Representing Objects & Register Allocation

Eelco Visser

TU Delft

CS4200 | Compiler Construction | December 17, 2020

This Lecture

Objects

- classes and methods in ChocoPy
- object layout

Register Allocation

Objects

Classes and Methods in ChocoPy

class definition

inheritance

```
class animal(object):
  makes_noise:bool = False

  def make_noise(self: "animal") → object:
    if (self.makes_noise):
      print(self.sound())

  def sound(self: "animal") → str:
    return "???"

class cow(animal):
  def __init__(self: "cow"):
    self.makes_noise = True

  def sound(self: "cow") → str:
    return "moo"

c:animal = None
c = cow()
c.make_noise()
```

attribute

method definition

method call

object initialization

attribute reference

object construction

method call

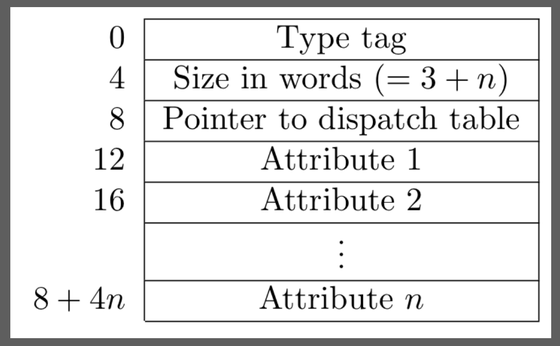
Object Layout

0	Type tag
4	Size in words $(= 3 + n)$
8	Pointer to dispatch table
12	Attribute 1
16	Attribute 2
	$\vdots$
$8 + 4n$	Attribute n

Type tag	**Type**
0	(reserved)
1	`int`
2	`bool`
3	`str`
-1	`[T]`

Prototypes

```
class animal(object):
  makes_noise:bool = False

  def make_noise(self: "animal") -> object:
    if (self.makes_noise):
      print(self.sound())

  def sound(self: "animal") -> str:
    return "???"

class cow(animal):
  def __init__(self: "cow"):
    self.makes_noise = True

  def sound(self: "cow") -> str:
    return "moo"

c:animal = None
c = cow()
c.make_noise()
```



0	Type tag
4	Size in words $(= 3 + n)$
8	Pointer to dispatch table
12	Attribute 1
16	Attribute 2
	⋮
$8 + 4n$	Attribute n

```
.globl $object$prototype
$object$prototype:
  .word 0                        # Type tag for class: object
  .word 3                        # Object size
  .word $object$dispatchTable # Pointer to dispatch table
  .align 2

.globl $int$prototype
$int$prototype:
  .word 1                       # Type tag for class: int
  .word 4                       # Object size
  .word $int$dispatchTable   # Pointer to dispatch table
  .word 0                       # Initial value of attribute: __int__
  .align 2

.globl $animal$prototype
$animal$prototype:
  .word 4                        # Type tag for class: animal
  .word 4                        # Object size
  .word $animal$dispatchTable # Pointer to dispatch table
  .word 0                        # Initial value of attribute: makes_noise
  .align 2

.globl $cow$prototype
$cow$prototype:
  .word 5                        # Type tag for class: cow
  .word 4                        # Object size
  .word $cow$dispatchTable    # Pointer to dispatch table
  .word 0                        # Initial value of attribute: makes_noise
  .align 2
```

Prototypes & Dispatch Tables

```
class animal(object):
  makes_noise:bool = False

  def make_noise(self: "animal") -> object:
    if (self.makes_noise):
      print(self.sound())

  def sound(self: "animal") -> str:
    return "???"

class cow(animal):
  def __init__(self: "cow"):
    self.makes_noise = True

  def sound(self: "cow") -> str:
    return "moo"

c:animal = None
c = cow()
c.make_noise()
```

```
.globl $animal$prototype
$animal$prototype:
  .word 4                       # Type tag for class: animal
  .word 4                       # Object size
  .word $animal$dispatchTable   # Pointer to dispatch table
  .word 0                       # Initial value of attribute: makes_noise
  .align 2

.globl $cow$prototype
$cow$prototype:
  .word 5                       # Type tag for class: cow
  .word 4                       # Object size
  .word $cow$dispatchTable      # Pointer to dispatch table
  .word 0                       # Initial value of attribute: makes_noise
  .align 2
```

same interface as super class

include inherited methods

override methods

```
.globl $animal$dispatchTable
$animal$dispatchTable:
  .word $object.__init__    # Implementation for method: animal.__init__
  .word $animal.make_noise  # Implementation for method: animal.make_noise
  .word $animal.sound       # Implementation for method: animal.sound

.globl $cow$dispatchTable
$cow$dispatchTable:
  .word $cow.__init__       # Implementation for method: cow.__init__
  .word $animal.make_noise  # Implementation for method: cow.make_noise
  .word $cow.sound          # Implementation for method: cow.sound
```

Object Creation & Initialization

```python
class animal(object):
  makes_noise:bool = False

  def make_noise(self: "animal") -> object:
    if (self.makes_noise):
      print(self.sound())

  def sound(self: "animal") -> str:
    return "???"

class cow(animal):
  def __init__(self: "cow"):
    self.makes_noise = True

  def sound(self: "cow") -> str:
    return "moo"

c:animal = None
c = cow()
c.make_noise()
```

alloc copies prototype

```
  la a0, $cow$prototype       # Load pointer to prototype of: cow
  jal alloc                   # Allocate new object in A0

  sw a0, -12(fp)              # Push on stack slot 3
  sw a0, -16(fp)              # Push argument 0 from last.
  addi sp, fp, -16            # Set SP to last argument.
  lw a1, 8(a0)                # Load address of object's dispatch table
  lw a1, 0(a1)                # Load address of method: cow.__init__
  jalr a1                     # Invoke method: cow.__init__

  addi sp, fp, -@..main.size # Set SP to stack frame top.
  lw a0, -12(fp)              # Pop stack slot 3
  sw a0, $c, t0               # Assign global: c (using tmp register)
```

constructor calls __init_ method

```
.globl $cow.__init__
$cow.__init__:
  …
  li a0, 1                          # Load boolean literal: true
  sw a0, -12(fp)                    # Push on stack slot 3
  lw a0, 0(fp)                      # Load var: cow.__init__.self
  mv a1, a0                         # Move object
  lw a0, -12(fp)                    # Pop stack slot 3
  bnez a1, label_11                 # Ensure not None
  j error.None                      # Go to error handler
label_11:
  sw a0, 12(a1)                     # Set attribute: cow.makes_noise
  …
  jr ra                             # Return to caller
```

0	Type tag
4	Size in words $(= 3 + n)$
8	Pointer to dispatch table
12	Attribute 1
16	Attribute 2
	⋮
$8 + 4n$	Attribute n

Method Call: Dynamic Dispatch

```
class animal(object):
  makes_noise:bool = False

  def make_noise(self: "animal") -> object:
    if (self.makes_noise):
      print(self.sound())

  def sound(self: "animal") -> str:
    return "???"

class cow(animal):
  def __init__(self: "cow"):
    self.makes_noise = True

  def sound(self: "cow") -> str:
    return "moo"

c:animal = None
c = cow()
c.make_noise()
```

not null check

do not invoke function label directly

```
  lw a0, $c         # Load global: c
  bnez a0, label_1 # Ensure not None
  j error.None      # Go to error handler

label_1:
  sw a0, -16(fp)    # Push argument 0 from last.
  lw a0, -16(fp)    # Peek stack slot 3
  lw a1, 8(a0)      # Load address of object's dispatch table
  lw a1, 4(a1)      # Load address of method: animal.make_noise
  addi sp, fp, -16 # Set SP to last argument.
  jalr a1           # Invoke method: animal.make_noise
```

look up address of actual method in dispatch table

0	Type tag
4	Size in words $(= 3 + n)$
8	Pointer to dispatch table
12	Attribute 1
16	Attribute 2
	⋮
$8 + 4n$	Attribute n

```
.globl $animal$dispatchTable
$animal$dispatchTable:
  .word $object.__init__   # Implementation for method: animal.__init__
  .word $animal.make_noise # Implementation for method: animal.make_noise
  .word $animal.sound      # Implementation for method: animal.sound

.globl $cow$dispatchTable
$cow$dispatchTable:
  .word $cow.__init__      # Implementation for method: cow.__init__
  .word $animal.make_noise # Implementation for method: cow.make_noise
  .word $cow.sound         # Implementation for method: cow.sound
```

Accessing Attributes

```
class animal(object):
  makes_noise:bool = False

  def make_noise(self: "animal") -> object:
    if (self.makes_noise):
      print(self.sound())

  def sound(self: "animal") -> str:
    return "???"

class cow(animal):
  def __init__(self: "cow"):
    self.makes_noise = True

  def sound(self: "cow") -> str:
    return "moo"

c:animal = None
c = cow()
c.make_noise()
```

offset in object in memory

```
.globl $animal.make_noise
$animal.make_noise:

  …

  lw a0, 0(fp)          # Load var: animal.make_noise.self
  bnez a0, label_5      # Ensure not None
  j error.None          # Go to error handler
label_5:
  lw a0, 12(a0)         # Get attribute: animal.makes_noise
  beqz a0, label_4      # Branch on false.

  …

label_4:
  mv a0, zero           # Load None
  j label_3             # Jump to function epilogue
label_3:
  …
  jr ra                 # Return to caller
```

0	Type tag
4	Size in words $(= 3 + n)$
8	Pointer to dispatch table
12	Attribute 1
16	Attribute 2
	⋮
$8 + 4n$	Attribute n

Boxed vs Unboxed Values

4.2 Unwrapped Values

Parameters, local variables, global variables, and attributes whose static types are `int` or `bool` are represented by simple integer values. This is possible because of the rule in ChocoPy that `None` is not a value of either type, so that there can be no confusion between 0 or false on the one hand, and `None` on the other. We say that these two types are usually *unwrapped* or *unboxed*. Only when assigning them to variables of type `object` is it necessary to "wrap" or "box" them into the object representations described in Section 4.1 so that their actual types can be recovered by functions that expect to receive pointers to objects. The unwrapped values are the same as those that would be stored in the `__int__` or `__bool__` attributes of the object forms. This unwrapped representation considerably speeds up the execution of code that manipulates integer and boolean values.

Register Allocation

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

Overview

Interference graphs

- construction during liveness analysis

Graph Coloring

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

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

Overview

Interference graphs

- construction during liveness analysis

Graph Coloring

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

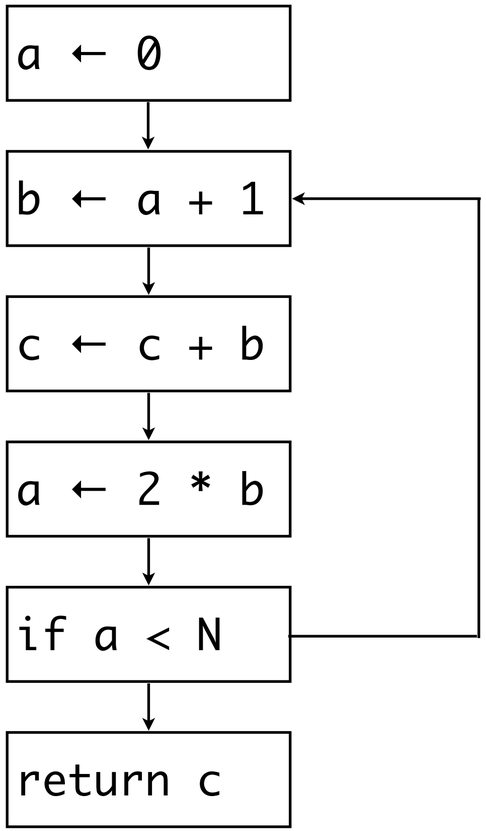
Coalescing

- handle move instructions

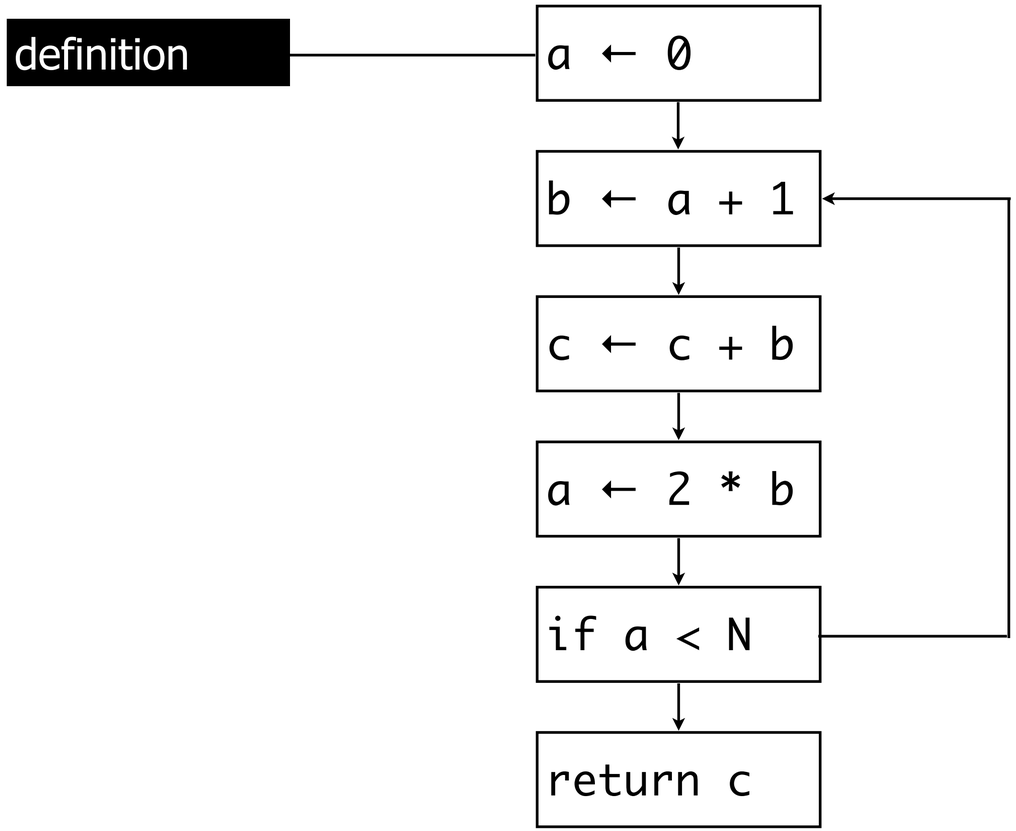
Pre-colored nodes

Interference Graphs

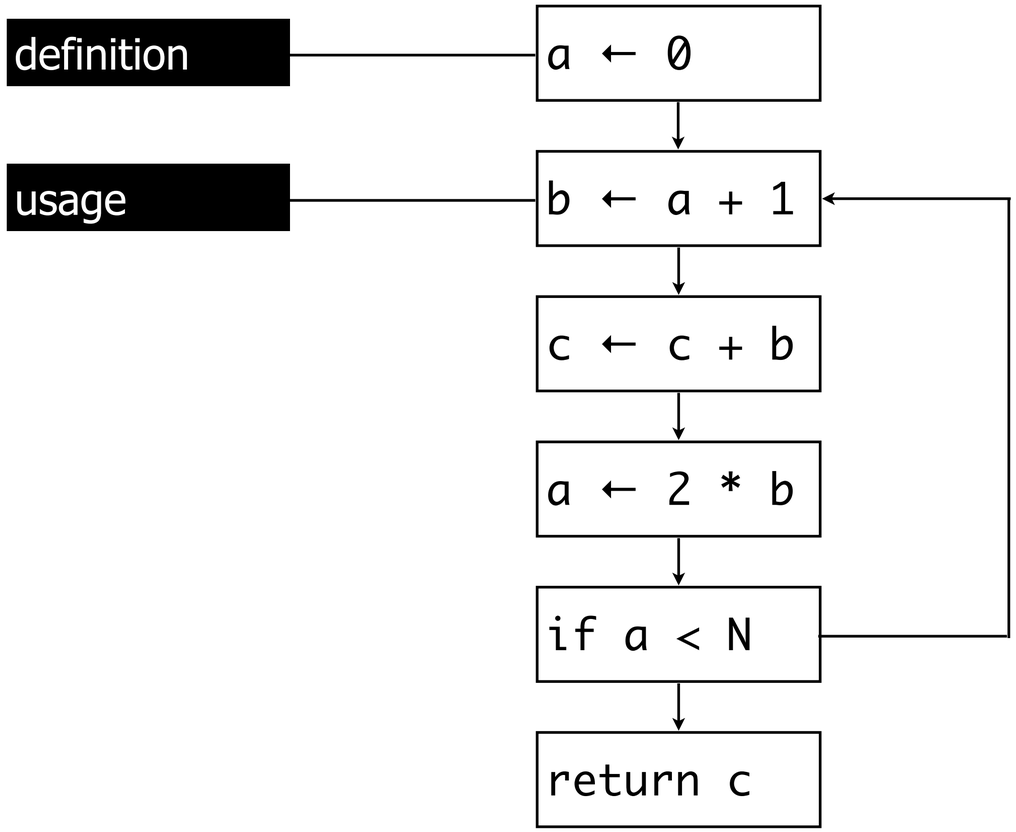
Liveness Analysis



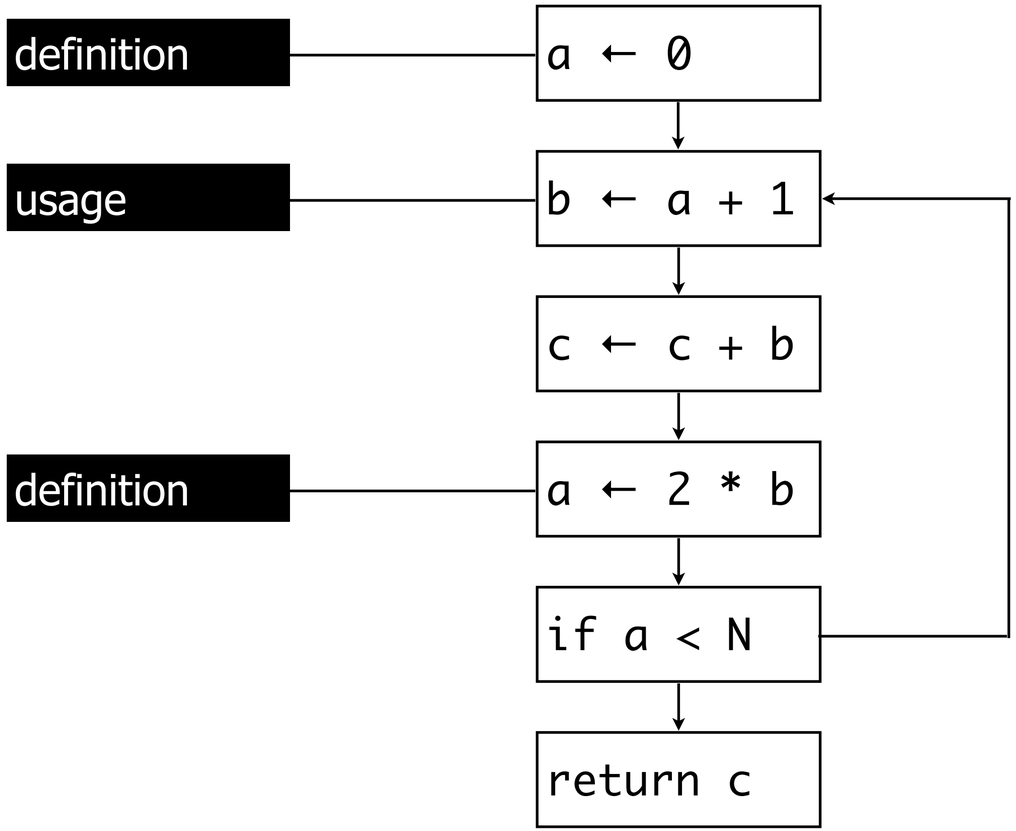
Liveness Analysis



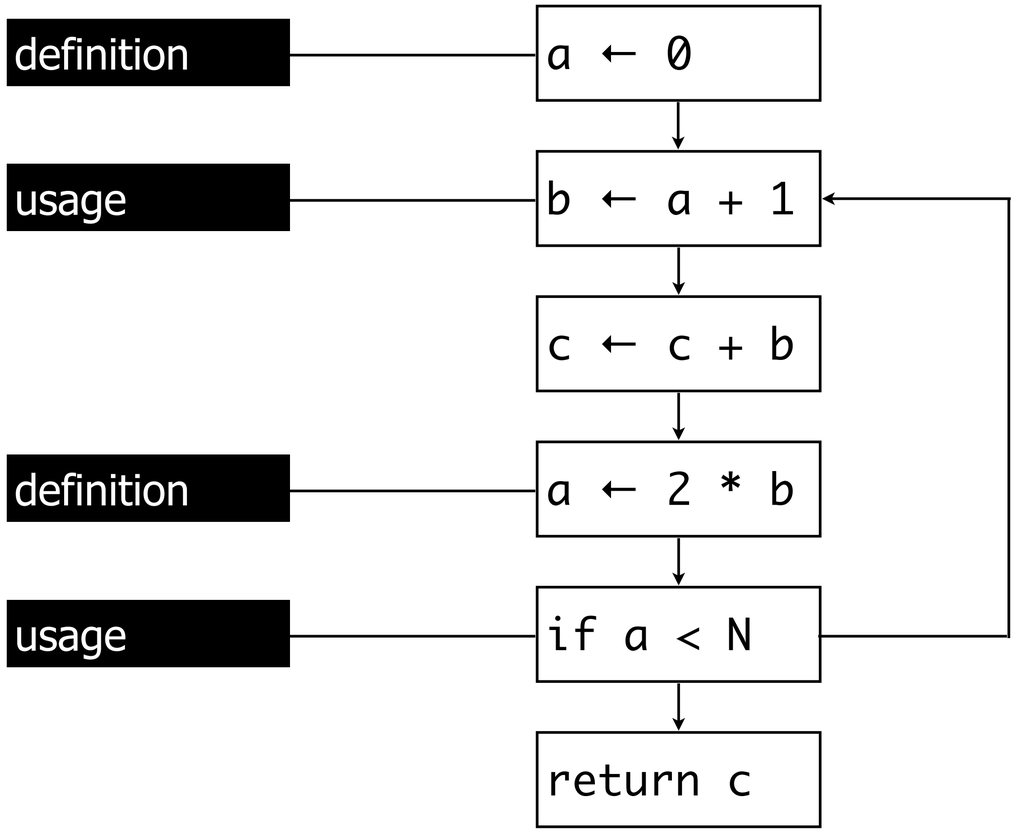
Liveness Analysis



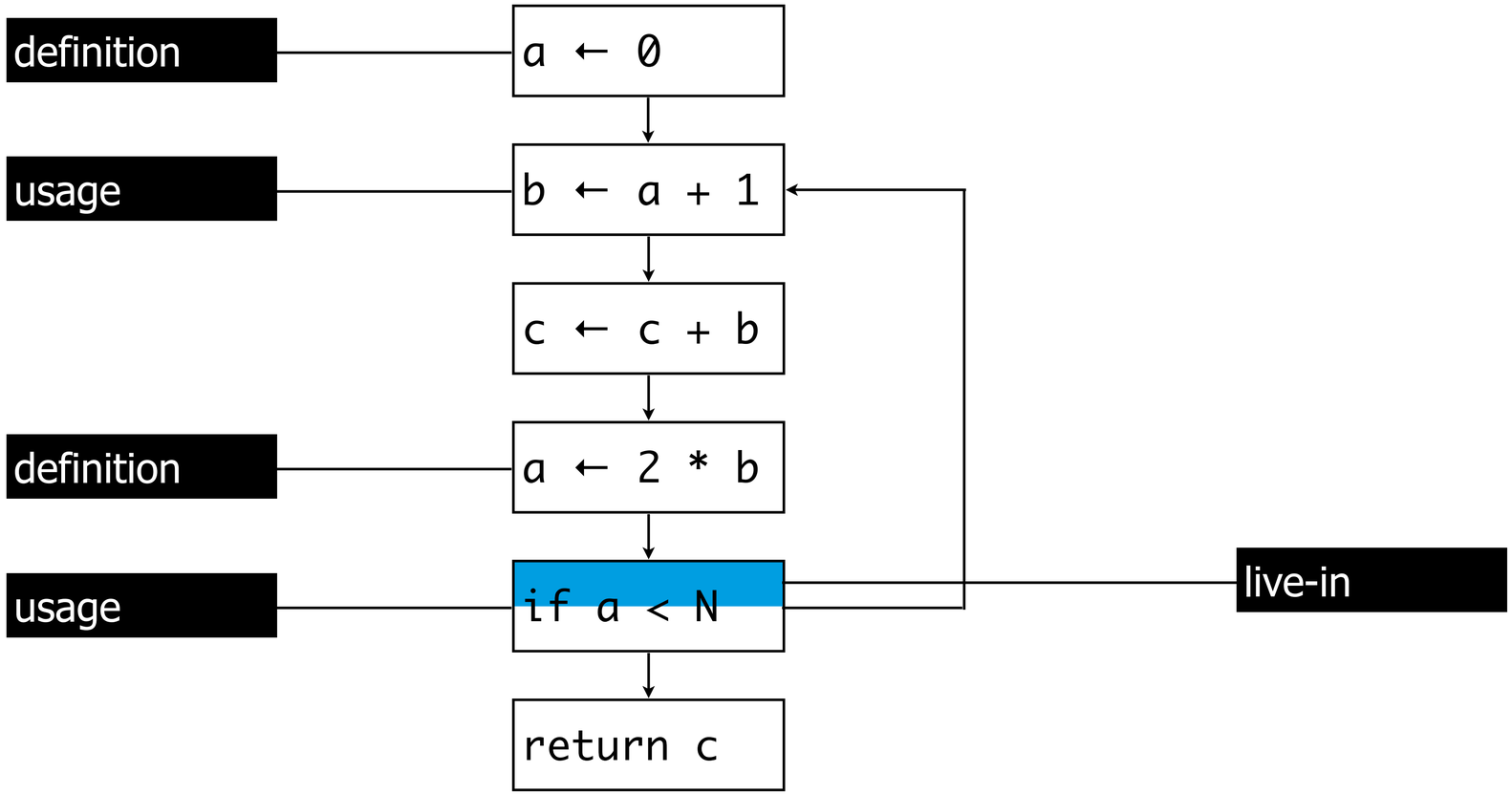
Liveness Analysis



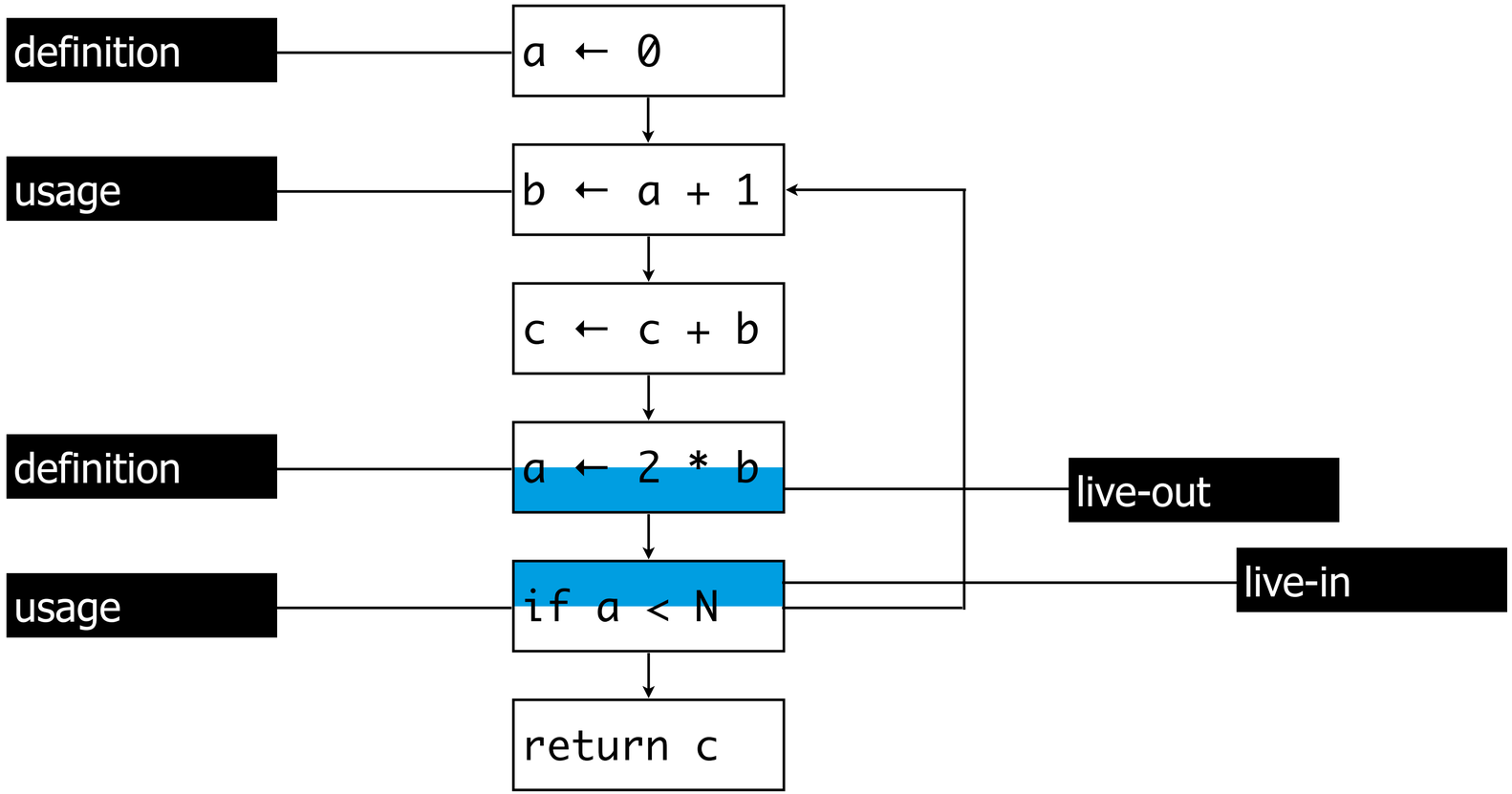
Liveness Analysis



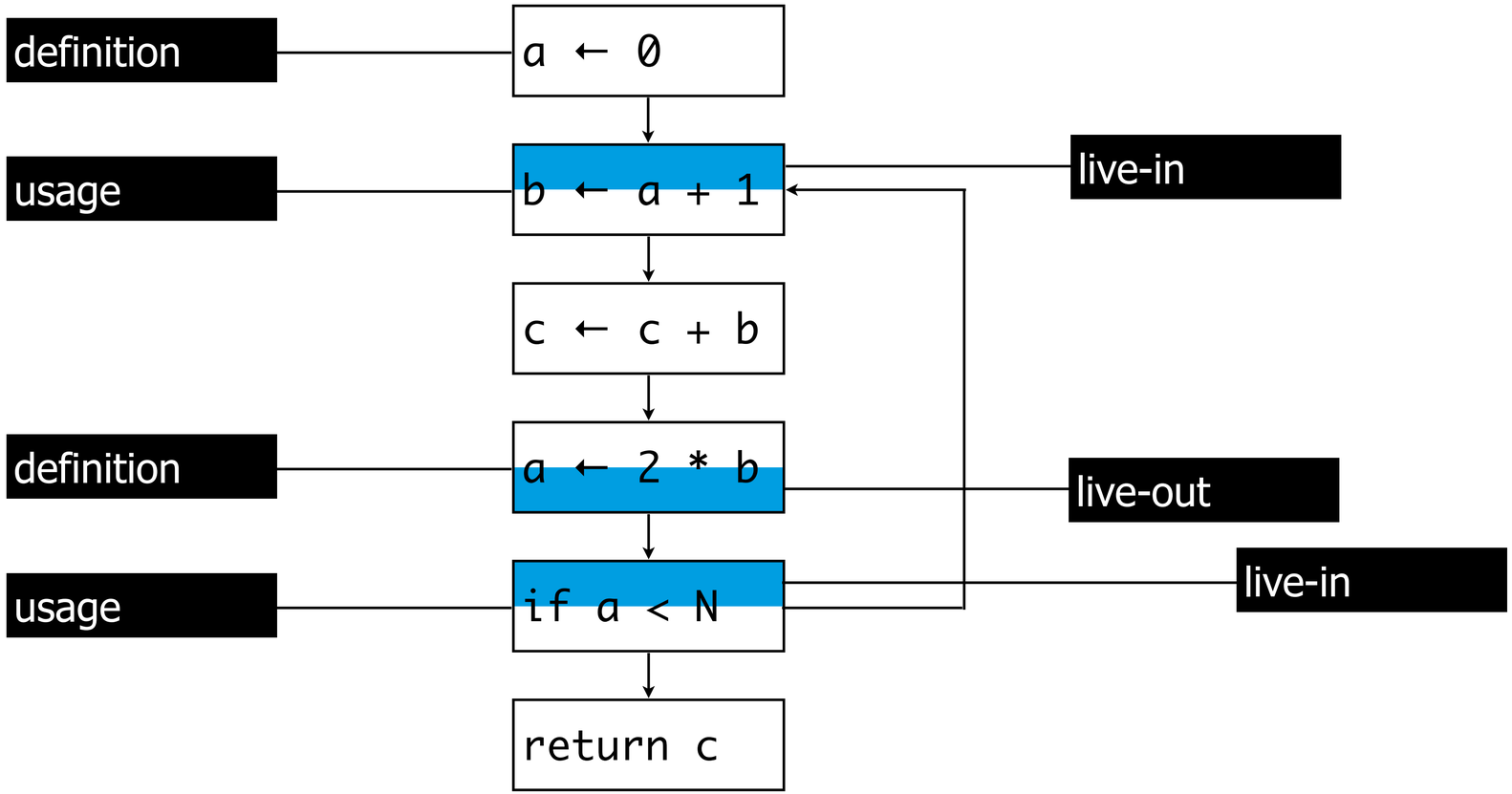
Liveness Analysis



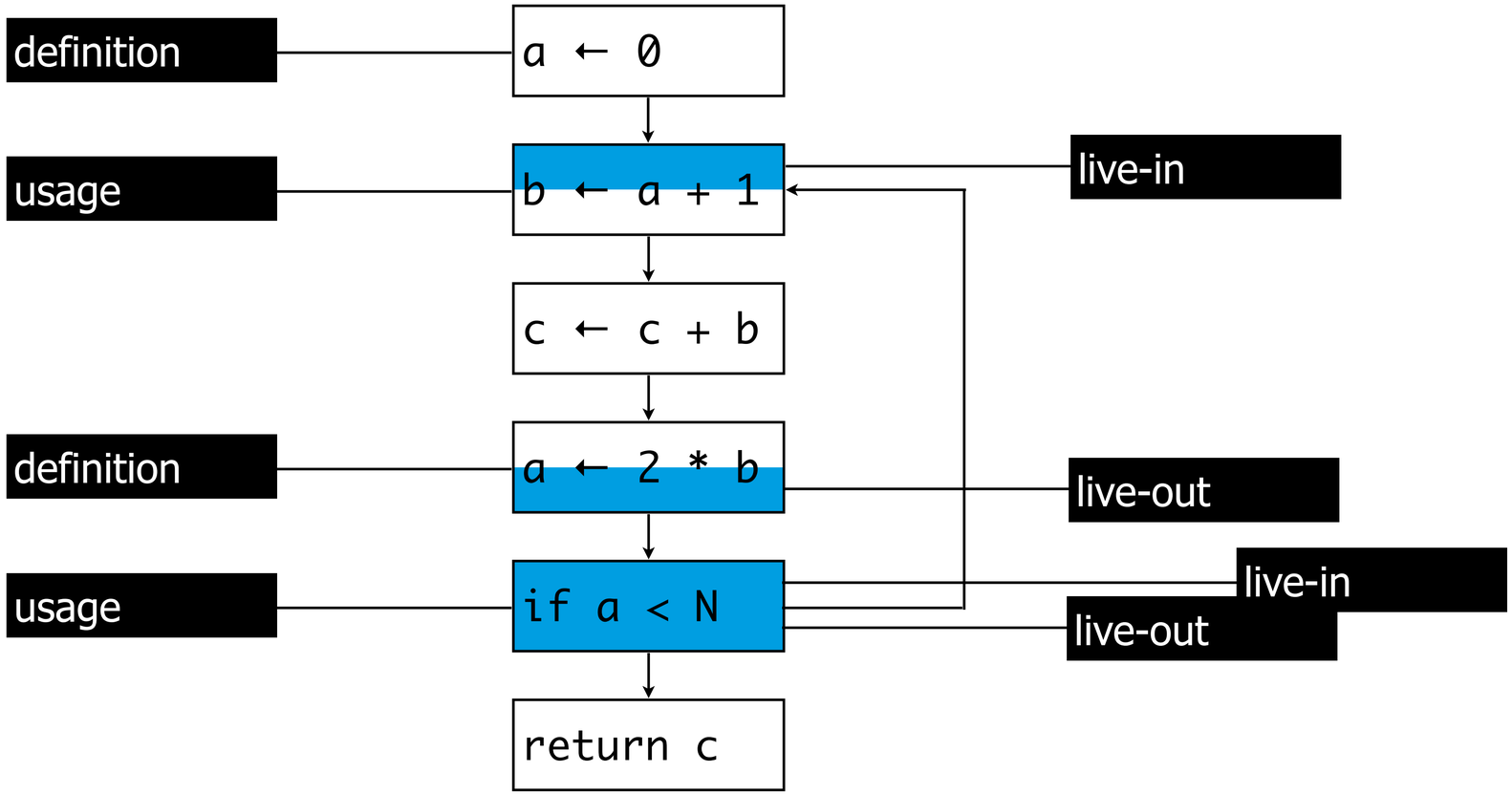
Liveness Analysis



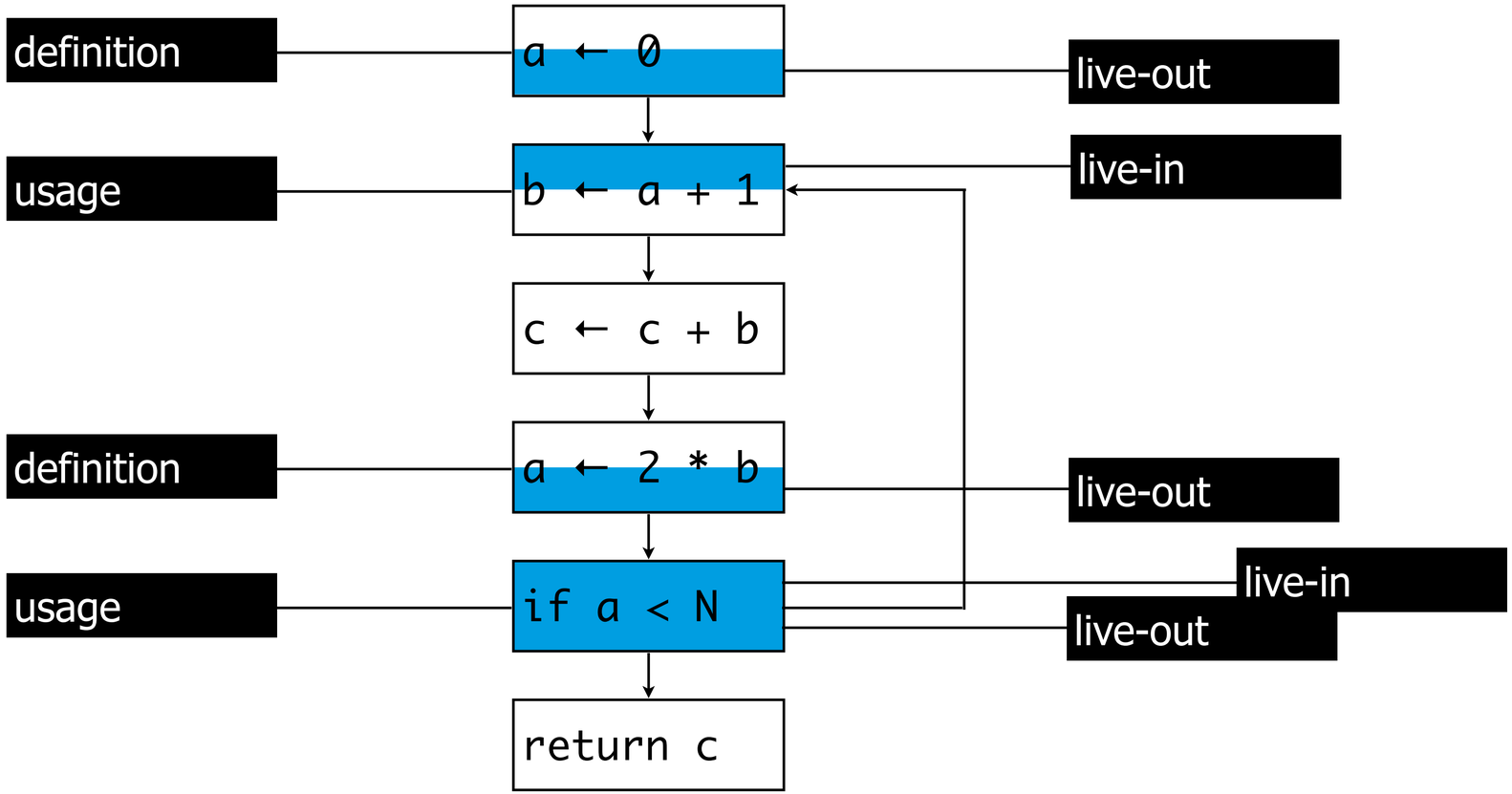
Liveness Analysis



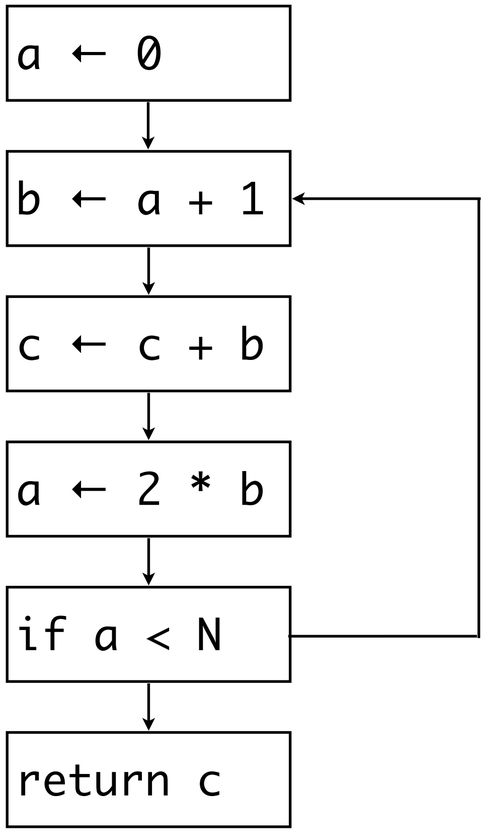
Liveness Analysis



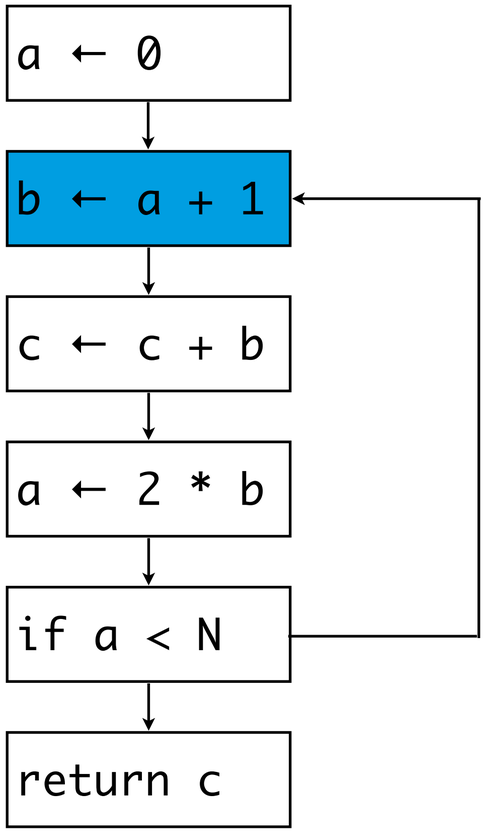
Liveness Analysis



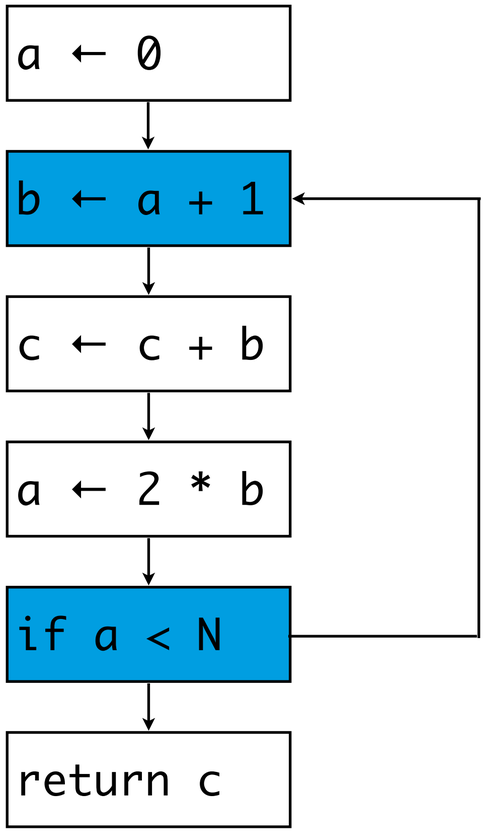
Liveness Analysis



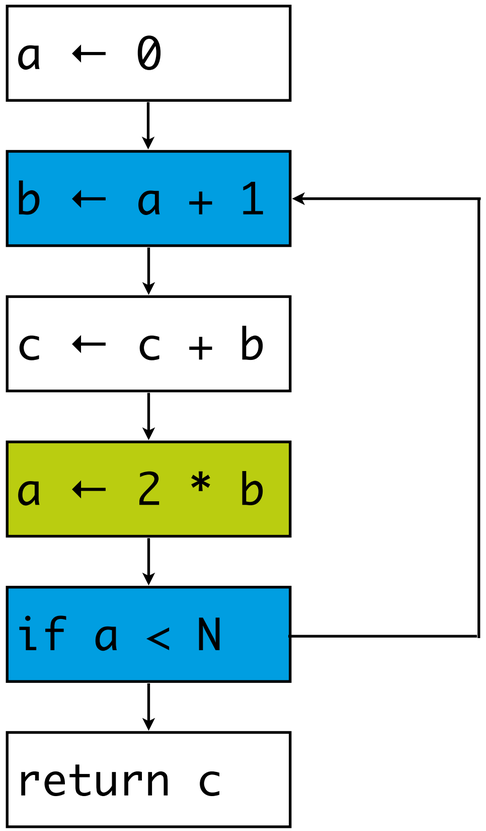
Liveness Analysis



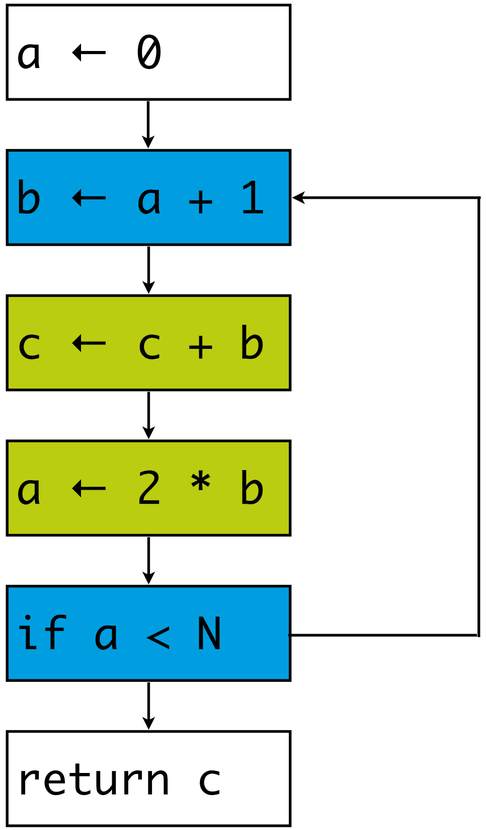
Liveness Analysis



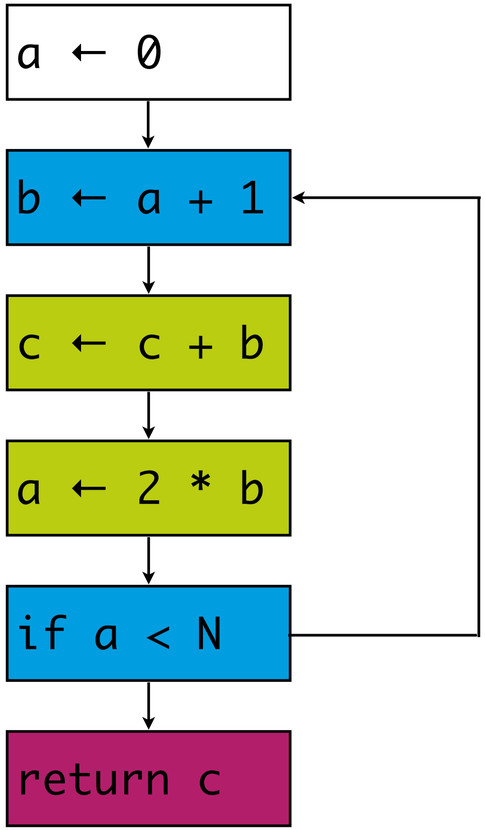
Liveness Analysis



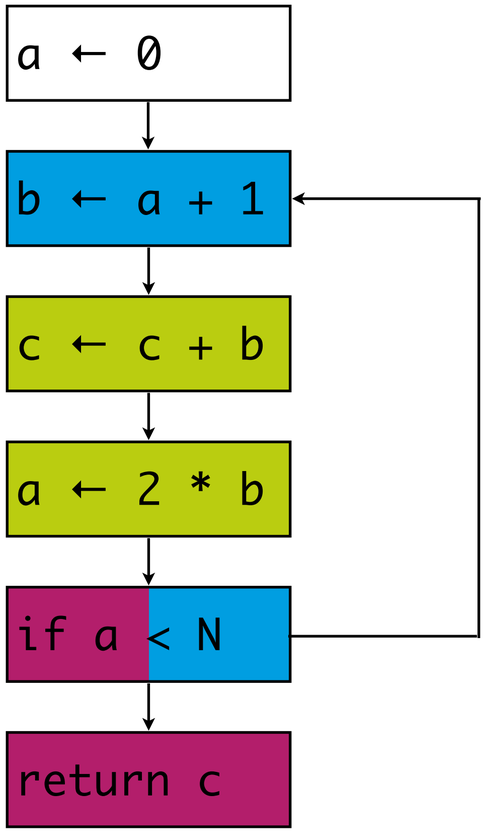
Liveness Analysis



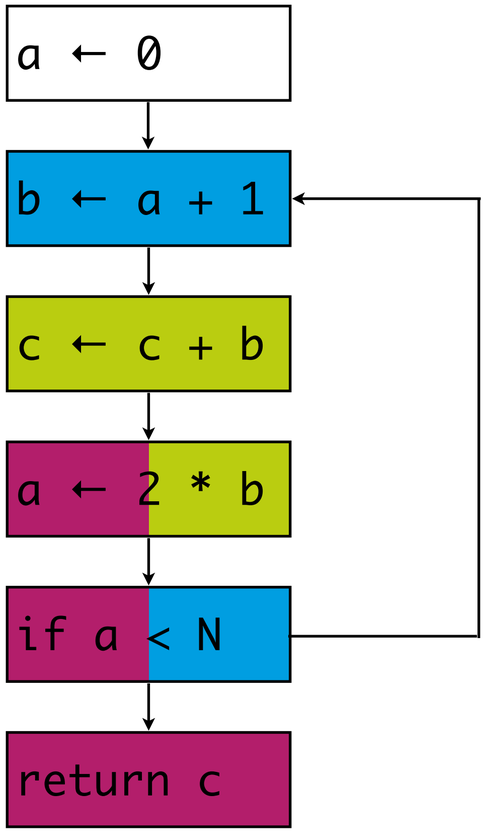
Liveness Analysis



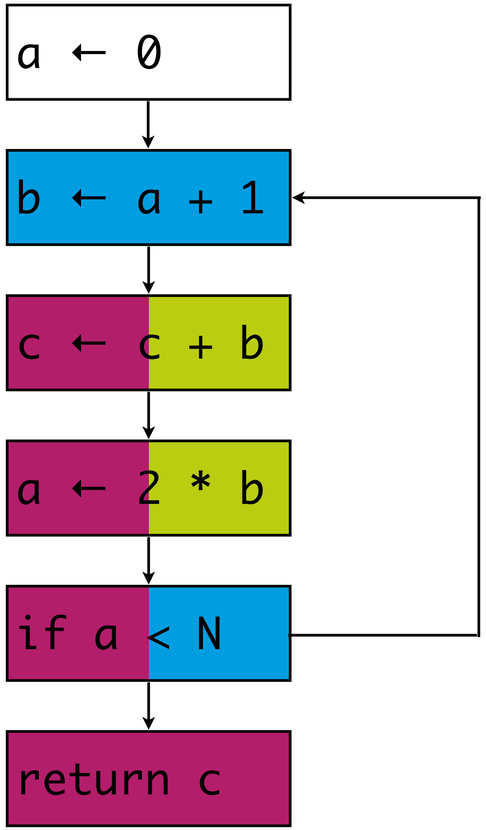
Liveness Analysis



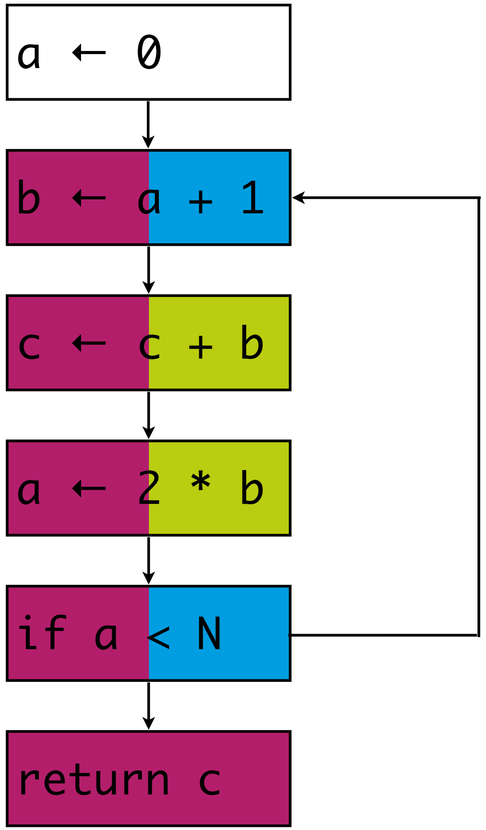
Liveness Analysis



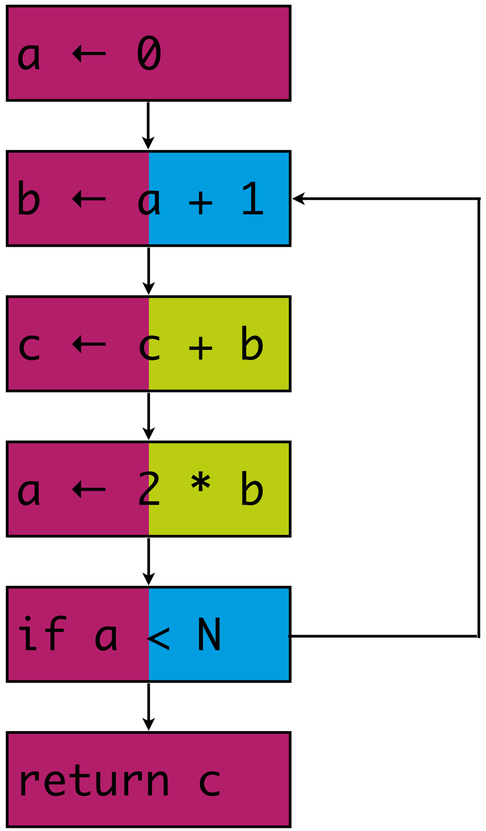
Liveness Analysis



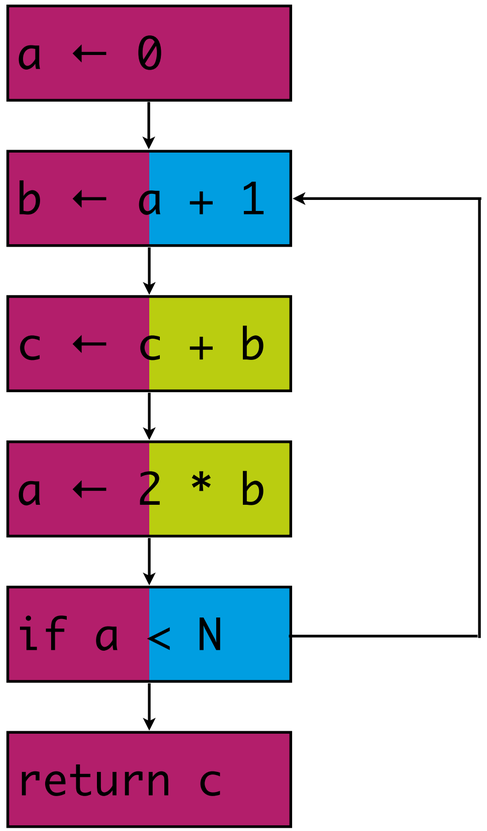
Liveness Analysis



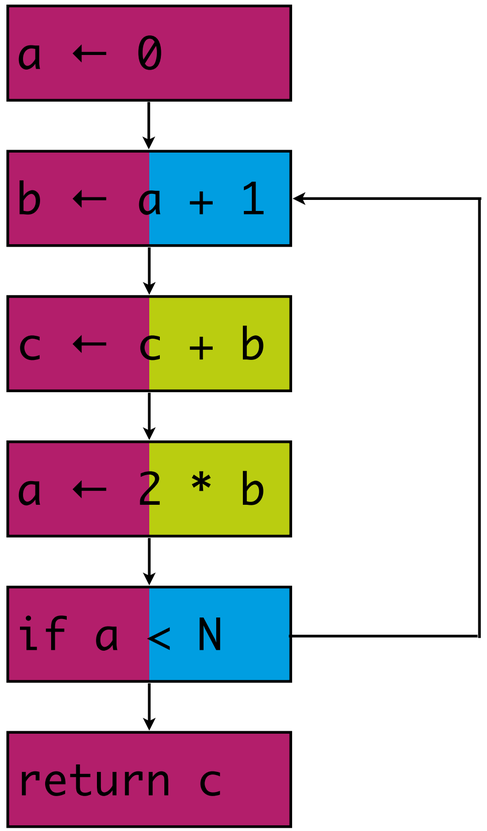
Liveness Analysis

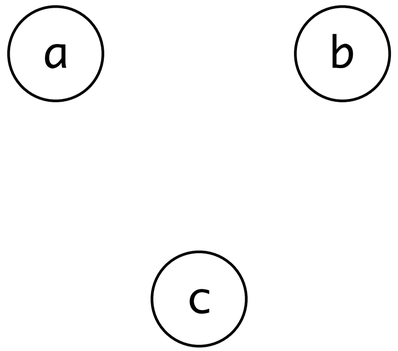


Interference Graphs

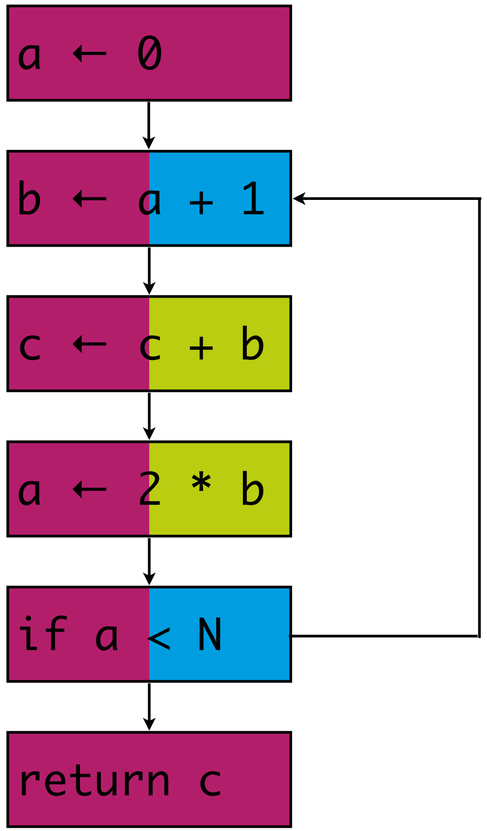


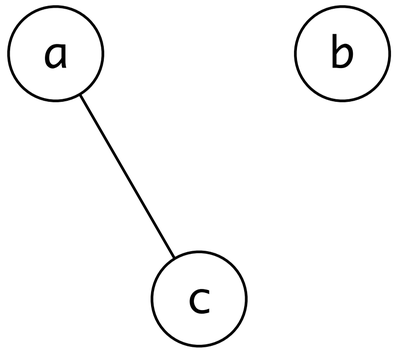
Interference Graphs



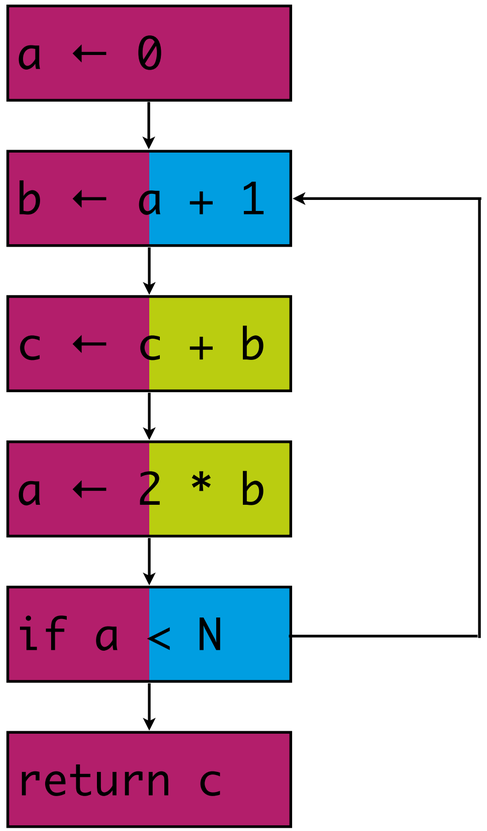


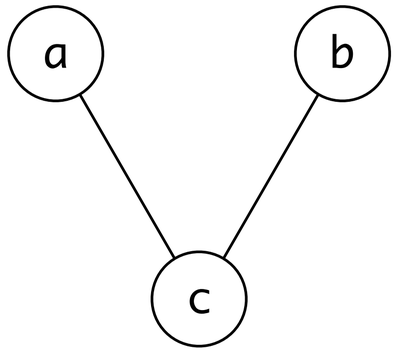
Interference Graphs





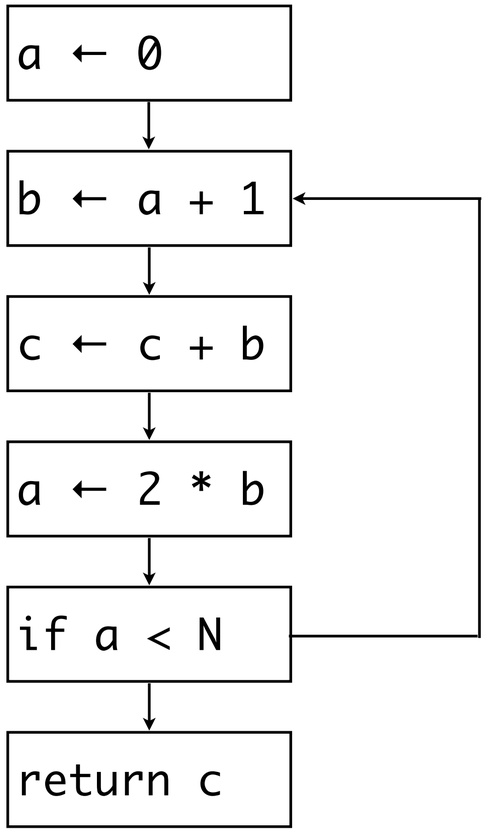
Interference Graphs

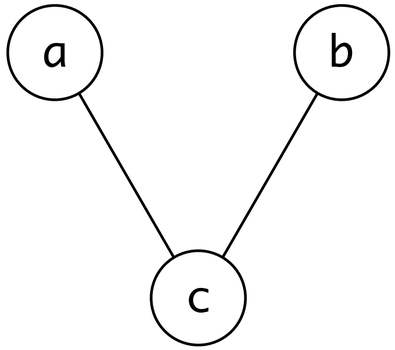




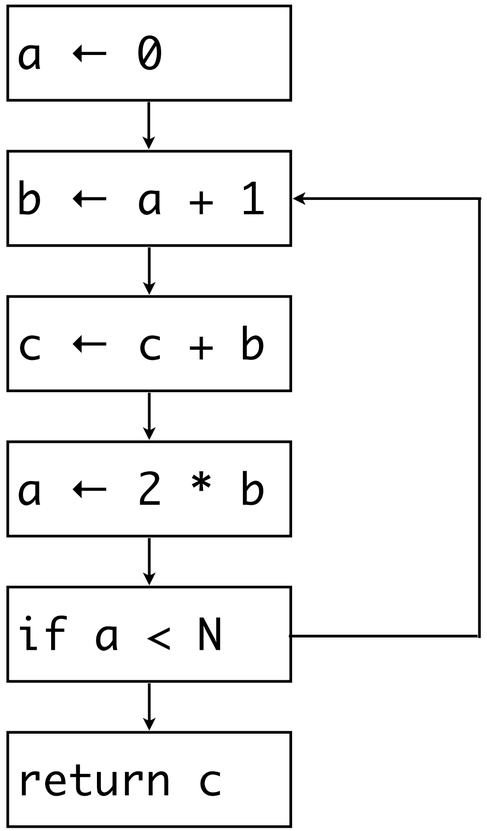
Graph Coloring

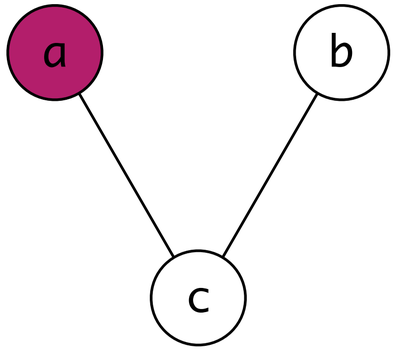
Graph Coloring



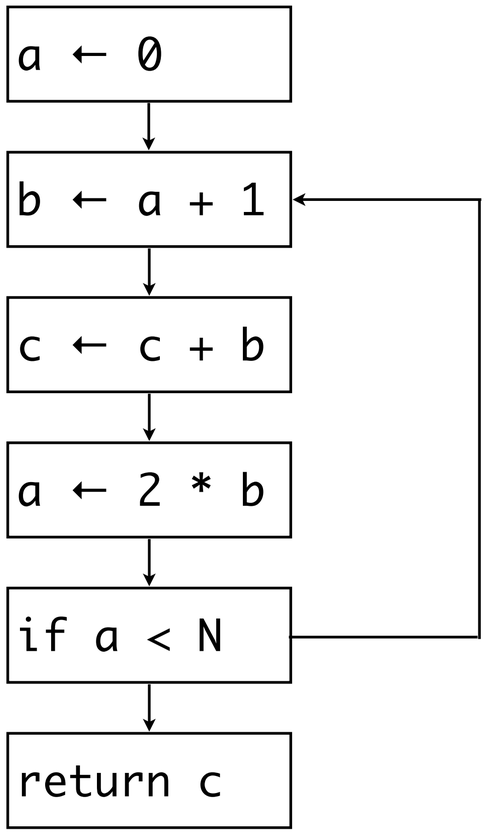


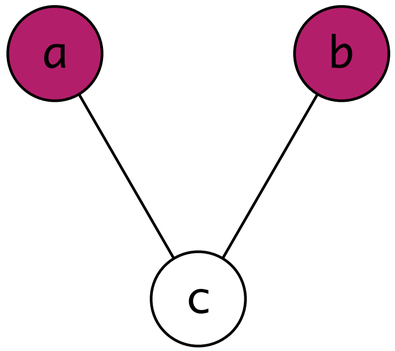
Graph Coloring



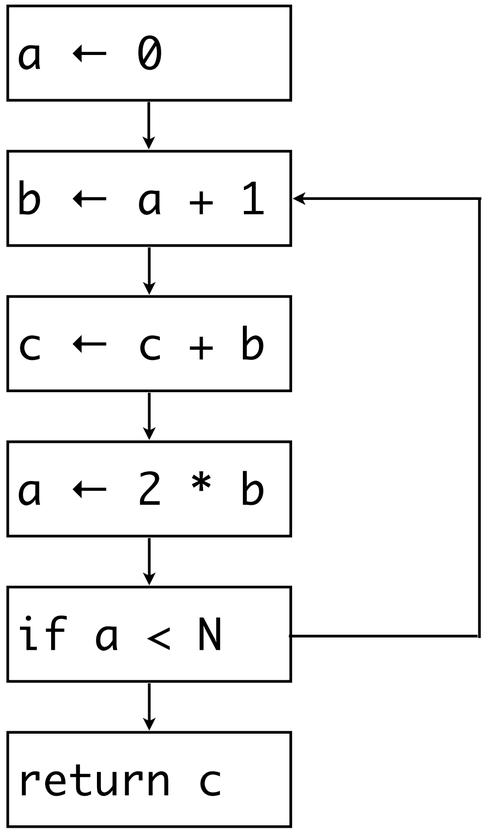


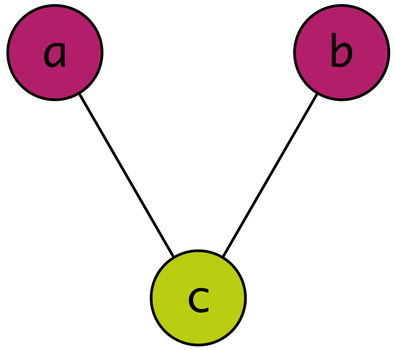
Graph Coloring



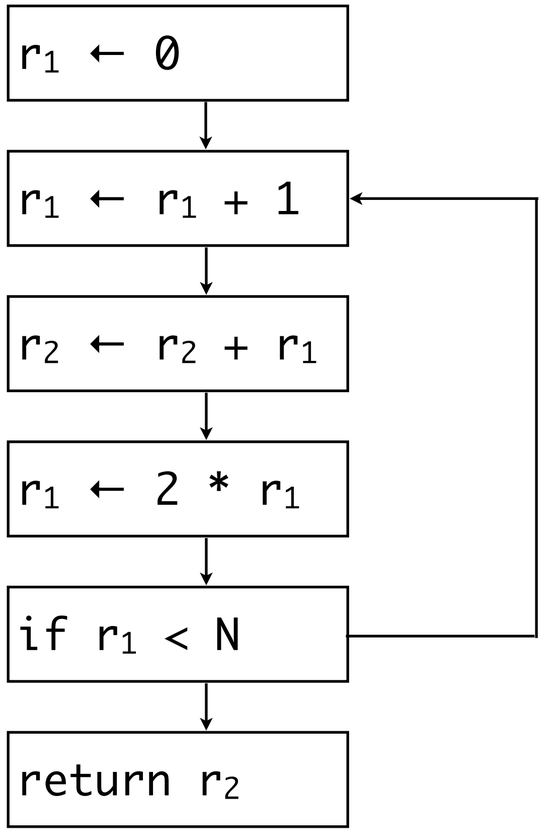


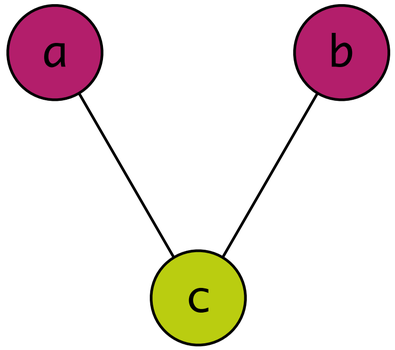
Graph Coloring





Graph Coloring





Graph Coloring: Steps

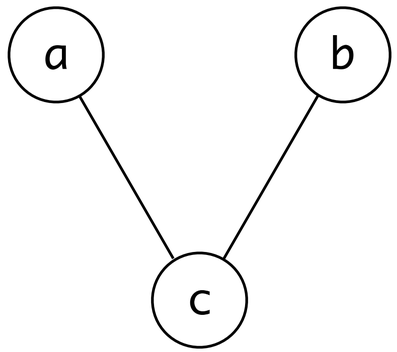
Simplify

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

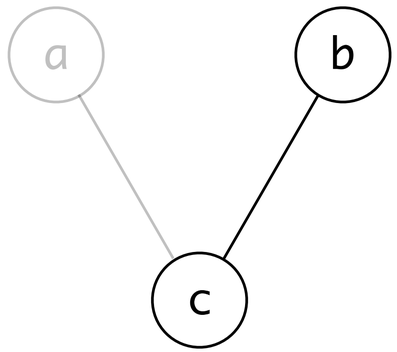
Select

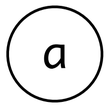
- add node, select color

Graph Coloring: Example with 2 Colors

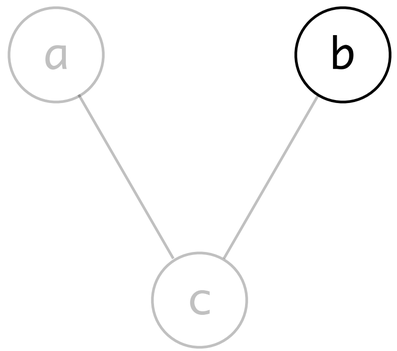


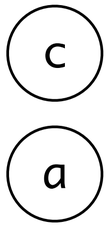
Graph Coloring: Example with 2 Colors



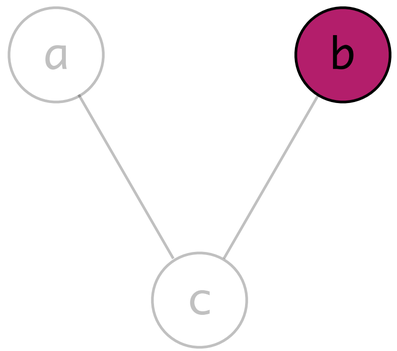


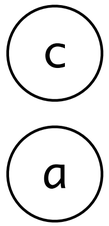
Graph Coloring: Example with 2 Colors



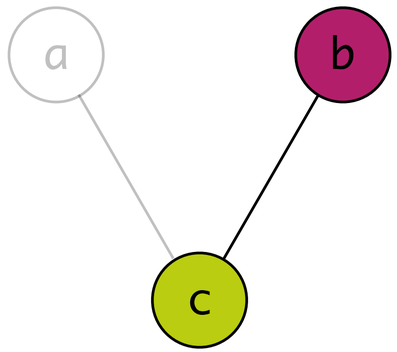


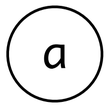
Graph Coloring: Example with 2 Colors



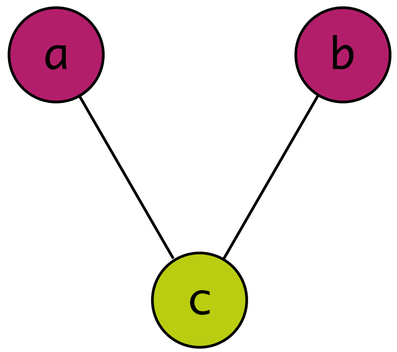


Graph Coloring: Example with 2 Colors





Graph Coloring: Example with 2 Colors

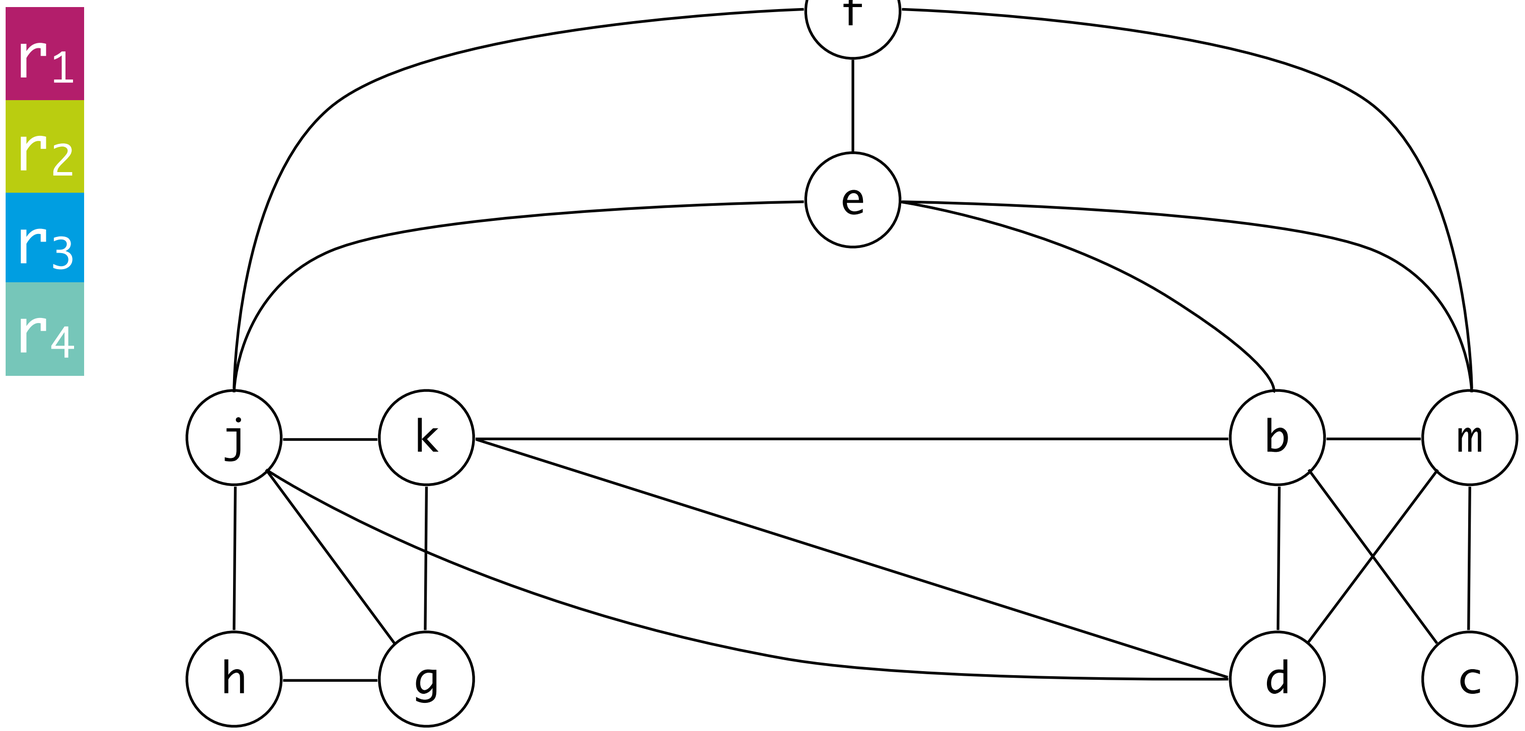


Graph Coloring

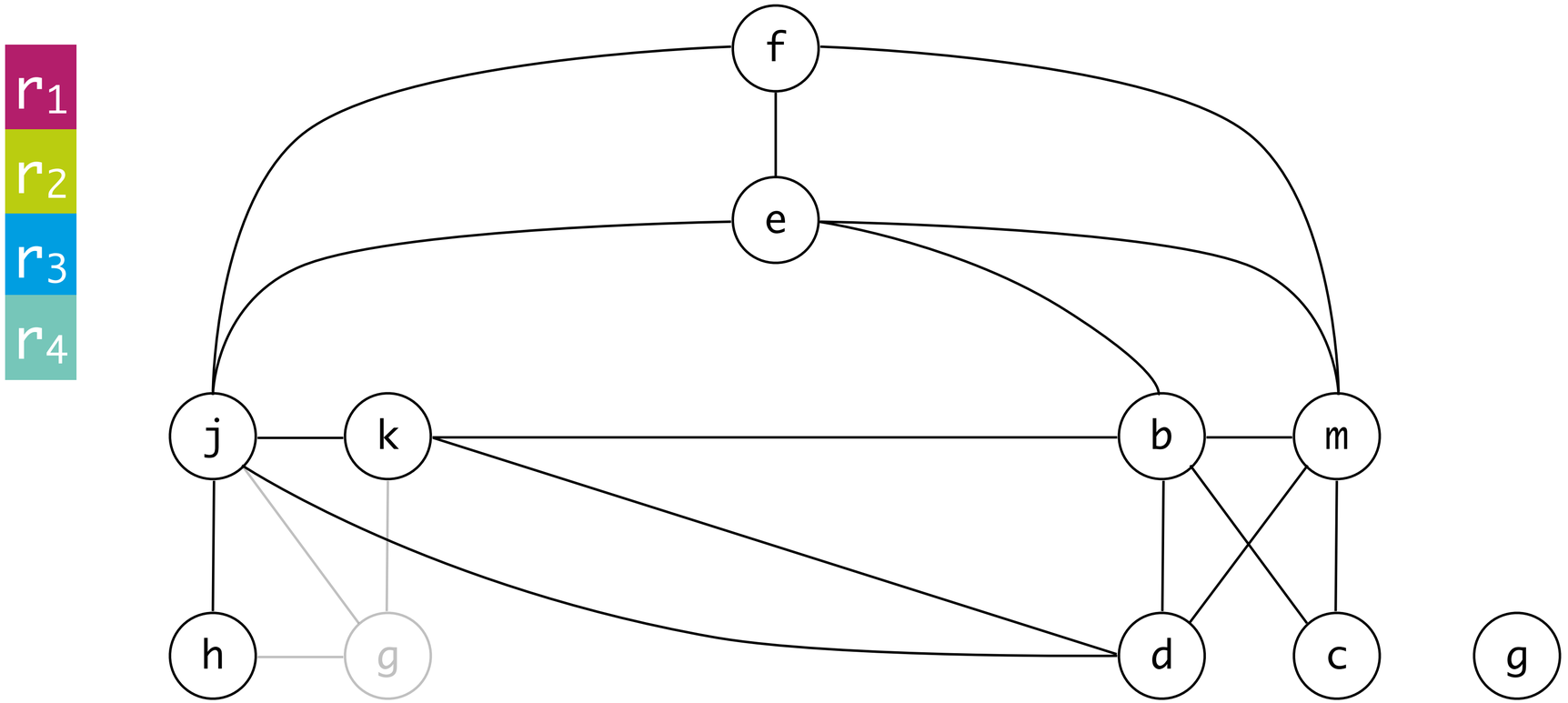
```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Graph Coloring: Example with 4 Colors

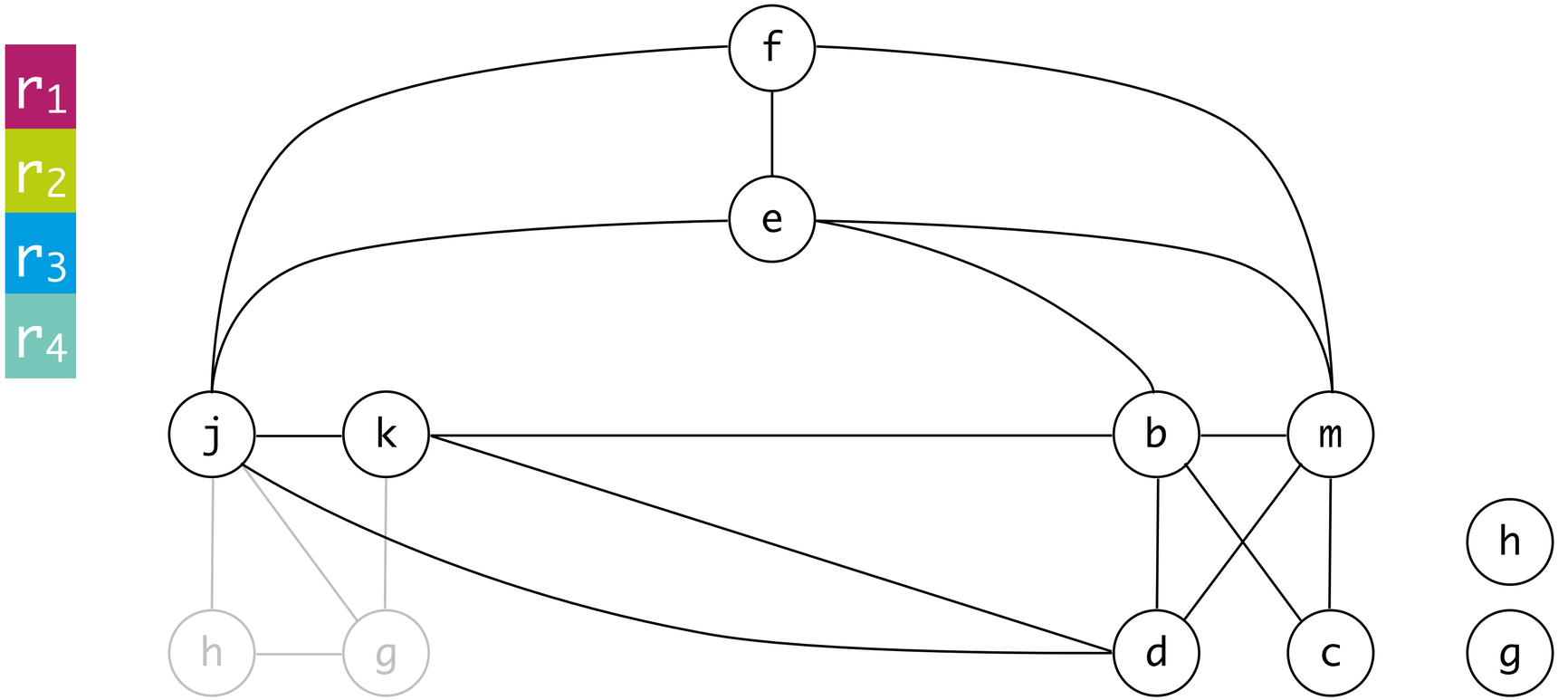
```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```



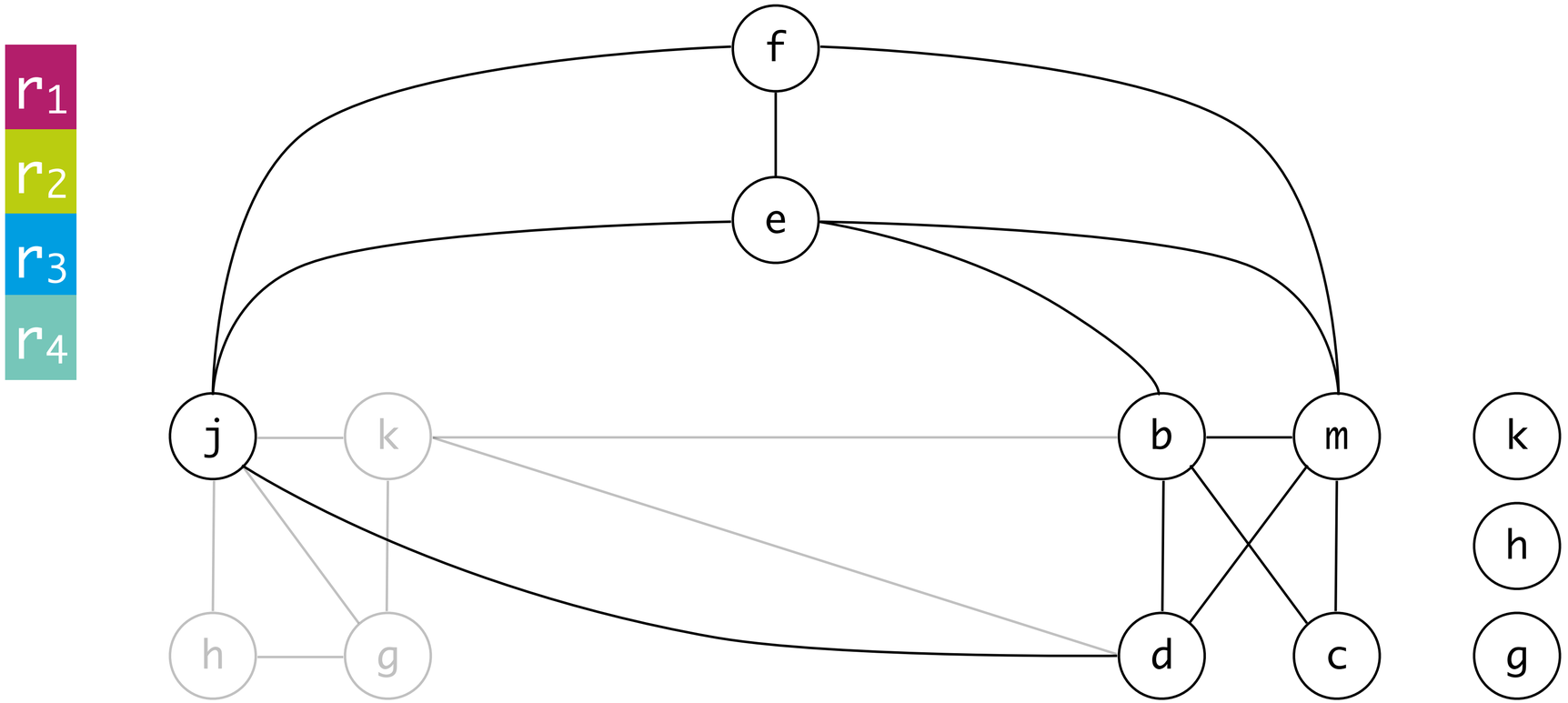
Graph Coloring: Example with 4 Colors



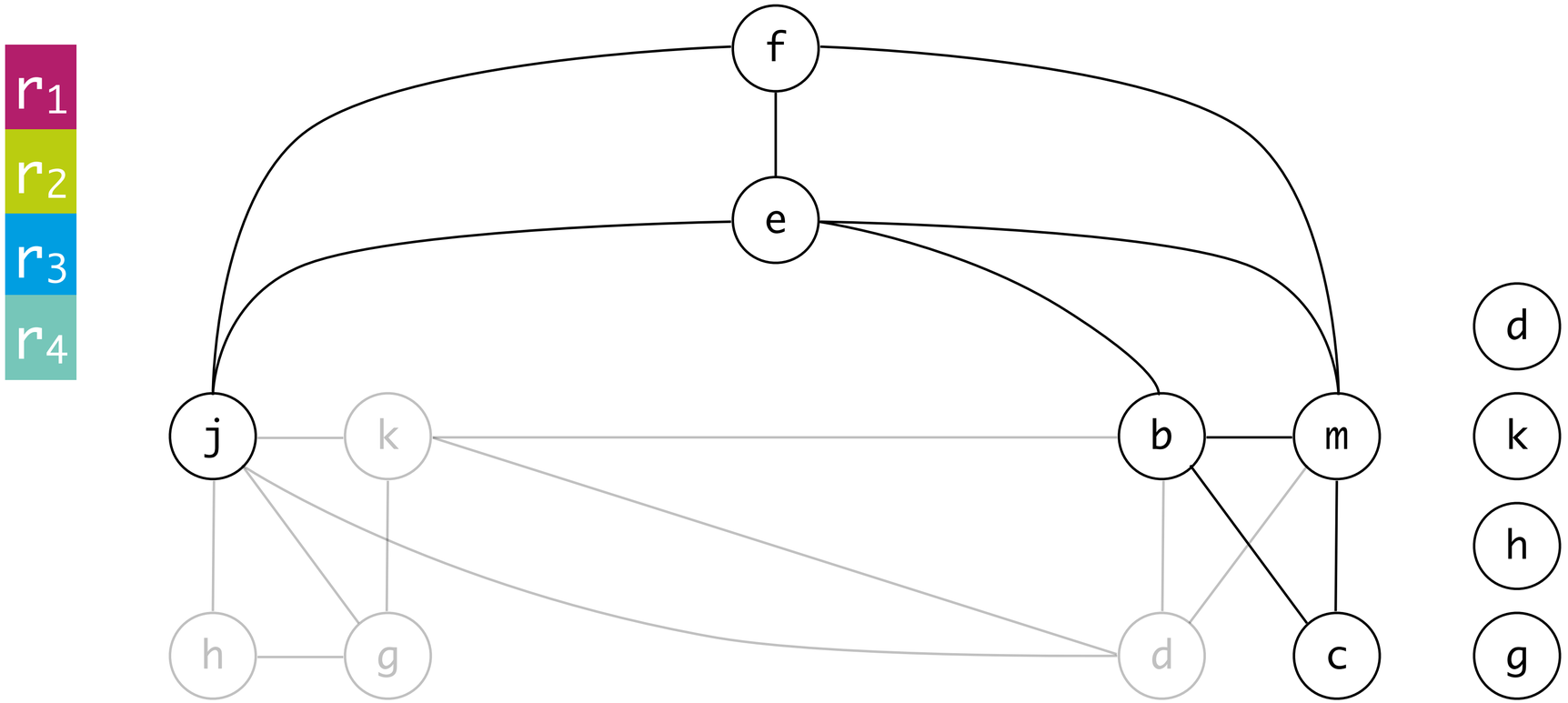
Graph Coloring: Example with 4 Colors



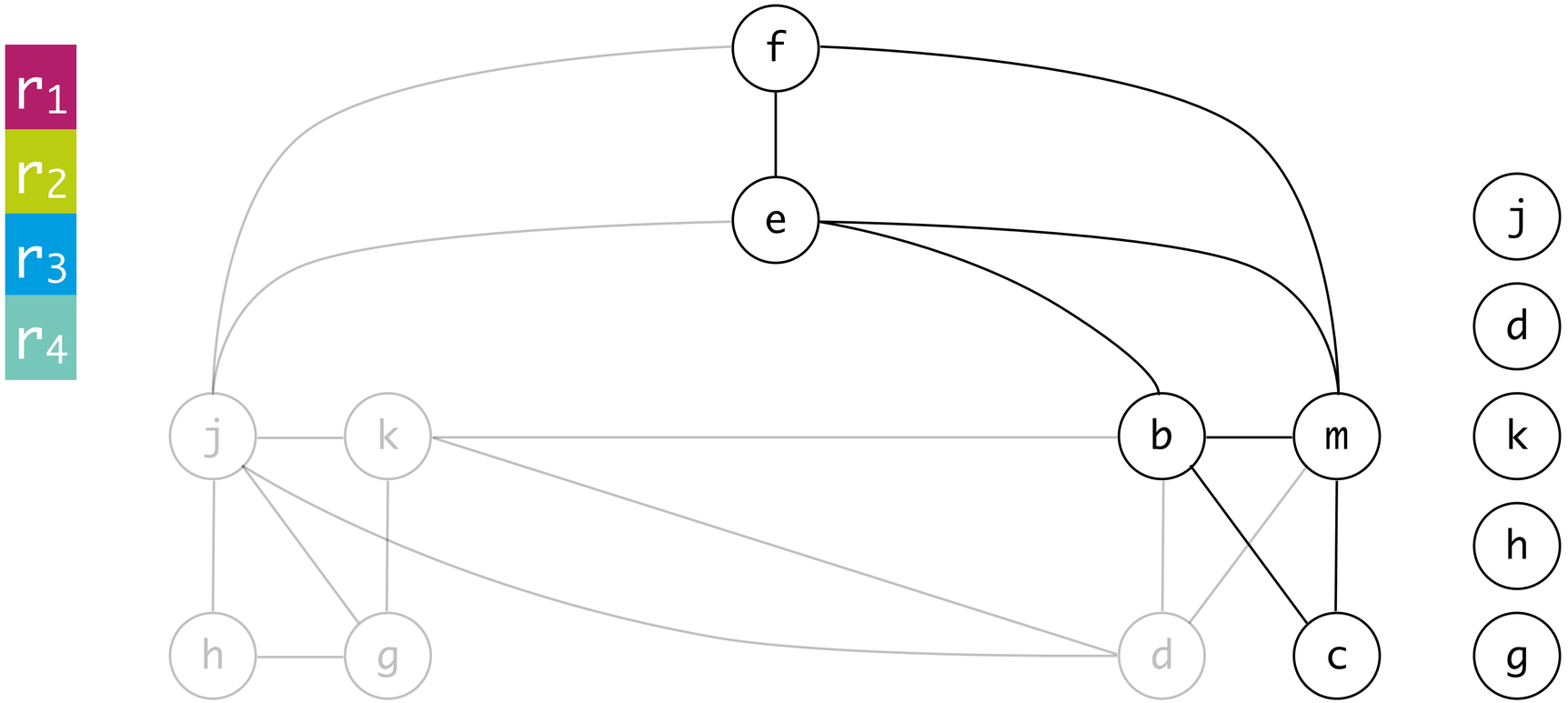
Graph Coloring: Example with 4 Colors



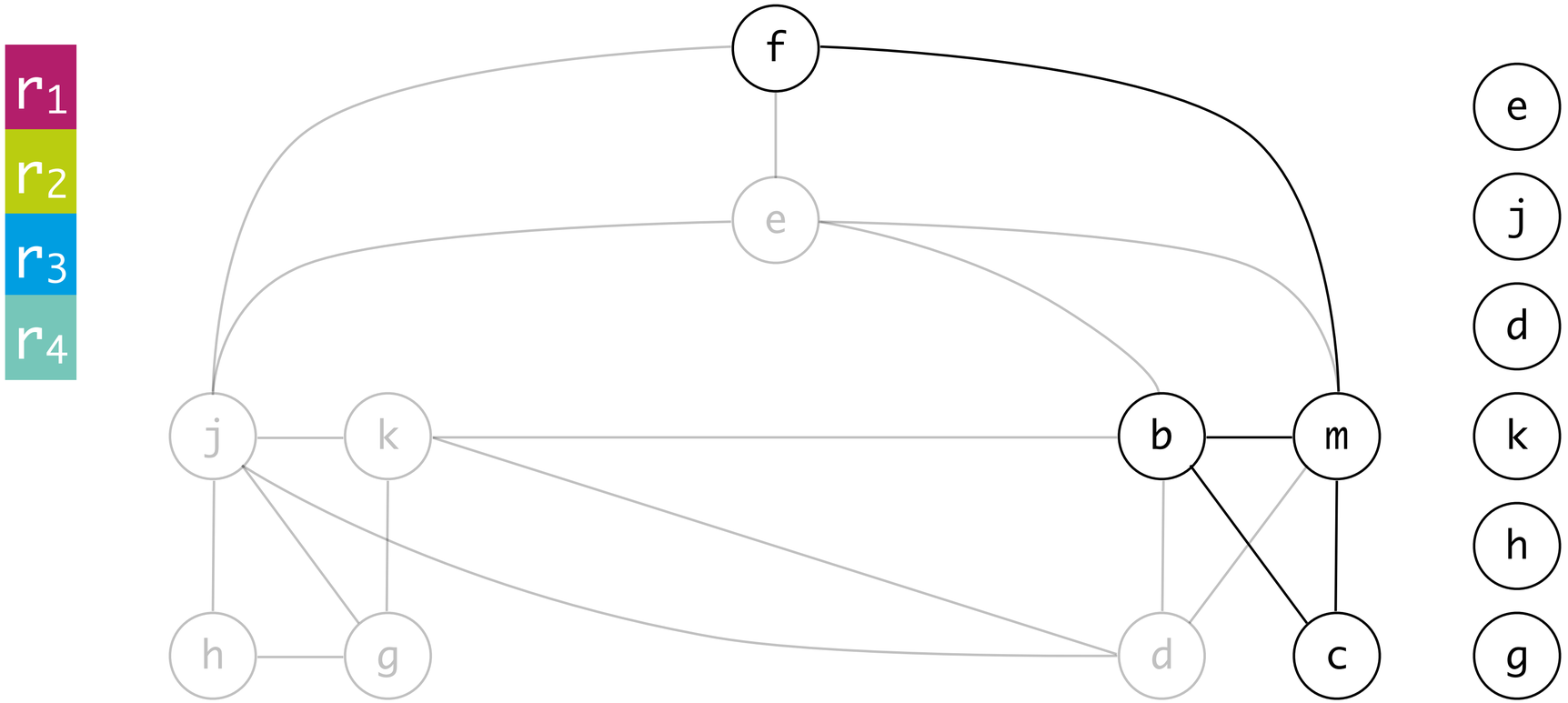
Graph Coloring: Example with 4 Colors



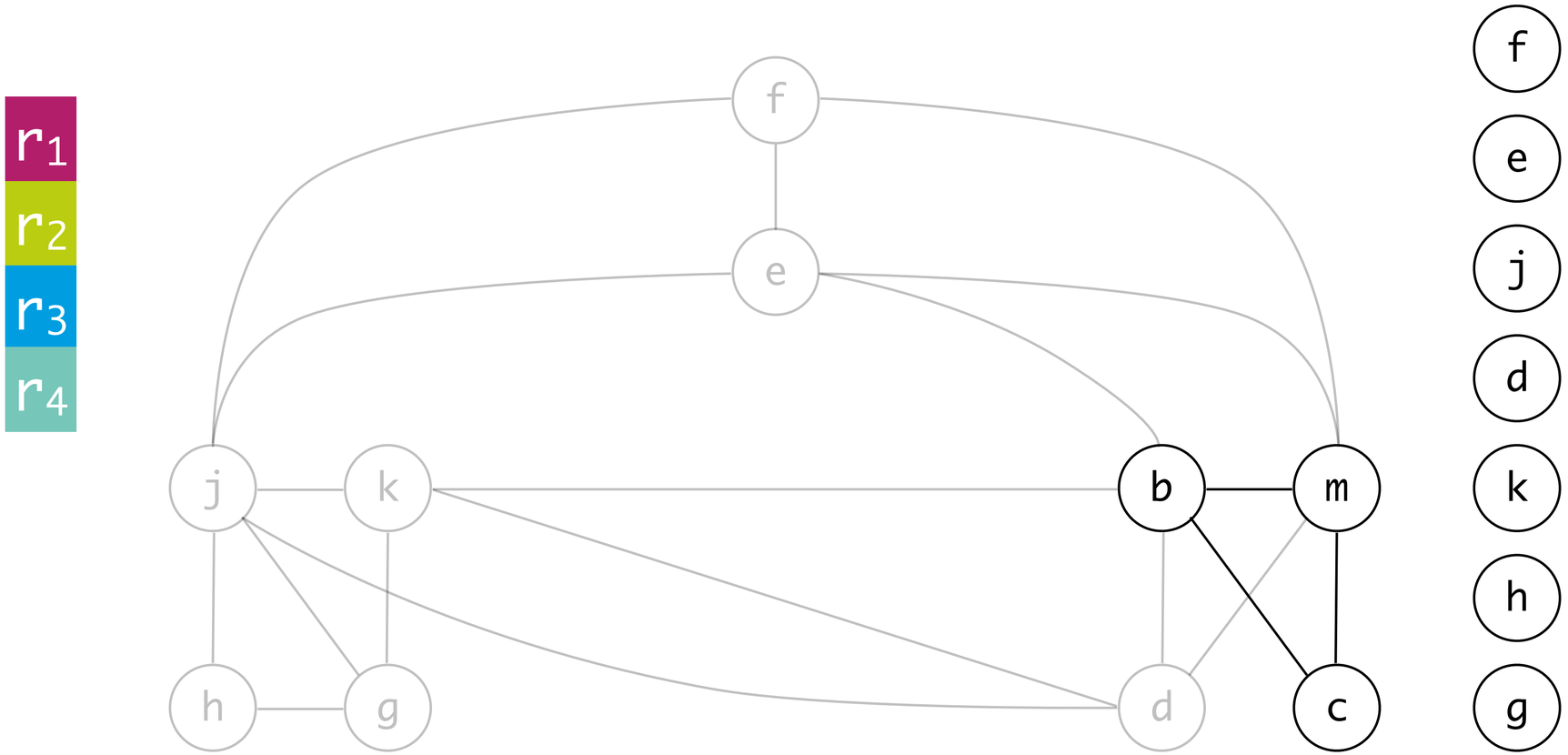
Graph Coloring: Example with 4 Colors



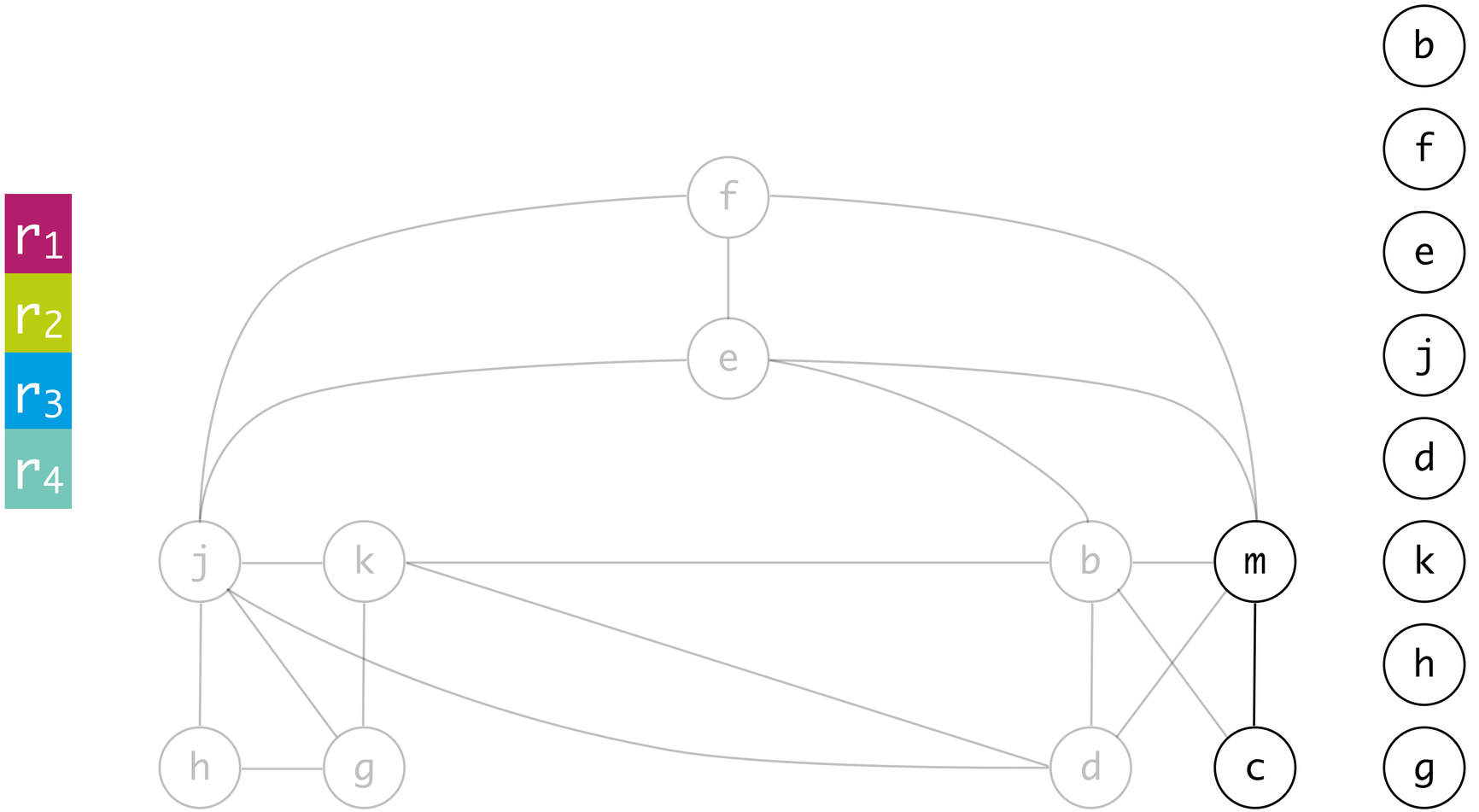
Graph Coloring: Example with 4 Colors



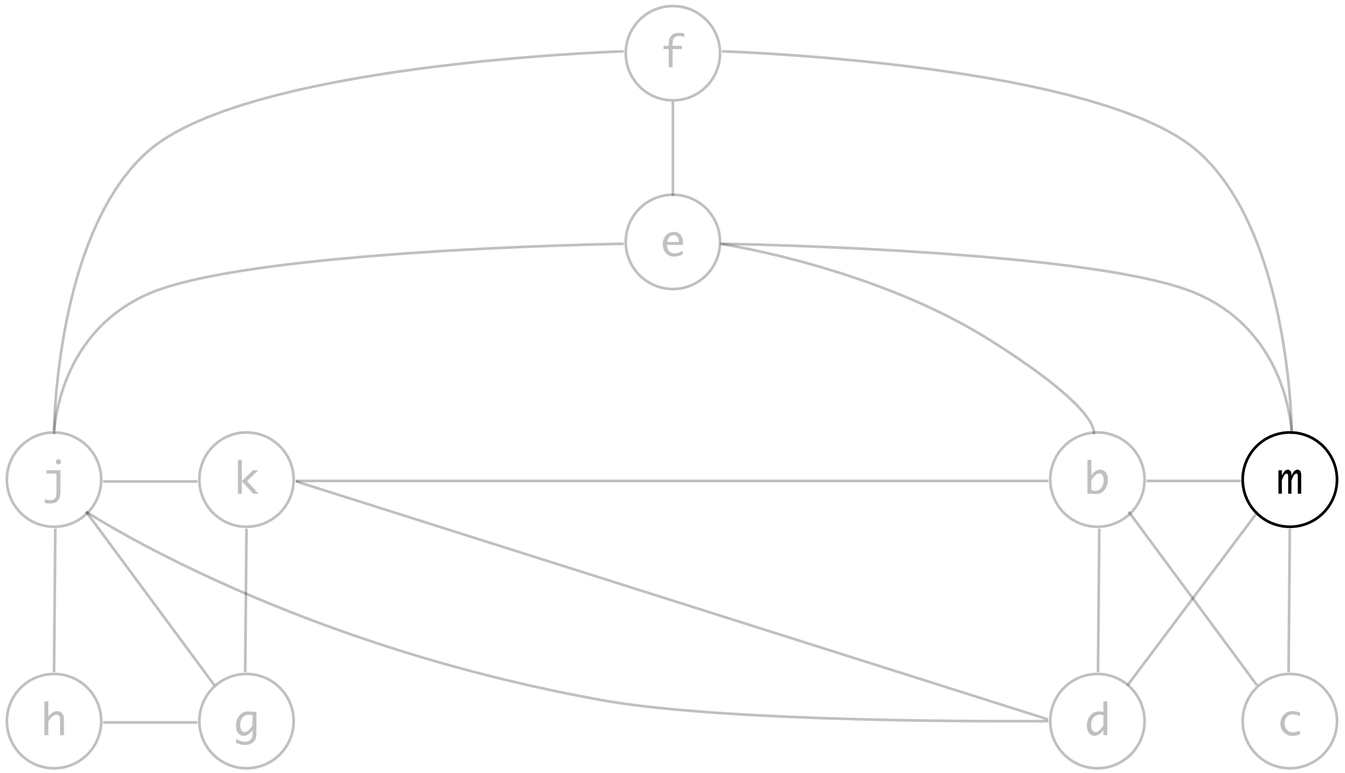
Graph Coloring: Example with 4 Colors

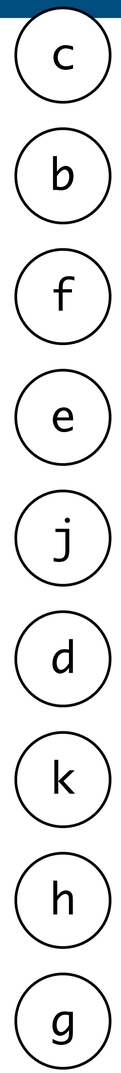


Graph Coloring: Example with 4 Colors



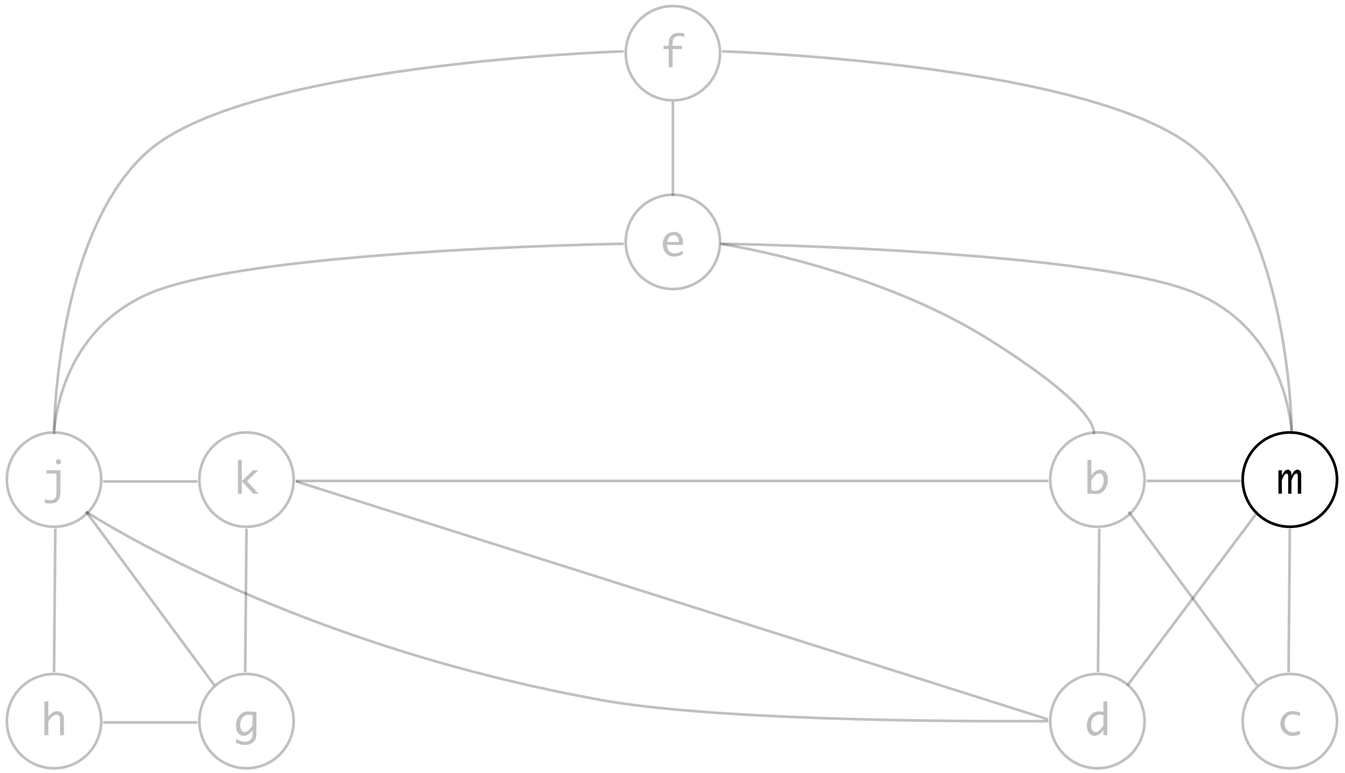
Graph Coloring: Example with 4 Colors






```
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
```

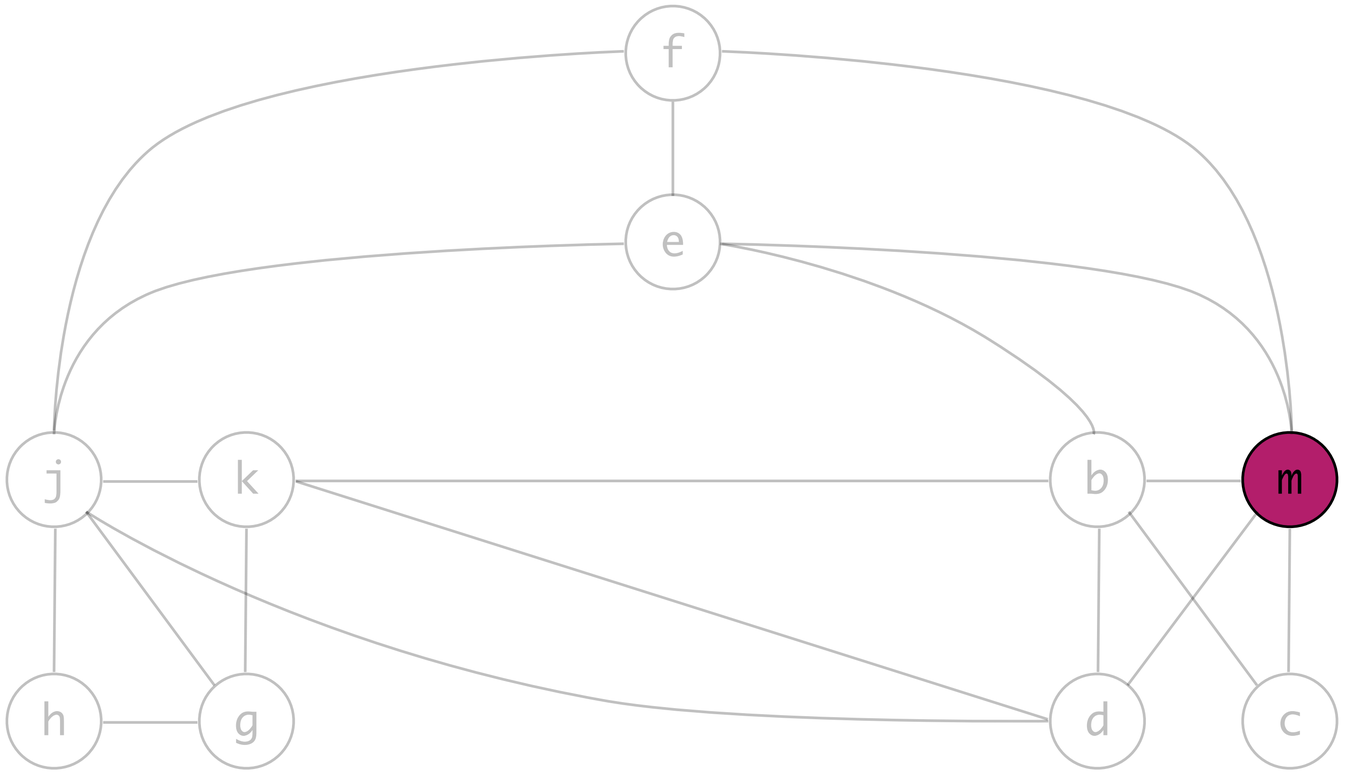
Graph Coloring: Example with 4 Colors



c
b
f
e
j
d
k
h
g

```
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
```

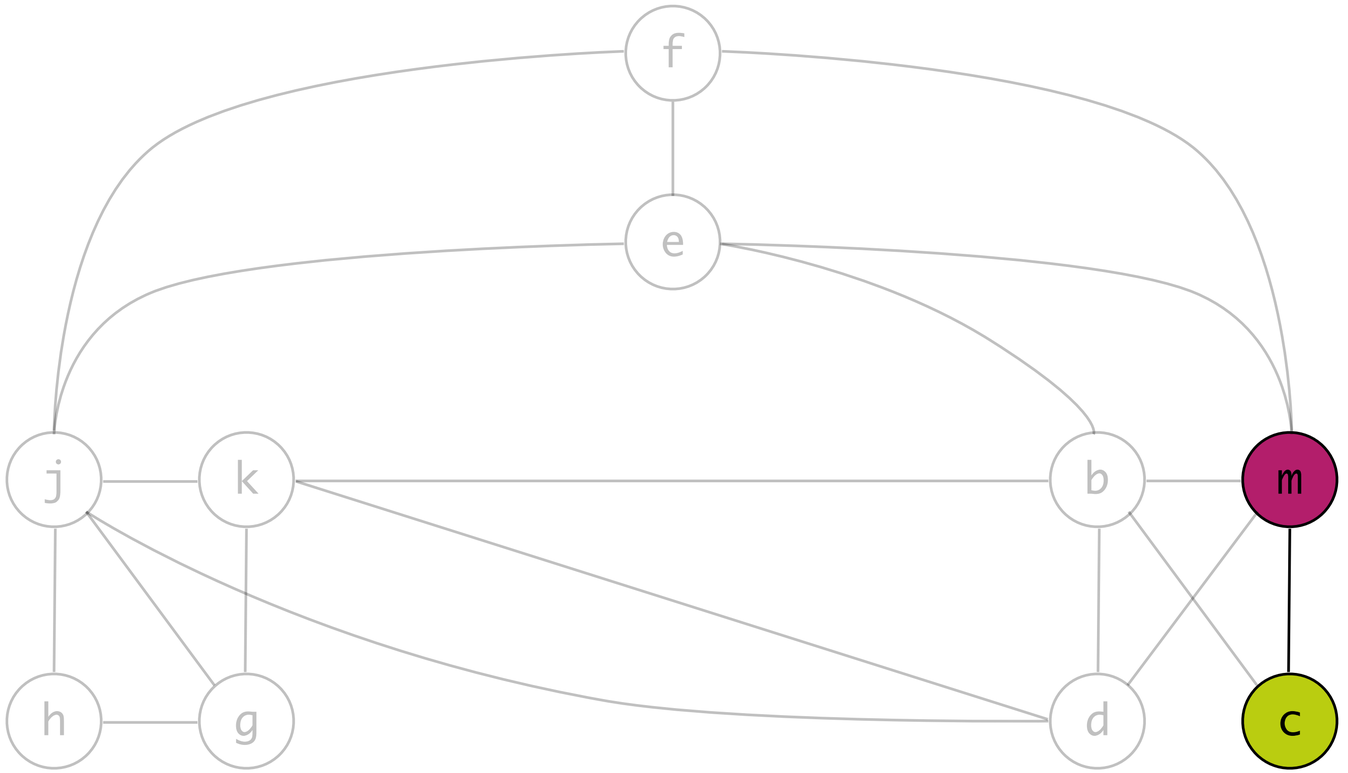
Graph Coloring



c
b
f
e
j
d
k
h
g

```
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
```

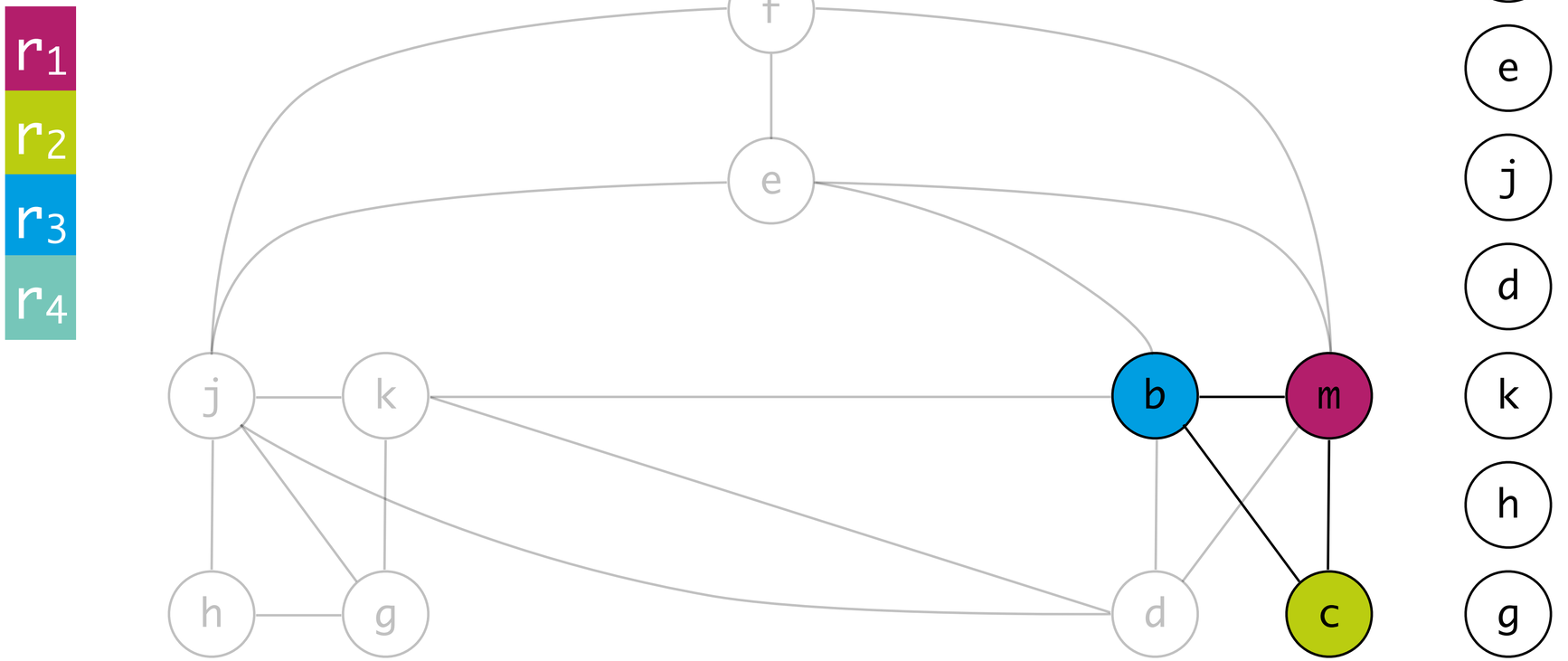
Graph Coloring



b
f
e
j
d
k
h
g

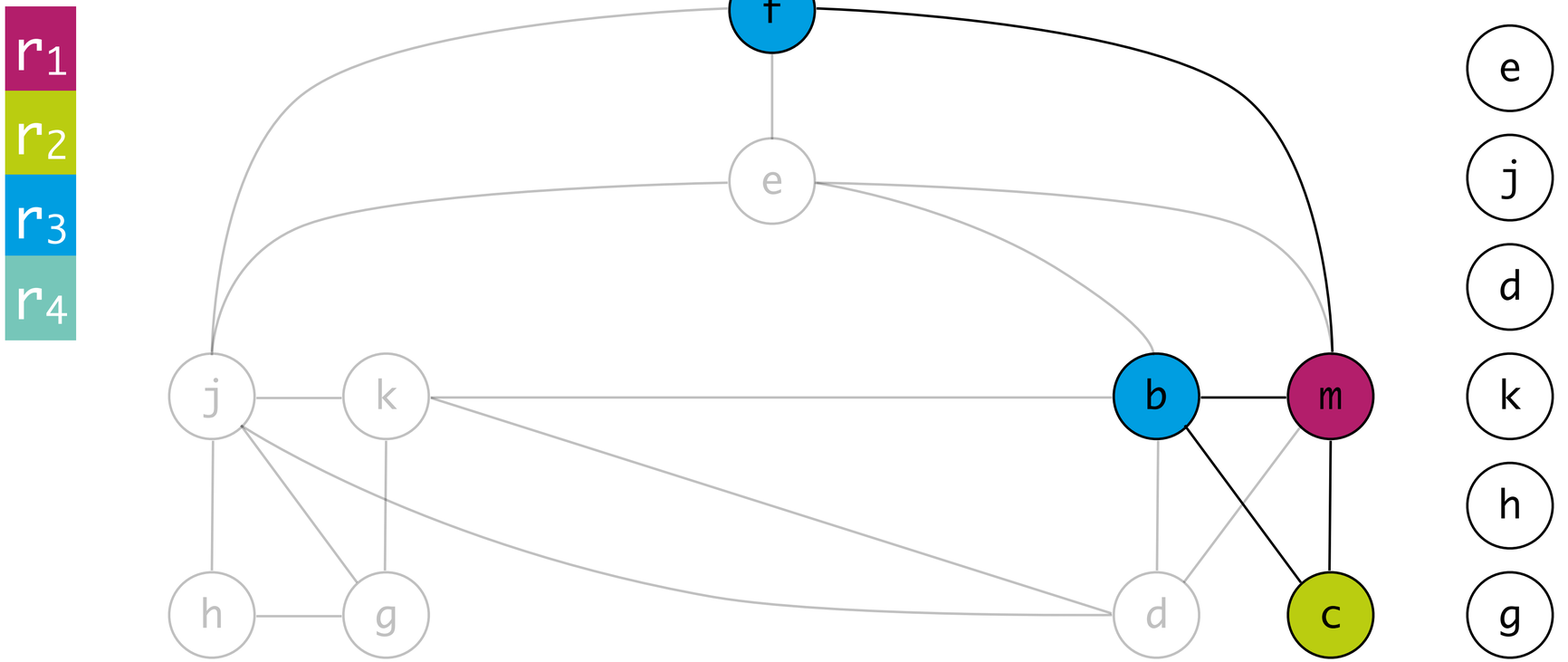
```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
r1 := mem[j + 16]
b := mem[f]
r2 := e + 8
d := r2
k := r1 + 4
j := b
live out: d k j
```

Graph Coloring



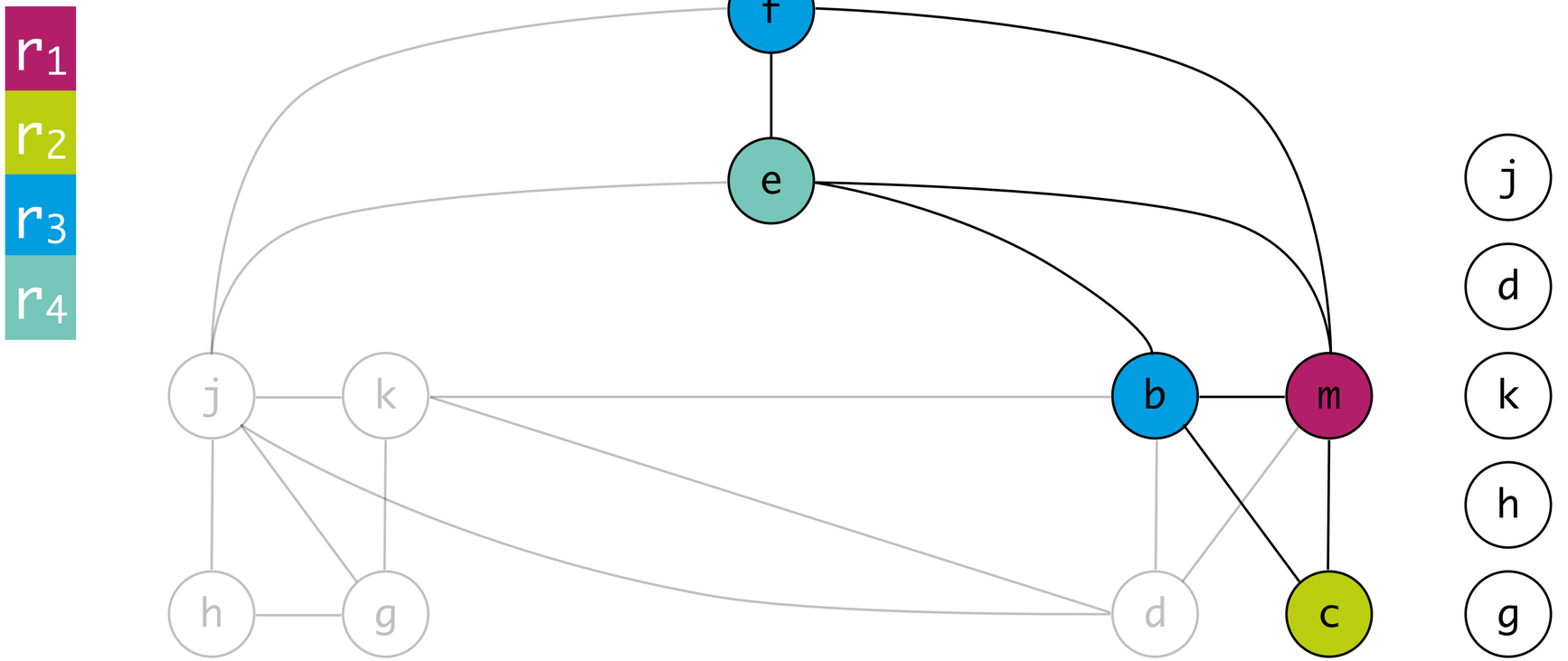

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
r1 := mem[j + 16]
r3 := mem[f]
r2 := e + 8
d := r2
k := r1 + 4
j := r3
live out: d k j
```

Graph Coloring



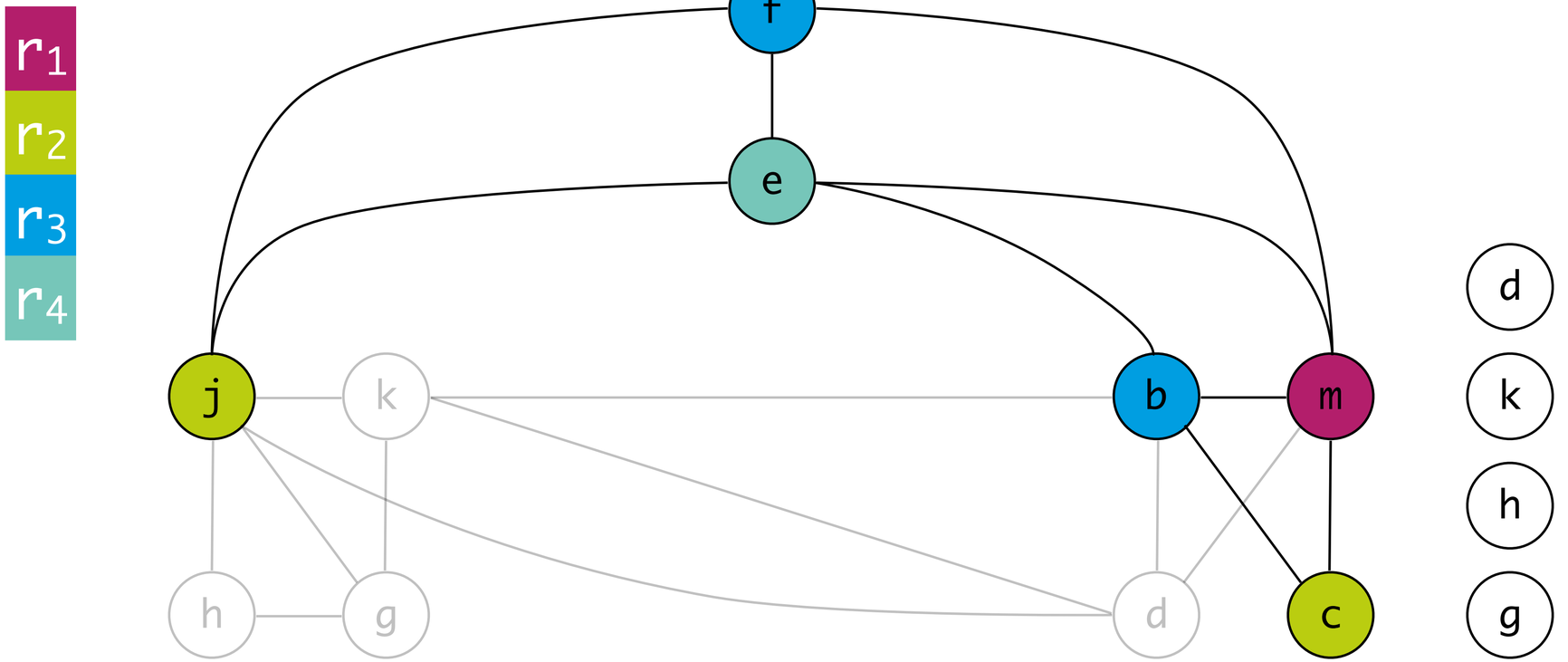

```
live-in: k j
g := mem[j + 12]
h := k - 1
r3 := g * h
e := mem[j + 8]
r1 := mem[j + 16]
r3 := mem[r3]
r2 := e + 8
d := r2
k := r1 + 4
j := r3
live out: d k j
```

Graph Coloring



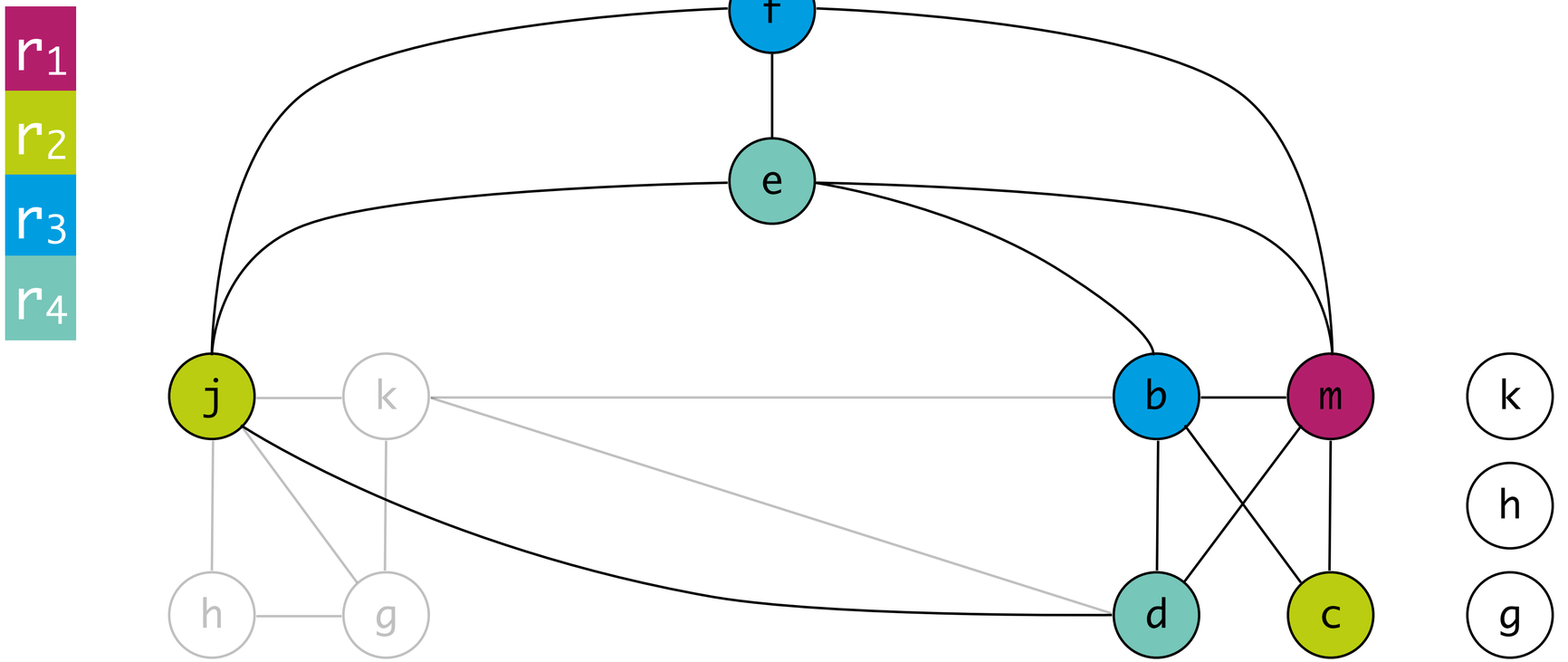

```
live-in: k j
g := mem[j + 12]
h := k - 1
r3 := g * h
r4 := mem[j + 8]
r1 := mem[j + 16]
r3 := mem[r3]
r2 := r4 + 8
d := r2
k := r1 + 4
j := r3
live out: d k j
```

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
```

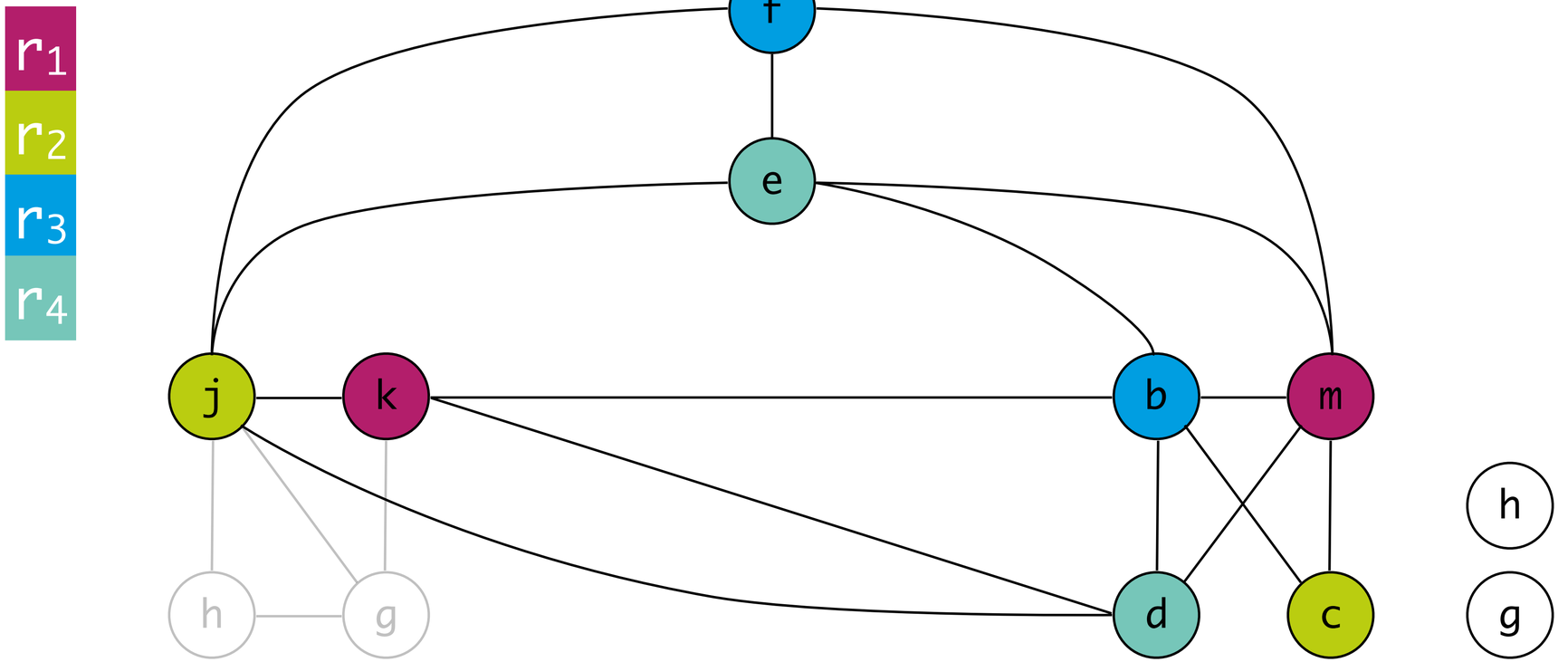
Graph Coloring




```
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
```

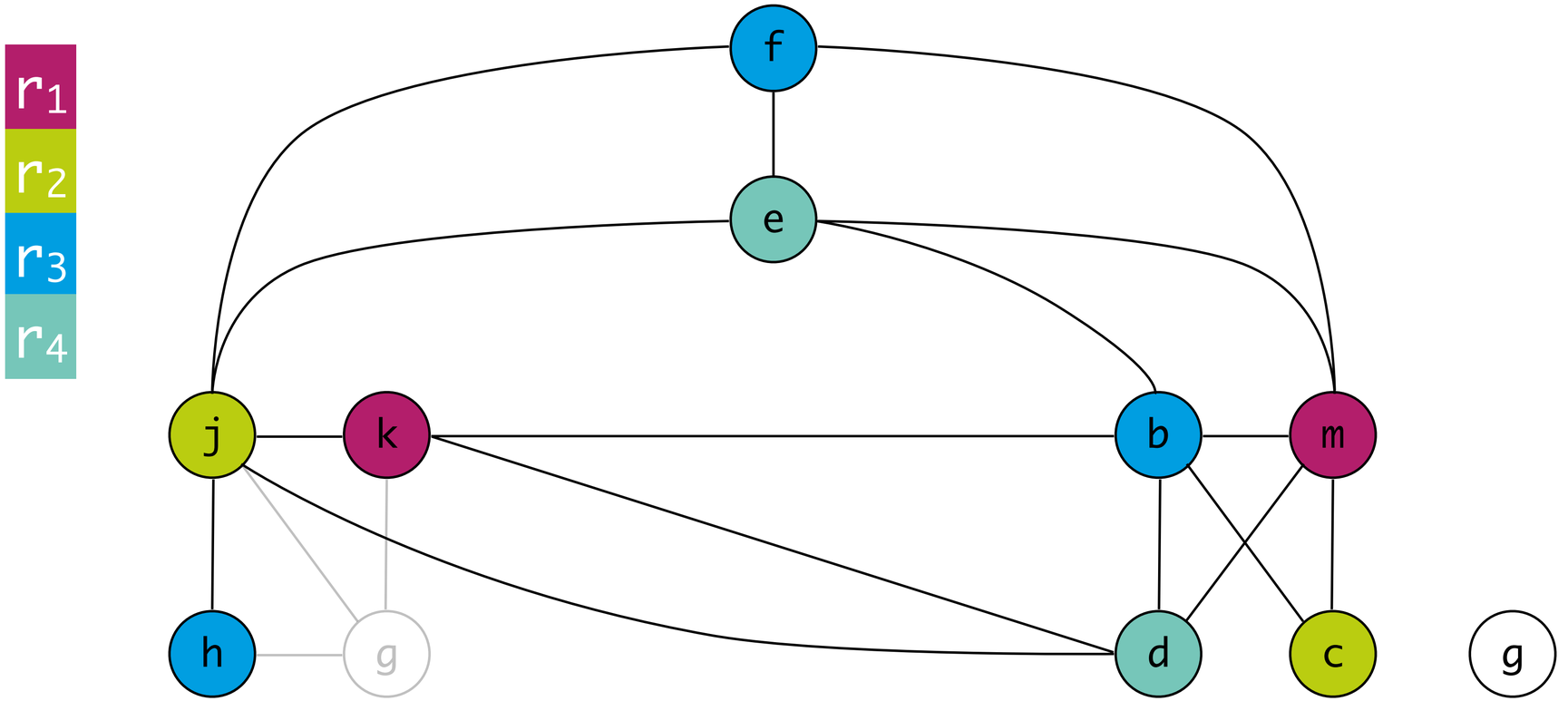
Graph Coloring

```
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
```

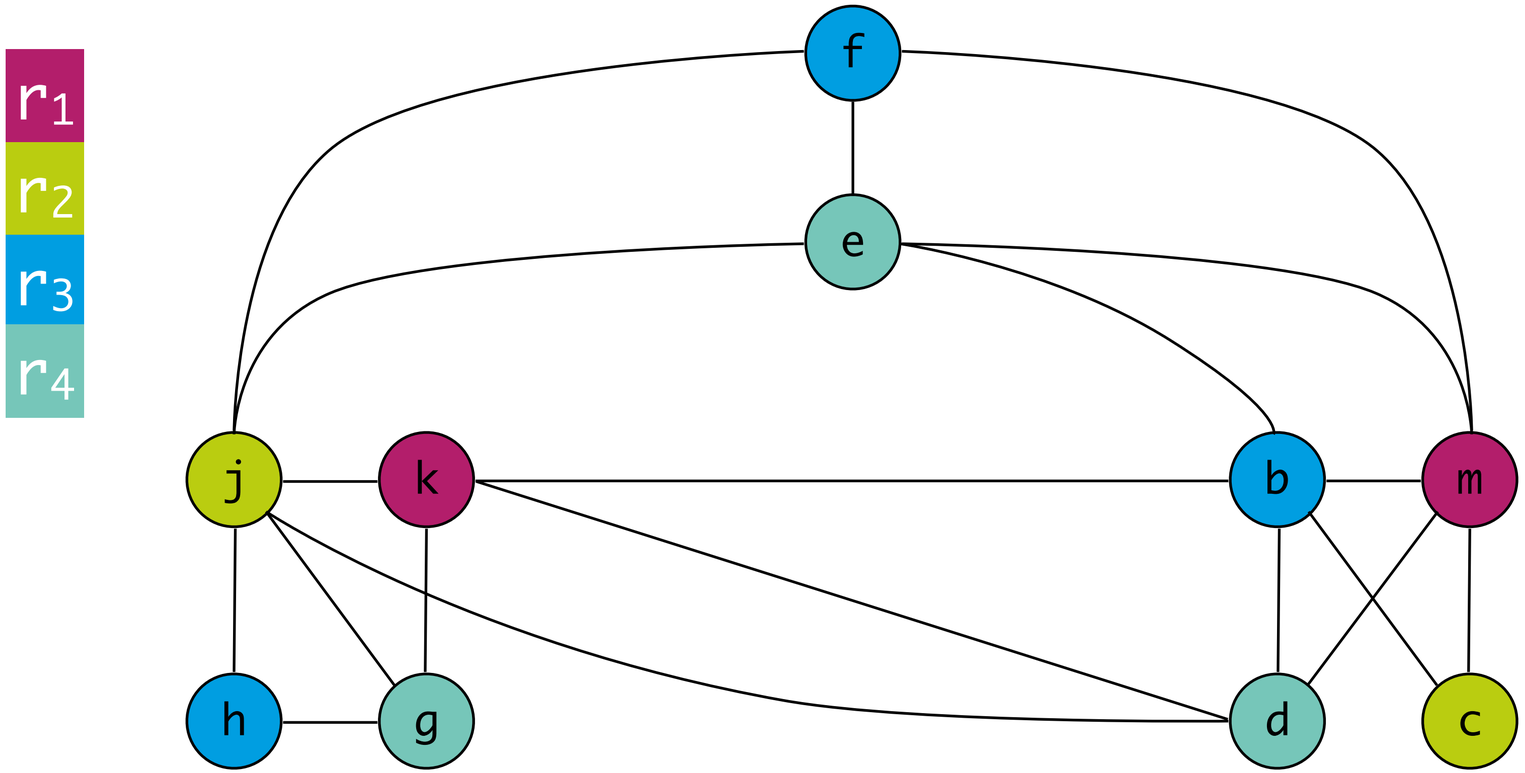


Graph Coloring

```
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)

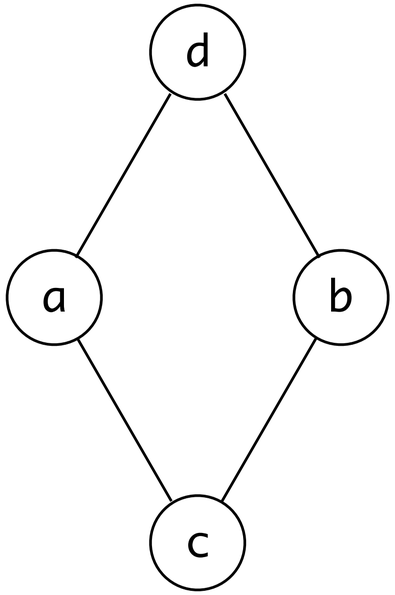
Spill

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

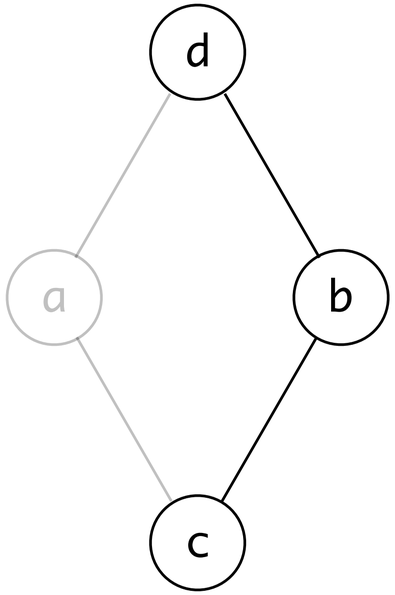
Select

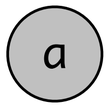
- add node, select color

Optimistic Coloring: Example with 2 Colors

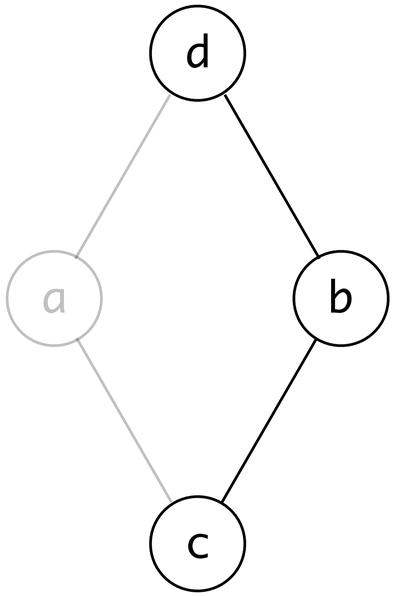


Optimistic Coloring: Example with 2 Colors





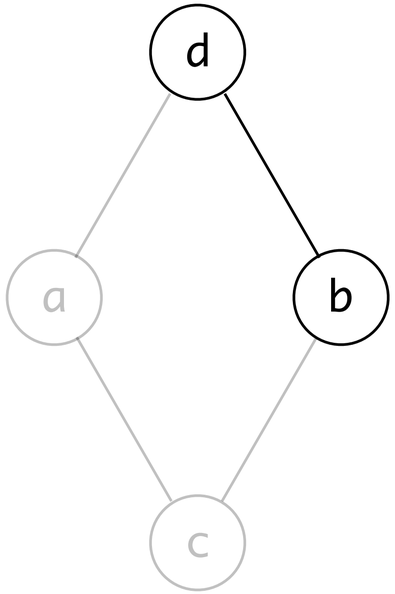
Optimistic Coloring: Example with 2 Colors



potential spill

a

Optimistic Coloring: Example with 2 Colors

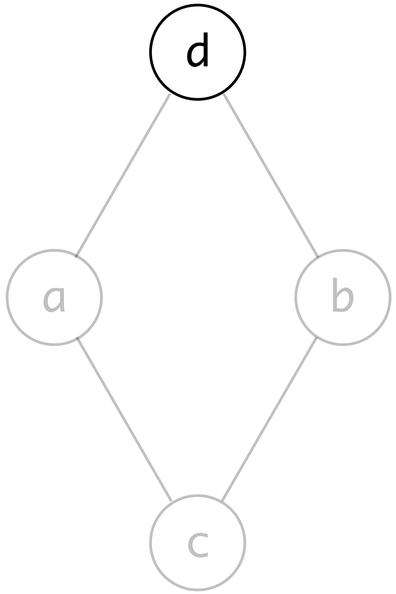


potential spill

c

a

Optimistic Coloring: Example with 2 Colors



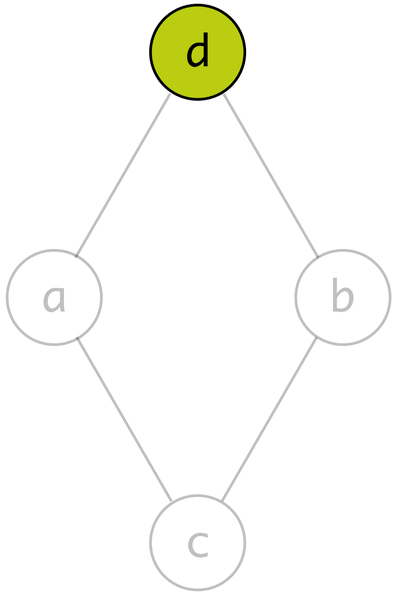
potential spill

b

c

a

Optimistic Coloring: Example with 2 Colors



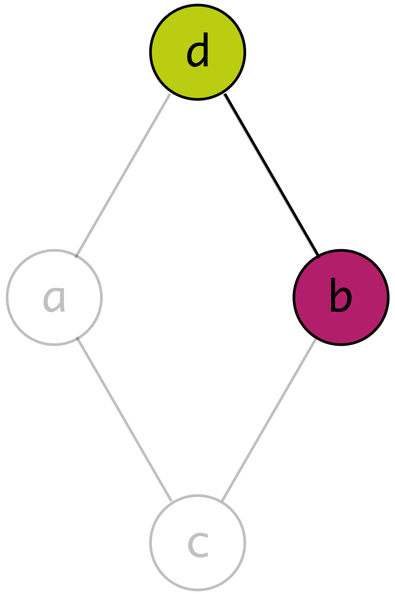
potential spill

b

c

a

Optimistic Coloring: Example with 2 Colors

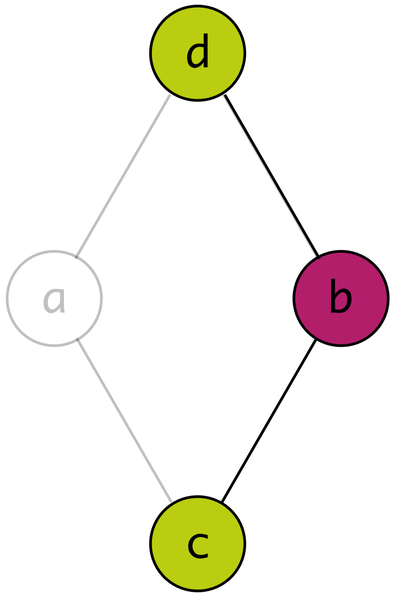


potential spill

c

a

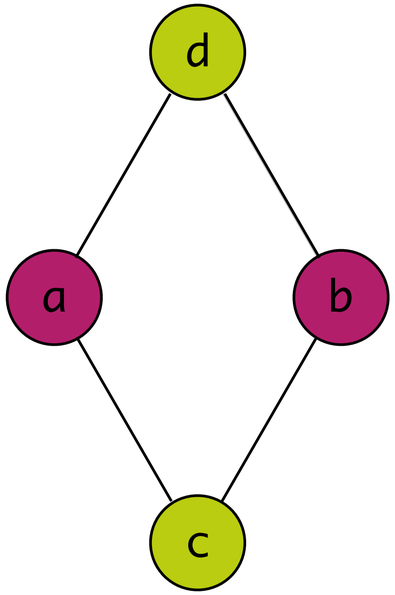
Optimistic Coloring: Example with 2 Colors



potential spill

a

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)

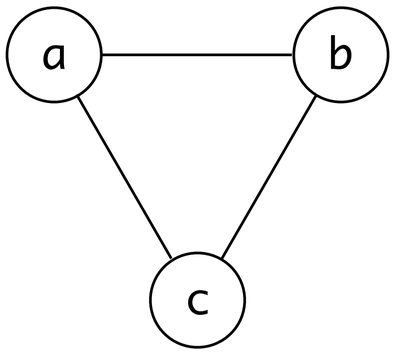
Select

- add node, select color

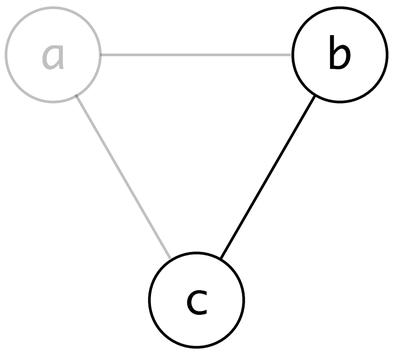
Actual spill

Start over

Spilling: example with 2 colors



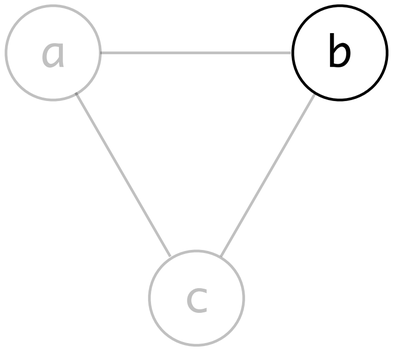
Spilling: example with 2 colors



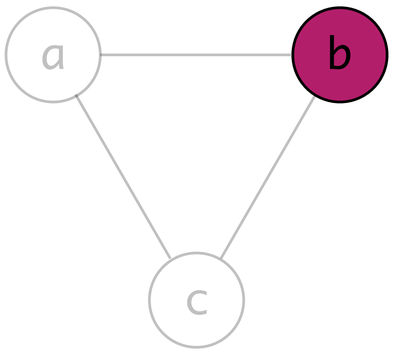
potential spill

a

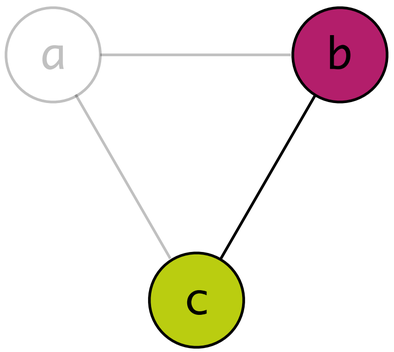
Spilling: example with 2 colors



Spilling: example with 2 colors



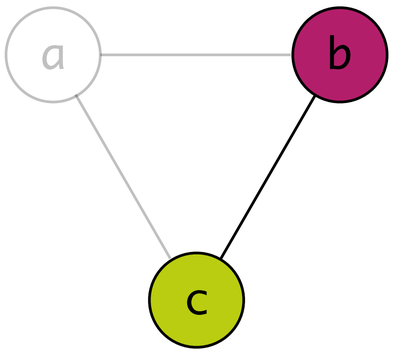
Spilling: example with 2 colors



potential spill

a

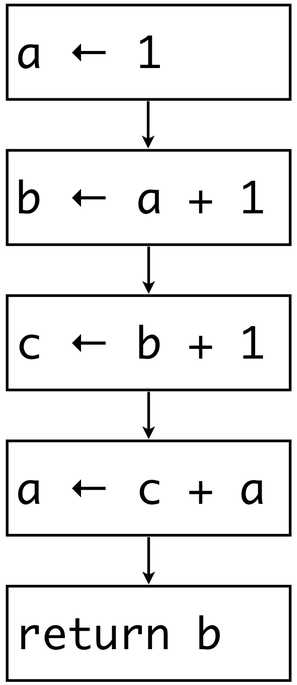
Spilling: example with 2 colors



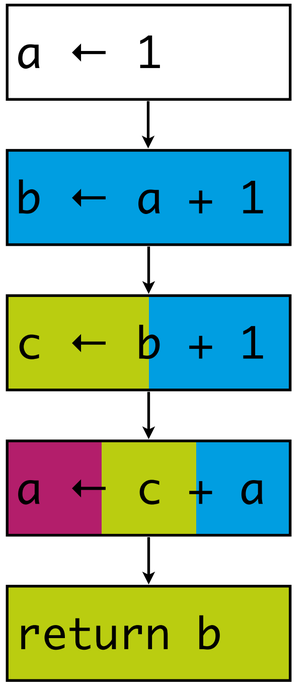
actual spill

a

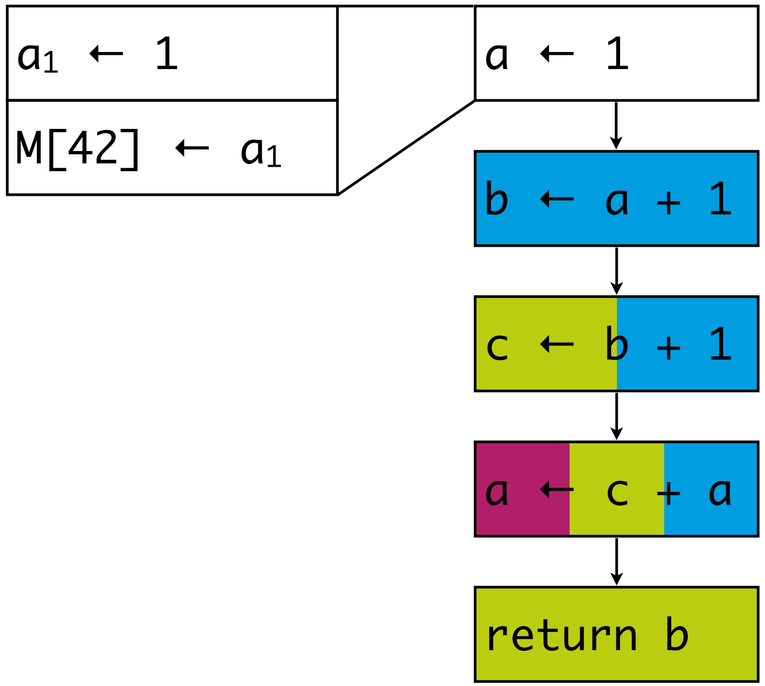
Spilling: Example



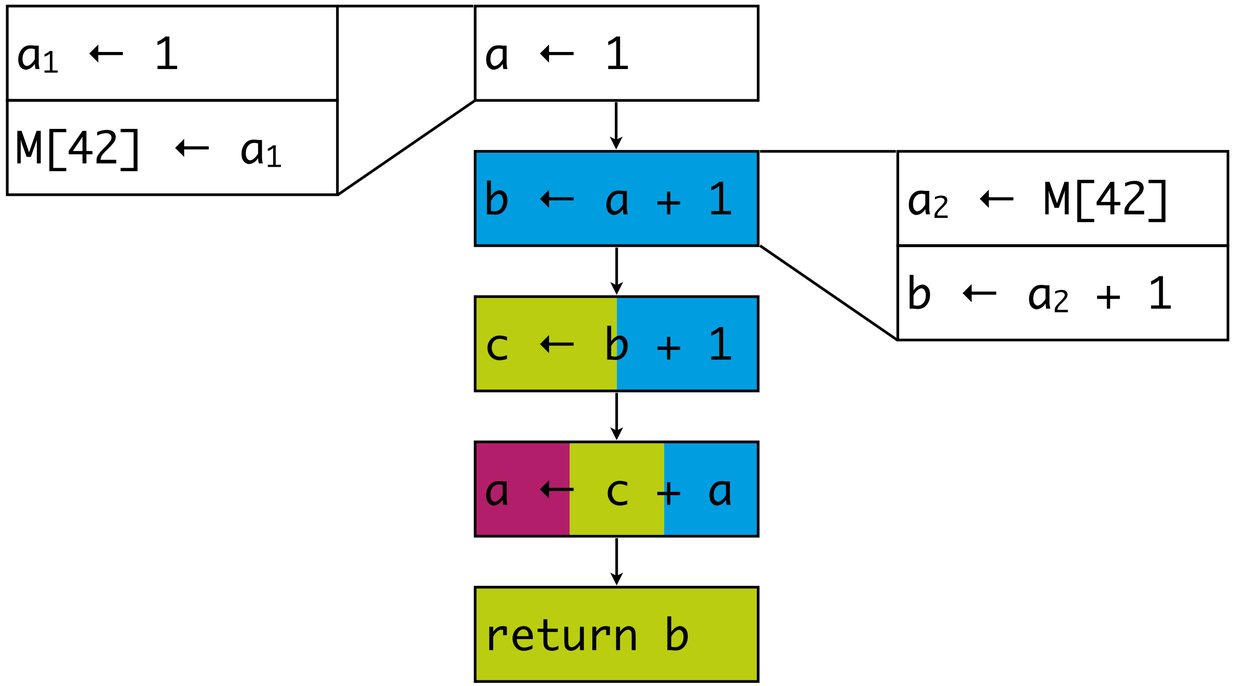
Spilling: Example



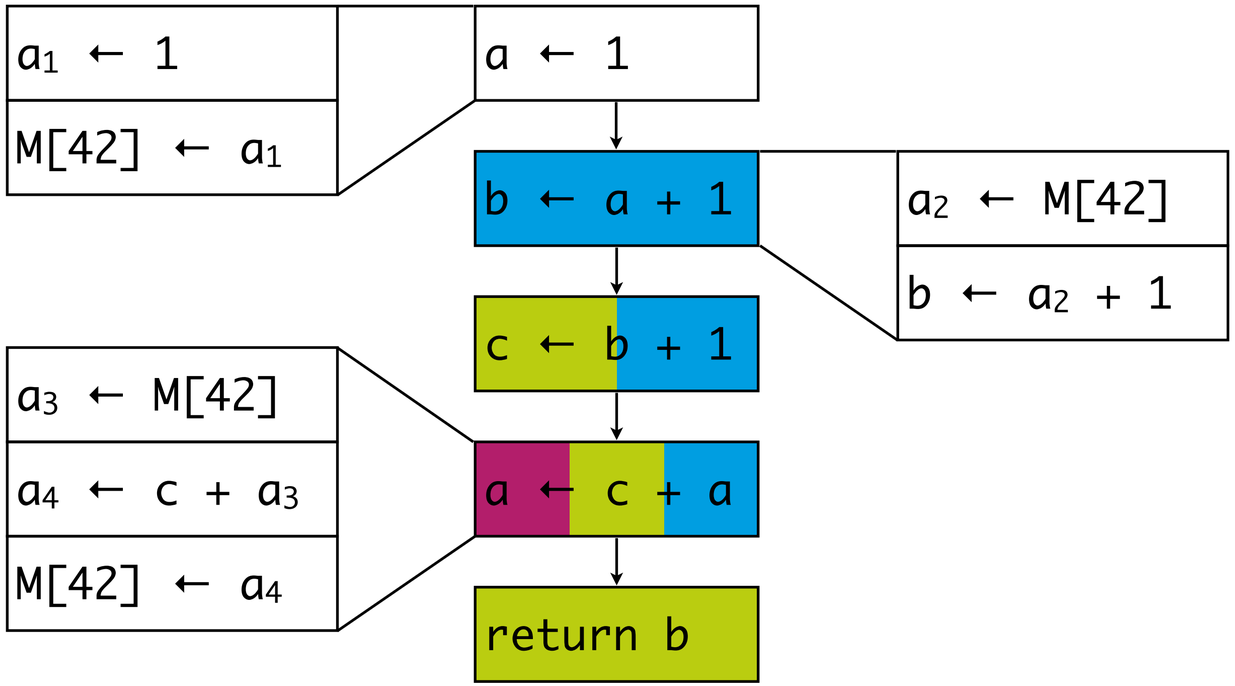
Spilling: Example



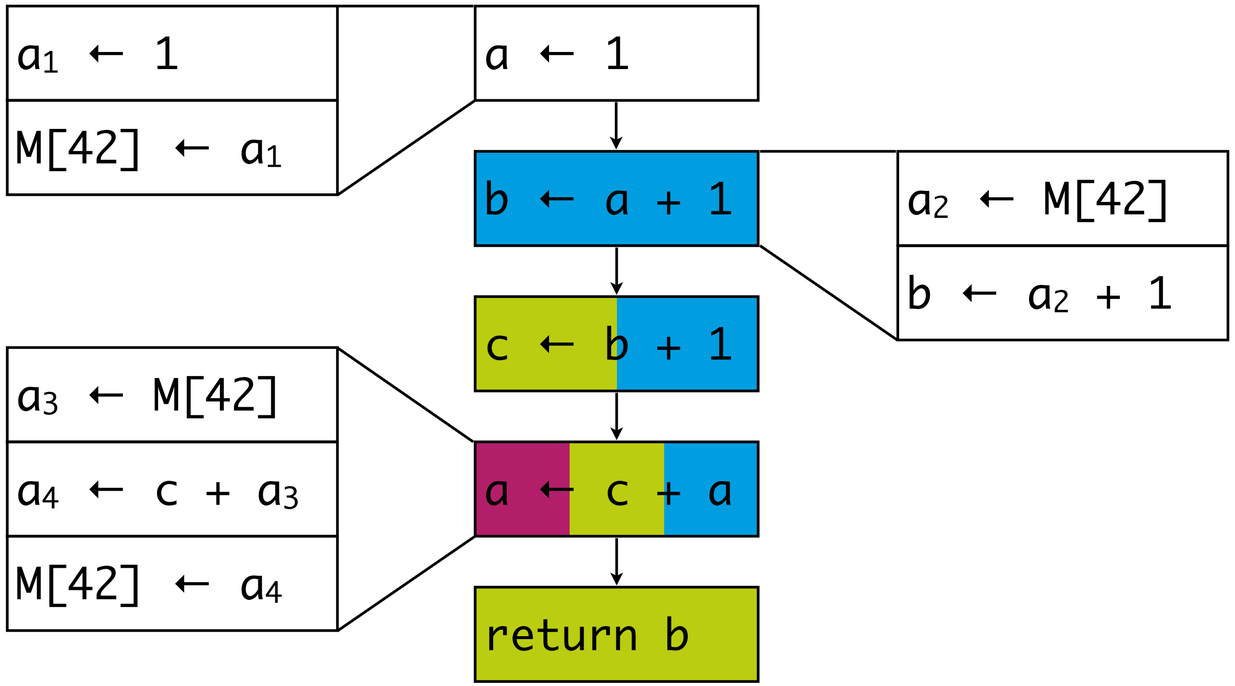
Spilling: Example



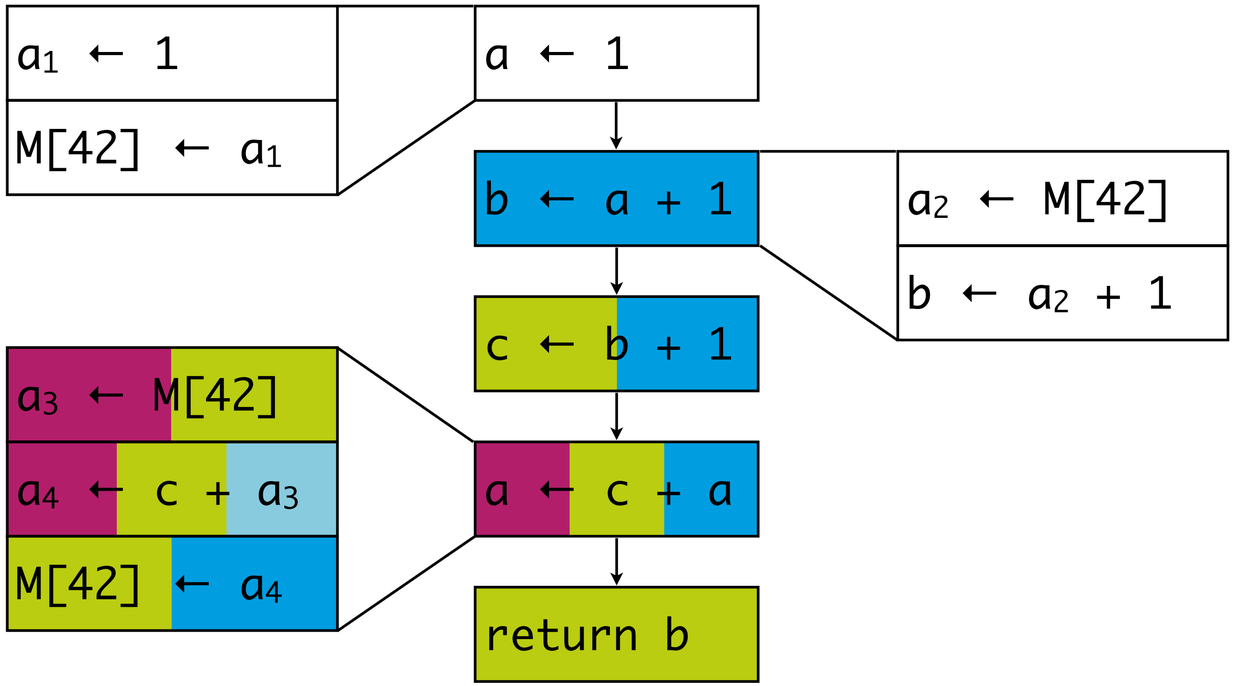
Spilling: Example



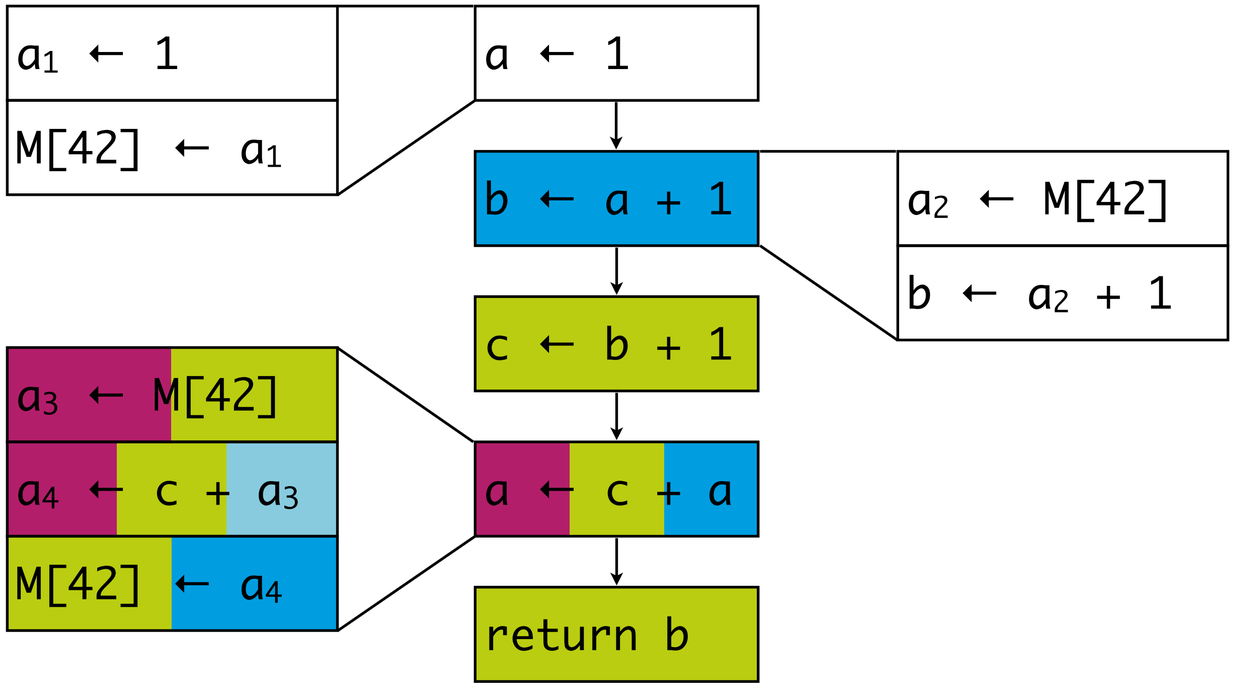
Spilling: Example



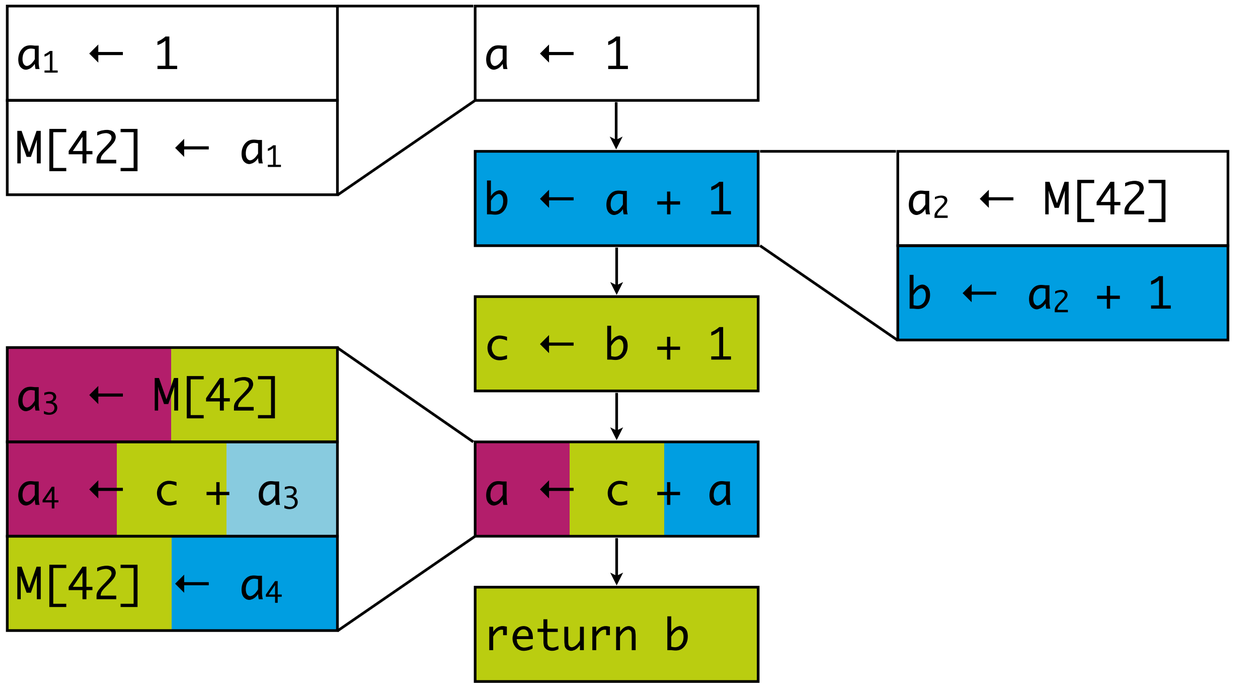
Spilling: Example



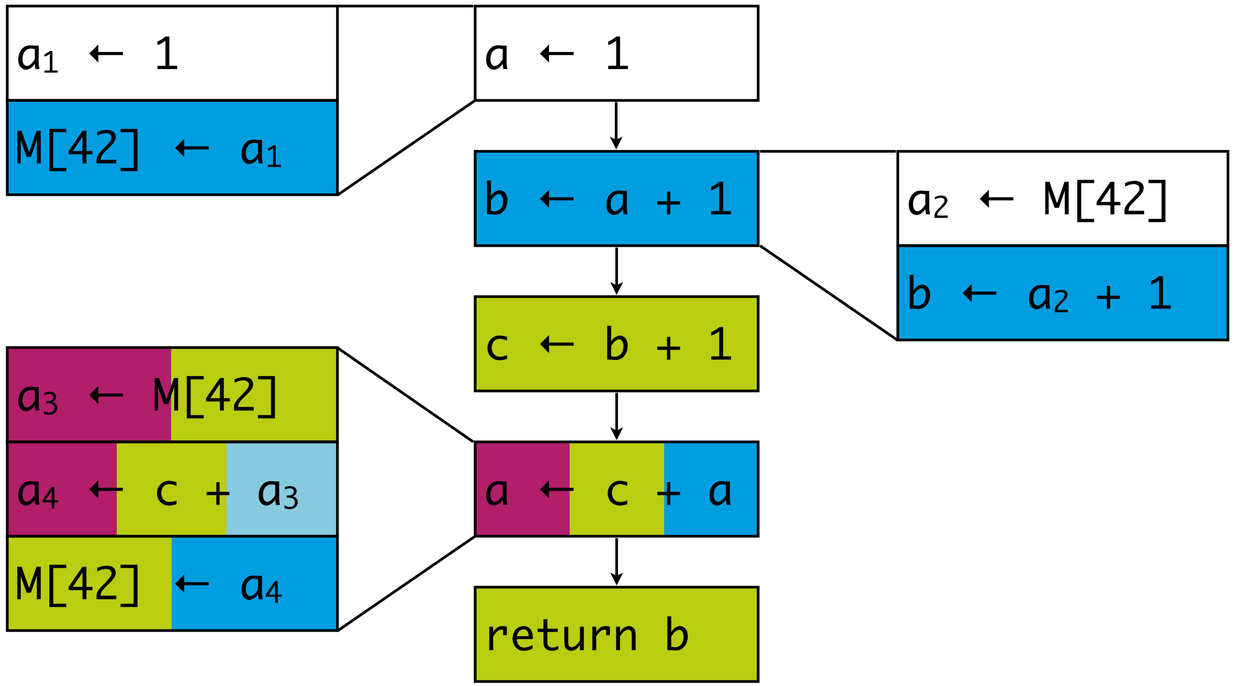
Spilling: Example



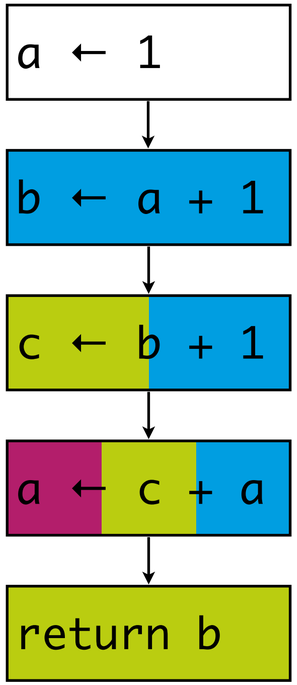
Spilling: Example



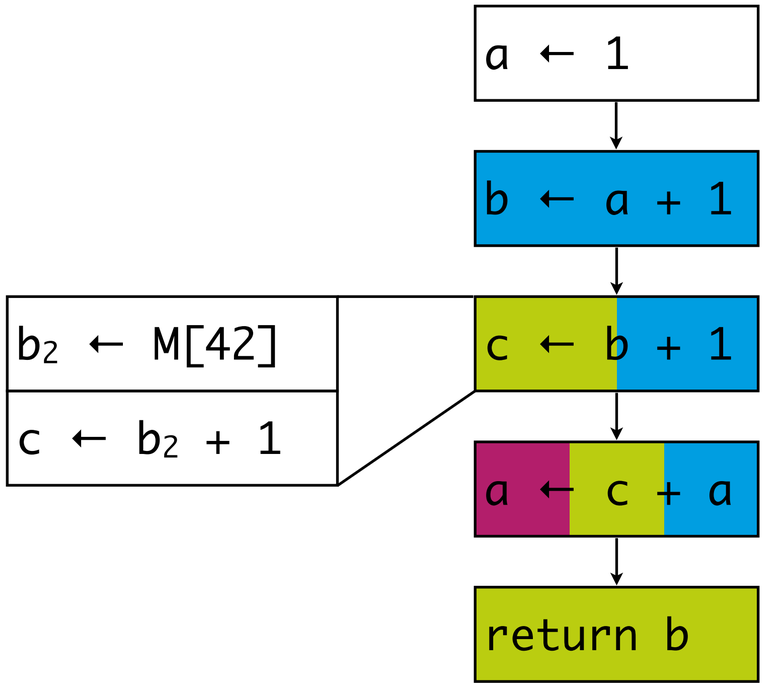
Spilling: Example



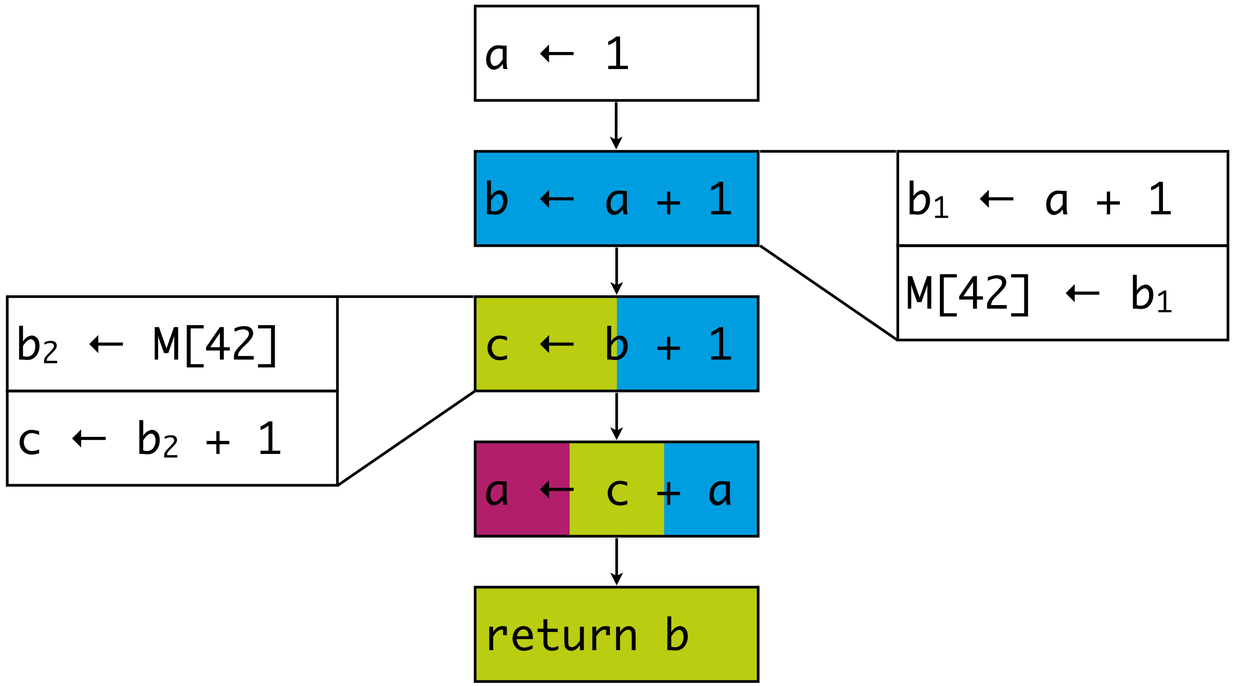
Spilling: Example



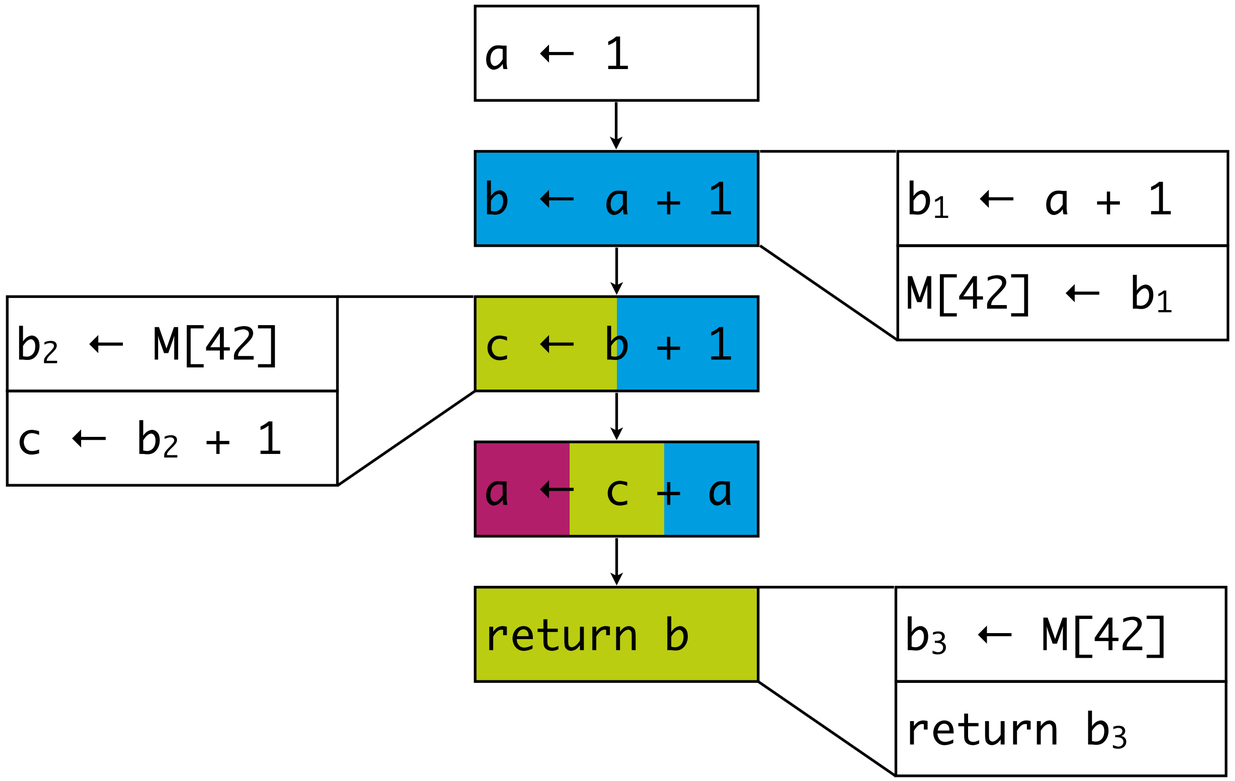
Spilling: Example



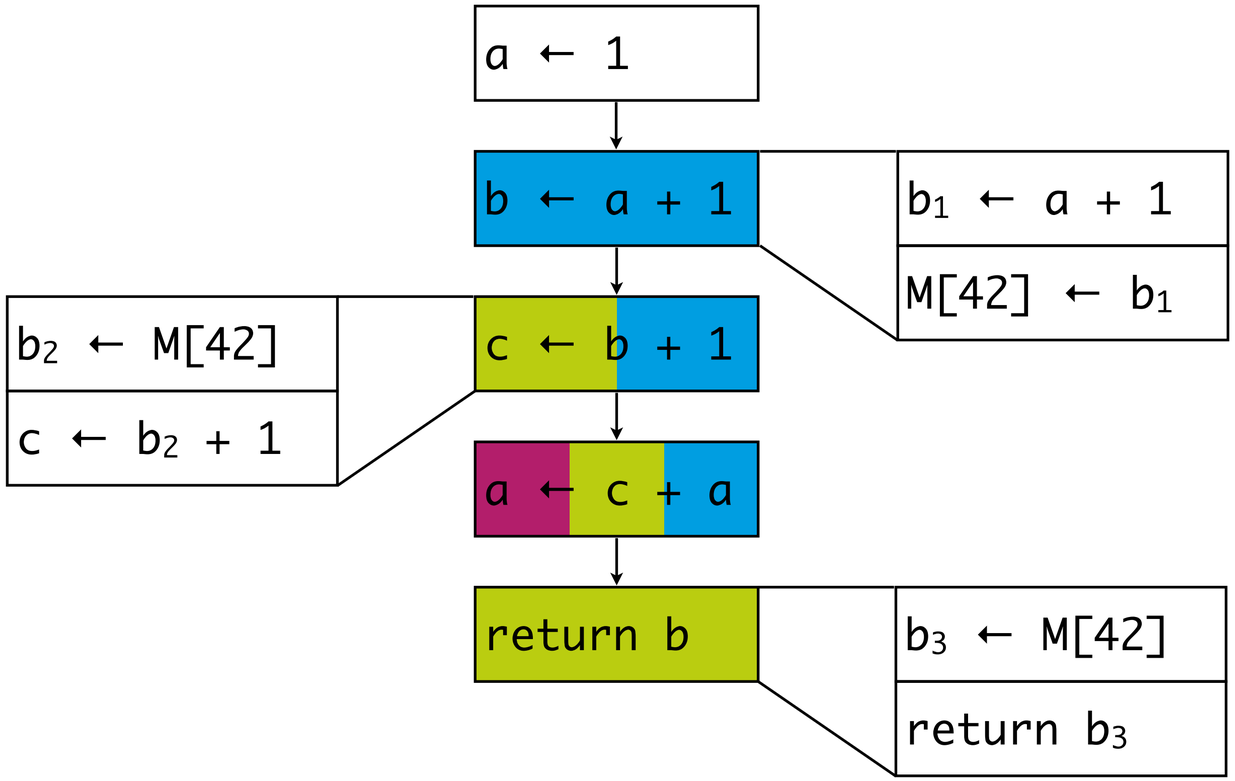
Spilling: Example



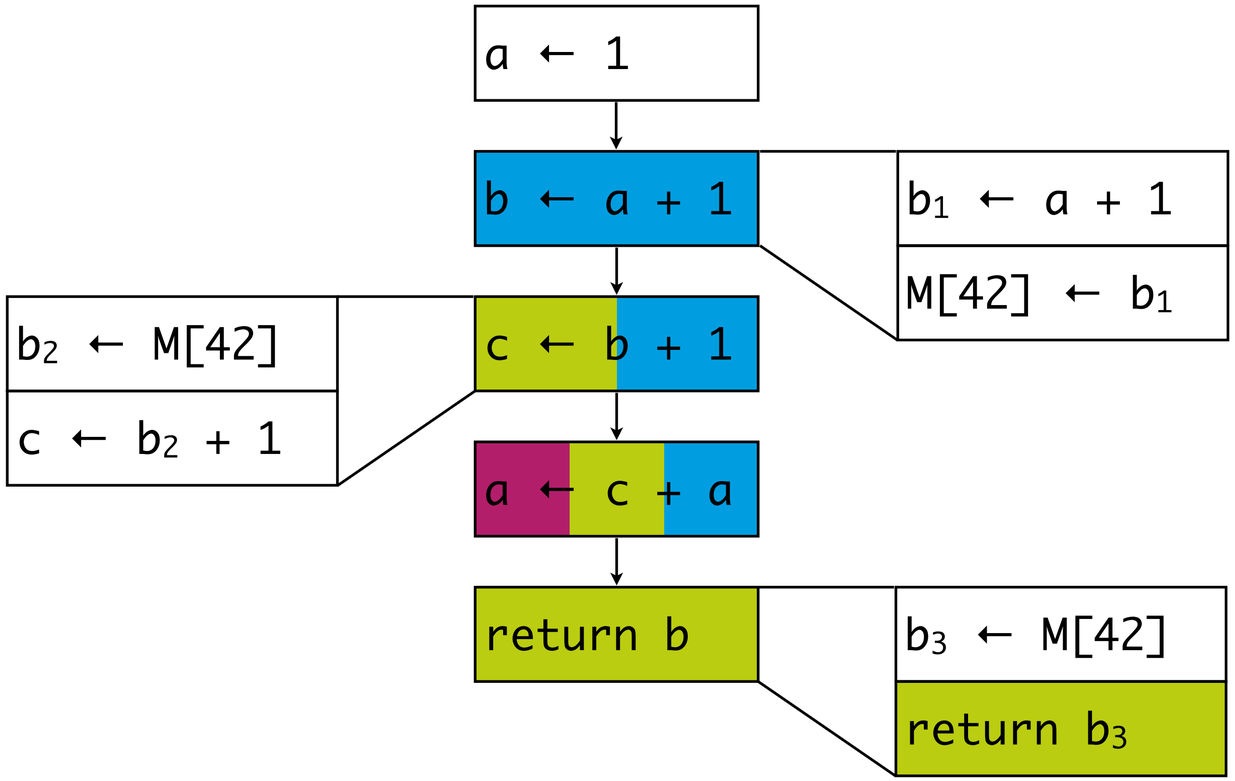
Spilling: Example



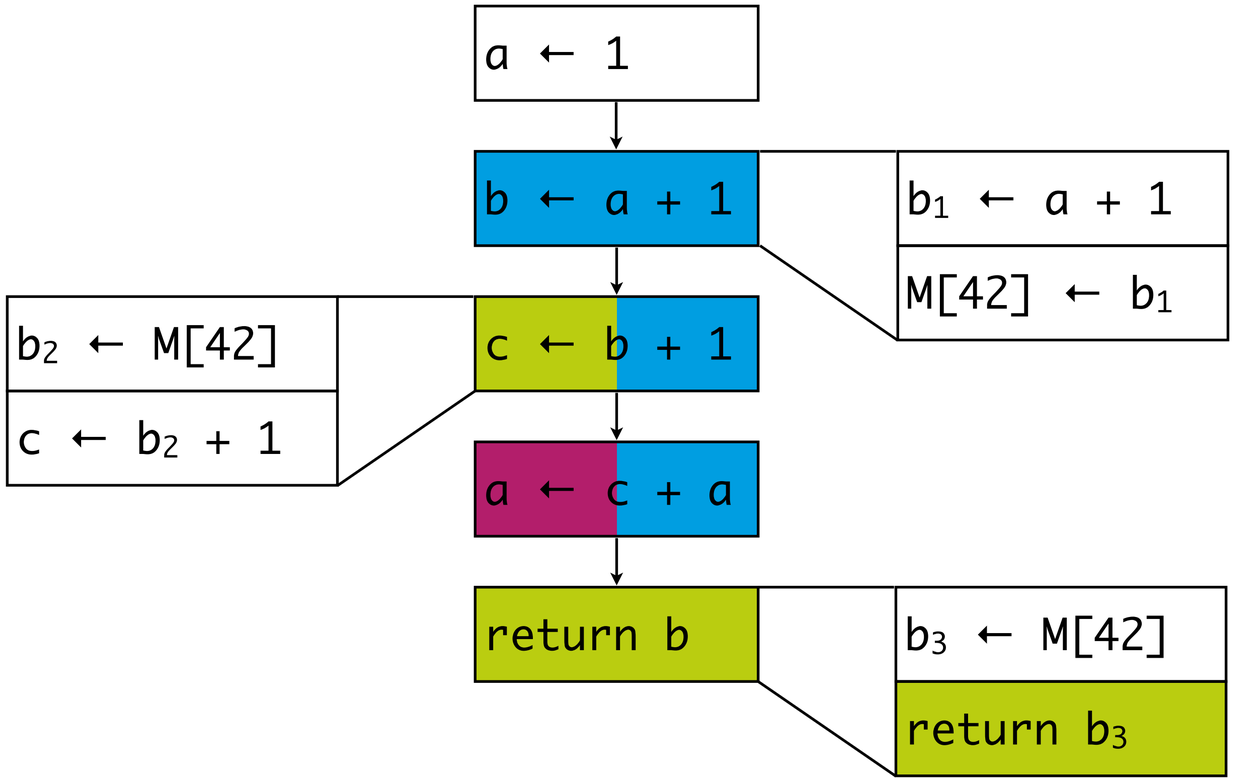
Spilling: Example



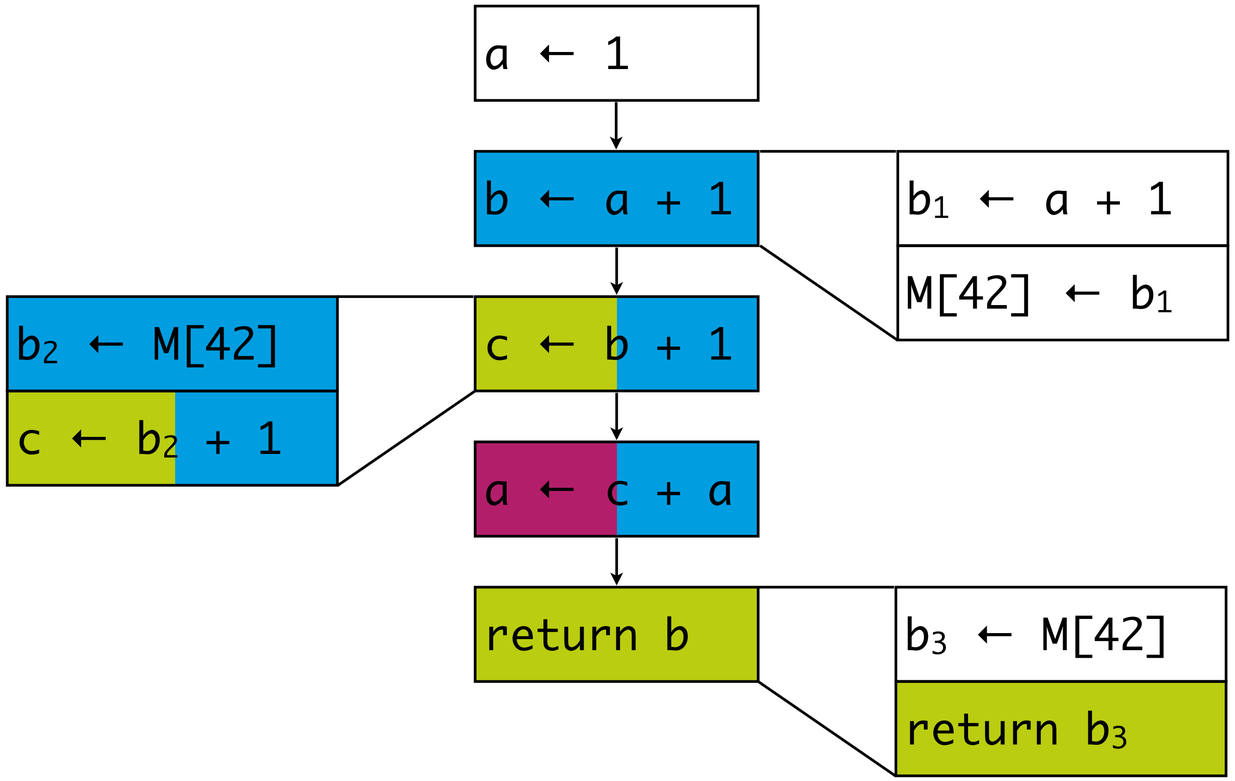
Spilling: Example



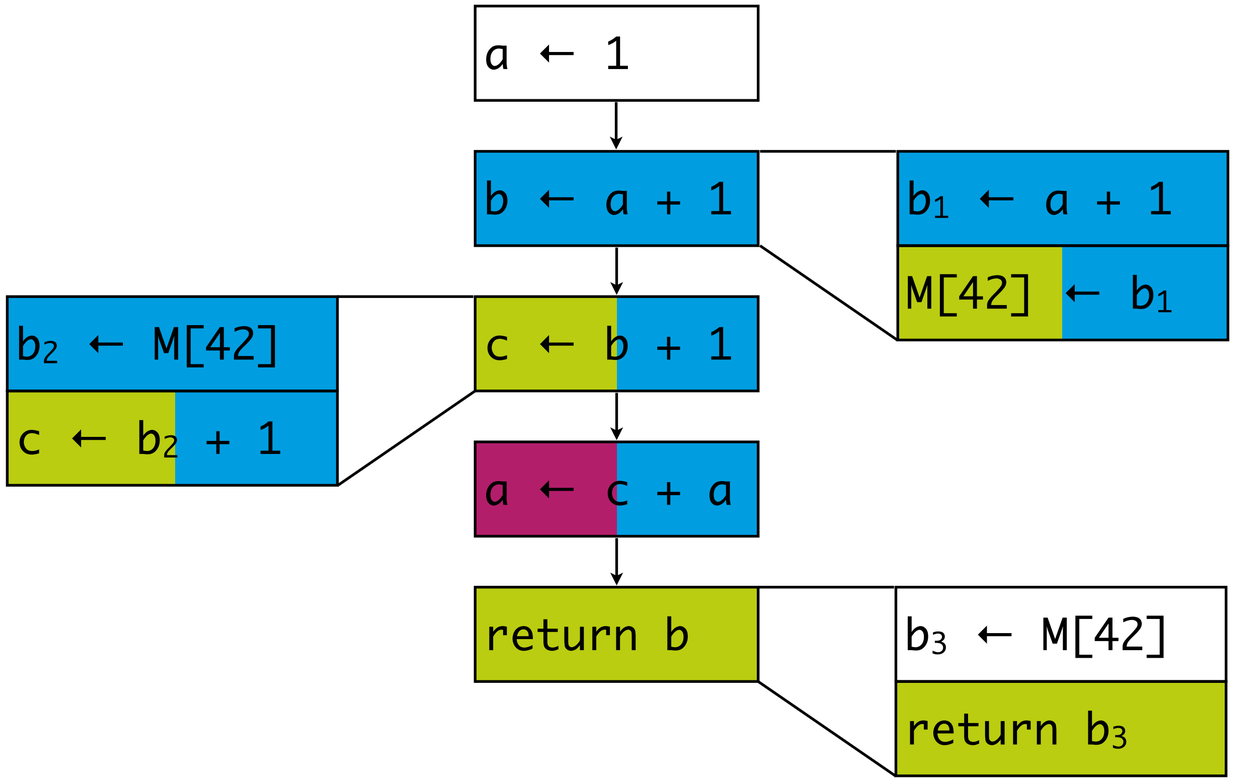
Spilling: Example



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
```

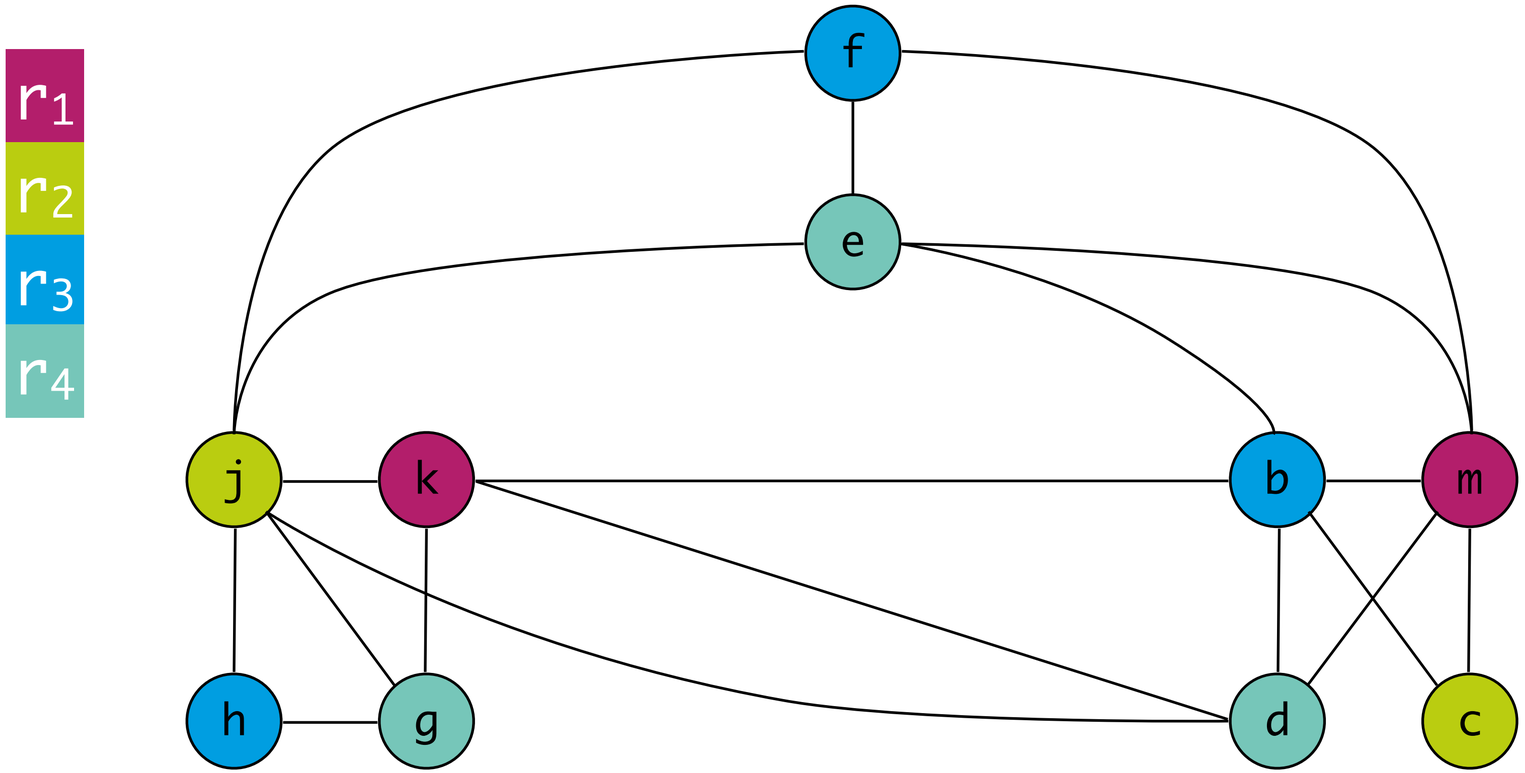
Eliminating Move Instructions

```
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

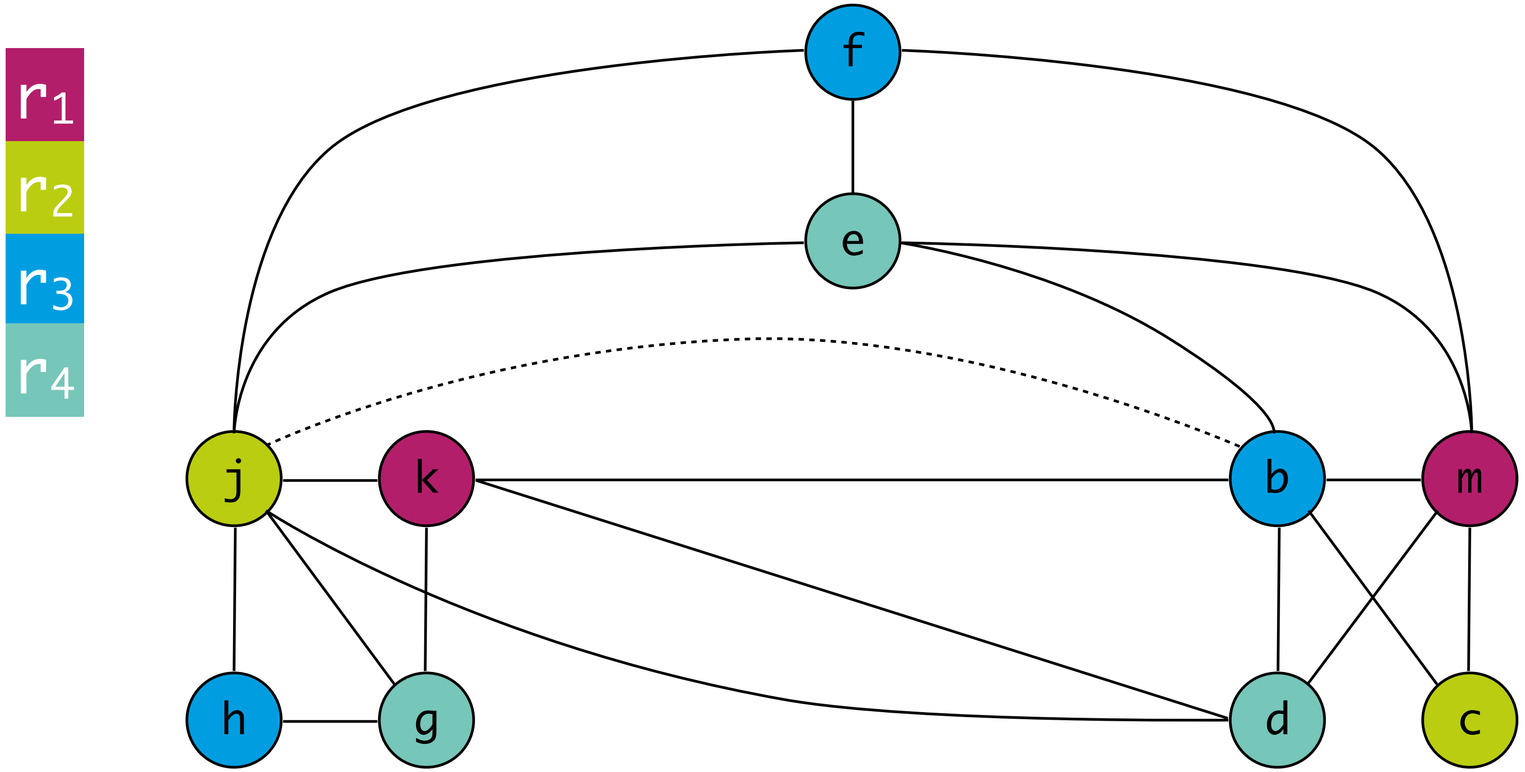
coalesce |ˌkəʊəˈlɛs|
verb *[no object]*
come together to form one mass or whole: *the puddles had **coalesced into** shallow streams*.
• *[with object]* combine (elements) in a mass or whole: *his idea served to **coalesce** all that happened **into** one connected whole*.

Recap: Graph Coloring



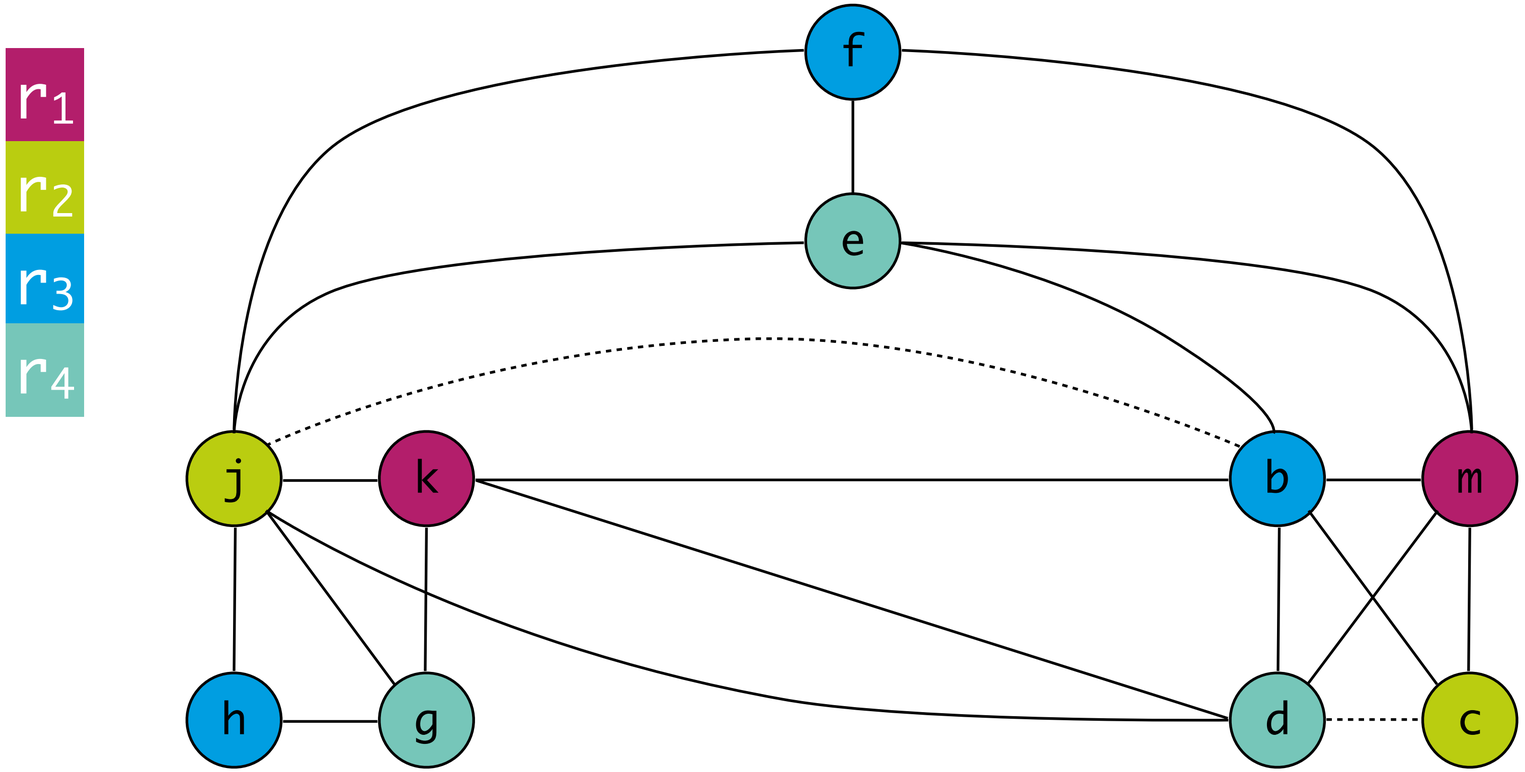

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Recap: Graph Coloring



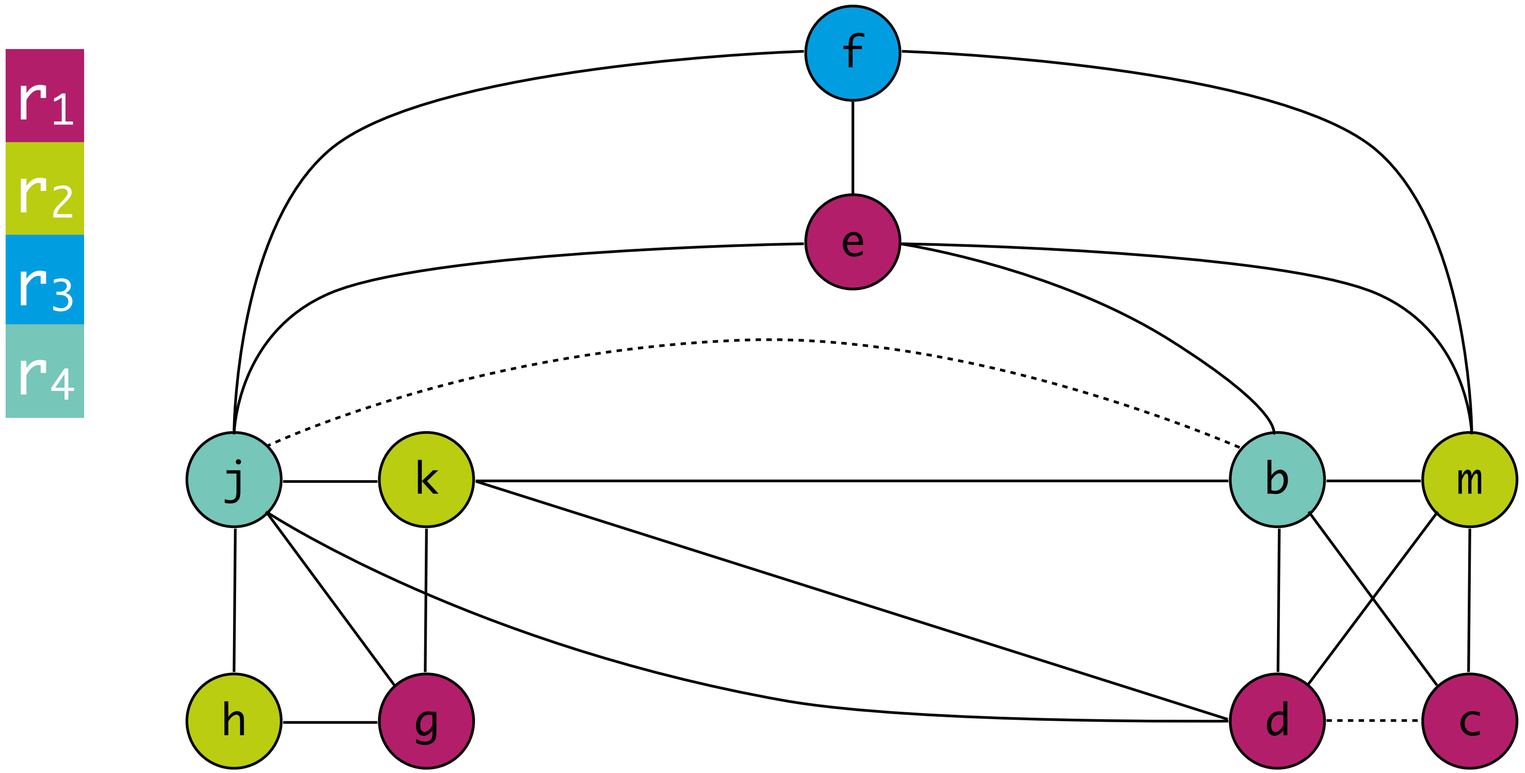

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Recap: Graph Coloring



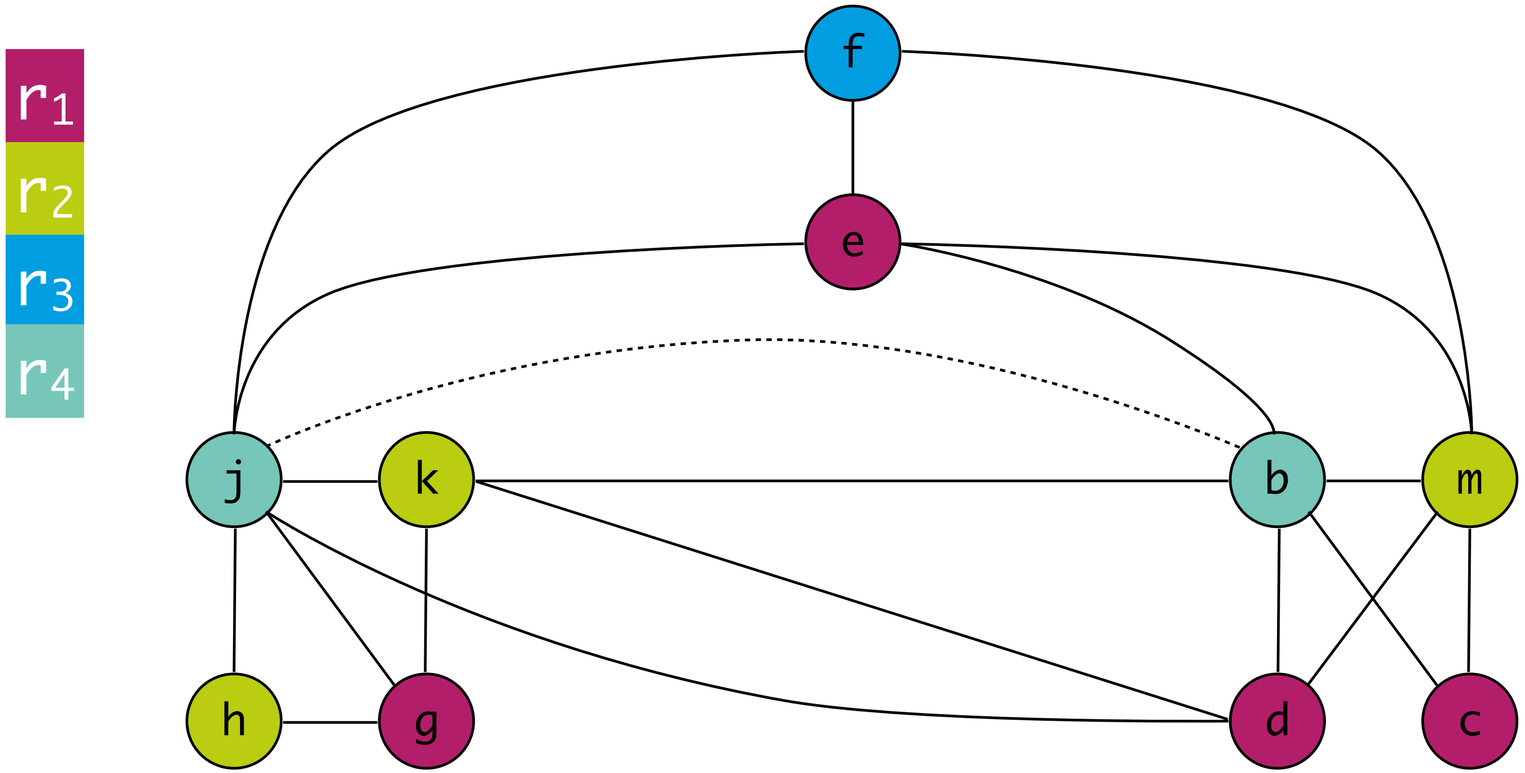

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing: better solution



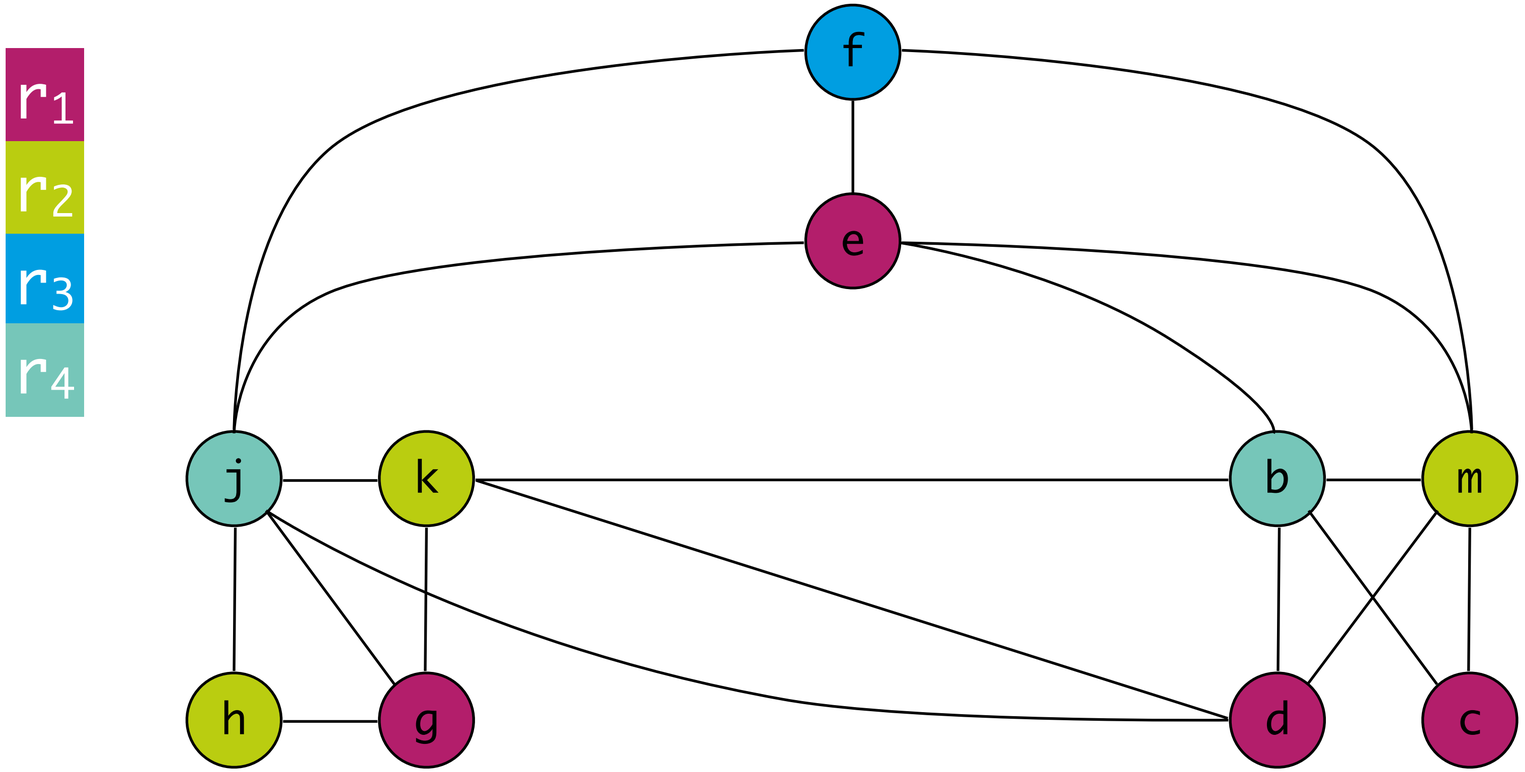

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing: better solution



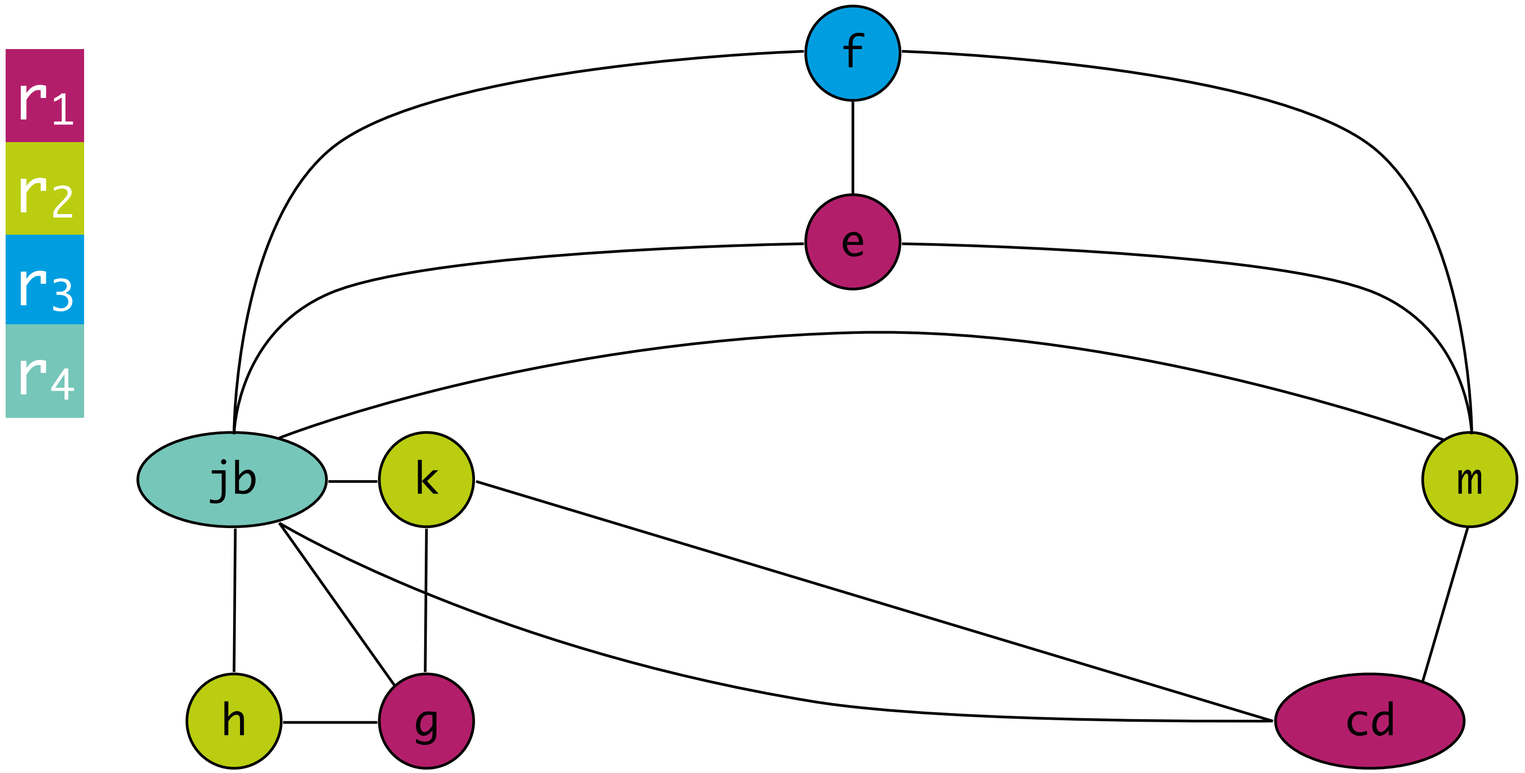

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing: 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

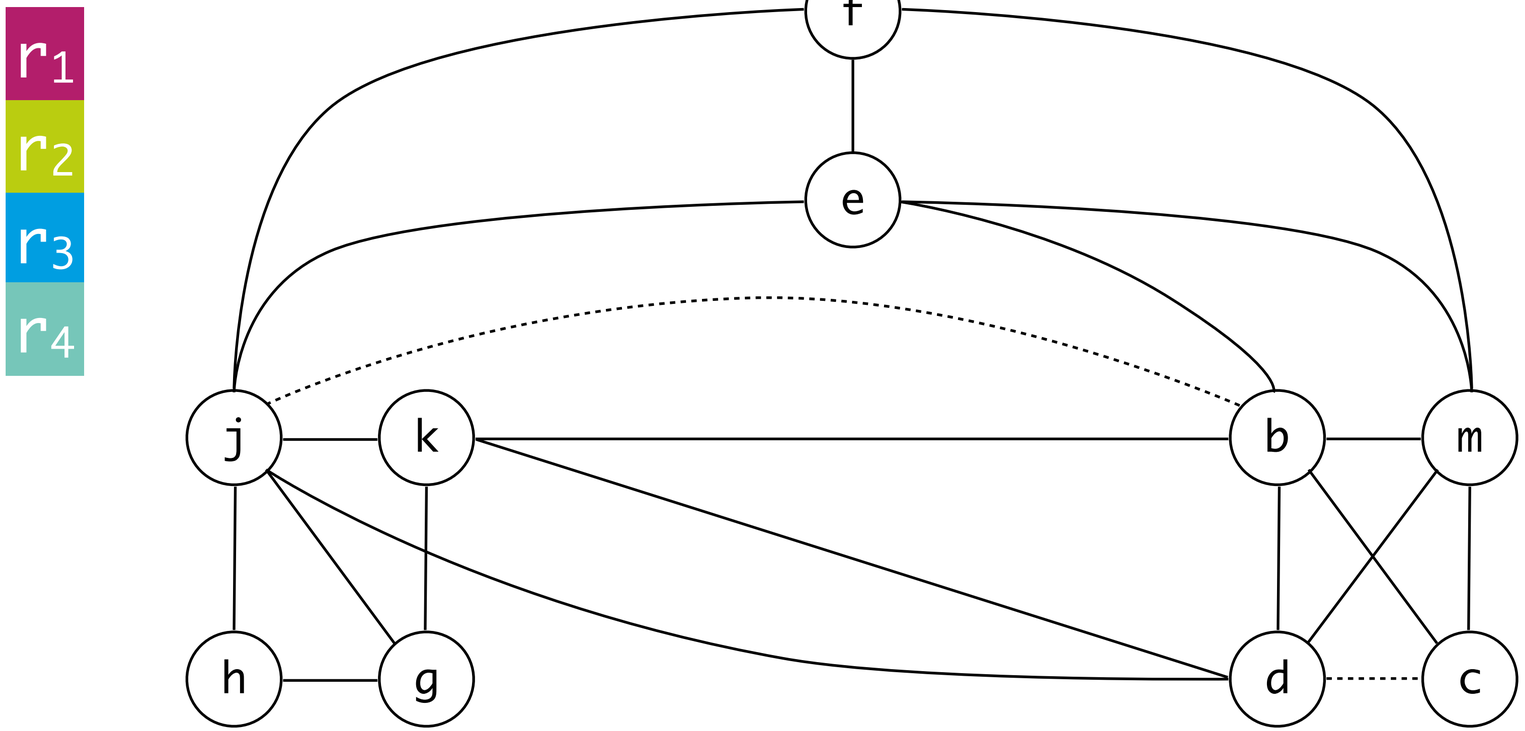
- turn move-related node of insignificant degree into non-move-related

Spill

Select

Start over

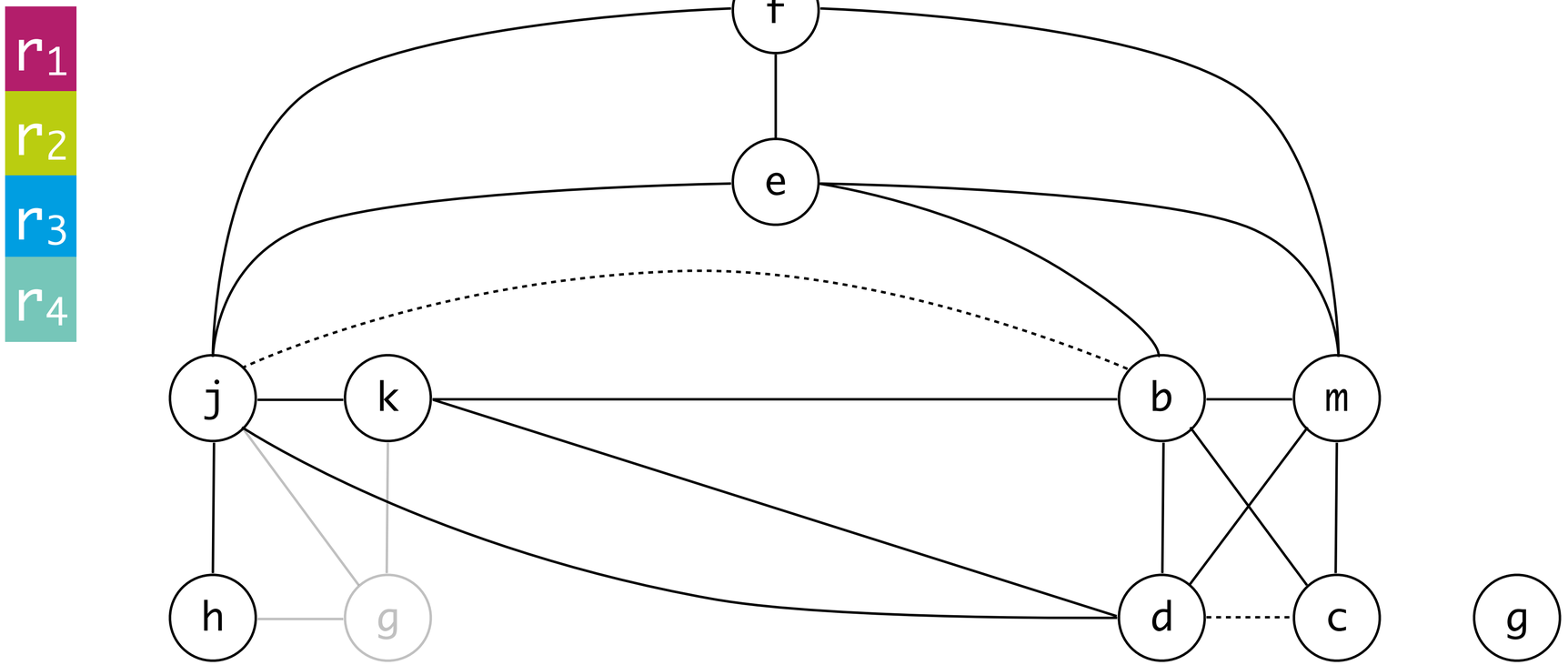
Coalescing: example




```
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
```

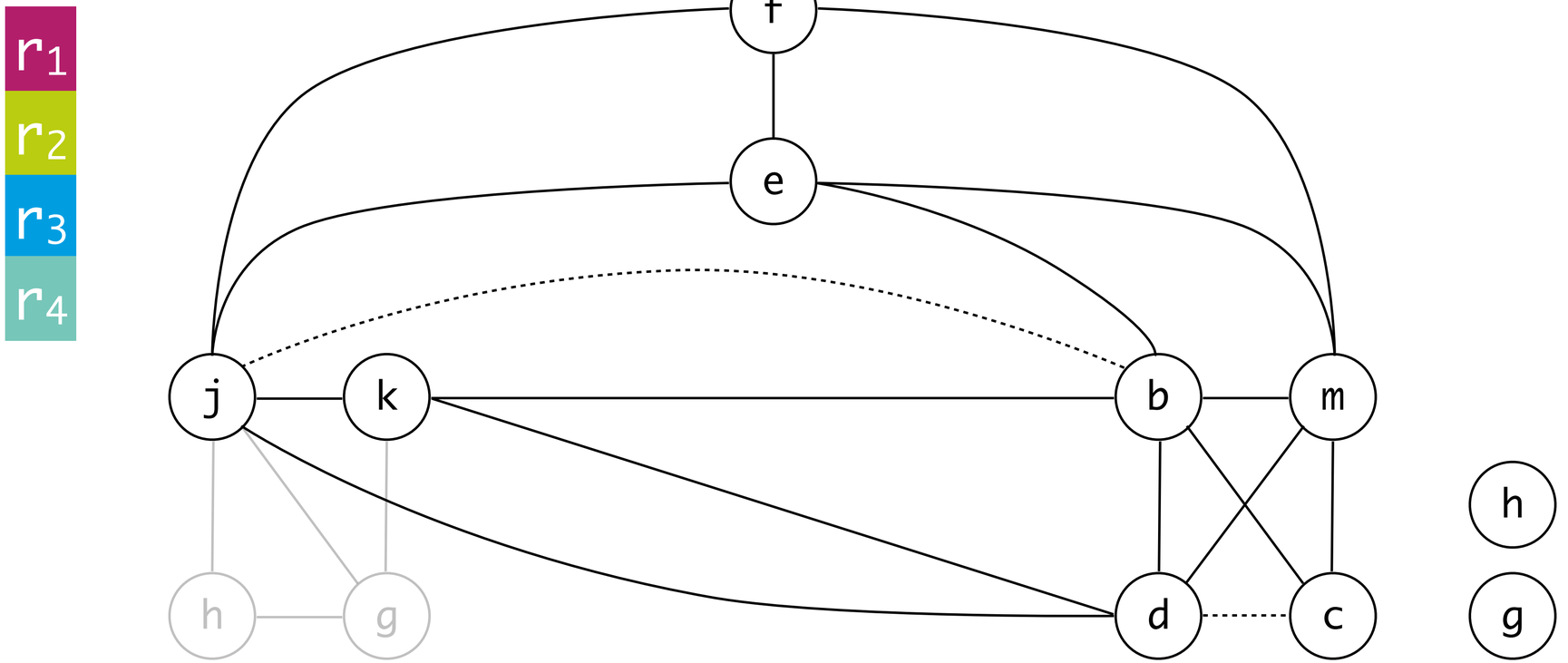
Coalescing

```
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
```



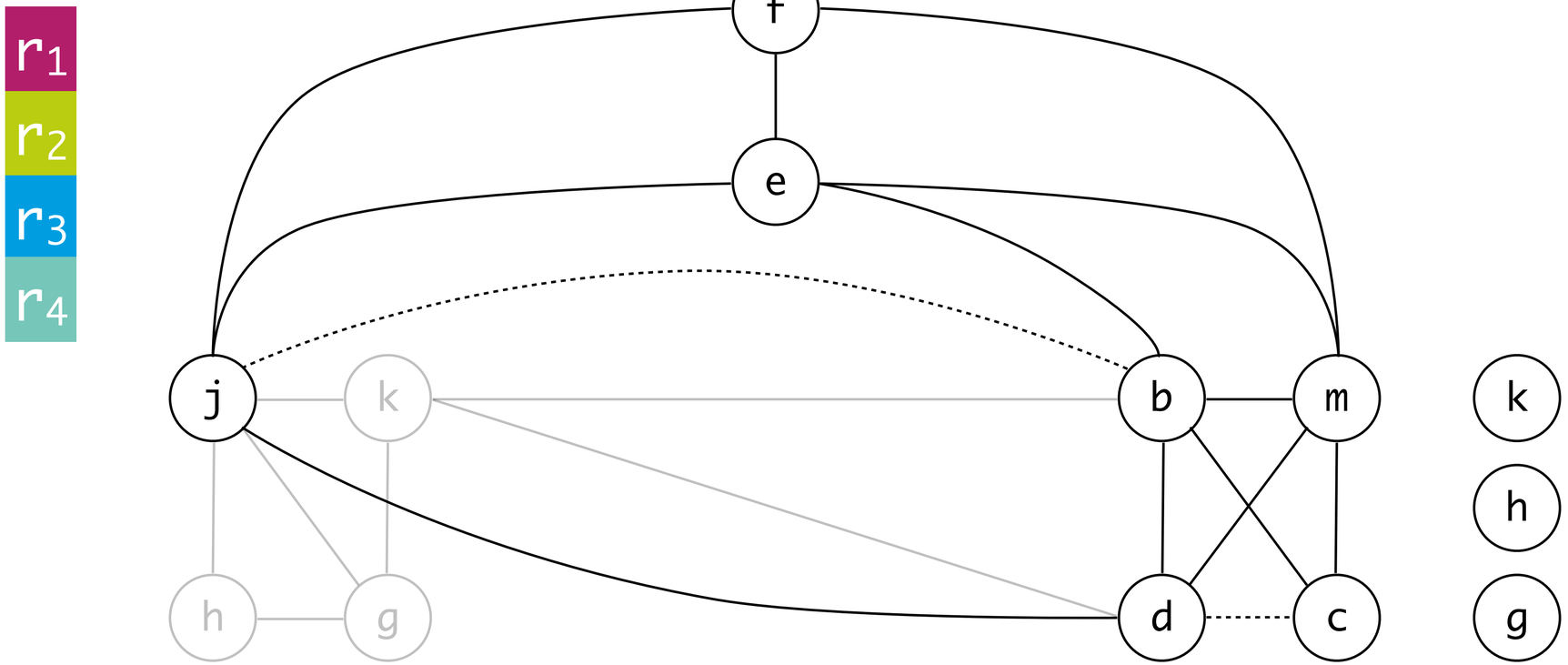
Coalescing

```
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
```



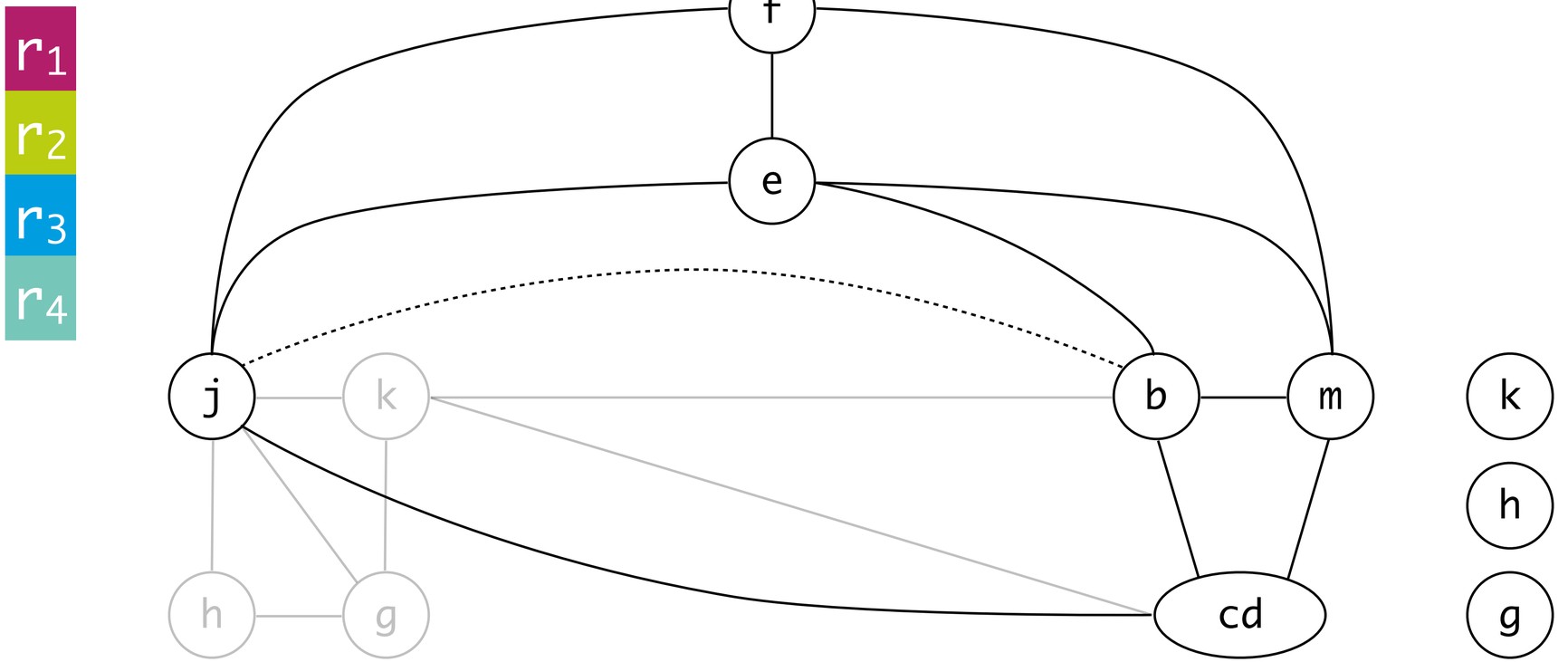
Coalescing

```
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
```



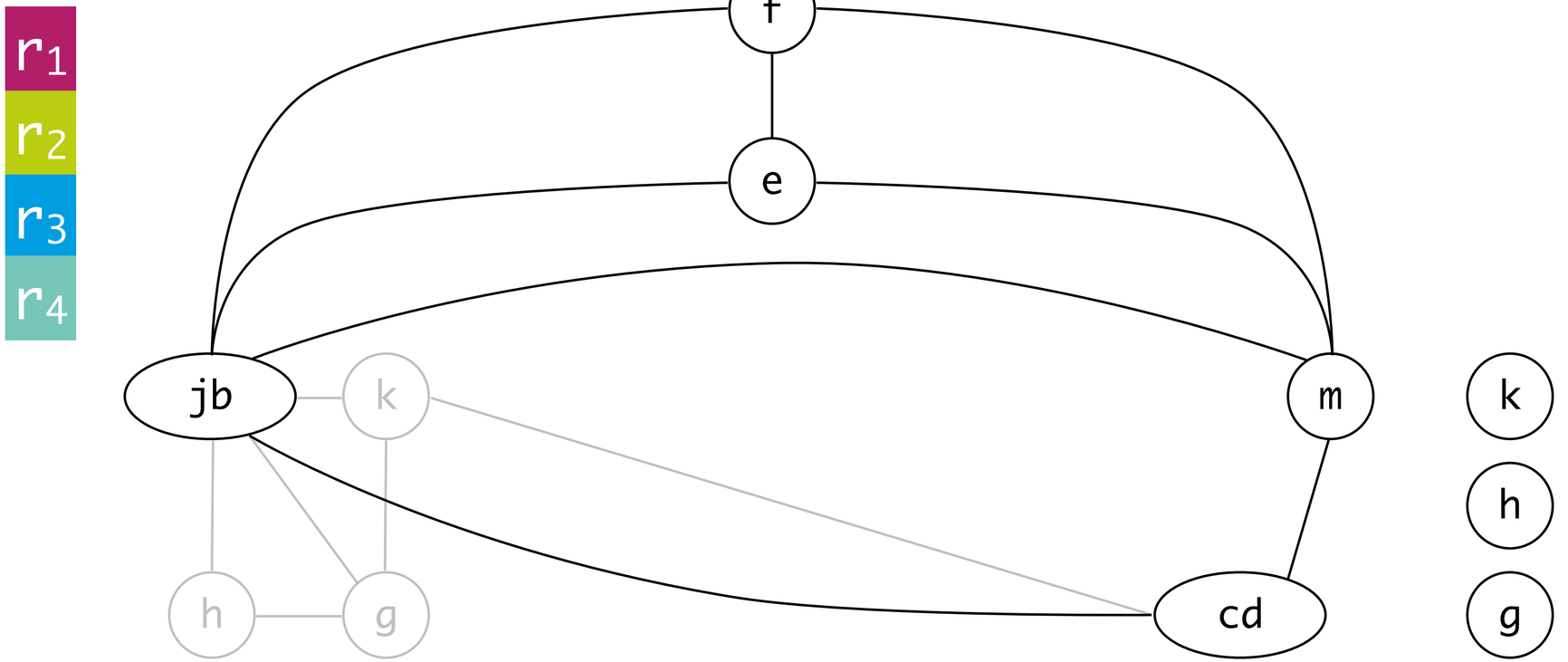
Coalescing

```
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
```



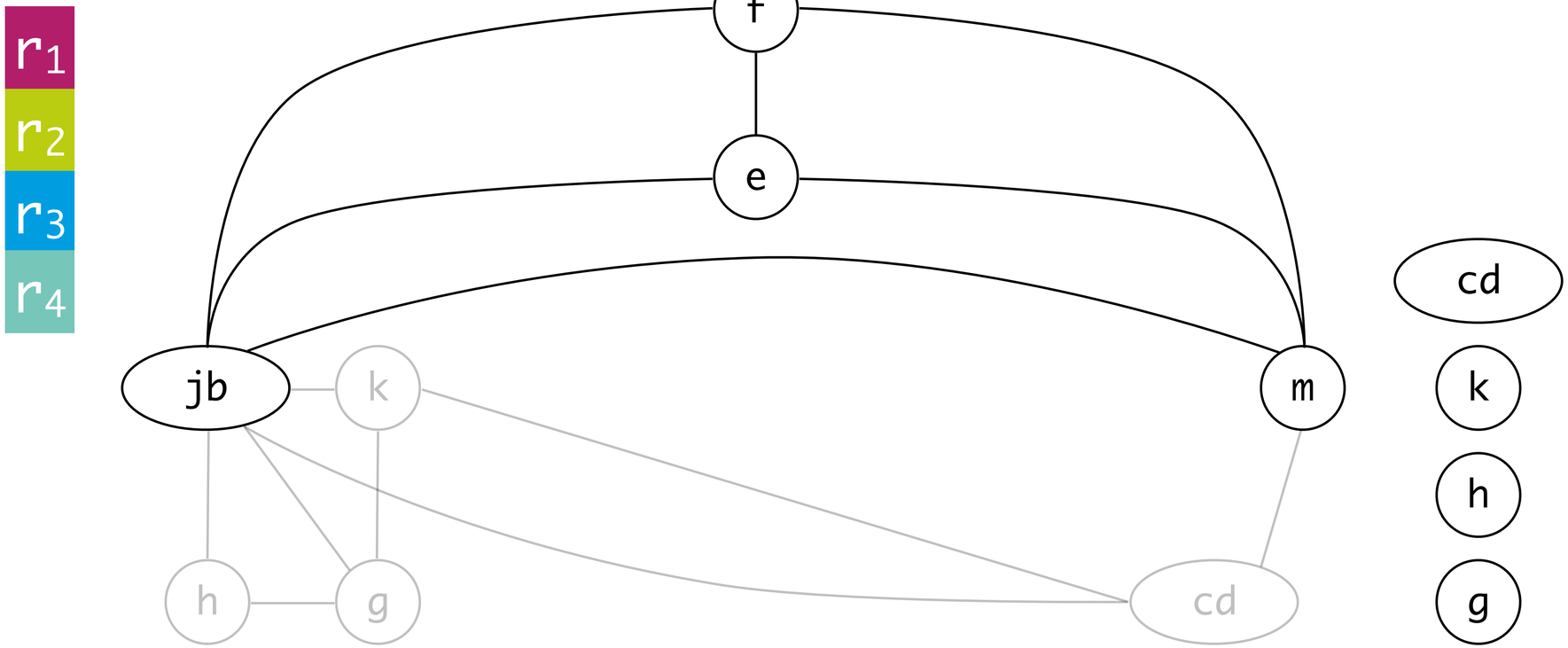
Coalescing

```
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
```

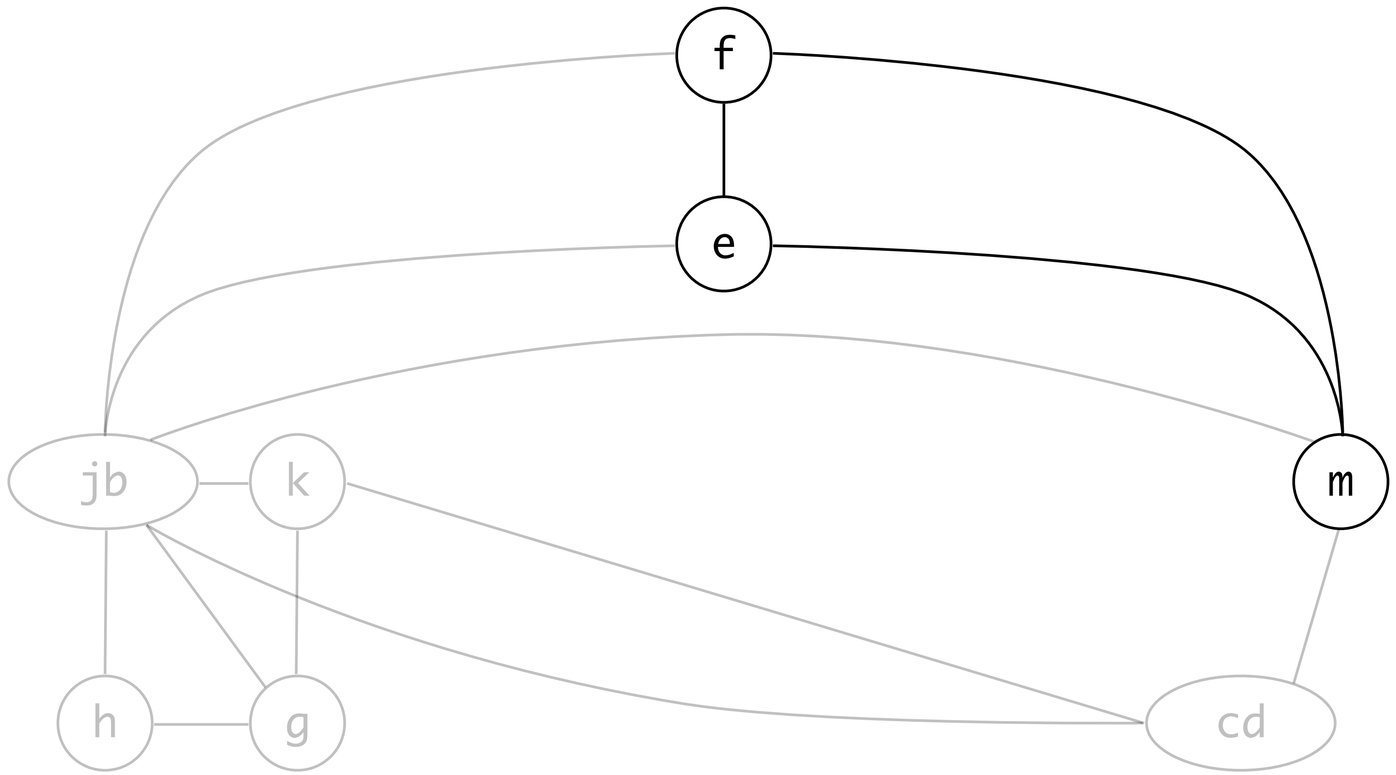


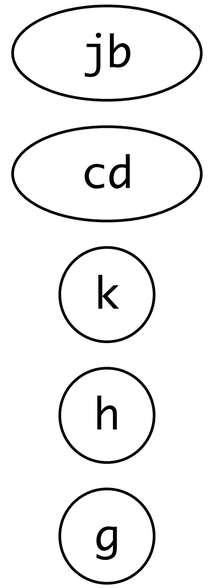
Coalescing

```
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
```



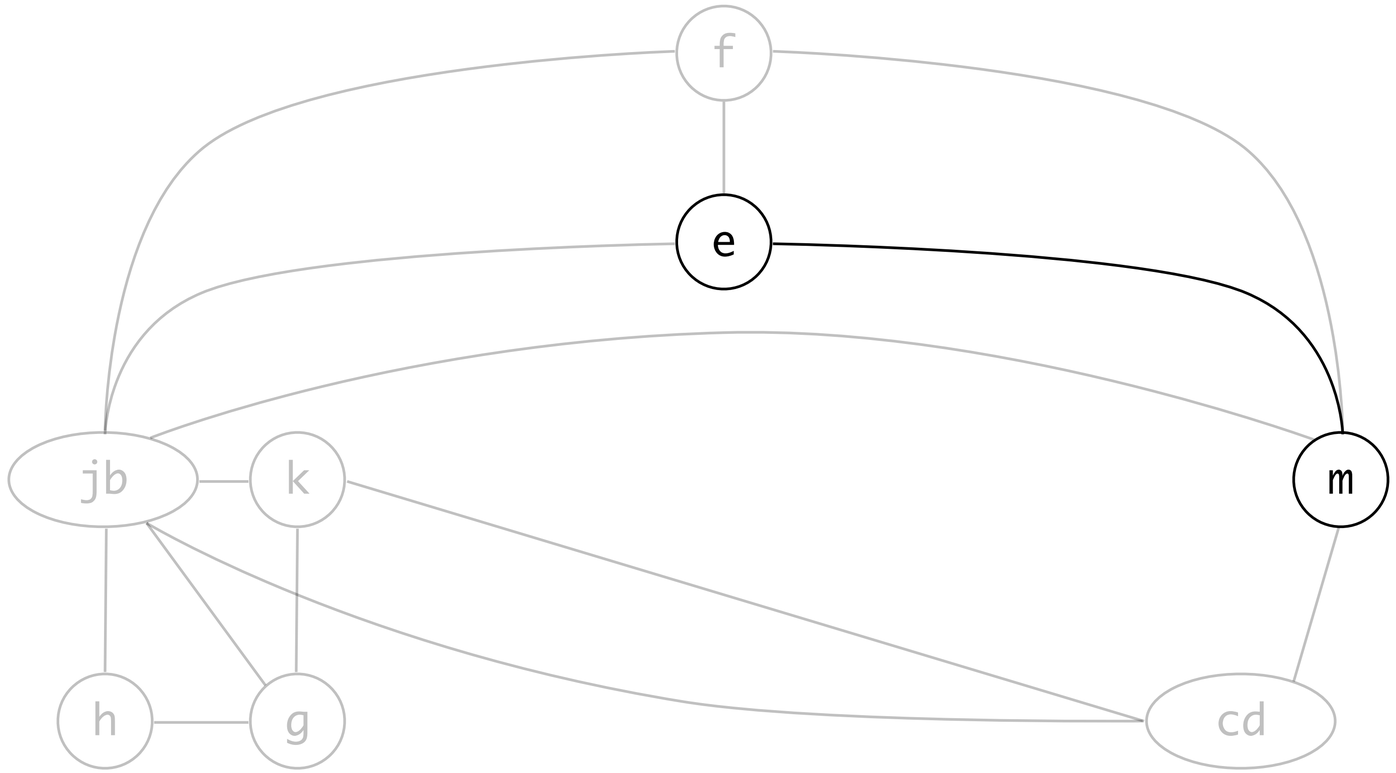
Coalescing

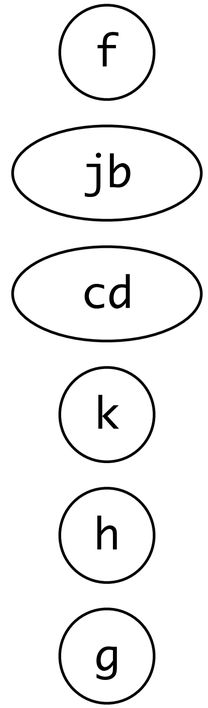





```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

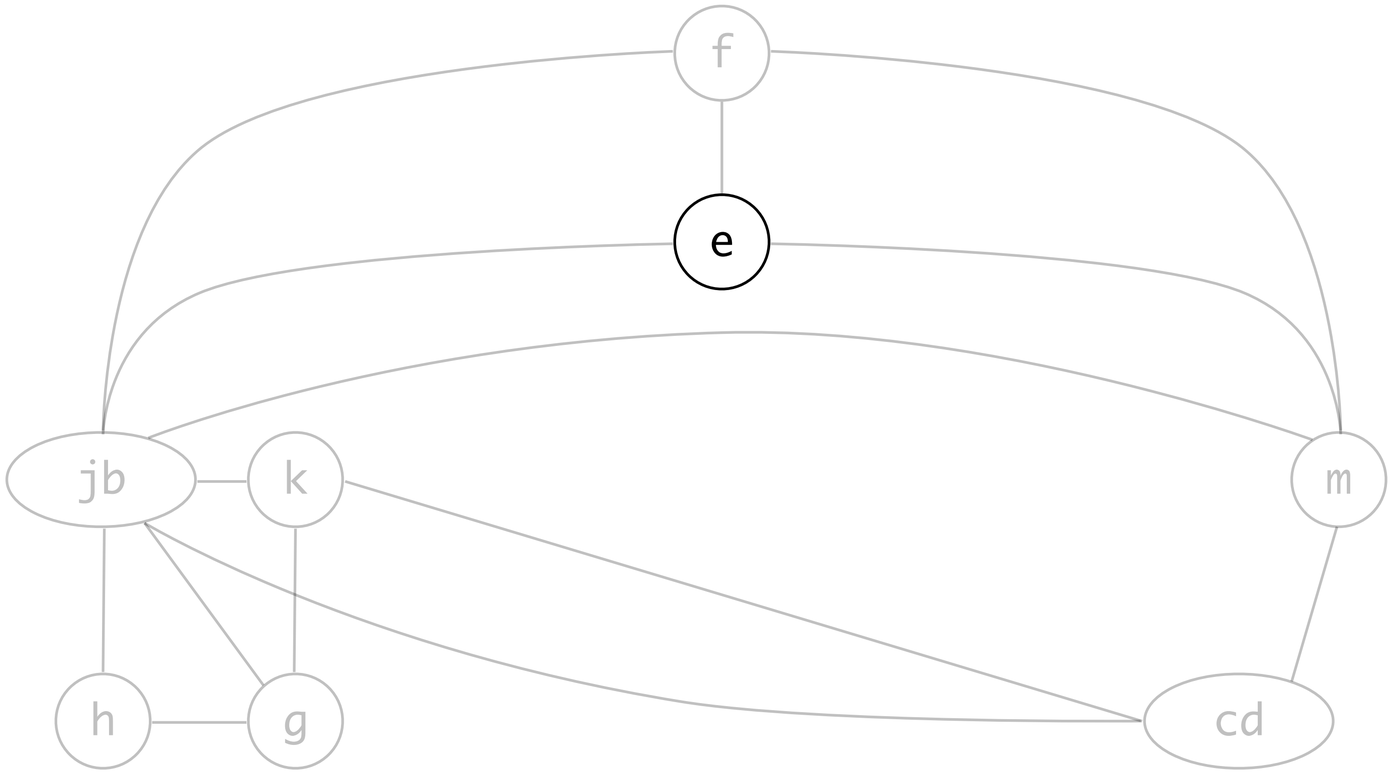
Coalescing

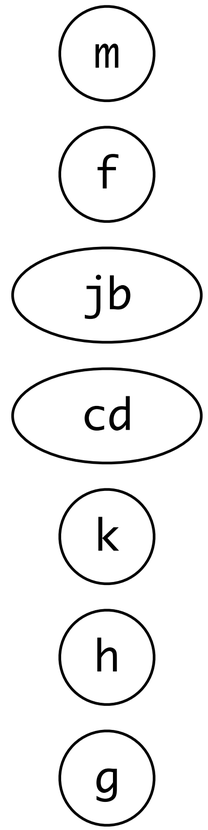





```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

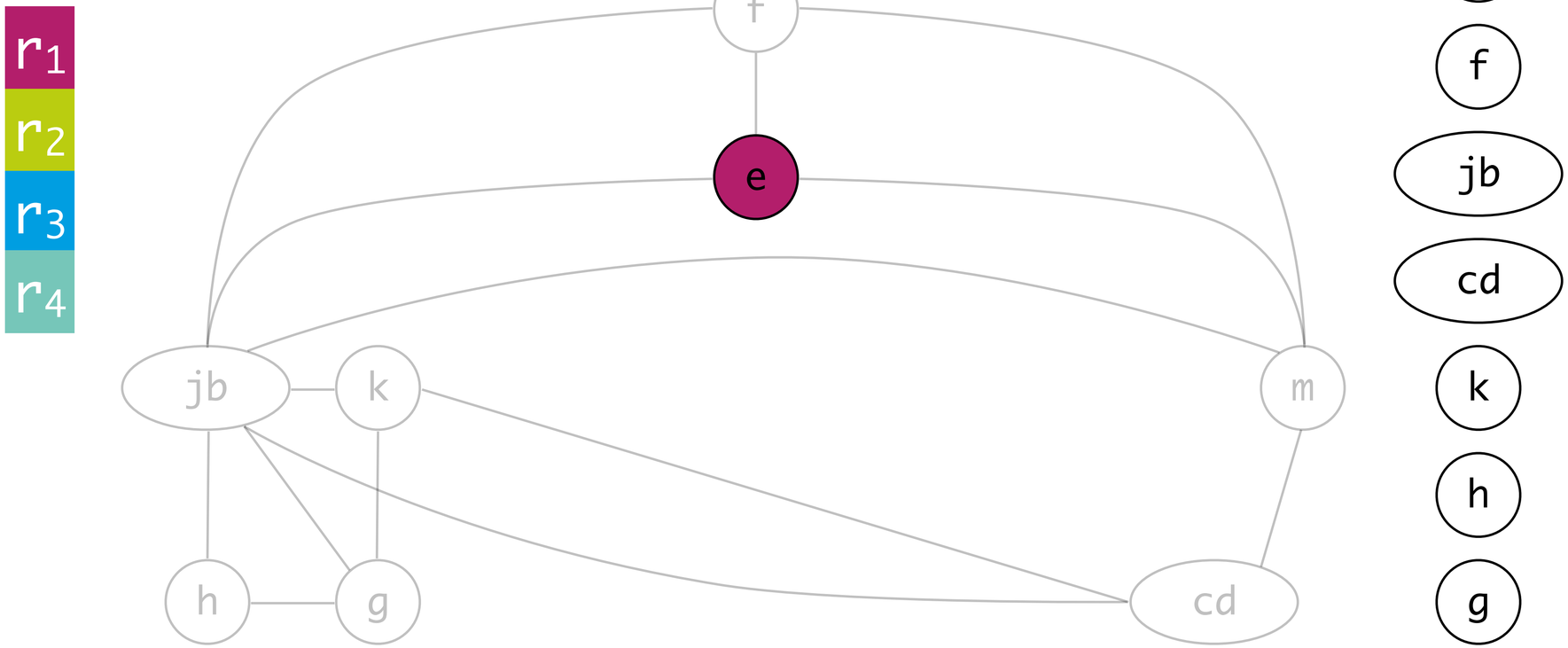
Coalescing





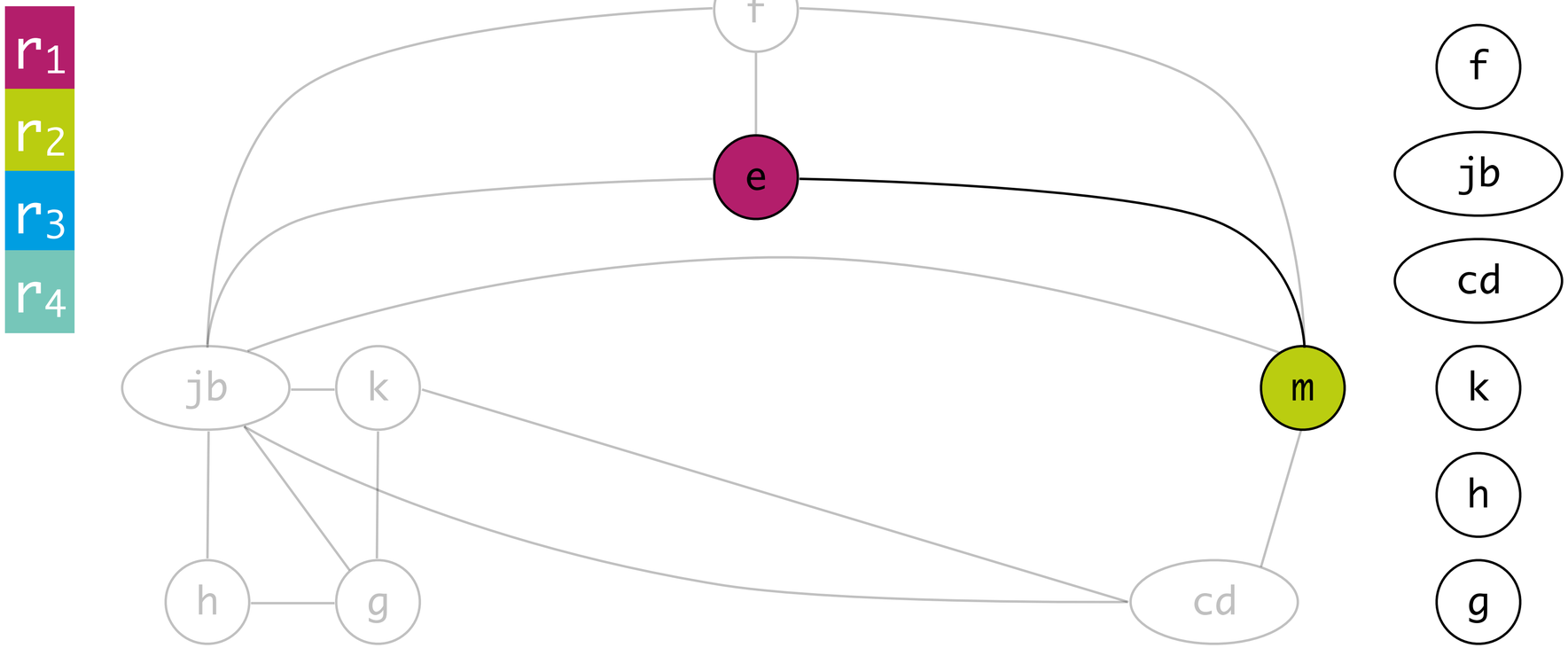

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
e := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



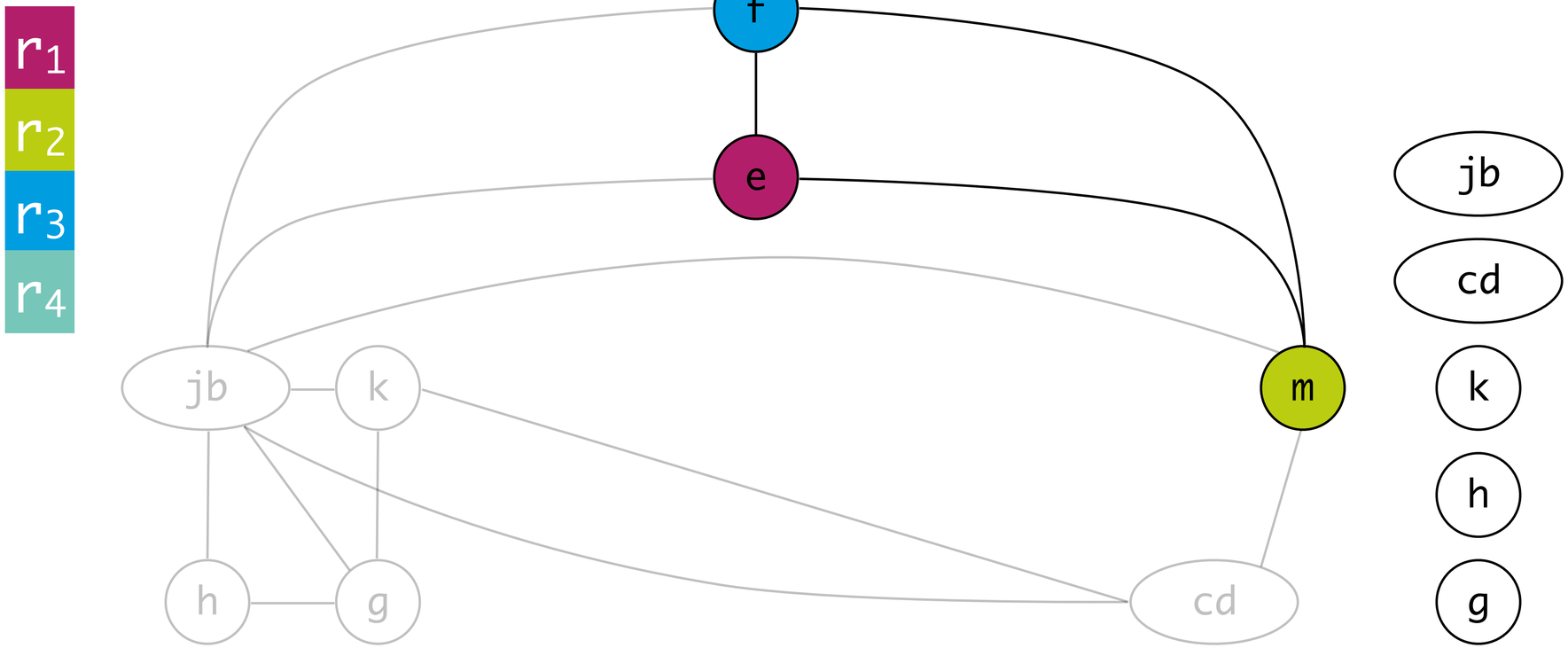

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
r1 := mem[j + 8]
m := mem[j + 16]
b := mem[f]
c := r1 + 8
d := c
k := m + 4
j := b
live out: d k j
```

Coalescing



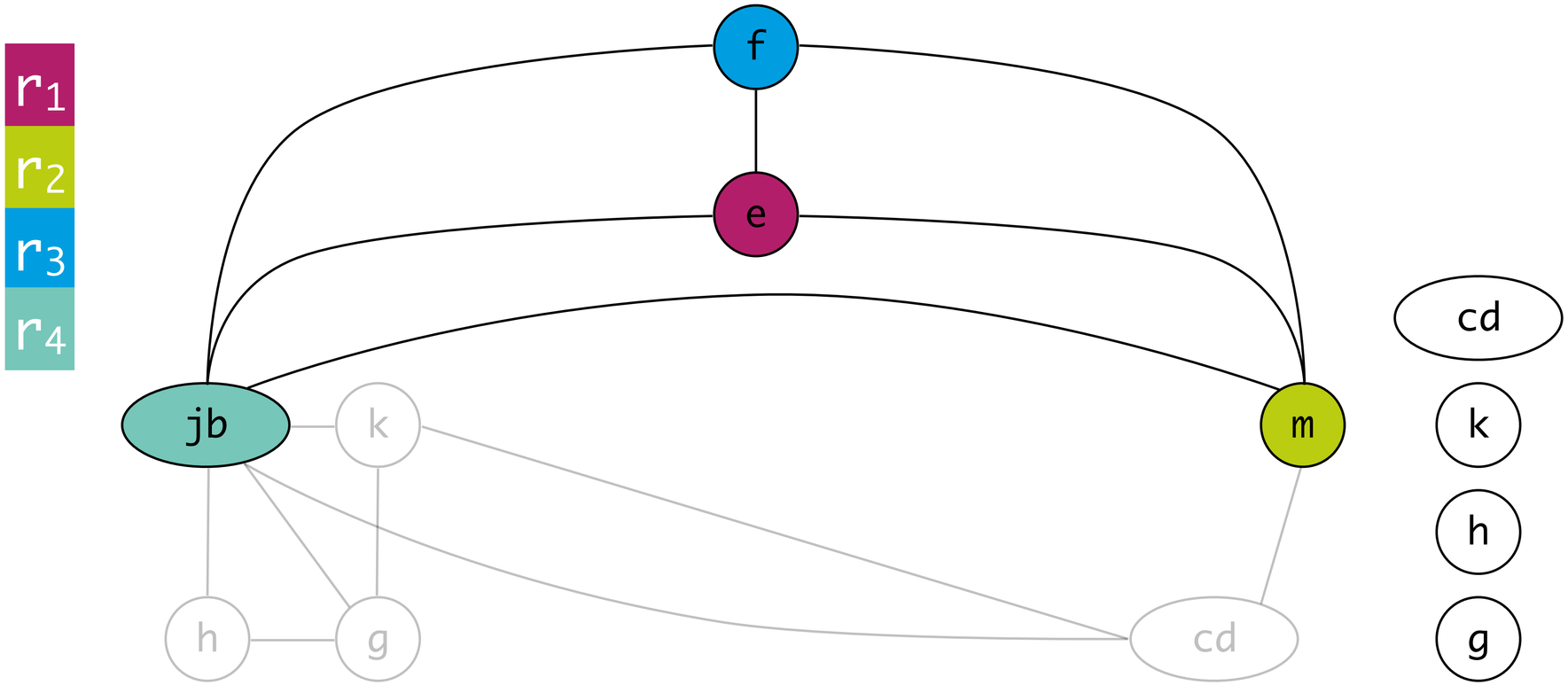

```
live-in: k j
g := mem[j + 12]
h := k - 1
f := g * h
r1 := mem[j + 8]
r2 := mem[j + 16]
b := mem[f]
c := r1 + 8
d := c
k := r2 + 4
j := b
live out: d k j
```

Coalescing



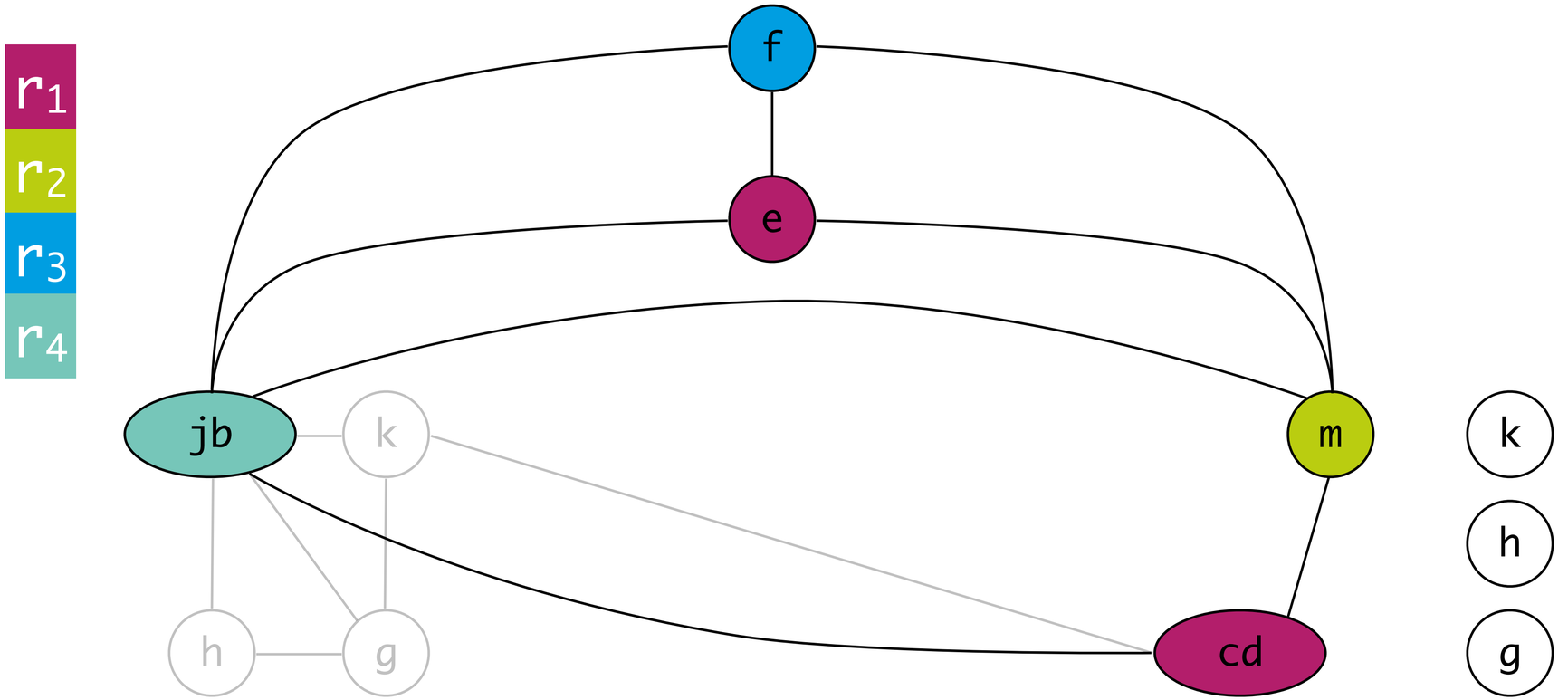

```
live-in: k j
g := mem[j + 12]
h := k - 1
r3 := g * h
r1 := mem[j + 8]
r2 := mem[j + 16]
b := mem[r3]
c := r1 + 8
d := c
k := r2 + 4
j := b
live out: d k j
```

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
```

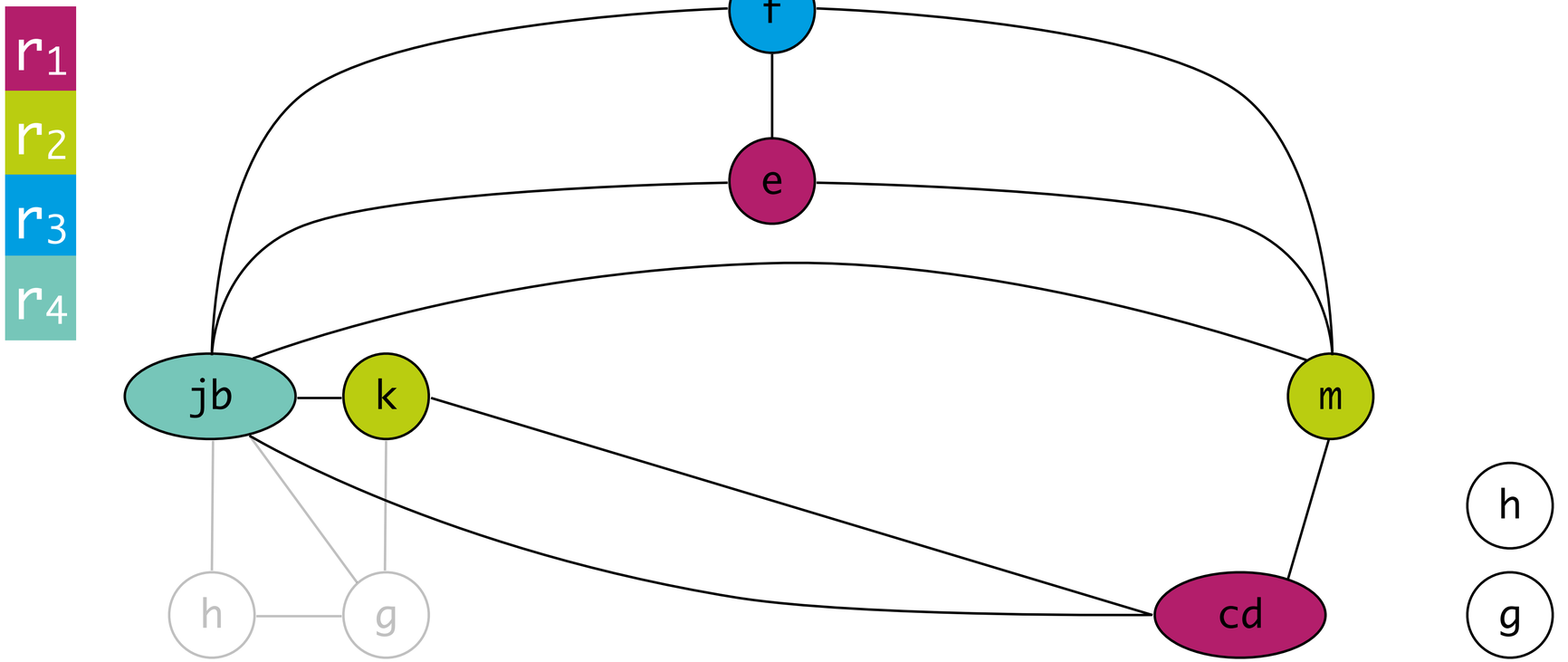
Coalescing



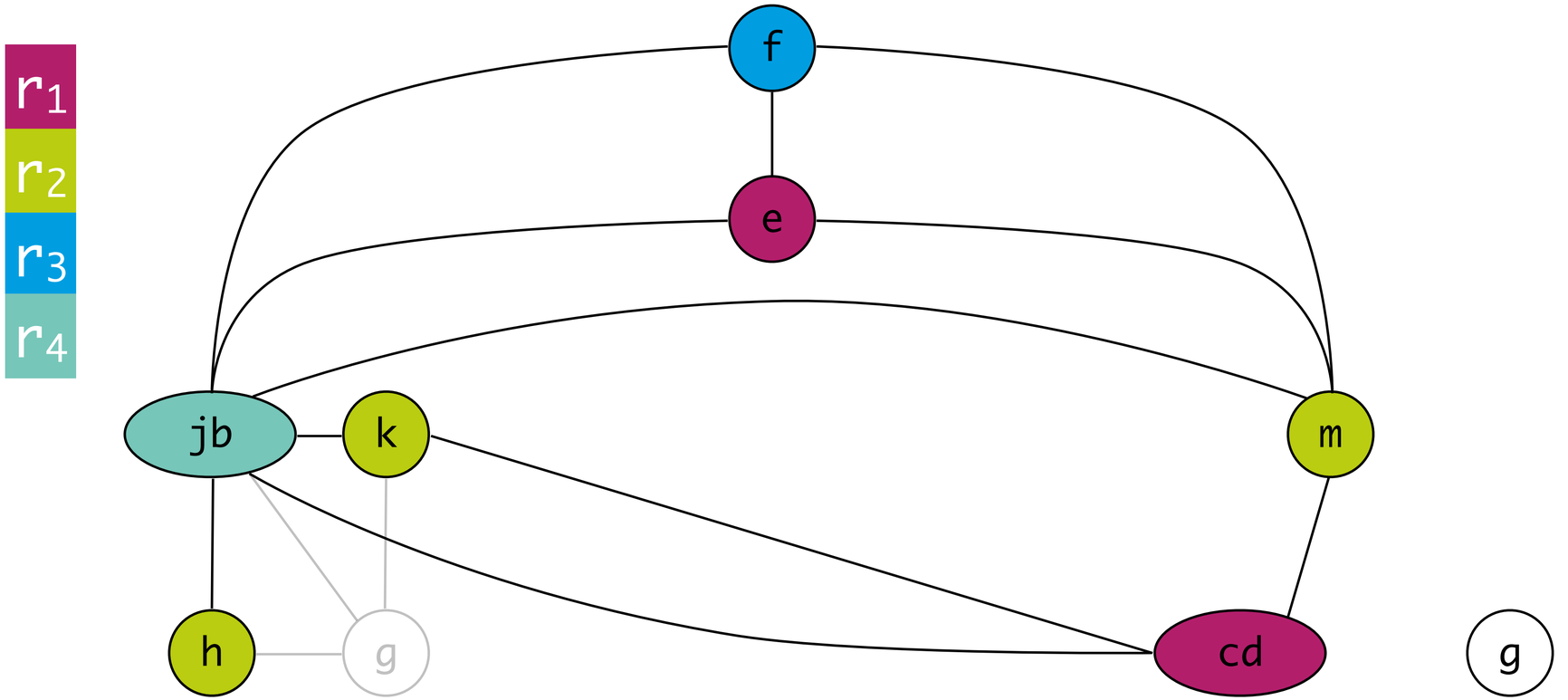

```
live-in: k r4
g := mem[r4 + 12]
h := k - 1
r3 := g * h
r1 := mem[r4 + 8]
r2 := mem[r4 + 16]
b := mem[r3]
r1 := r1 + 8
r1 := r1
k := r2 + 4
r4 := r4
live out: r1 k r4
```

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
```

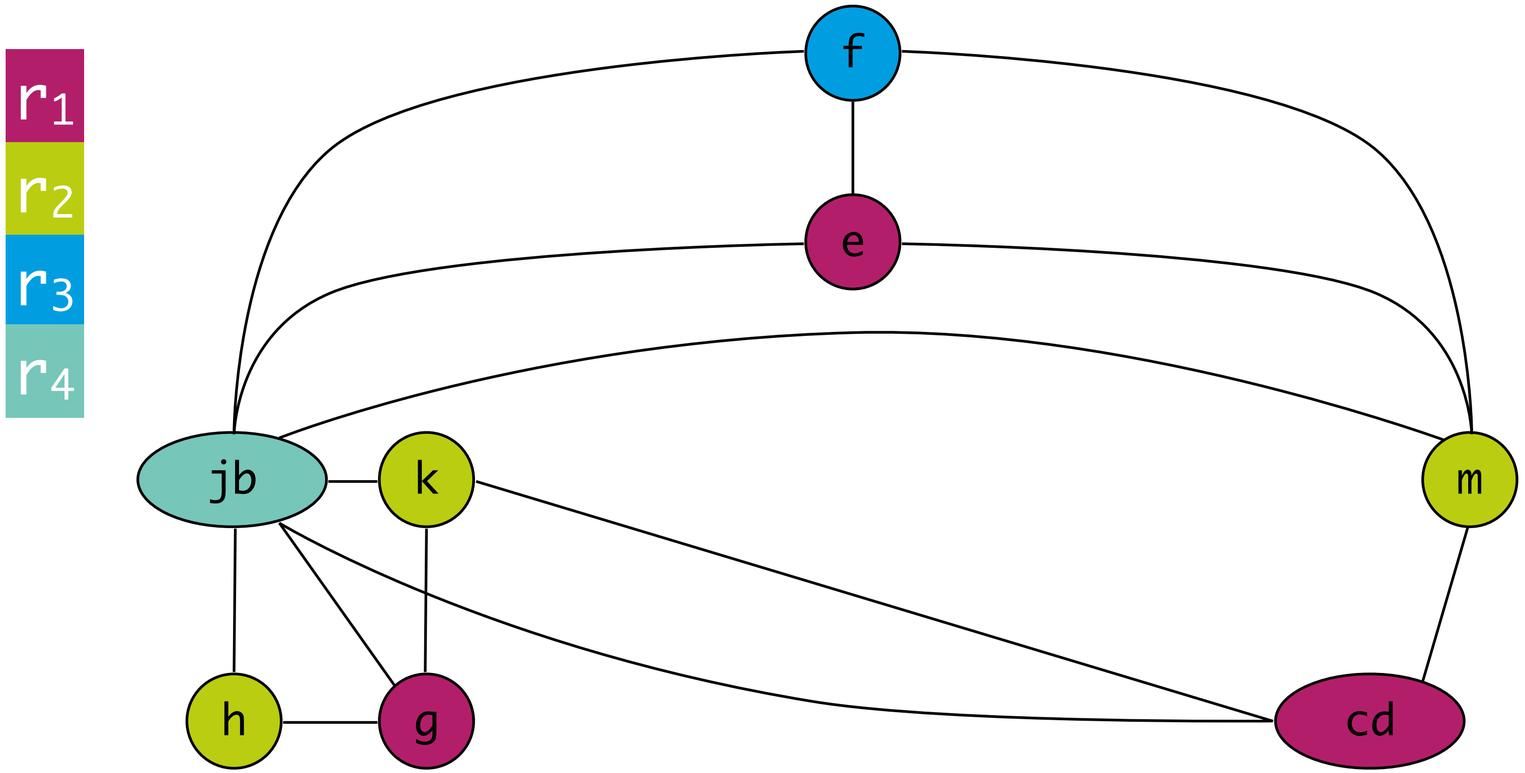


Coalescing




```
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)
```

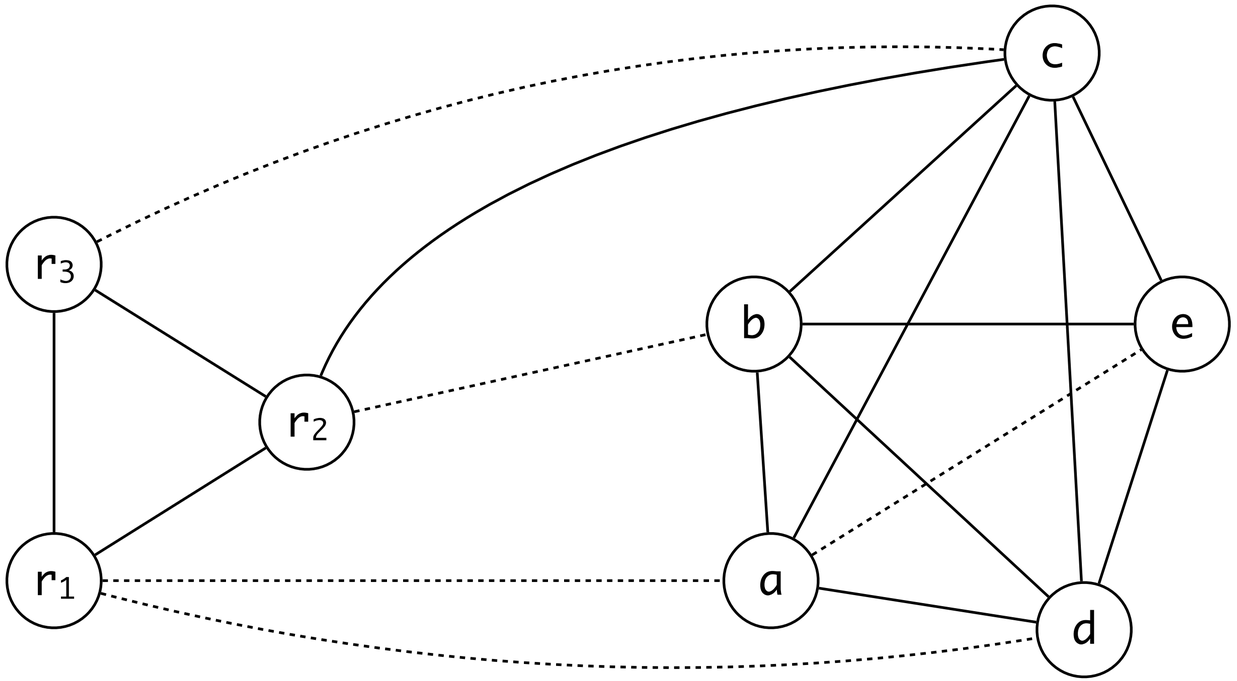
Pre-Colored Nodes

```
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

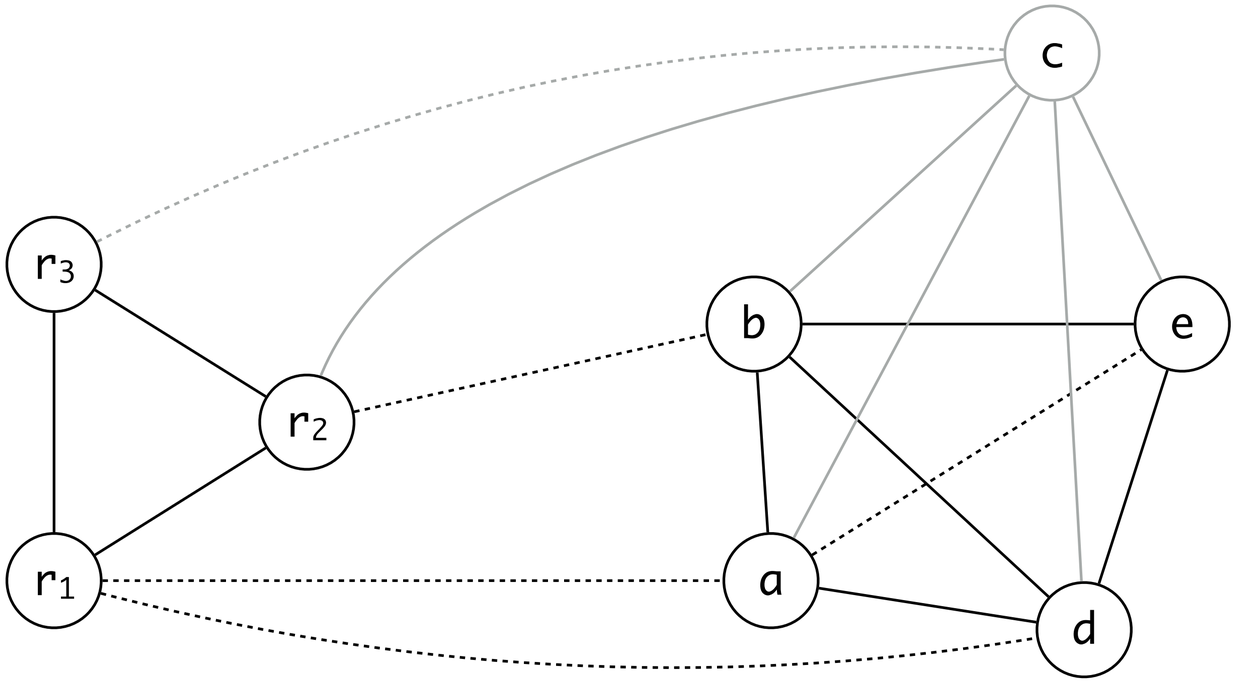


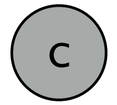

```
enter : c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return (r1, r3)
```

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

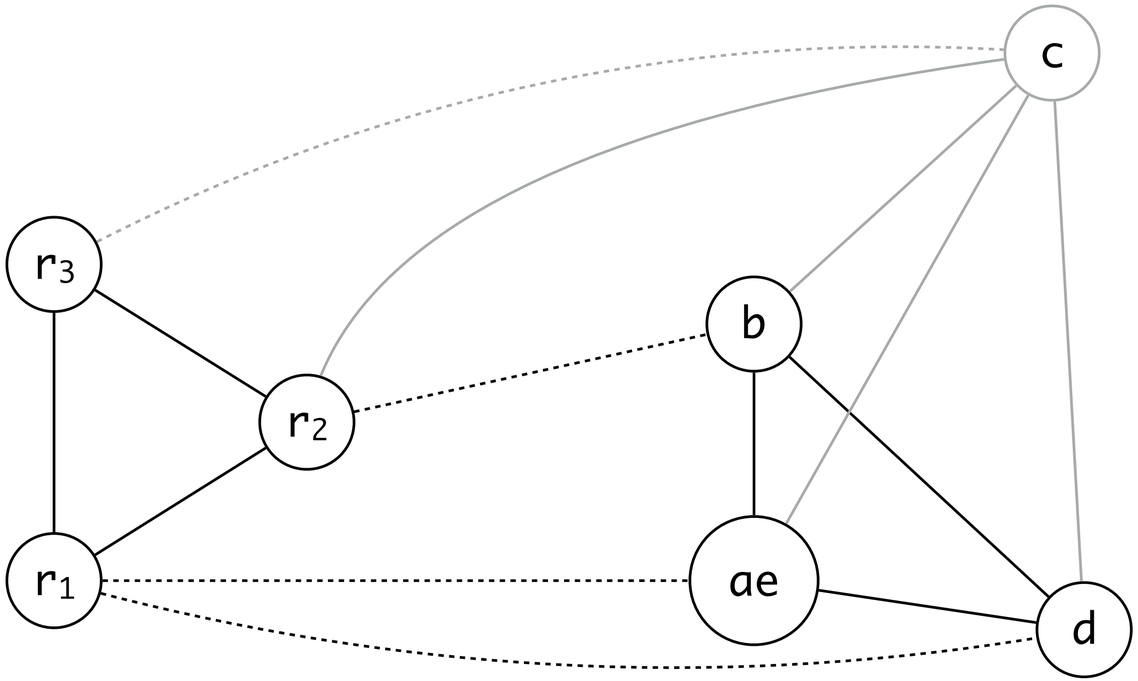


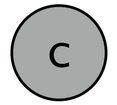


Pre-Colored Nodes

```
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)
```

coalesce a and e

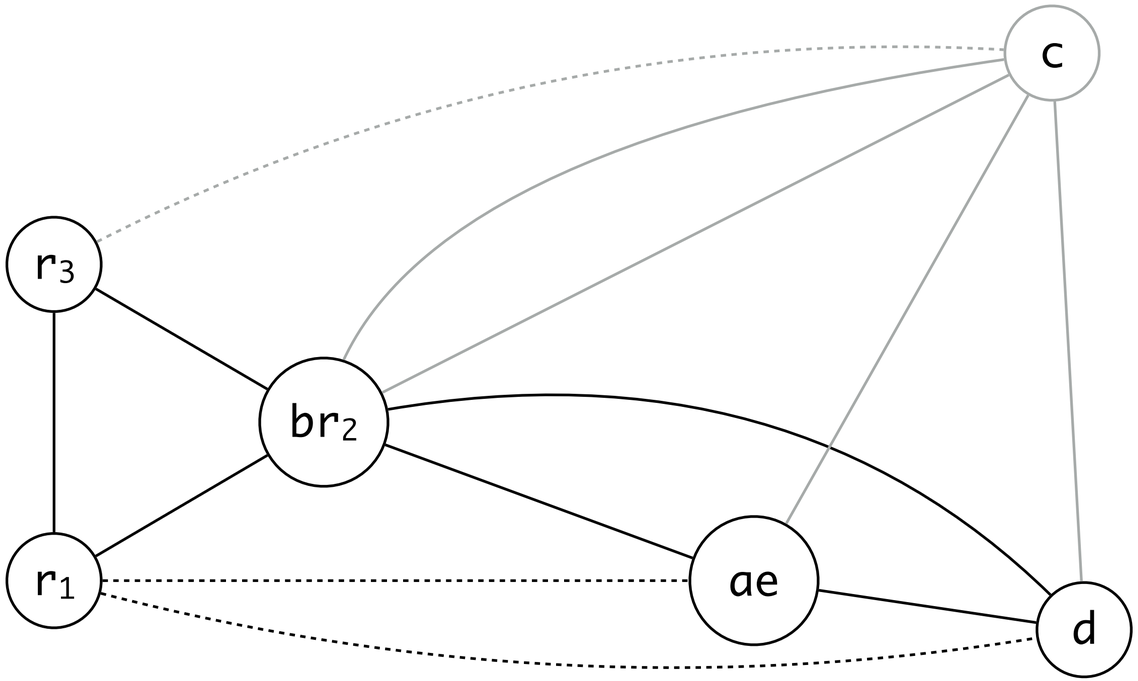


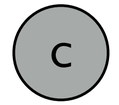


Pre-Colored Nodes

```
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)
```

coalesce r_2 and b

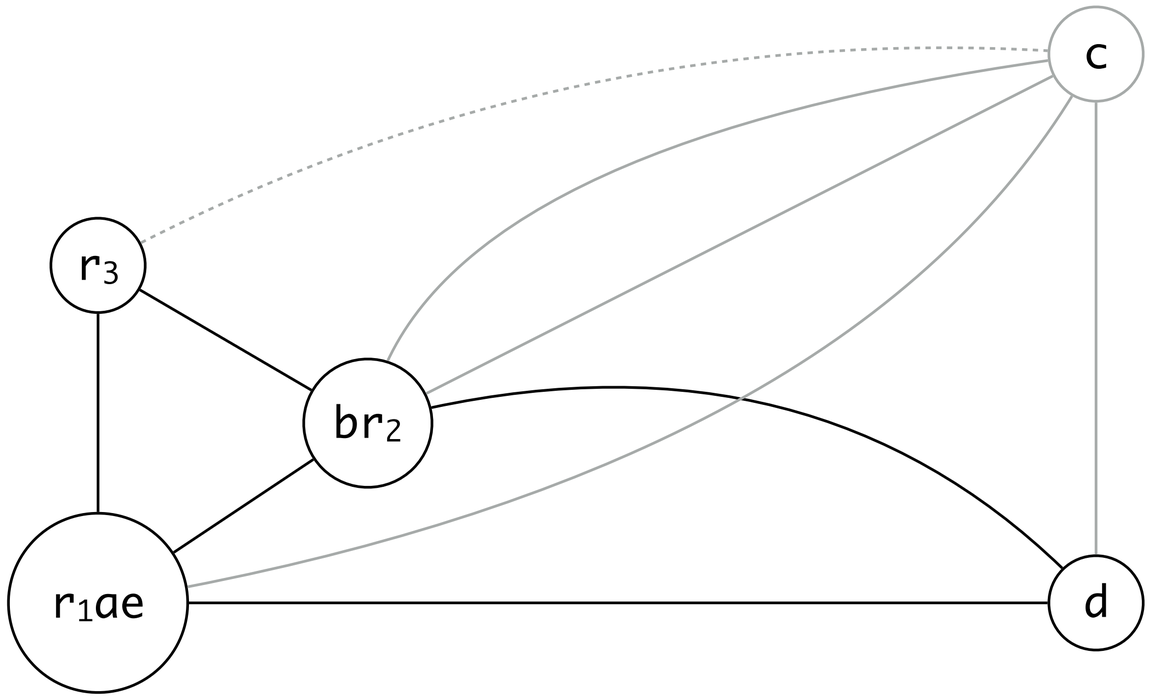


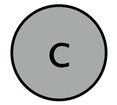


Pre-Colored Nodes

```
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)
```

coalesce r_1 and ae

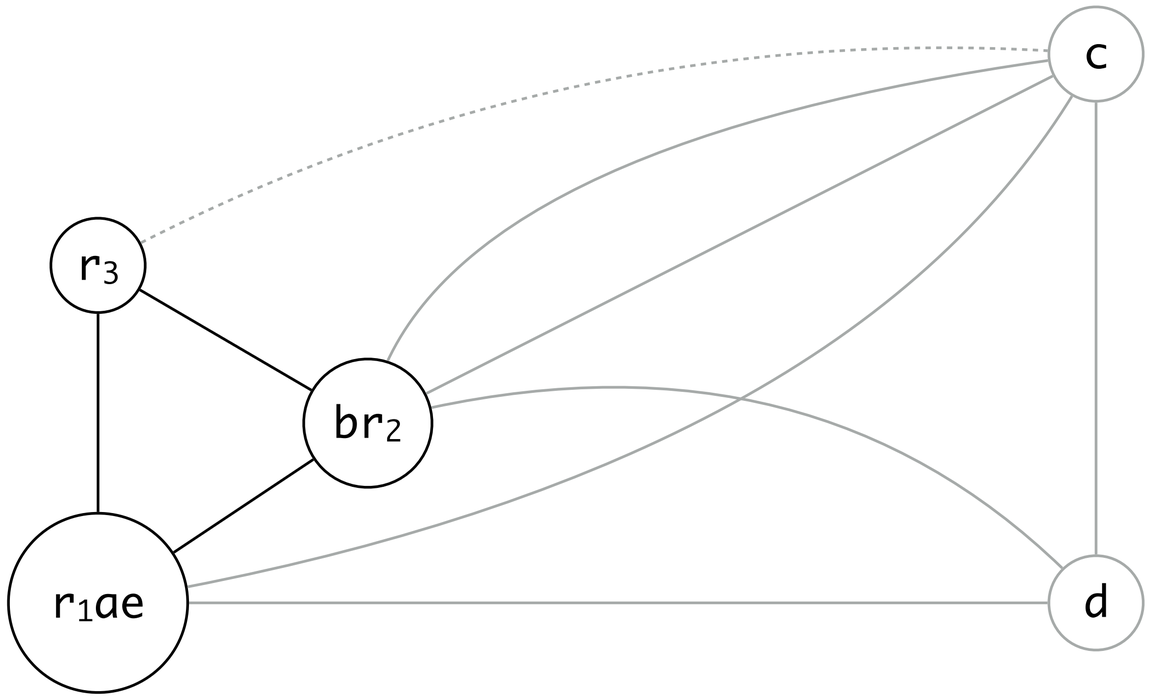


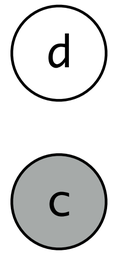


Pre-Colored Nodes

```
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)
```

simplify d

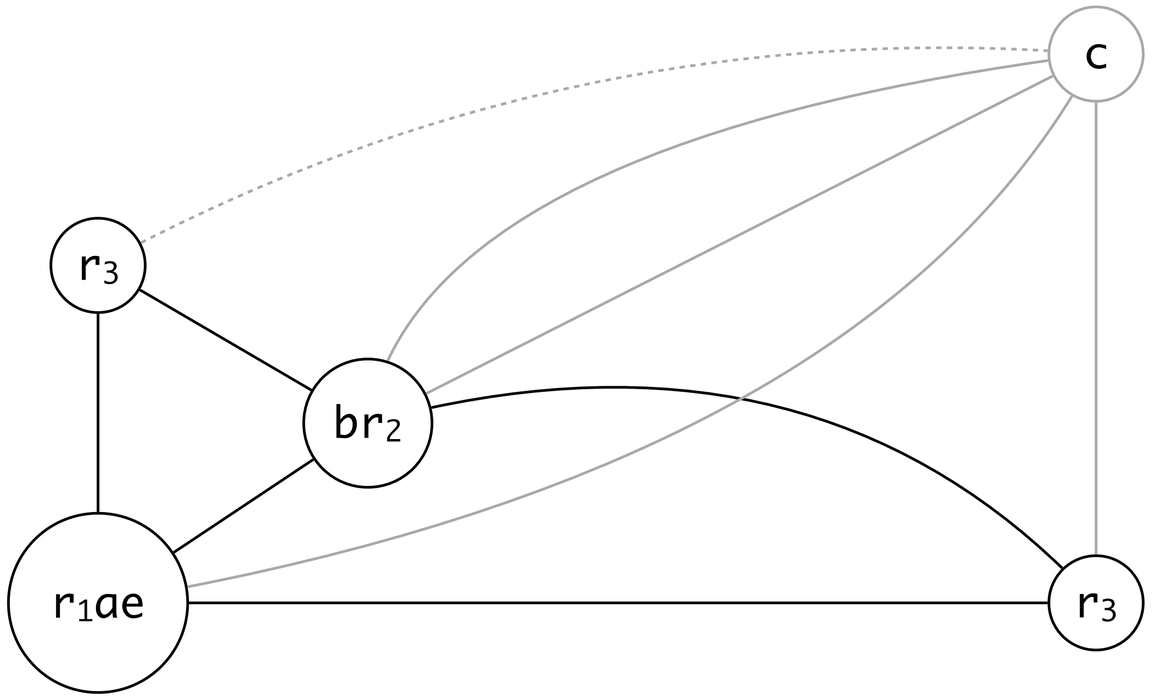


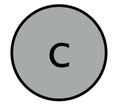


Pre-Colored Nodes

```
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)
```

color d as r_3

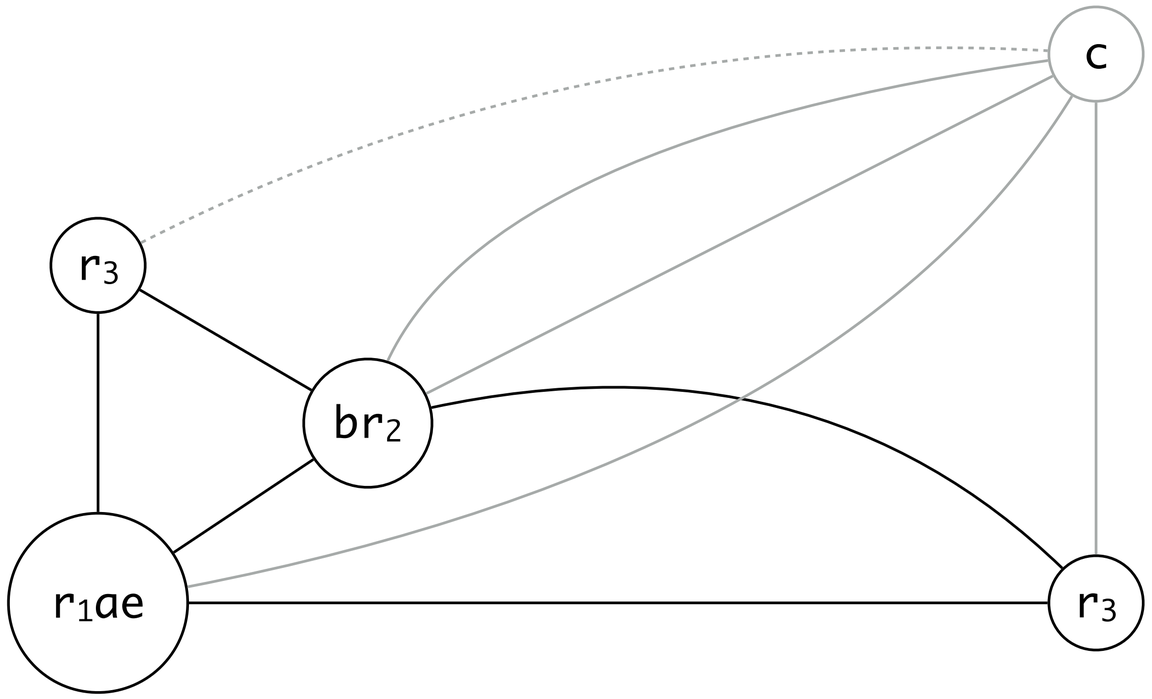


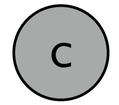


Pre-Colored Nodes

```
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)
```

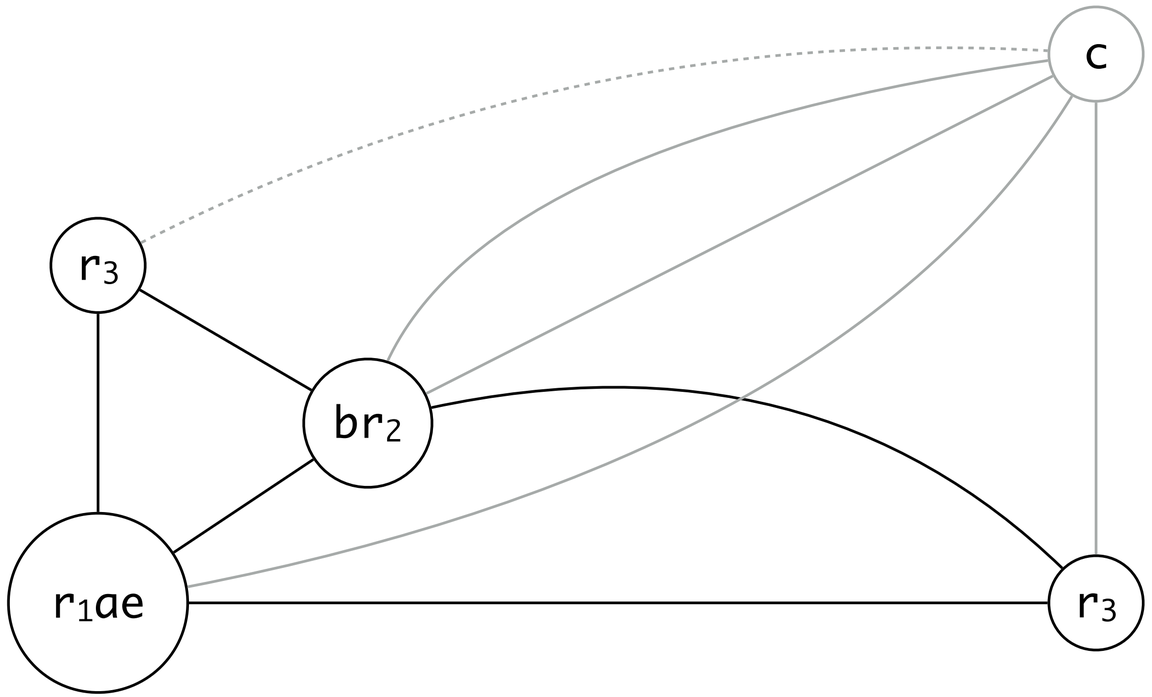
spill c





Pre-Colored Nodes

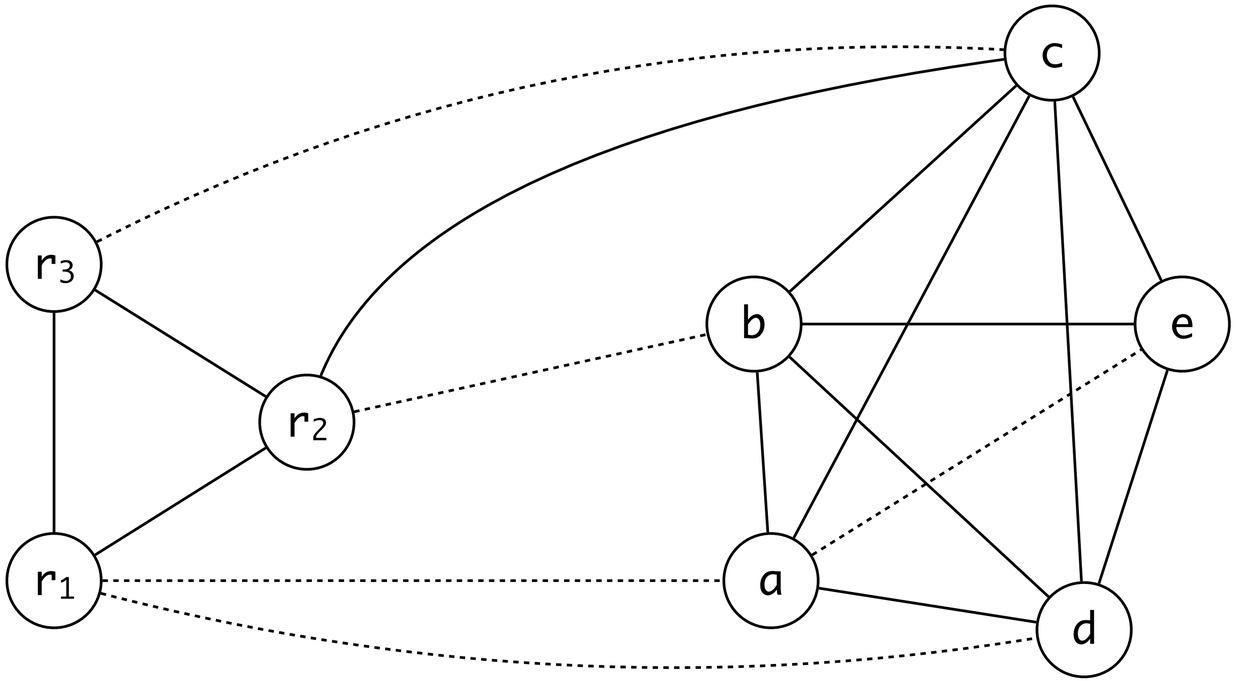
spill c




```
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)
```

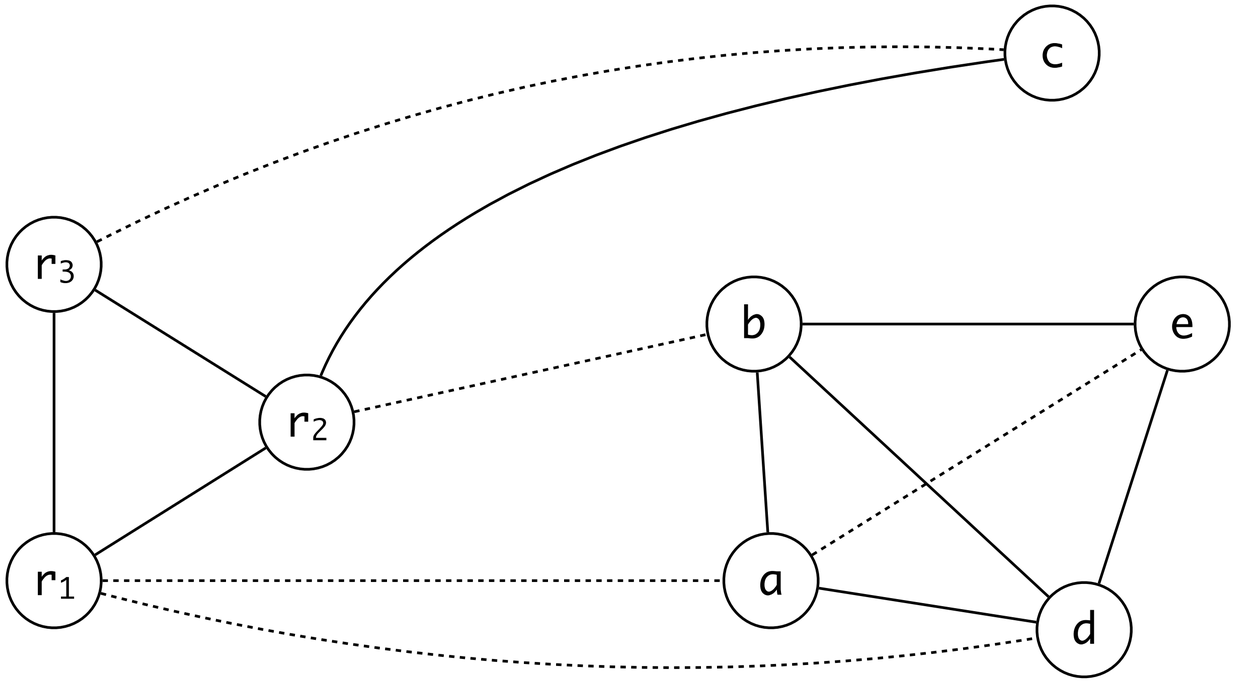
Pre-Colored Nodes

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)
```

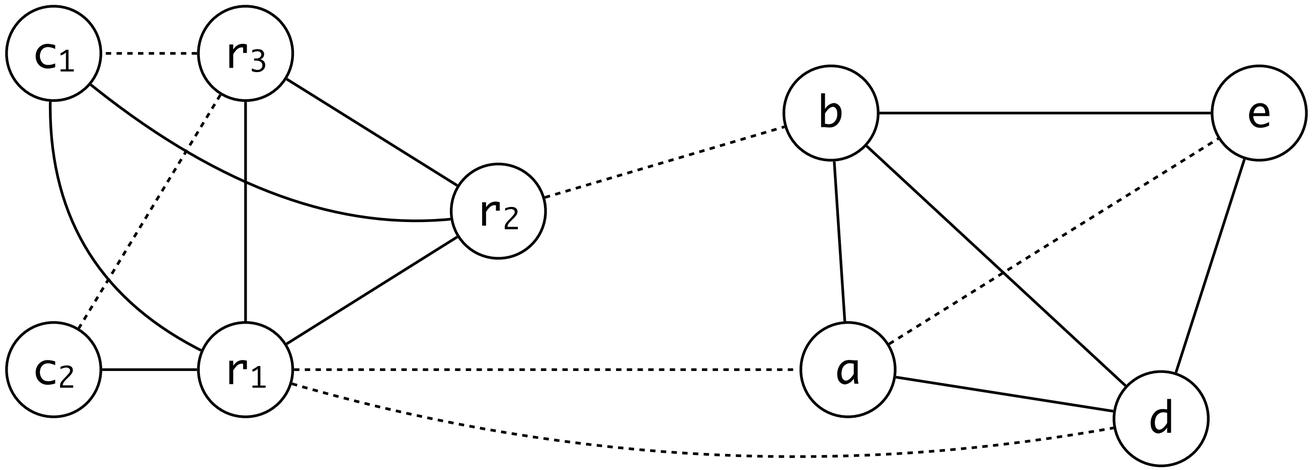
Pre-Colored Nodes




```
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

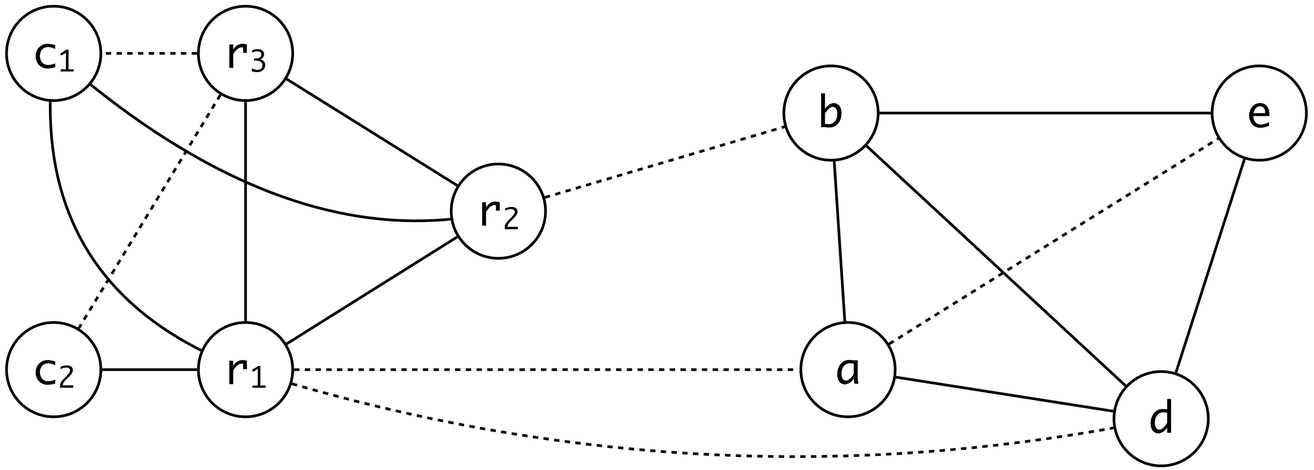
new graph




```
enter : c1 ← r3
        M[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

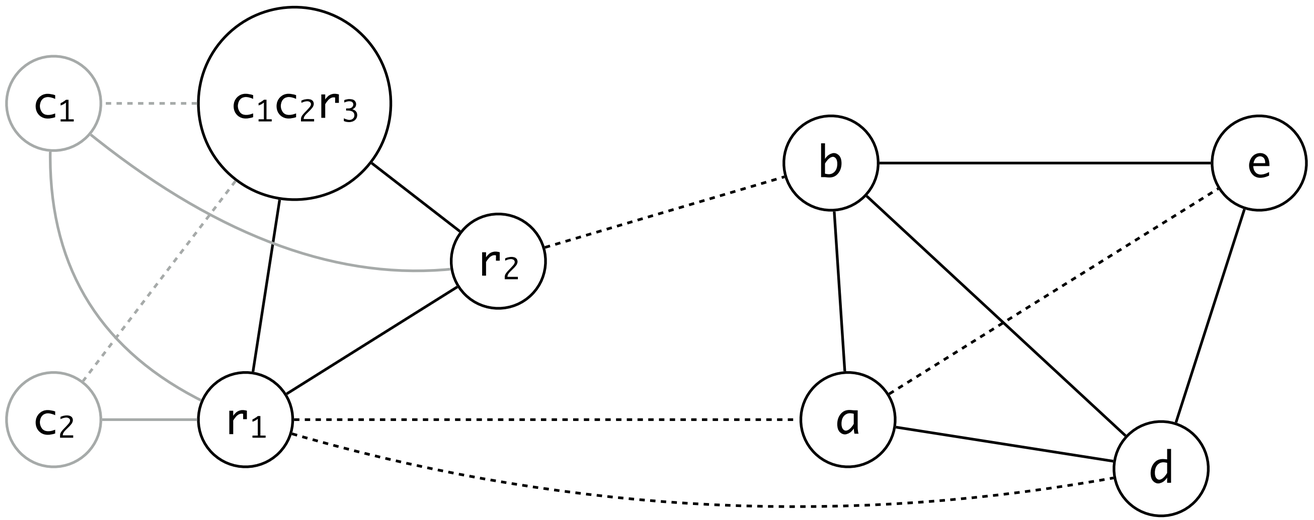
coalesce c_1, c_2, r_3




```
enter : c1 ← r3
        M[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

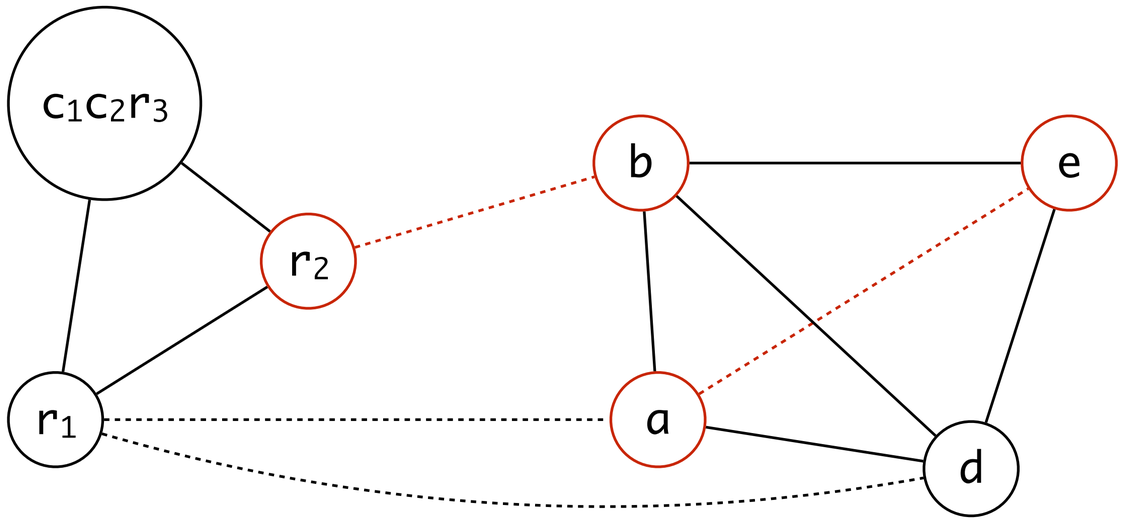
coalesce c_1, c_2, r_3




```
enter : c1 ← r3
        M[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

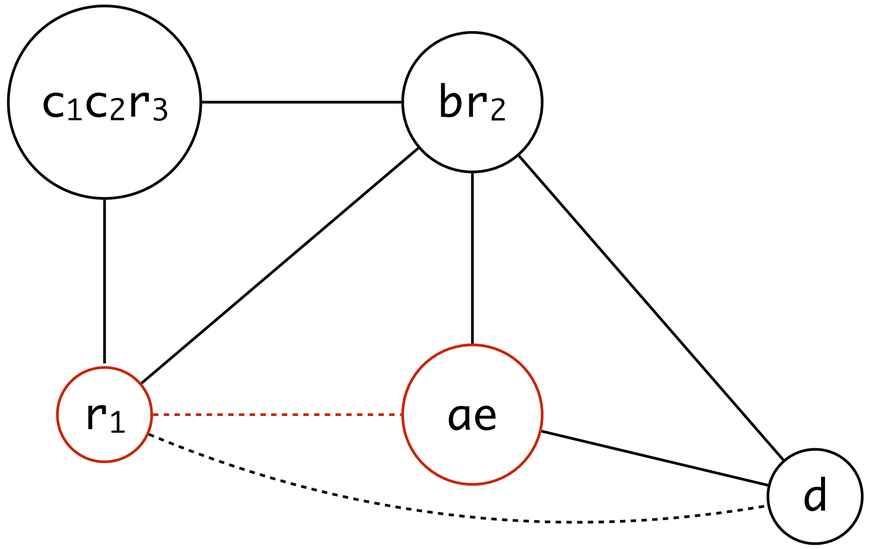
coalesce (b, r_2) and (a, e)




```
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)

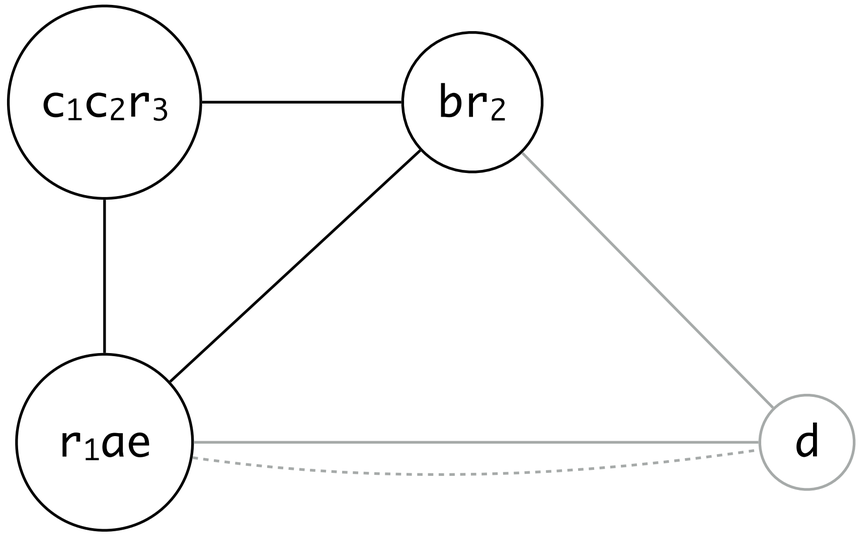


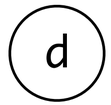

```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```

Pre-Colored Nodes

```
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)
```

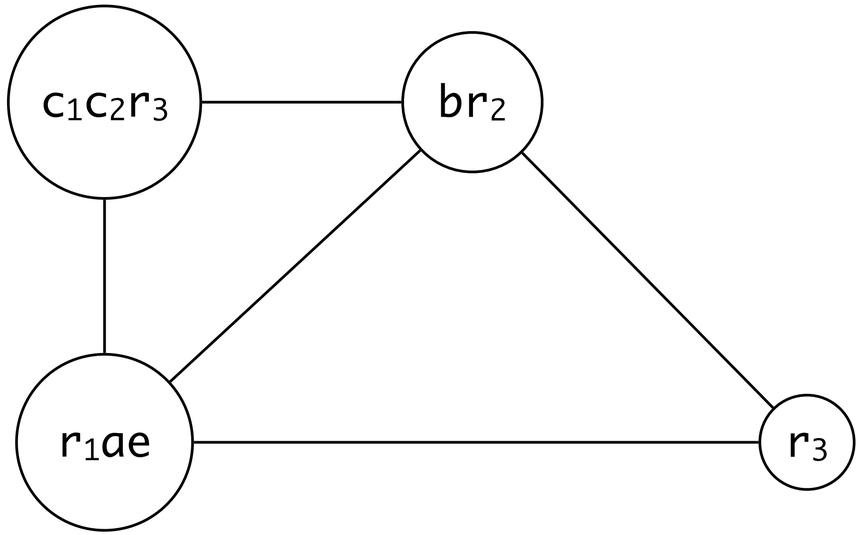
simplify d



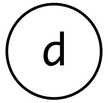


Pre-Colored Nodes

color d as r_3

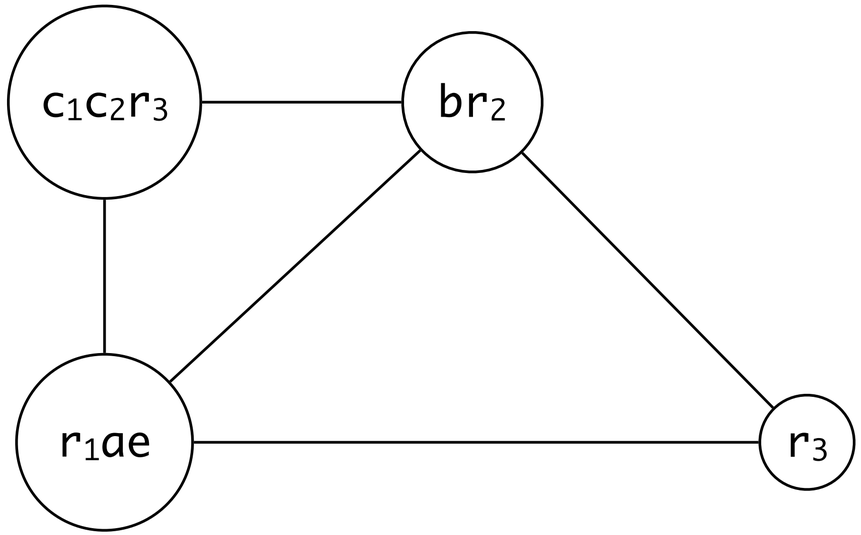



```
enter : c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e - 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c2
        c2 ← M[cloc]
        return (r1, r3)
```



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[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[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