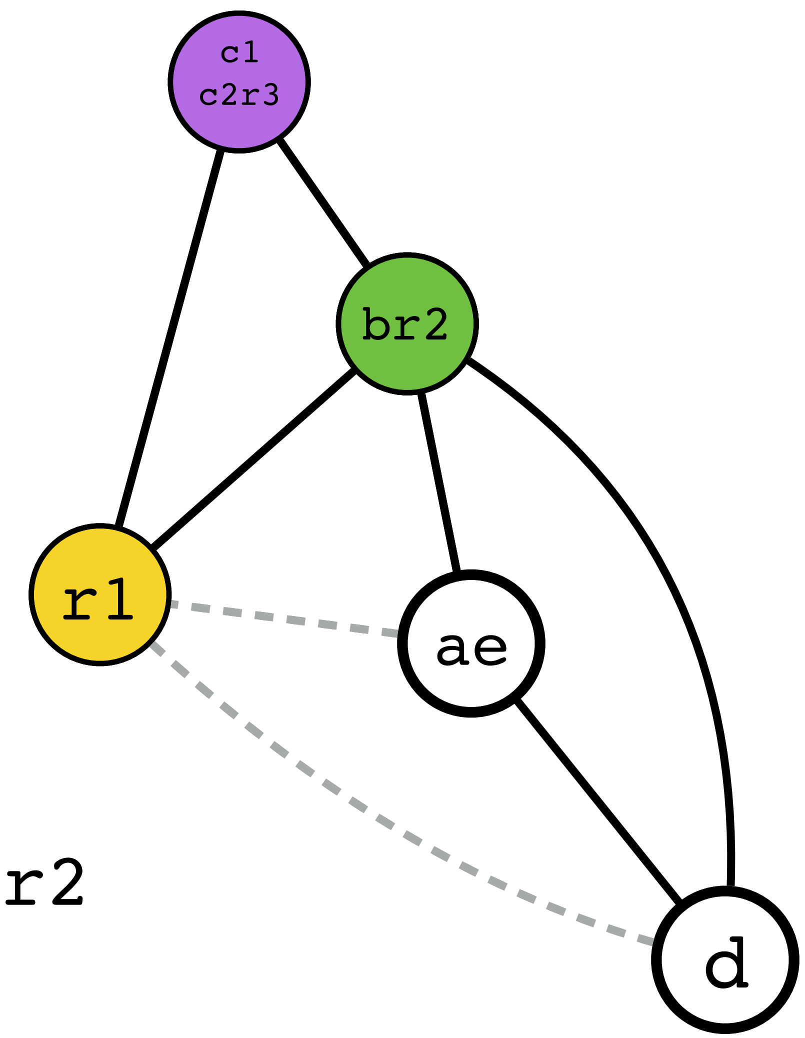
Example



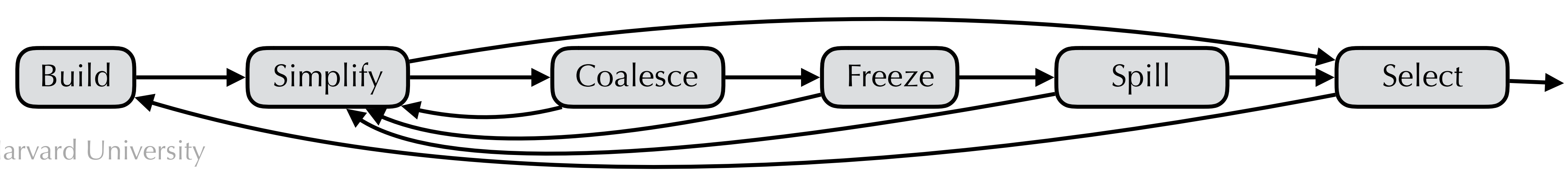
As before, coalesce `a` and `e`, and then `b` and `r2`



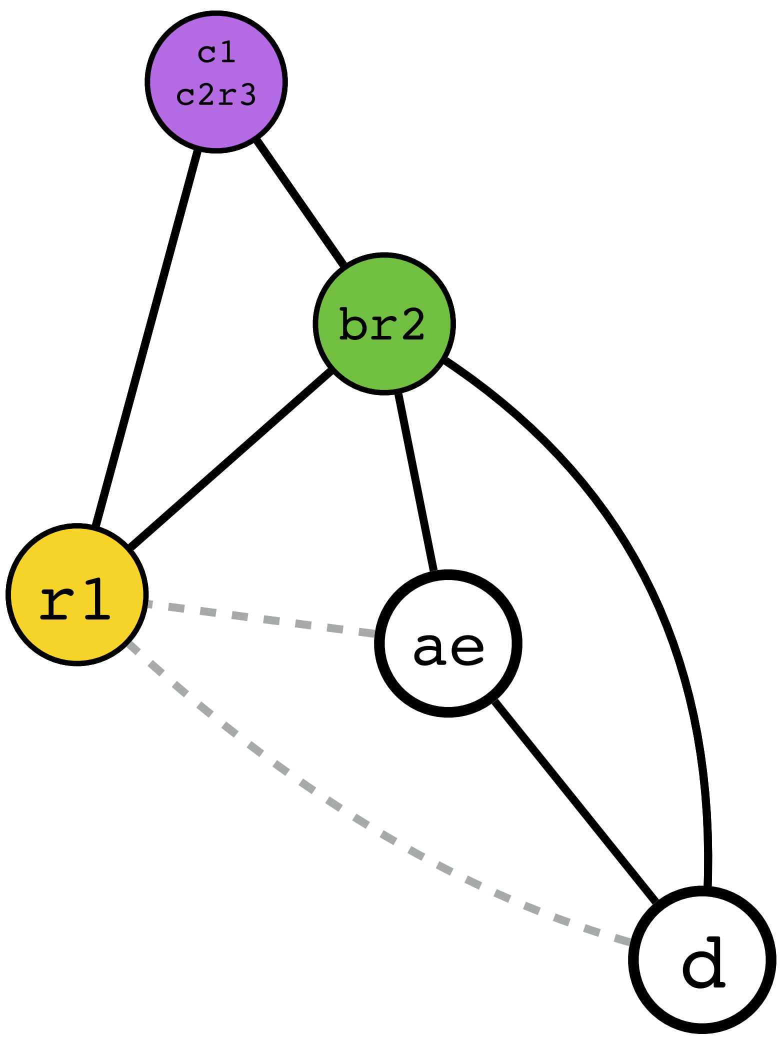
Example



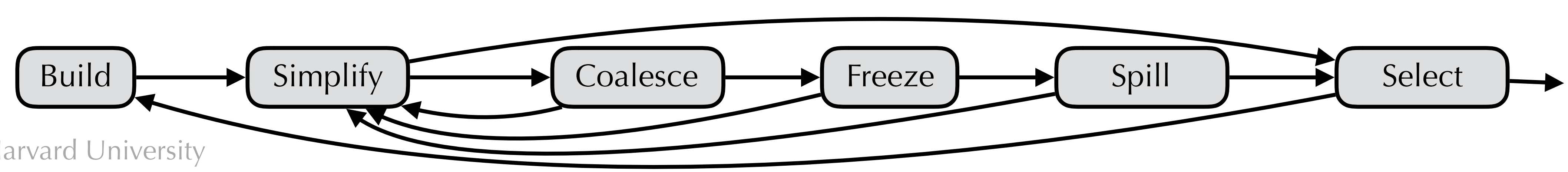
As before, coalesce `a` and `e`, and then `b` and `r2`



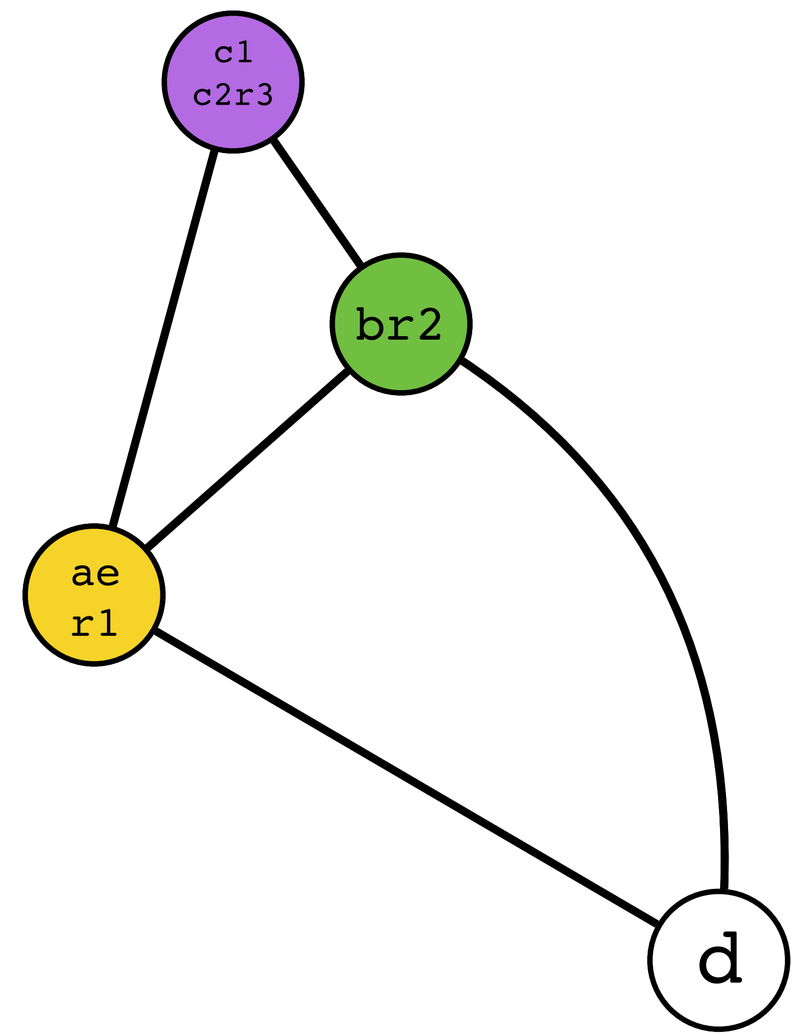
Example



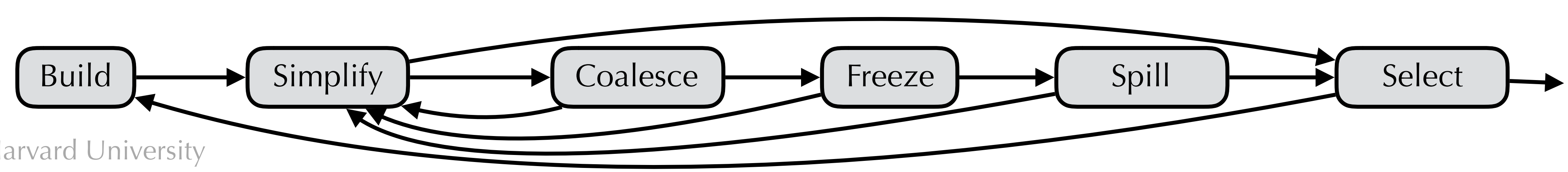
As before, coalesce **`ae`** and **`r1`**



Example



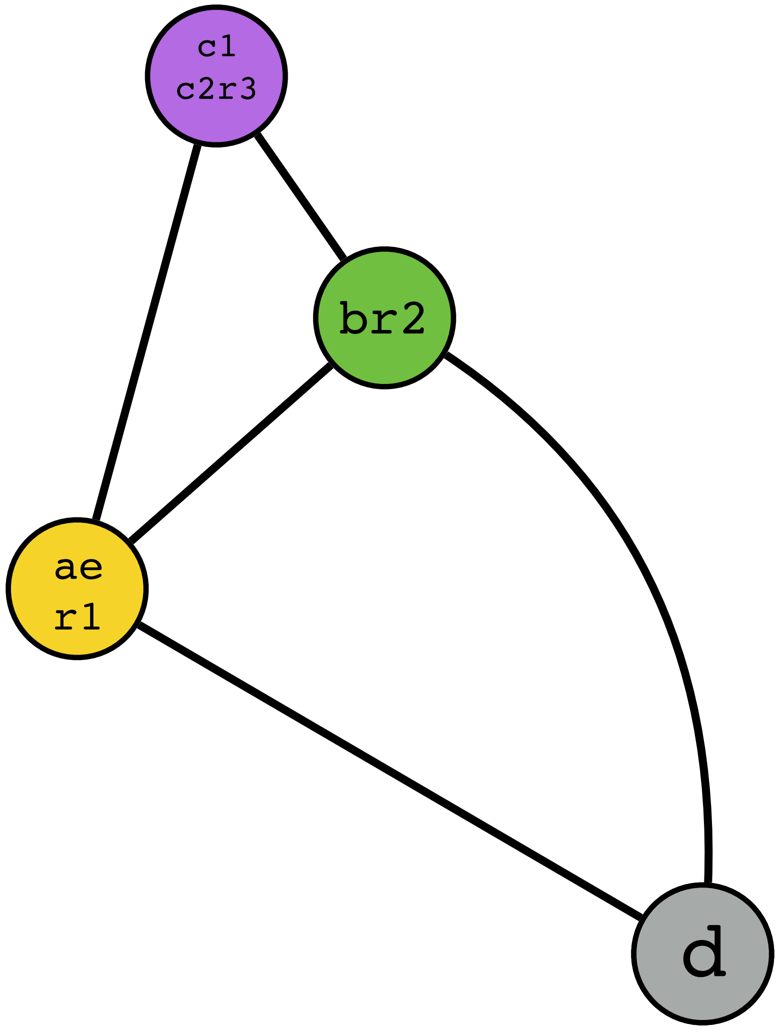
As before, coalesce `ae` and `r1`



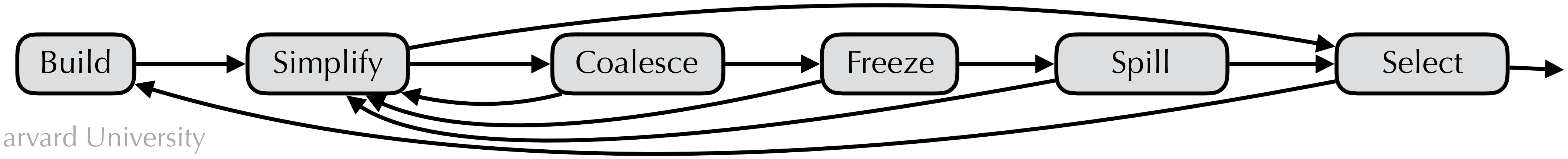
Example

Stack:

`d`



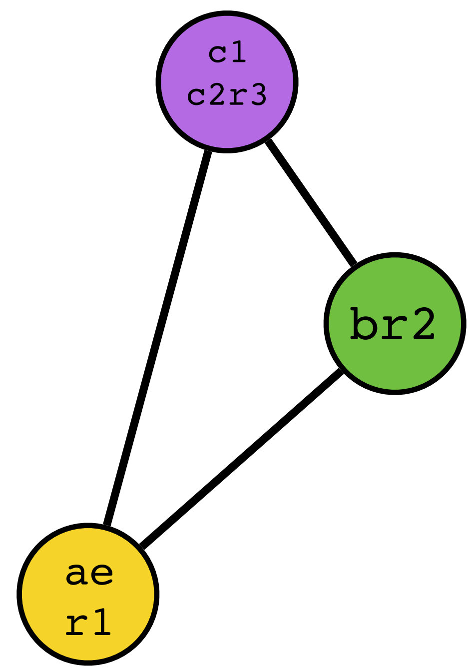
Simplify `d`



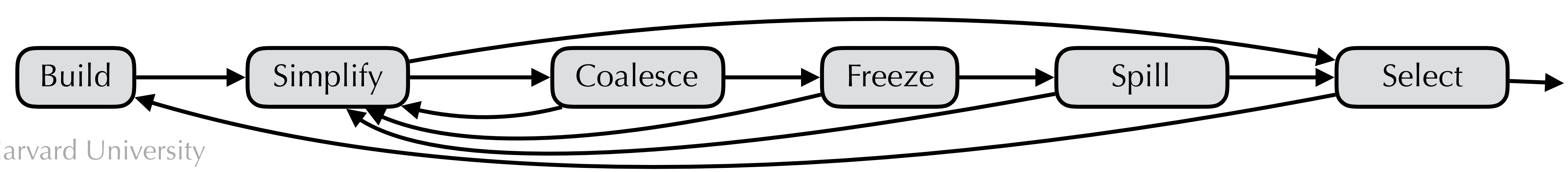
Example

Stack:

d



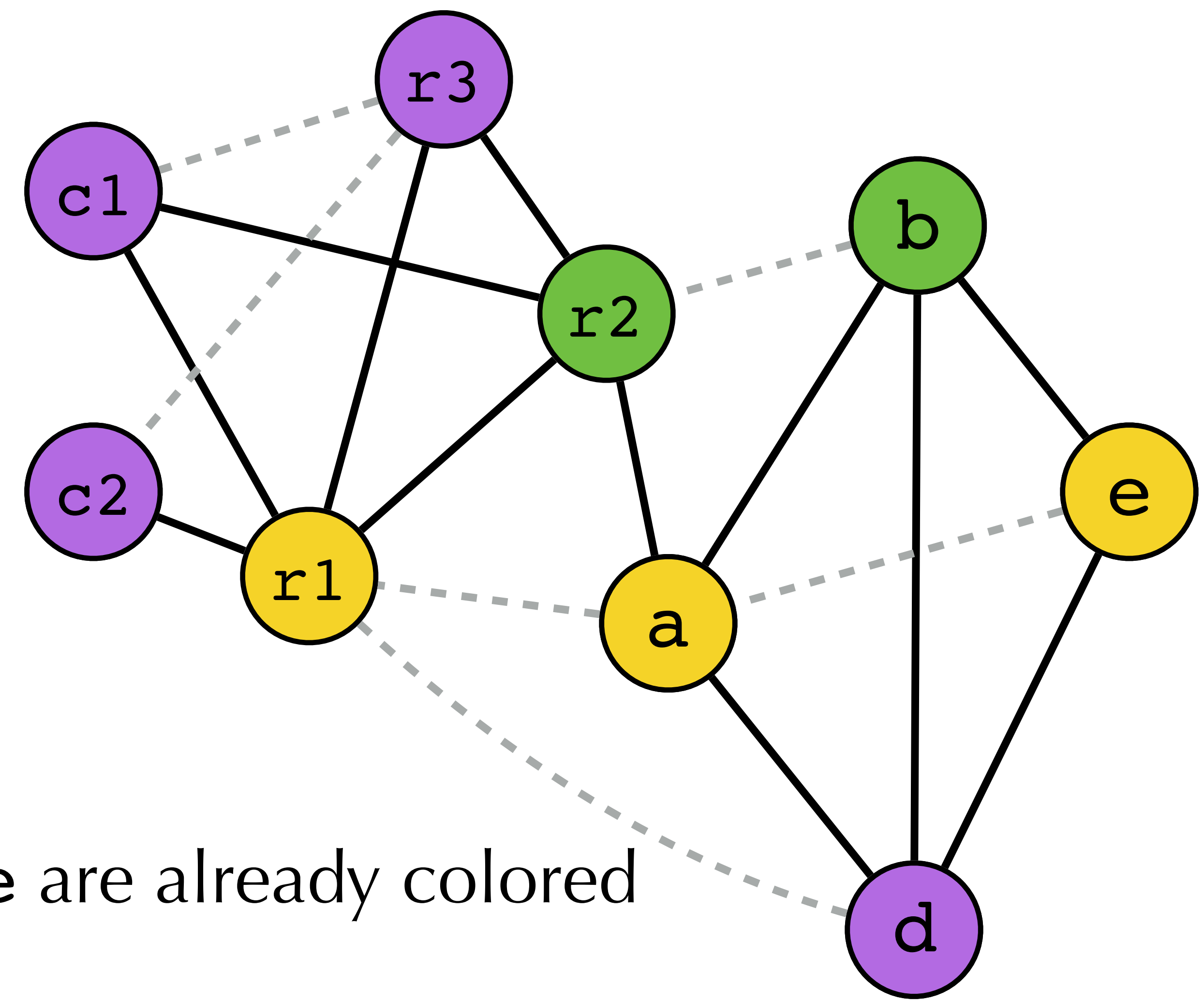
Only pre-colored nodes left, we're ready to move to Select phase!



Example

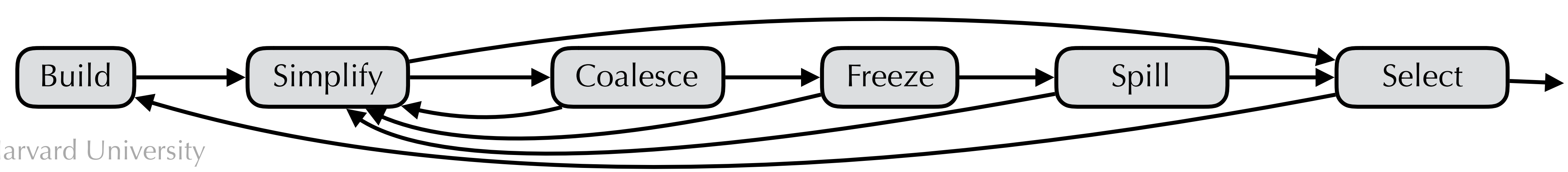
Stack:

`d`



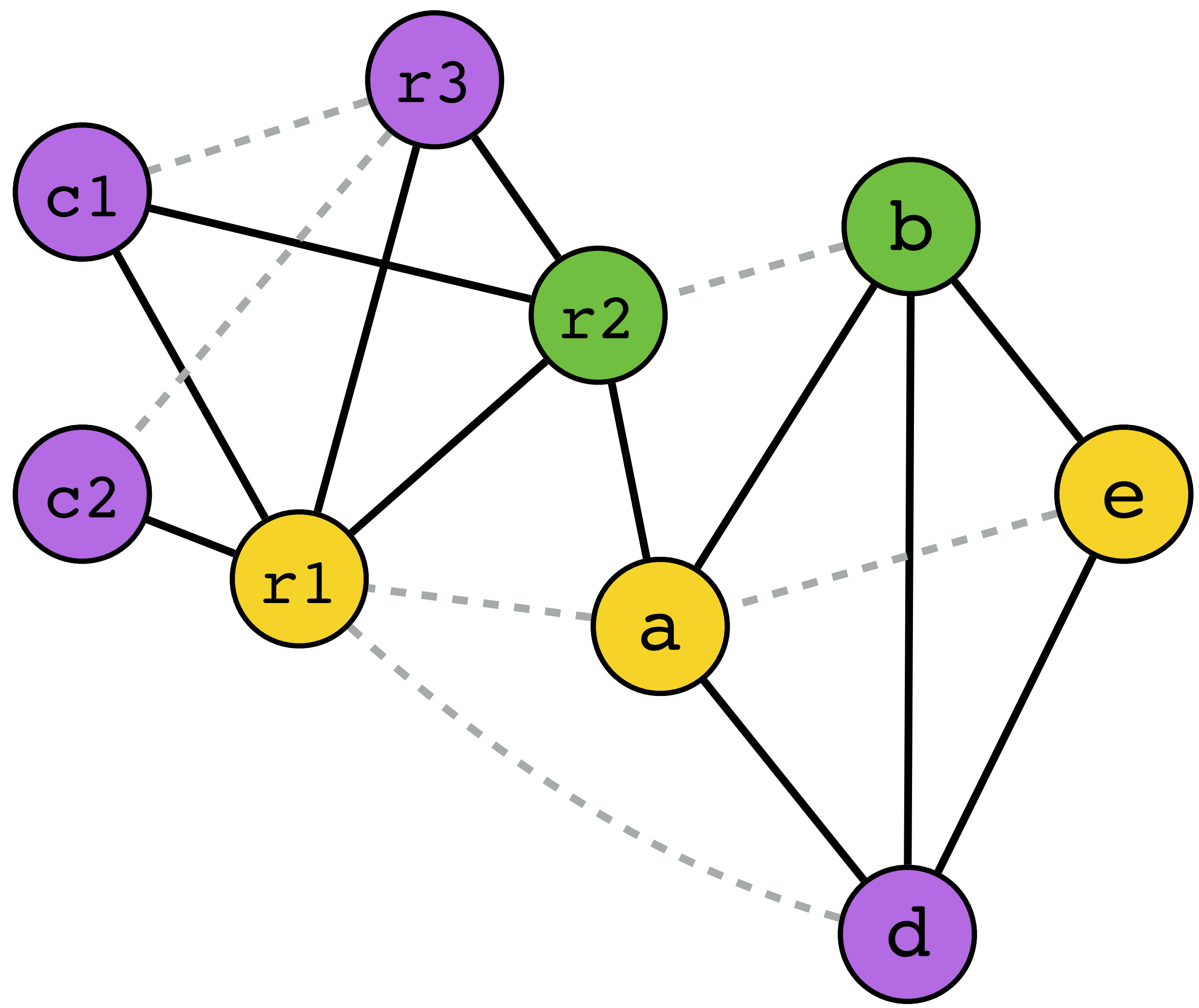
Due to coalescing, `c1`, `c2`, `b`, `a`, and `e` are already colored

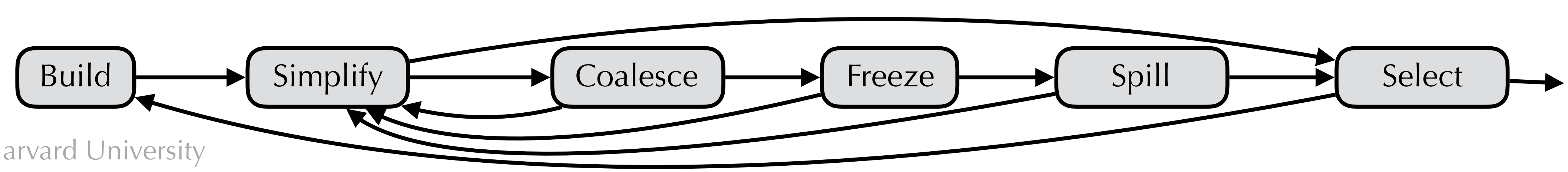
Pop `d` and color



Example

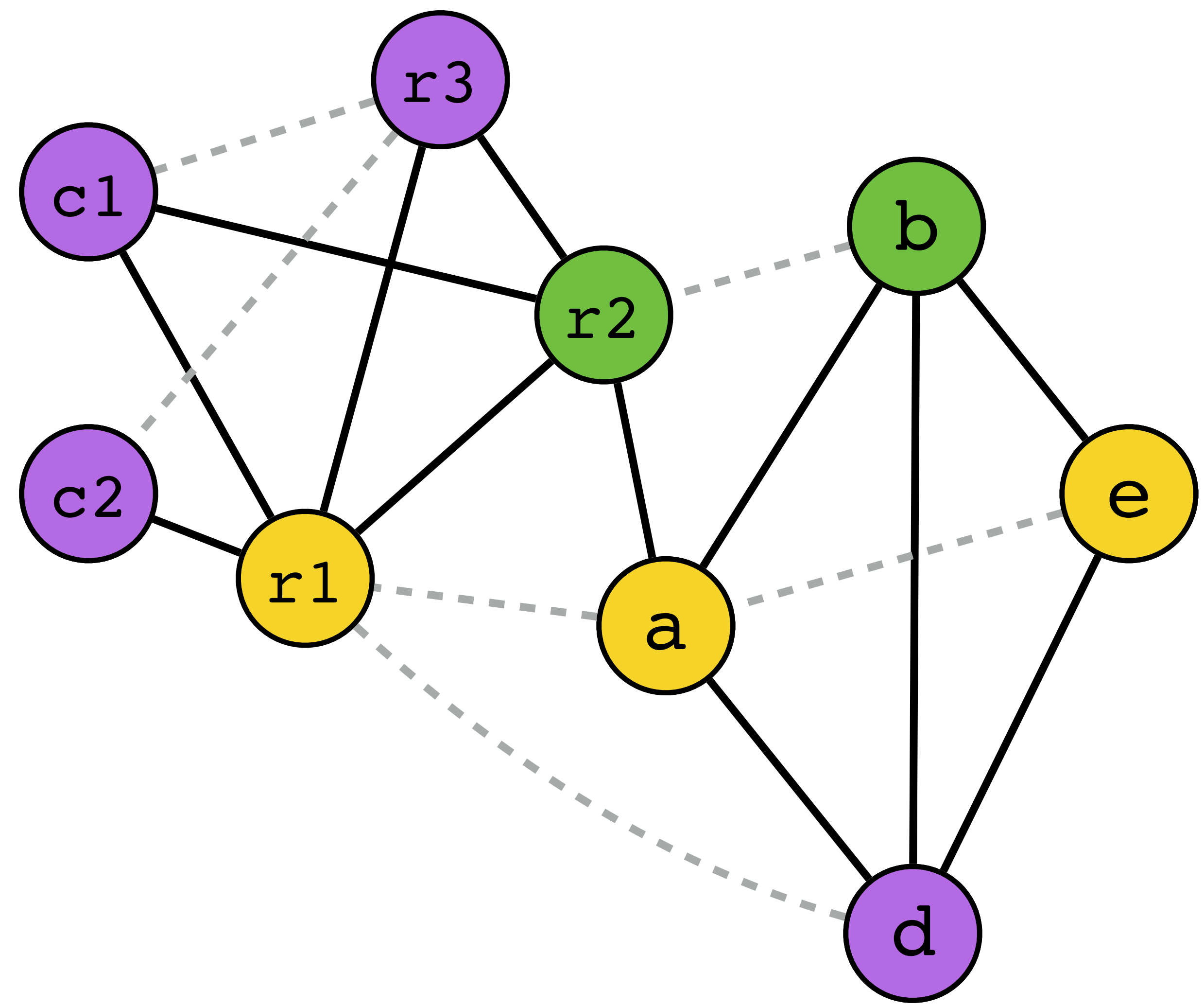
```
f: c1 := $r3
   Mem[bp+i] := c1
   a := $r1
   b := $r2
   d := 0
   e := a
loop:
   d := d + b
   e := e - 1
   if e > 0 loop else end
end:
   r1 := d
   c2 := Mem[bp+i]
   r3 := c2
   return
```

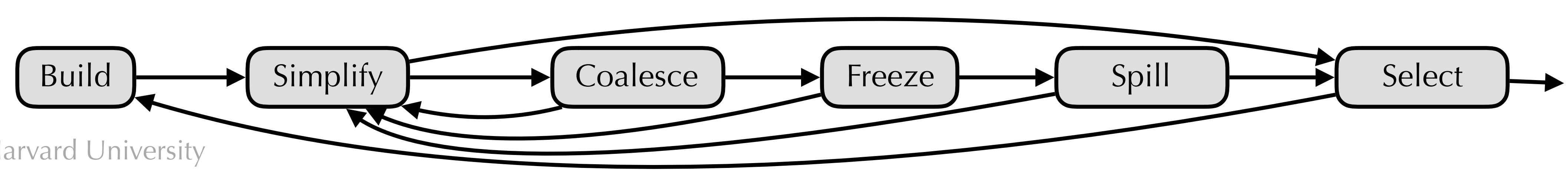




Example

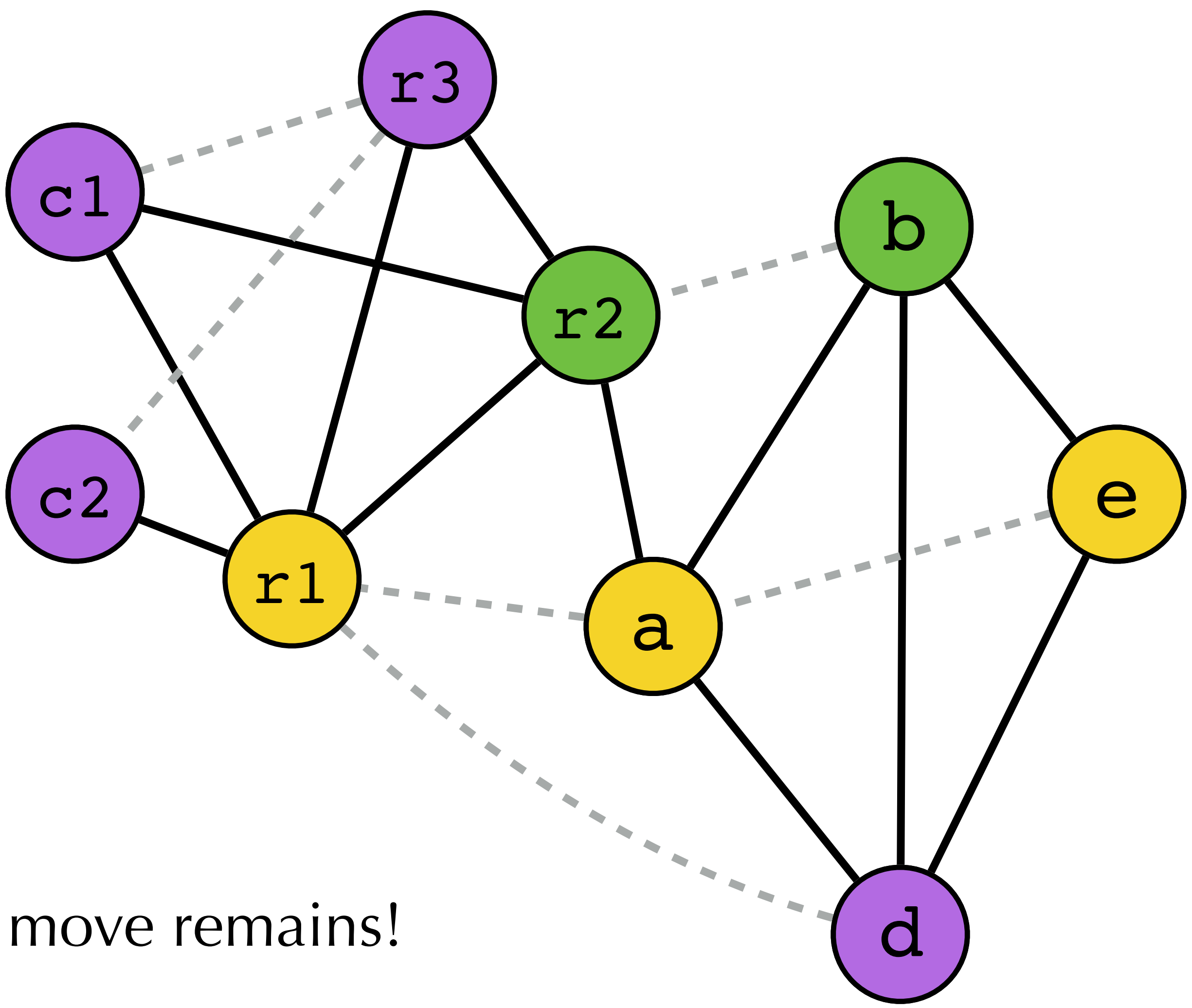
```
f: $r3 := $r3
   Mem[bp+i] := $r3
   $r1 := $r1
   $r2 := $r2
   $r3 := 0
   $r1 := $r1
loop:
   $r3 := $r3 + $r2
   $r1 := $r1 - 1
   if $r1 > 0 loop else end
end:
   $r1 := $r3
   $r3 := Mem[bp+i]
   $r3 := $r3
   return
```





Example

```
f: Mem[bp+i] := $r3
   $r3 := 0
loop:
   $r3 := $r3 + $r2
   $r1 := $r1 - 1
   if $r1 > 0 loop else end
end:
   $r1 := $r3
   $r3 := Mem[bp+i]
   return
```



Only one non-coalesced move remains!