Compilers I

Register Allocation I

Dr. William Harrison
harrisonwl@missouri.edu

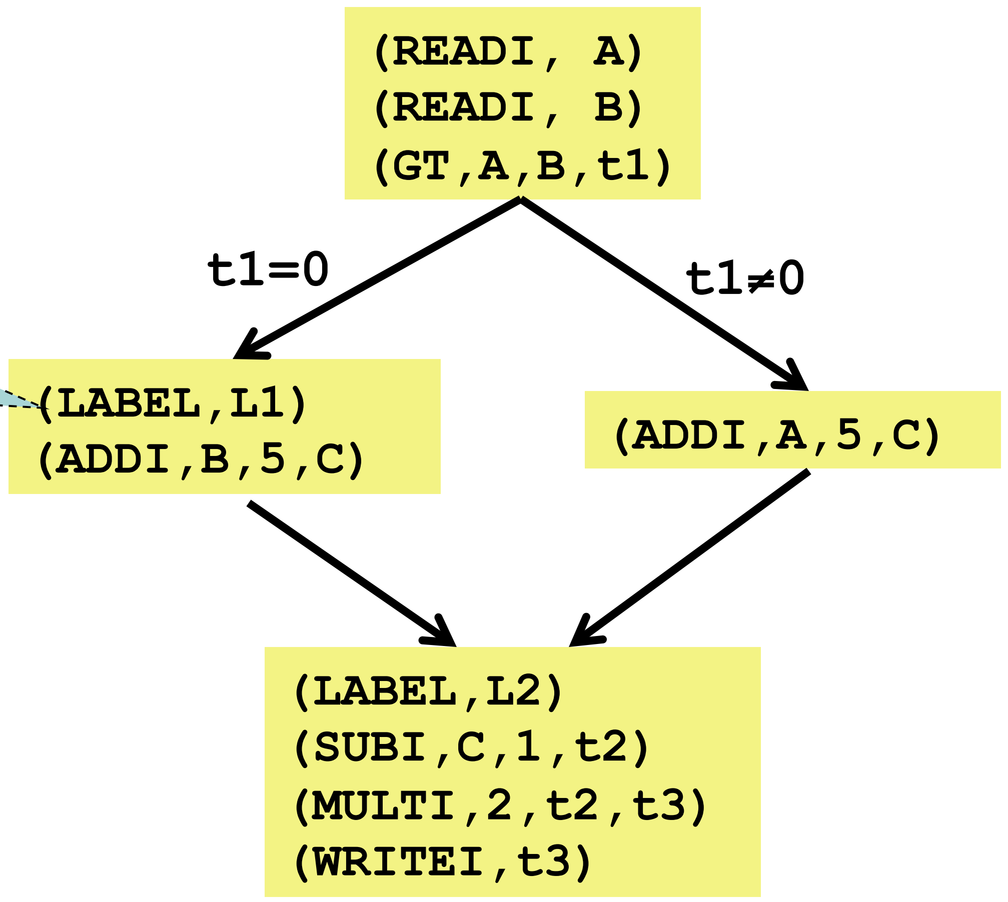
Today's Class

- Register Allocation
  - Review: Liveness Analysis
  - Register Allocation
    - Review: constructing the interference graph
    - k-coloring the interference graph
      - if possible
      - approximating *k*-coloring

Graphical Representation

Control Flow Graph (CFG)

Basic Block: *linear sequences of tuples containing no branches until the end. Such branches are usually represented as arrows.*



Virtual Registers

Variables in the IR are sometimes referred to as “virtual registers”

```
   a ← 0
L: b ← a + 1
   c ← c + b
   a ← b * 2
   if (a < N) goto L
   return c
```

* registers on a microprocessor are called “physical registers”
** virtual registers sometimes distinguished with a “$” – e.g., “$a”

Register allocation answers the question

```
   a ← 0
L: b ←a + 1
   c ← c + b
   a ←b * 2
   if (a < N) goto L
   return c
```

Should we store **c** in a physical register or on the run-time stack?

Liveness Analysis

- …determines when the value within a virtual register may still be used
  - a.k.a. its value is “live”
- …and when it won’t
  - a.k.a. its value is “dead”
- This property, “liveness”, may be approximated statically

Some Flow Graph Definitions

Flow graph terminology

pred[N] = set of predecessors.

  pred[2] = { 1, 5 }

succ[N] = set of successors.

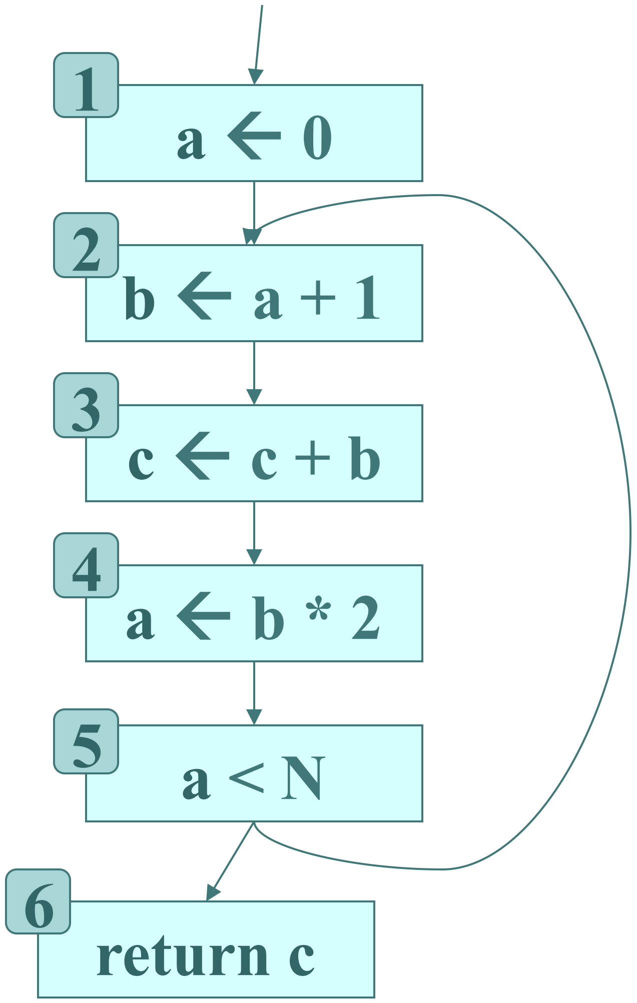
  succ[3] = { 4 }

def[N] = set of registers assigned-to in N.

  def[3] = { c }

use[N] = set of registers used in N.

  use[2] = { a }



Liveness summary

- Start with the places that **use** registers
- Propagate liveness backwards to the **def**initions.

- This is typically done via an iterative process.
- Can be expensive ➔ O(n^4)
- It is always correct to approximate
  - for example - everything is live
    - Leads to poor register allocation
  - Some algorithms compromise.
    - Cheaper to compute
    - Useable liveness information.

Example: intra-block analysis

- Perform liveness analysis on this program
  - defined use and def for each instruction.
  - find liveness for each variable.

Variables are

b, c, d, e, f ,g, h, j, k, m

Remember:

where is a variable **use**d?

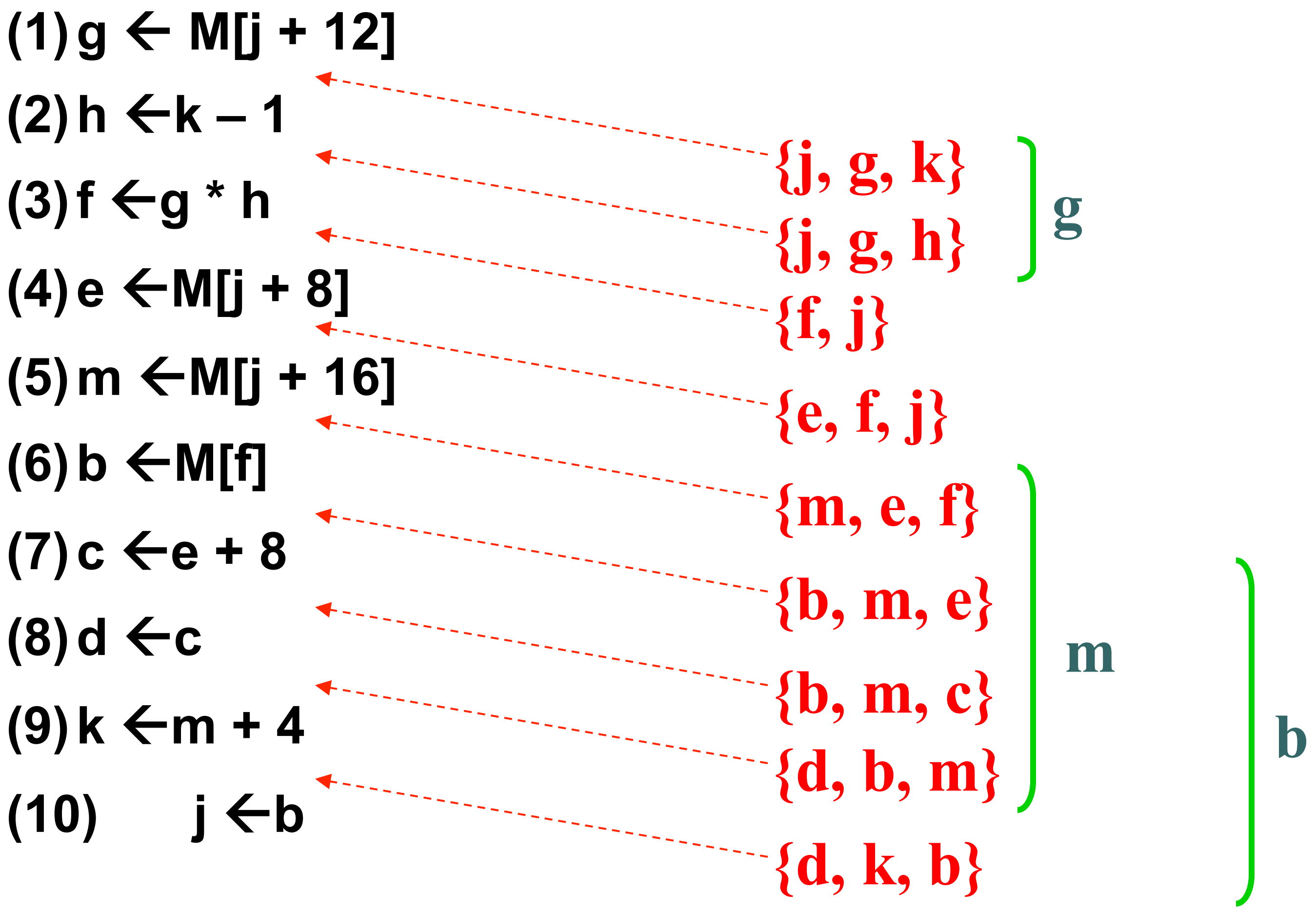
work backwards to its **def**inition.

live in: k j

```
(1) g ← M[j + 12]
(2) h ←k – 1
(3) f ←g * h
(4) e ←M[j + 8]
(5) m ←M[j + 16]
(6) b ←M[f]
(7) c ←e + 8
(8) d ←c
(9) k ←m + 4
(10)     j ←b
```

live out: d k j

liveout = {j, k}



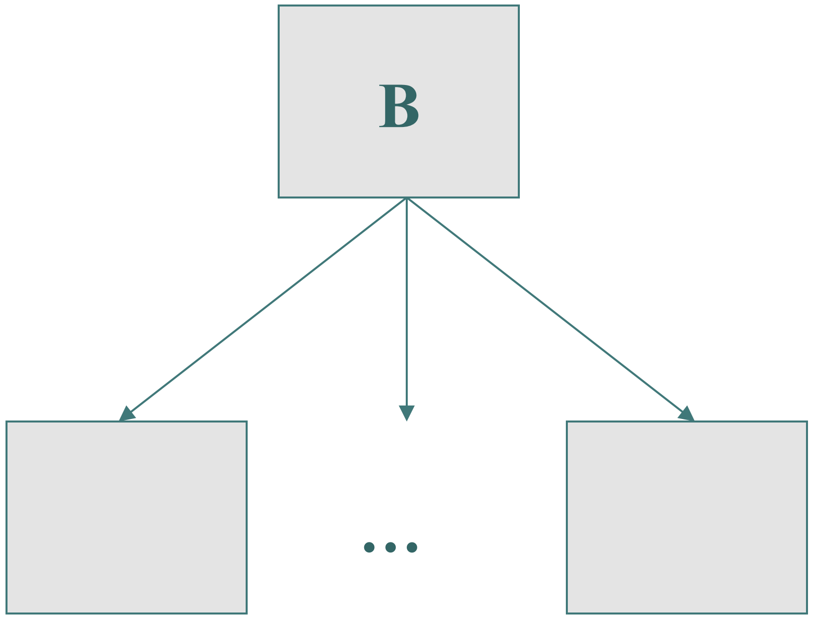
```
(1)  g ← M[j + 12]
                              {j, g, k}
(2)  h ← k – 1
                              {j, g, h}
(3)  f ← g * h
                              {f, j}
(4)  e ← M[j + 8]
                              {e, f, j}
(5)  m ← M[j + 16]
                              {m, e, f}
(6)  b ← M[f]
                              {b, m, e}
(7)  c ← e + 8
                              {b, m, c}
(8)  d ← c
                              {d, b, m}
(9)  k ← m + 4
                              {d, k, b}
(10)      j ← b
```

liveout={d, k, j}

Inter-block data flow analysis

Q: Where does the initial "liveout" come from?

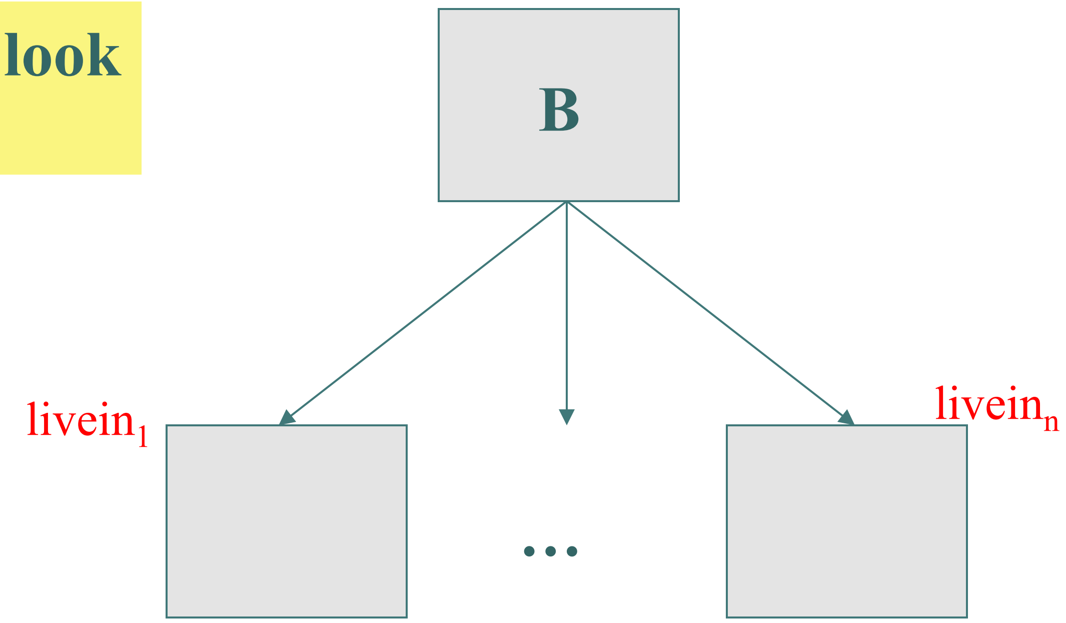
In control-flow graph, look at all successors of B



Inter-block data flow analysis

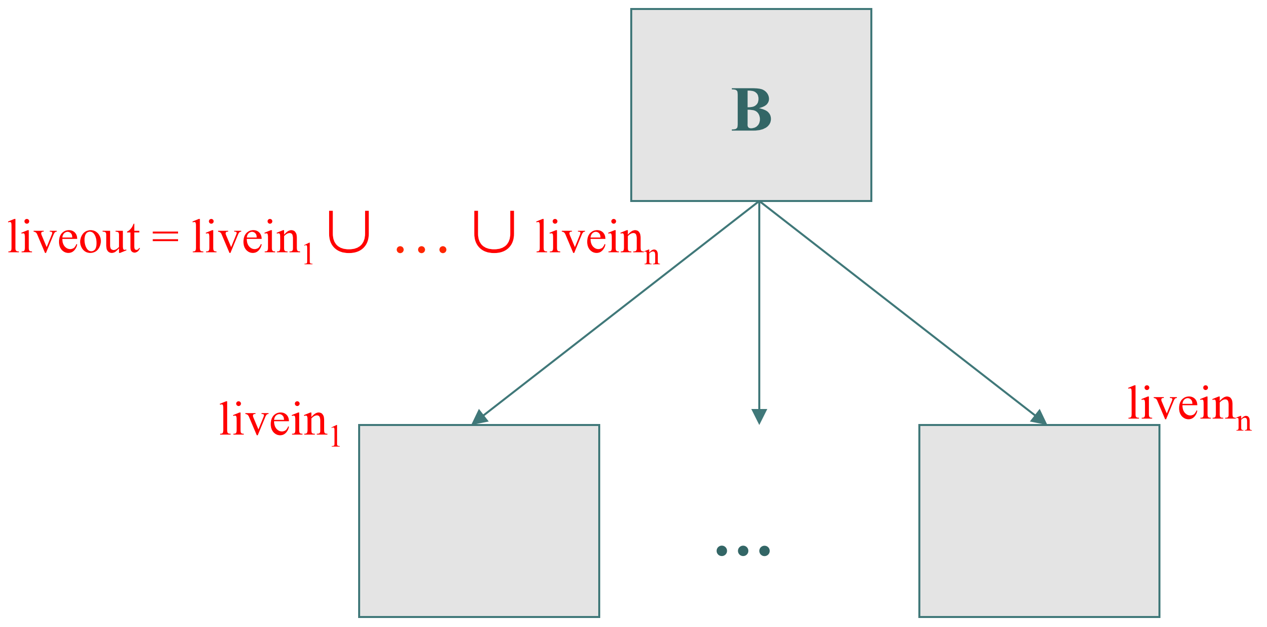
Q: Where does the initial “liveout” come from?

In control-flow graph, look at all successors of B



Inter-block data flow analysis

Q: Where does the initial “liveout” come from?



B
$\text{liveout} = \text{livein}_1 \cup \ldots \cup \text{livein}_n$
livein_1
…
livein_n

Register Allocation

ADDI	$\$r_1 \leftarrow \%r_0 + \mathbf{V_a}$
ADD	$\$r_2 \leftarrow \mathbf{fp} + \$r_1$
LOAD	$\$r_3 \leftarrow M[\$r_2 + 0]$
ADDI	$\$r_4 \leftarrow r_0 + 4$
MUL	$\$r_5 \leftarrow \$i * \$r_4$
ADD	$\$r_6 \leftarrow \$r_3 + \$r_5$
ADDI	$\$r_7 \leftarrow r_0 + \mathbf{V_x}$
ADD	$\$r_8 \leftarrow \mathbf{fp} + \$r_7$
LOAD	$\$r_9 \leftarrow M[\$r_8 + 0]$
STORE	$M[\$r_6 + 0] \leftarrow \$r_9$

- We want to give physical register allocations for each r_n
- We use \$ as a prefix for virtual registers.
- So r_0 and fp are physical registers.
  - Remember r_0 is always zero
- $\$r_0$ and \$i are virtual registers.

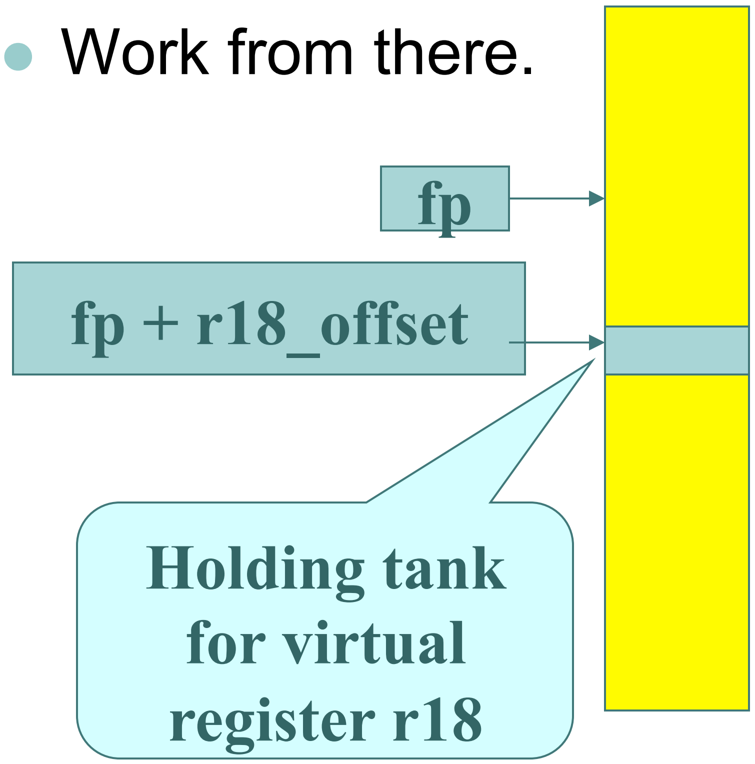
Register Spilling

```
ADD $r29 ← $r18 + $r131
```

- With any problem
  - Find a fallback base case.
  - Work from there.

```
LOAD r1 ← M [fp + r18_offset]
LOAD r2 ← M [fp + r133_offset]
ADD r1 ← r1 + r2
STORE M[fp + r29_offset] , r1
```

Needs 2 free physical registers!



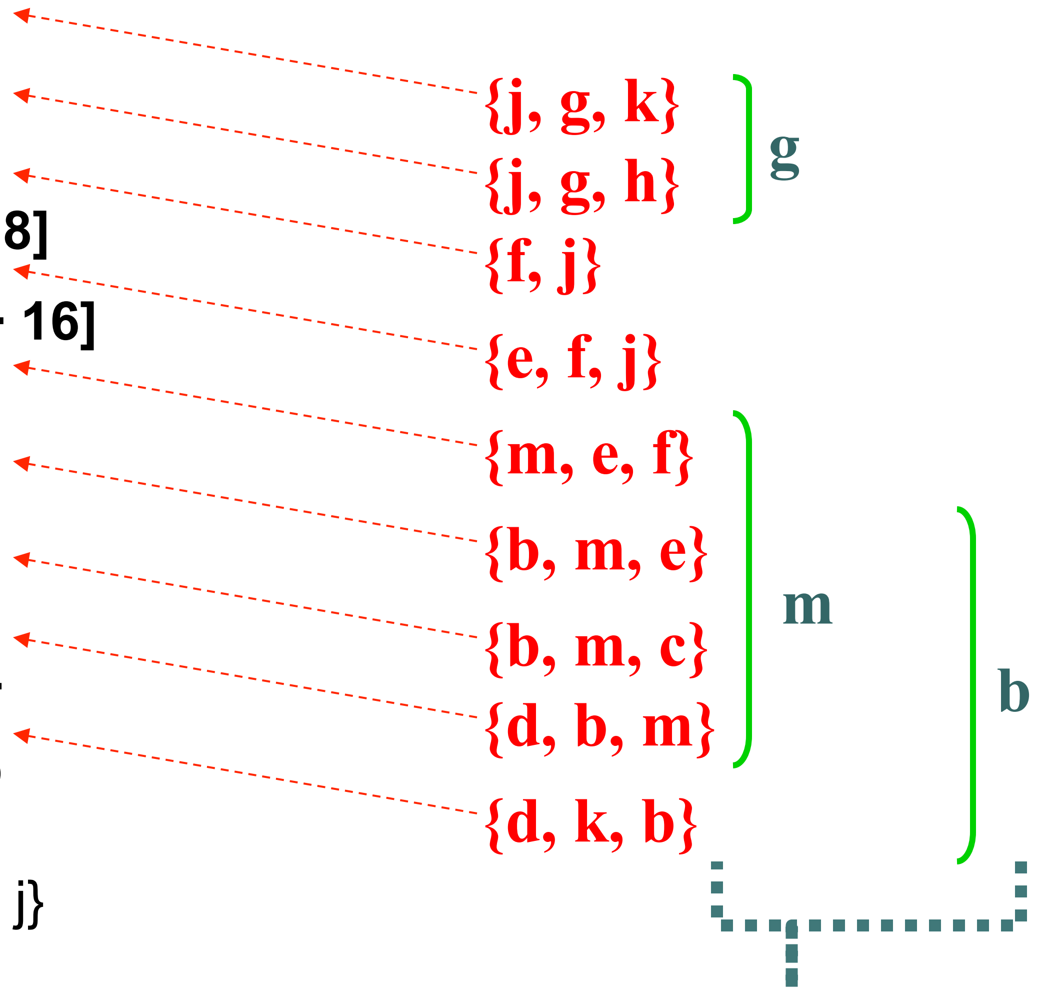
Register Allocation

- We now know when variables are **live**.
- Let us use this information to allocate registers!
  - typical approach: construct an “interference graph”
  - try to “color” it
    - number of colors = number of registers necessary

liveout = {j, k}

```
(1) g ← M[j + 12]
(2) h ← k – 1
(3) f ← g * h
(4) e ← M[j + 8]
(5) m ← M[j + 16]
(6) b ← M[f]
(7) c ← e + 8
(8) d ← c
(9) k ← m + 4
(10)      j ← b
```

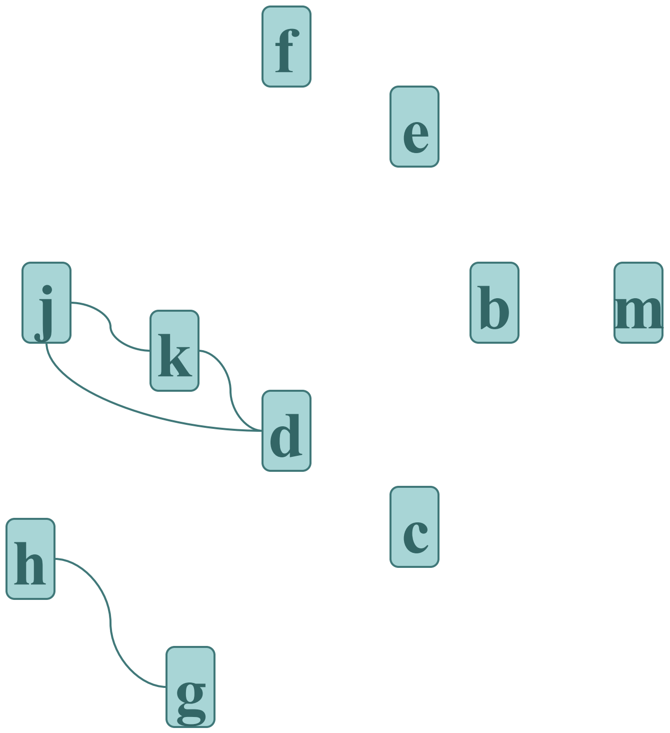
liveout={d, k, j}



overlapping lifetimes “interfere”

Graph of interference (1)

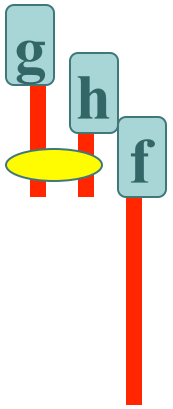
For every edge on the control flow graph. Add an edge on the interference graph between every register that is live at that point.



live in: k j

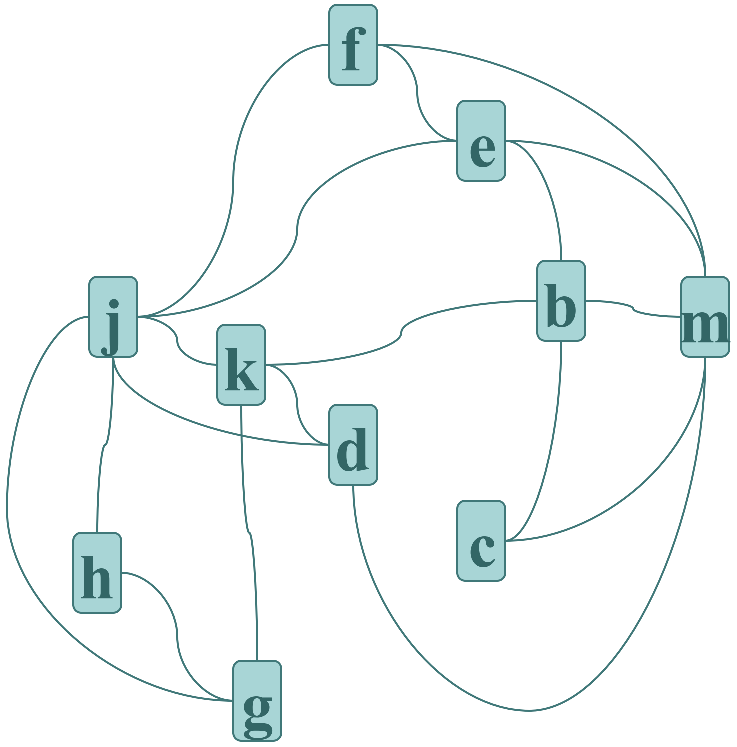
(1) g ← M[j + 12]
(2) h ← k – 1
(3) f ← g * h
(4) e ← M[j + 8]
(5) m ← M[j + 16]
(6) b ← M[f]
(7) c ← e + 8
(8) d ← c
(9) k ← m + 4
(10) j ← b

live out: d k j



Graph of interference (2)

An edge between two (virtual) registers means that they can not be assigned to the same physical register.



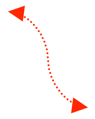
live in: k j

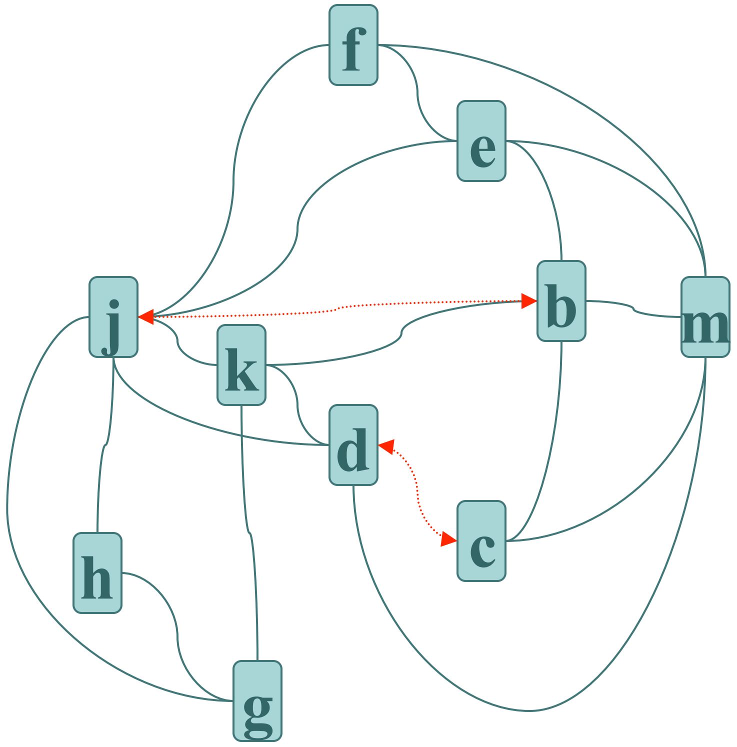
(1) g ← M[j + 12]
(2) h ←k – 1
(3) f ←g * h
(4) e ←M[j + 8]
(5) m ←M[j + 16]
(6) b ←M[f]
(7) c ←e + 8
(8) d ←c
(9) k ←m + 4
(10) j ← b

live out: d k j

Graph of interference (3)

We also mark **move** instructions with bidirectional arrows.





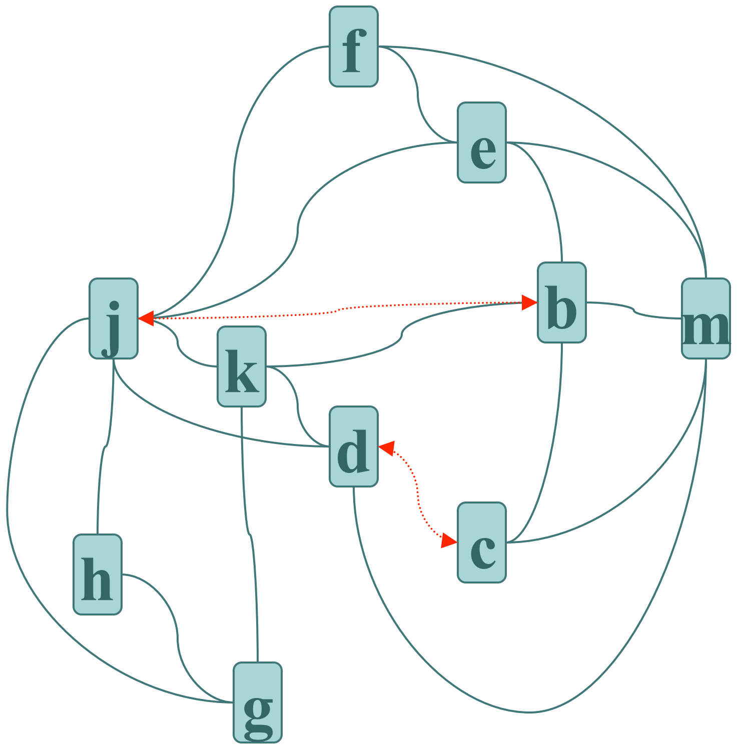
live in: k j

(1) g ← M[j + 12]
(2) h ←k – 1
(3) f ←g * h
(4) e ←M[j + 8]
(5) m ←M[j + 16]
(6) b ←M[f]
(7) c ←e + 8
(8) d ← c
(9) k ←m + 4
(10) j ← b

live out: d k j

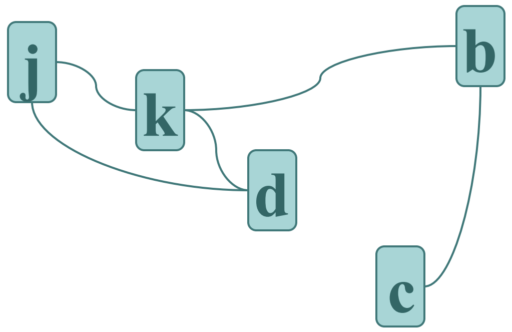
How can we allocate registers?

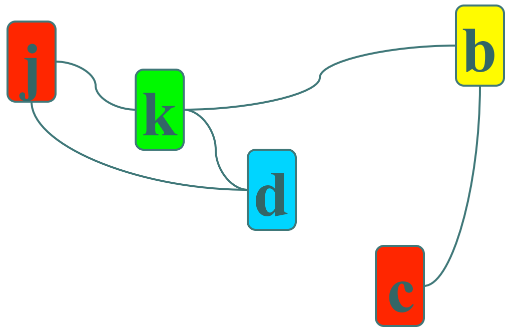
- Graph coloring!
  - Comes for map making
- Color the edges such that
  - No two **connected** nodes have the same color.
  - But we **want** the bidirectional arrows to point to the same color.
    - Moves between the same register can be omitted!



How do we color our interference graph?

Simple Example





This is a good coloring

- There are four colors used.

k-coloring the graph

- Assuming
  - the interference graph *G* associated with our program
    - program still uses virtual registers (e.g., $r1)
  - we have a target architecture with *k* registers,
- *k*-coloring: coloring *G* with *k* or fewer colors
  - if possible, means spill-free register allocation is possible
  - if not possible, insert some number of spills.
- Big Issue: *k*-coloring problem is NP-hard
  - not computationally tractable (unless P=NP, of course)
  - ∴ use an *approximation algorithm* to test for *k*-colorability

"Approximation Algorithm?"

- It is a heuristic which may determine if *G* is *k*-colorable
  - should run quickly
  - it's an approximation: may give a "false negative" but never a "false positive"
  - may be many reasonable approximations
- Example: divide and conquer approach
  - remove each node (of degree < *k*) and its edges in some order
  - then, in reverse order, reinsert the nodes & edges, coloring as you go
  - the bidirectional red arrows "don't count"

“Coloring by Simplification” Heuristic

```
Coloring simpcolor (Graph G,Int k)
    pick node m such that degree(m) < k;
    if (no such m exists)
        then FAIL;
        else {
            c := simpcolor (G\{m}); /* might FAIL */
            c’:= c + (m ← freecolor(c));
            return c’;
            }
```

Works because if `simpcolor(G\{m})` succeeds and `degree(m)<k`, then there is at least one free color left for `m`

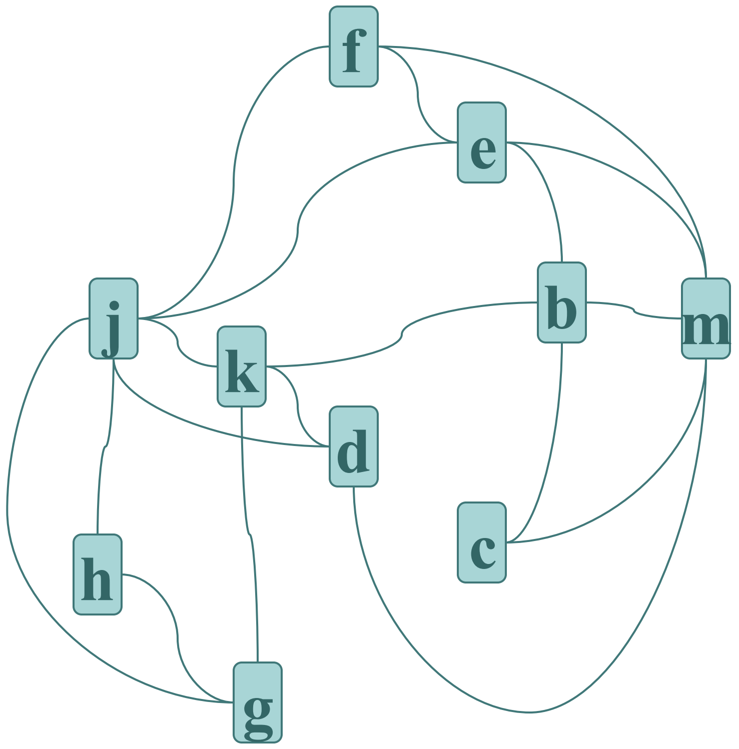
Graph coloring (1)

- Let us color the graph!
  - We need four colors (K)

Start with **g**

Always remove nodes with K-1 (or less) edges.

Remove g.

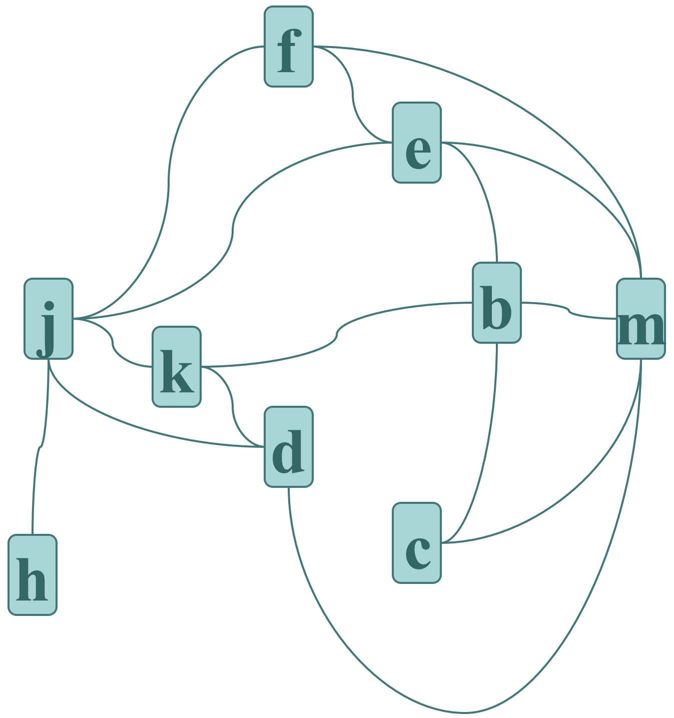


Graph coloring (2)

Always remove nodes with K-1 (or less) edges.

Remove g.

Remove h.

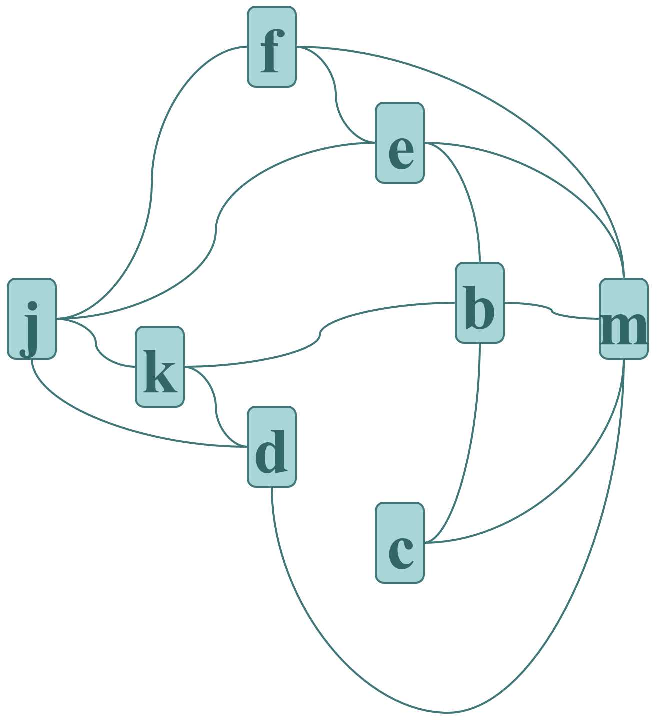


Graph coloring (3)

Remove g.

Remove h.

Remove k.



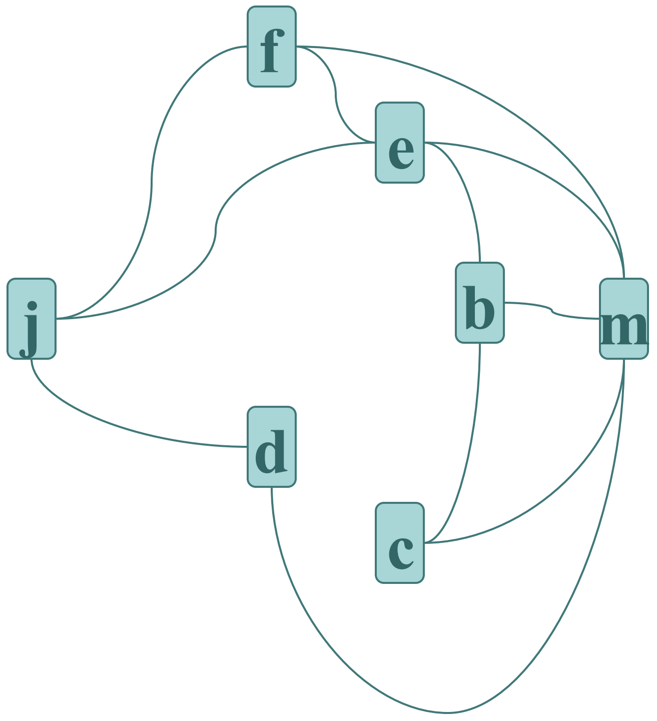
Graph coloring (5)

Remove g.

Remove h.

Remove k.

Remove d.



Graph coloring (6)

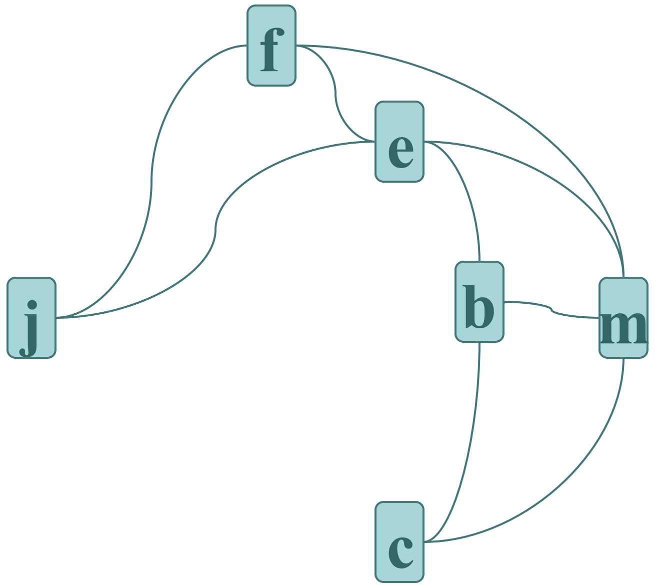
Remove g.

Remove h.

Remove k.

Remove d.

Remove j.



Graph coloring (7)

Remove g.

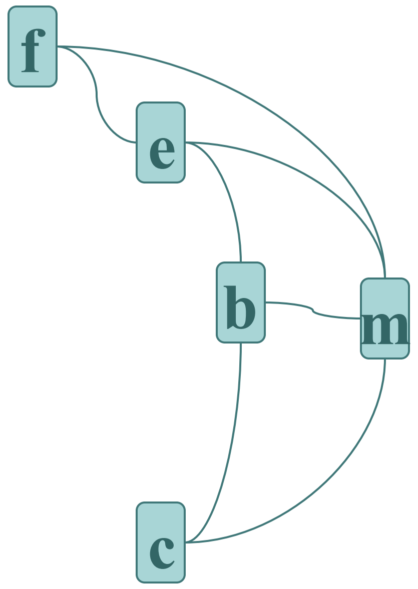
Remove h.

Remove k.

Remove d.

Remove j.

Remove e.



Graph coloring (8)

Remove g.

Remove h.

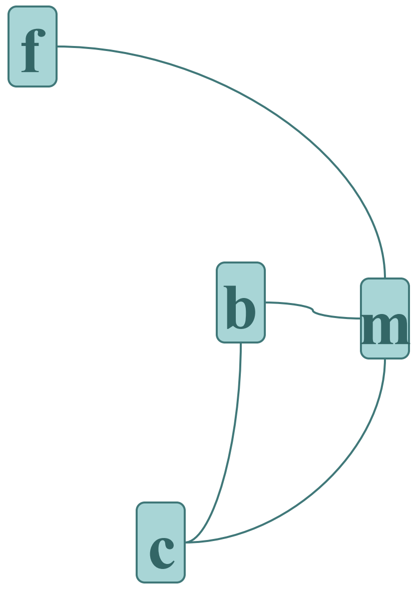
Remove k.

Remove d.

Remove j.

Remove e.

Remove f.



Graph coloring (9)

Remove g.

Remove h.

Remove k.

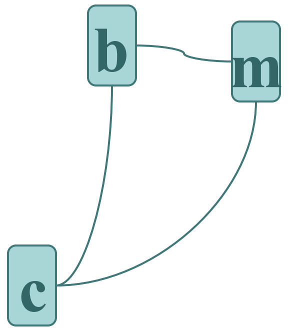
Remove d.

Remove j.

Remove e.

Remove f.

Remove b.



Graph coloring (10)

Remove g.

Remove h.

Remove k.

Remove d.

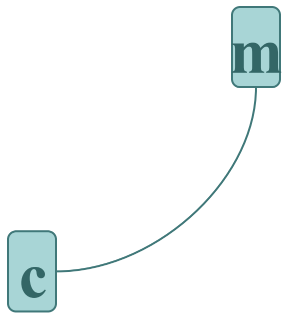
Remove j.

Remove e.

Remove f.

Remove b.

Remove c.



Graph coloring (11)

Remove g.

Remove h.

Remove k.

Remove d.

Remove j.

Remove e.

Remove f.

Remove b.

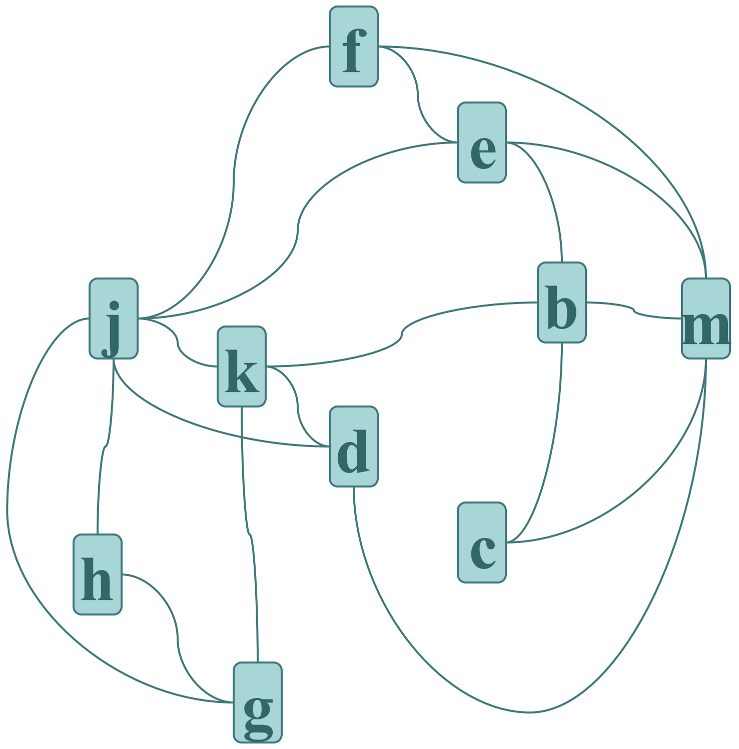
Remove c.

Remove m.

Graph coloring (12)

Now go backwards through this list, coloring the graph…

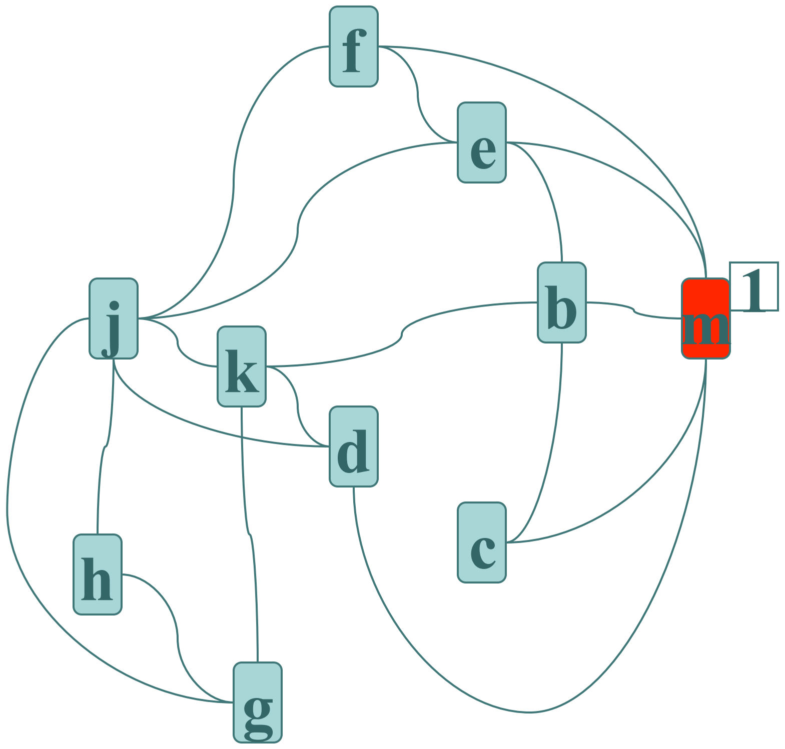
Remove g.
Remove h.
Remove k.
Remove d.
Remove j.
Remove e.
Remove f.
Remove b.
Remove c.
Remove m.



Graph coloring (13)

Now go backwards through this list, coloring the graph…

Remove g.
Remove h.
Remove k.
Remove d.
Remove j.
Remove e.
Remove f.
Remove b.
Remove c.
Remove m.



Graph coloring (14)

Now go backwards through this list, coloring the graph…

Remove g.

Remove h.

Remove k.

Remove d.

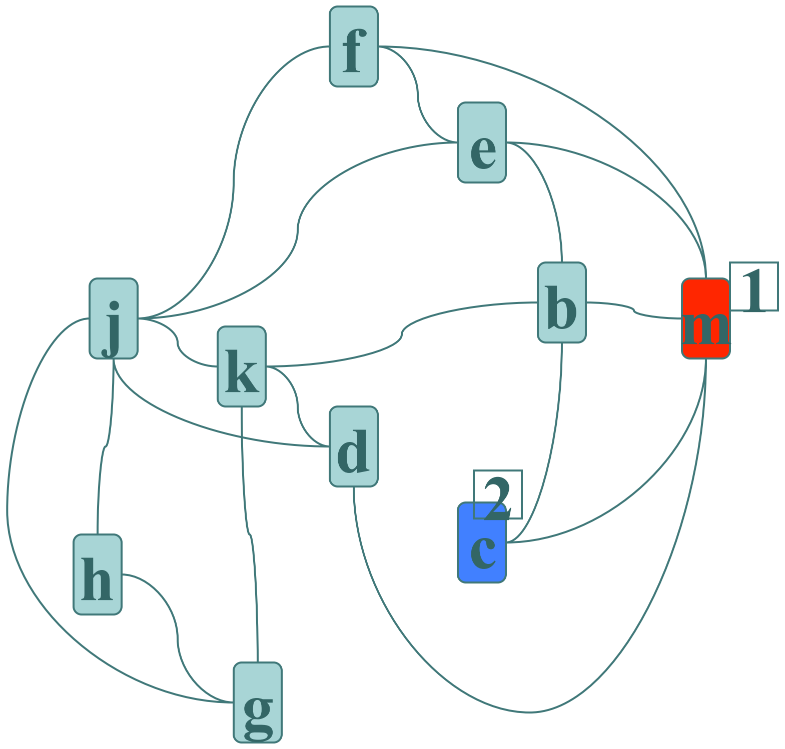
Remove j.

Remove e.

Remove f.

Remove b.

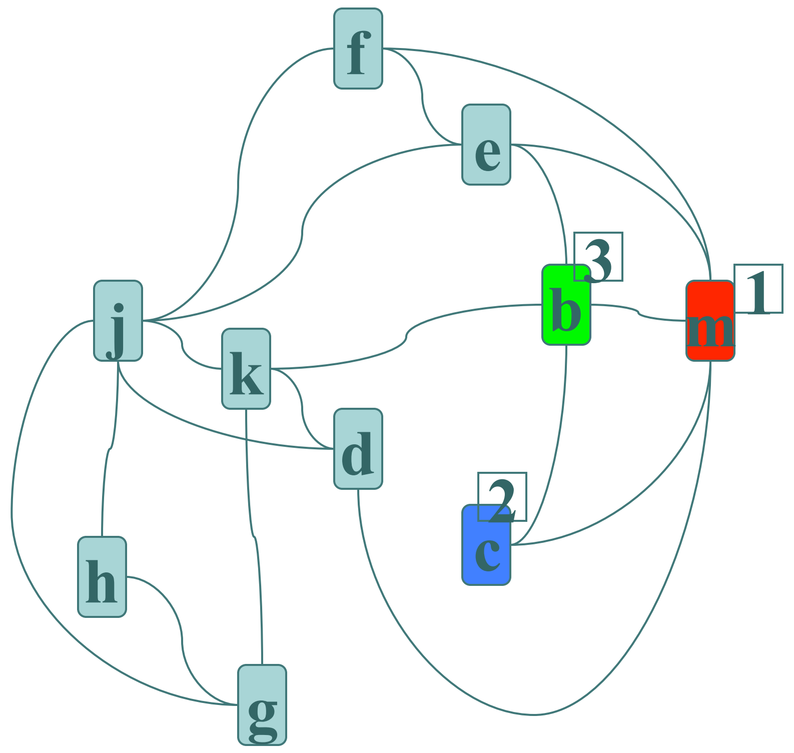
Remove c.



Graph coloring (15)

Now go backwards through this list, coloring the graph…

Remove g.
Remove h.
Remove k.
Remove d.
Remove j.
Remove e.
Remove f.
Remove b.



Graph coloring (16)

Now go backwards through this list, coloring the graph…

Remove g.

Remove h.

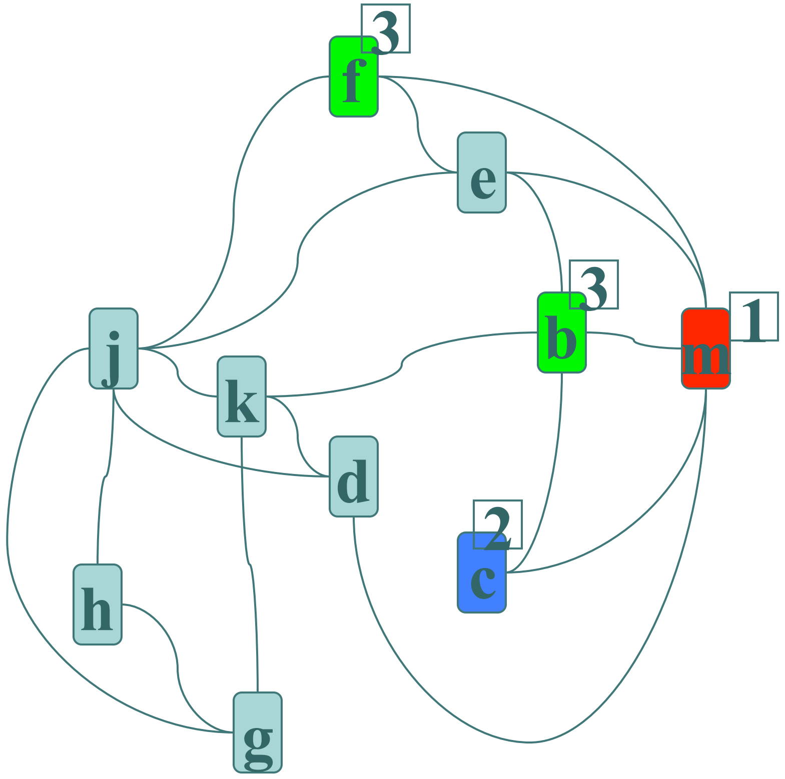
Remove k.

Remove d.

Remove j.

Remove e.

Remove f.



Graph coloring (17)

Now go backwards through this list, coloring the graph…

Remove g.

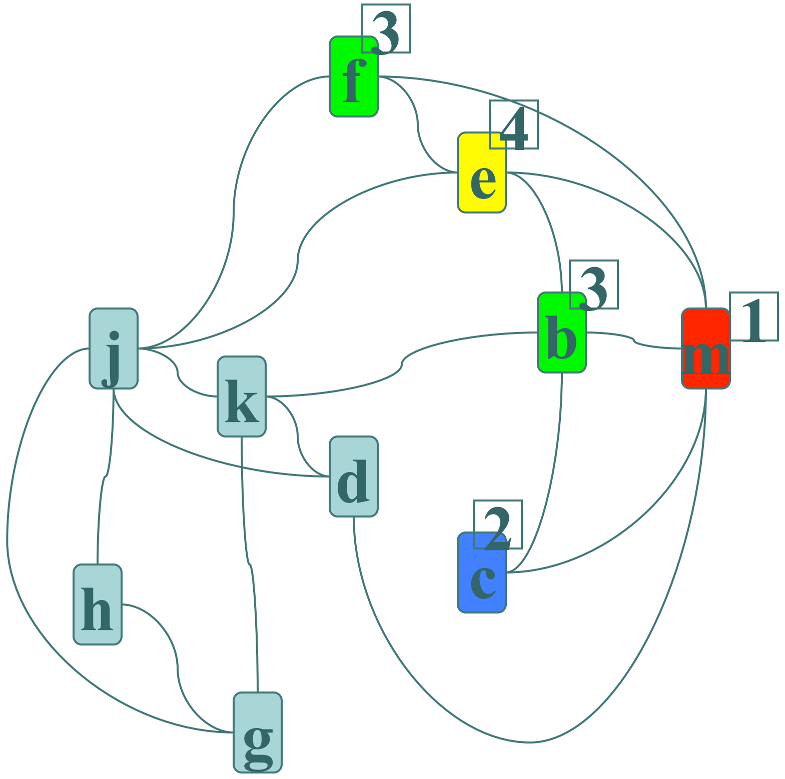
Remove h.

Remove k.

Remove d.

Remove j.

Remove e.



Graph coloring (18)

Now go backwards through this list, coloring the graph…

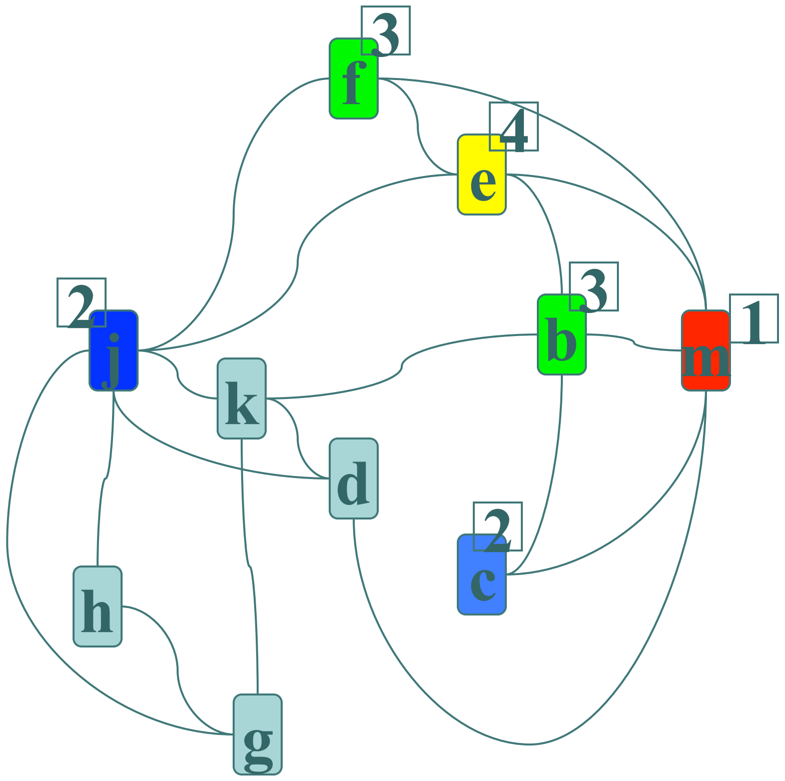
Remove g.

Remove h.

Remove k.

Remove d.

Remove j.



Graph coloring (19)

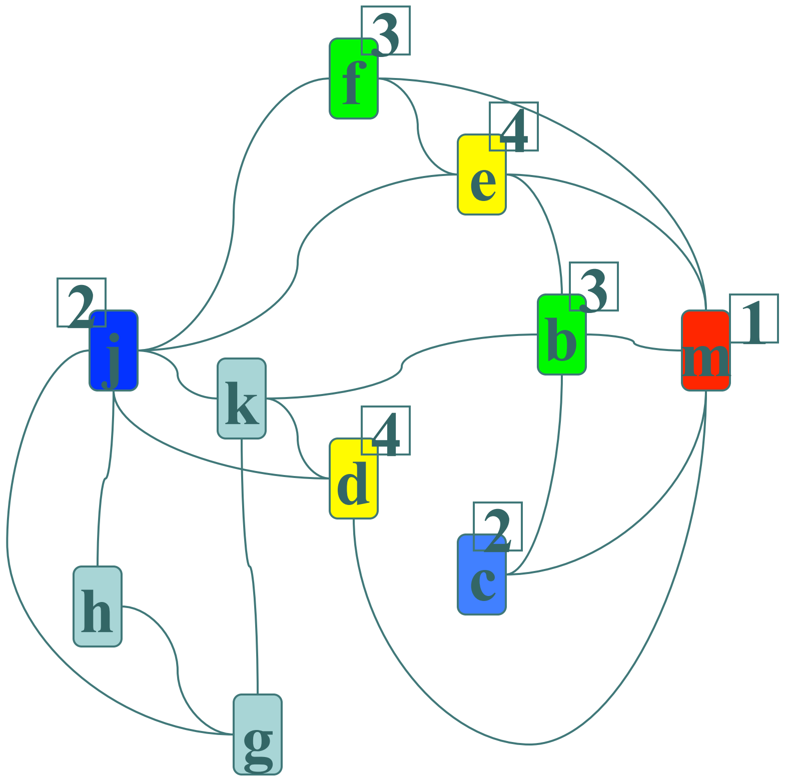
Now go backwards through this list, coloring the graph…

Remove g.

Remove h.

Remove k.

Remove d.



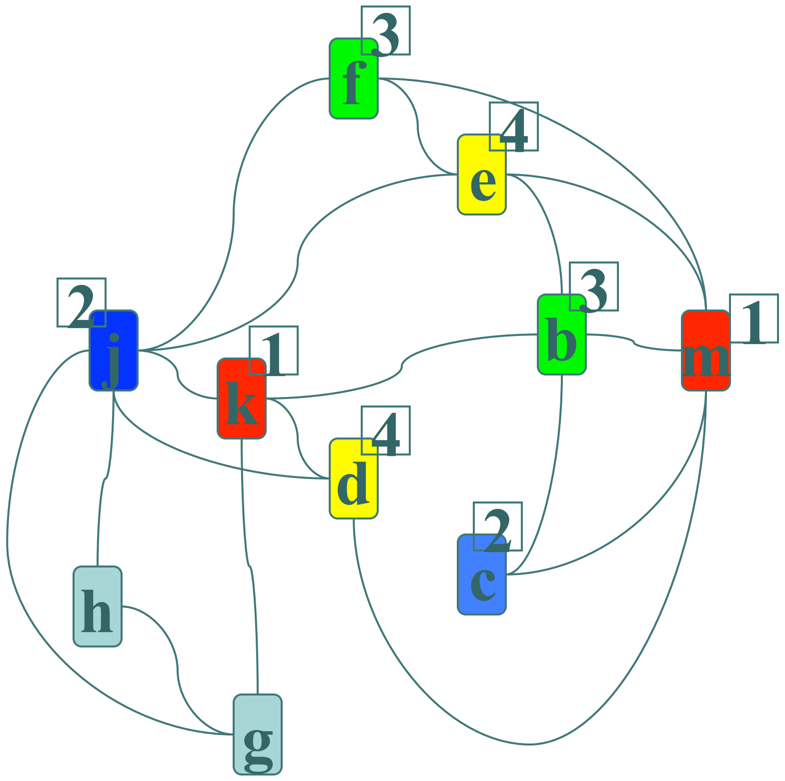
Graph coloring (20)

Now go backwards through this list, coloring the graph…

Remove g.

Remove h.

Remove k.

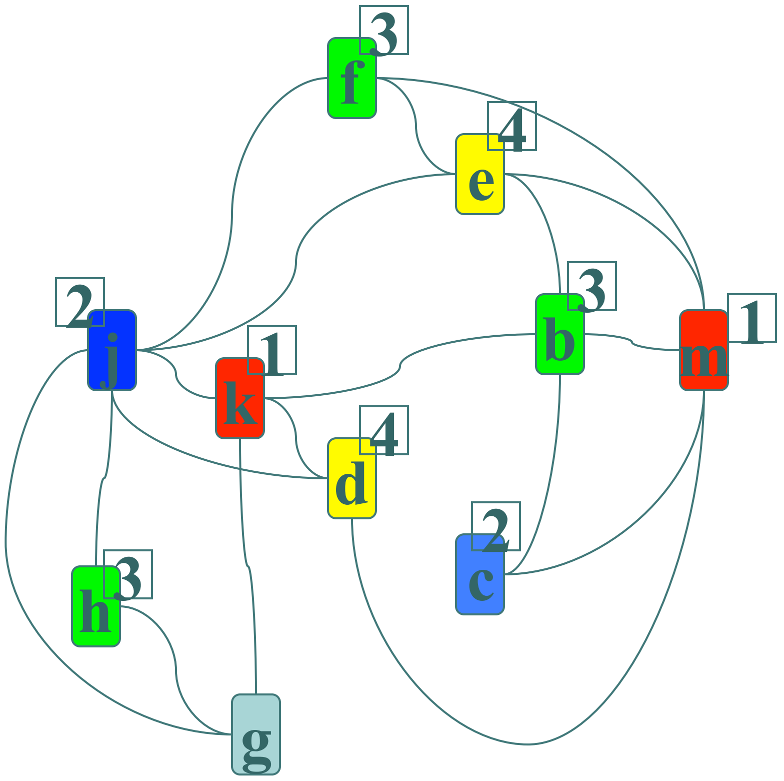


Graph coloring (22)

Now go backwards through this list, coloring the graph…

Remove g.

Remove h.

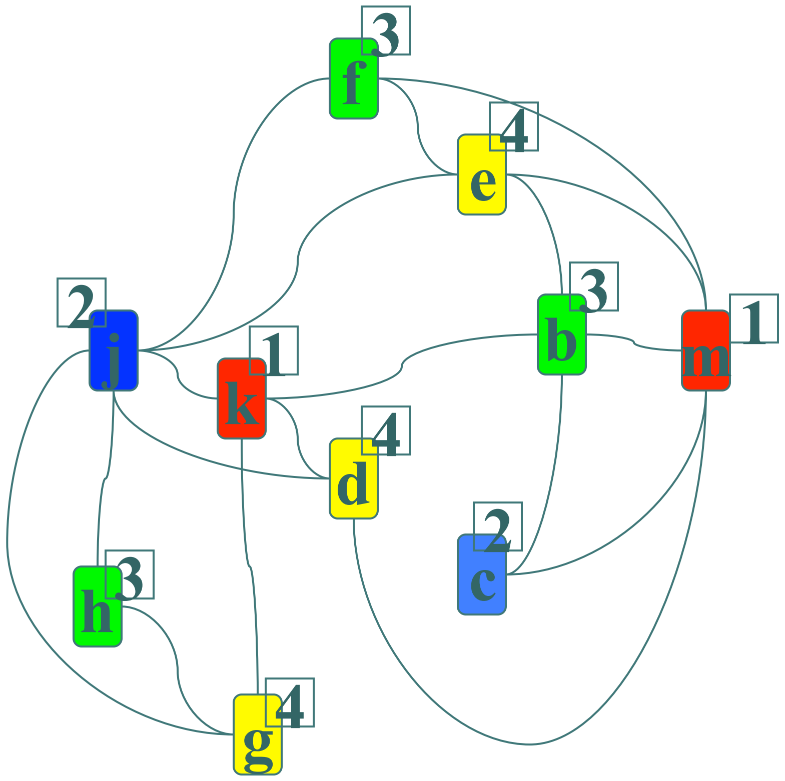


Graph coloring (23)

Now go backwards through this list, coloring the graph…

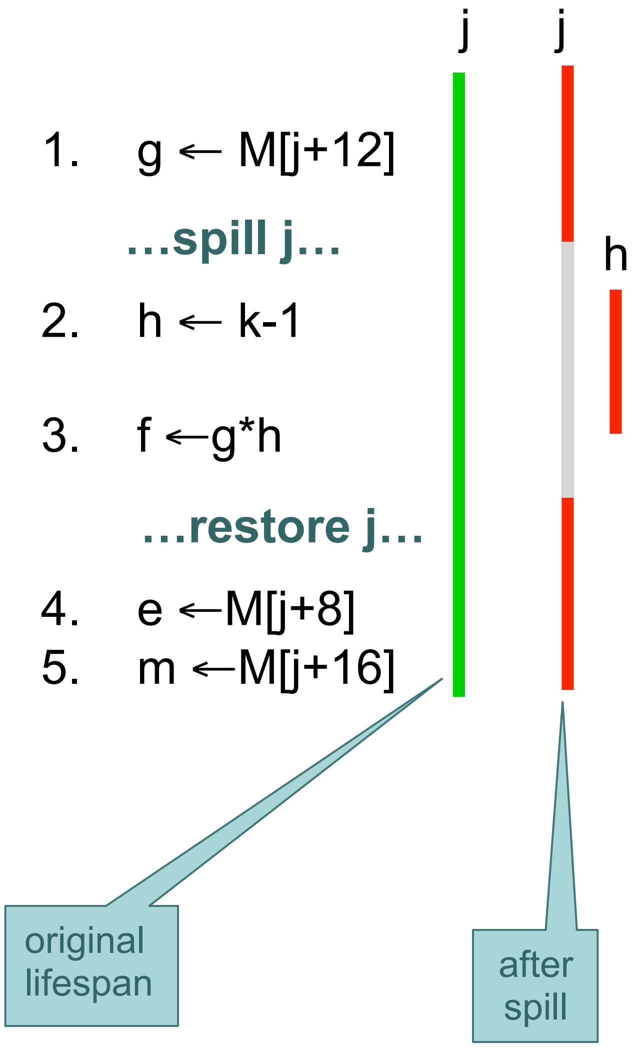
Remove g.

And were done. This is a valid coloring of this graph.



What if we have too few colors?

- "Spill" a virtual register to memory
- This will reduce interference
- …possibly reducing number of colors required
- In this example, "h" and "j" no longer interfere

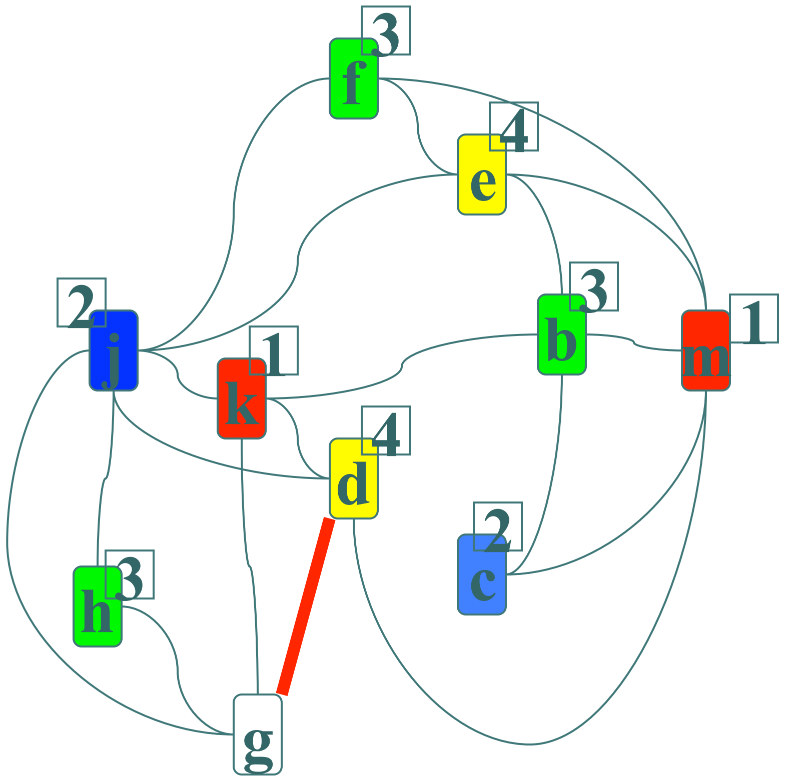


```
1.   g ← M[j+12]
       …spill j…
2.   h ← k-1
3.   f ←g*h
       …restore j…
4.   e ←M[j+8]
5.   m ←M[j+16]
```

Graph coloring (23) redux

What if we had an extra edge* in the interference graph?

We can’t color “g” with red, yellow, blue or green



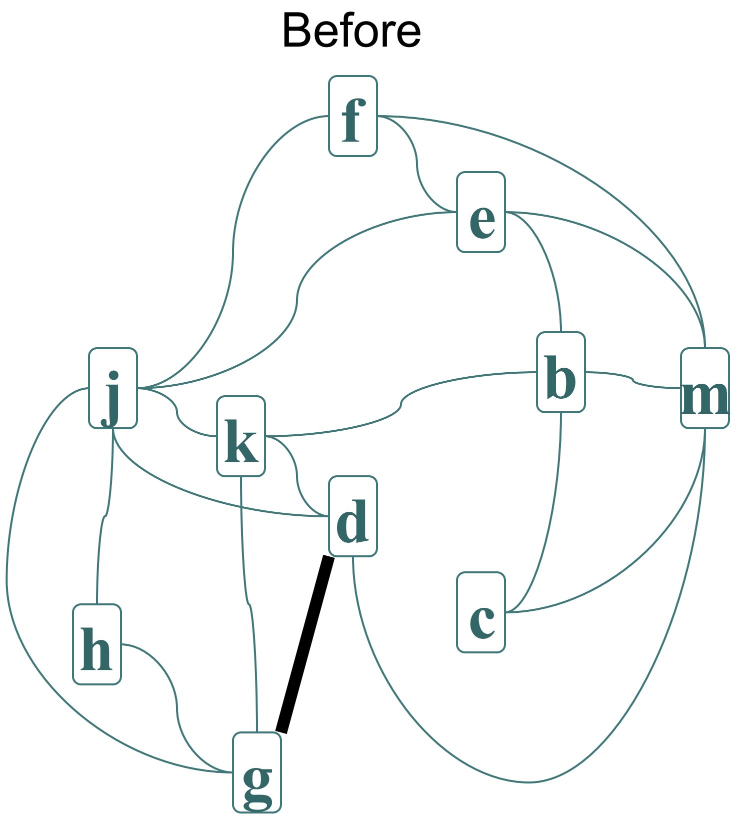
* Edge doesn’t actually appear in interference graph

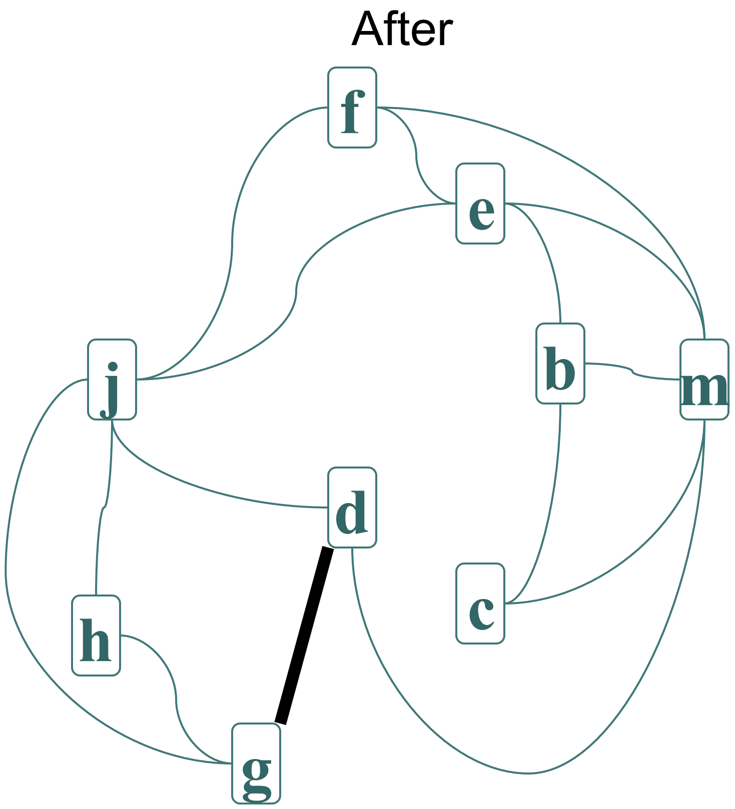
Effect of spilling k

```
(1) g ← M[j + 12]
(2) h ←k – 1
(3) f ←g * h
(4) e ←M[j + 8]
(5) m ←M[j + 16]
(6) b ←M[f]
(7) c ←e + 8
(8) d ←c
(9) k ←m + 4
(10) j ← b
```

```
(1) g ← M[j + 12]
(2) h ← M[fp+offset(k)] – 1
(3) f ←g * h
(4) e ←M[j + 8]
(5) m ←M[j + 16]
(6) b ←M[f]
(7) c ←e + 8
(8) d ←c
(9) M[fp+offset(k)] ←m + 4
(10) j ← b
```

Effect of spilling "k" on I-graph

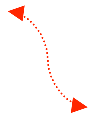


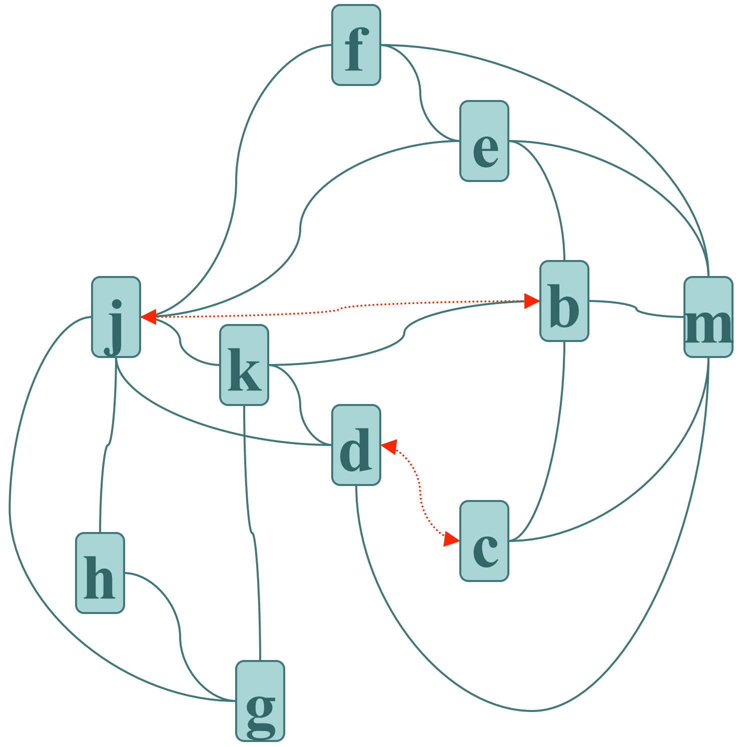


Next Step: try recoloring

Register Coalescing

We also mark **move** instructions with bidirectional arrows.





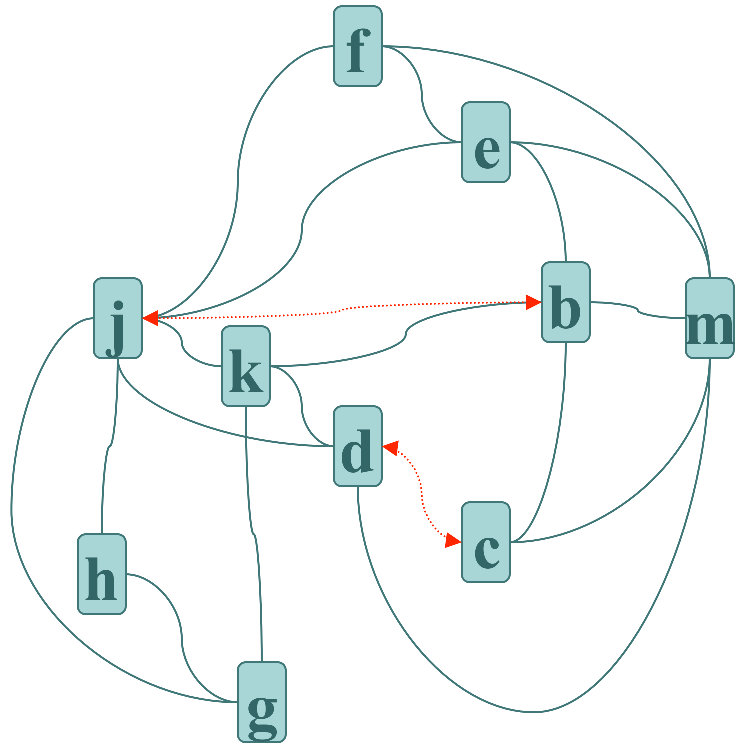
(1) g ← M[j + 12]
(2) h ← k – 1
(3) f ← g * h
(4) e ← M[j + 8]
(5) m ← M[j + 16]
(6) b ← M[f]
(7) c ← e + 8
(8) d ← c
(9) k ← m + 4
(10) j ← b

Register Coalescing

- A register to register "move instruction" is an opportunity to remove an instruction
  - Ex: "$j \leftarrow b$" in the liveness analysis example
  - if j,b are not connected in interference graph, then
    - merge them into one node in G
    - reflect the changes in the IR
      - basically, makes writes to "b" into writes to "j"
- Transformation known as "copy propagation"

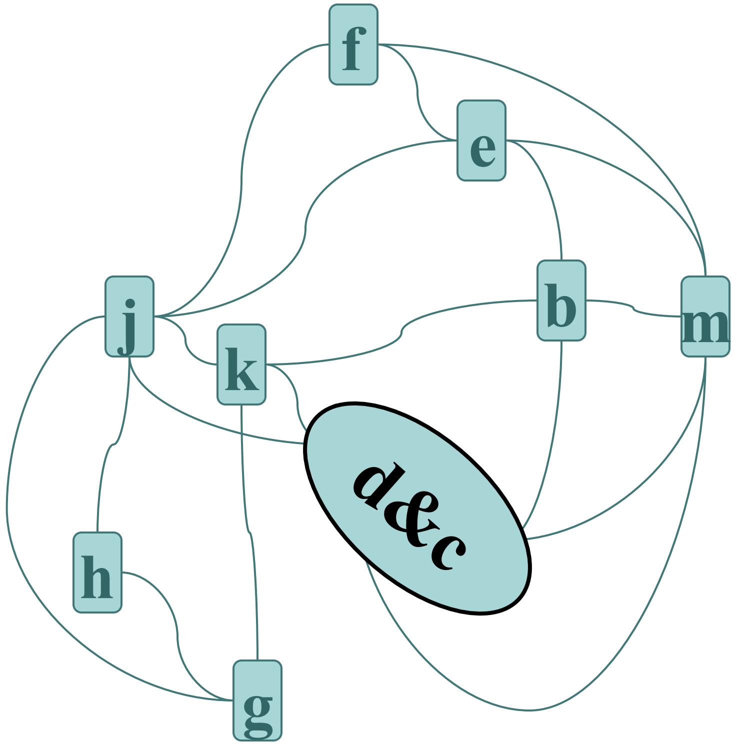
Coalescing (1)

- We would like the bidirectional arrows to point to the same color
- If there is no interference edge between node$_1 \leftrightarrow$ node$_2$, then join (coalesce) two nodes into single node



Coalescing (2)

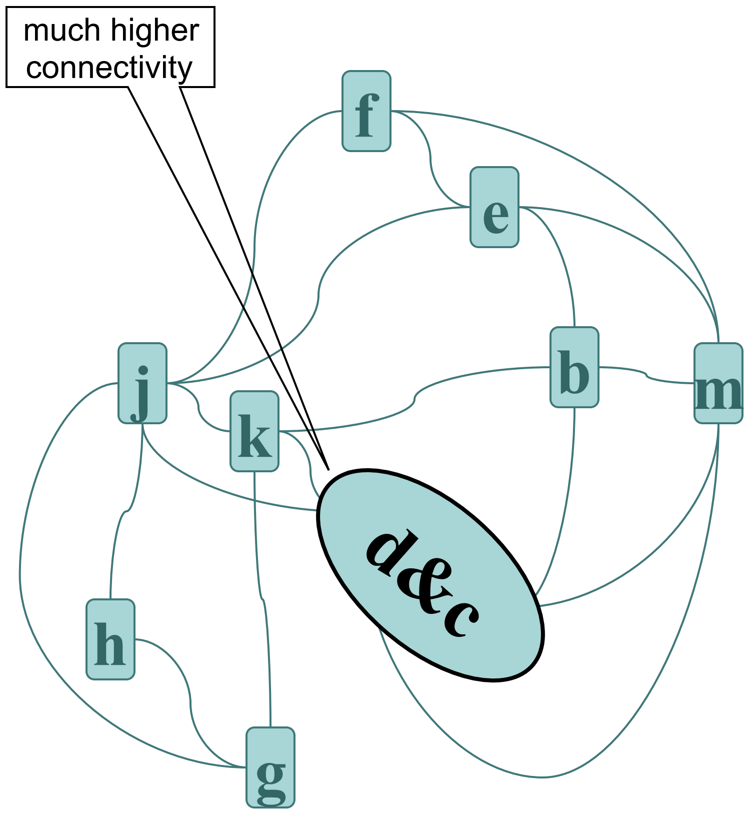
- We would like the bidirectional arrows to point to the same color
- If there is no interference edge between $node_1 \leftrightarrow node_2$, then join (coalesce) two nodes into single node



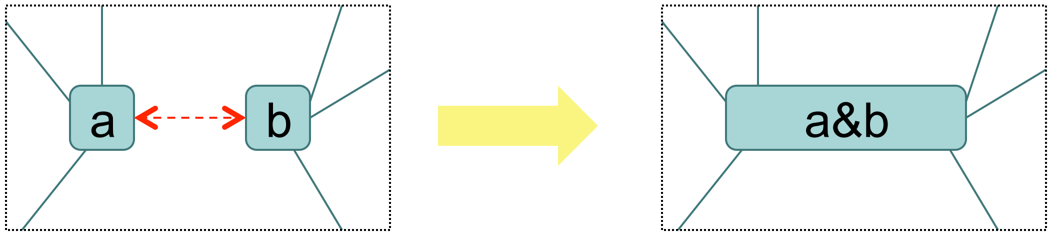
* The coalescing of j&b is not accurate (i.e., k is clobbered,etc.).

Be careful with coalescing!

- After coalescing, resulting graph may no longer be k-colorable
- Leaving trivial move instructions is better than spilling
- Want conservative coalescing
  - heuristics that don't change the graph's colorability

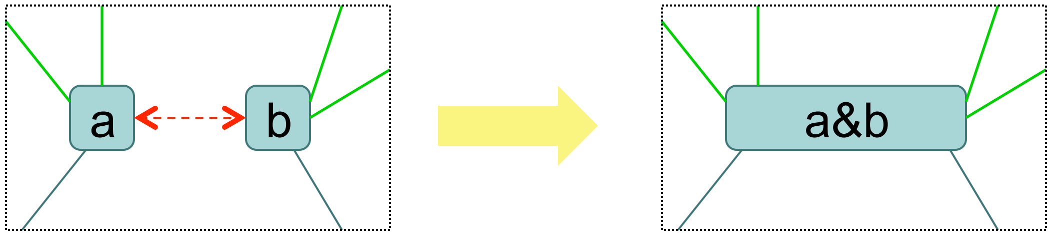


“Chaitin” heuristic



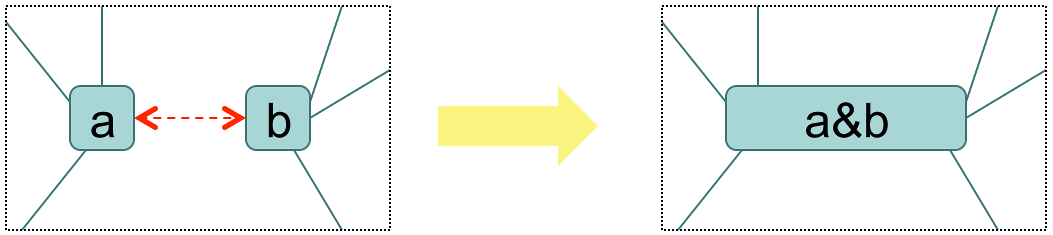
- Aggressive: coalesces any two non-interfering nodes in graph

“Briggs” heuristic



- Permitted only when less than k neighbors of “a&b” are of degree at least k
  - edges to these neighbors shown in green
- permitted for k=5
- not permitted for k=4

“George” heuristic



- Permitted when every neighbor of “a”
  - already interferes with “b”
  - or, is of insignificant degree (<< k)