CS153: Compilers
Lecture 21:
Register Allocation II

Stephen Chong

https://www.seas.harvard.edu/courses/cs153
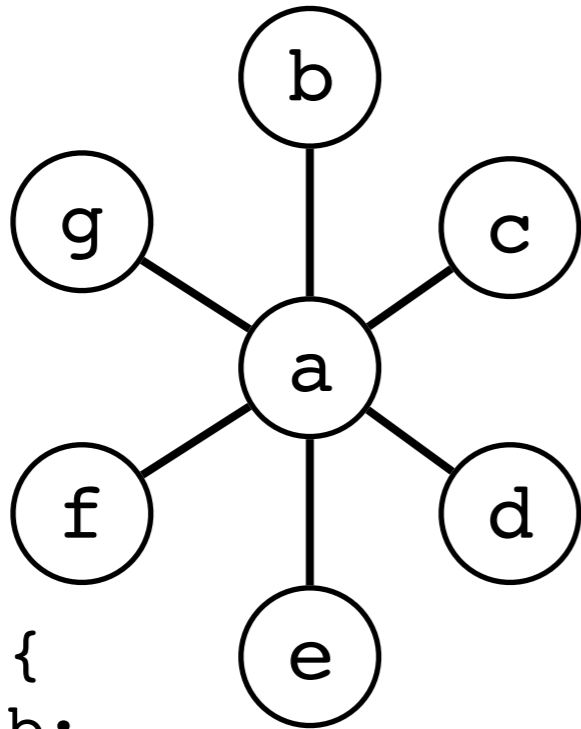
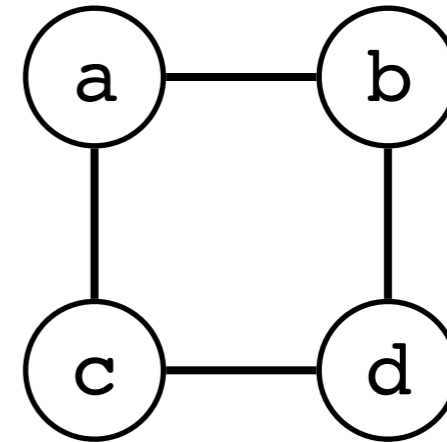# Pre-class Puzzle

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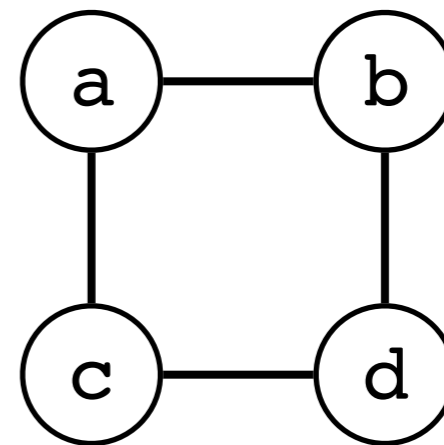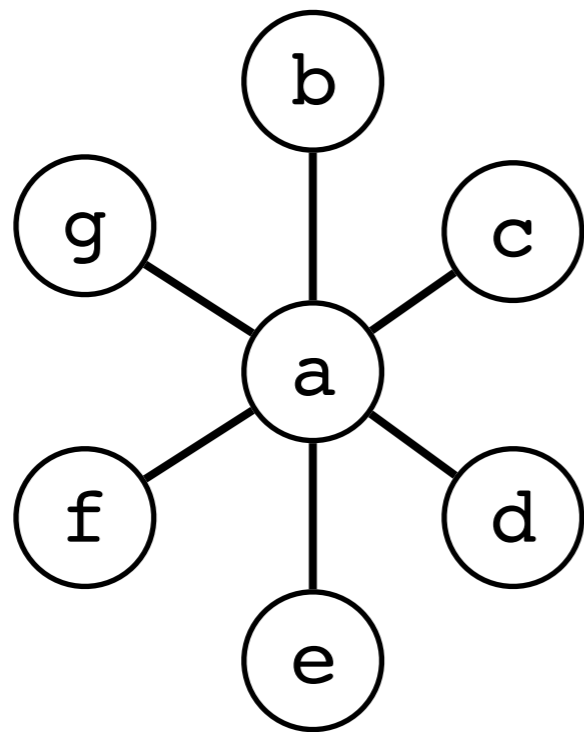
Pre-class Puzzle


```
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

- Project 5 due today!
- Project 6 out
  - Due Tuesday Nov 20 (7 days)
- Project 7 out
  - Due Thursday Nov 29 (16 days)
- Project 8 will be released today
  - Due Saturday Dec 8 (25 days)

Today

- Register allocation continued
  - Coalescing
  - Coloring with coalescing
    - Pre-colored nodes to handle callee-save, caller-save, and special purpose registers

Recall Last Lecture's Algorithm

| Build | → | Simplify | → | Spill | → | Select | → |
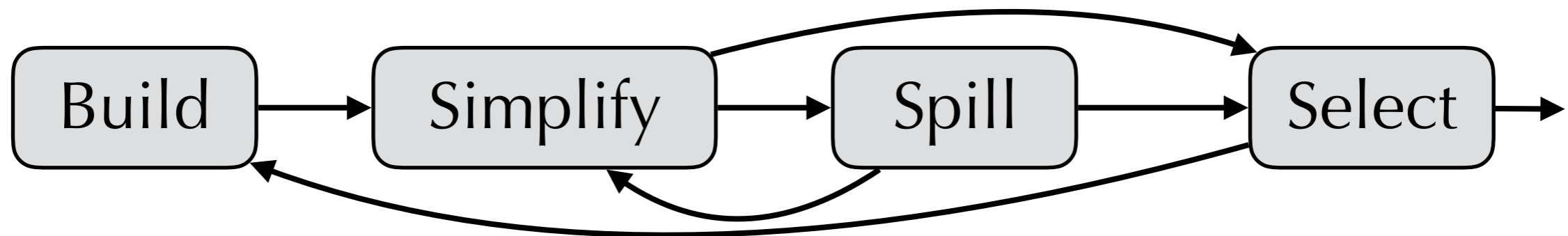
- **Build:** construct interference graph, using dataflow analysis to find for each program point vars that are live at the same time
- **Simplify:** color based on simple heuristic
  - If graph G has node n with k-1 edges, then G-{n} is k-colorable iff G is k-colorable
  - So remove nodes with degree $<k$
- **Spill:** if graph has only nodes with degree $\geq k$, choose one to **potentially spill** (i.e., that may need to be saved to stack)
  - Then continue with Simplify
- **Select:** when graph is empty, start restoring nodes in reverse order and color them
  - When we encounter a potential spill node, try coloring it. If we can't, rewrite program to store it to stack after definition and load before use. Try again!

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}
```
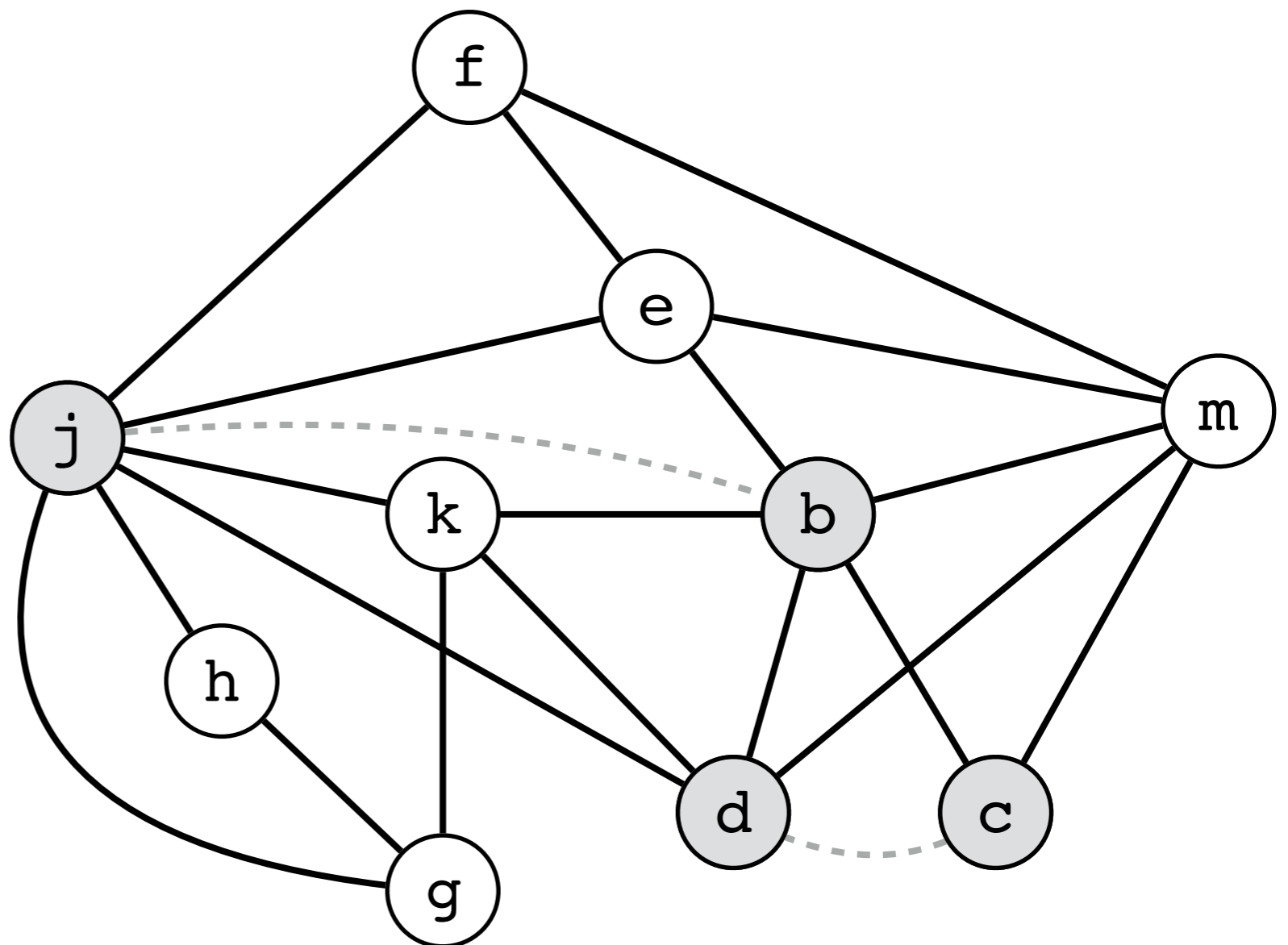
Coalescing Heuristics

- 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!
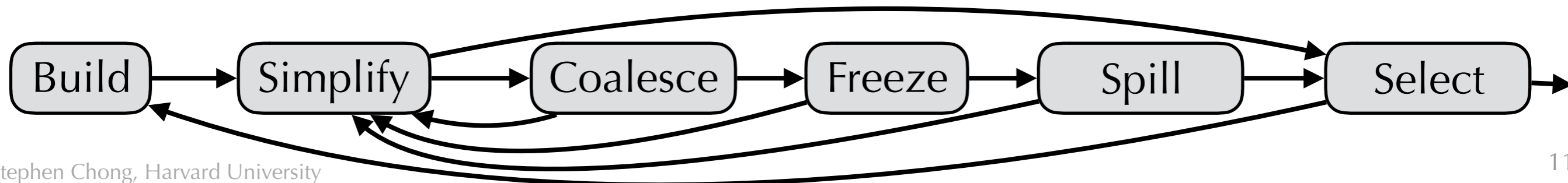
Build → Simplify → Coalesce → Freeze → Spill → Select

Example (4 registers)

Stack:

j and b, and d and c
are move related

Build → Simplify → Coalesce → Freeze → Spill → Select →
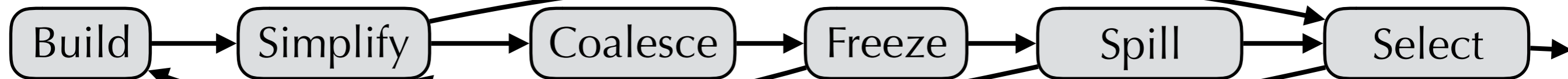
12

Example (4 registers)

Stack:

g
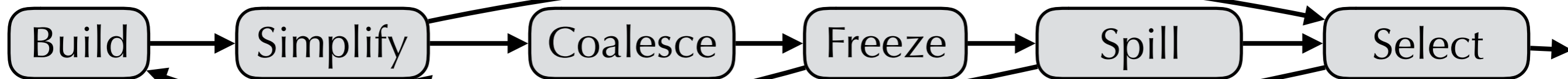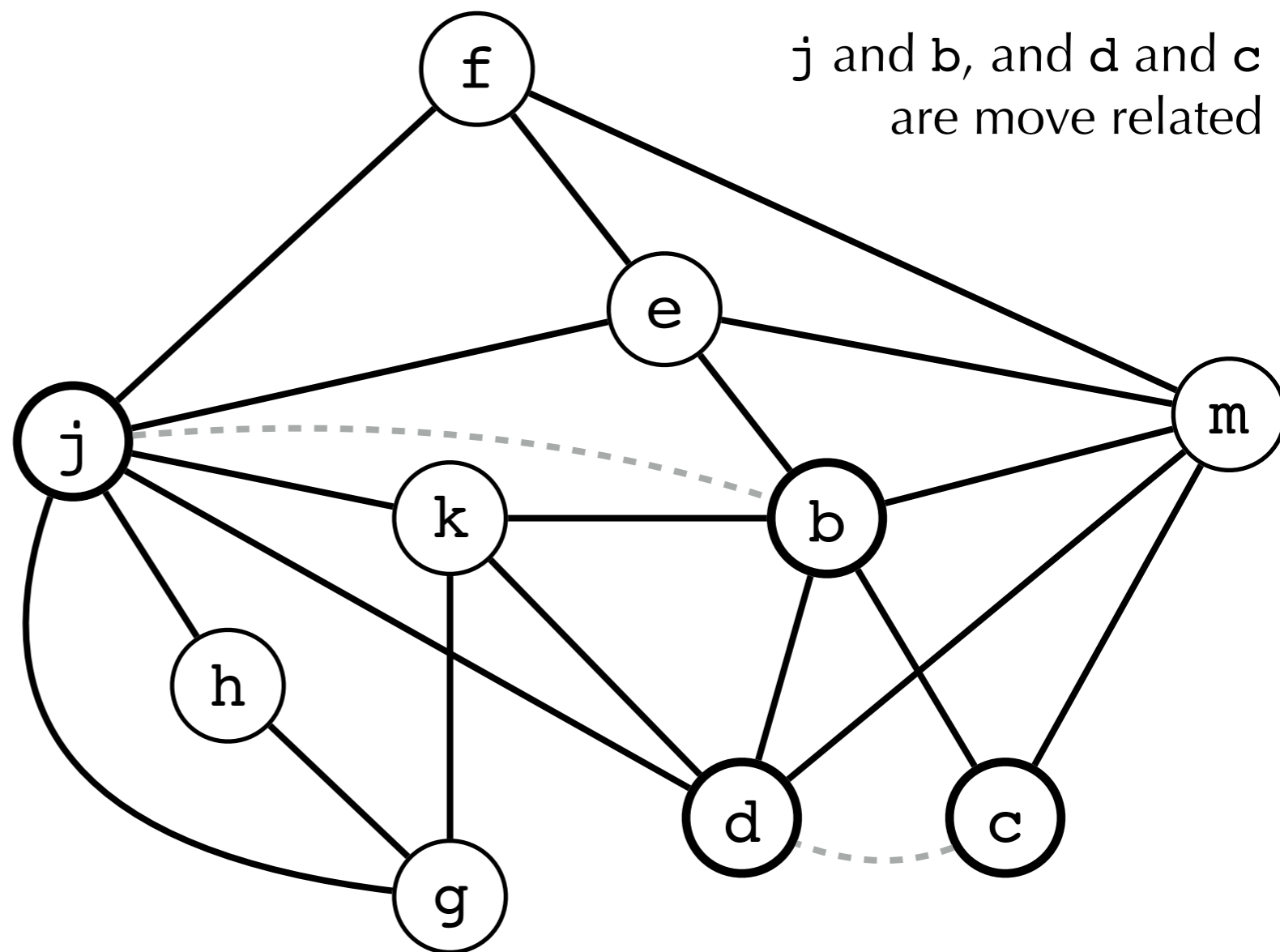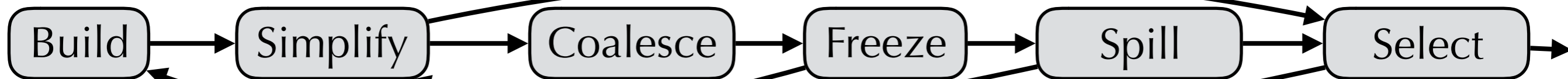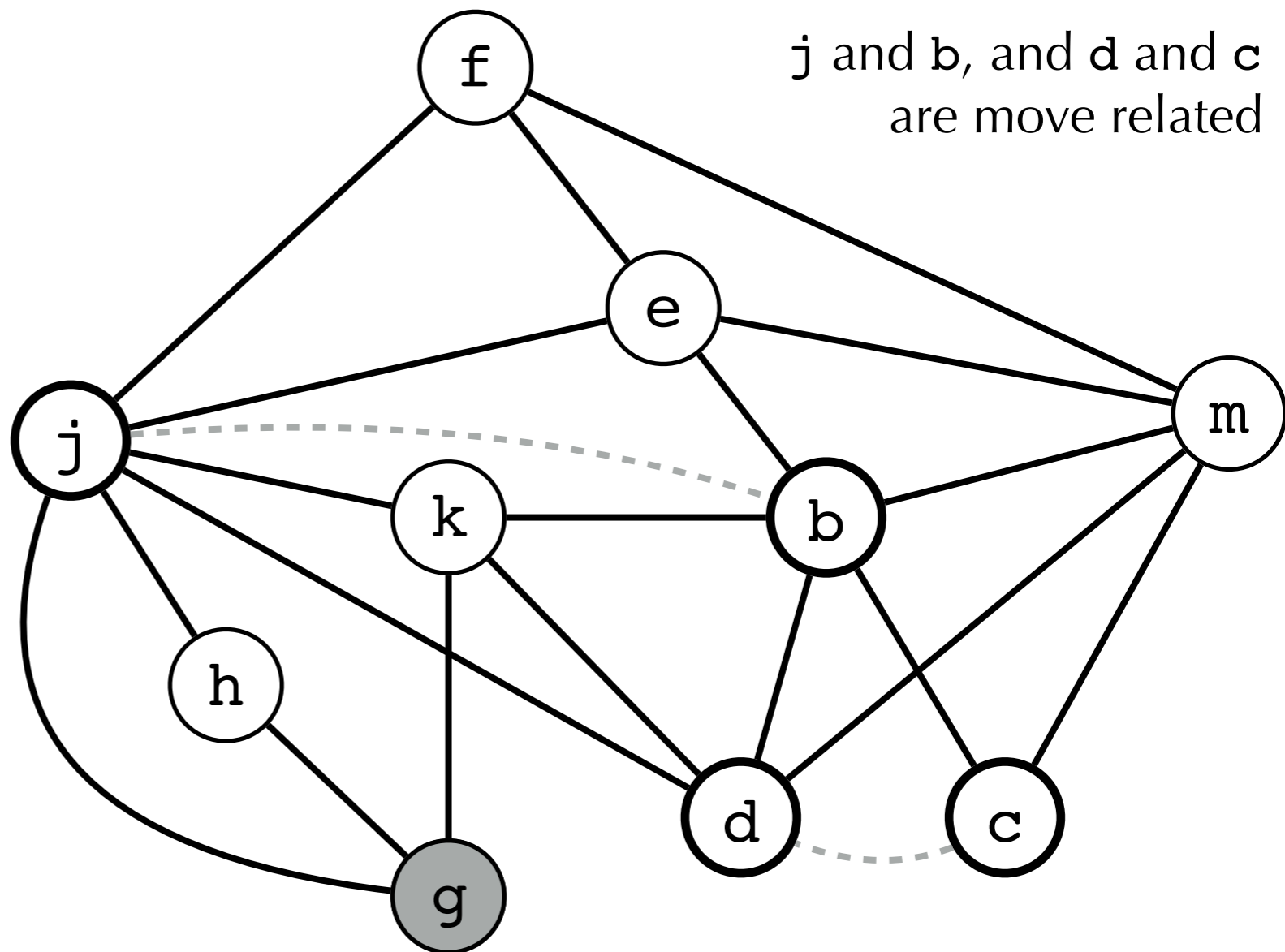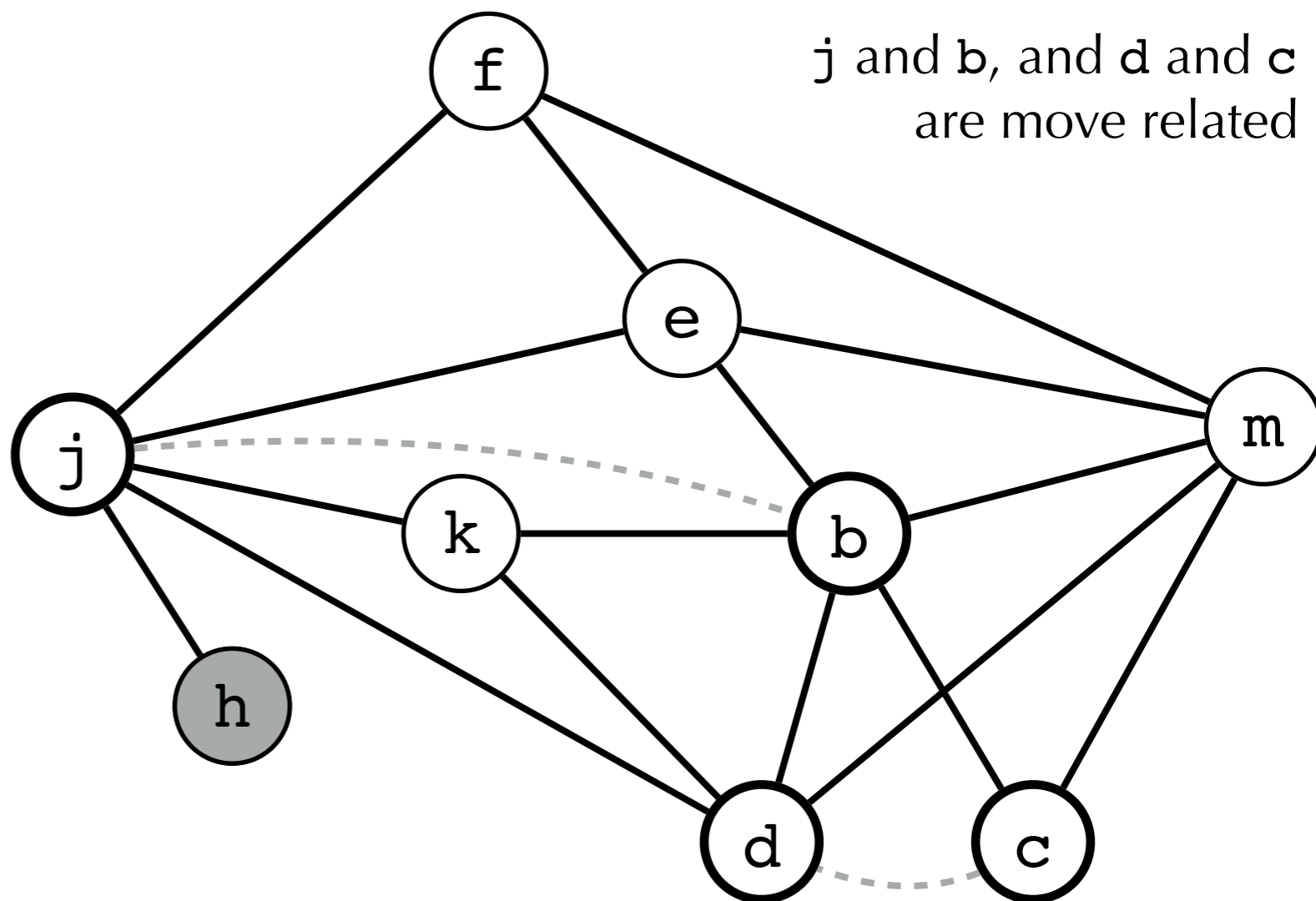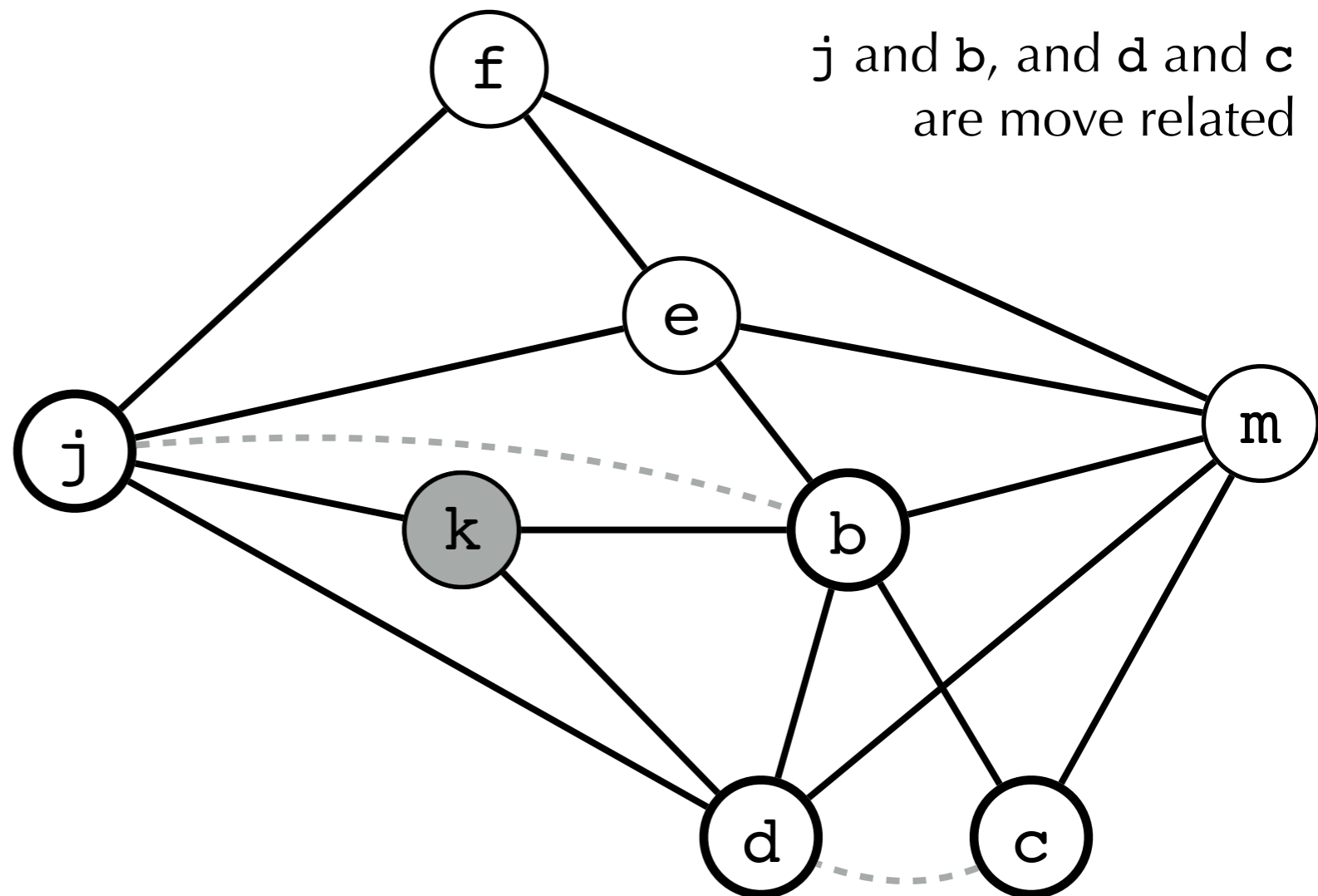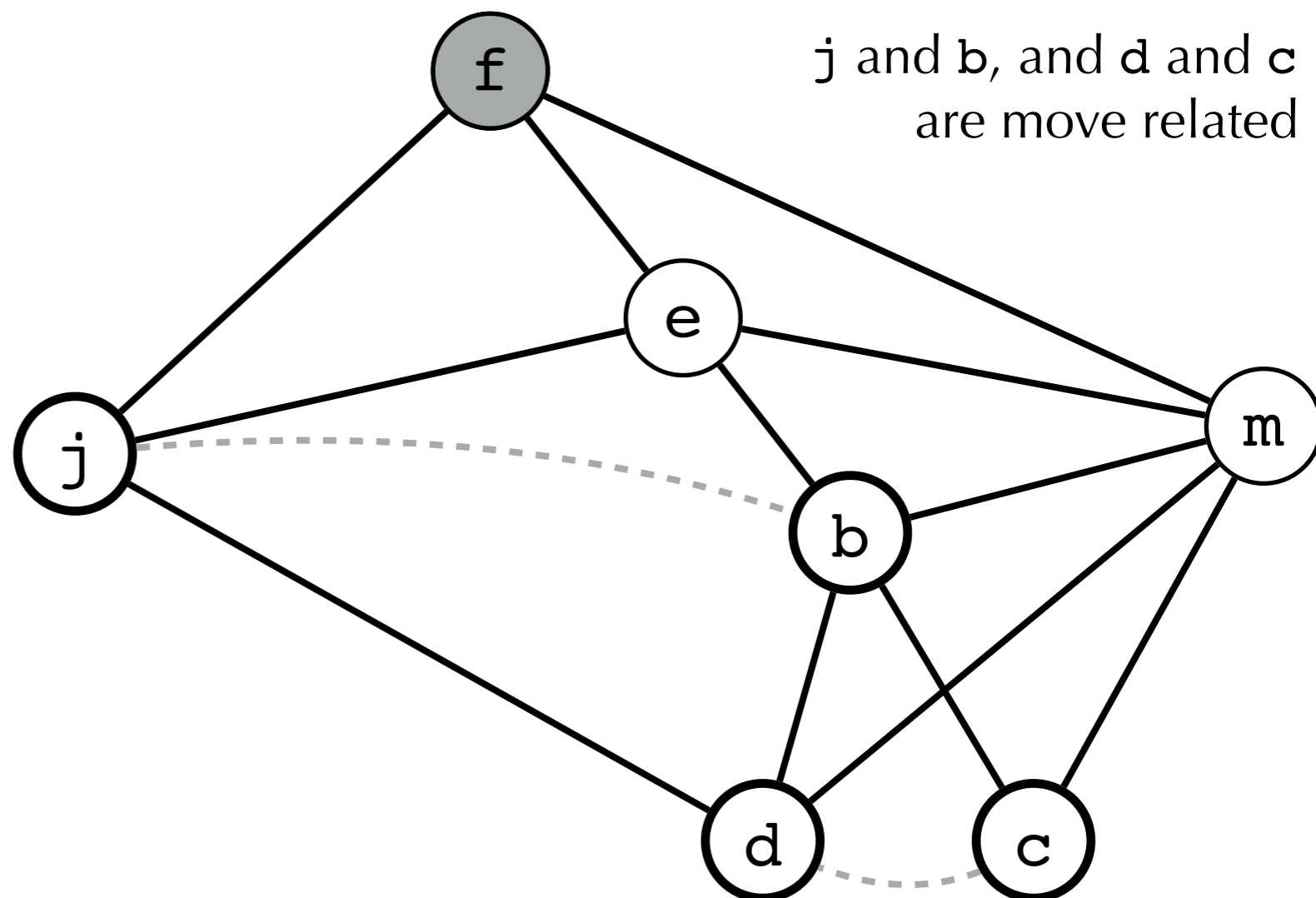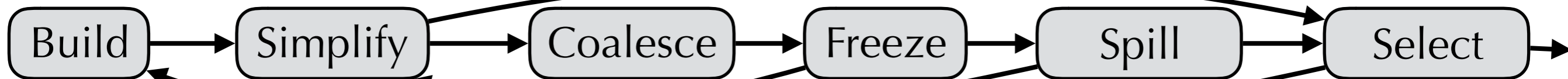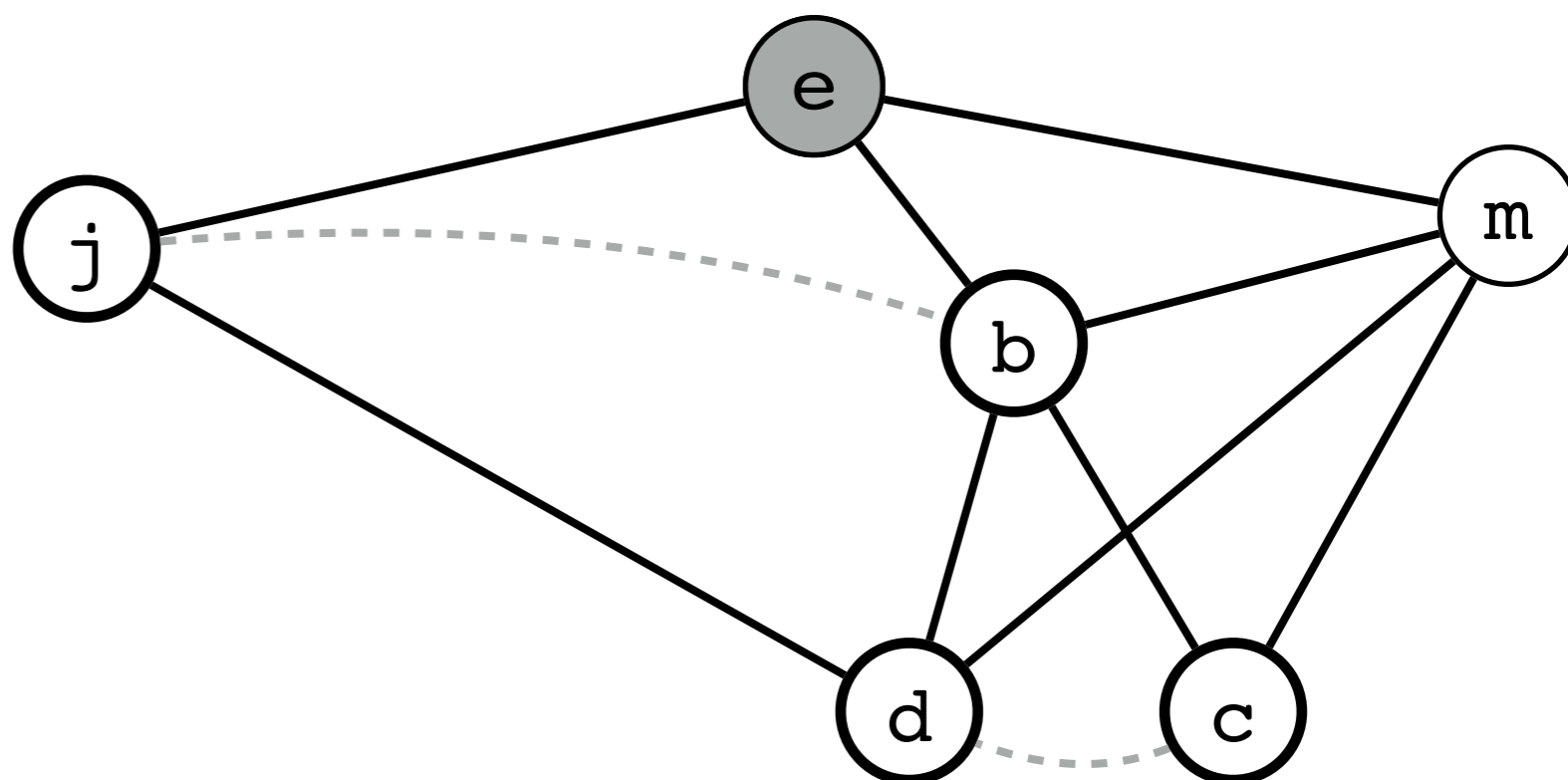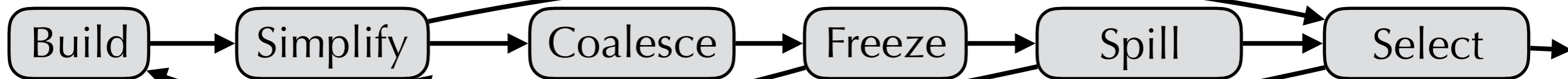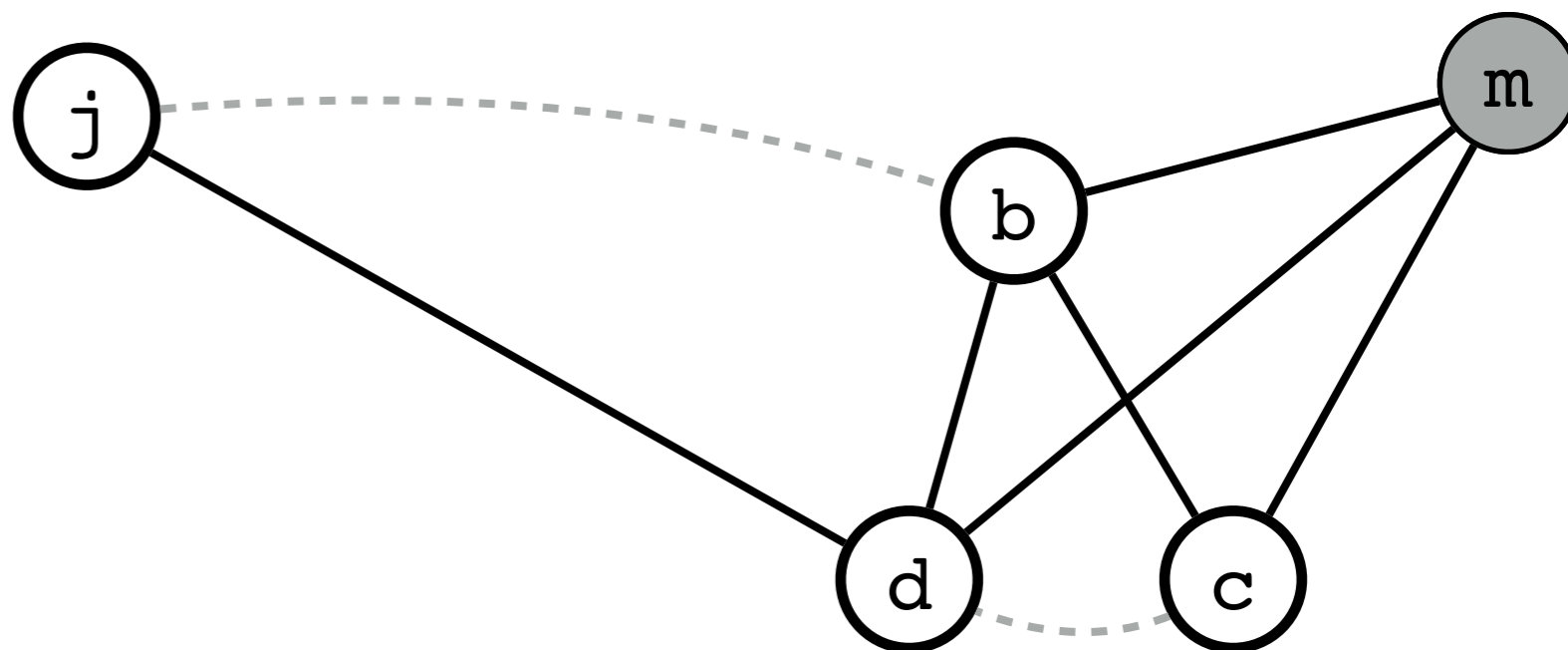
j and **b**, and **d** and **c**
are move related

Example (4 registers)

Stack:

g
h

f

e

m

j

k

b

h

d

c

j and b, and d and c are move related

Build → Simplify → Coalesce → Freeze → Spill → Select →

Example (4 registers)

Stack:

g
h
k

j and b, and d and c
are move related

Build → Simplify → Coalesce → Freeze → Spill → Select
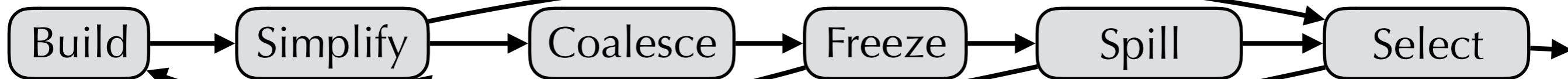
Example (4 registers)

Stack:

```
g
h
k
f
```


j and b, and d and c are move related

Build → Simplify → Coalesce → Freeze → Spill → Select
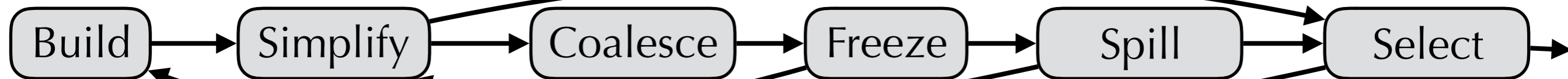
Example (4 registers)

Stack:

g
h
k
f
e

j and b, and d and c
are move related

Stephen Chong, Harvard University
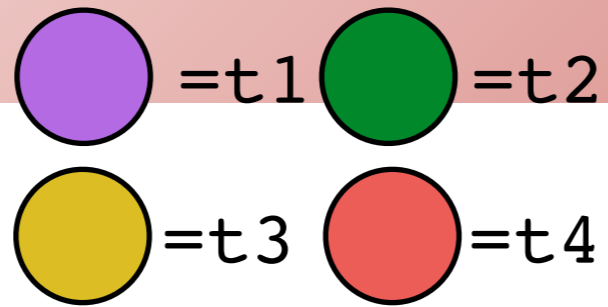
Example (4 registers)

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

Stack:

```
g
h
k
f
e
m
```

Example (4 registers)

Stack:

g
h
k
f
e
m

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

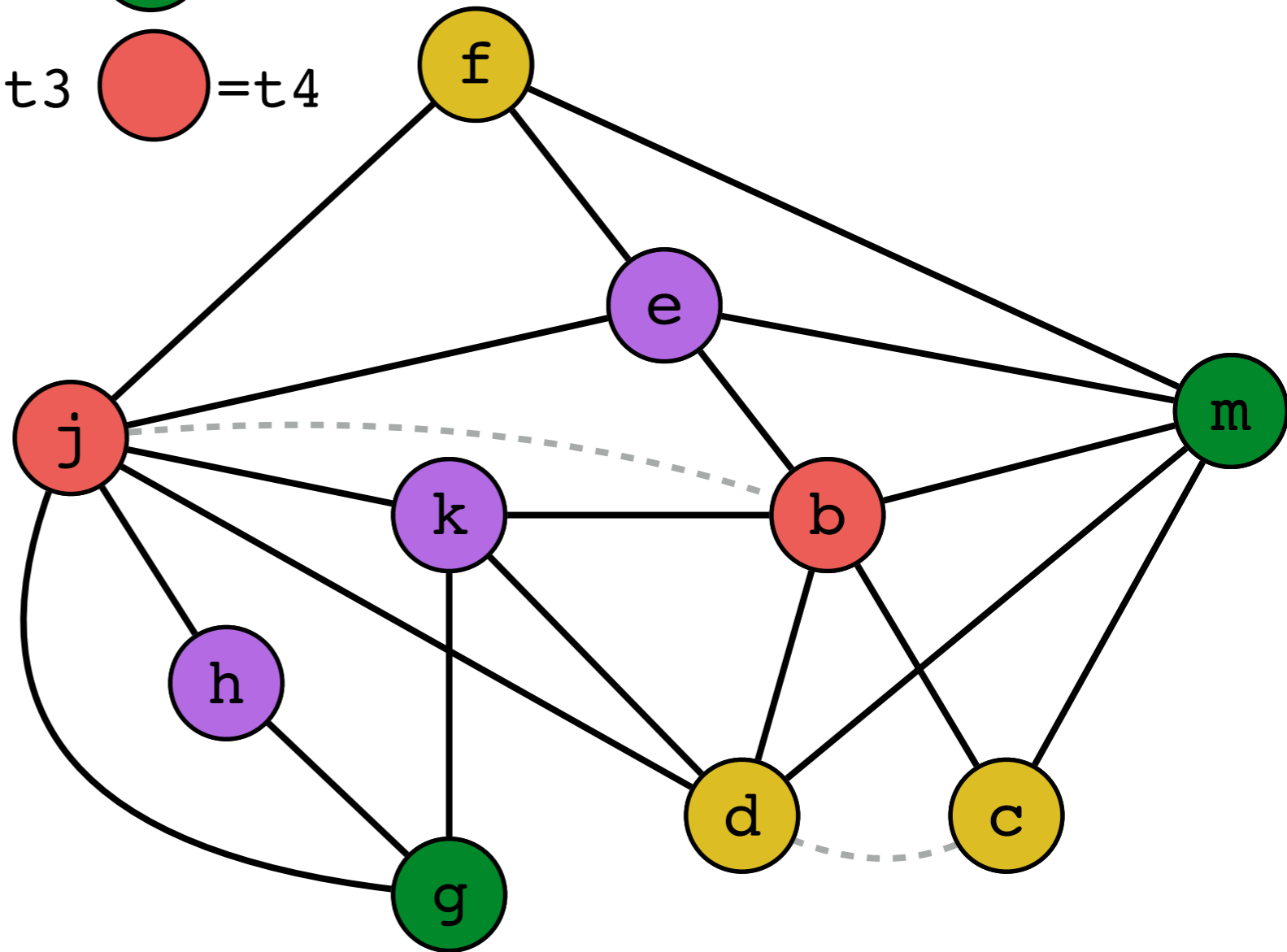Remaining nodes are move related, so coalesce

Build → Simplify → Coalesce → Freeze → Spill → Select

Example (4 registers)

d and **c**
are move related

Stack:

g
h
k
f
e
m

Remaining nodes are move related, so coalesce

Build → Simplify → Coalesce → Freeze → Spill → Select

Example (4 registers)

Stack:

d and **c**
are move related

g

h

k

f

e

m

jb

Build → Simplify → Coalesce → Freeze → Spill → Select

Example (4 registers)

Stack:

d and c
are move related

g

h

k

f

e

m

jb

d c

Build → Simplify → Coalesce → Freeze → Spill → Select →

Example (4 registers)

Stack:

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

dc

```
Build → Simplify → Coalesce → Freeze → Spill → Select →
```

Example (4 registers)

Example (4 registers)

=t1   =t2

=t3   =t4
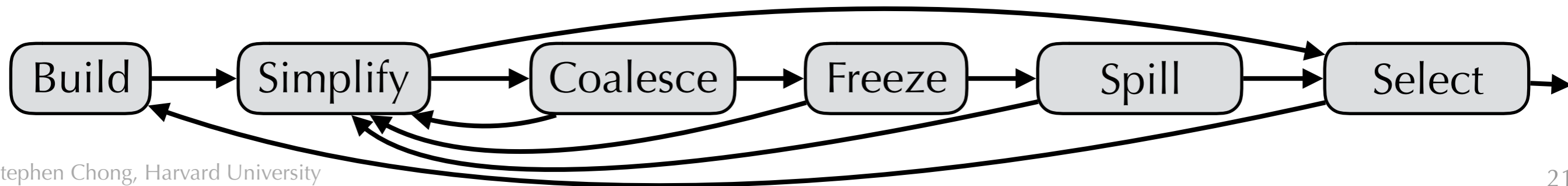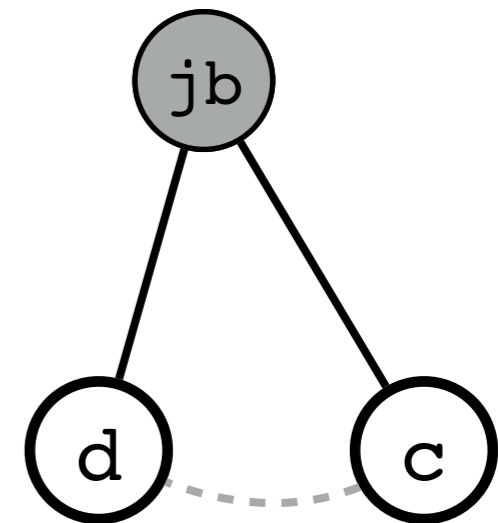
```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
{live-out: d,j,k}
```
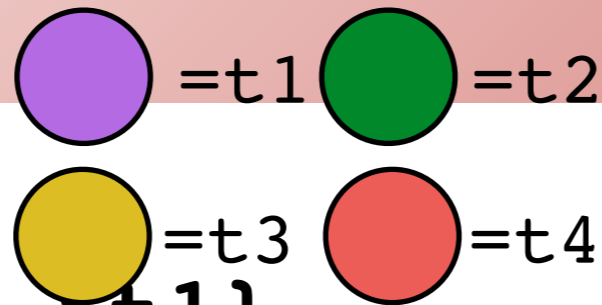
Example (4 registers)

=t1 =t2

=t3 =t4

```
{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}
```
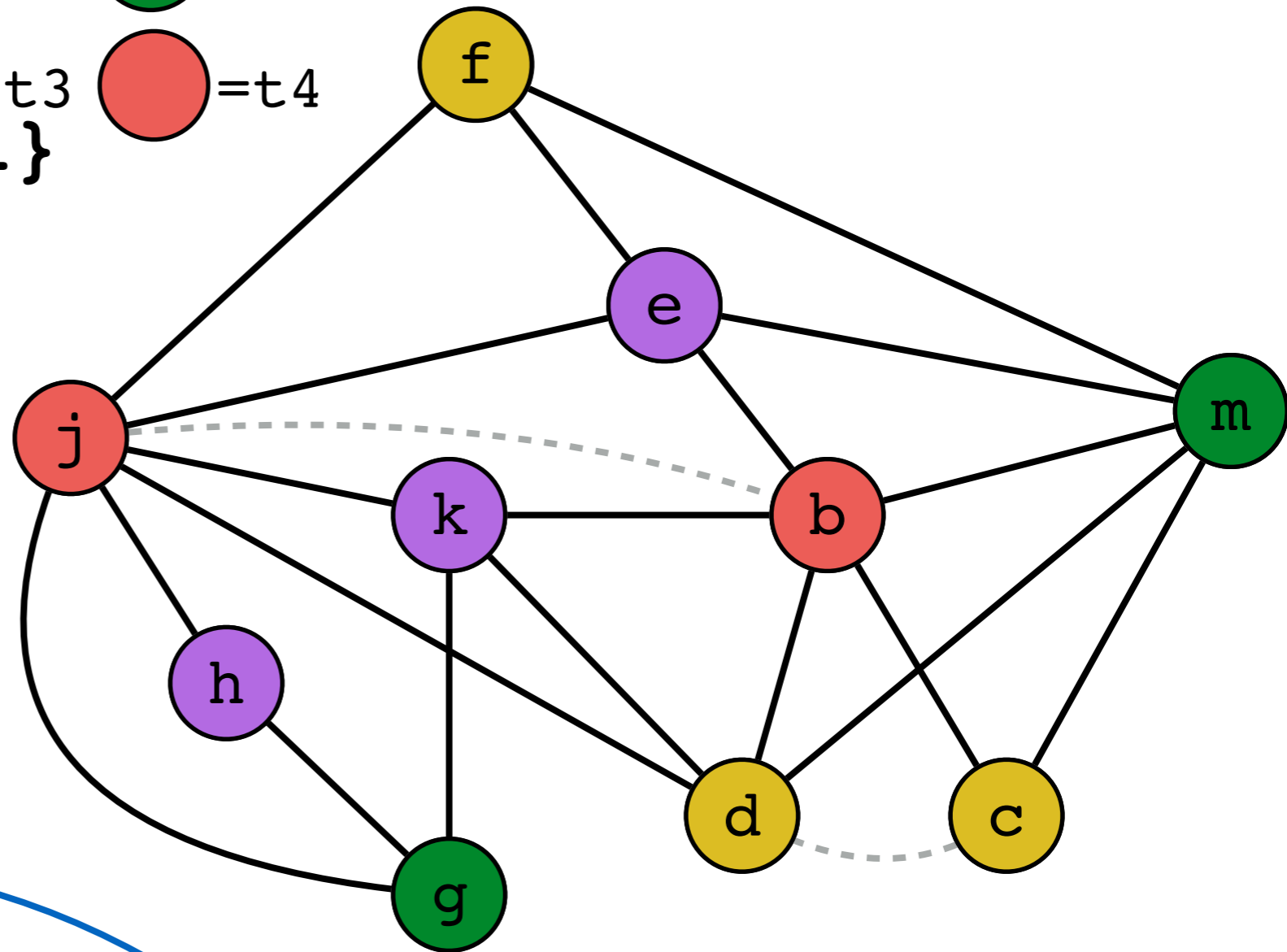
This is the result of coalescing!

Pre-colored Temps

- The IR often includes machine registers
  - e.g., `$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 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 `$a0-$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
- 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

Example

- 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
```
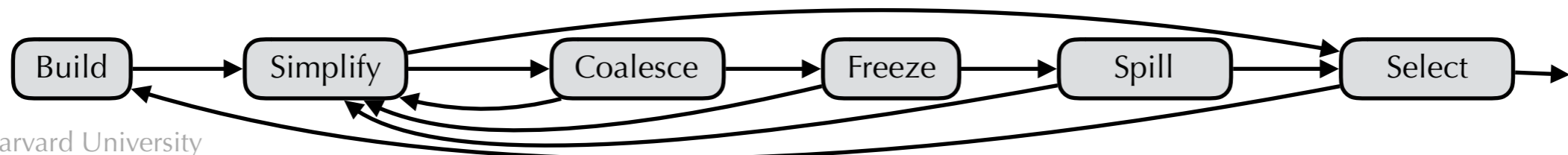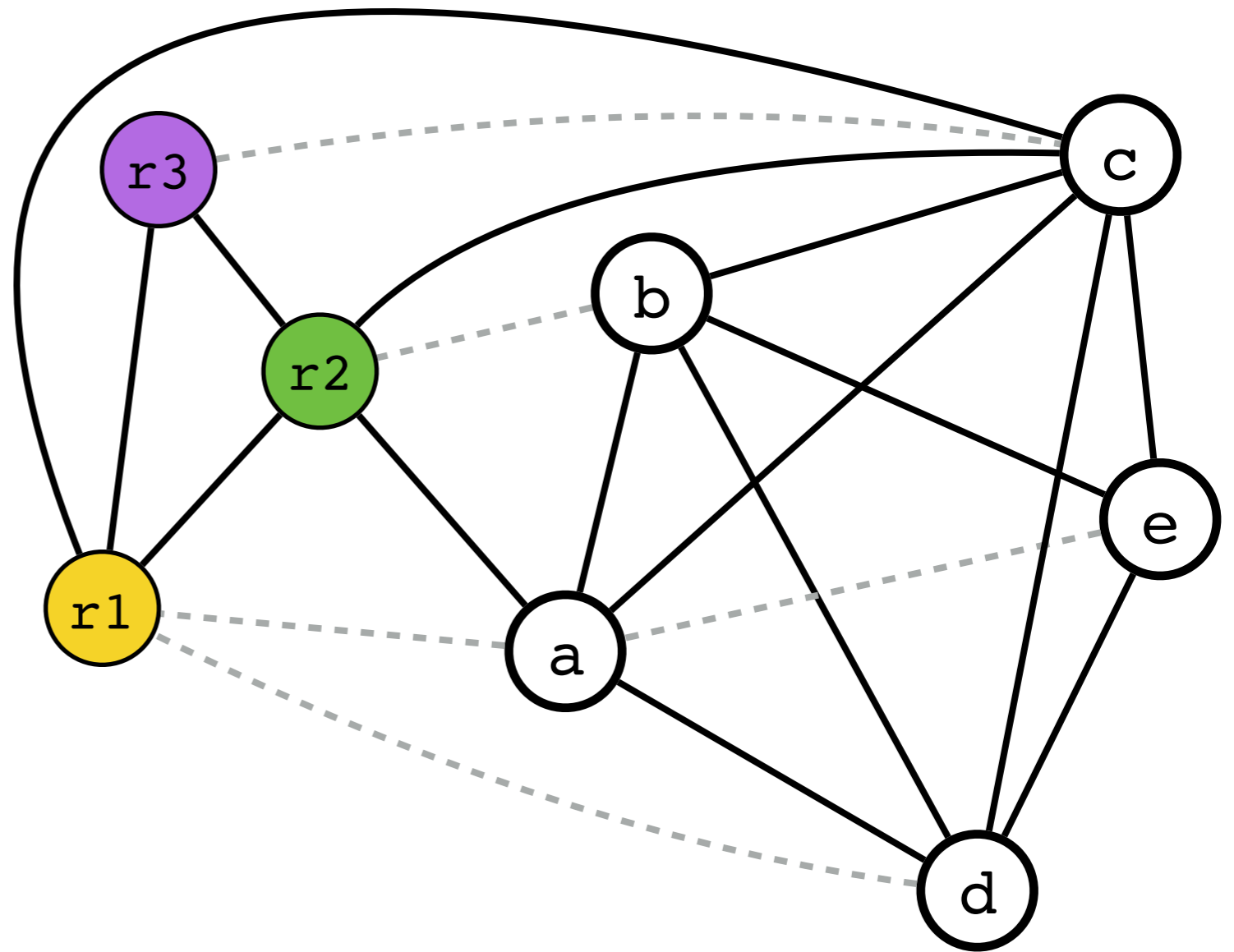
Example

Stack:

c *spill?*
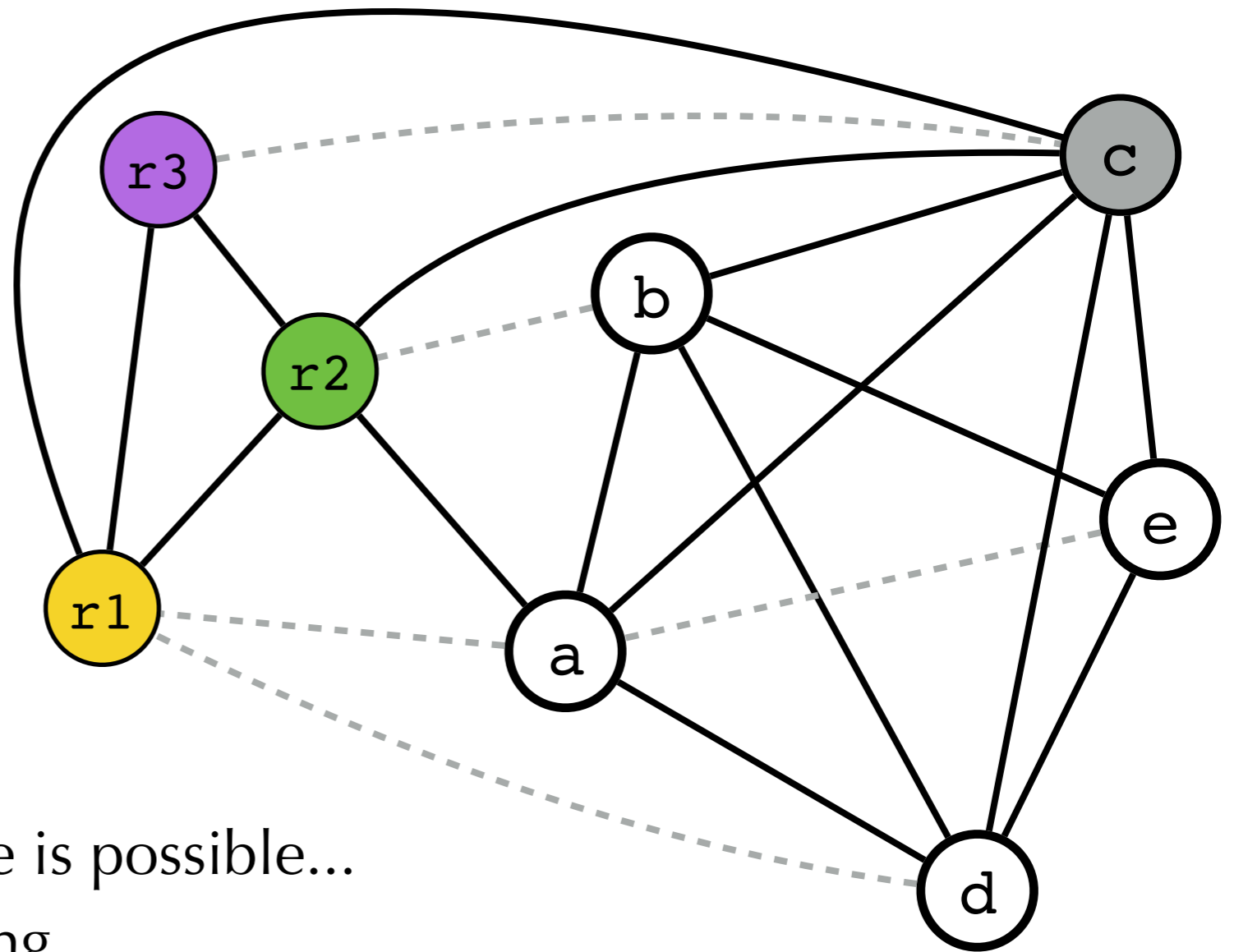
No simplify, coalesce, or freeze is possible...
c is a good candidate for spilling...

Build → Simplify → Coalesce → Freeze → Spill → Select
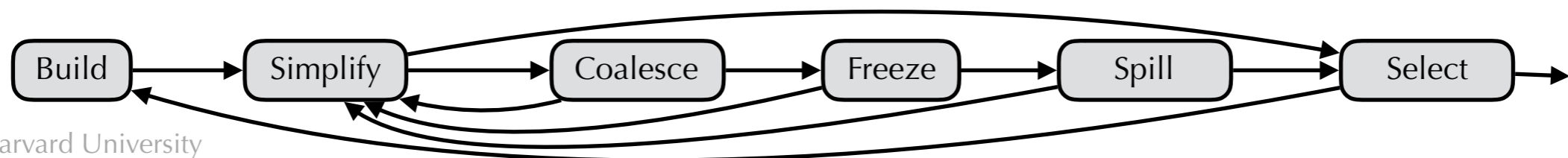
Example
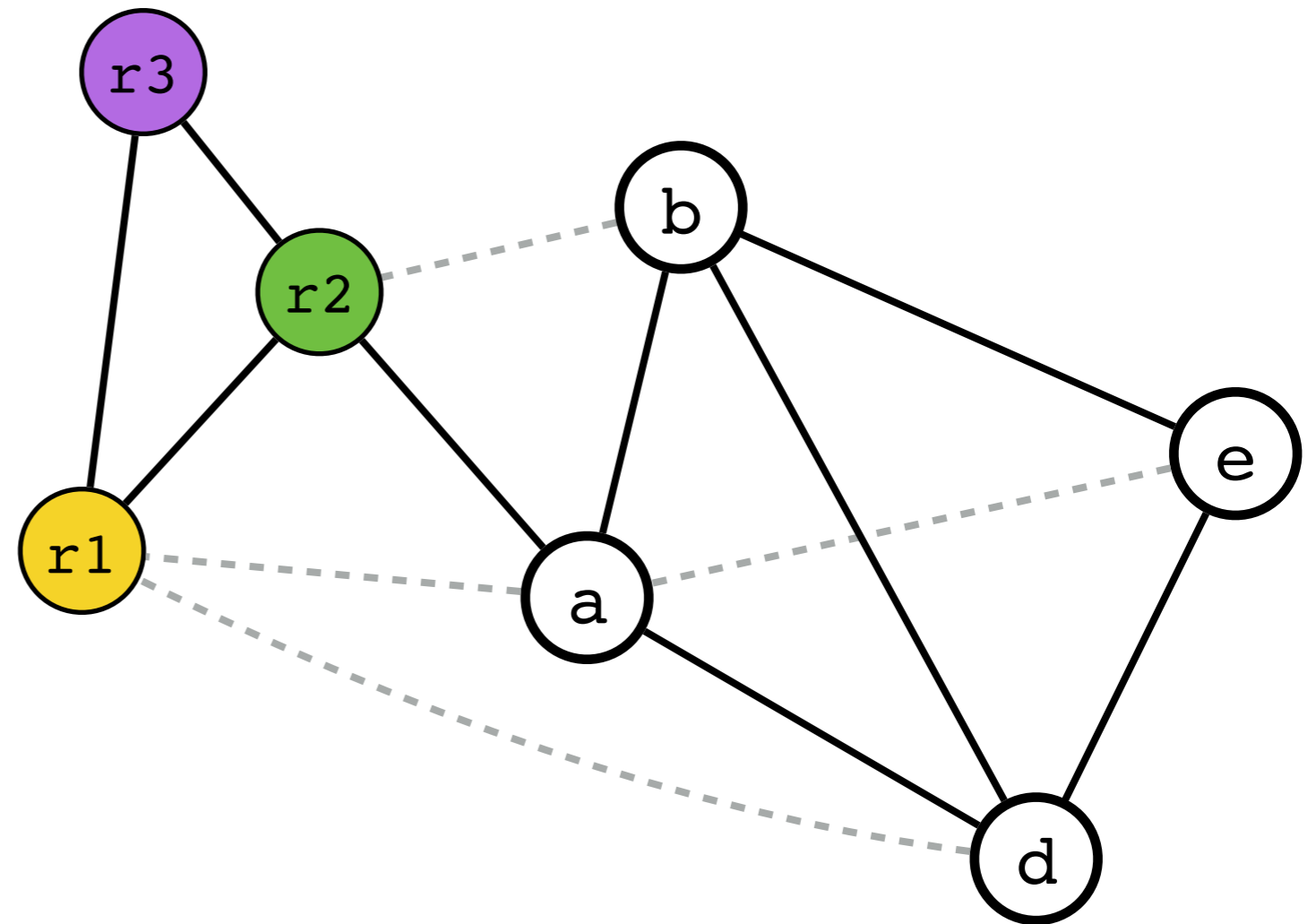
Stack:

c *spill?*

No simplify is possible...

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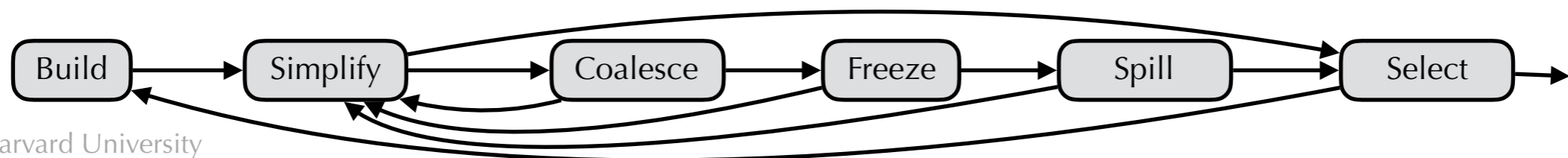
Example

Stack:

c *spill?*
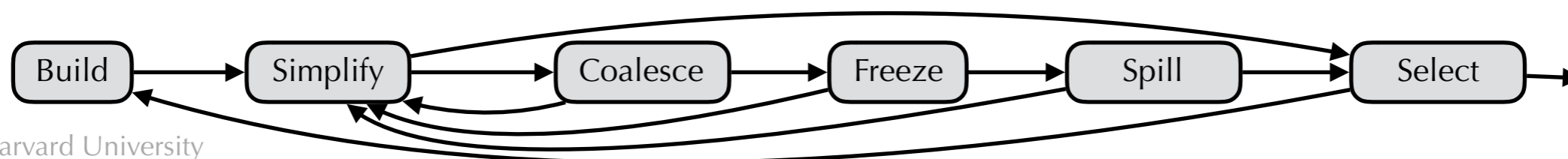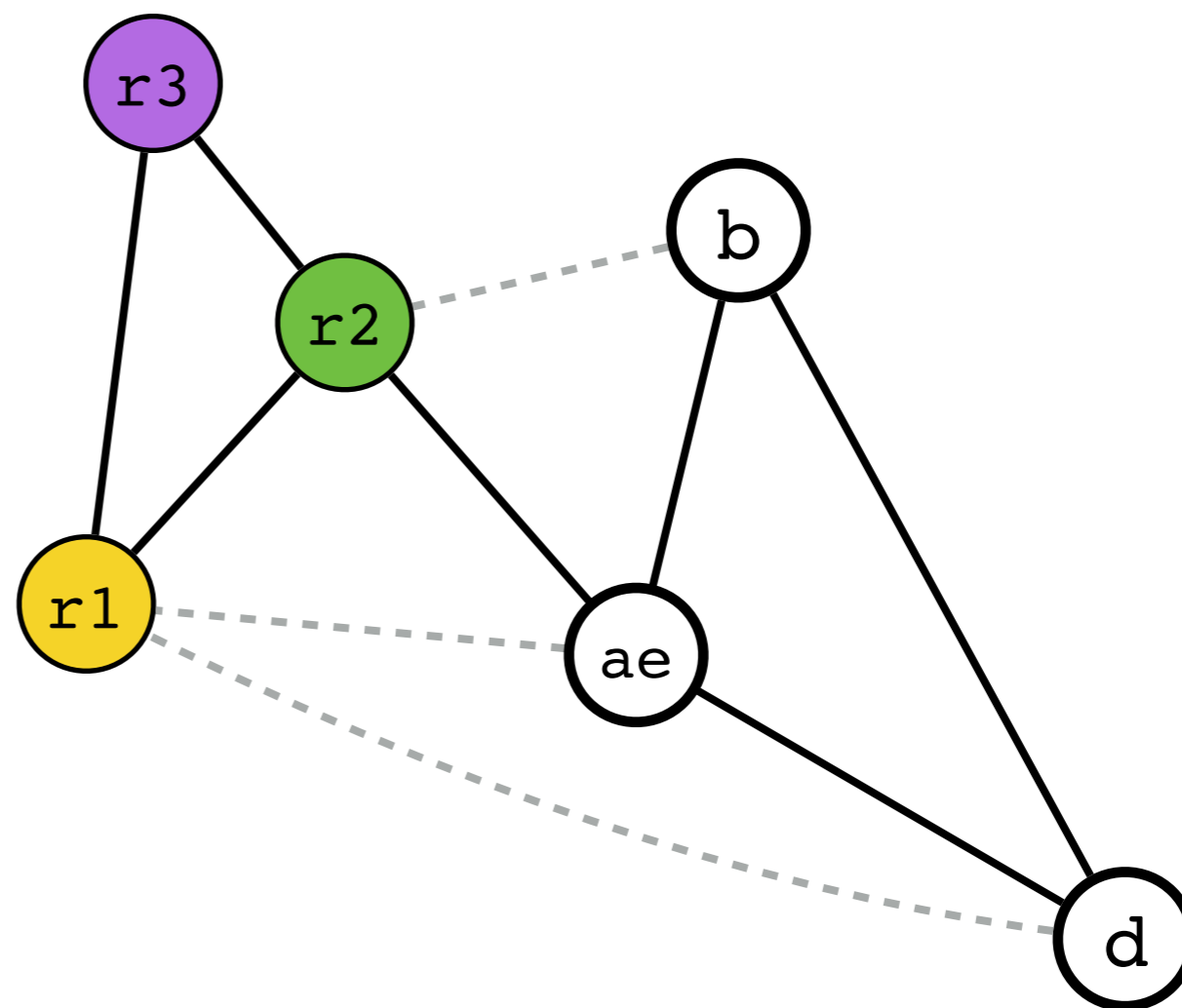
Example

Stack:

c *spill?*
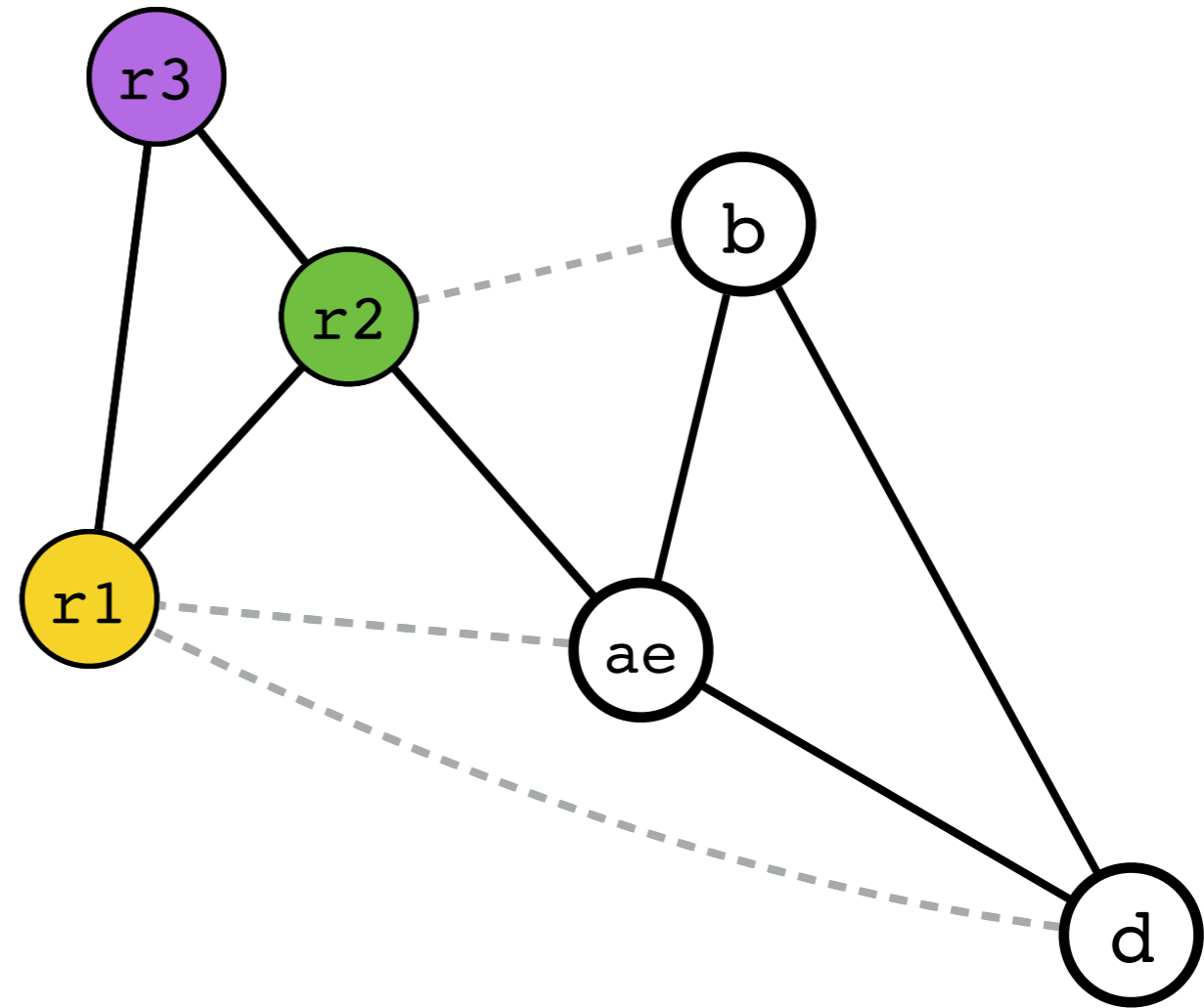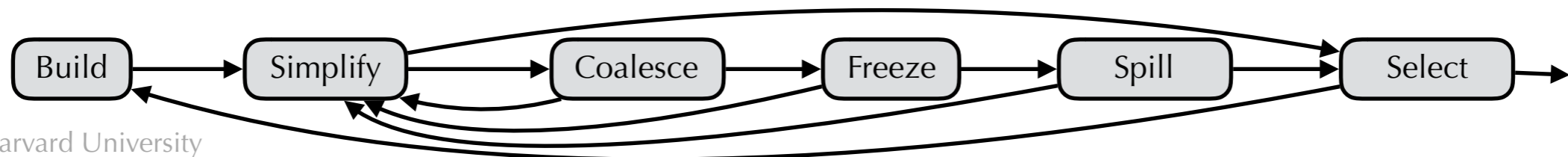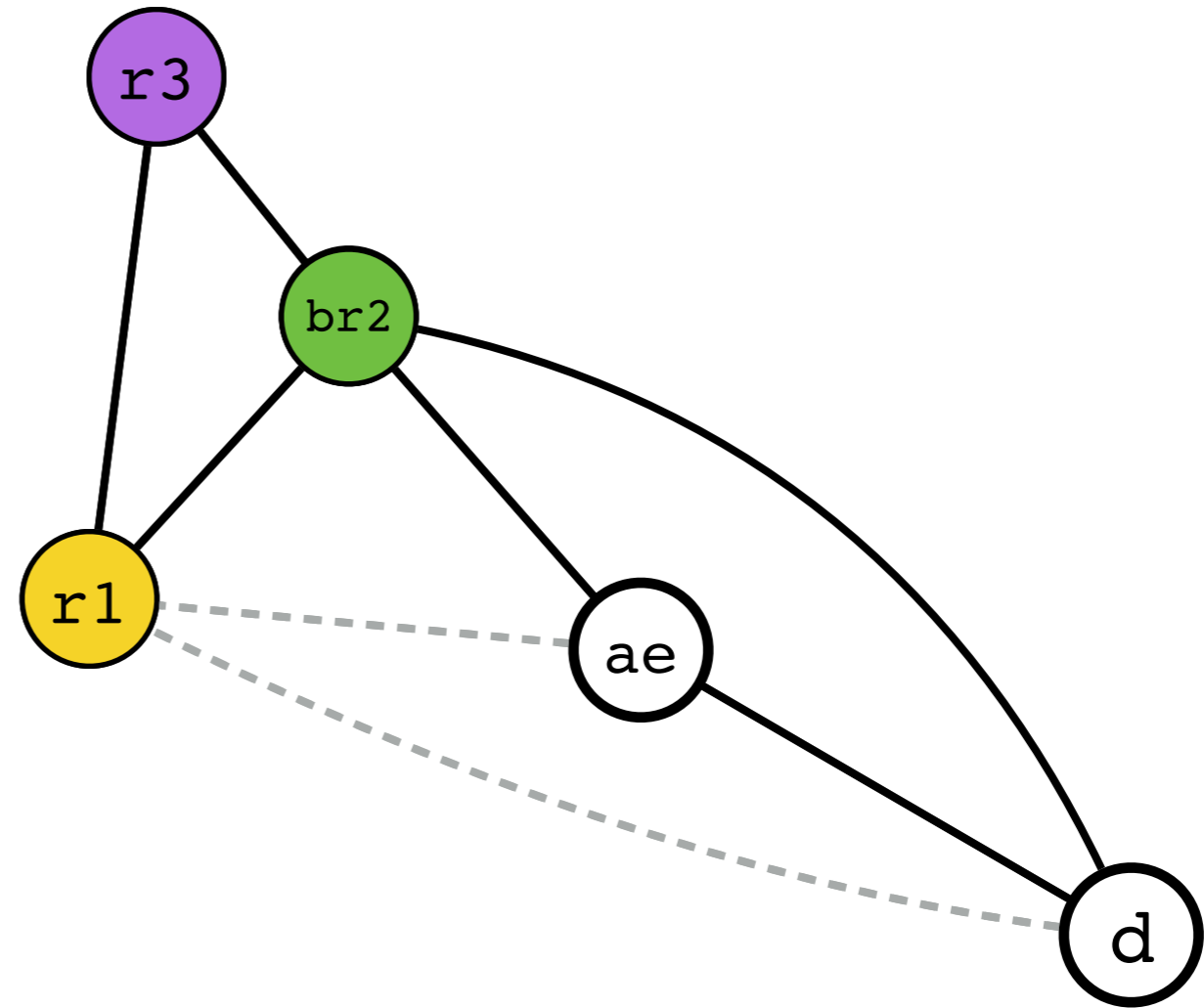
No simplify is possible...

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

Build → Simplify → Coalesce → Freeze → Spill → Select

Example

Stack:

c *spill?*

No simplify is possible...

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

Build → Simplify → Coalesce → Freeze → Spill → Select
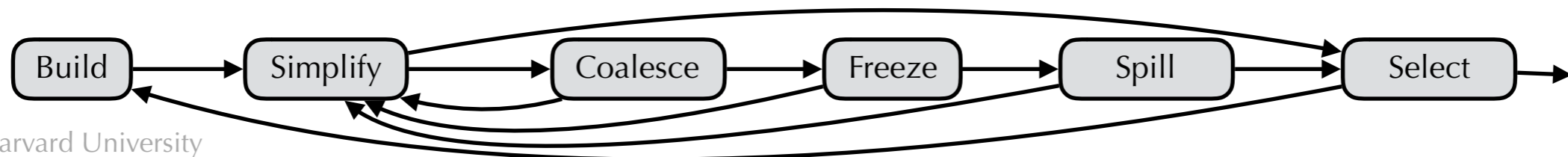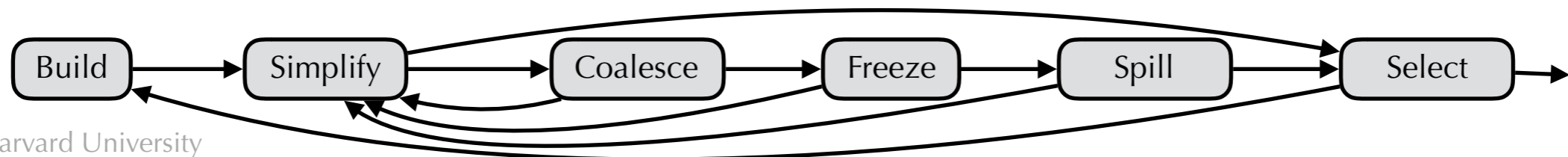
Example

Stack:

c *spill?*

No simplify is possible...

Coalesce `r1` and `ae`

Example

Stack:

c *spill?*

No simplify is possible…

Coalesce **r1** and **ae**

Build → Simplify → Coalesce → Freeze → Spill → Select
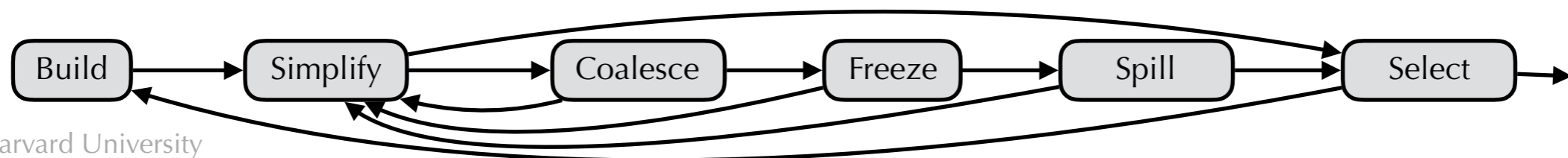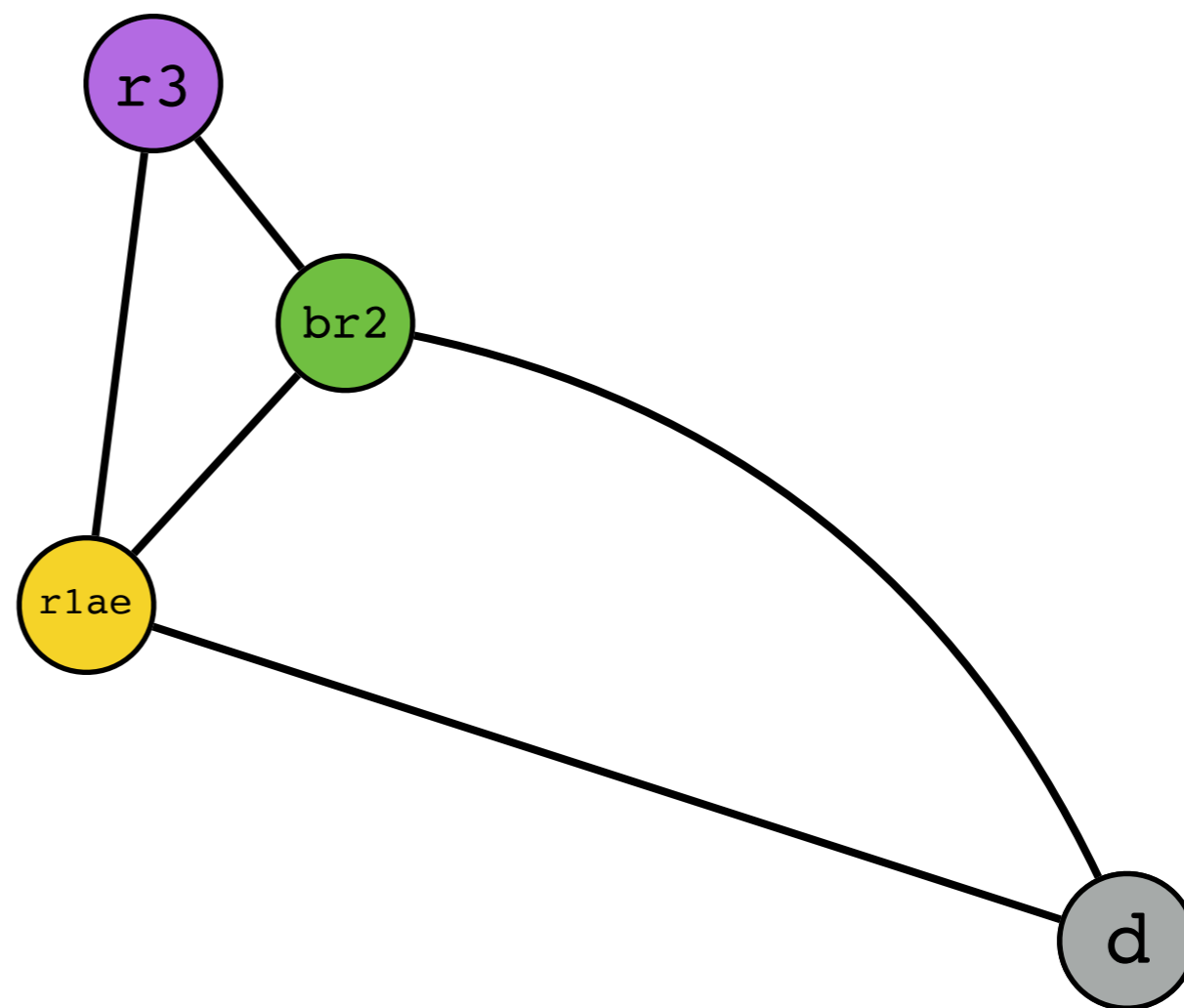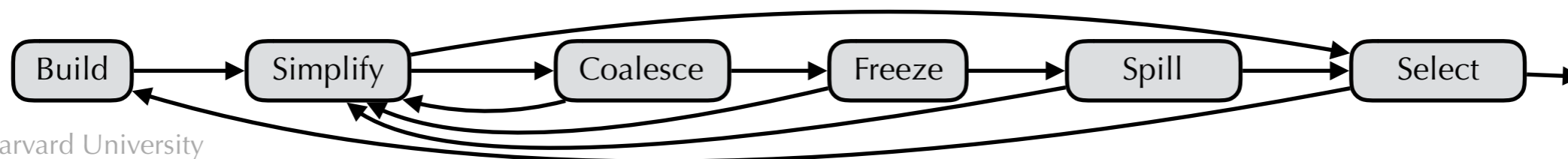
Example

Example
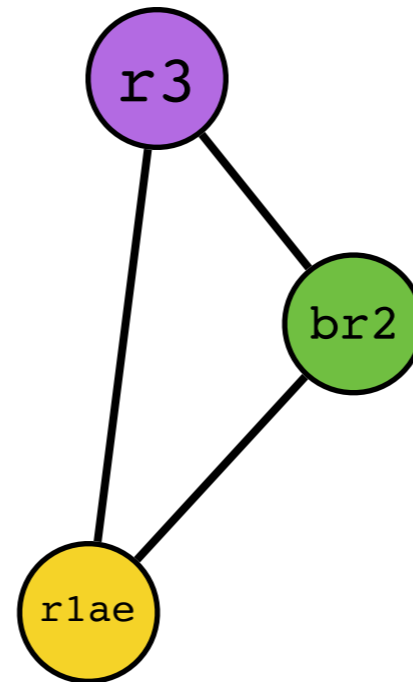
Stack:
c *spill?*
d

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

```
Build → Simplify → Coalesce → Freeze → Spill → Select →
```
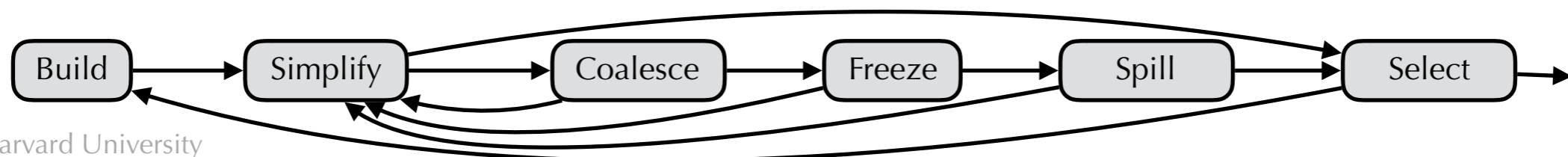
Example

Stack:

c *spill?*

d
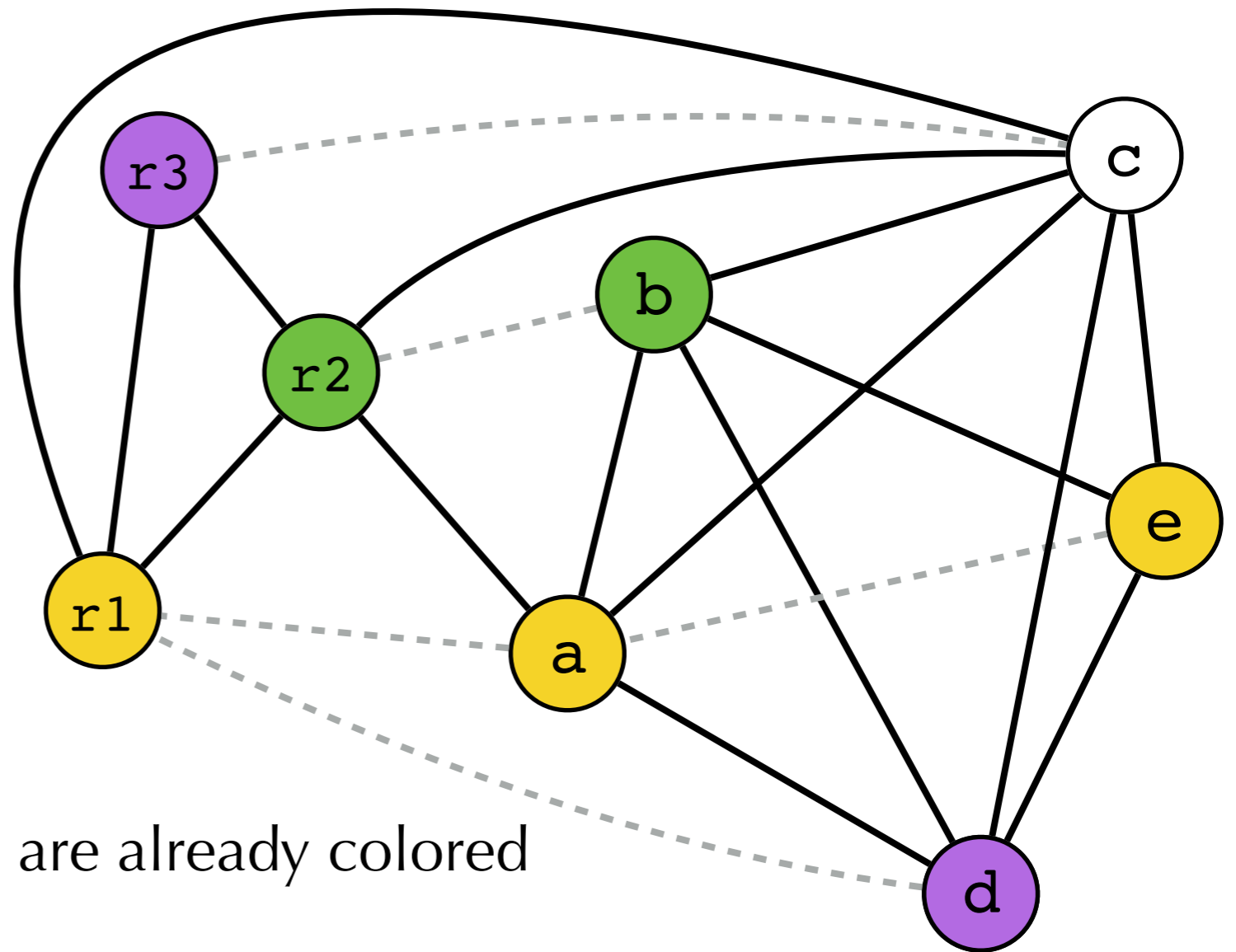
Due to coalescing, **b**, **a**, and **e** are already colored
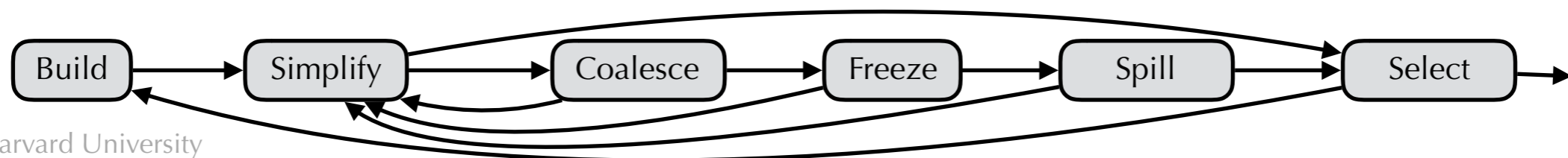
Pop **d** and color it

43
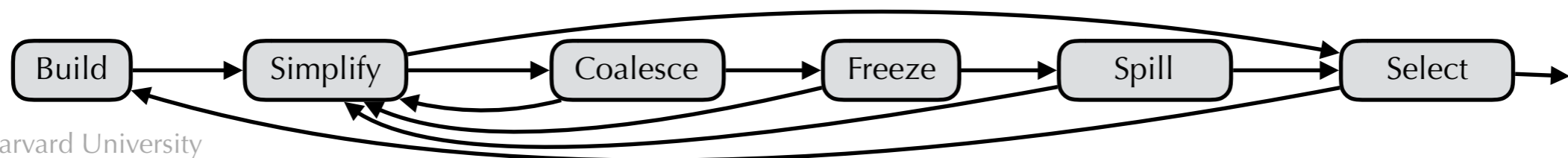
Example

Stack:
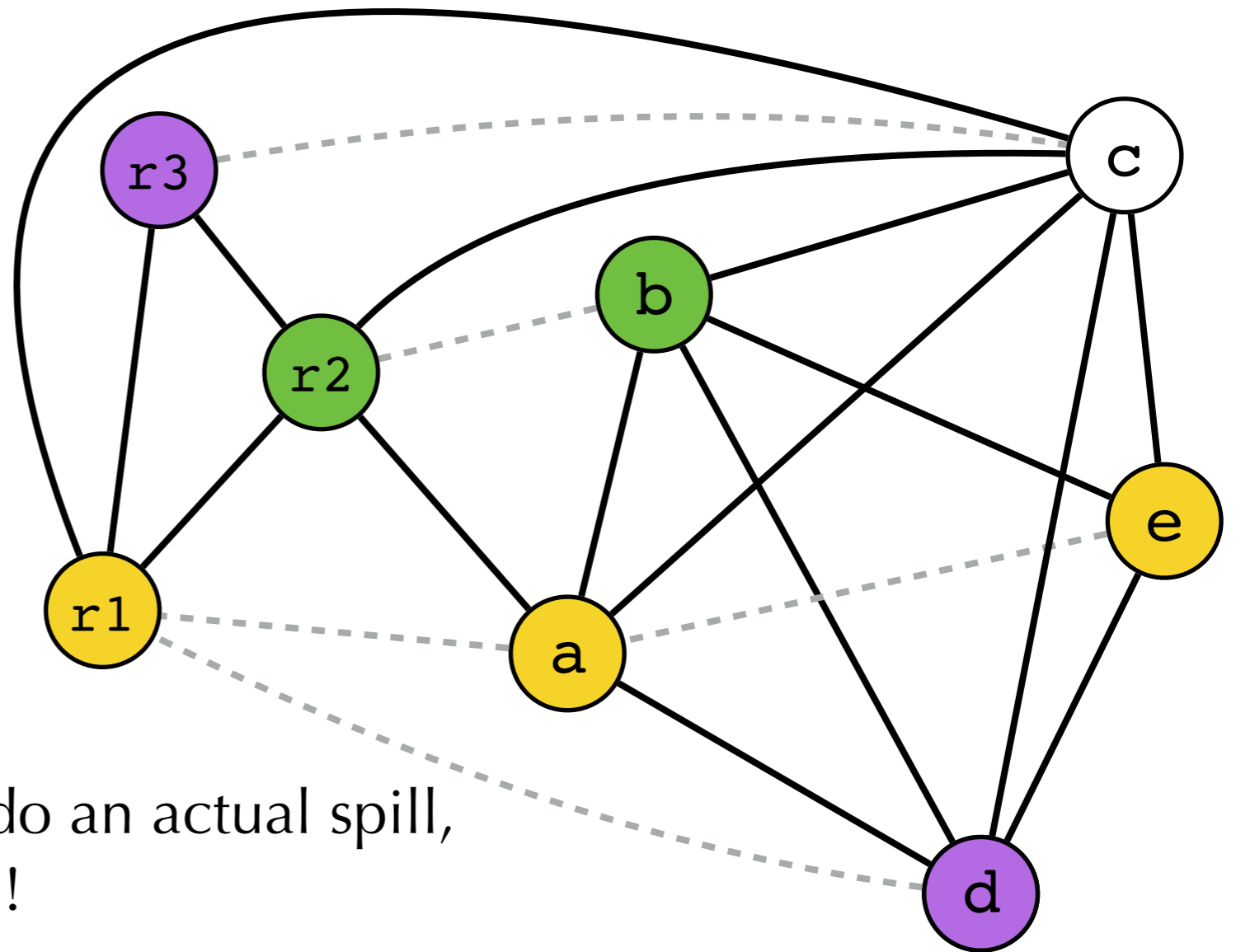
c *spill?*

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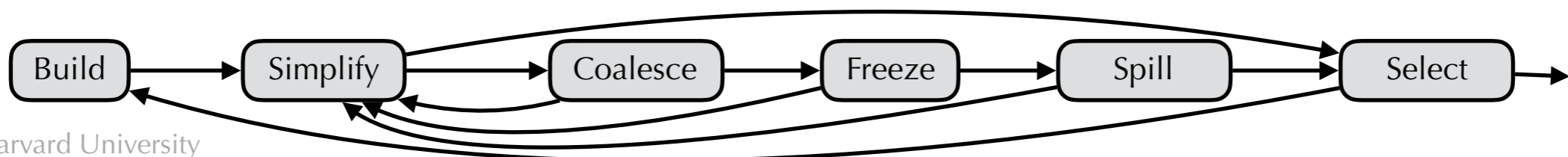i.e., rewrite code and try again!

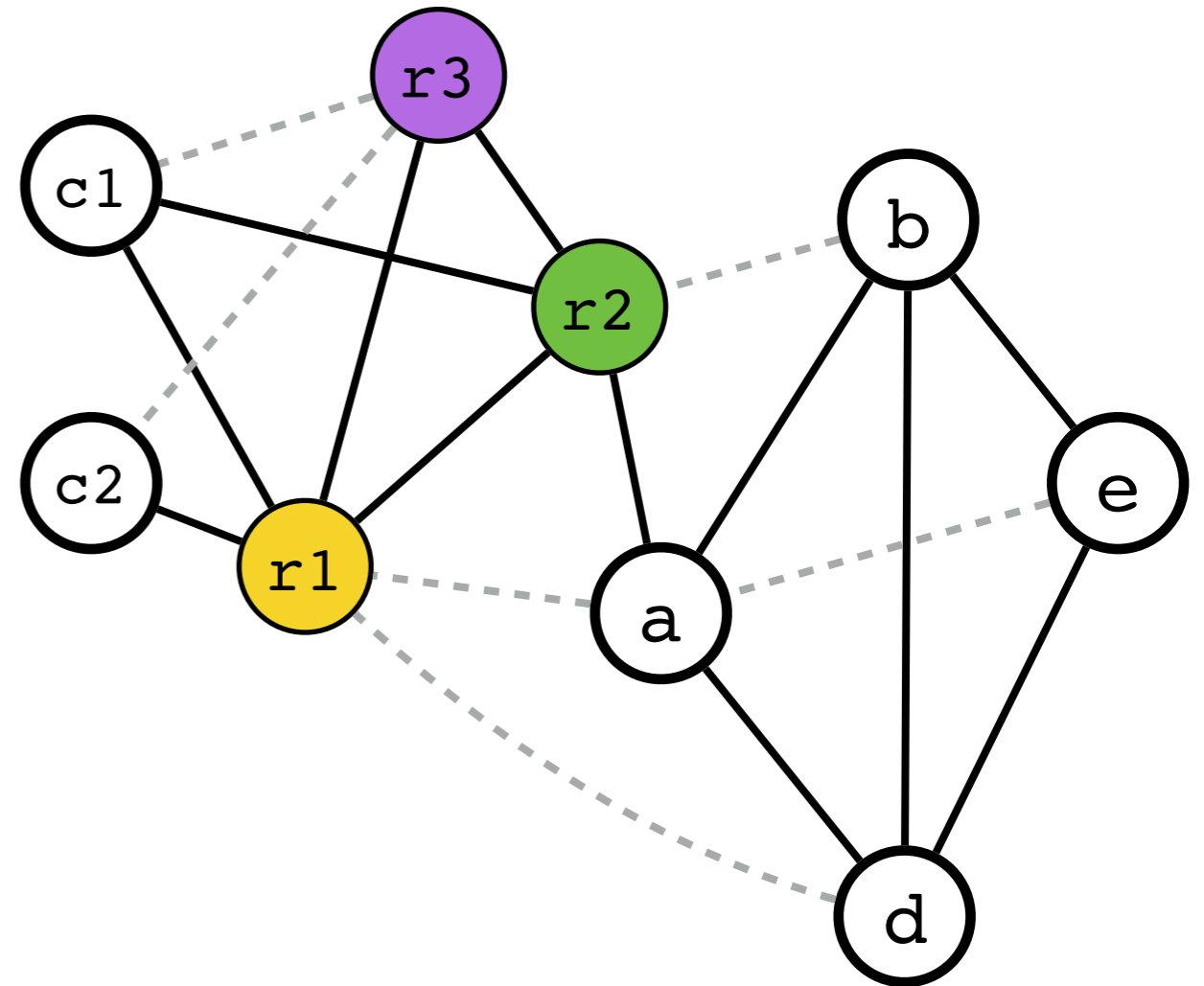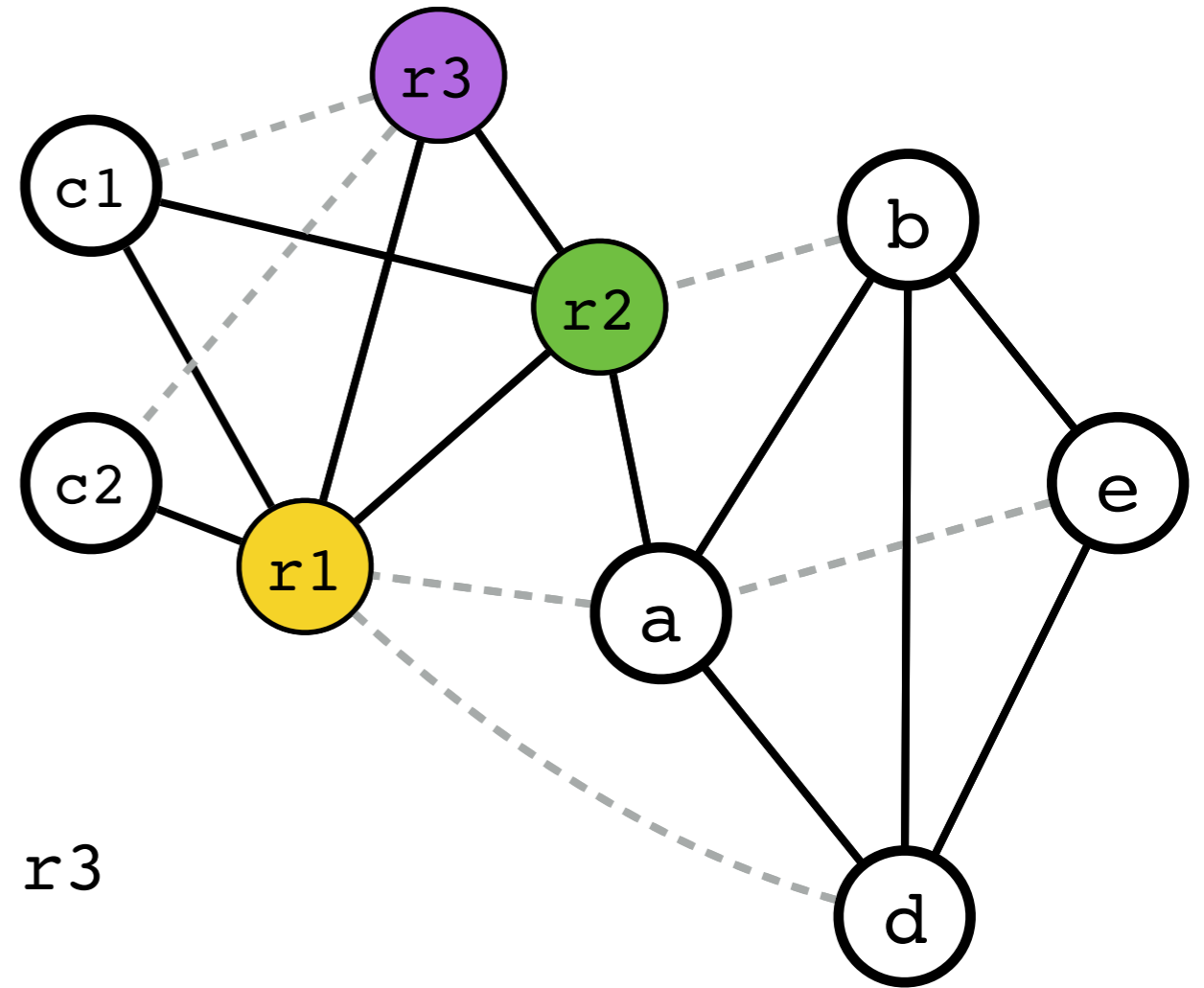Build → Simplify → Coalesce → Freeze → Spill → Select
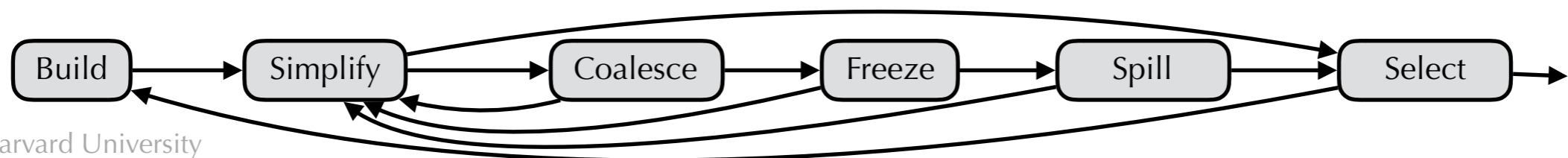
Example

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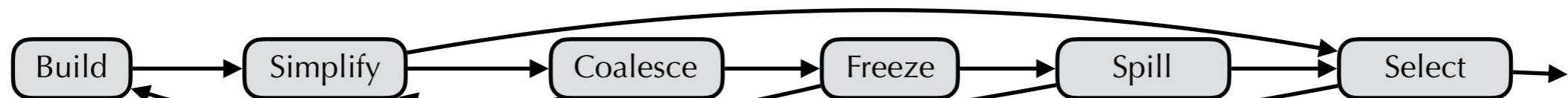
```
f: c1 := $r3
   Mem[fp+i] := c1
   a  := $r1
   b  := $r2
   d  := 0
   e  := a
loop:
   d  := d + b
   e  := e - 1
   if e > 0 loop else end
end:
   r1 := d
   c2 := Mem[fp+i]
   r3 := c2
   return
```
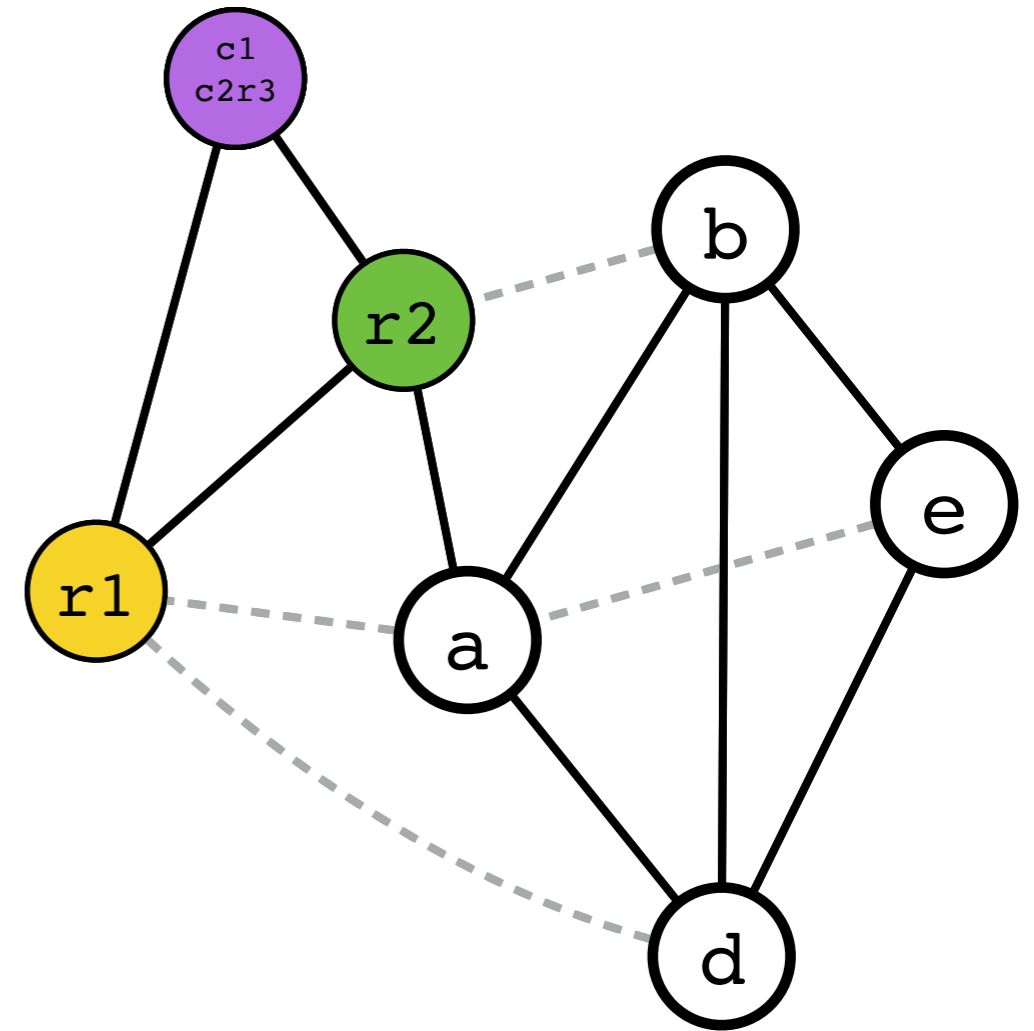
Build → Simplify → Coalesce → Freeze → Spill → Select

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
```
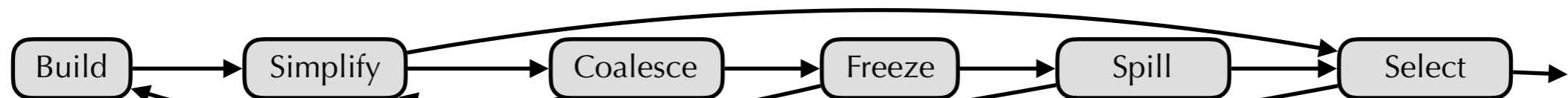
Example

Coalesce `c1` and `r3`, and then `c2` and `r3`
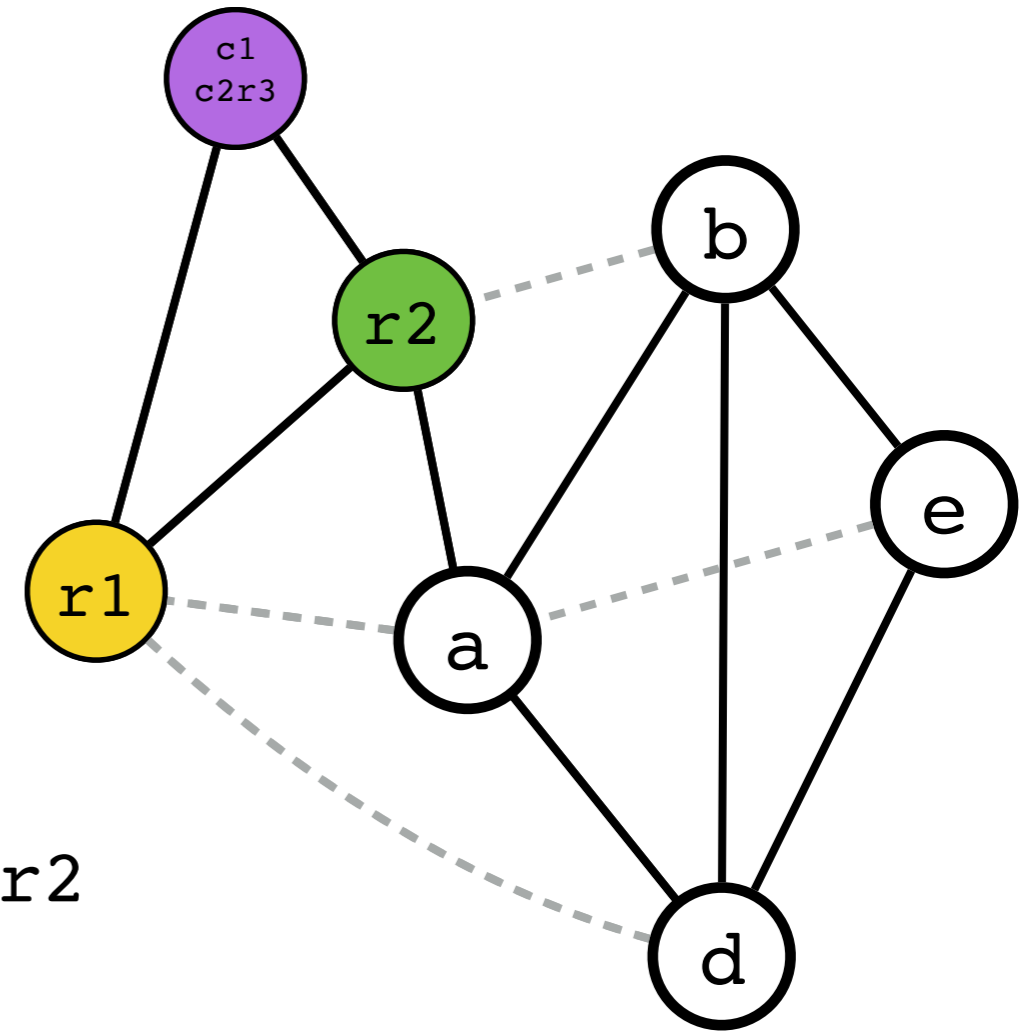
Build → Simplify → Coalesce → Freeze → Spill → Select

Example

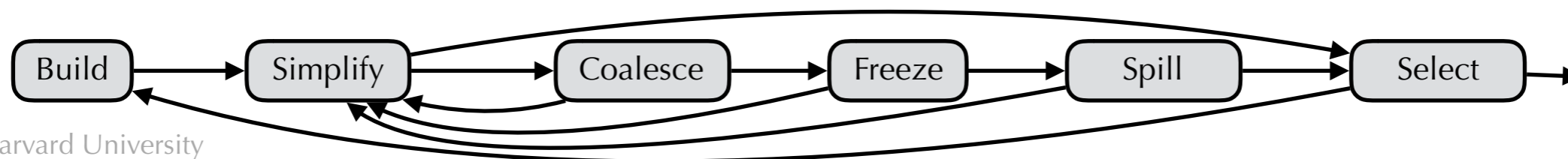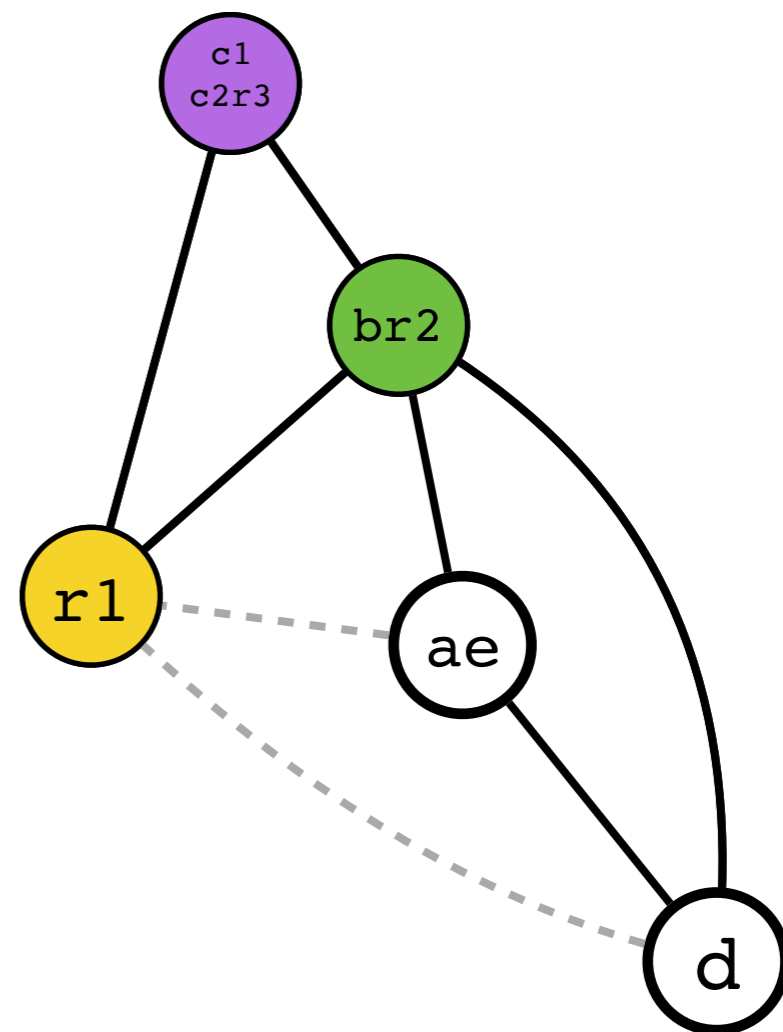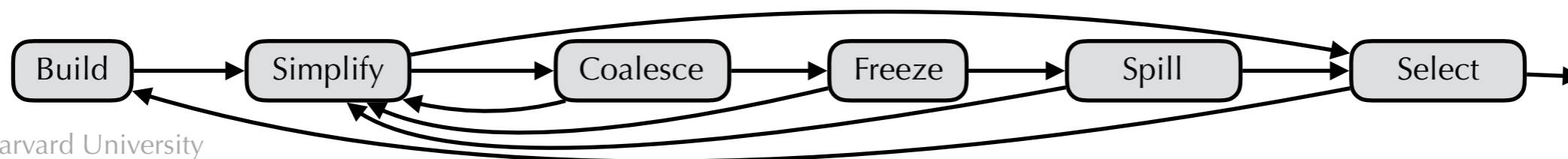Coalesce `c1` and `r3`, and then `c2` and `r3`

Example
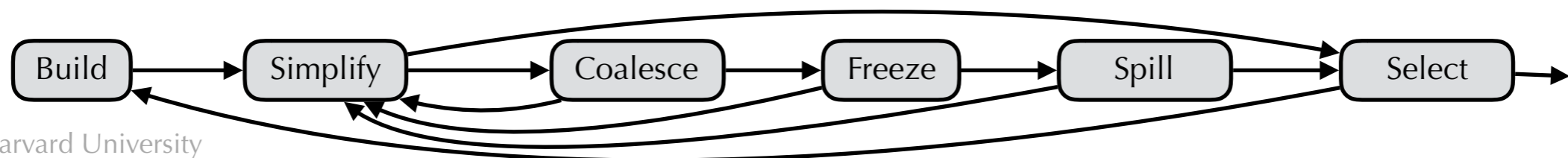
As before, coalesce **a** and **e**, and then **b** and **r2**

Example

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

Build → Simplify → Coalesce → Freeze → Spill → Select

Example

As before, coalesce `ae` and `r1`

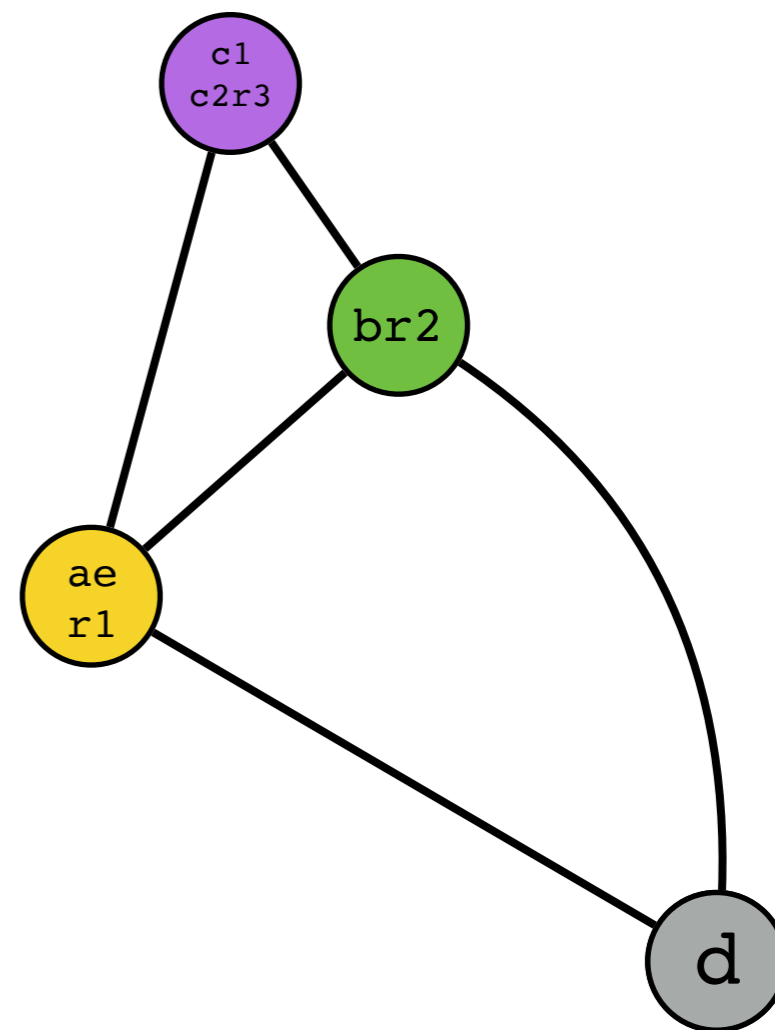Build → Simplify → Coalesce → Freeze → Spill → Select

Example

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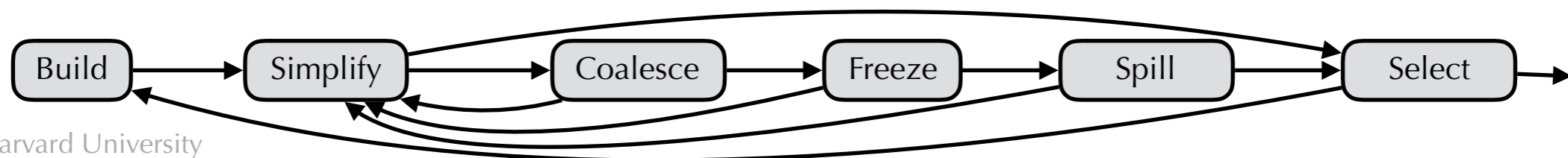
Example

Stack:

d

Simplify **d**
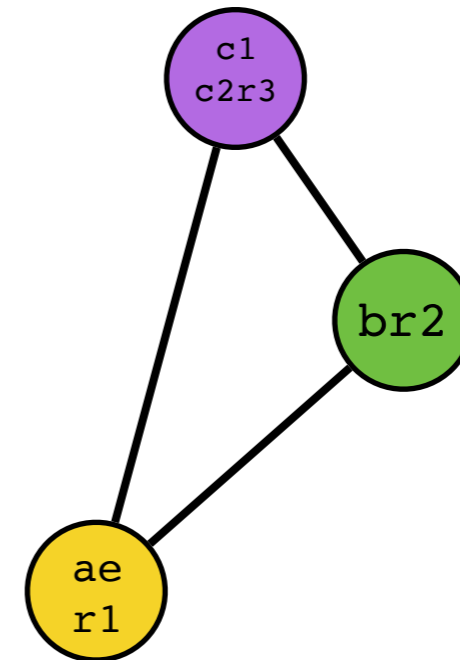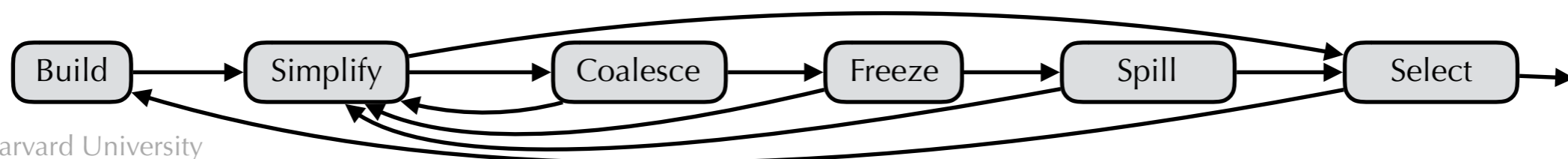
```
Build → Simplify → Coalesce → Freeze → Spill → Select →
```
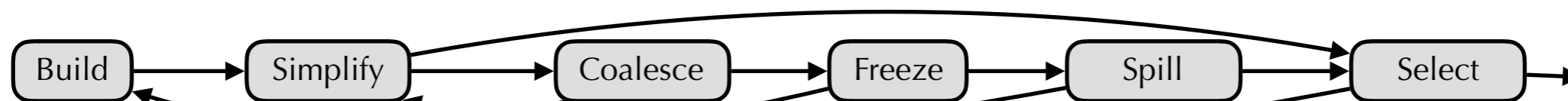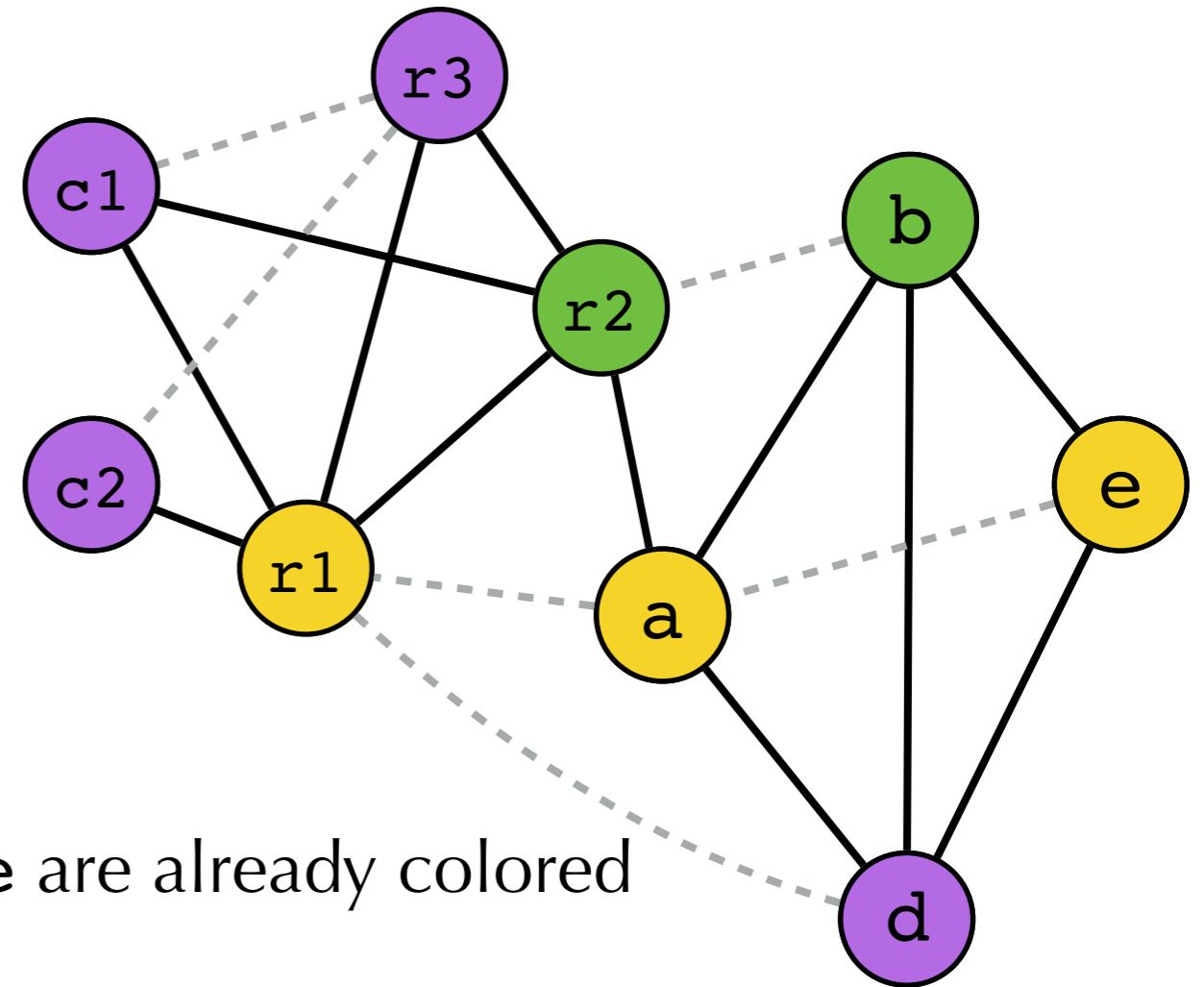
Example

Stack:

d

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

Build → Simplify → Coalesce → Freeze → Spill → Select

Example

Stack:

d

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

Pop **d** and color

Build → Simplify → Coalesce → Freeze → Spill → Select

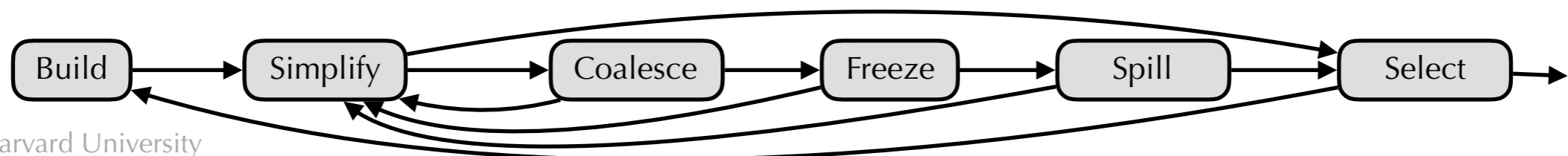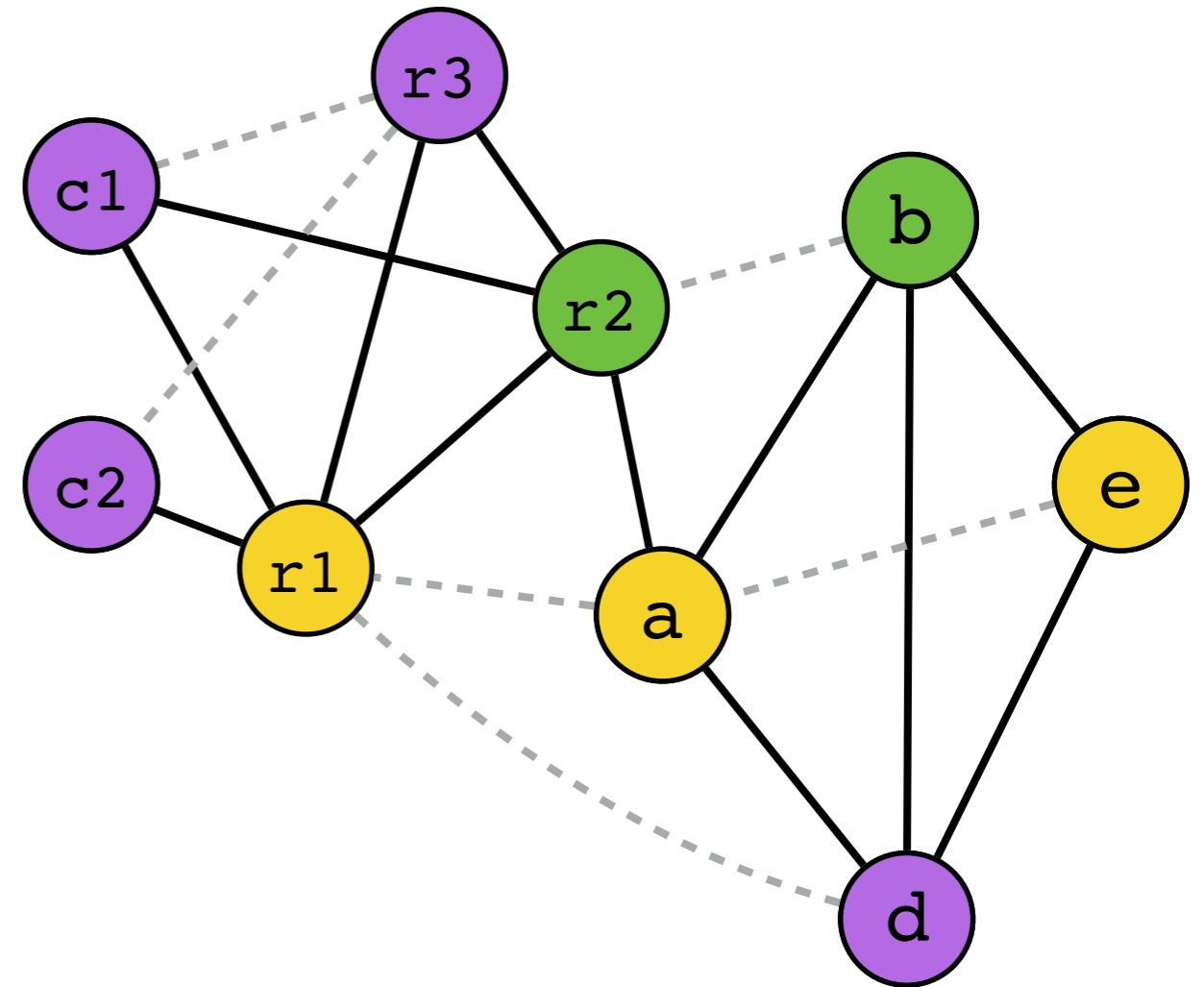# Example

```
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
```


56
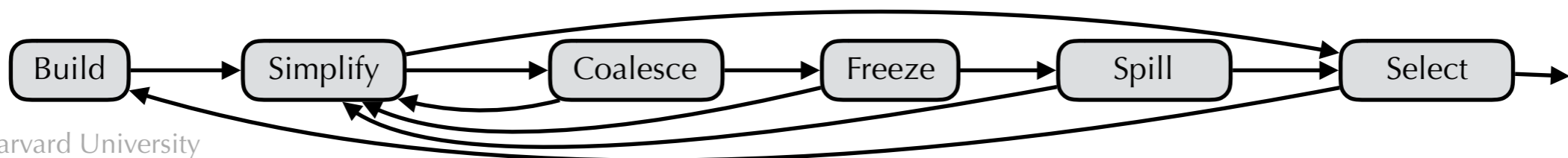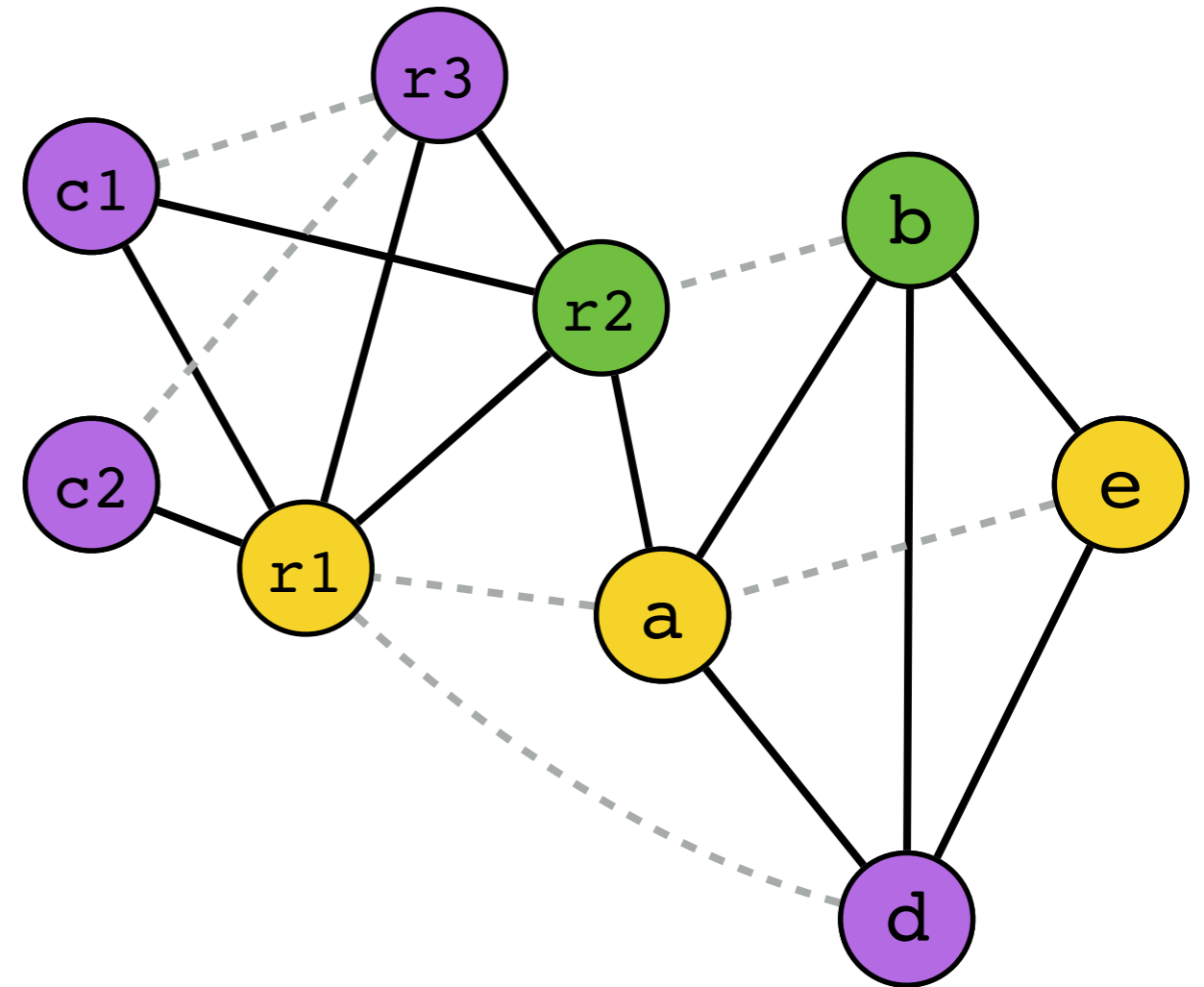
Example

```
f: $r3 := $r3
   Mem[fp+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[fp+i]
   $r3 := $r3
   return
```


Build → Simplify → Coalesce → Freeze → Spill → Select

Example

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

Only one non-coalesced move remains!

Build → Simplify → Coalesce → Freeze → Spill → Select