Register Allocation

Eelco Visser

TU Delft

CS4200 | Compiler Construction | November 25, 2021

Allocate Minimal Number of Registers

```
live-in: k j
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 k j
```

Overview

Interference graphs

- construction during liveness analysis

Graph Coloring

- assign registers to local variables and compiler temporaries
- store local variables and temporaries in memory

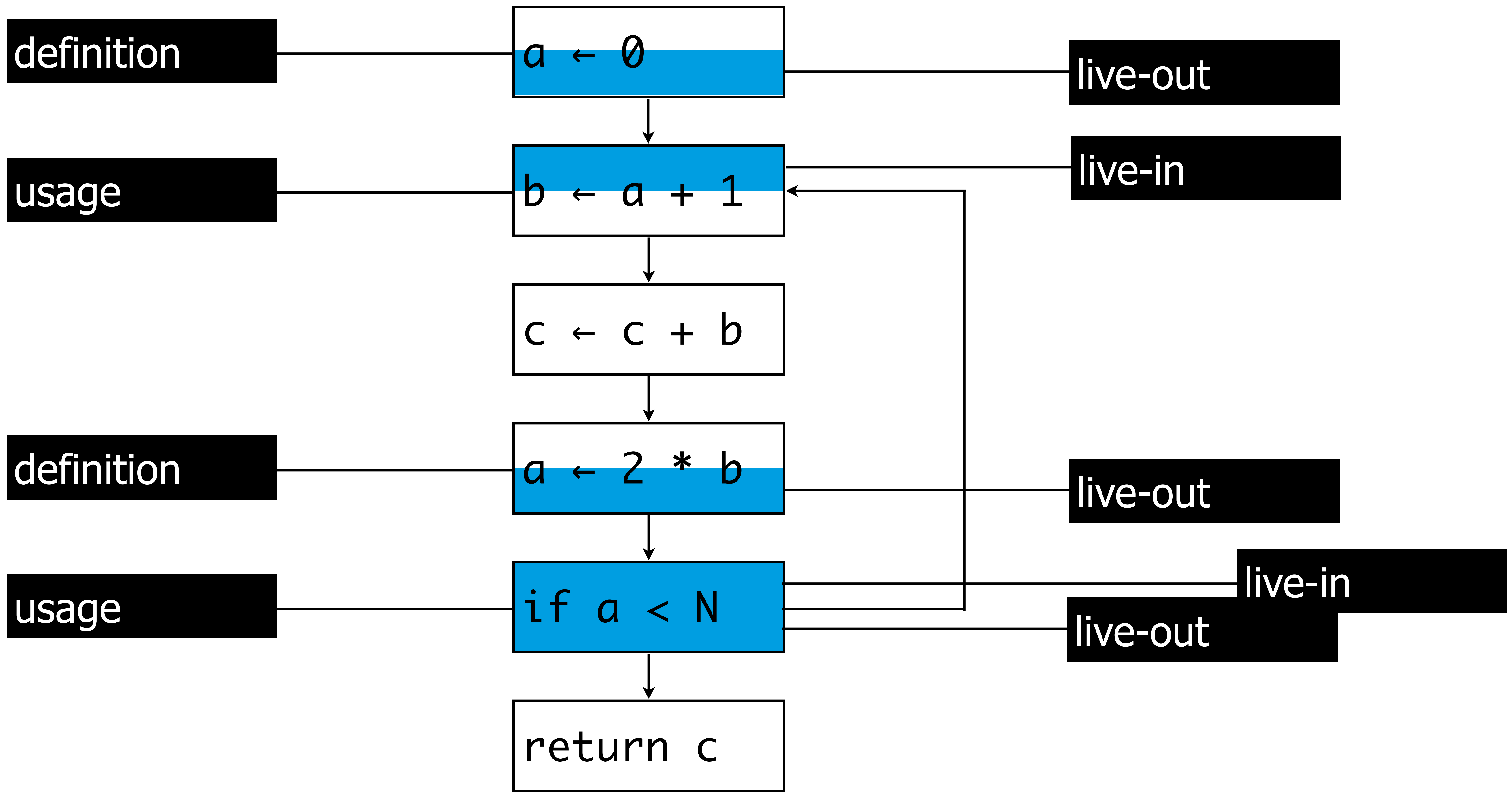
Coalescing

- handle move instructions

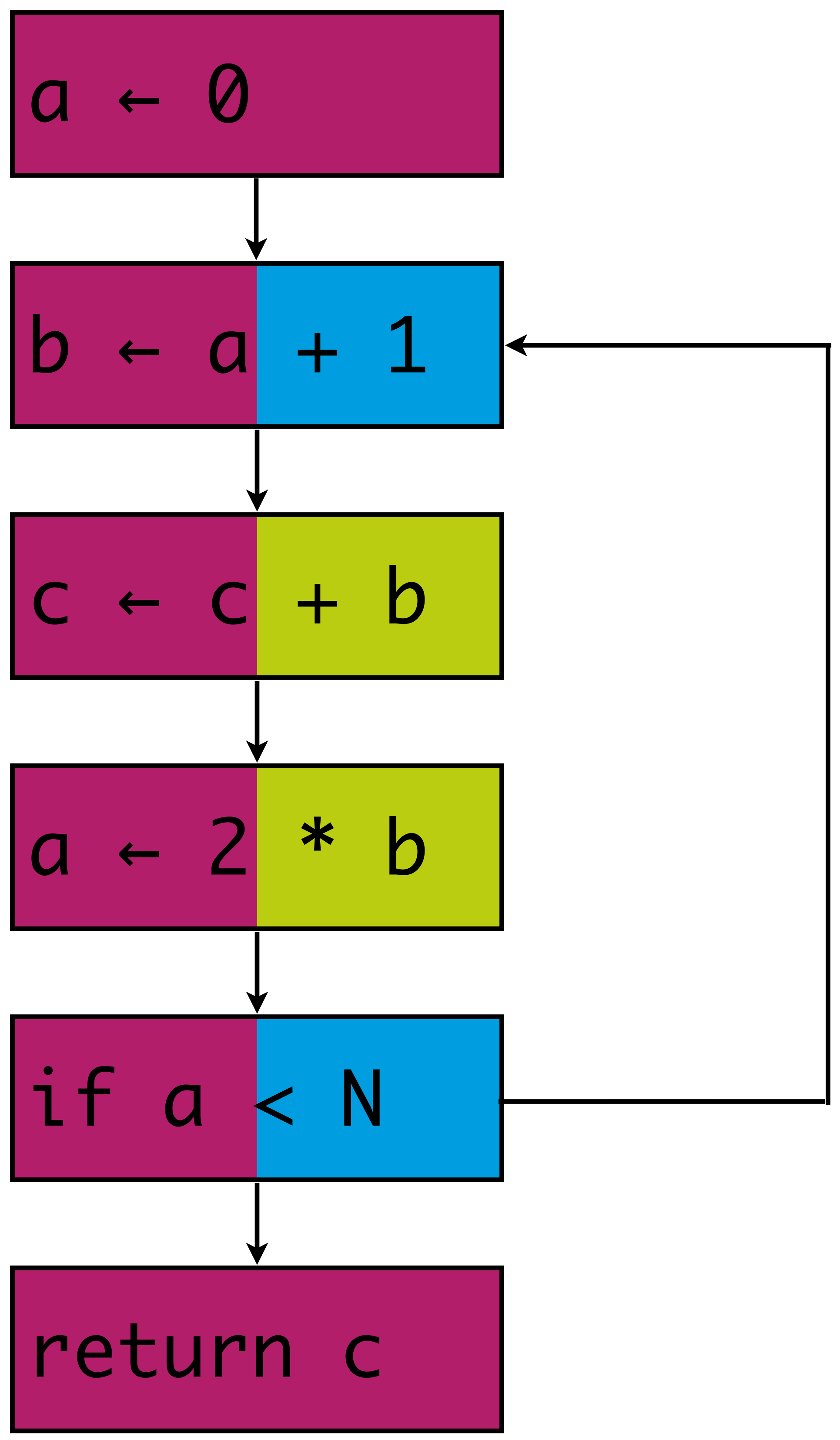
Pre-colored nodes

Interference Graphs

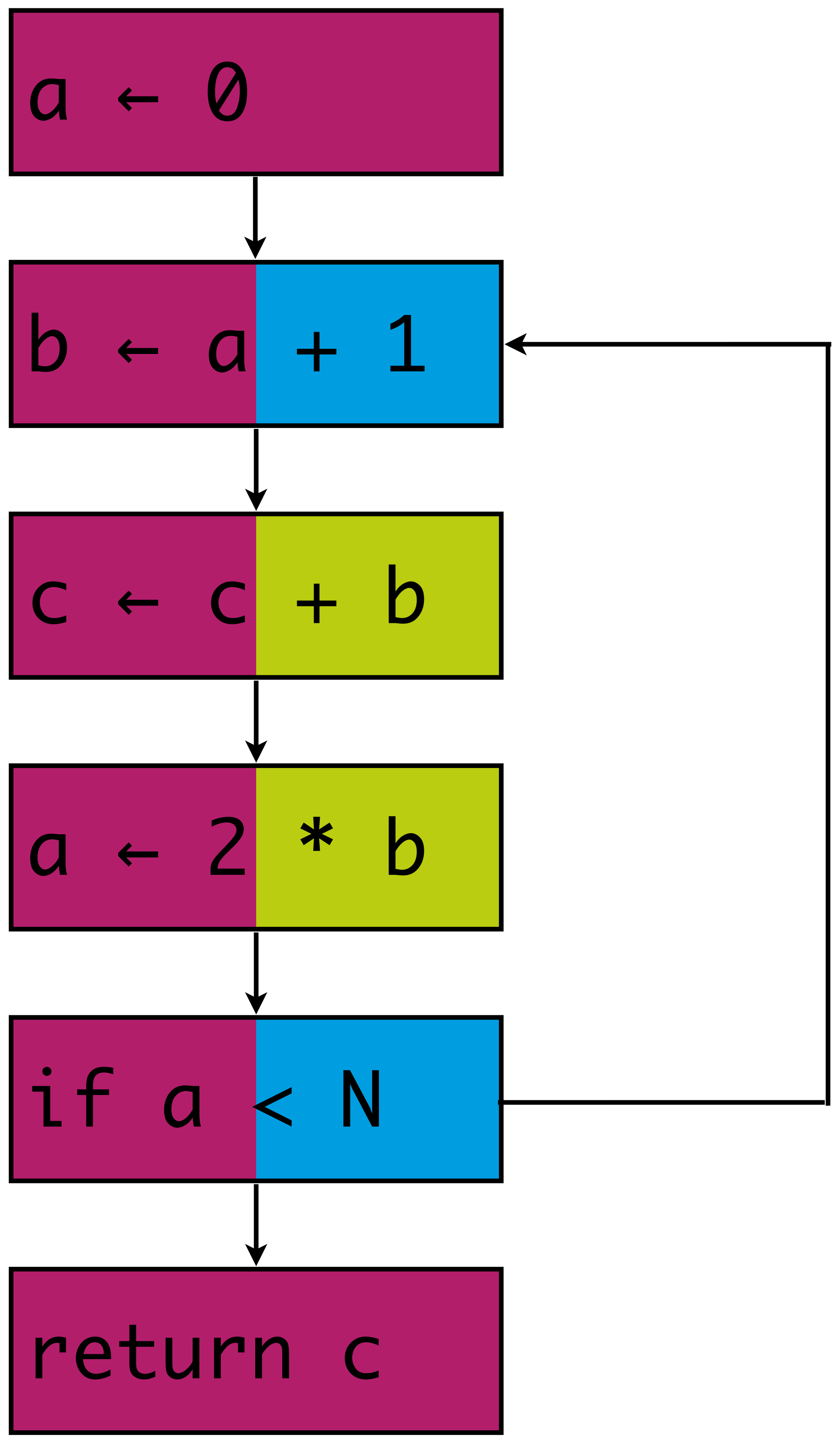
Liveness Analysis

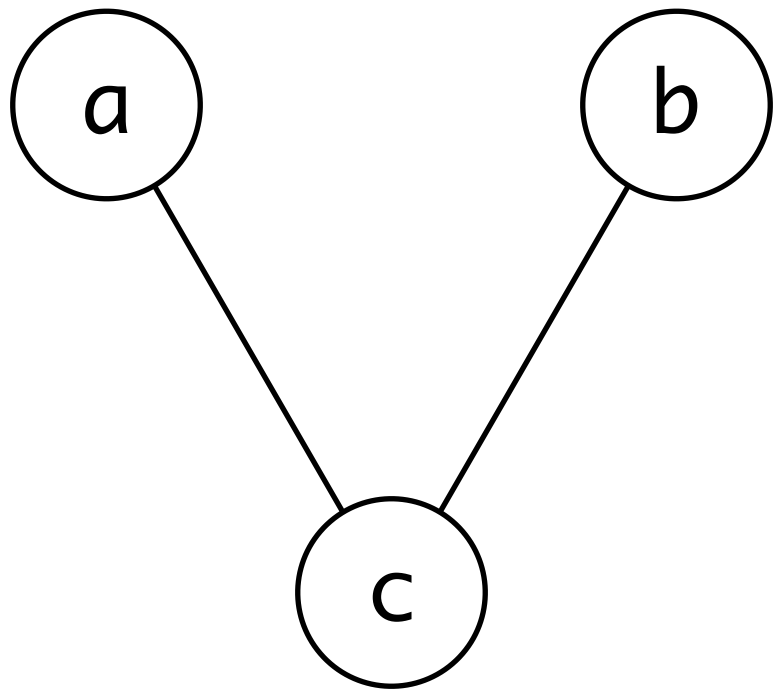


Liveness Analysis



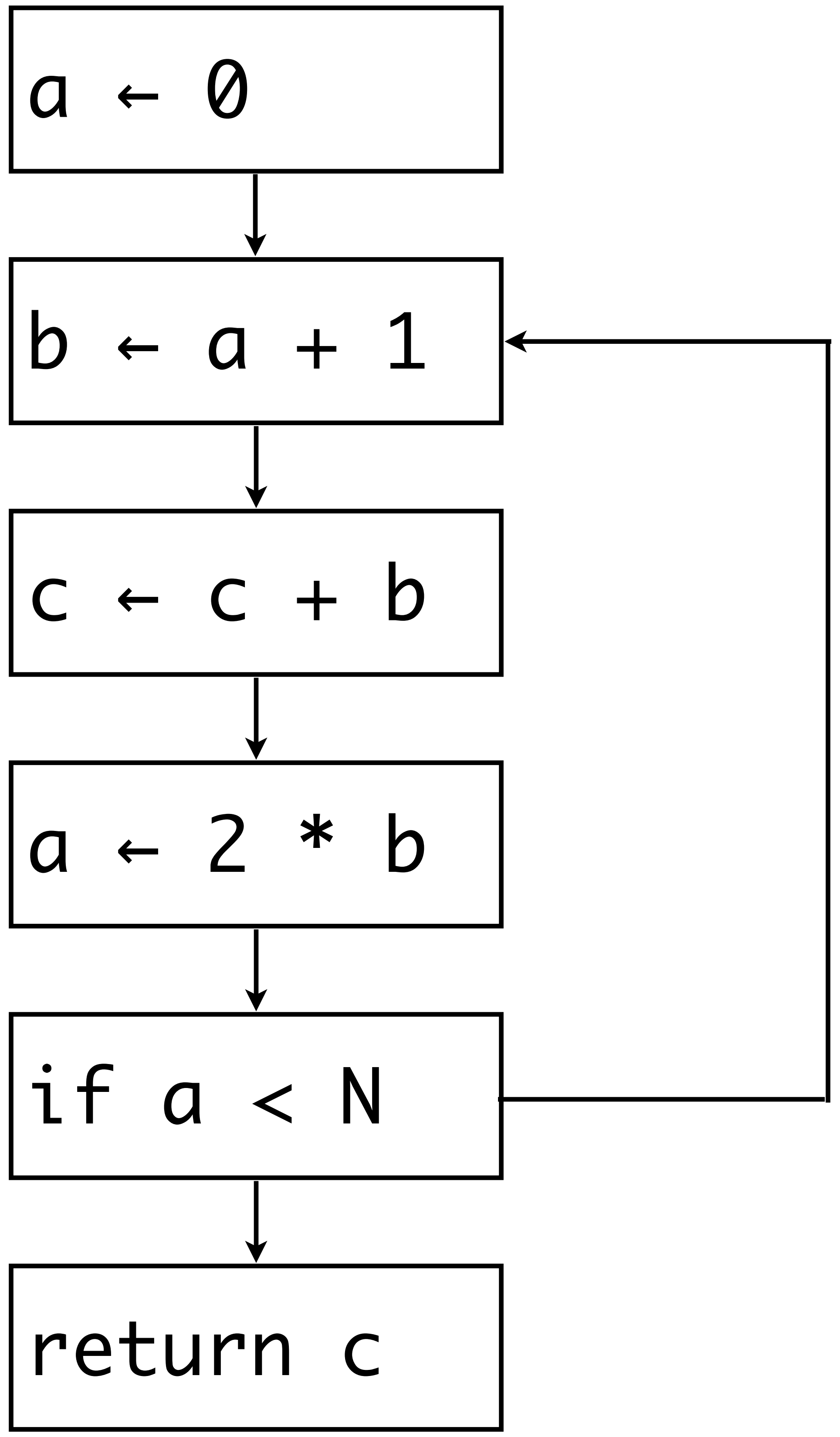
Interference Graphs

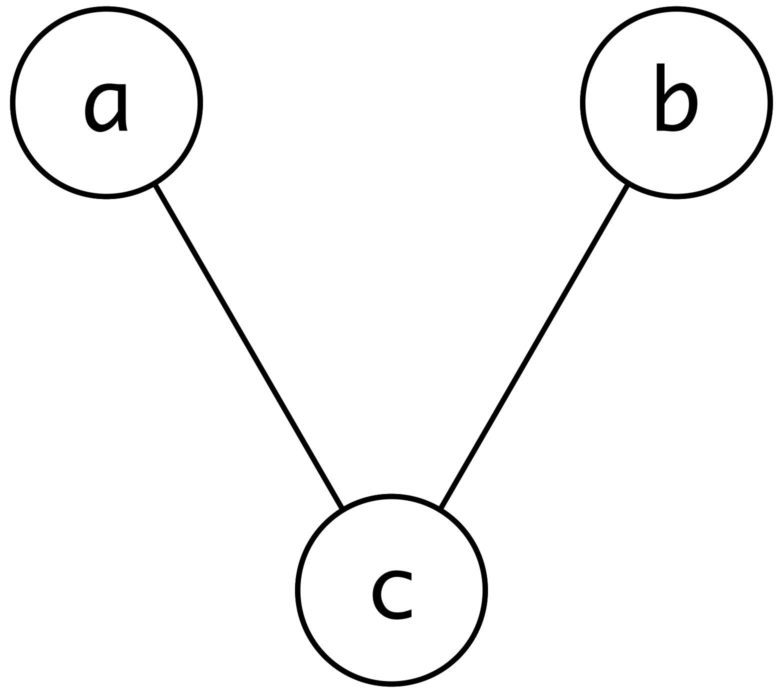




Graph Coloring

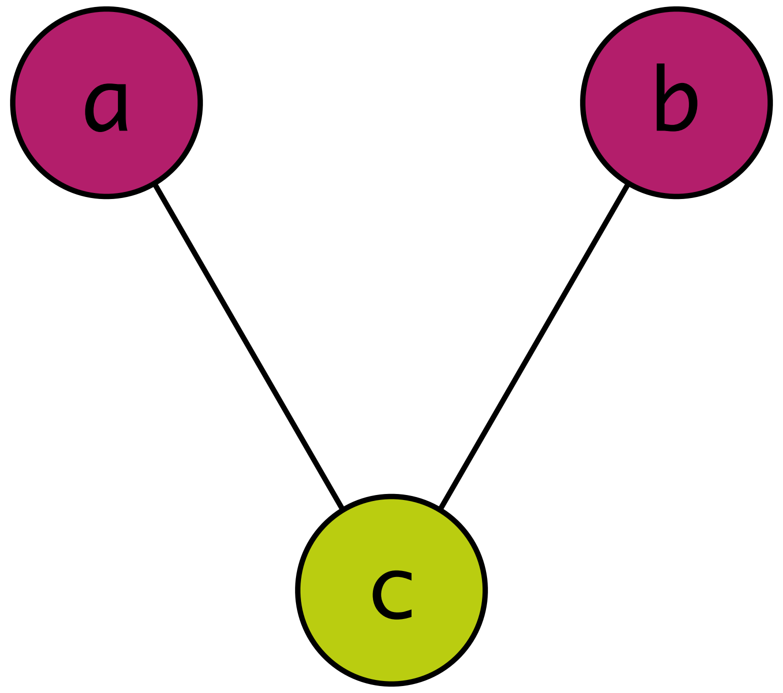
Graph Coloring





Graph Coloring

```
r1 ← 0
r1 ← r1 + 1
r2 ← r2 + r1
r1 ← 2 * r1
if r1 < N
return r2
```



Graph Coloring: Steps

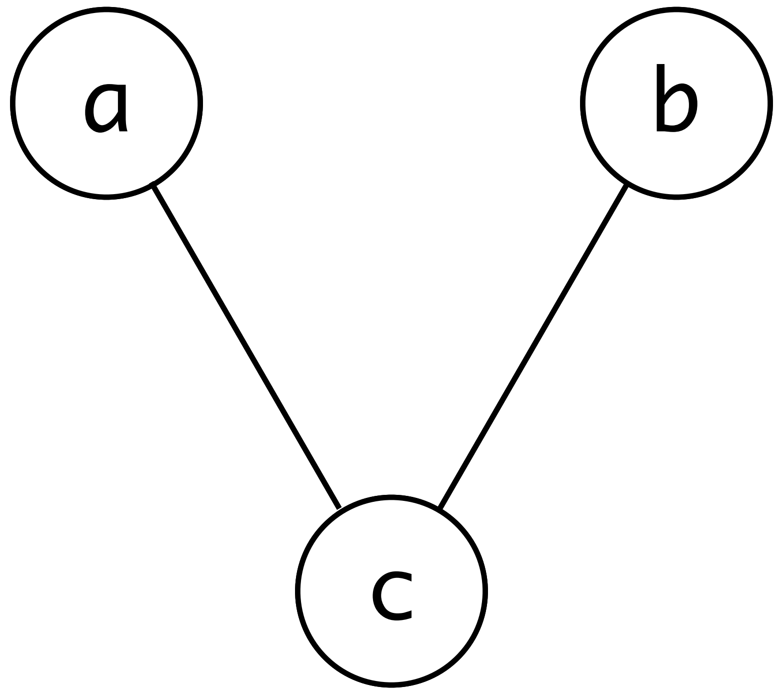
Simplify

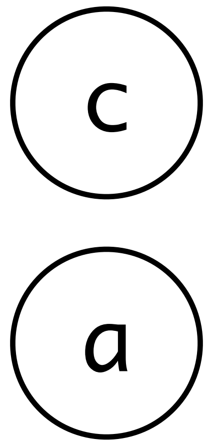
– remove node of insignificant degree (fewer than k edges)

Select

– add node, select color

Graph Coloring: Example with 2 Colors

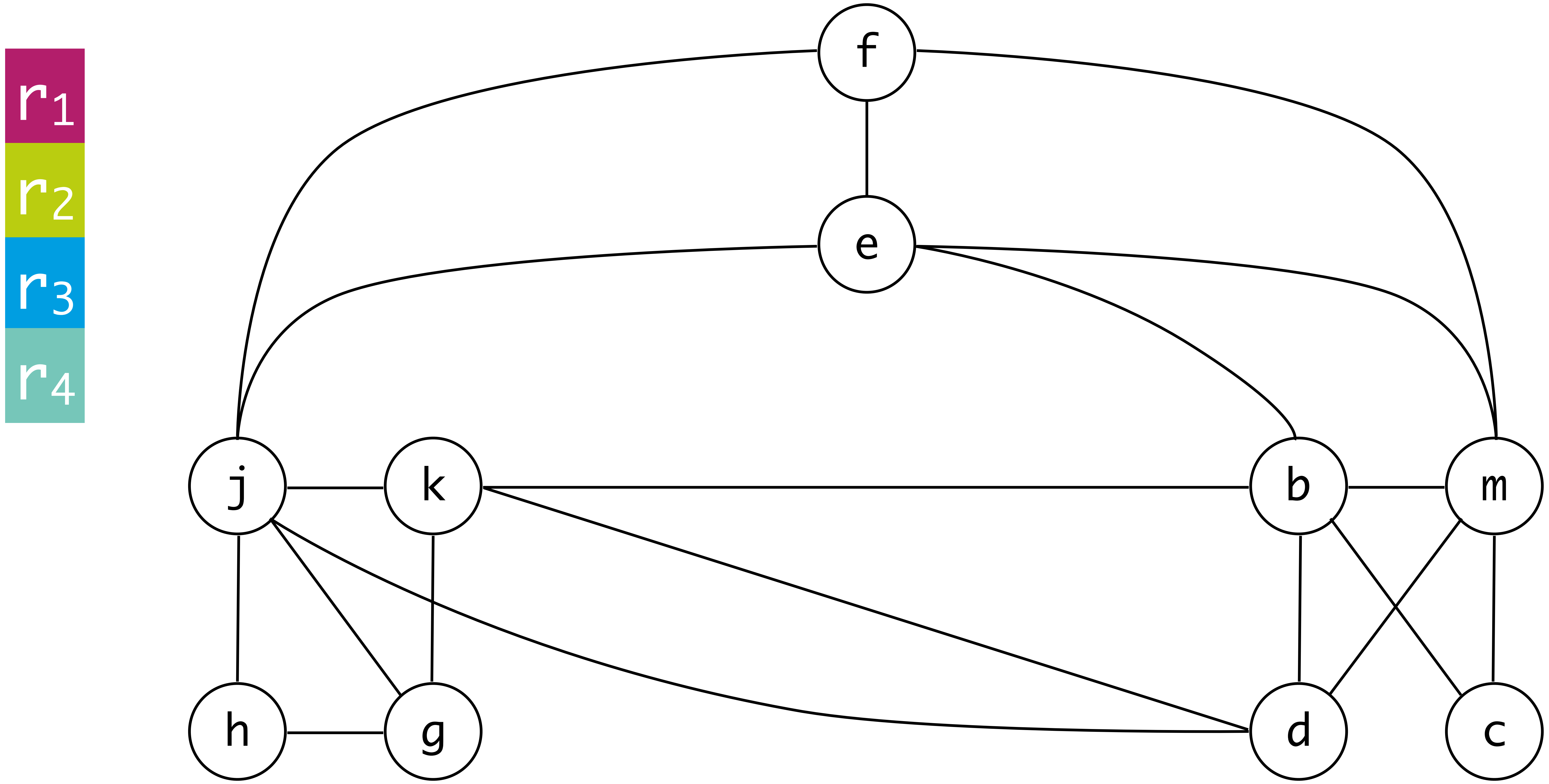




Graph Coloring

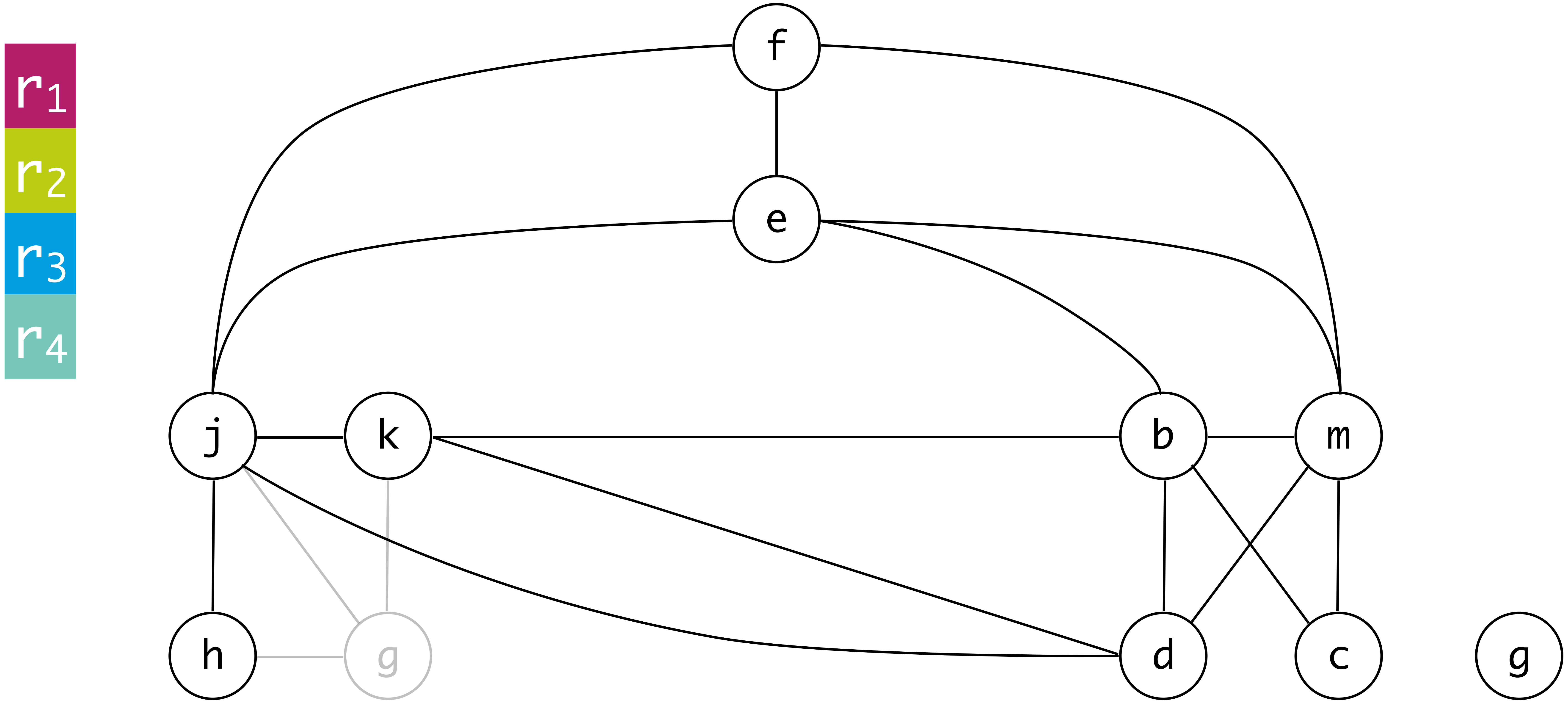
```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Graph Coloring: Example with 4 Colors

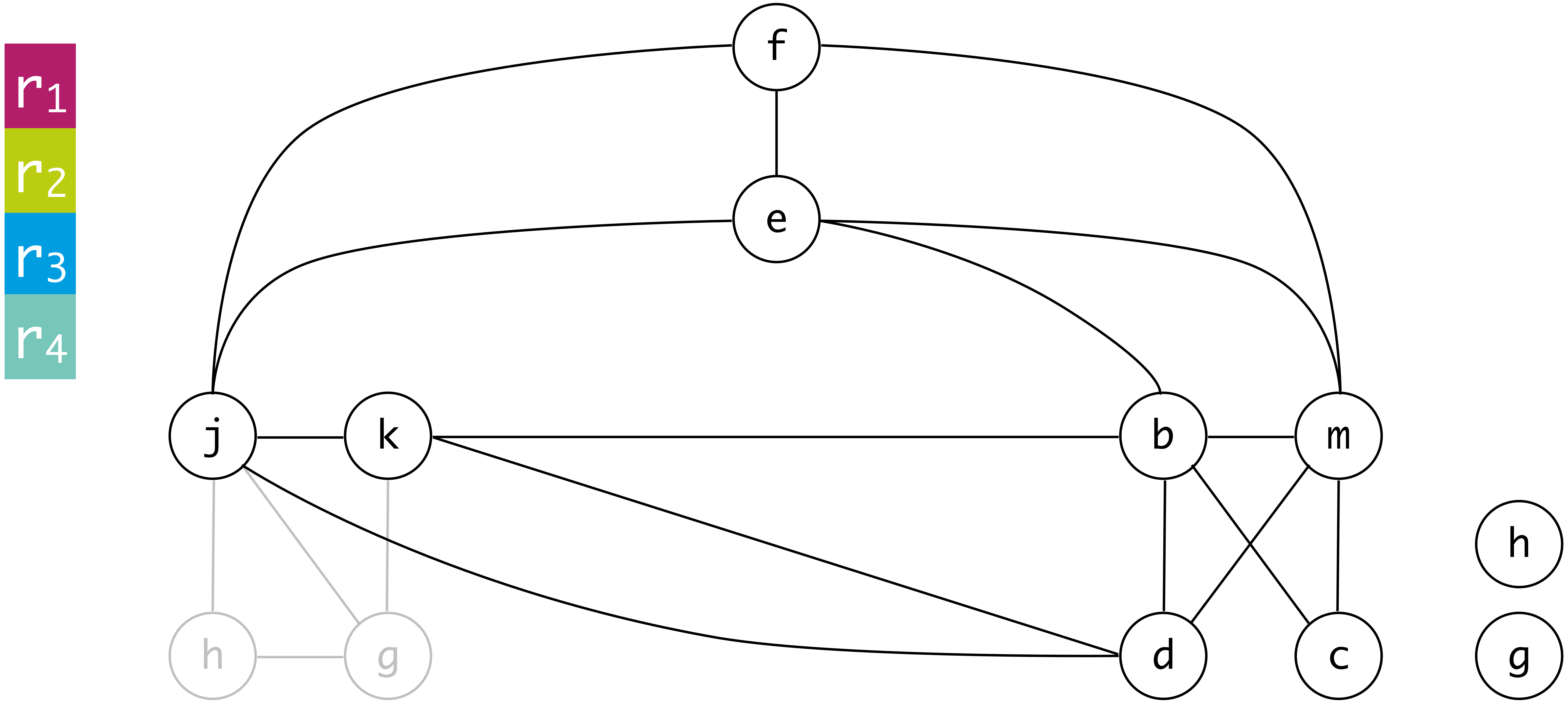



```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

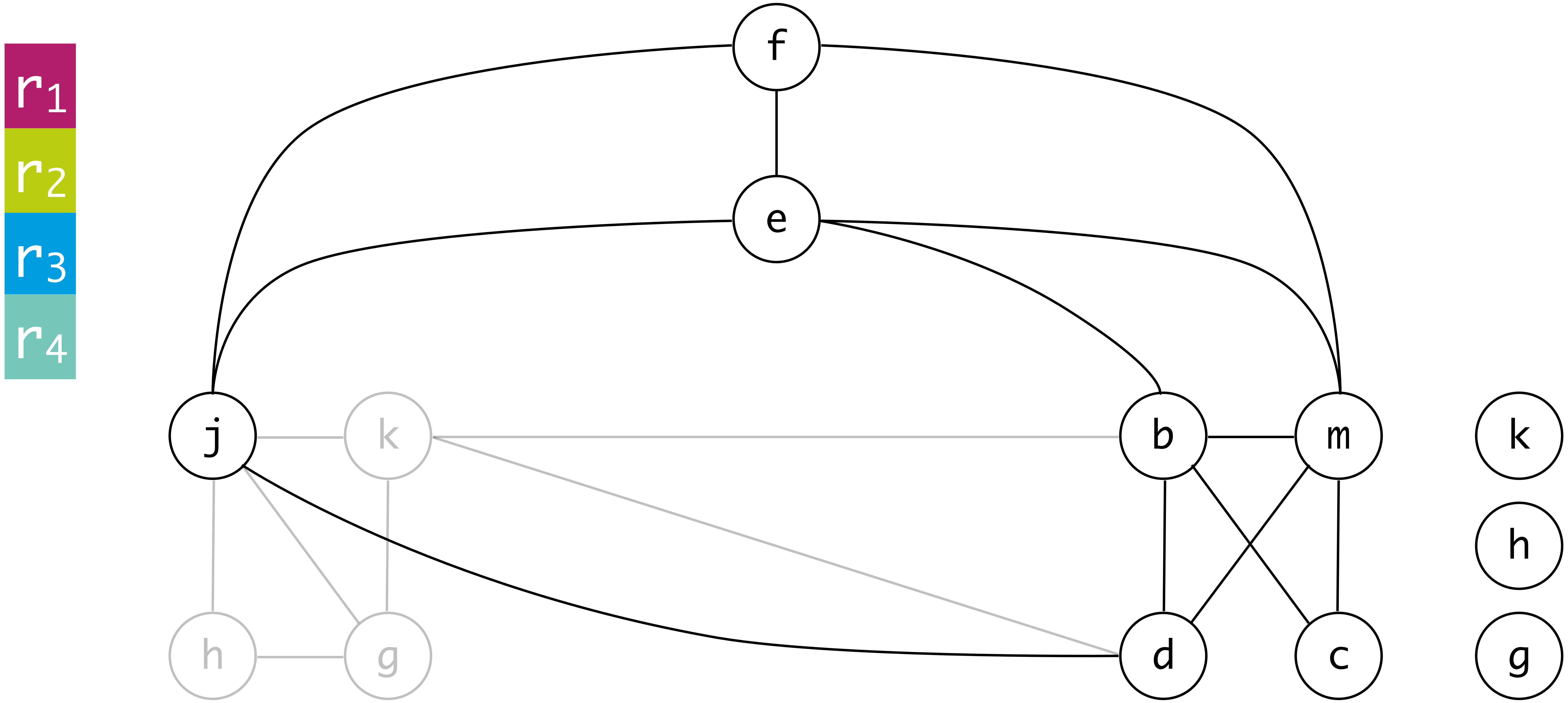
Graph Coloring: Example with 4 Colors



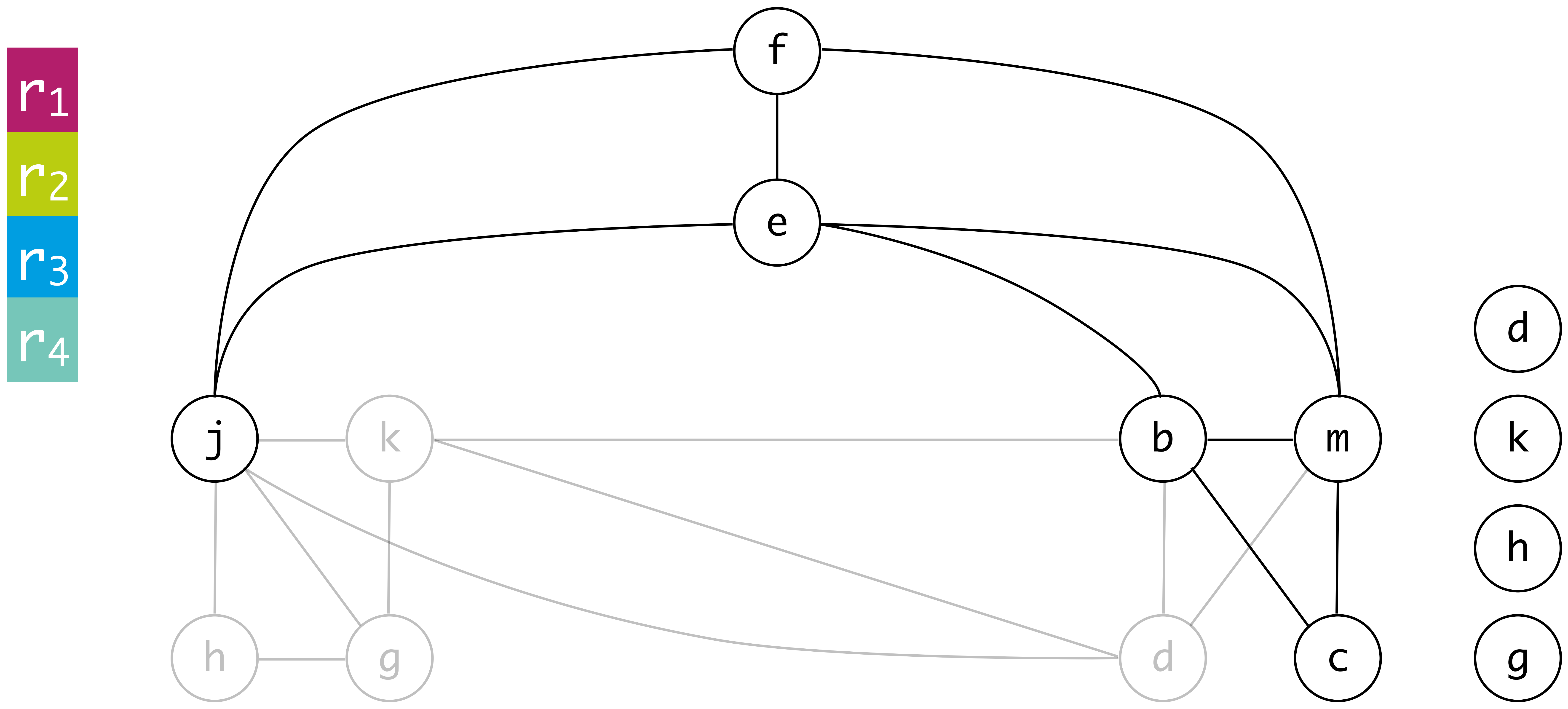
Graph Coloring: Example with 4 Colors



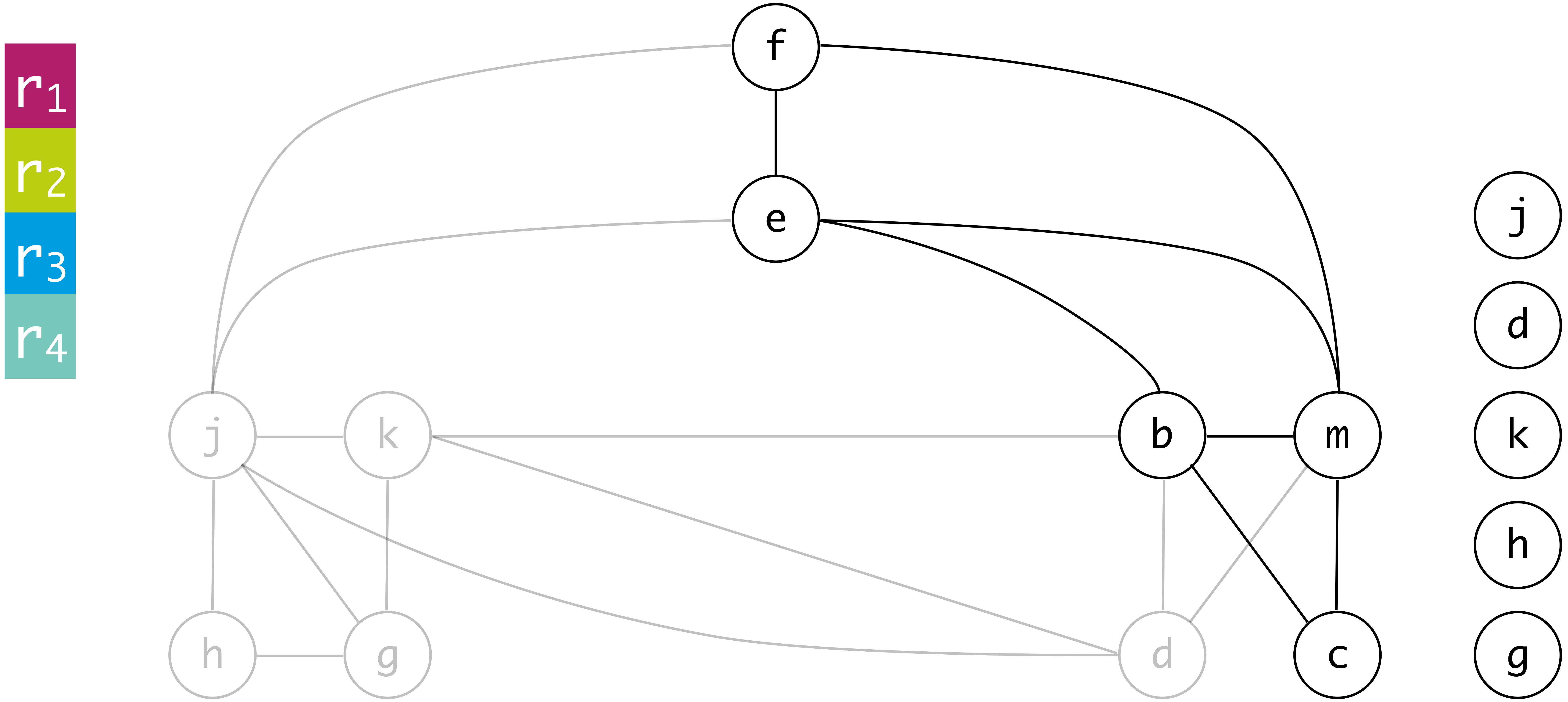
Graph Coloring: Example with 4 Colors



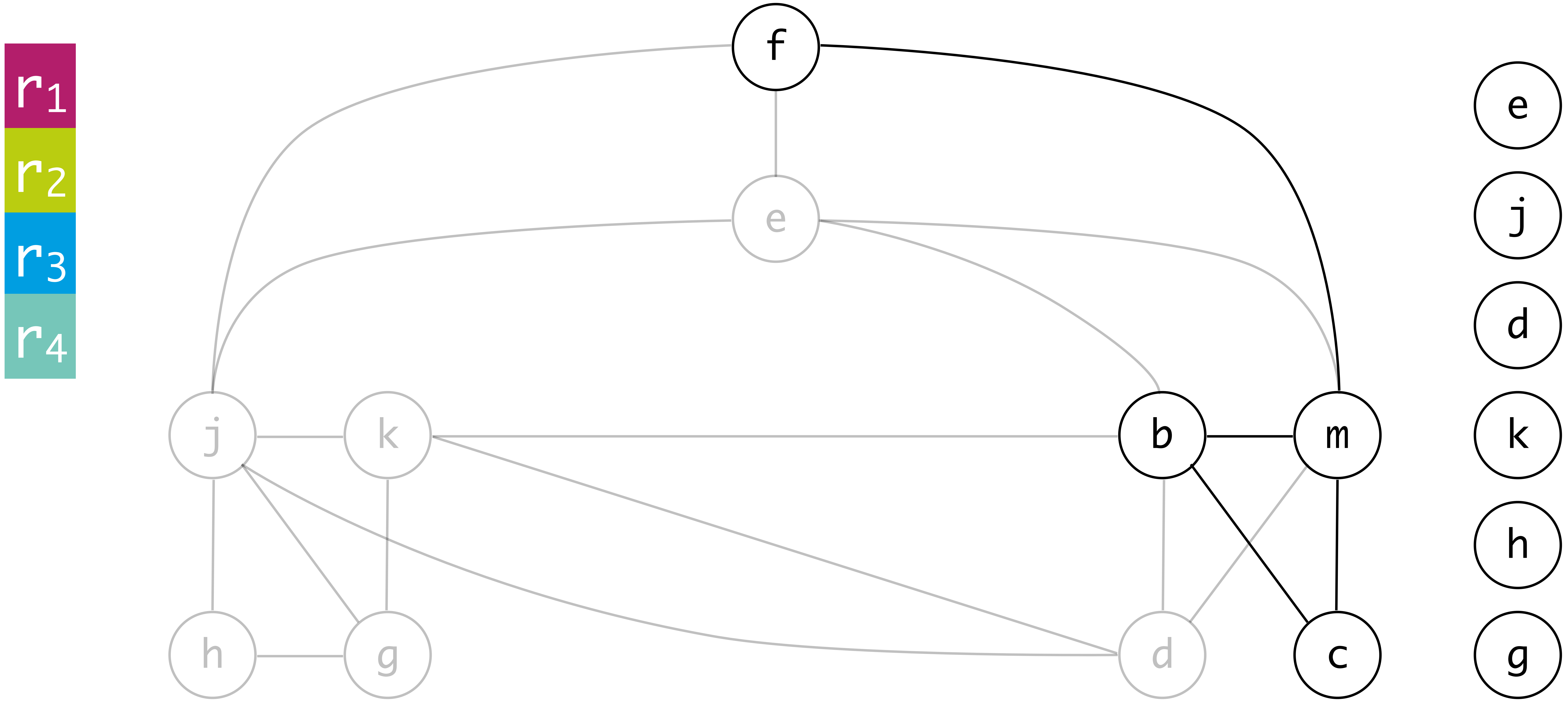
Graph Coloring: Example with 4 Colors



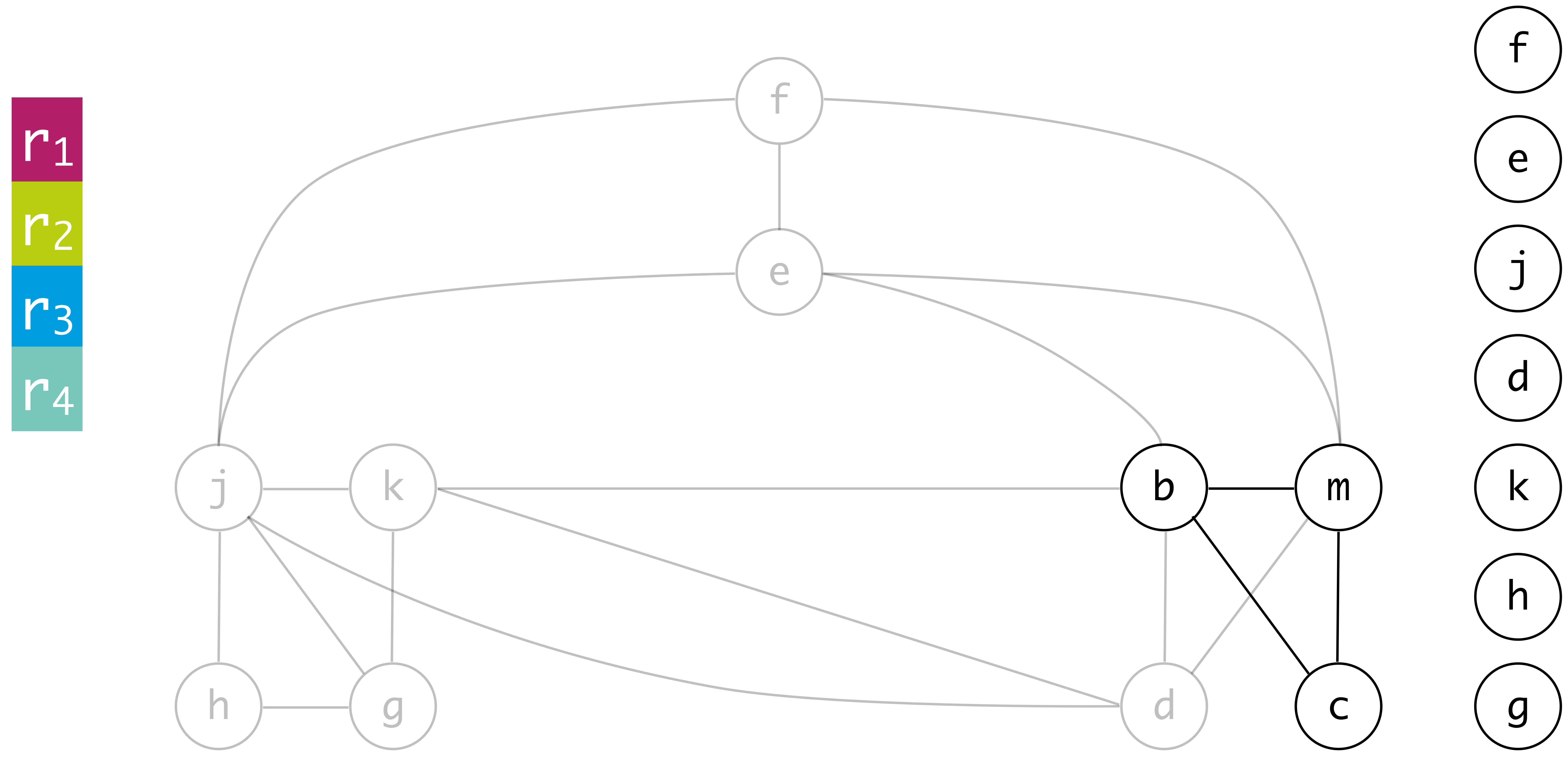
Graph Coloring: Example with 4 Colors



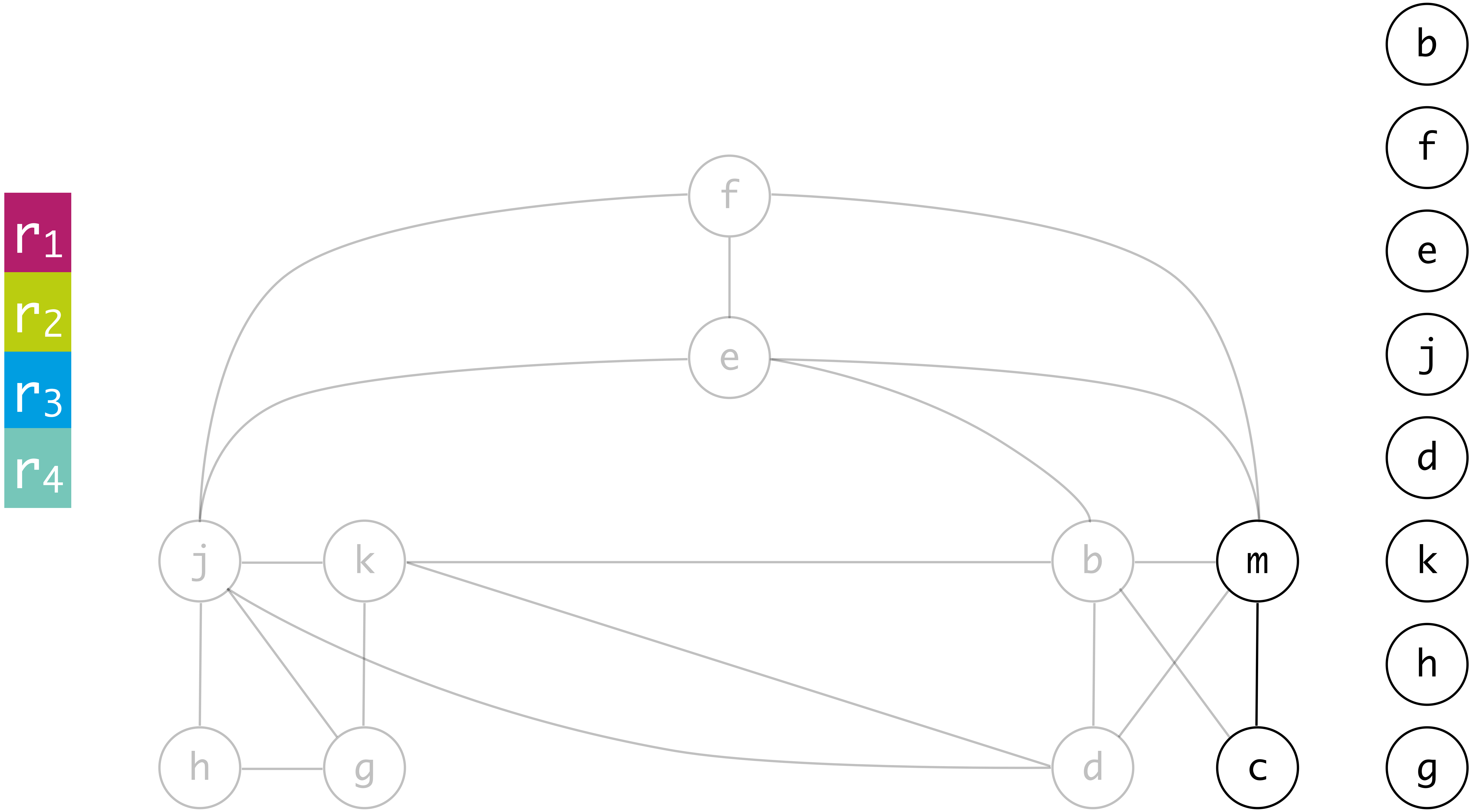
Graph Coloring: Example with 4 Colors



Graph Coloring: Example with 4 Colors

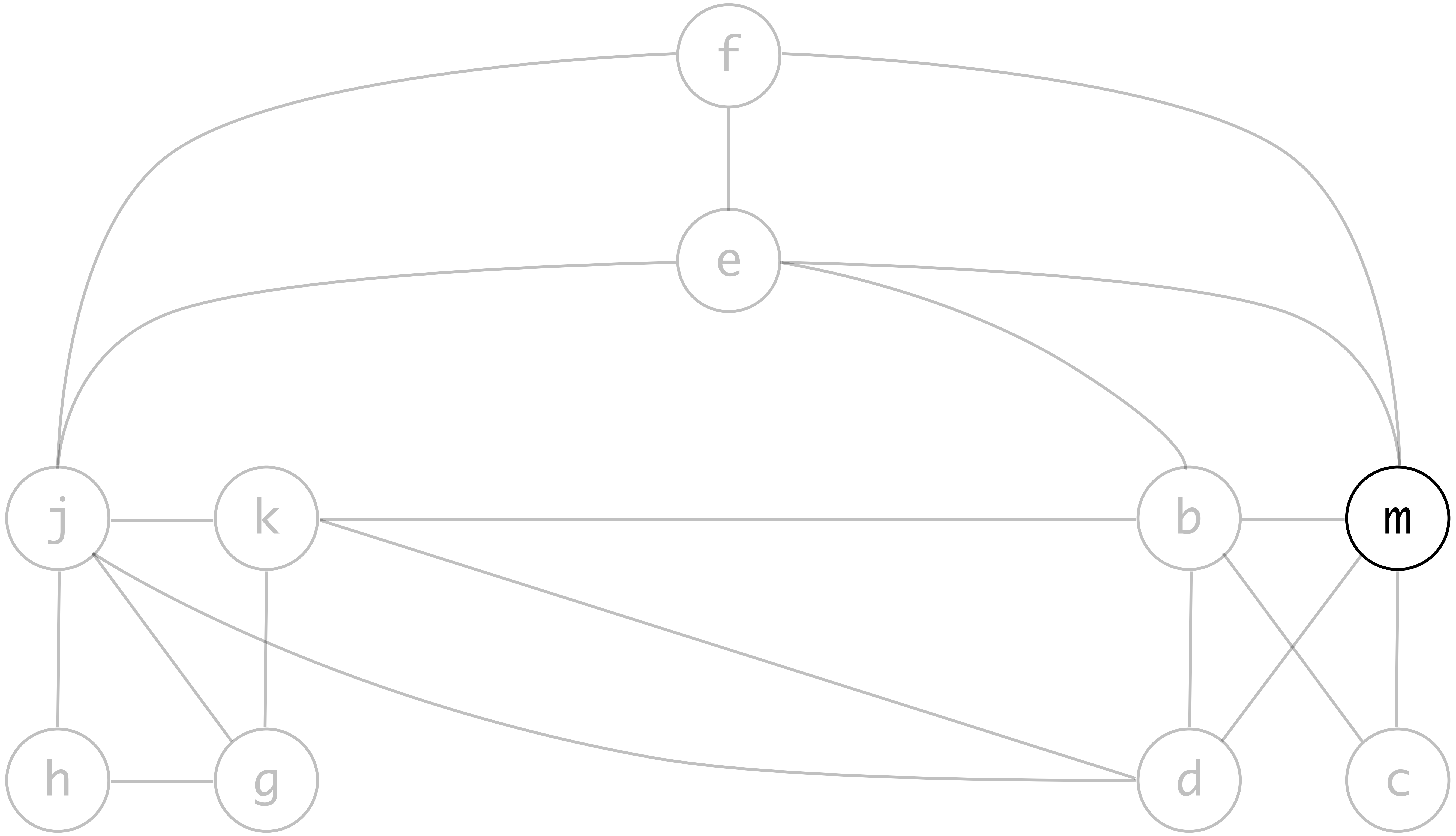


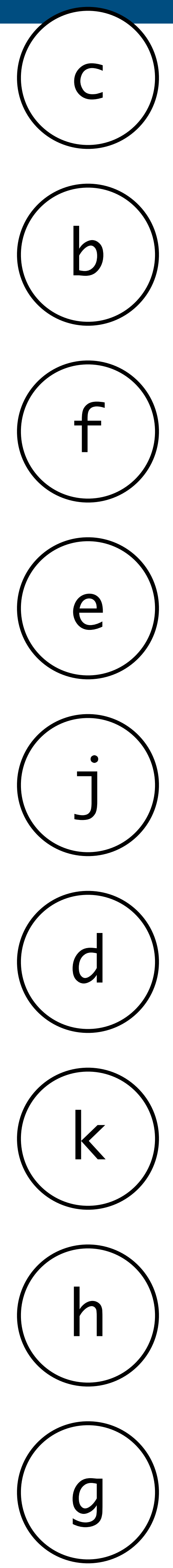
Graph Coloring: Example with 4 Colors



Graph Coloring: Example with 4 Colors

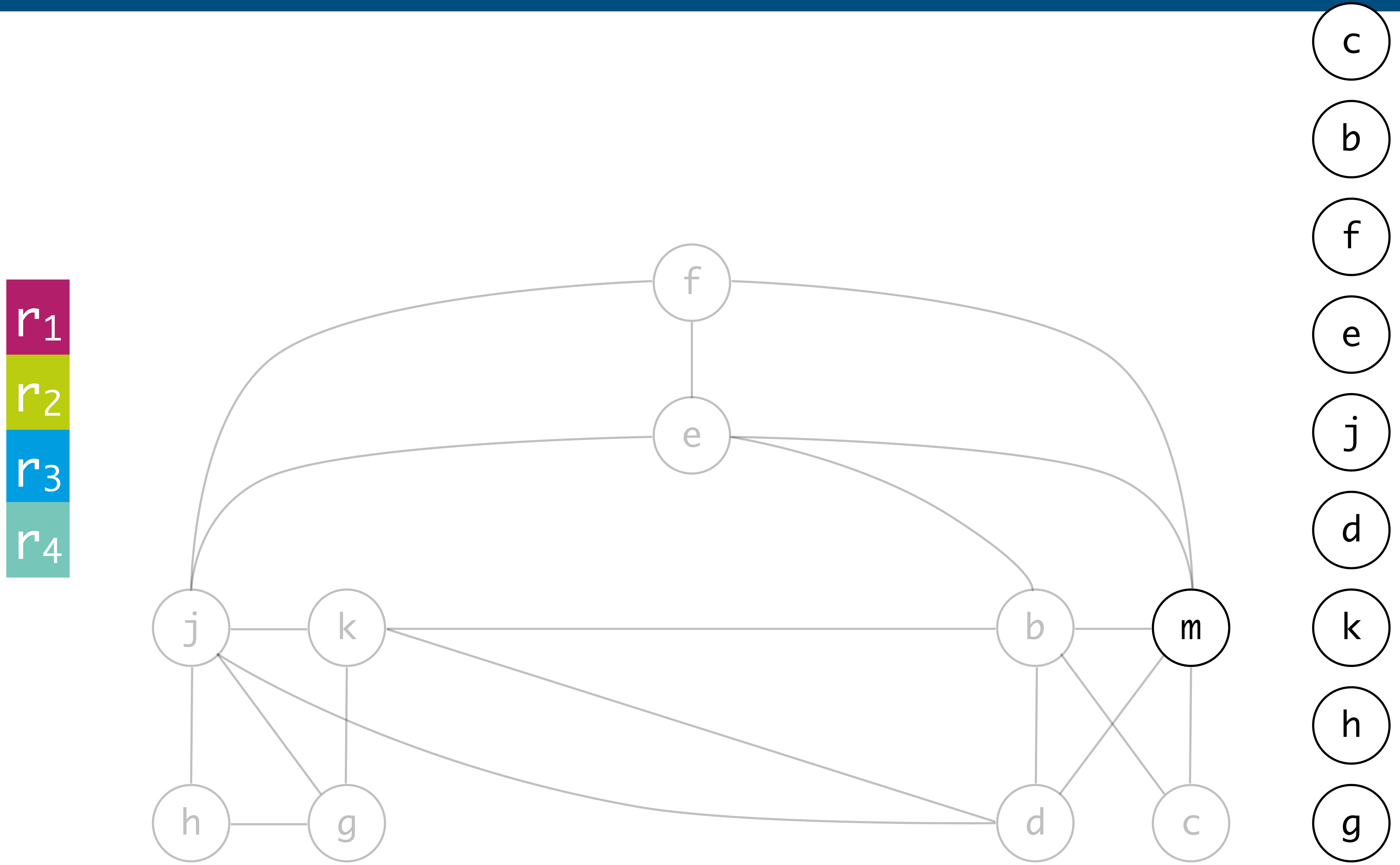
r_1
r_2
r_3
r_4





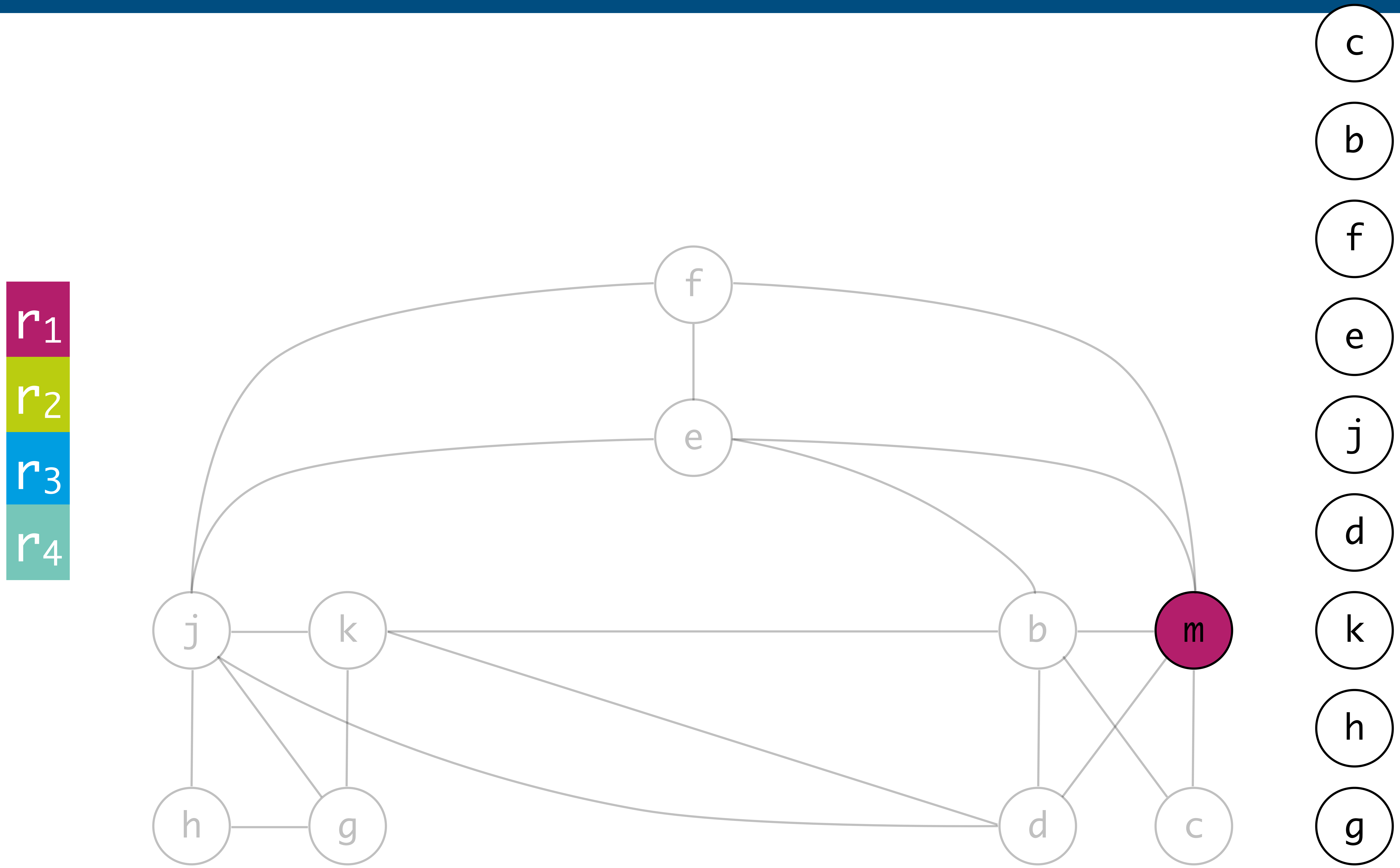

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Graph Coloring: Example with 4 Colors



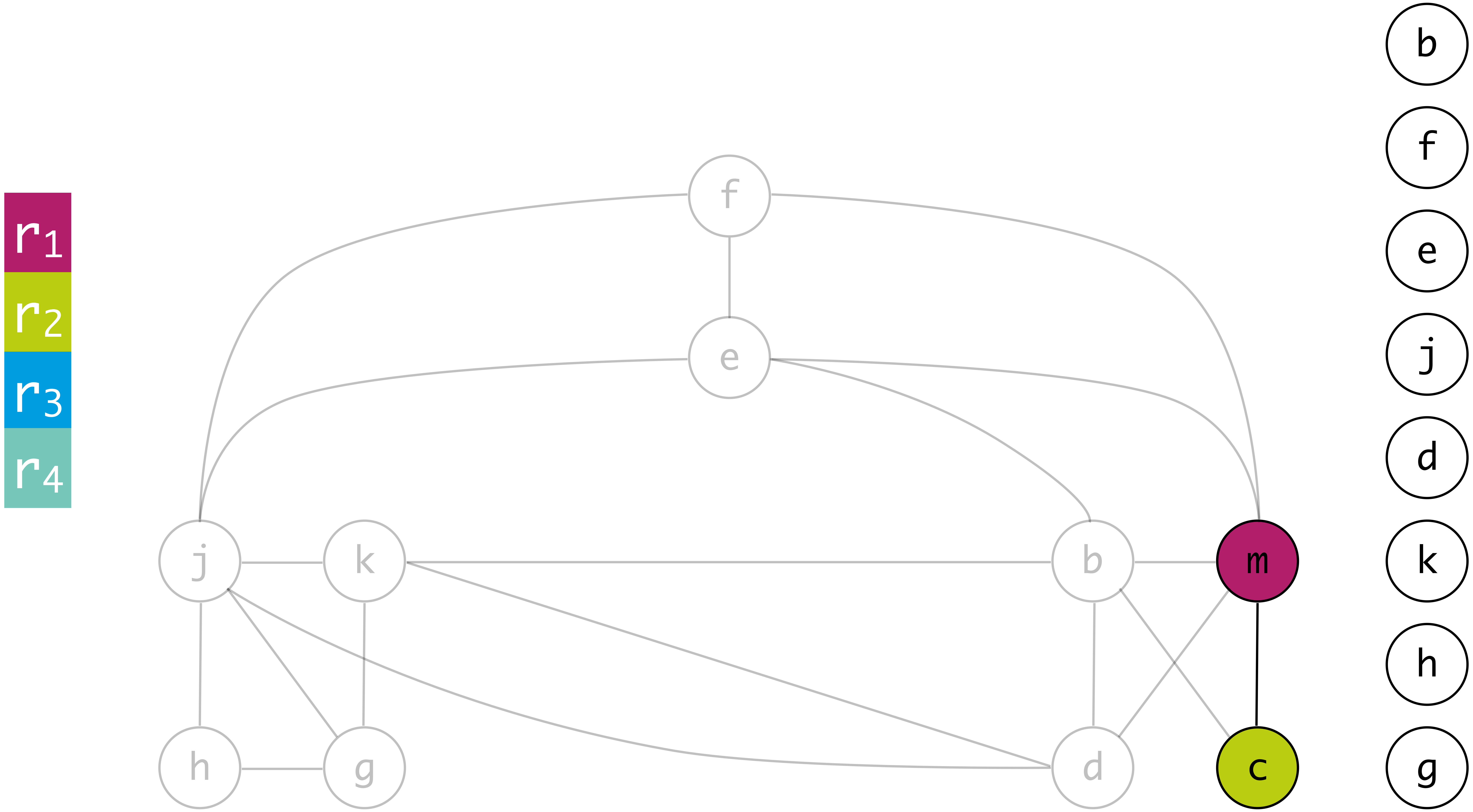

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Graph Coloring



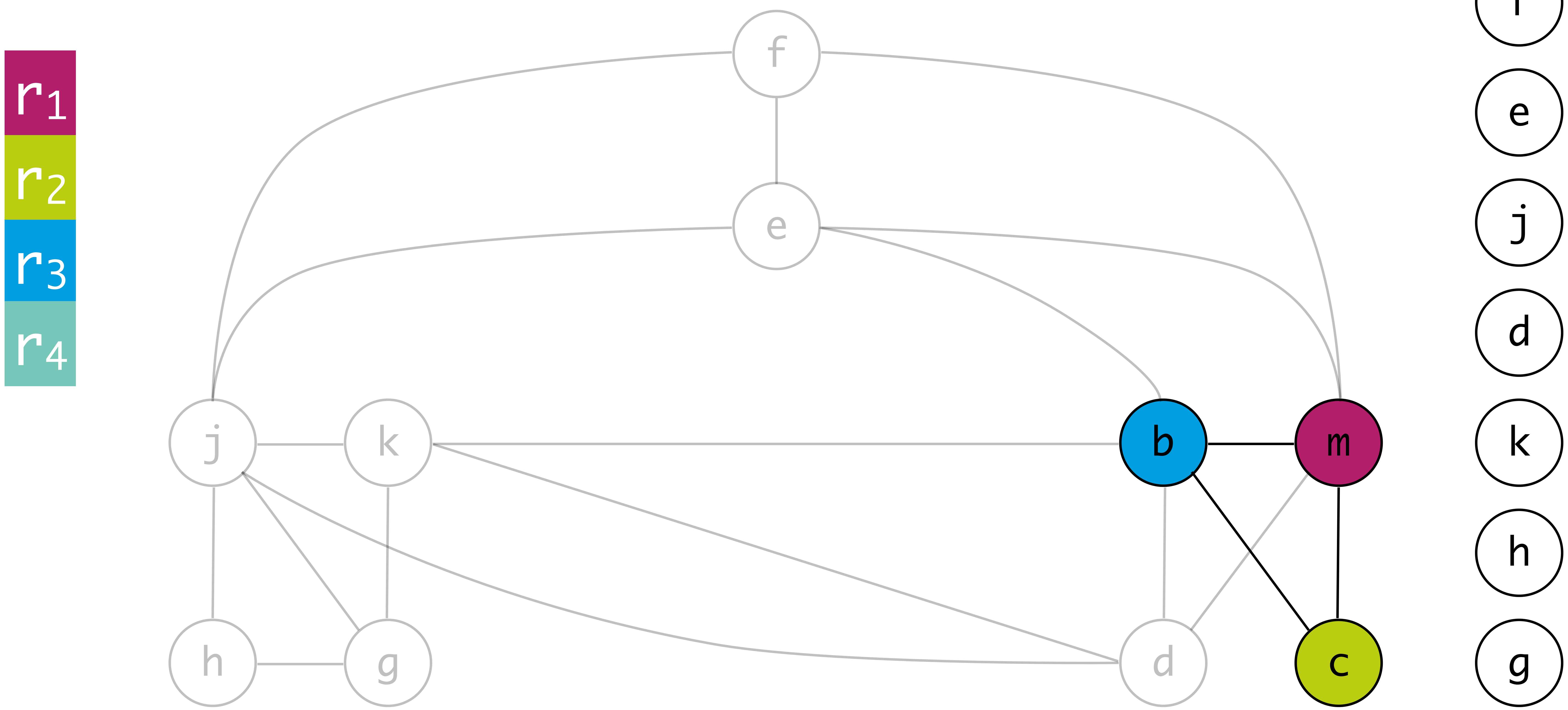

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
r1 := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := r1 + 4
j := b
live out: d k j
```

Graph Coloring



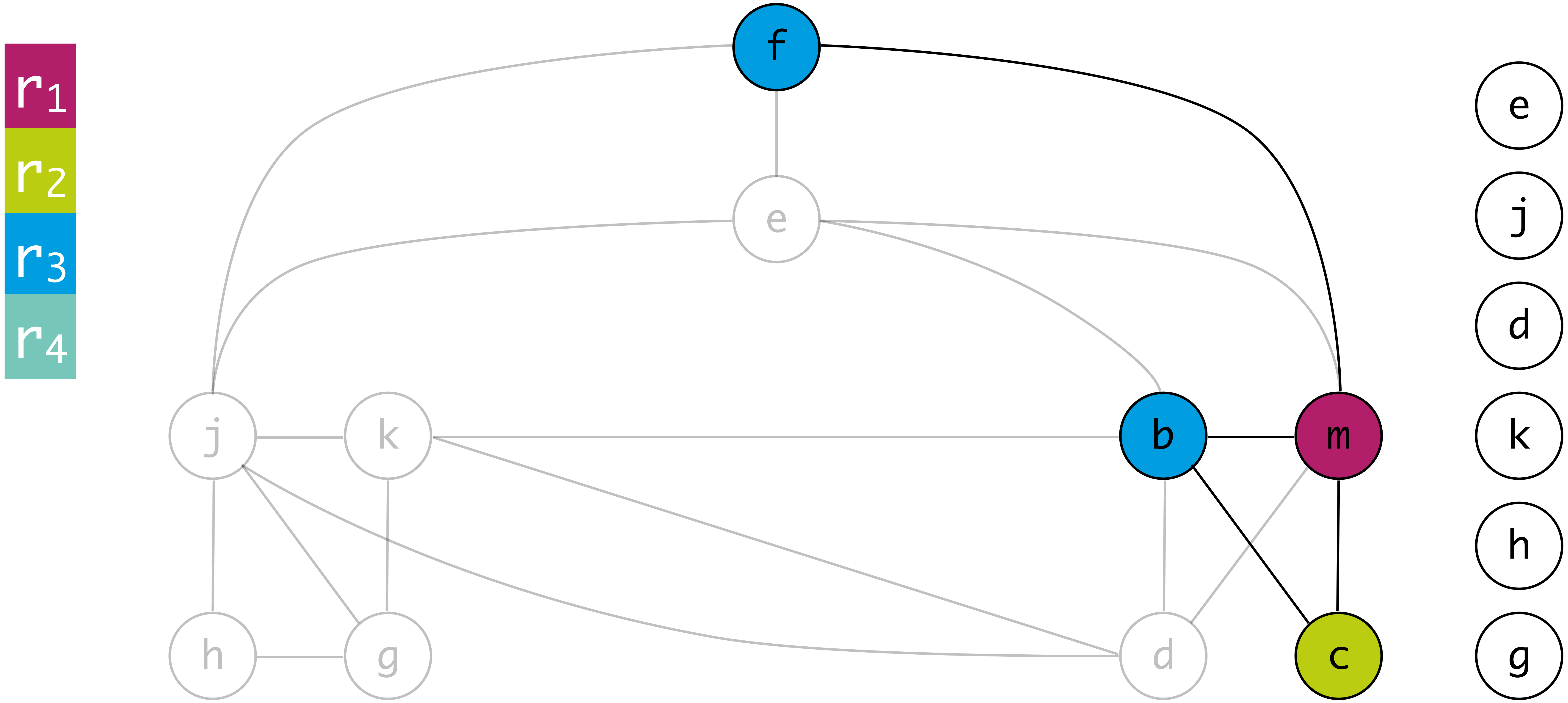

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
r1 := mem[j + 16]
b := mem[f]
r2 := e + 8
d := r2
k := r1 + 4
j := b
live out: d k j
```

Graph Coloring



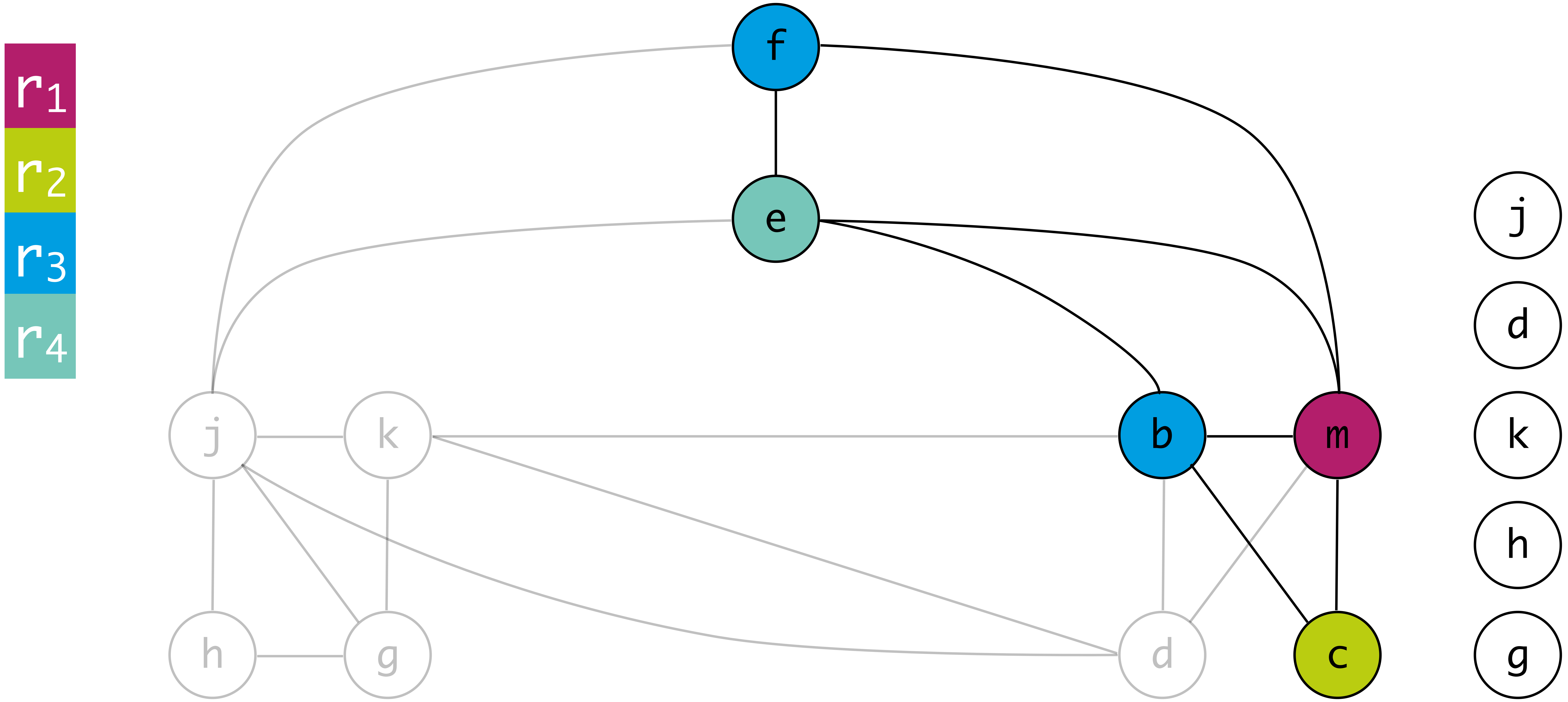

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
r1 := mem[j + 16]
r3 := mem[f]
r2 := e + 8
d := r2
k := r1 + 4
j := r3
live out: d k j
```

Graph Coloring



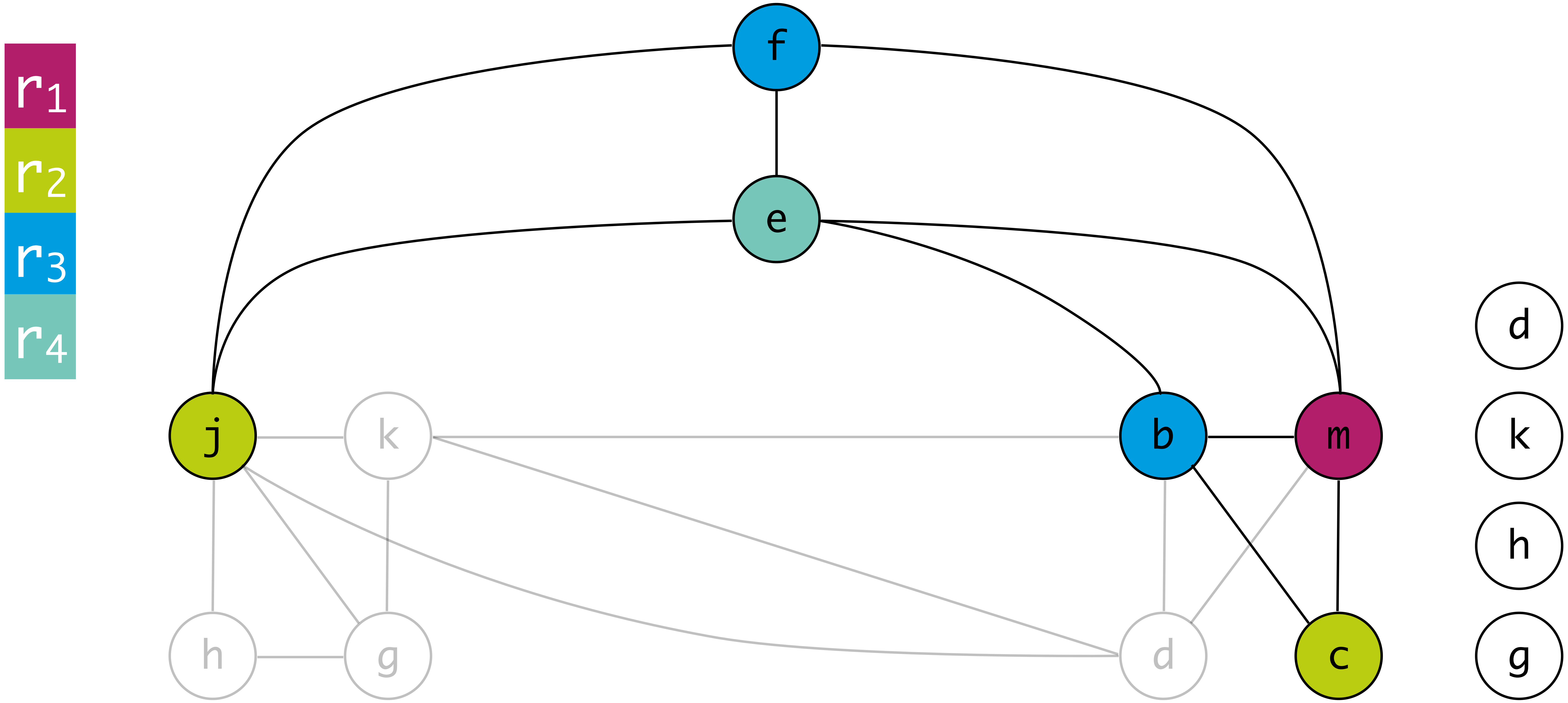

```
live-in: k j
g := mem[j + 12]
h := k - 1
r3 := g * h
e := mem[j + 8]
r1 := mem[j + 16]
r3 := mem[r3]
r2 := e + 8
d := r2
k := r1 + 4
j := r3
live out: d k j
```

Graph Coloring



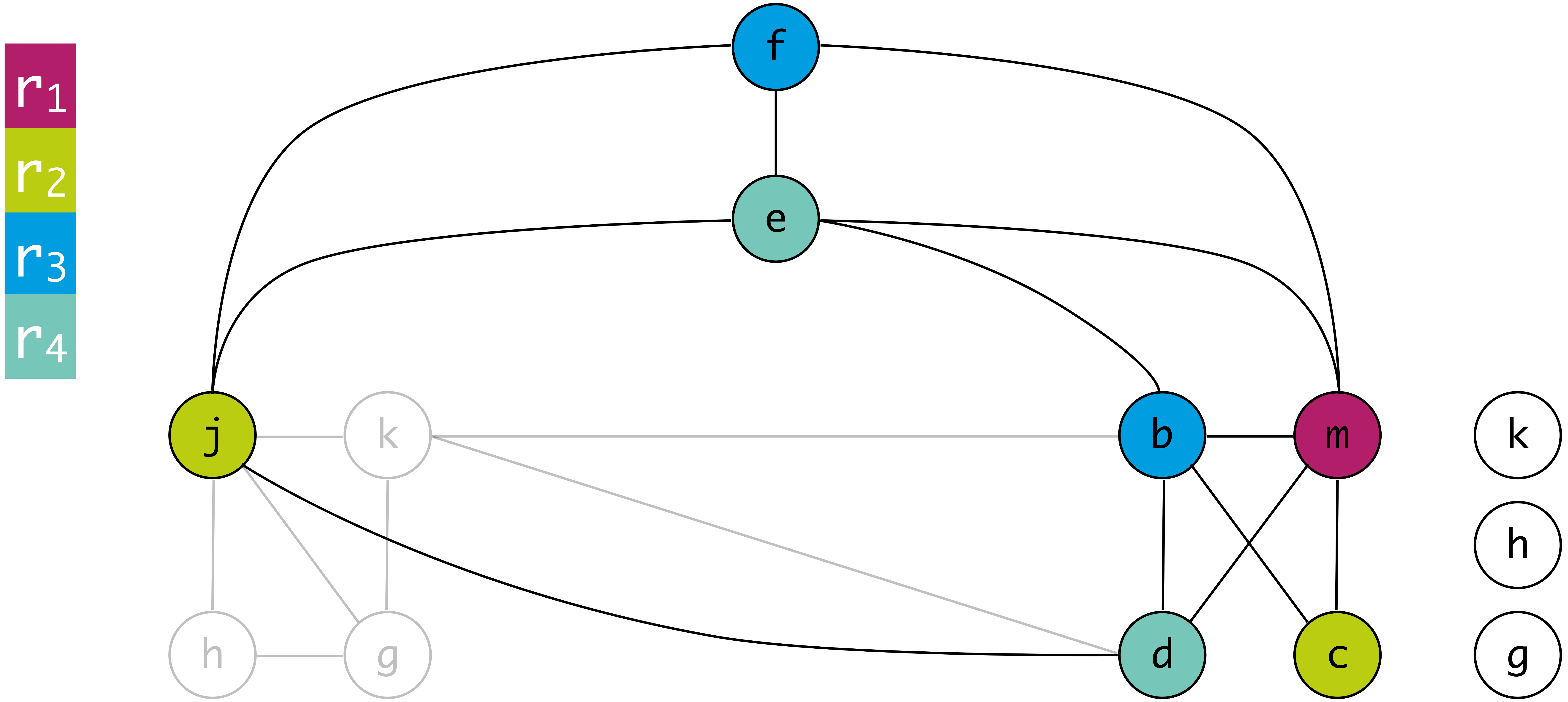

```
live-in: k j
g := mem[j + 12]
h := k - 1
r3 := g * h
r4 := mem[j + 8]
r1 := mem[j + 16]
r3 := mem[r3]
r2 := r4 + 8
d := r2
k := r1 + 4
j := r3
live out: d k j
```

Graph Coloring



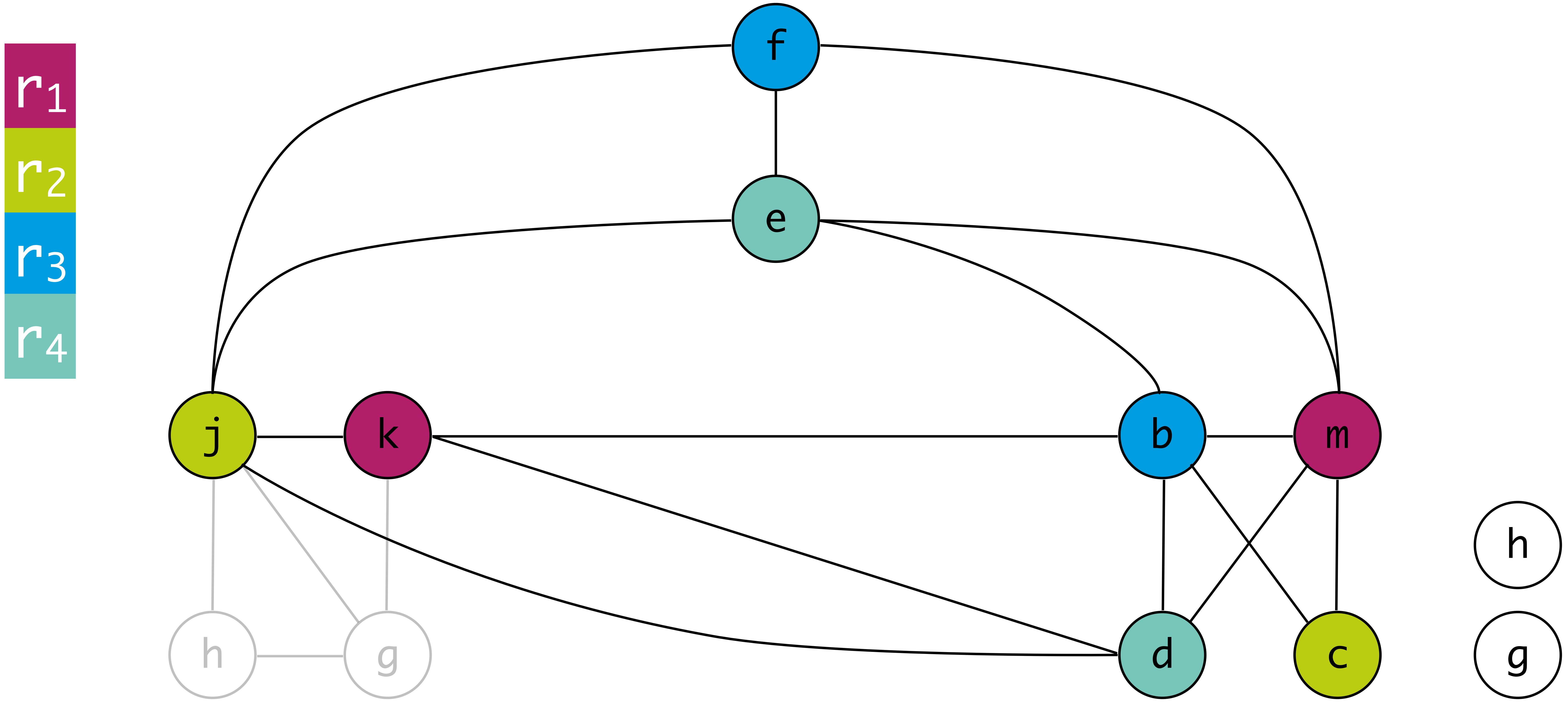

```
live-in: k r2
g := mem[r2 + 12]
h := k - 1
r3 := g * h
r4 := mem[r2 + 8]
r1 := mem[r2 + 16]
r3 := mem[r3]
r2 := r4 + 8
d := r2
k := r1 + 4
r2 := r3
live out: d k r2
```

Graph Coloring



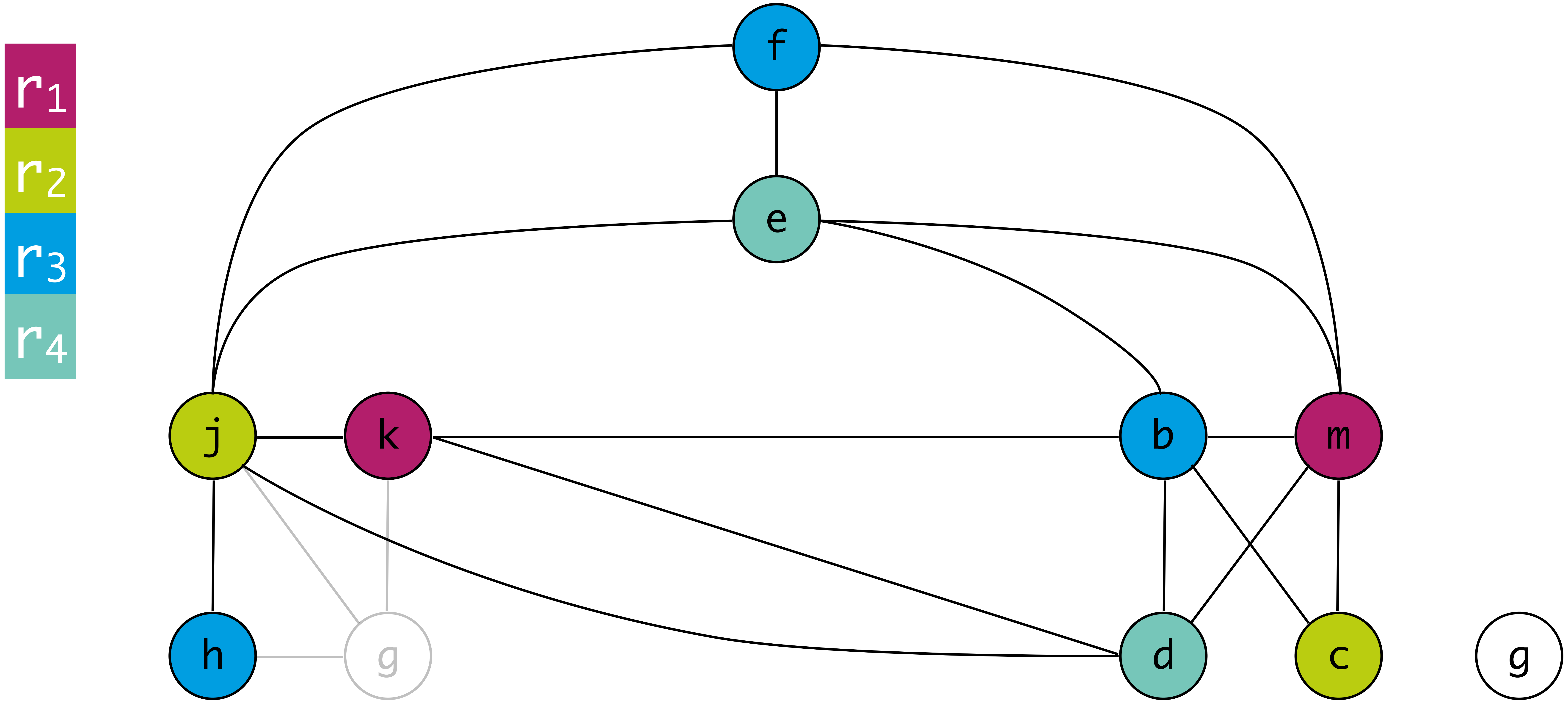

```
live-in: k r2
g  := mem[r2 + 12]
h  := k - 1
r3  := g * h
r4  := mem[r2 + 8]
r1 := mem[r2 + 16]
r3  := mem[r3]
r2  := r4 + 8
r4  := r2
k  := r1 + 4
r2  := r3
live out: r4 k r2
```

Graph Coloring



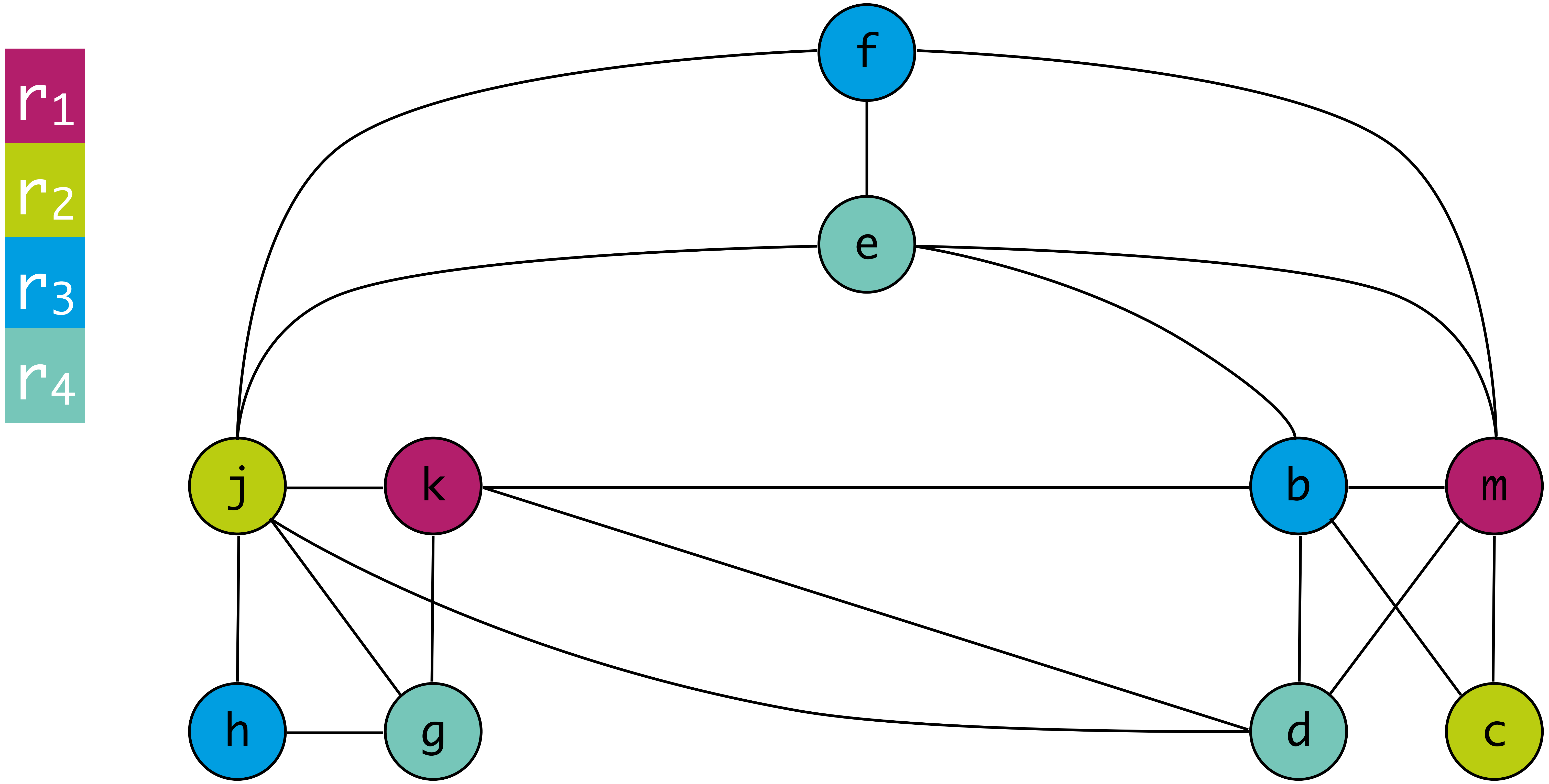

```
live-in: r1 r2
g := mem[r2 + 12]
h := r1 - 1
r3 := g * h
r4 := mem[r2 + 8]
r1 := mem[r2 + 16]
r3 := mem[r3]
r2 := r4 + 8
r4 := r2
r1 := r1 + 4
r2 := r3
live out: r4 r1 r2
```

Graph Coloring




```
live-in: r1 r2
g := mem[r2 + 12]
r3 := r1 - 1
r3 := g * r3
r4 := mem[r2 + 8]
r1 := mem[r2 + 16]
r3 := mem[r3]
r2 := r4 + 8
r4 := r2
r1 := r1 + 4
r2 := r3
live out: r4 r1 r2
```

Graph Coloring




```
live-in: r1 r2
r4 := mem[r2 + 12]
r3 := r1 - 1
r3 := r4 * r3
r4 := mem[r2 + 8]
r1 := mem[r2 + 16]
r3 := mem[r3]
r2 := r4 + 8
r4 := r2
r1 := r1 + 4
r2 := r3
live out: r4 r1 r2
```

Spilling

Optimistic Coloring

Simplify

– remove node of insignificant degree (fewer than k edges)

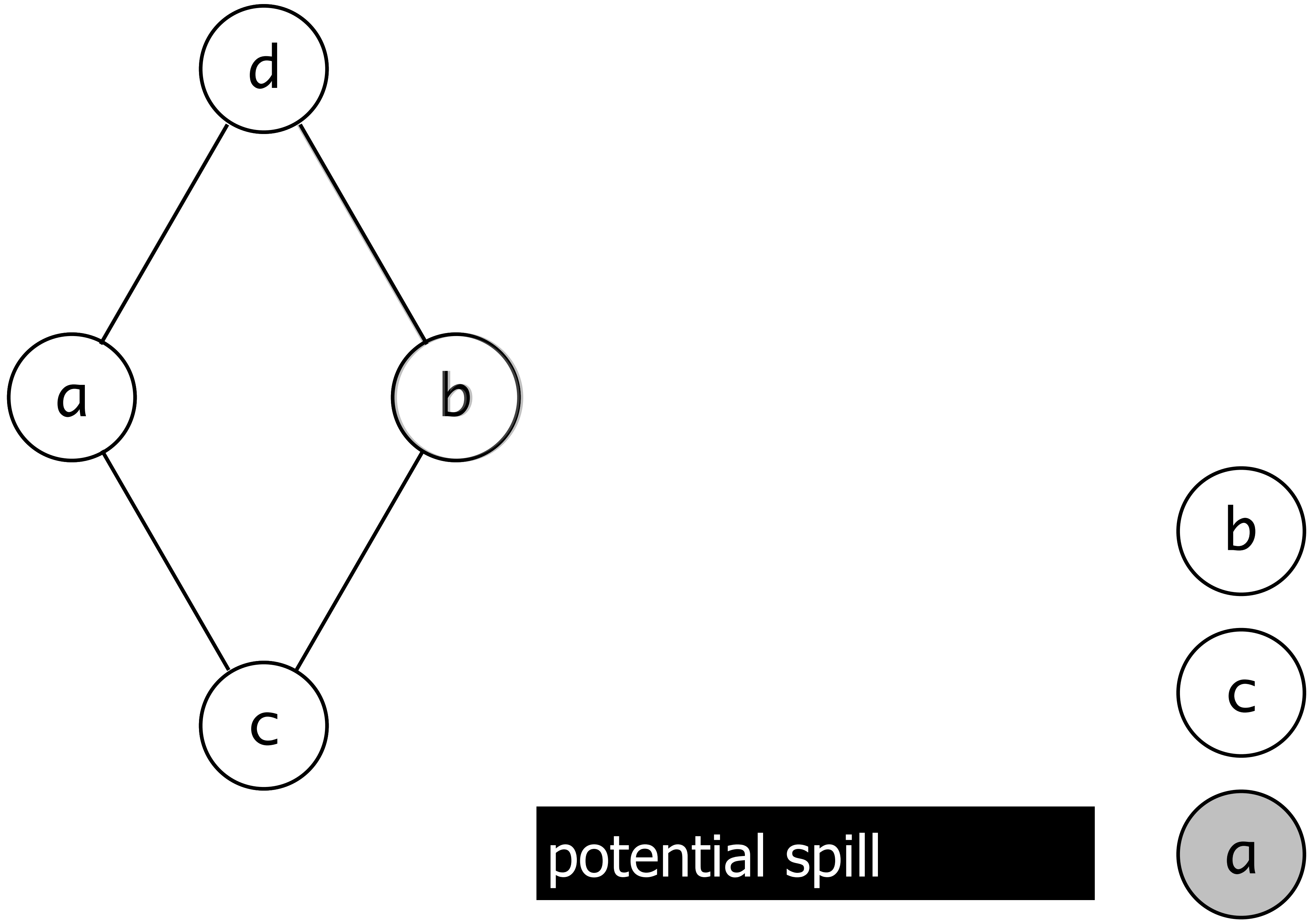
Spill

– remove node of significant degree (k or more edges)

Select

– add node, select color

Optimistic Coloring: Example with 2 Colors



Spilling

Simplify

– remove node of insignificant degree (less than k edges)

Spill

– remove node of significant degree (k or more edges)

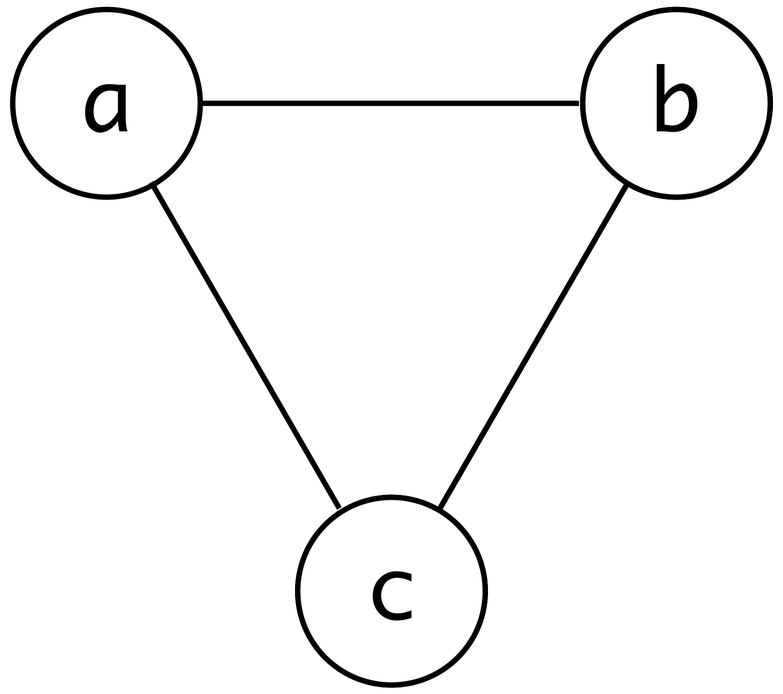
Select

– add node, select color

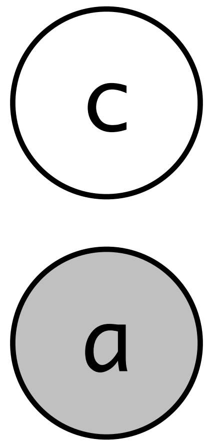
Actual spill

Start over

Spilling: example with 2 colors



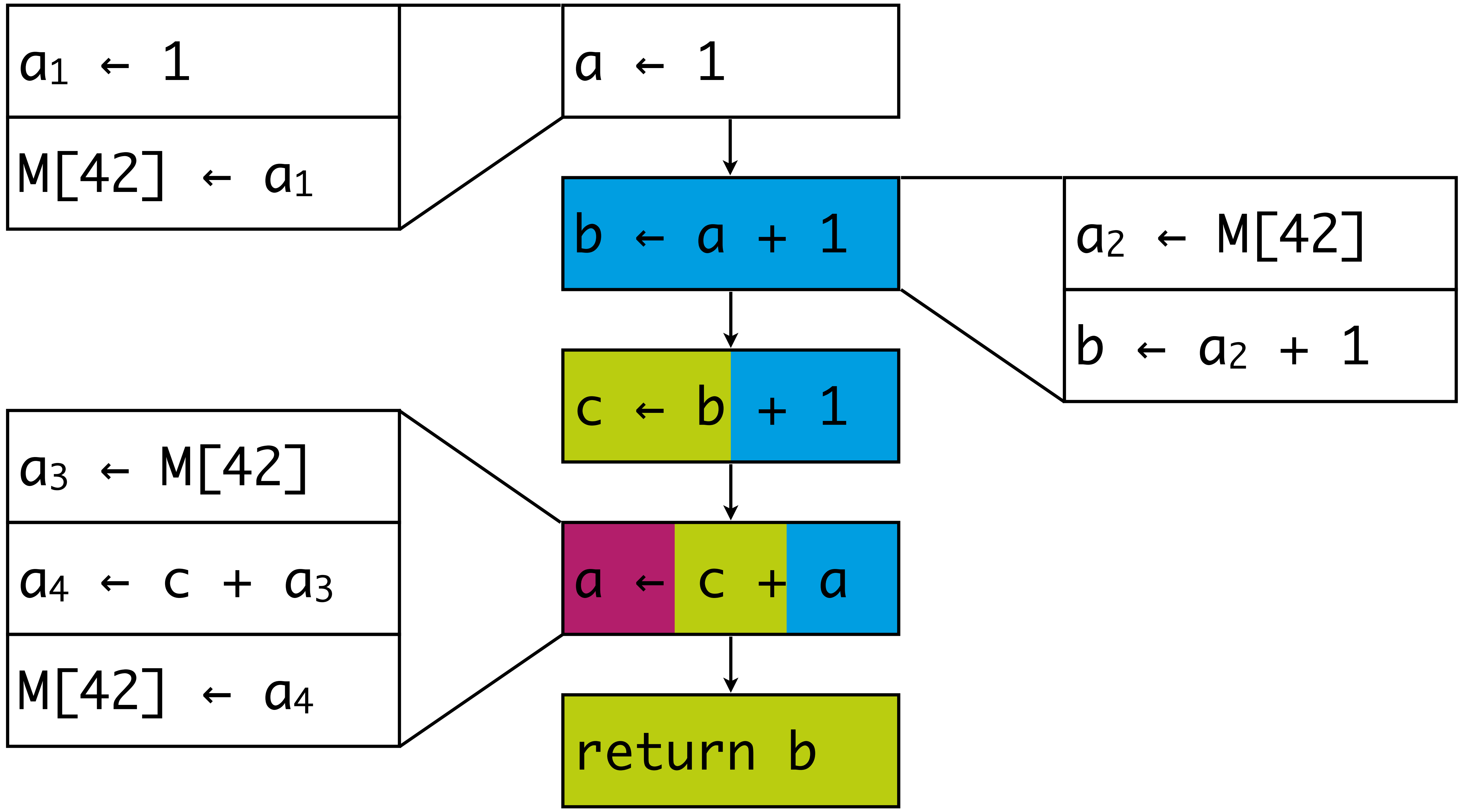
actual spill



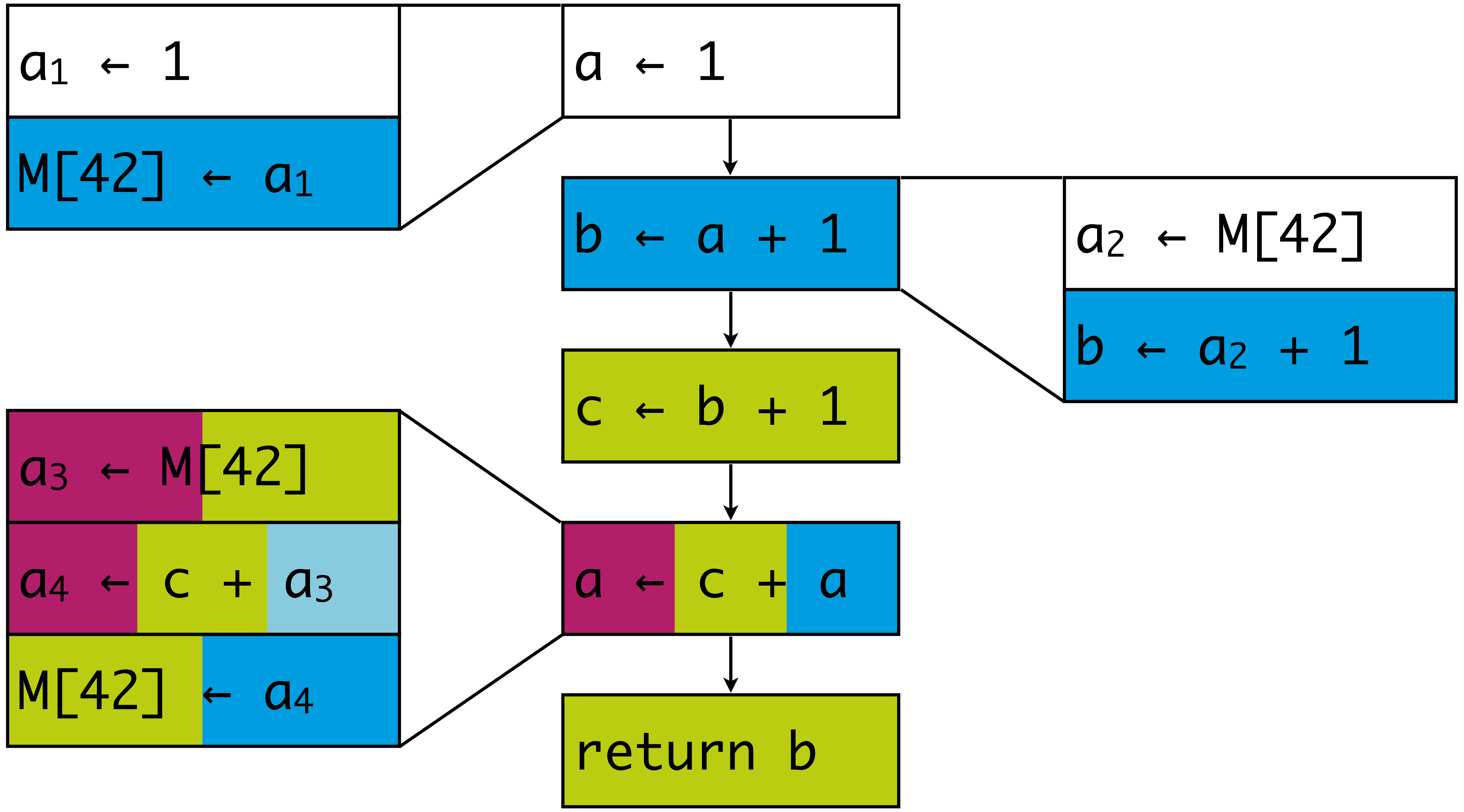
Spilling: Example

```
a ← 1
b ← a + 1
c ← b + 1
a ← c + a
return b
```

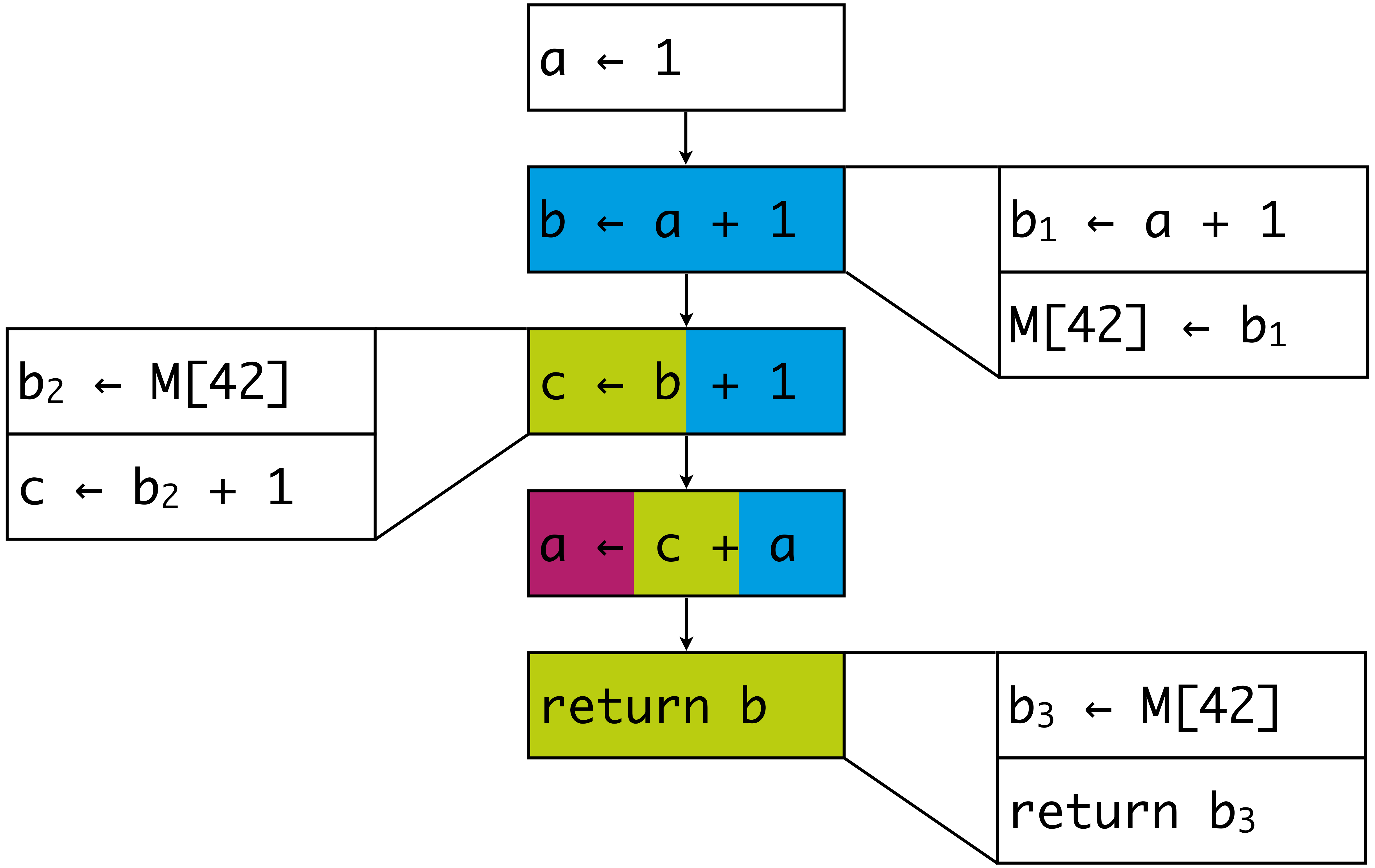
Spilling: Example



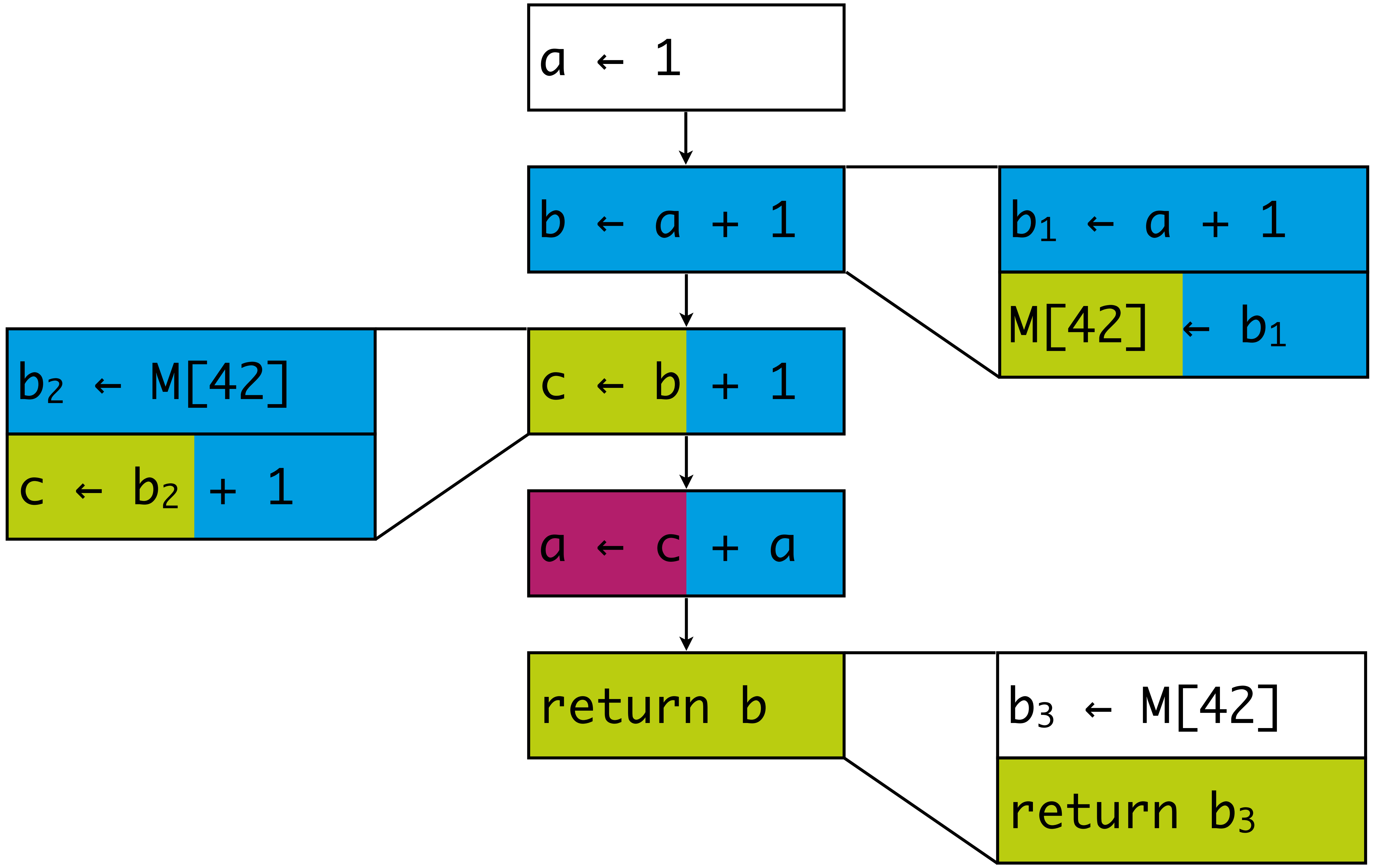
Spilling: Example



Spilling: Example



Spilling: Example



Coalescing

Eliminating Move Instructions

```
live-in: k j
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 k j
```

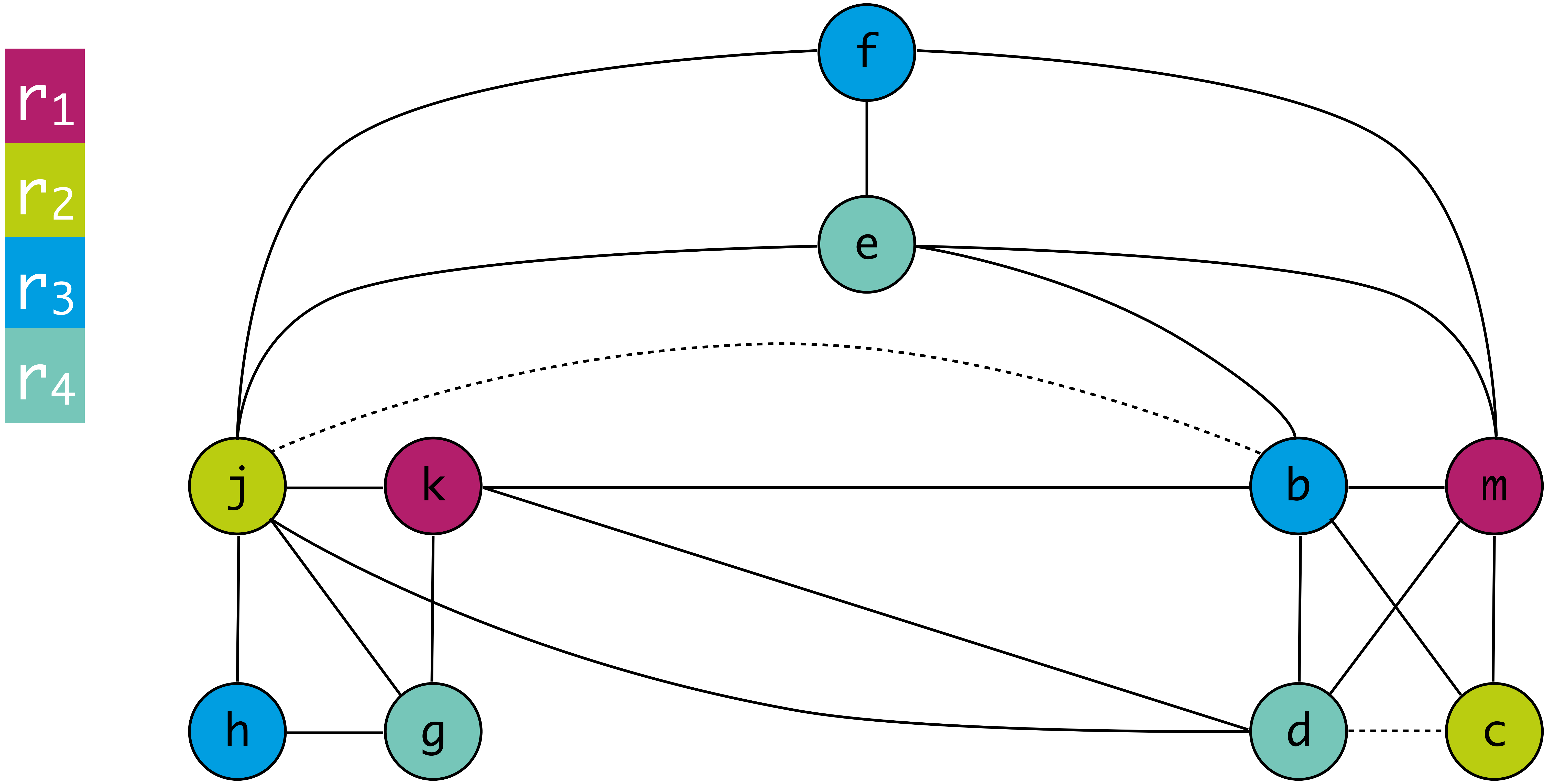
Eliminating Move Instructions

```
live-in: k j
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 k j
```

Coalescing

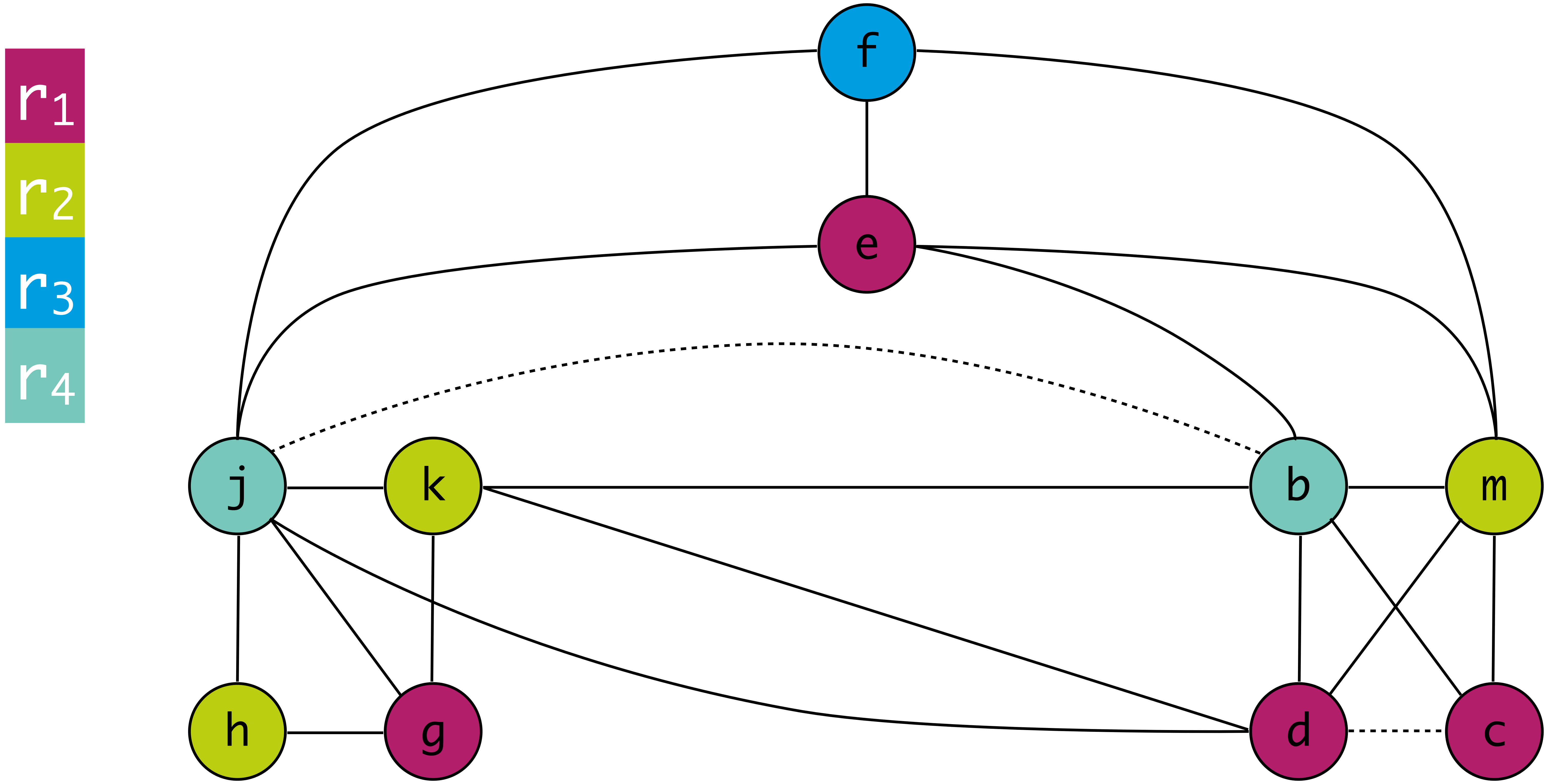
coalesce |ˌkəʊəˈlɛs|
verb *[no object]*
come together to form one mass or whole: *the puddles had* ***coalesced into*** *shallow streams*.
• *[with object]* combine (elements) in a mass or whole: *his idea served to* ***coalesce*** *all that happened* ***into*** *one connected whole*.

Recap: Graph Coloring



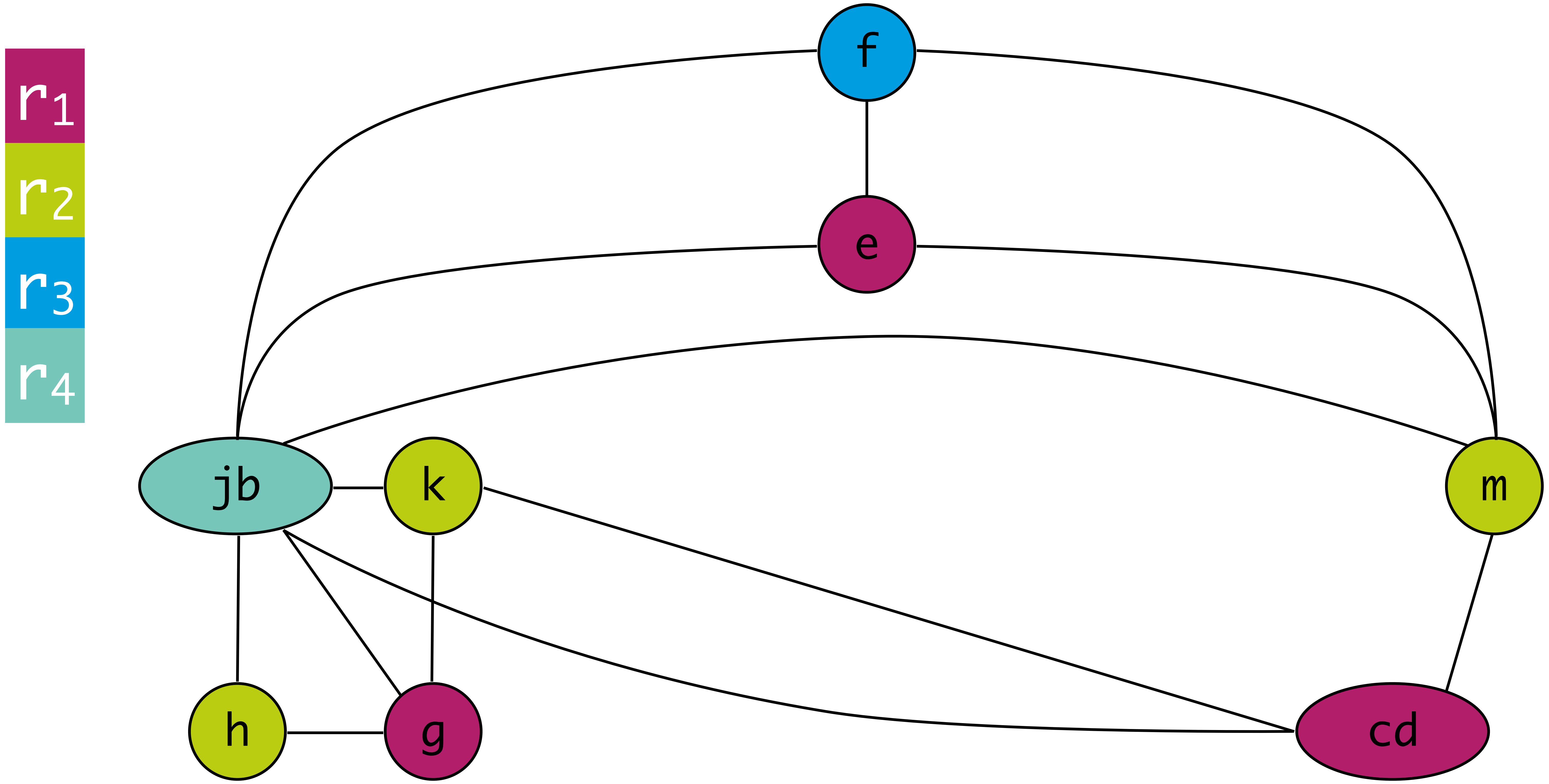

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing: better solution




```
live-in: k j
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 k j
```

Coalescing: coalescing nodes




```
live-in: k j
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 k j
```

Coalescing: conservative strategies

Briggs

- a/b has fewer than k neighbours of significant degree
- nodes of insignificant degree and a/b can be simplified
- remaining graph is colorable

George

- all neighbours of a of significant degree interfere also with b
- neighbours of a of insignificant degree can be simplified
- subgraph of original graph is colorable

Graph Coloring: Steps

Simplify

- remove non-move-related node of insignificant degree

Coalesce

Freeze

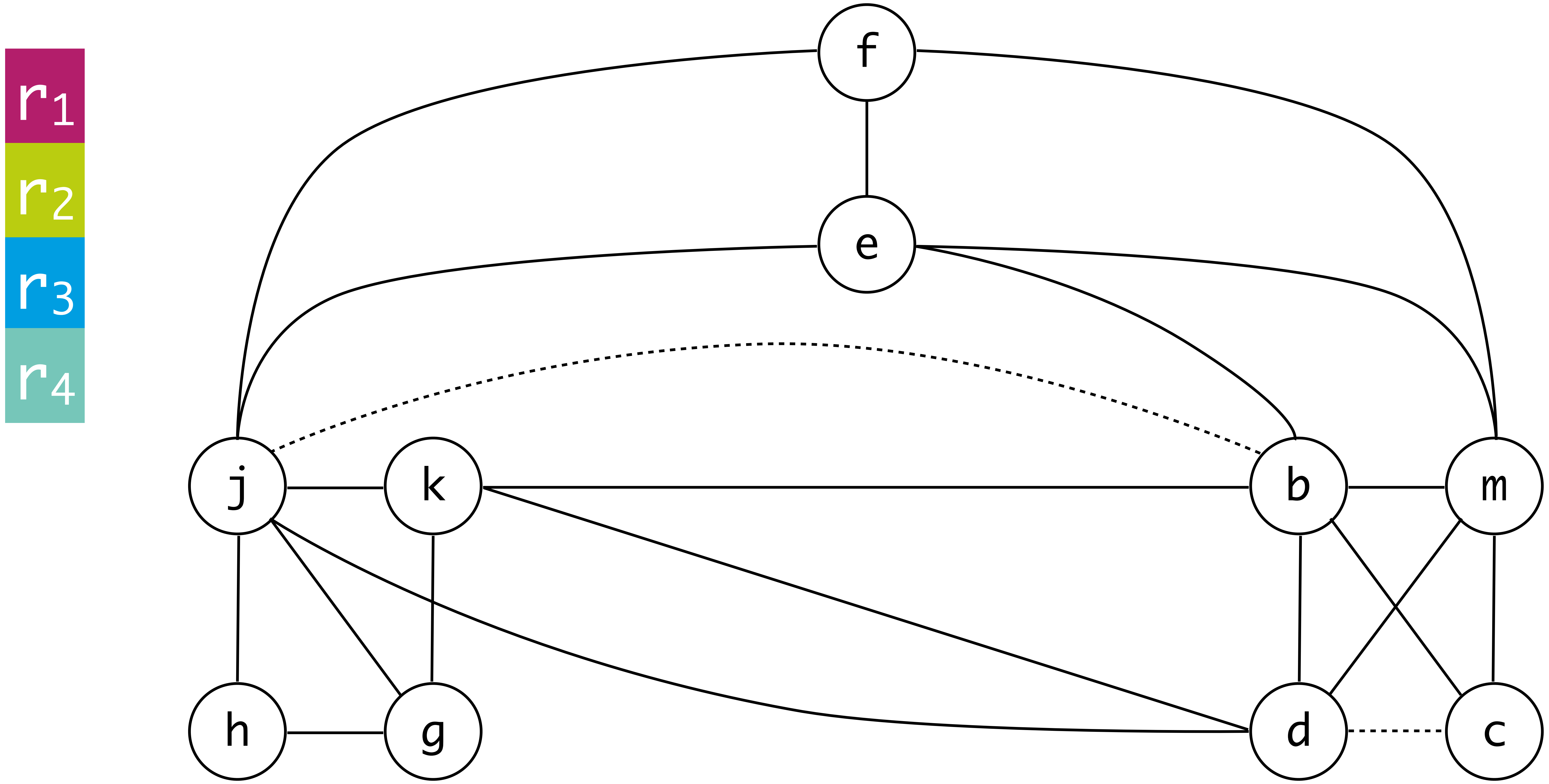
- turn move-related node of insignificant degree into non-move-related

Spill

Select

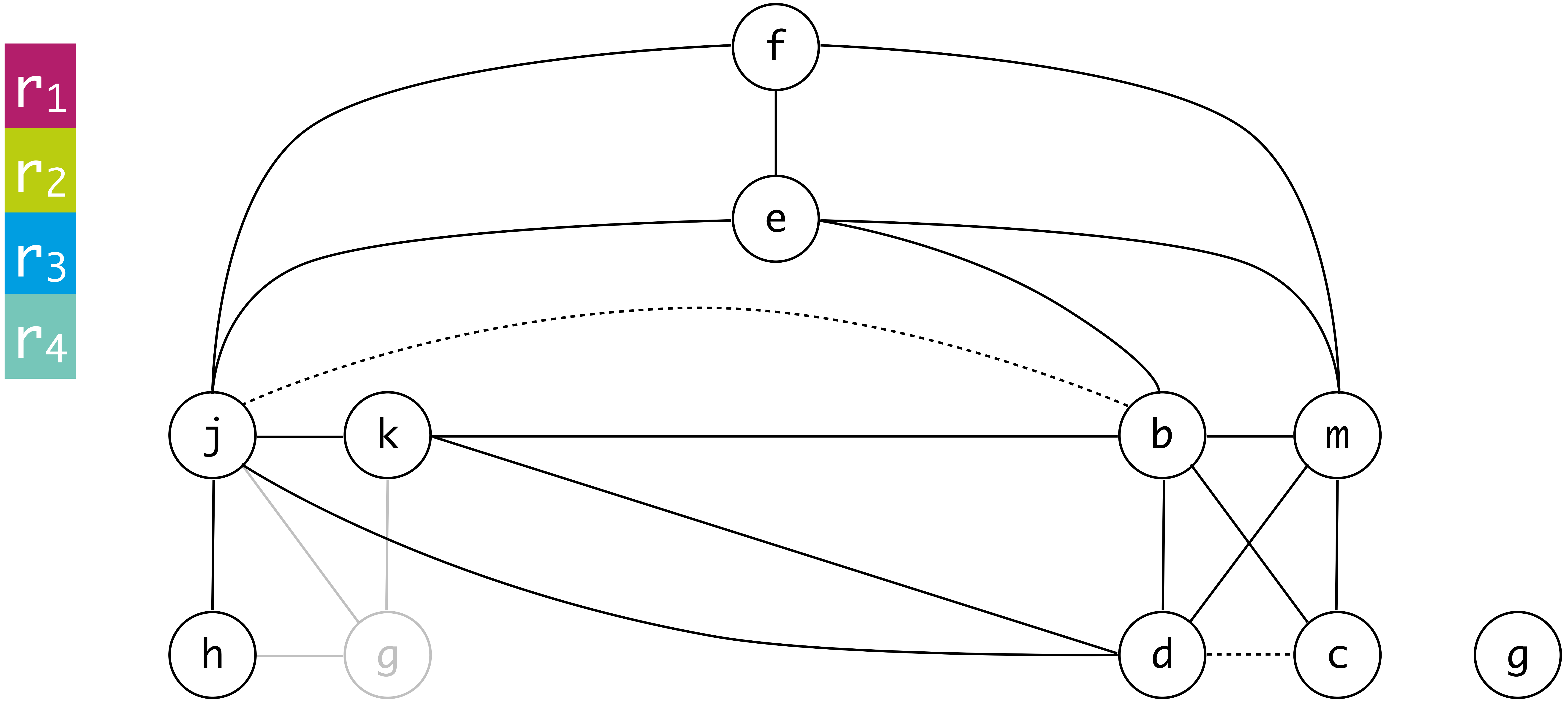
Start over

Coalescing: example



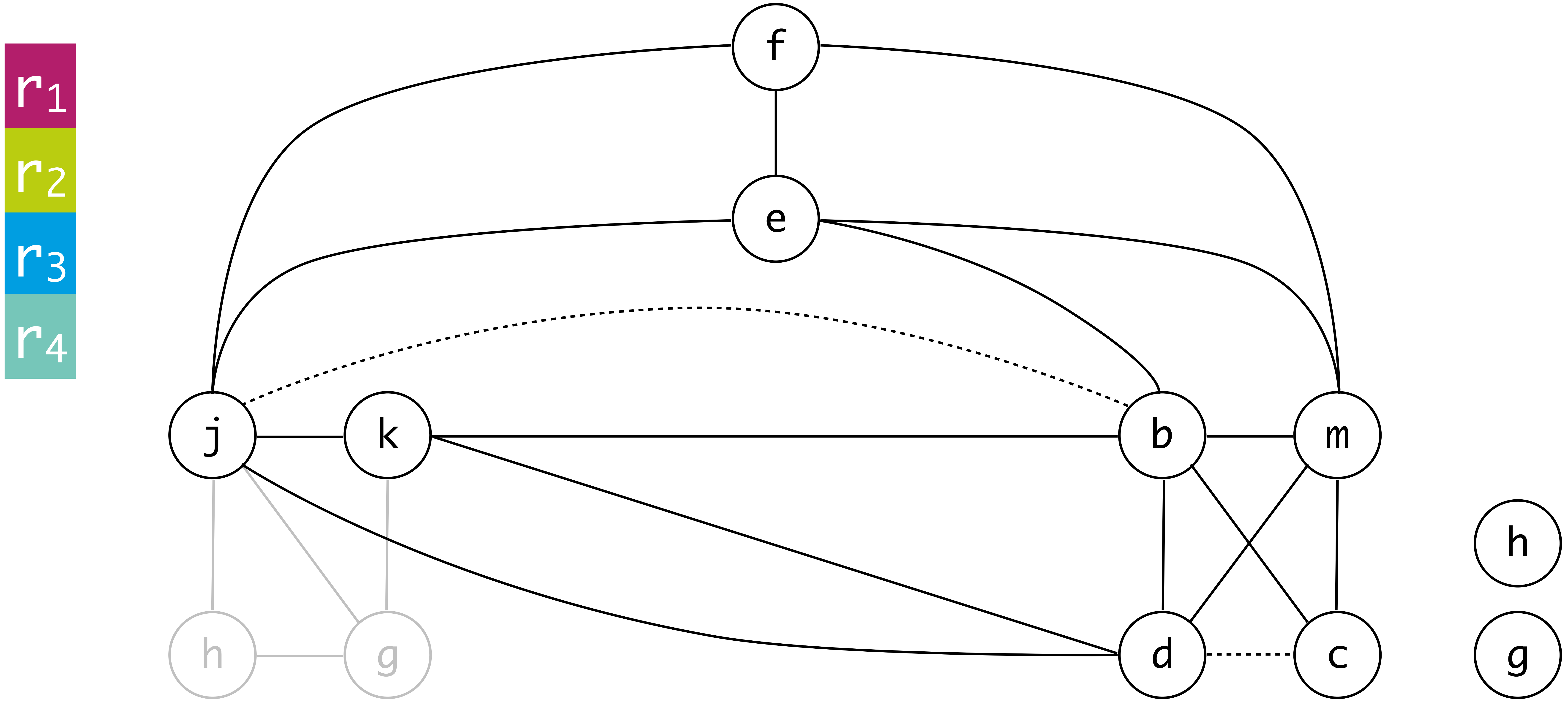

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



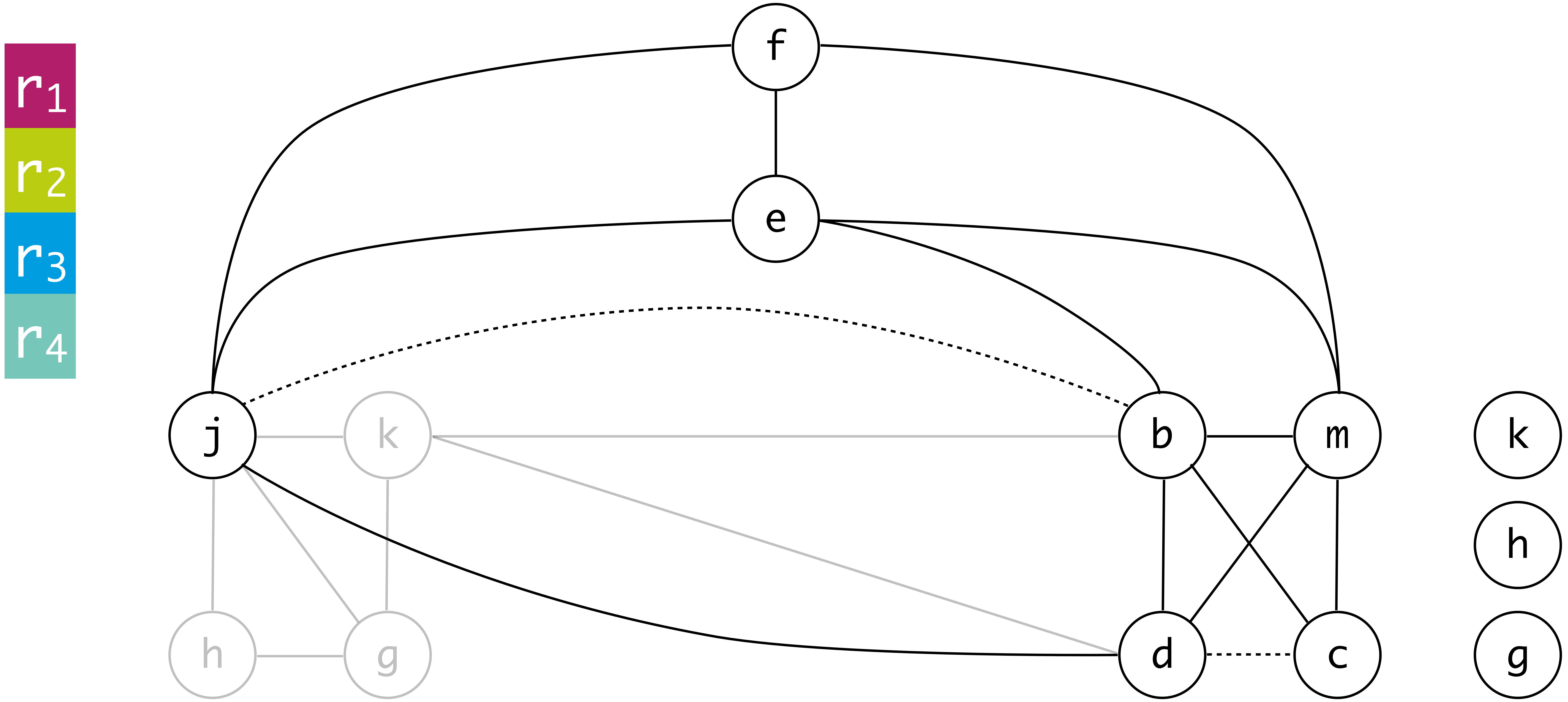

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



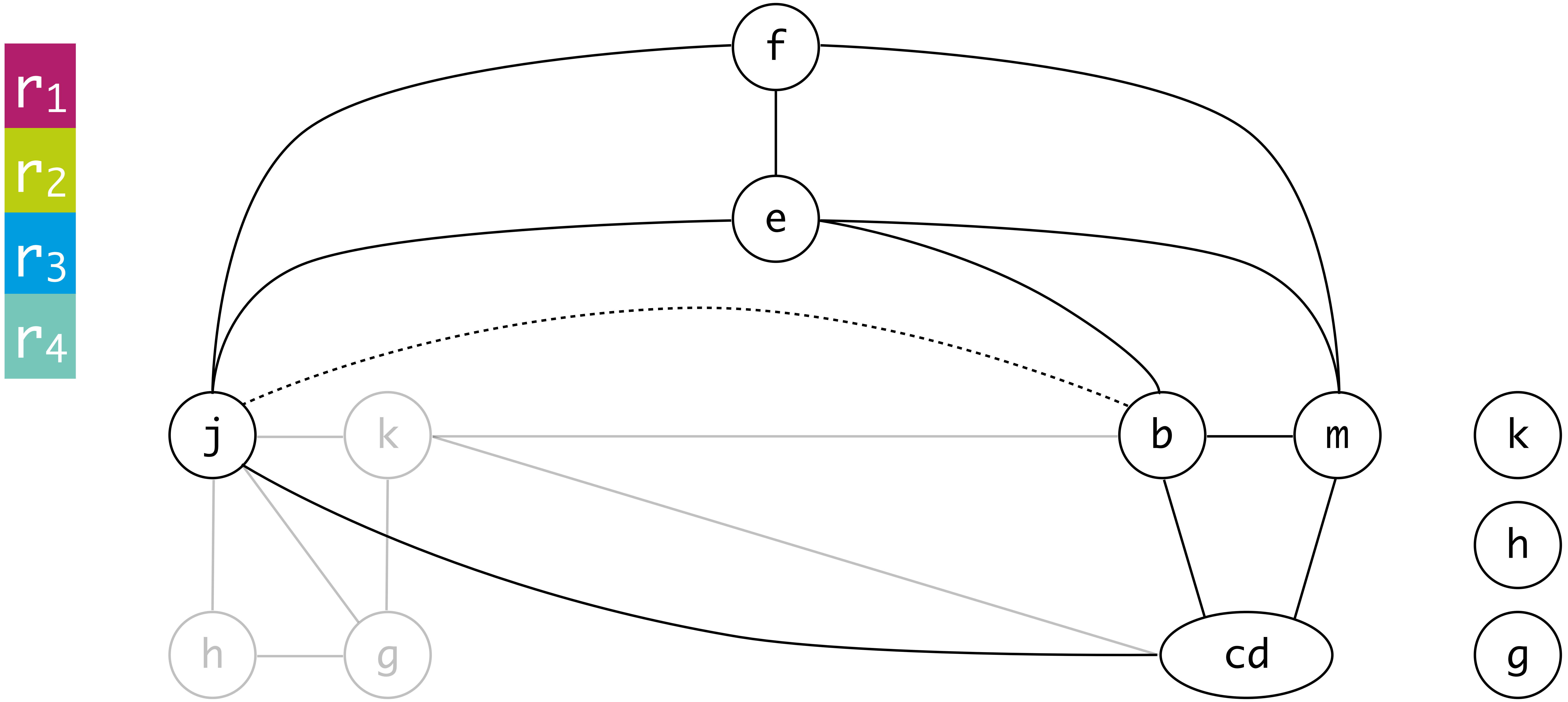

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



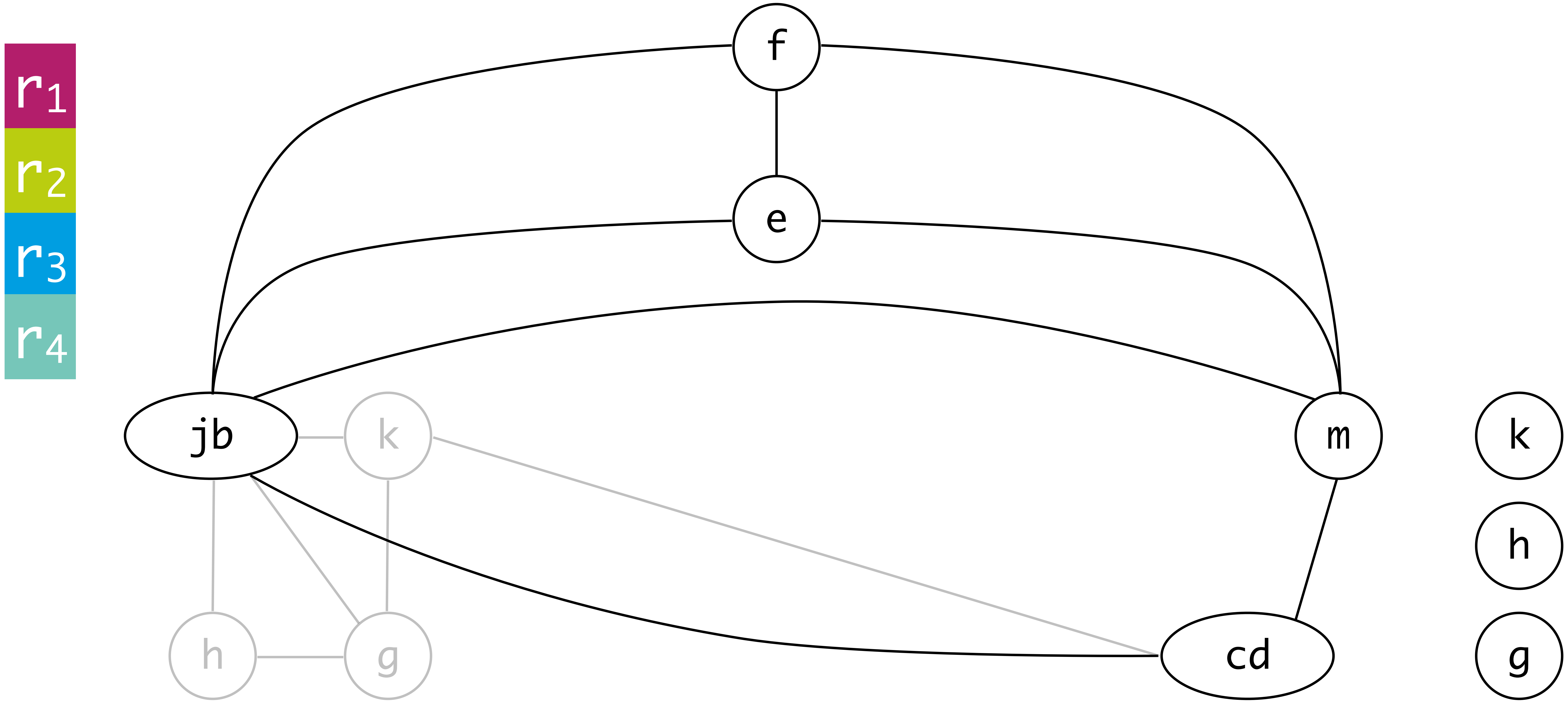

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



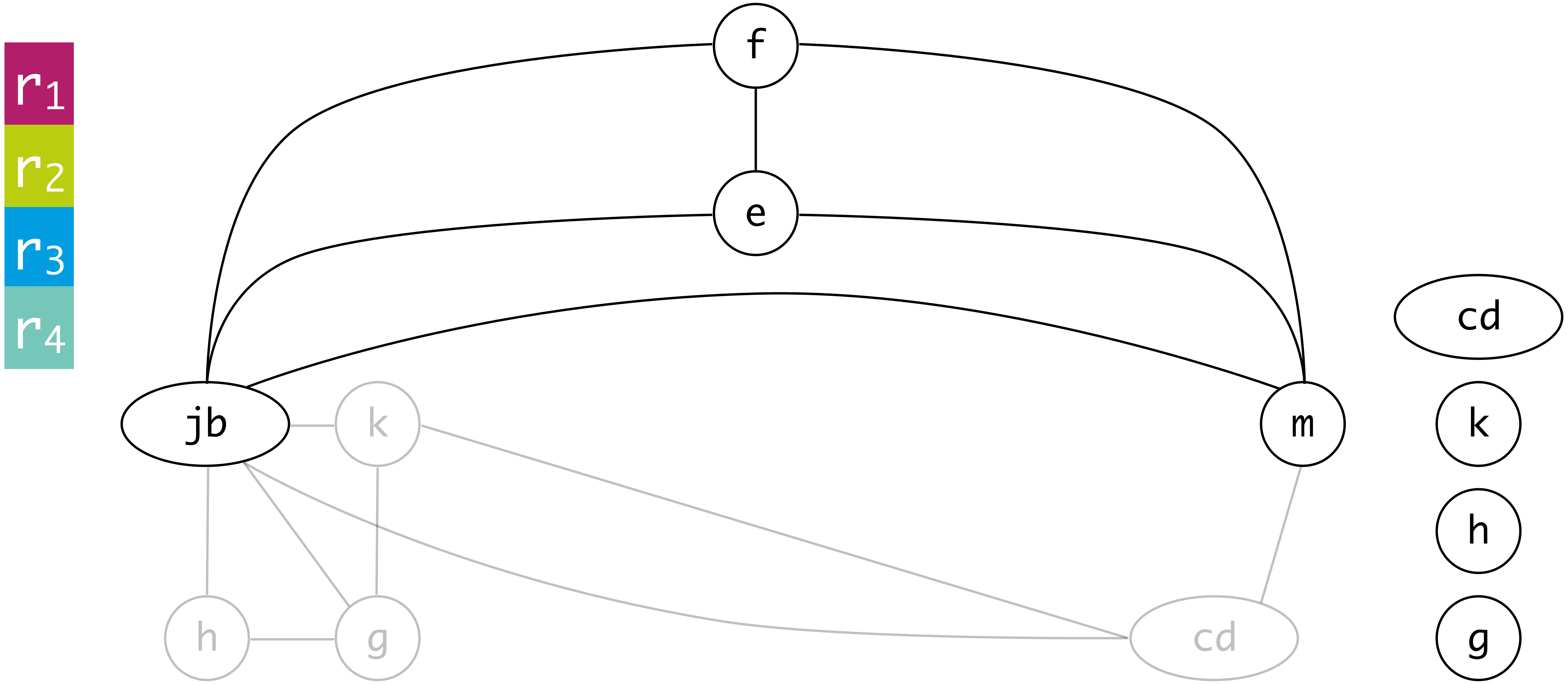

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



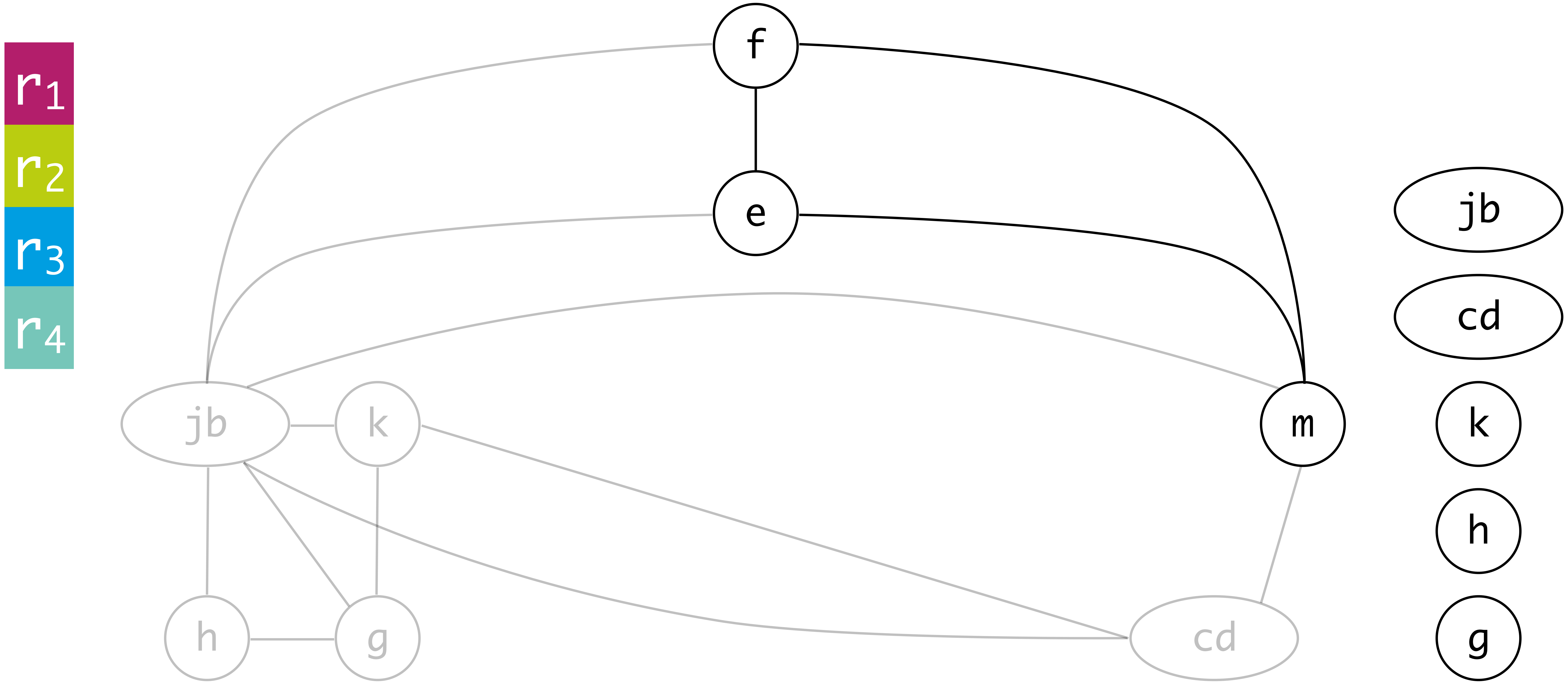

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



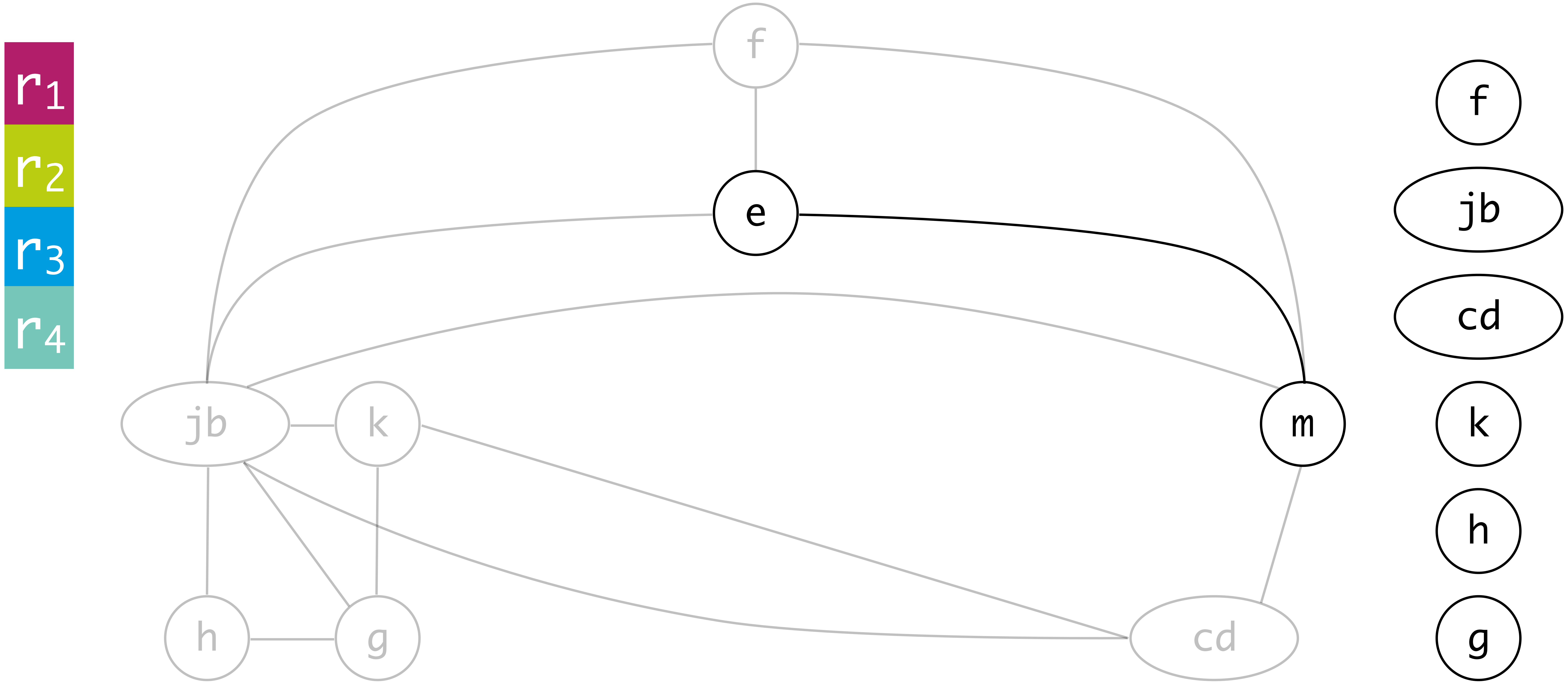

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



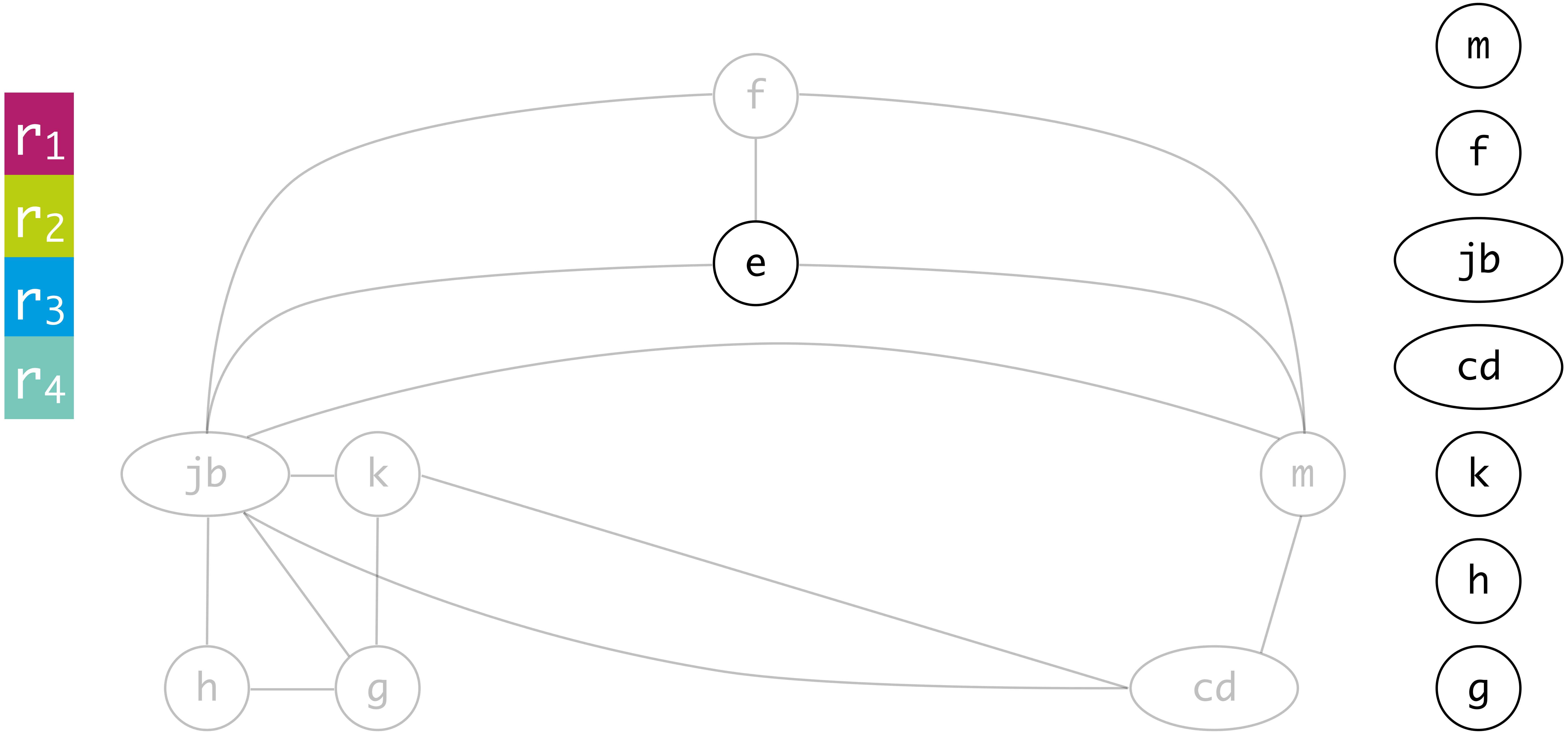

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



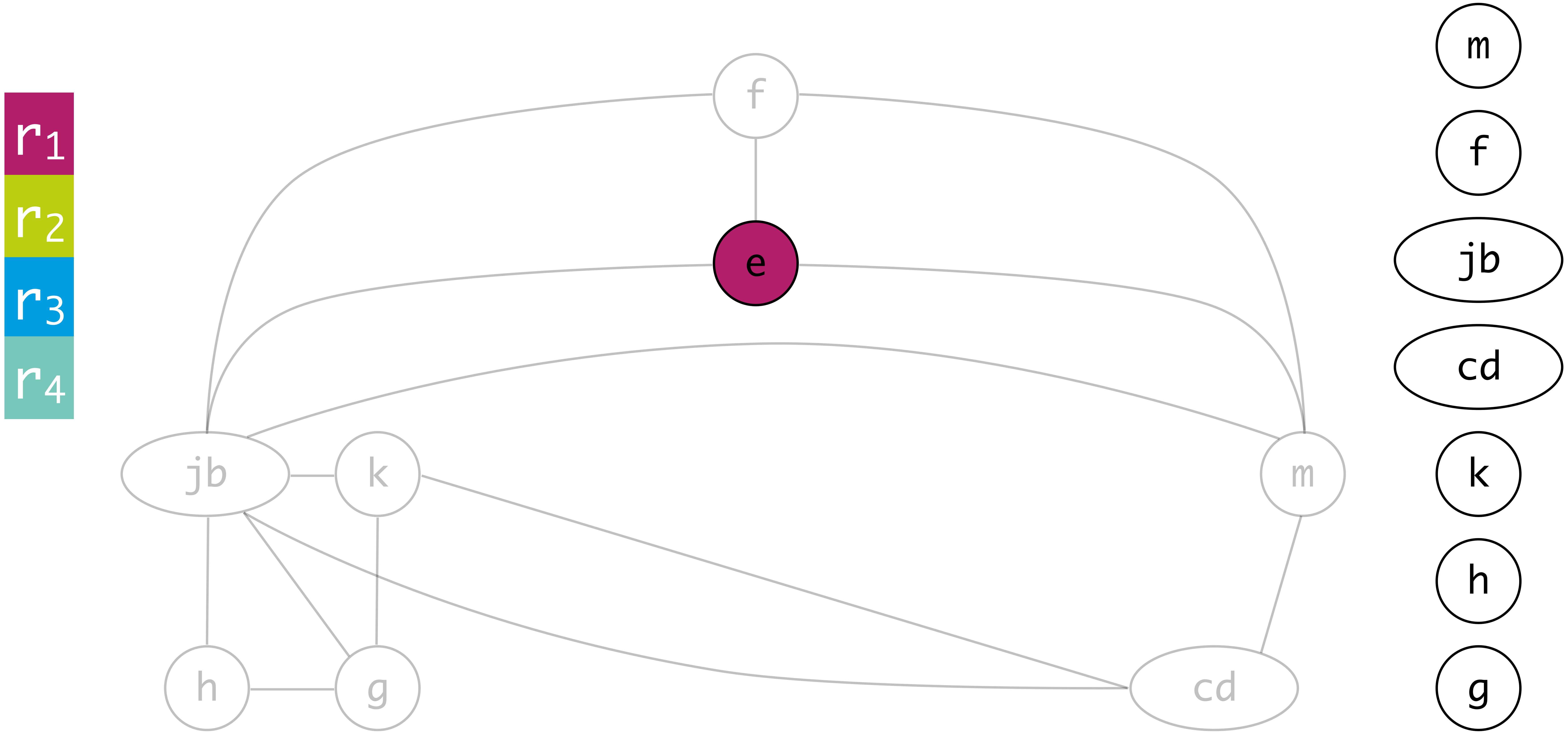

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



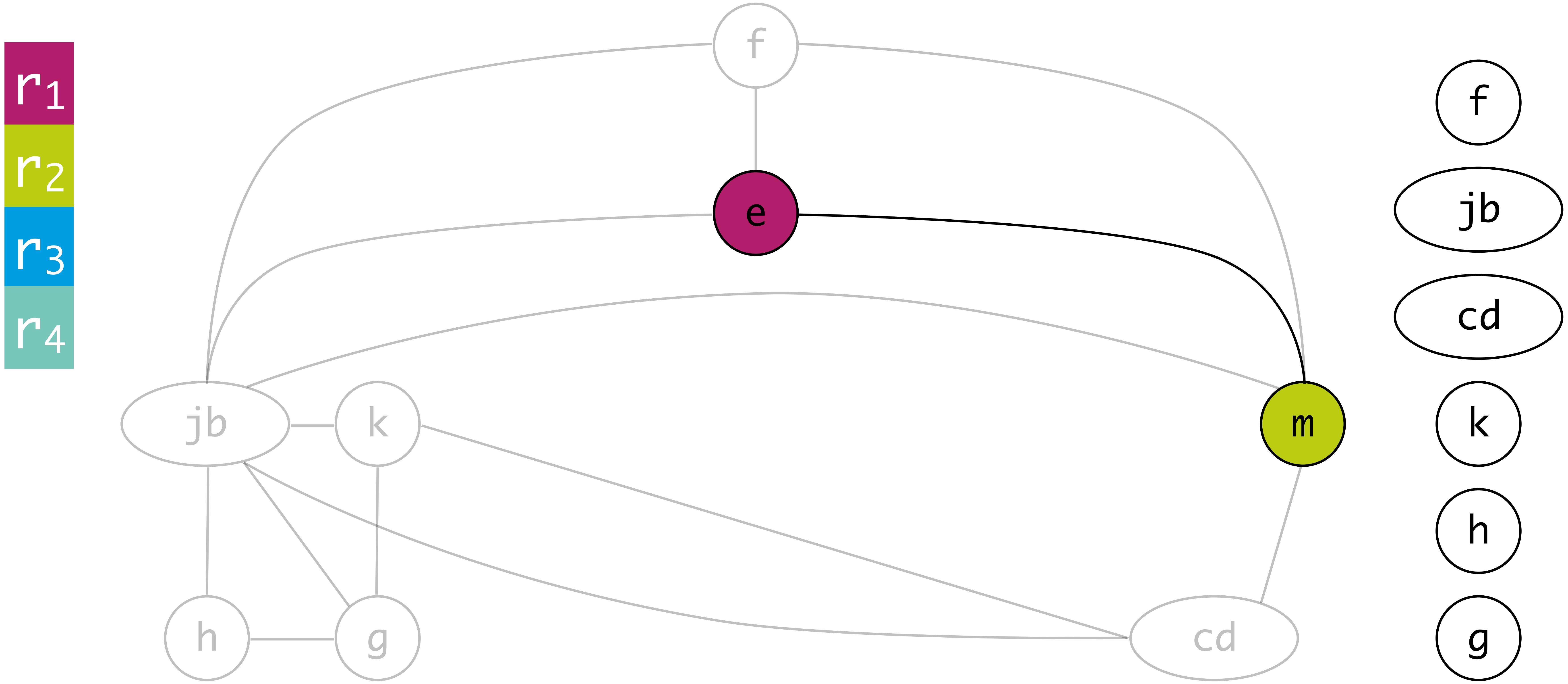

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



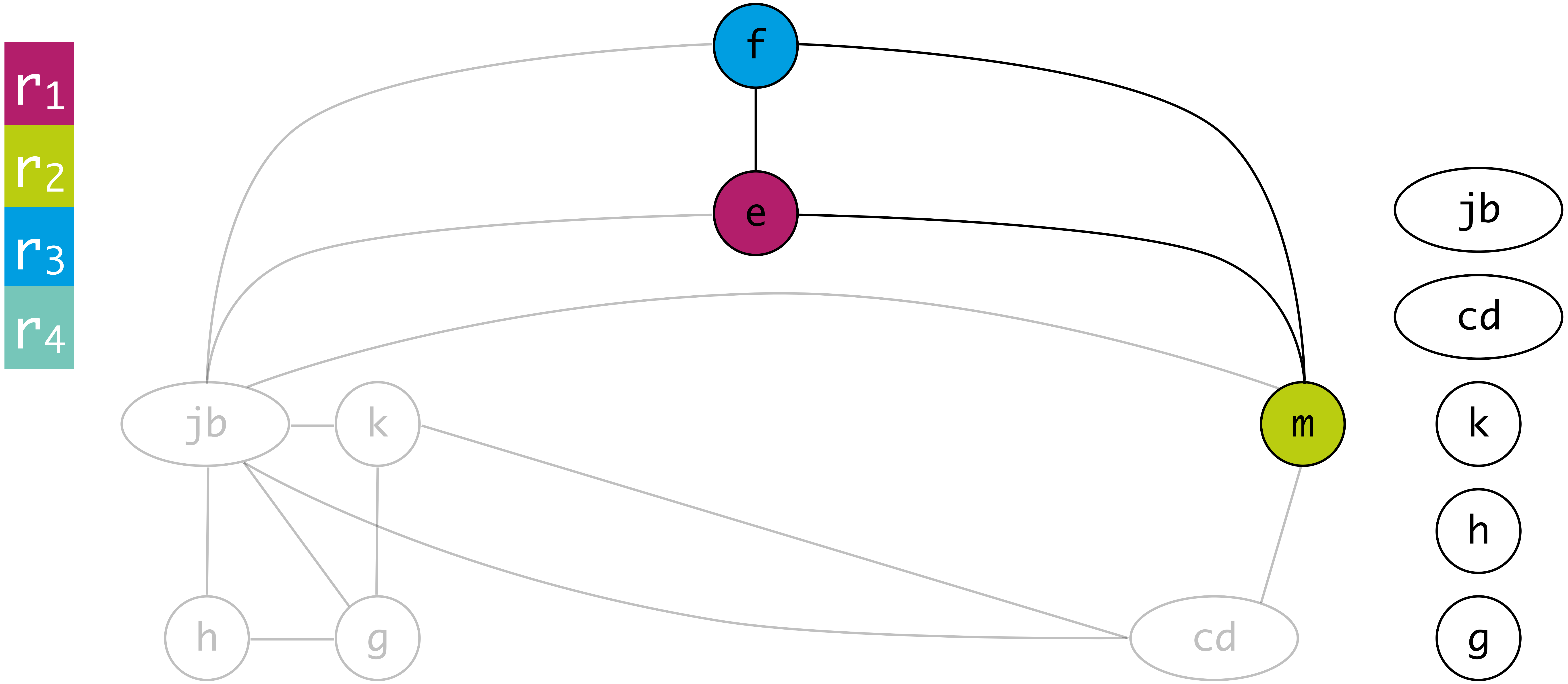

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
r₁ := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := r₁ + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



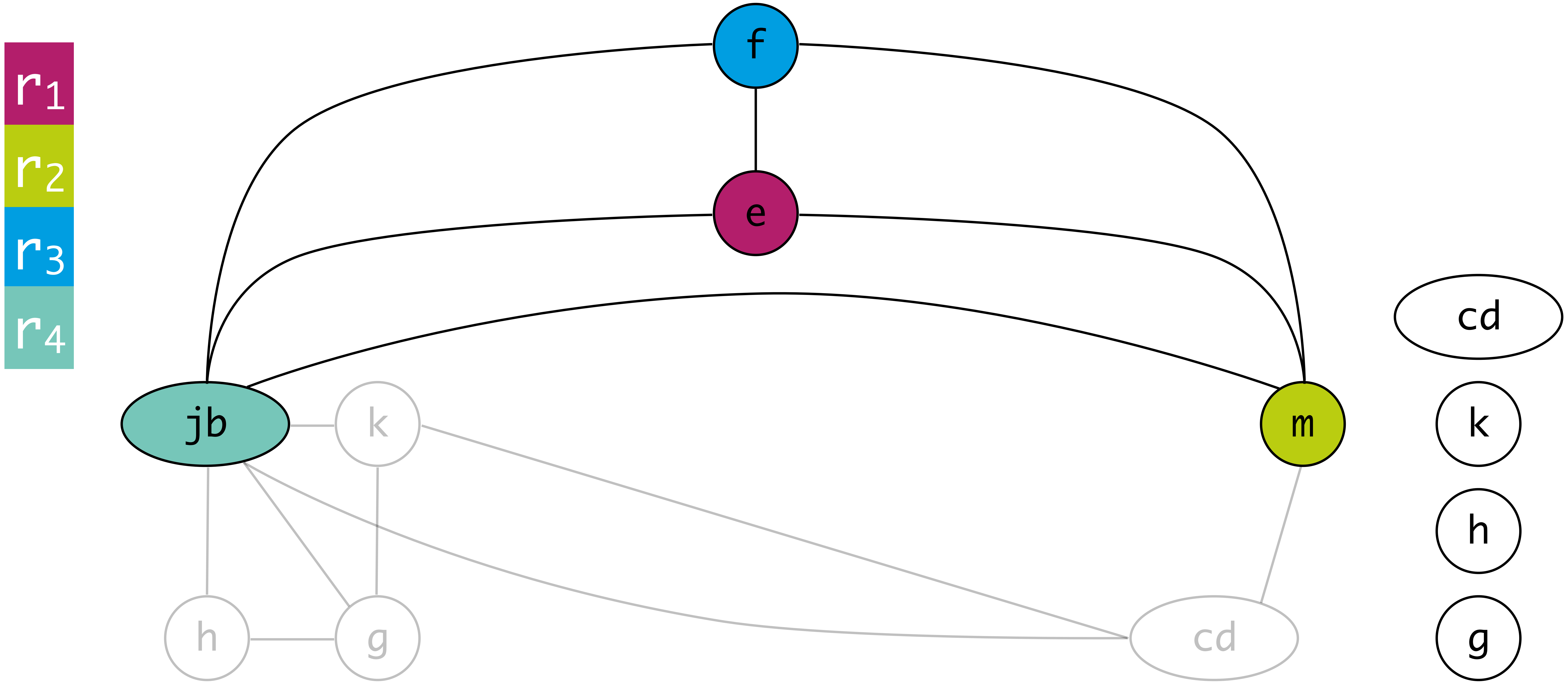

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
r₁ := mem[j + 8]
r₂ := mem[j + 16]
b := mem[f]
c := r₁ + 8
d := c
k := r₂ + 4
j := b
live out: d k j
```

Coalescing



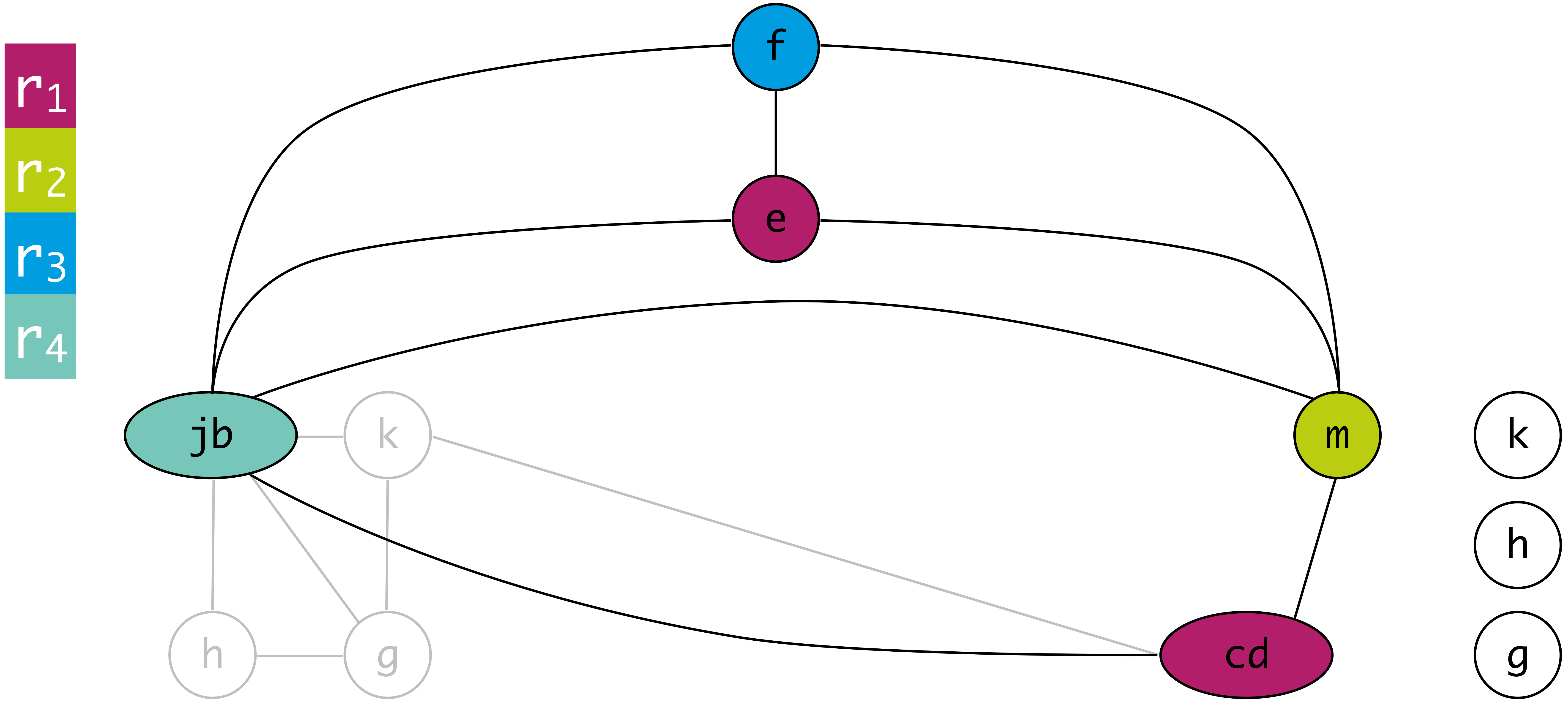

```
live-in: k j
g := mem[j + 12]
h := k - 1
r3 := g * h
r1 := mem[j + 8]
r2 := mem[j + 16]
b := mem[r3]
c := r1 + 8
d := c
k := r2 + 4
j := b
live out: d k j
```

Coalescing



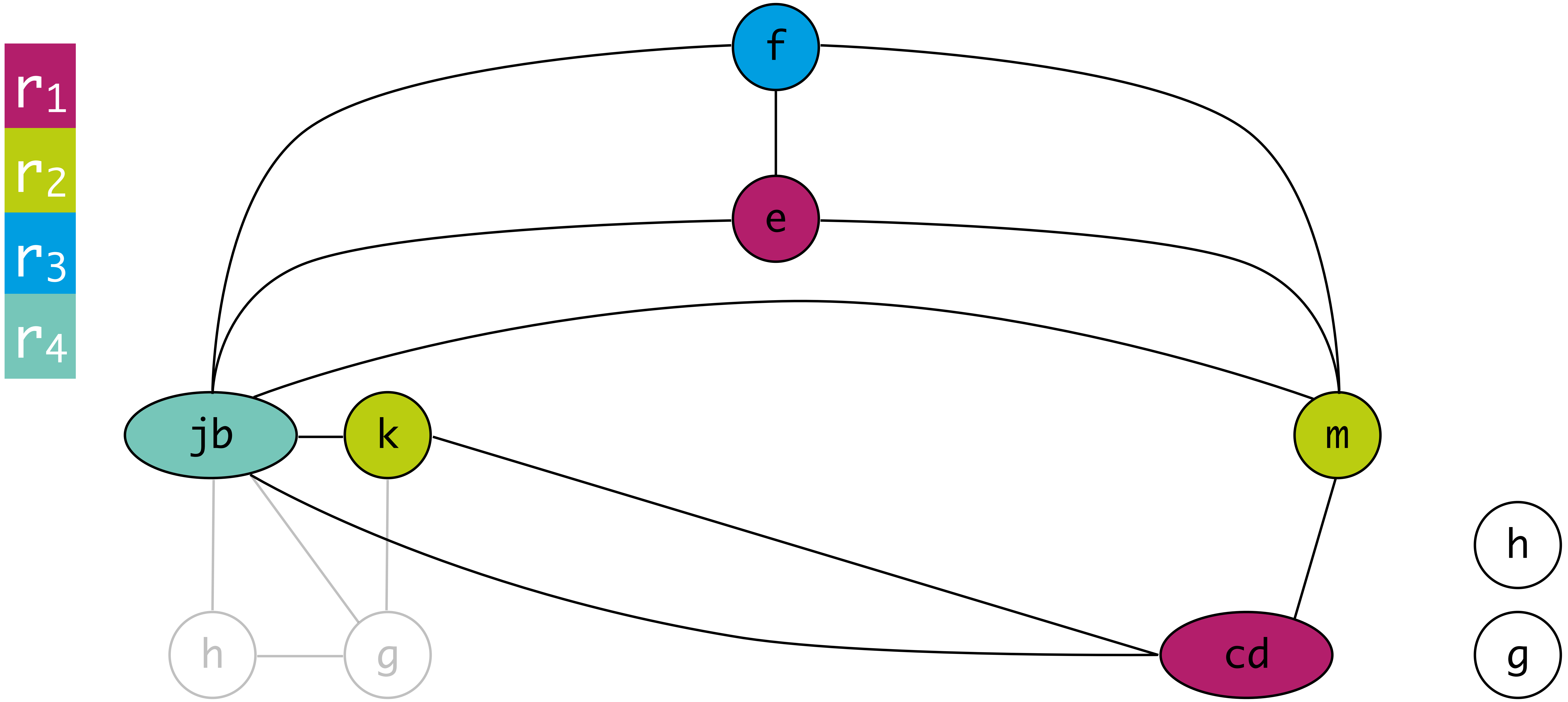

```
live-in: k r4
g := mem[r4 + 12]
h := k - 1
r3 := g * h
r1 := mem[r4 + 8]
r2 := mem[r4 + 16]
b := mem[r3]
c := r1 + 8
d := c
k := r2 + 4
r4 := r4
live out: d k r4
```

Coalescing



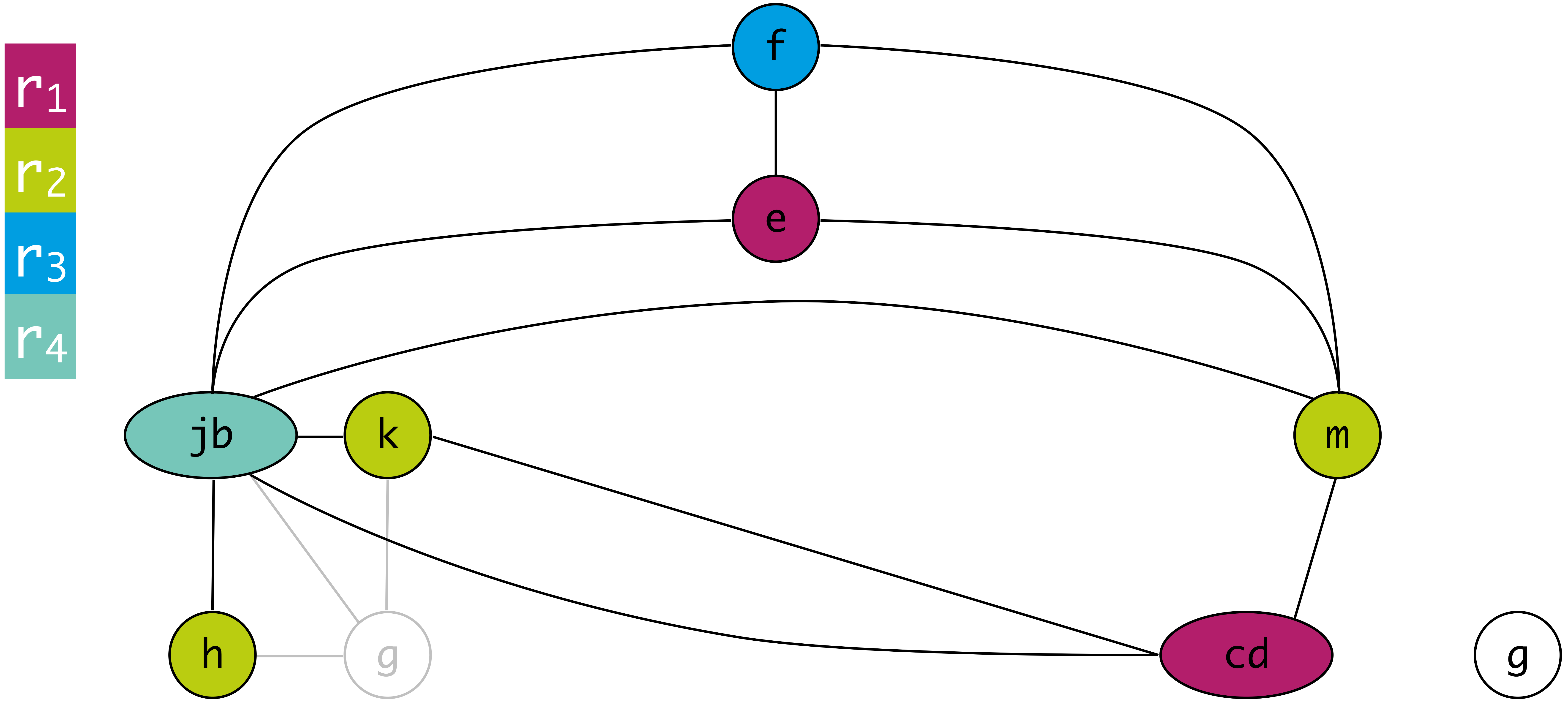

```
live-in: k r4
g := mem[r4 + 12]
h := k - 1
r3 := g * h
r1 := mem[r4 + 8]
r2 := mem[r4 + 16]
b := mem[r3]
r1 := r1 + 8
r1 := r1
k := r2 + 4
r4 := r4
live out: r1 k r4
```

Coalescing



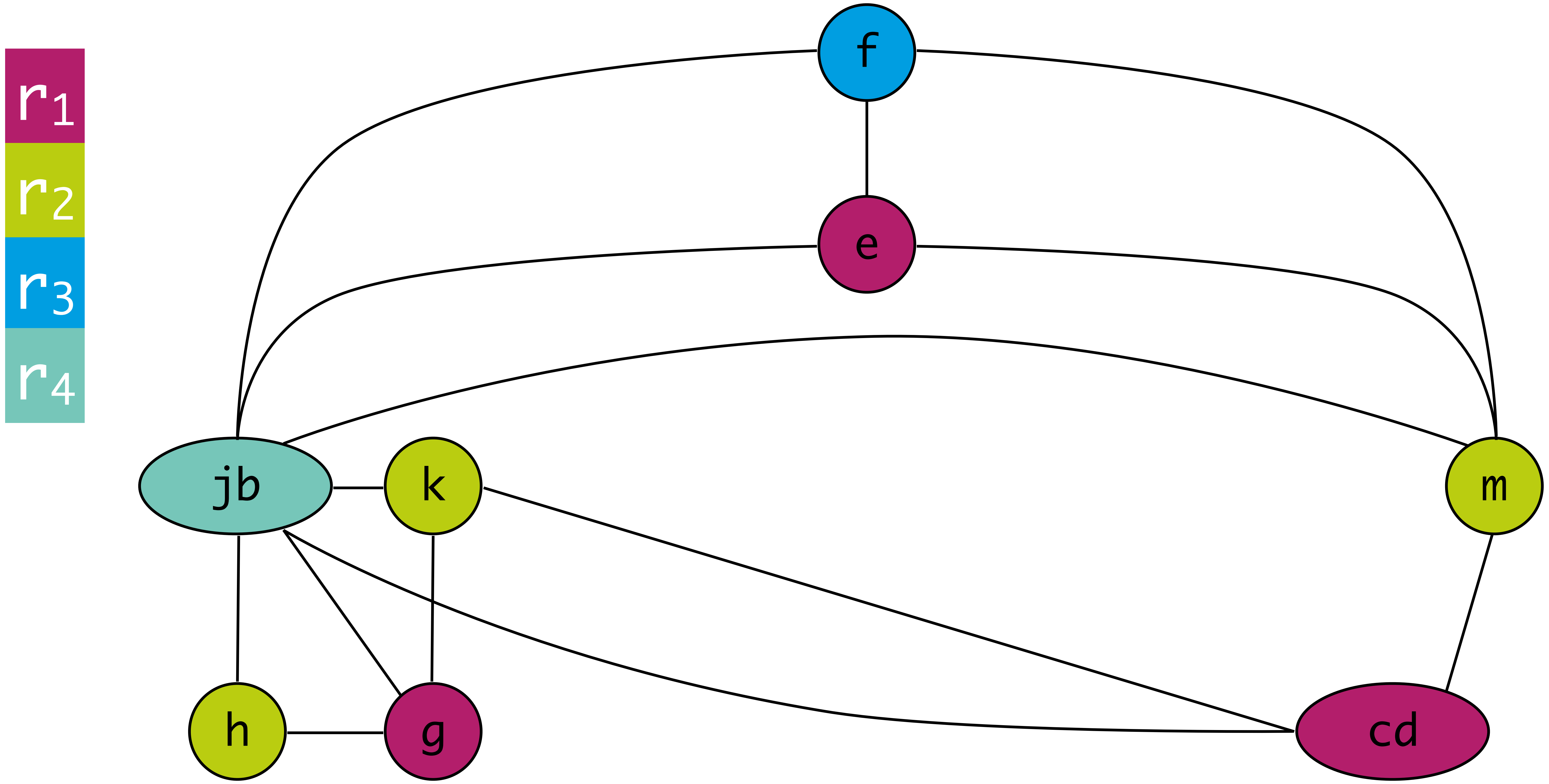

```
live-in: r2 r4
g := mem[r4 + 12]
h := r2 - 1
r3 := g * h
r1 := mem[r4 + 8]
r2 := mem[r4 + 16]
b := mem[r3]
r1 := r1 + 8
r1 := r1
k := r2 + 4
r4 := r4
live out: r1 r2 r4
```

Coalescing




```
live-in: r2 r4
g  := mem[r4 + 12]
r2  := r2 - 1
r3  := g * r2
r1  := mem[r4 + 8]
r2 := mem[r4 + 16]
b  := mem[r3]
r1  := r1 + 8
r1  := r1
k  := r2 + 4
r4  := r4
live out: r1 r2 r4
```

Coalescing




```
live-in: r2 r4
r1  := mem[r4 + 12]
r2  := r2 - 1
r3  := r1 * r2
r1  := mem[r4 + 8]
r2 := mem[r4 + 16]
b  := mem[r3]
r1  := r1 + 8
r1  := r1
k  := r2 + 4
r4  := r4
live out: r1 r2 r4
```

Pre-Colored Nodes

Recap: Calling Conventions: CDECL

Caller

- push parameters right-to-left on the stack
- clean-up stack after call

```
push 21
push 42
call _f
add  ESP 8
```

Callee

- save old BP
- initialise new BP
- save registers
- return result in AX
- restore registers
- restore BP

```
push EBP
mov  EBP ESP
mov  EAX [EBP + 8]
mov  EDX [EBP + 12]
add  EAX EDX
pop  EBP
ret
```

Recap: Calling Conventions: STDCALL

Caller

- push parameters right-to-left on the stack

```
push 21
push 42
call _f@8
```

Callee

- save old BP
- initialise new BP
- save registers
- return result in AX
- restore registers
- restore BP

```
push EBP
mov  EBP ESP
mov  EAX [EBP + 8]
mov  EDX [EBP + 12]
add  EAX EDX
pop  EBP
ret  8
```

Recap: Calling Conventions: FASTCALL

Caller

- passes parameters in registers
- pushes additional parameters right-to-left on the stack
- cleans up the stack

```
mov  ECX 21
mov  EDX 42
call @f@8
```

Callee

- save old BP, initialise new BP
- save registers
- return result in AX
- restore registers
- restore BP

```
push EBP
mov  EBP ESP
mov  EAX ECX
add  EAX EDX
pop  EBP
ret
```

Recap: Calling Conventions: Saving Registers

Not enough registers for all local variables across life time

- save register to memory to free for other use

Caller-save registers

- Caller is responsible for saving and restoring register

Callee-save registers

- Callee is responsible for saving and restoring register

Use callee-save registers to pass parameters

Pre-Colored Nodes: representing registers

Nodes

- register = pre-colored node
- no simplify, no spill
- coalesce possible

Edges

- all registers interfere with each other
- explicit usage of registers
- call and return instructions influence liveness

Callee-Save Register in Temporary

```
enter: def(r7)

         …

exit: use(r7)
```

```
enter:  def(r7)
        t ← r7

        …

        r7 ← t
exit:   use(r7)
```

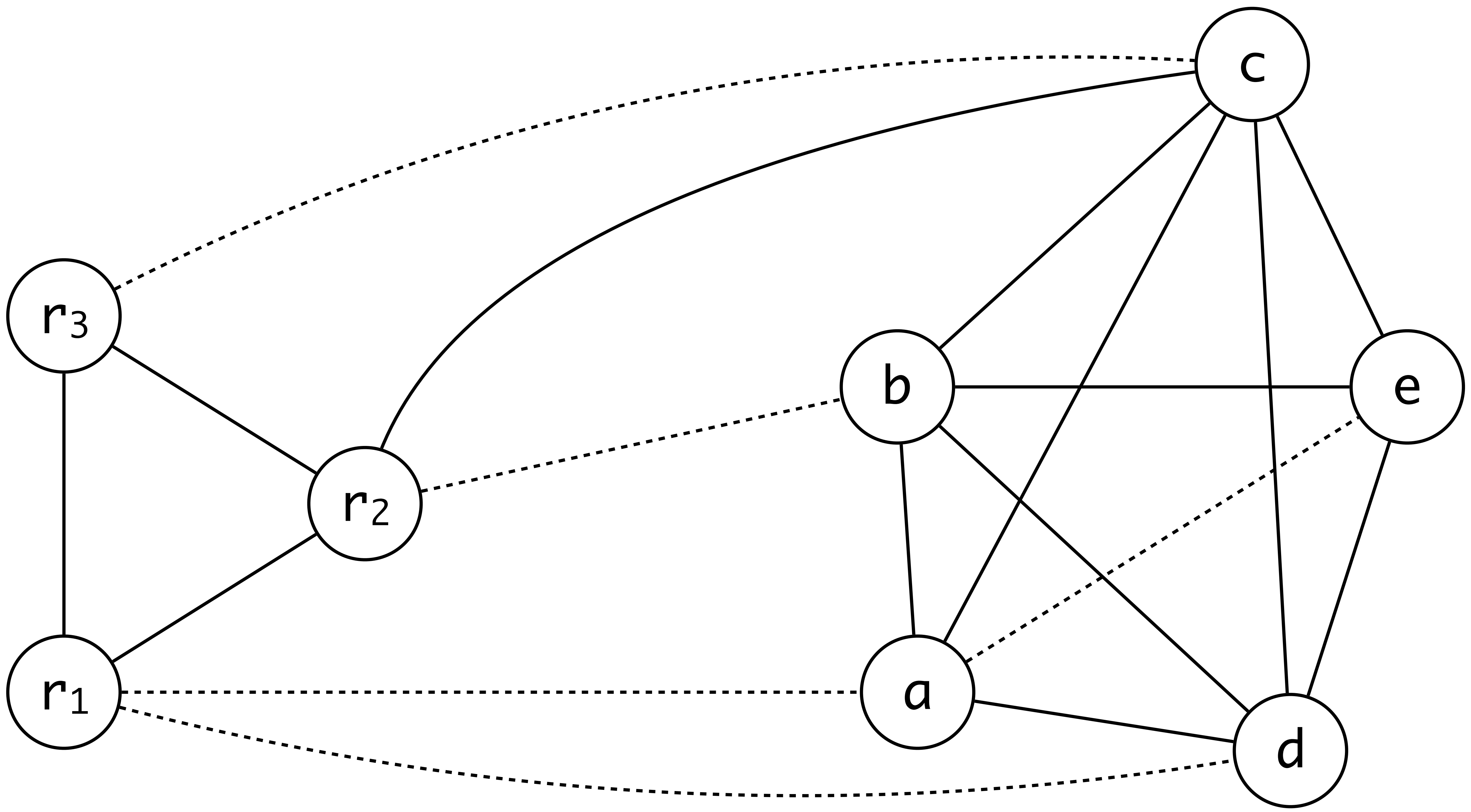
Pre-Colored Nodes

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

```
enter : c ← r3    // callee-save
        a ← r1    // caller-save
        b ← r2    // caller-save
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3 live out)
```

machine has 3 registers

Pre-Colored Nodes

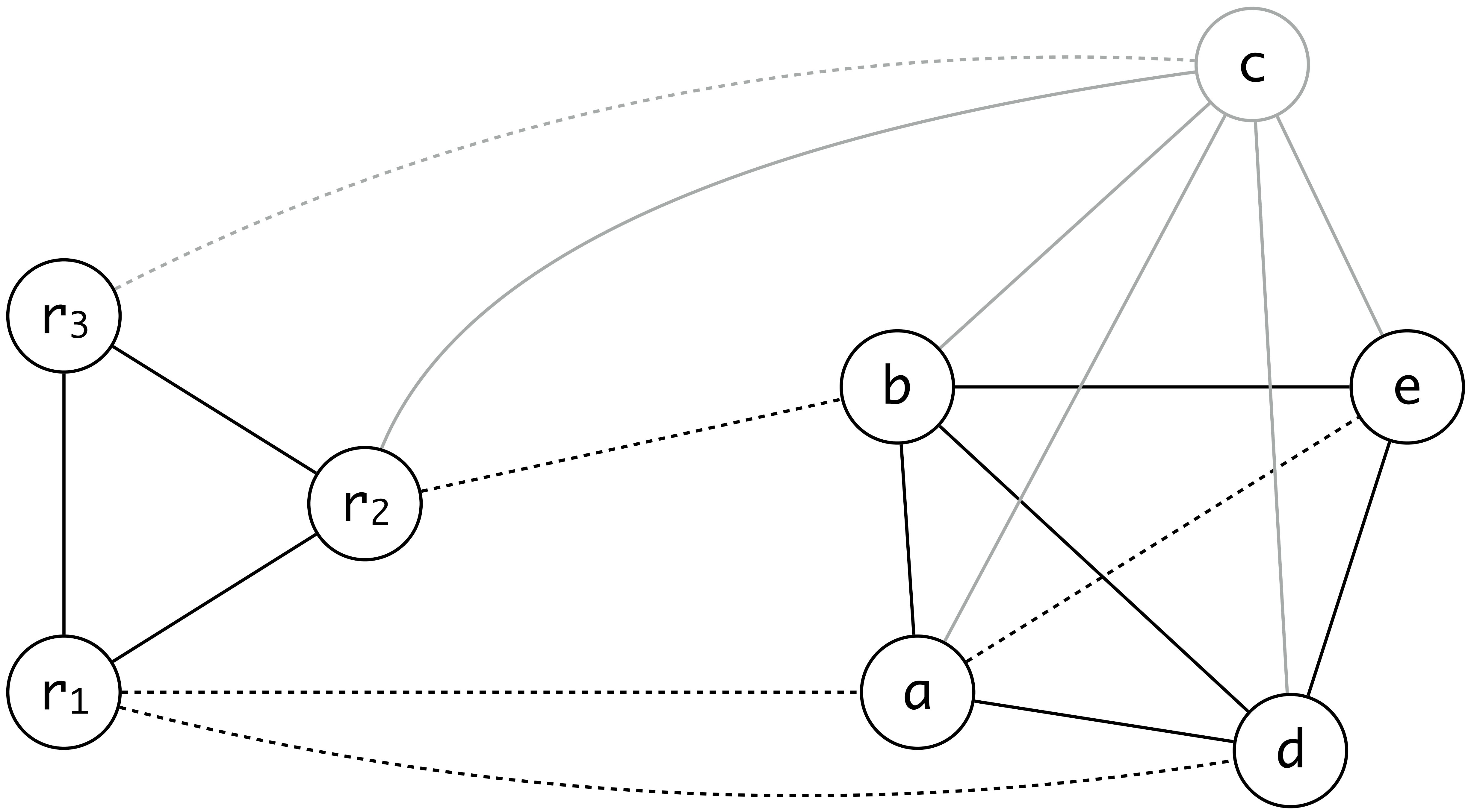


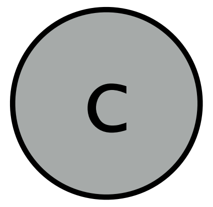

```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

Pre-Colored Nodes

```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

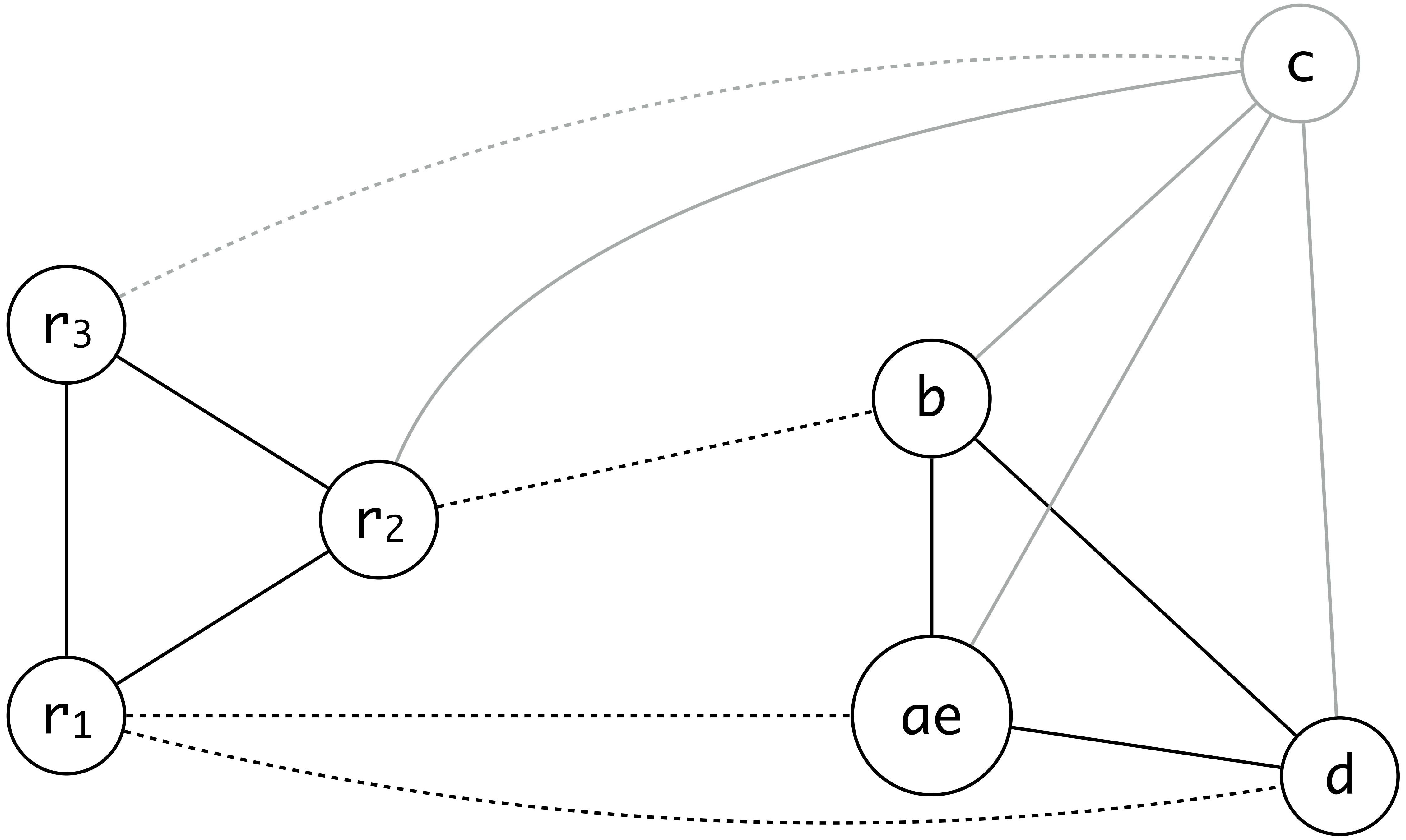
spill c



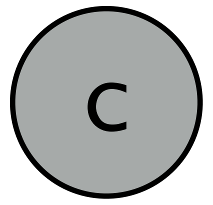


Pre-Colored Nodes

coalesce a and e

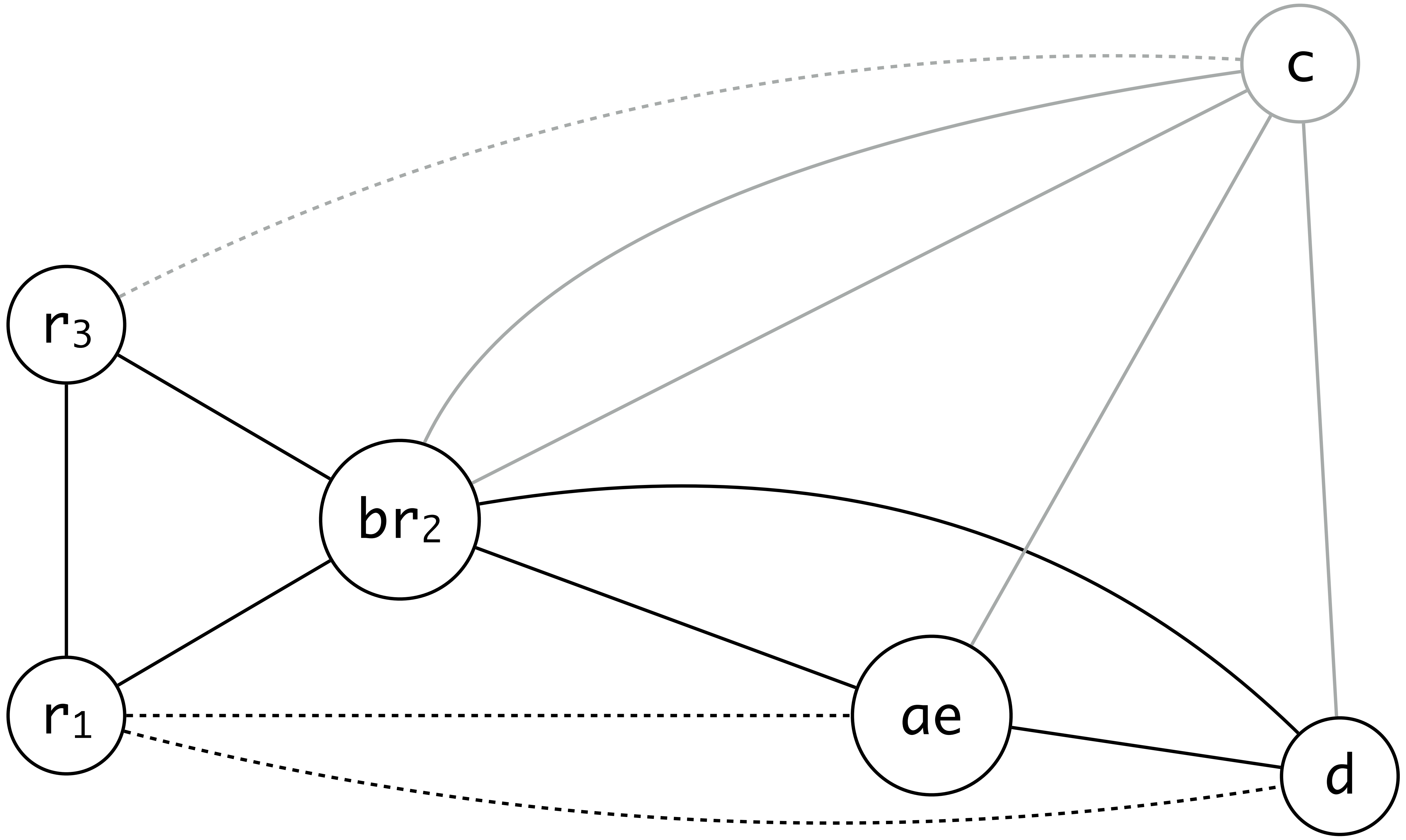



```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

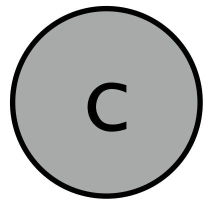


Pre-Colored Nodes

coalesce r_2 and b

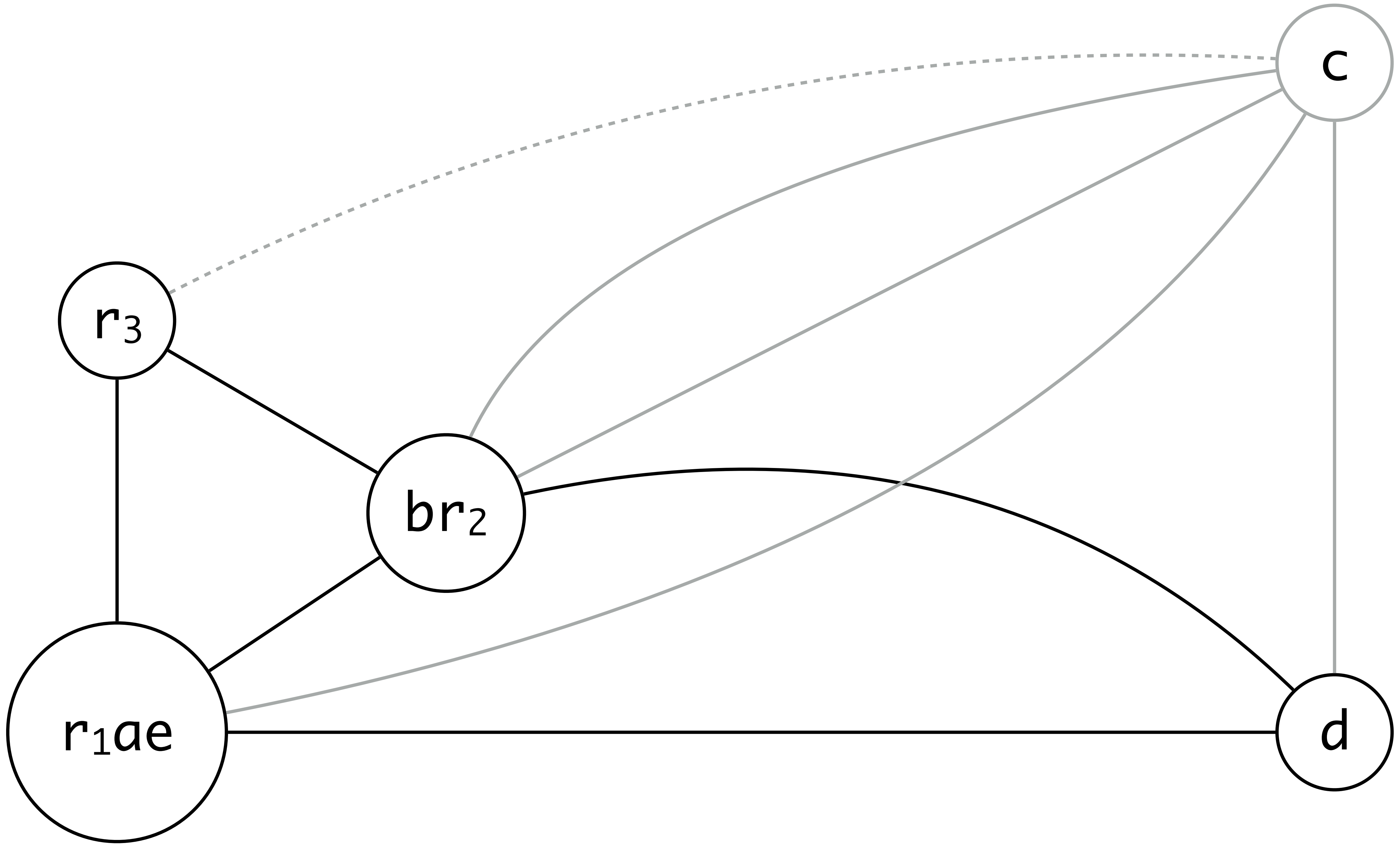



```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

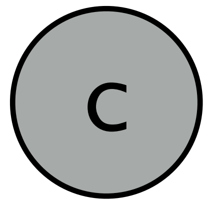


Pre-Colored Nodes

coalesce r_1 and ae

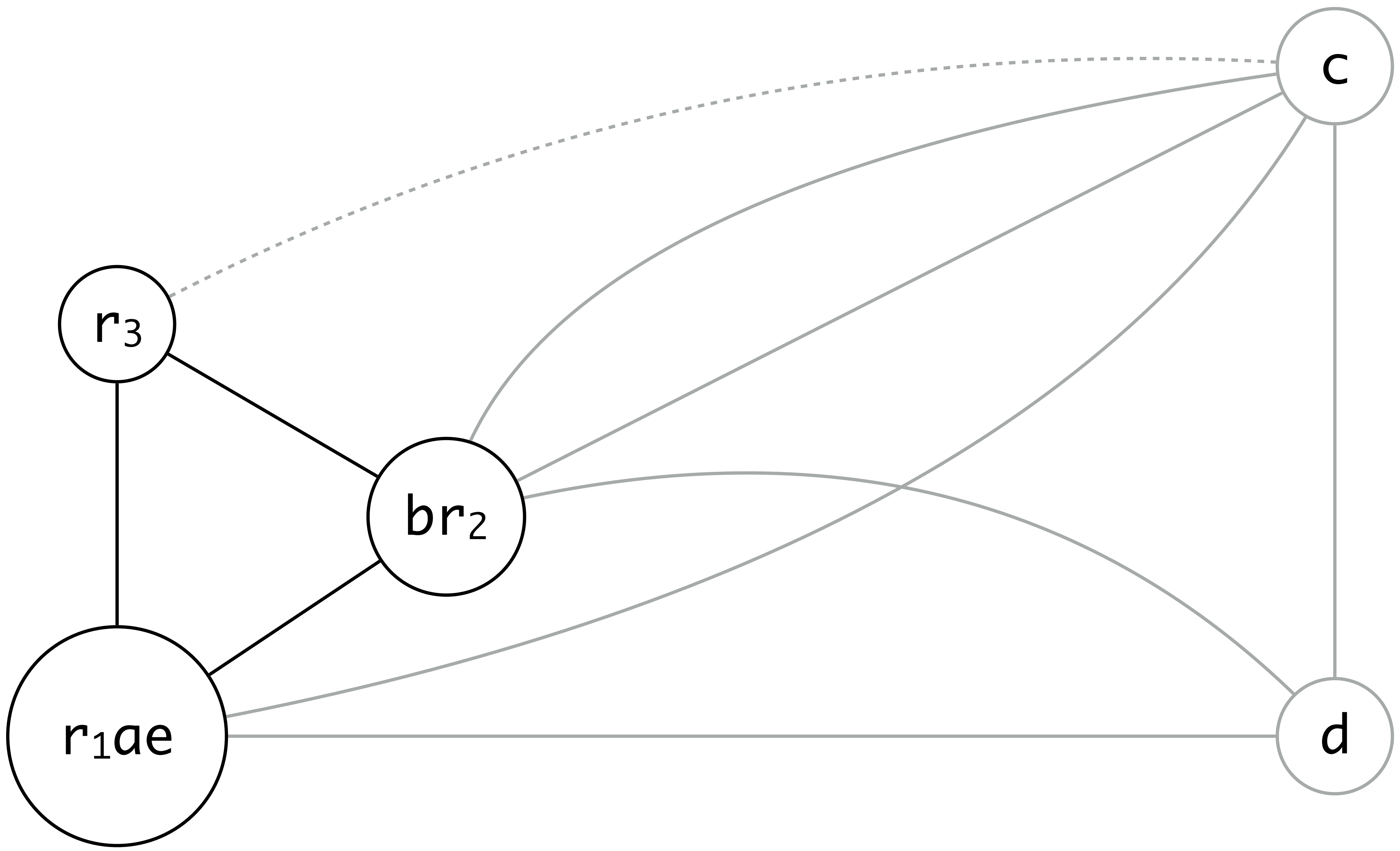



```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

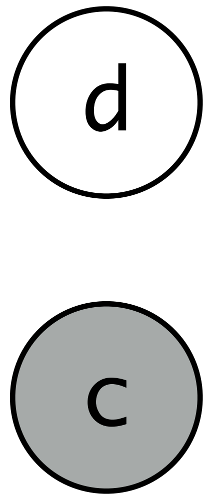


Pre-Colored Nodes

simplify d

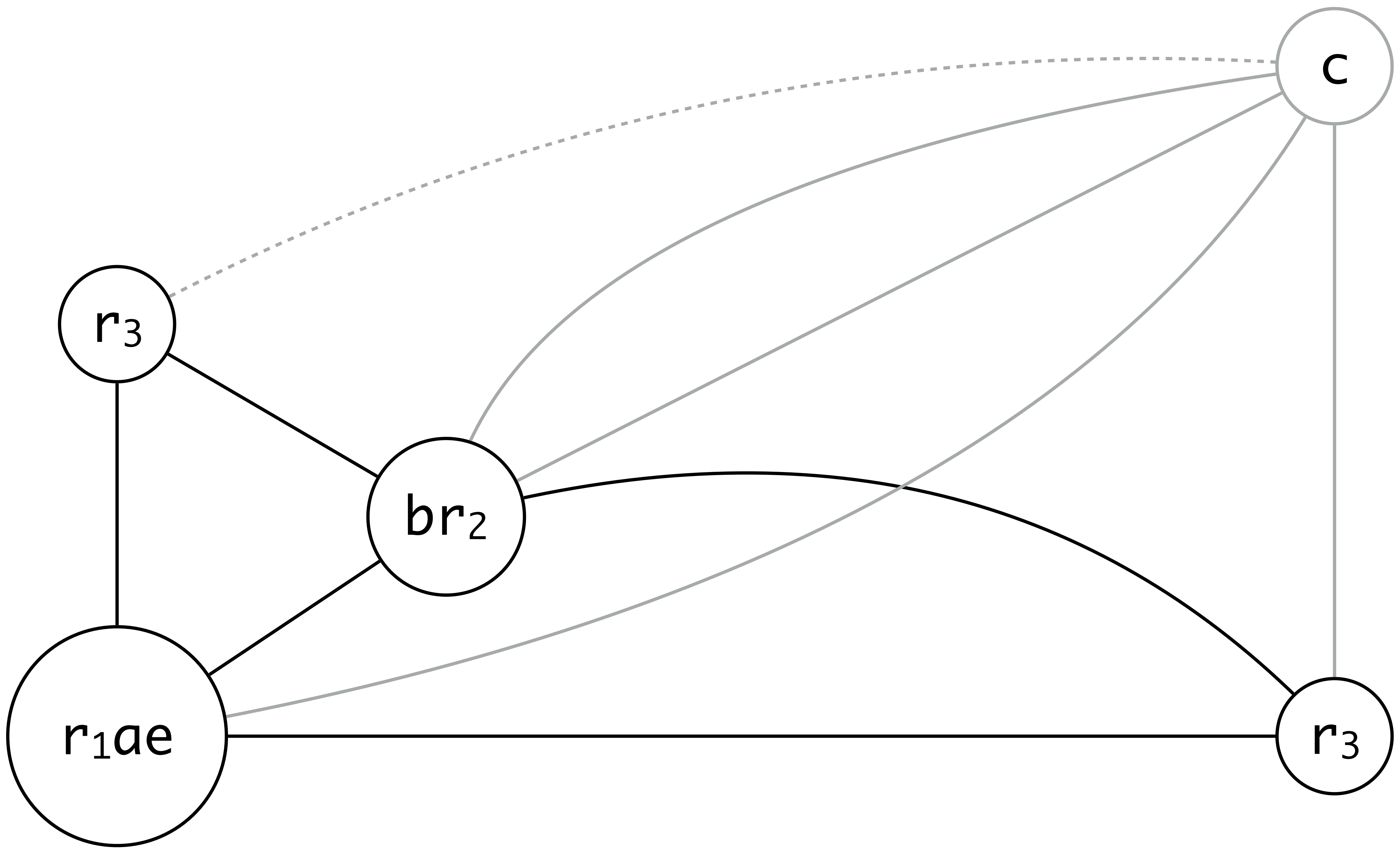



```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

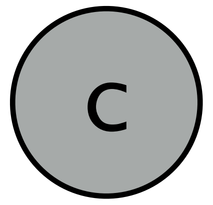


Pre-Colored Nodes

color d as r_3

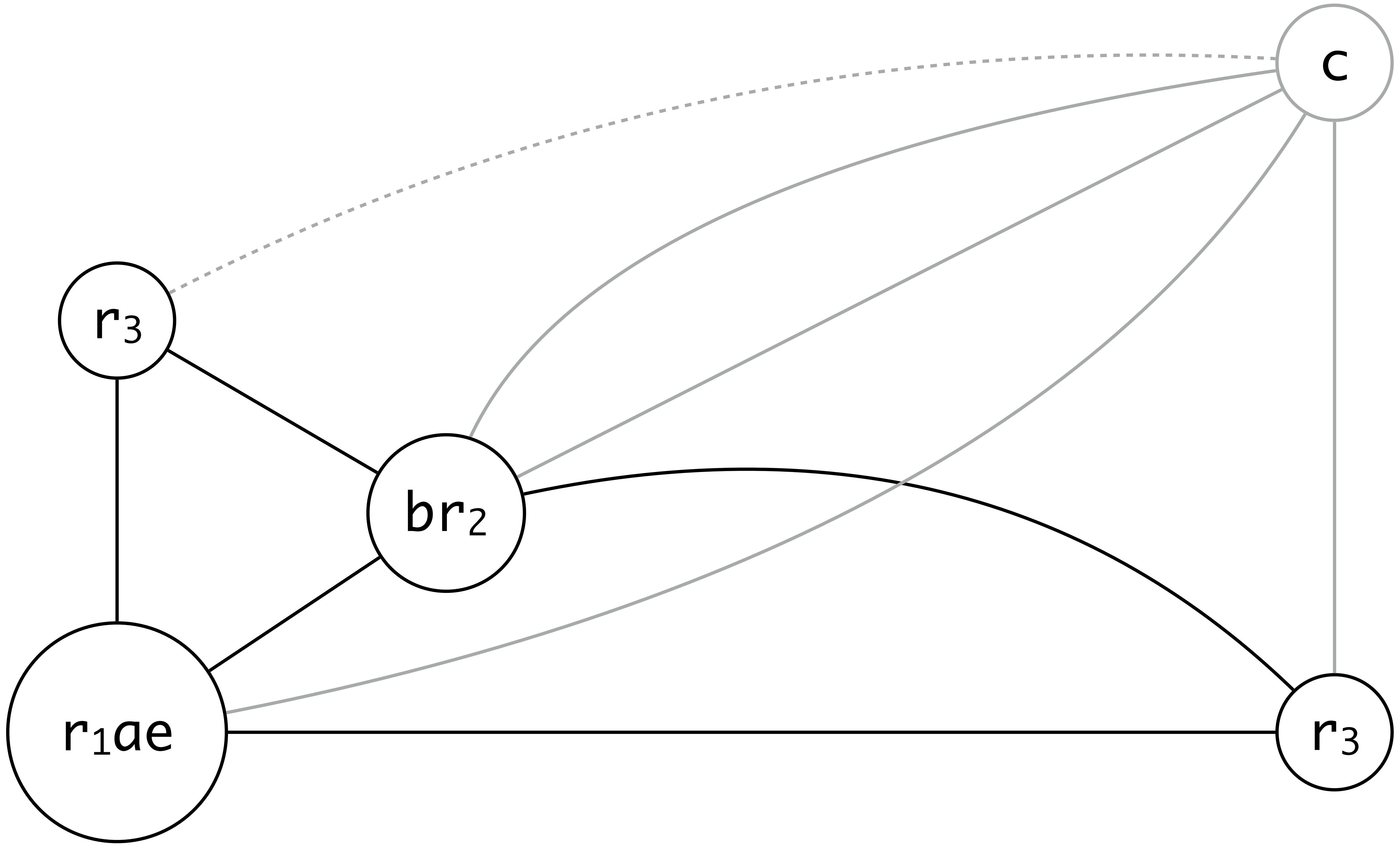



```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

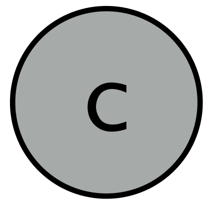


Pre-Colored Nodes

spill c

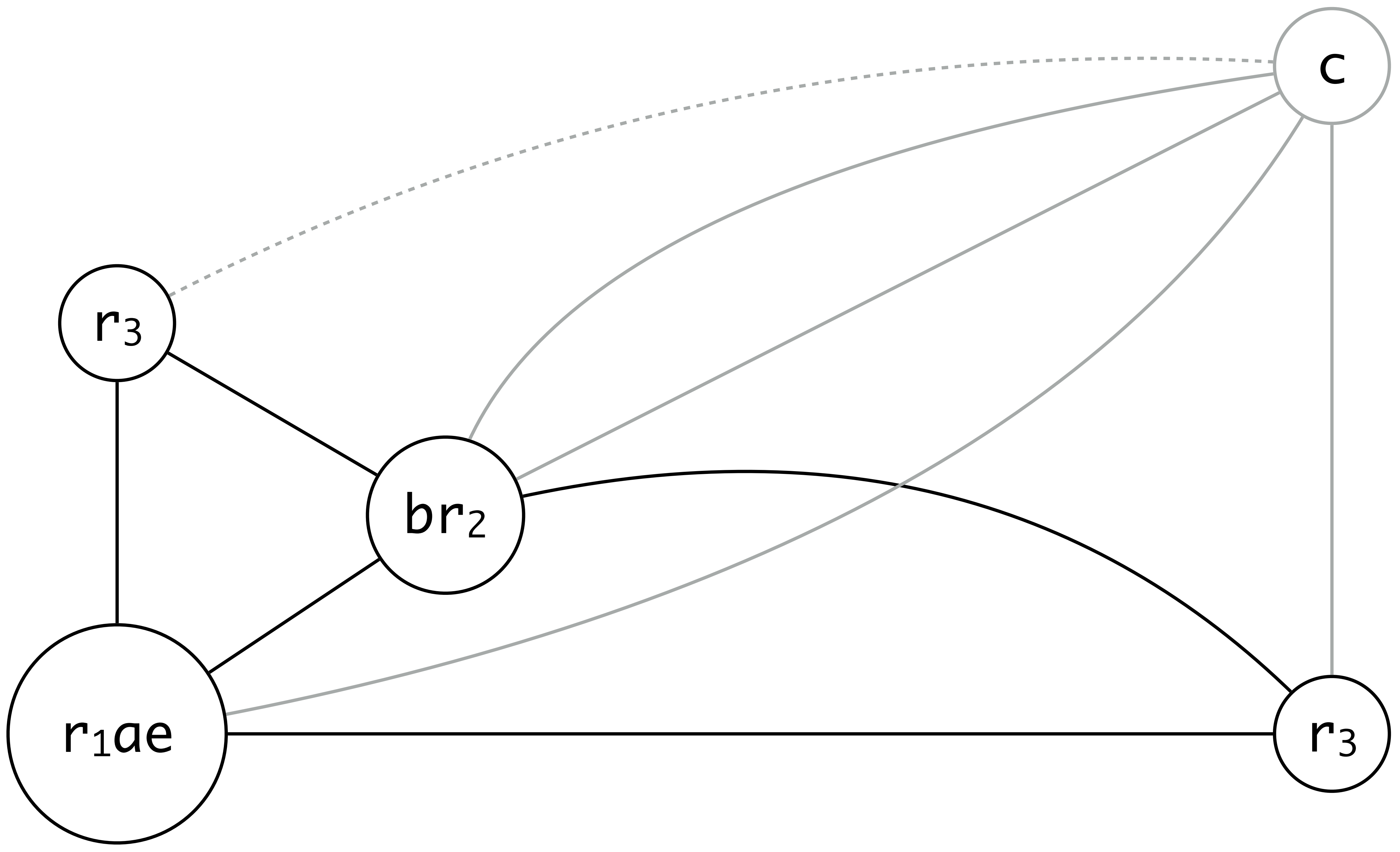



```
enter : c₁ ← r₃
        M[c_loc] ← c₁
        a ← r₁
        b ← r₂
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r₁ ← d
        r₃ ← c₂
        c₂ ← M[c_loc]
        return (r₁, r₃)
```



Pre-Colored Nodes

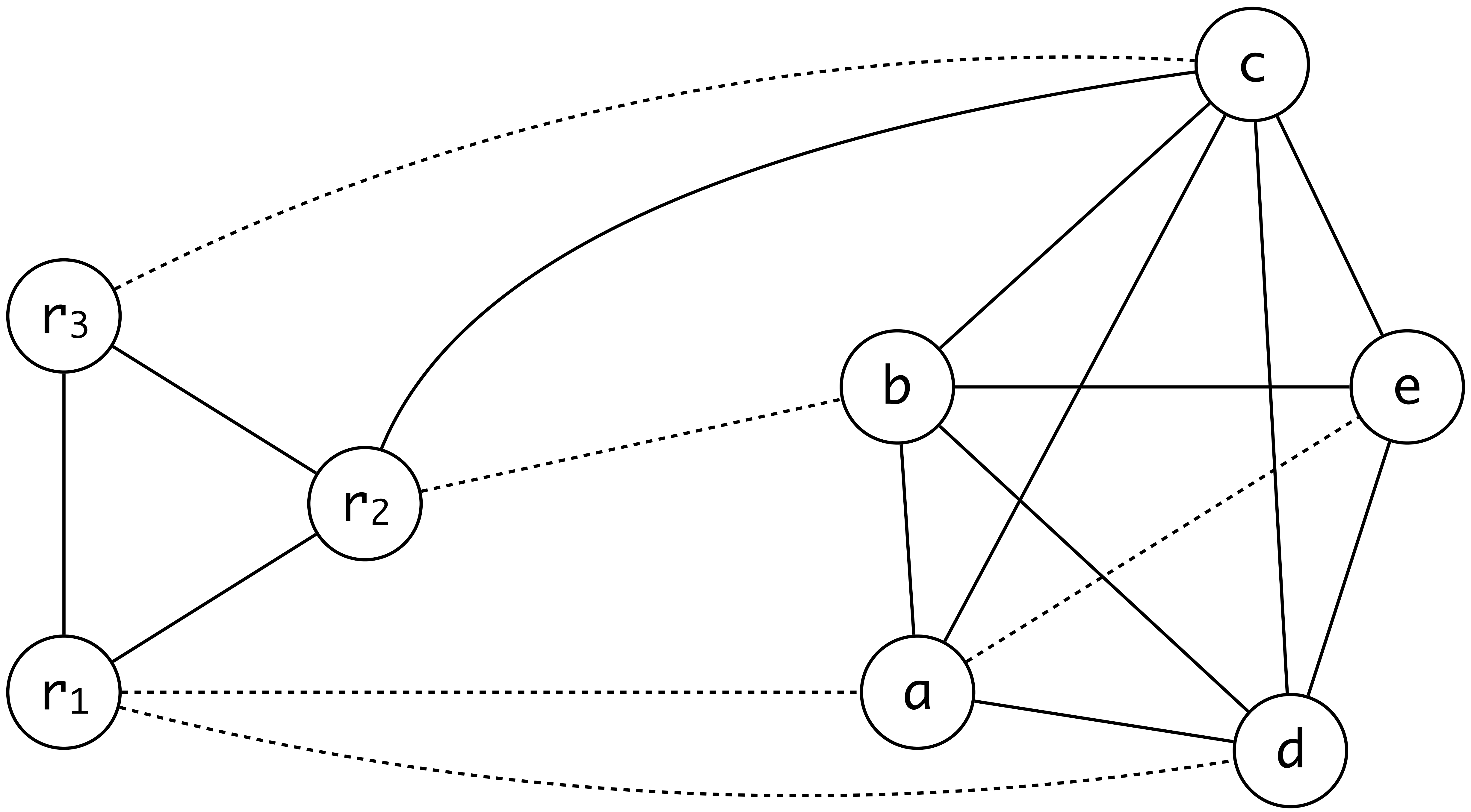
spill c




```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

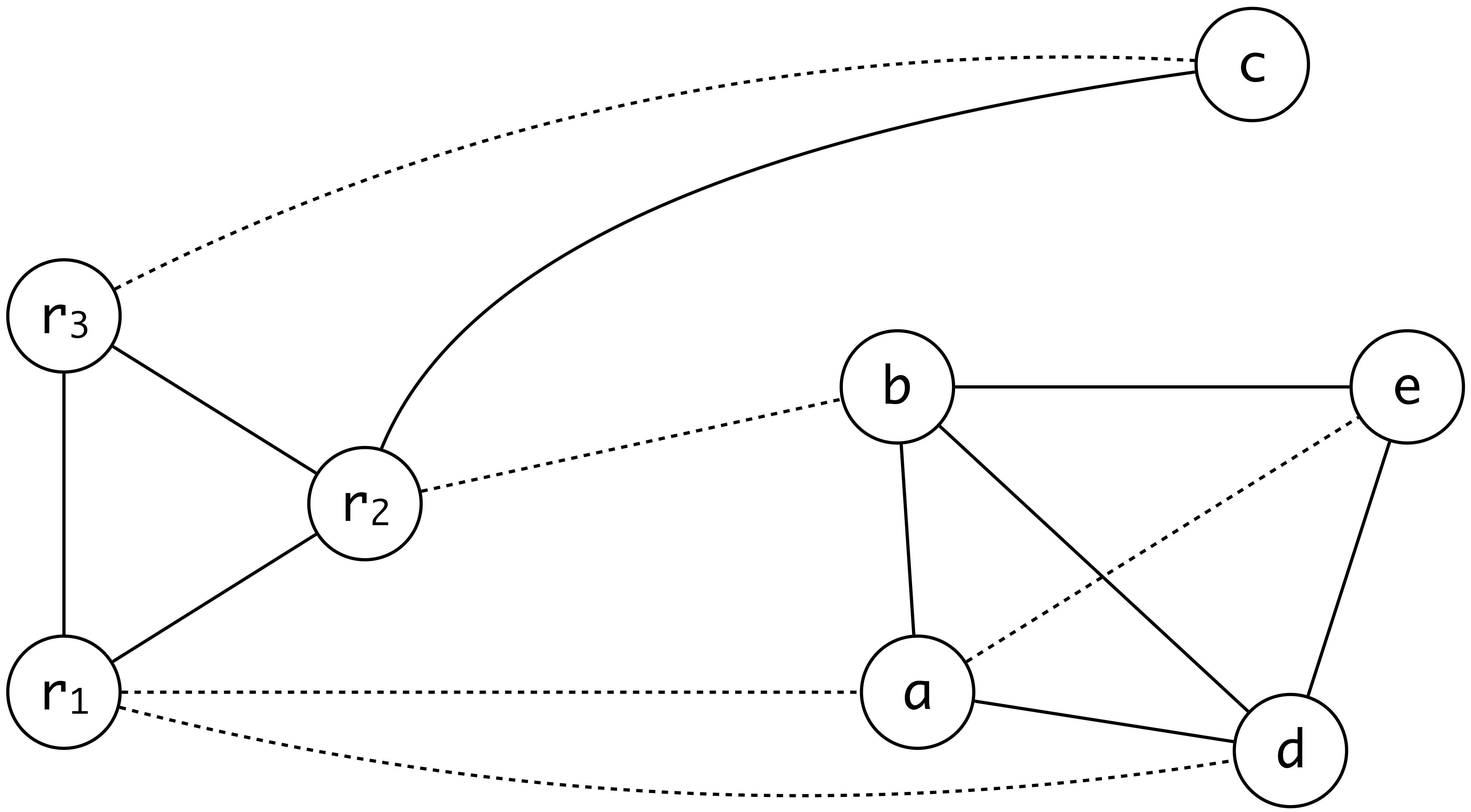
Pre-Colored Nodes

start over




```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

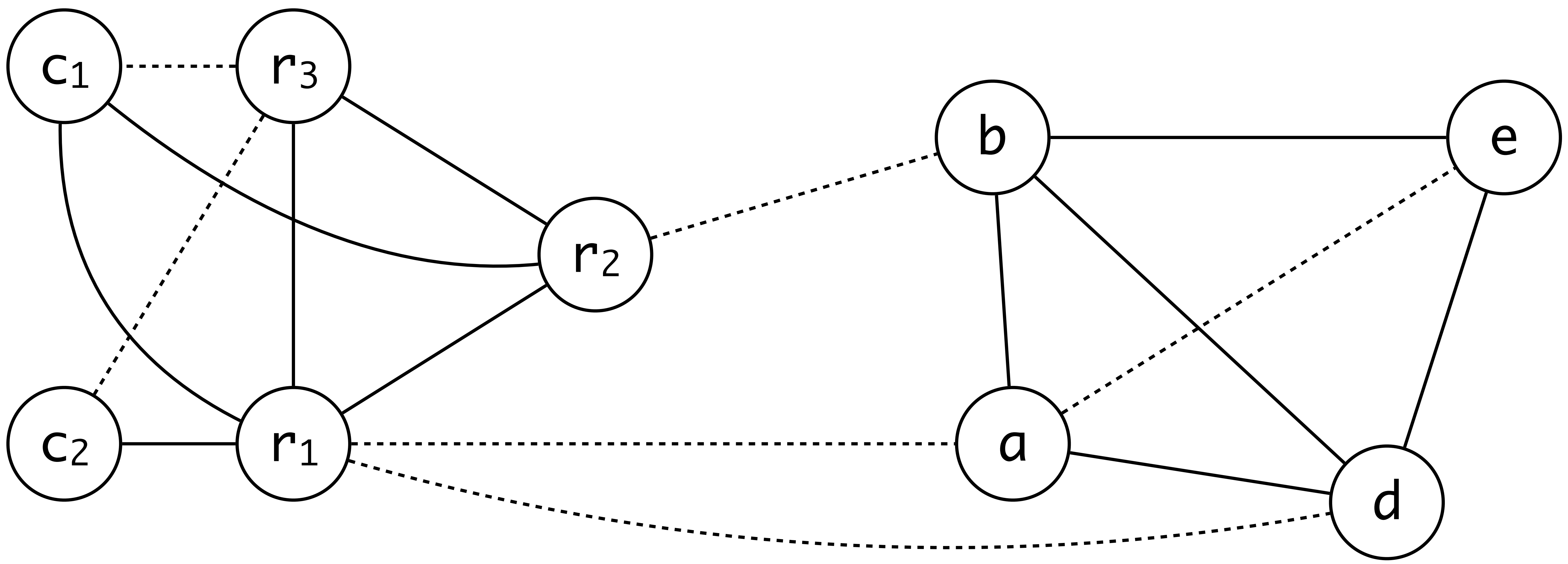
Pre-Colored Nodes




```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

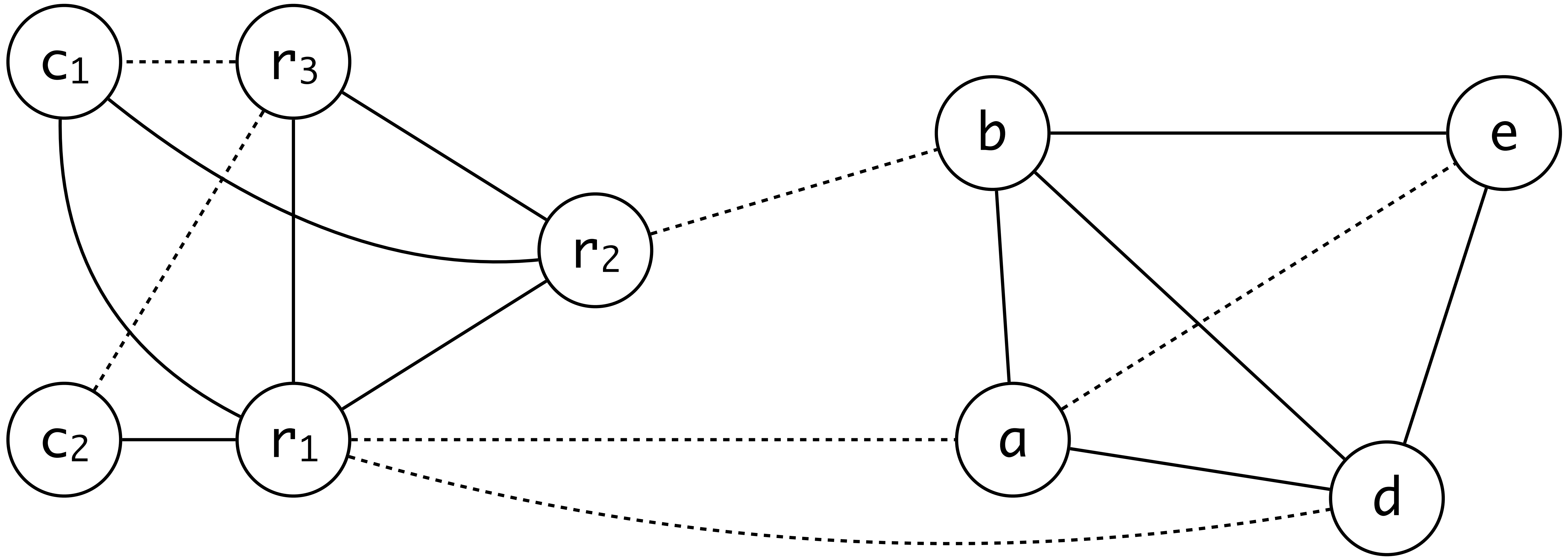
new graph




```
enter : c₁ ← r₃
        M[c_loc] ← c₁
        a ← r₁
        b ← r₂
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r₁ ← d
        r₃ ← c₂
        c₂ ← M[c_loc]
        return (r₁, r₃)
```

Pre-Colored Nodes

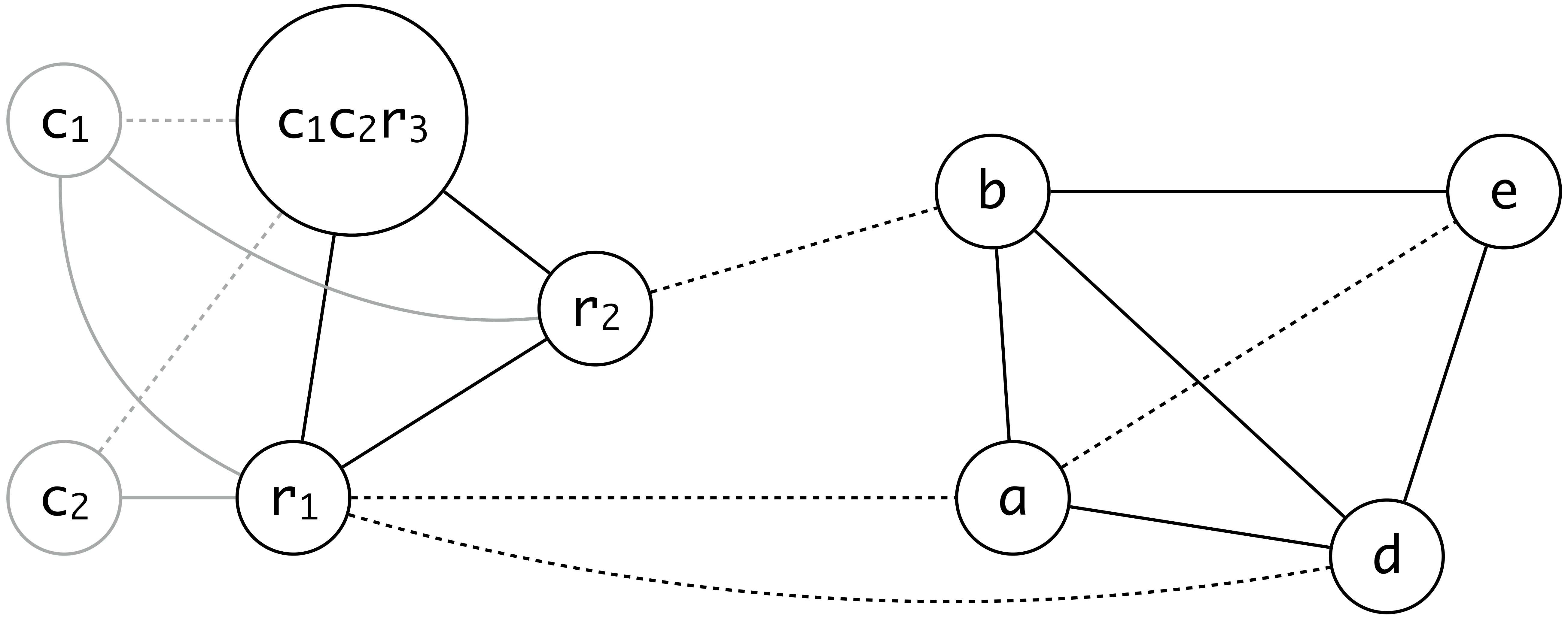
coalesce c_1, c_2, r_3




```
enter :  c1 ← r3
         M[cloc] ← c1
         a ← r1
         b ← r2
         d ← 0
         e ← a
loop  :  d ← d + b
         e ← e - 1
         if e > 0 goto loop
         r1 ← d
         r3 ← c2
         c2 ← M[cloc]
         return (r1, r3)
```

Pre-Colored Nodes

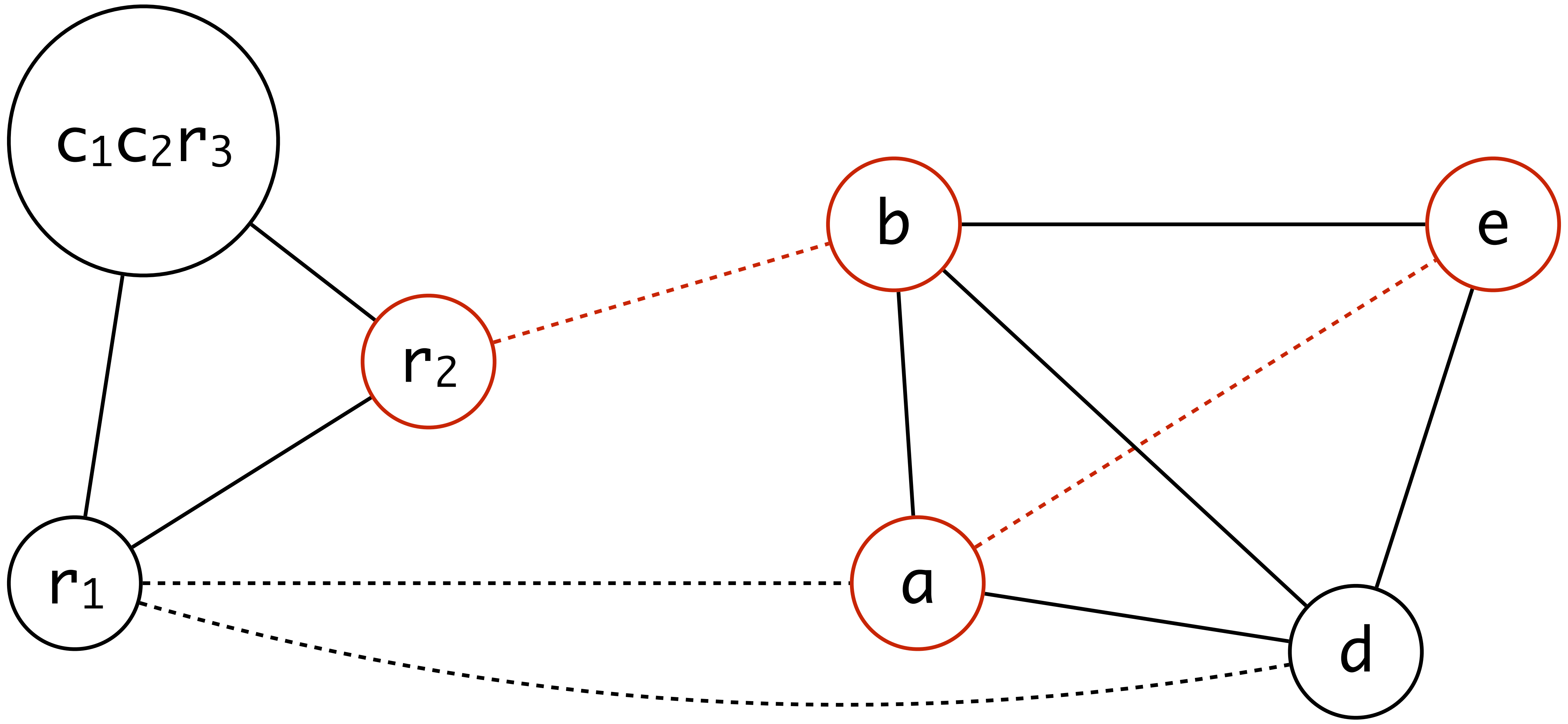
coalesce c_1, c_2, r_3




```
enter : c₁ ← r₃
        M[c_loc] ← c₁
        a ← r₁
        b ← r₂
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r₁ ← d
        r₃ ← c₂
        c₂ ← M[c_loc]
        return (r₁, r₃)
```

Pre-Colored Nodes

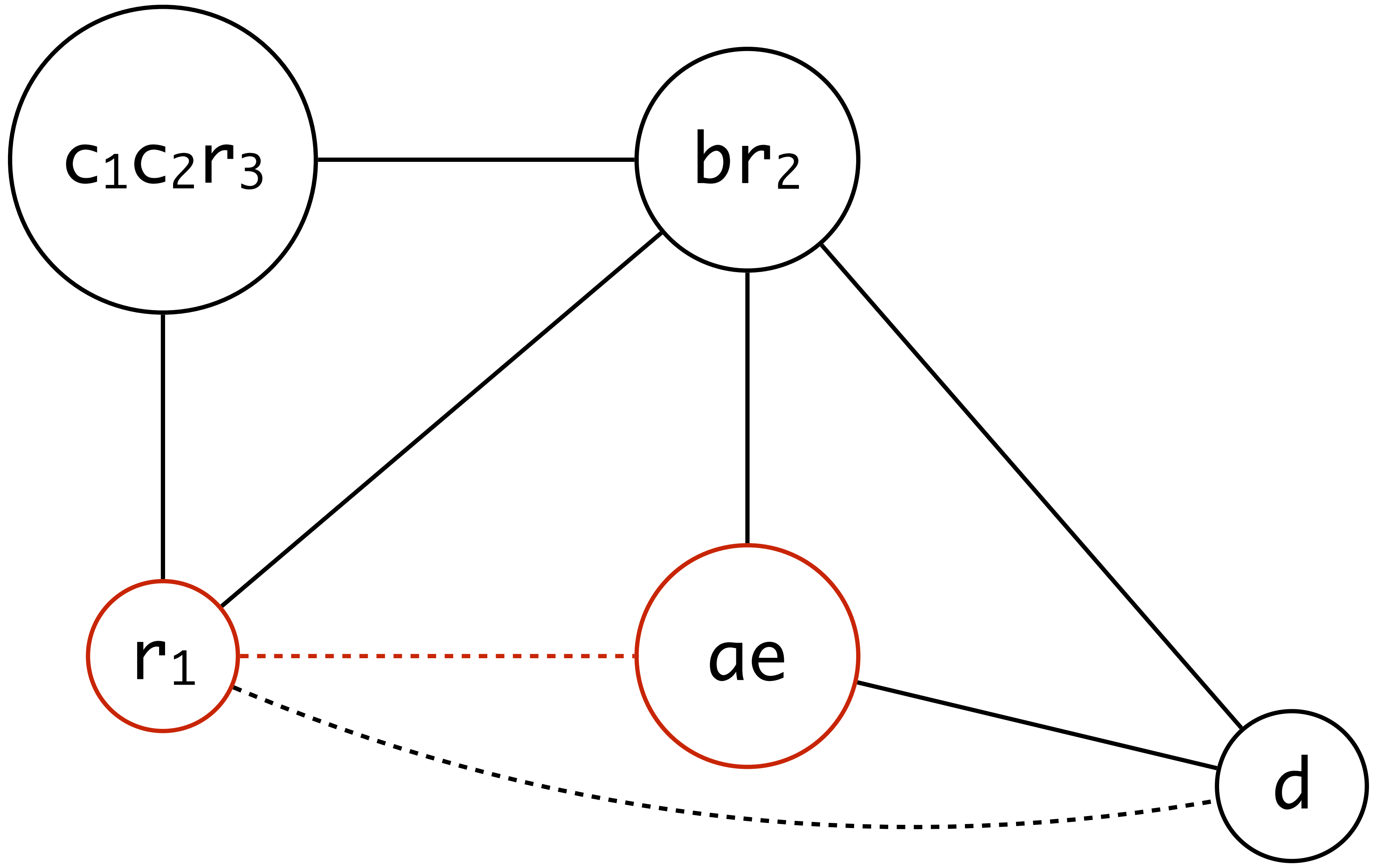
coalesce (b, r_2) and (a, e)




```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

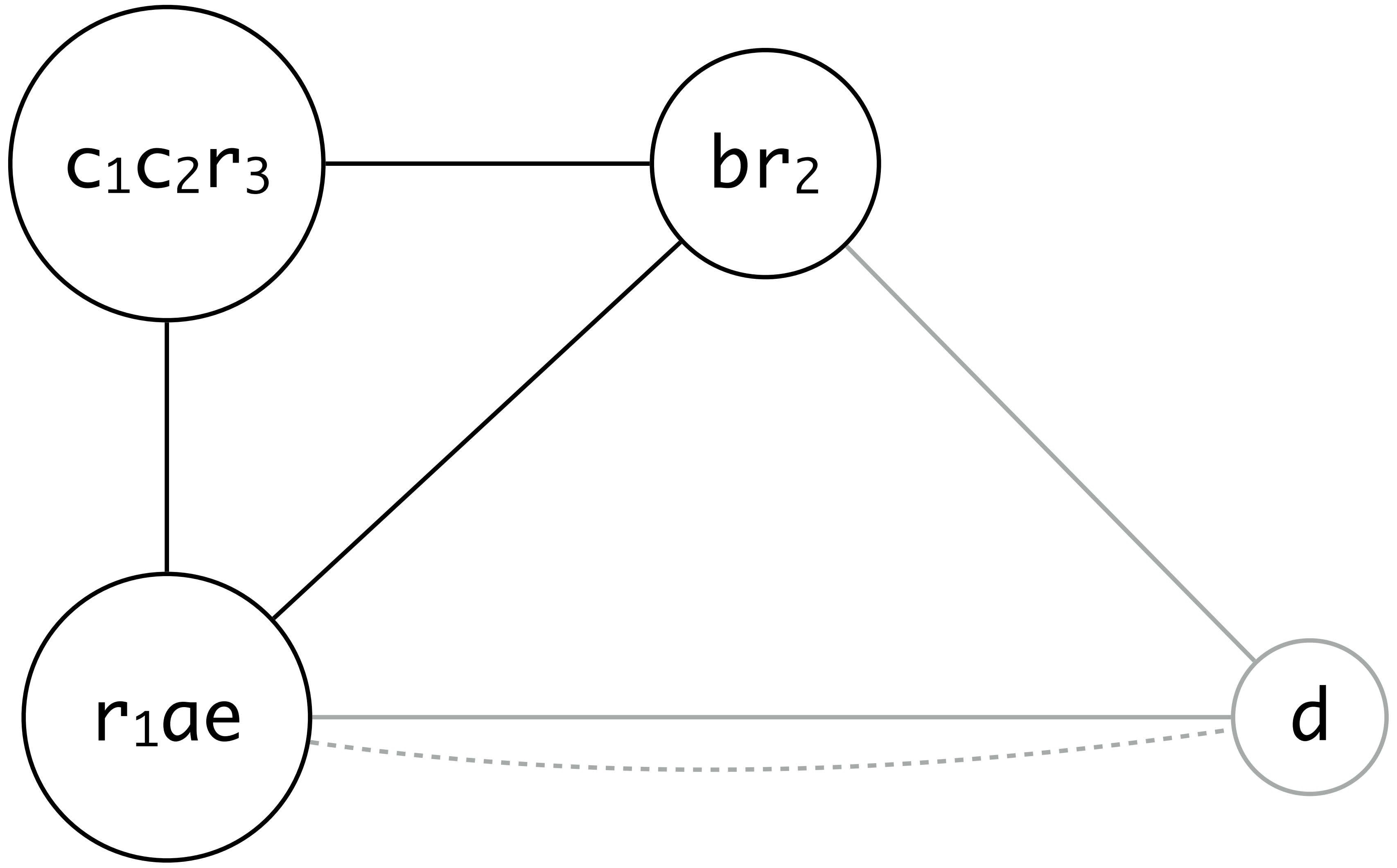
coalesce (ae, r_1)



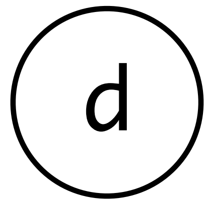

```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

simplify d

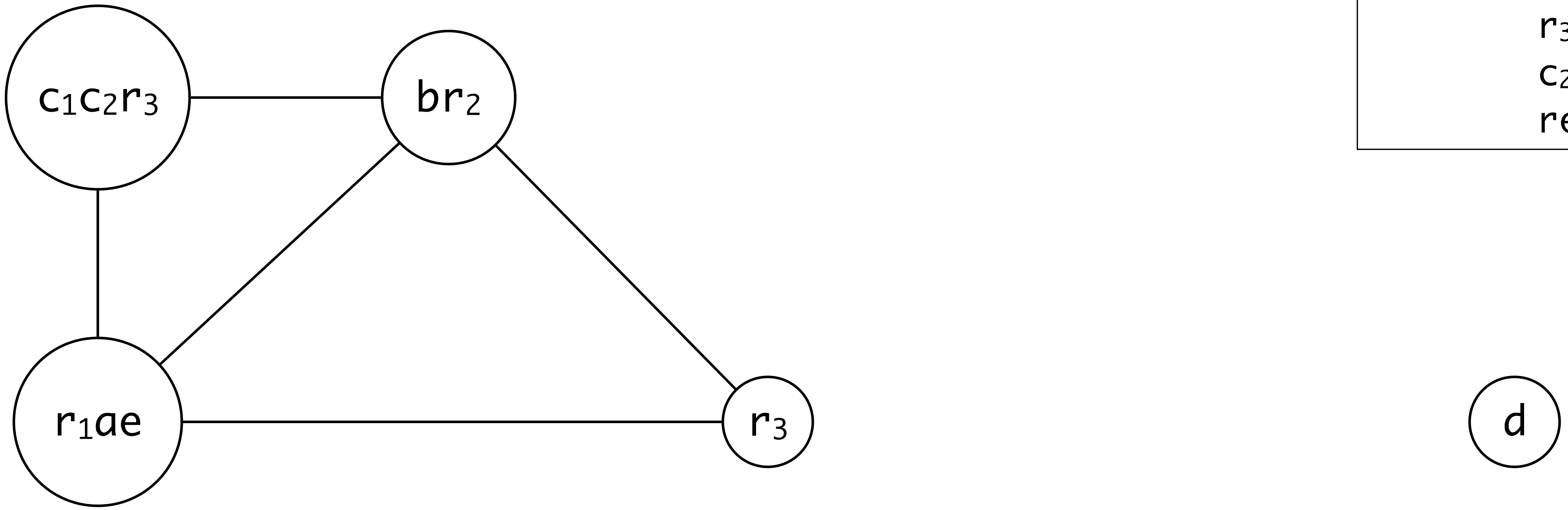



```
enter : c₁ ← r₃
        M[cloc] ← c₁
        a ← r₁
        b ← r₂
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r₁ ← d
        r₃ ← c₂
        c₂ ← M[cloc]
        return (r₁, r₃)
```



Pre-Colored Nodes

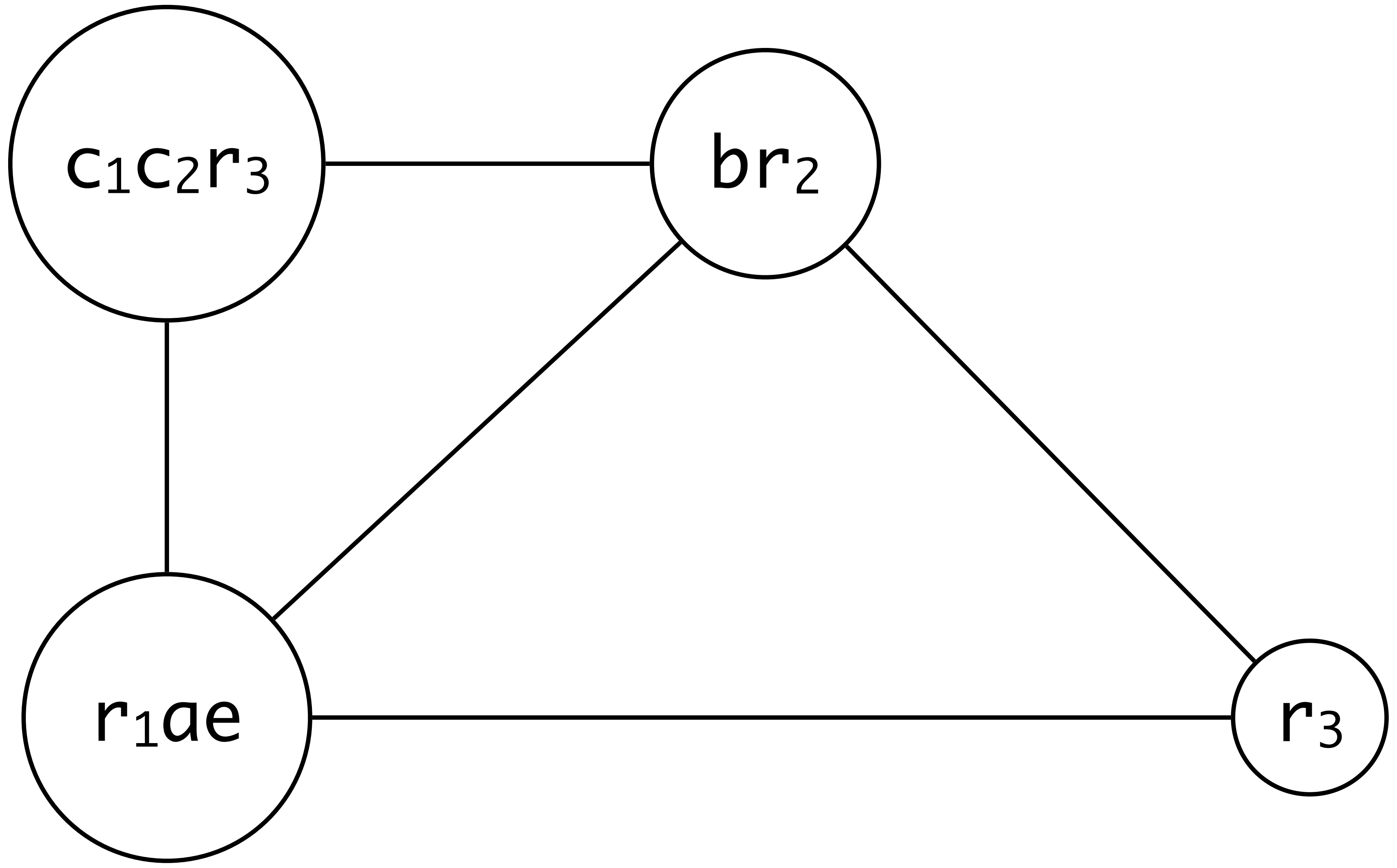
color d as r_3




```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

apply register assigment




```
enter : r3 ← r3
        M[cloc] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop :  r3 ← r3 + r2
        r1 ← r1 - 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← r3
        r3 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

```
enter : r3 ← r3
        M[c_loc] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop :  r3 ← r3 + r2
        r1 ← r1 - 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← r3
        r3 ← M[c_loc]
        return (r1, r3)
```

Pre-Colored Nodes

```
enter :  c ← r3
         a ← r1
         b ← r2
         d ← 0
         e ← a
loop :   d ← d + b
         e ← e - 1
         if e > 0 goto loop
         r1 ← d
         r3 ← c
         return (r1, r3)
```

```
enter :  r3 ← r3
         M[cloc] ← r3
         r1 ← r1
         r2 ← r2
         r3 ← 0
         r1 ← r1
loop :   r3 ← r3 + r2
         r1 ← r1 - 1
         if r1 > 0 goto loop
         r1 ← r3
         r3 ← r3
         r3 ← M[cloc]
         return (r1, r3)
```

Pre-Colored Nodes

```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

```
enter : M[cloc] ← r3
        r3 ← 0
loop :  r3 ← r3 + r2
        r1 ← r1 - 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

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

```
enter : M[c_loc] ← r3
        r3 ← 0
loop :  r3 ← r3 + r2
        r1 ← r1 - 1
        if r1 > 0 goto loop
        r3 ← M[c_loc]
        return (r1, r3)
```

Summary

Summary

How can we assign registers to local variables and temporaries?

- perform liveness analysis
- build interference graph
- color interference graph

What to do if the graph is not colorable?

- keep local variables in memory

How to handle move instructions efficiently?

- coalesce nodes safely

Literature

Andrew W. Appel, Jens Palsberg: Modern Compiler Implementation in Java, 2nd edition. 2002

Lal George, Andrew W. Appel: Iterative Register Coalescing. POPL 1996

Lal George, Andrew W. Appel: Iterative Register Coalescing. TOPLAS 18(3), 1996