Feel free to take candy!

(Subject to the following restrictions)

- For every pair of people, if your **first or last** names start with the same letter, you can’t take the same kind of candy.
- Stefan has already taken a Kit Kat

(Don’t worry, if this only leaves you with candy you don’t like/are allergic to/etc., you can get more)

CS443: Compiler Construction

Lecture 19: Register Allocation

Stefan Muller

Based on material by Steve Zdancewic

Register allocation: going from unlimited temporaries to fixed number of registers

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-7	t0-2
x8	s0/fp
x9	s1
x10-11	a0-1
x12-17	a2-7
x18-27	s2-11
x28-31	t3-t6

- x0–x4: Special purpose
- x5–x9: General purpose
- x10–x17: Sometimes special purpose (by convention)
- x18–x31: General purpose

Find: mapping from program variables to registers

- What if there aren't enough registers?

```
int annoying(int[] a) {
  int v0 = a[0];
  int v1 = a[1];
  int v2 = a[2];
  int v3 = a[3];
  int v4 = a[4];
  int v5 = a[5];
  int v6 = a[6];
  int v7 = a[7];
  int v8 = a[8];
  int v9 = a[9];
  …
  return v0 + v1 + v2 + v3 + v4 + …
}
```

Find: mapping from program variables to (registers ∪ stack locations)

“spill”

```
type alloc_res = InReg of R.reg
               | OnStack of int (* stack slot, 0-N *)
               | InMem of R.symbol (* globals on heap *)
```

Many quality metrics for allocation

- Program semantics is preserved (i.e. the behavior is the same)
- Register usage is maximized
- Moves between registers are minimized
- Calling conventions / architecture requirements are obeyed

Recall: A variable is “live” when its value is needed

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + x;
  return c;
}
```

- x is live
- a and x are live
- b and x are live
- c is live

Liveness analysis is based on uses and definitions

- For a node/statement s define:
  - use[s] : set of variables used (i.e. read) by s
  - def[s] : set of variables defined (i.e. written) by s

- Examples:
  - a = b + c   use[s] = {b,c}   def[s] = {a}
  - a = a + 1   use[s] = {a}   def[s] = {a}

Liveness analysis as a dataflow analysis (Steps 1-2)

- Facts: Live variables
- gen[n] = use[n]
- kill[n] = def[n]

- Constraints:
  - in[n] $\supseteq$ gen[n]
  - out[n] $\supseteq$ in[n’] if n’ $\in$ succ[n]
  - in[n] $\supseteq$ out[n] / kill[n]

Liveness analysis as a dataflow analysis (Steps 3-4)

- Equations:
  - $out[n] := \bigcup_{n' \in succ[n]} in[n']$
  - $in[n] := gen[n] \cup (out[n] / kill[n])$
- Initial values:
  - $out[n] := \emptyset$
  - $in[n] := \emptyset$

For register allocation: live(x)

- live(x) = set of variables that are live-in to the definition of x
  - (assuming SSA)

Linear Scan: a simple, greedy algorithm

1. Compute liveness information: `live(x)`
2. Let `regs` be the set of usable registers
3. Maintain "layout" `alloc` that maps uids to `alloc_reg`
4. Scan through the program. For each instruction that defines a var x
   - `used` = `{r | reg r = alloc(y)` s.t. `y` ∈ `live(x)}`
   - `available = regs - used`
   - If `available` is empty: *// no registers available, spill*
     `alloc(x) := OnStack n; n := !n + 1`
   - Otherwise, pick `r` in `available`: *// choose an available register*
     `alloc(x) := InReg r`

Linear Scan Example (registers: r0, r1, r2)

```
int f(int x) {
  int a = x + 2;
  int b = a * a;
  int c = b + a;
  return c;
}
```

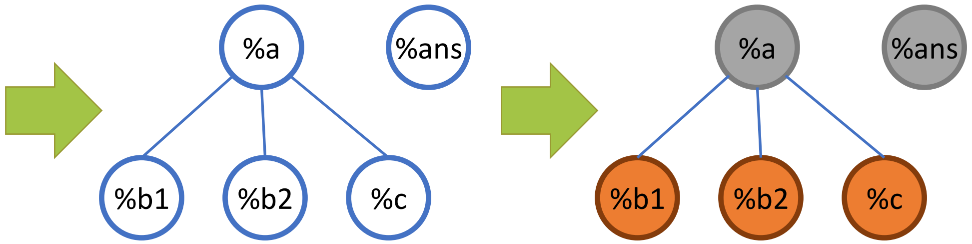
Available	
r0, r1, r2	a -> r0
r1, r2	b -> r1
r2	c -> r2

Linear scan is OK, but we can do better

Who had “reduce it to a graph problem” on their CS Bingo card?

- Nodes of the graph are variables
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
%b1 = add i32 %a, 2
%c = mult i32 %b1, %b1
%b2 = add i32 %c, 1
%ans = add i32 %b2, %a
return %ans;
```



Interference Graph

2-Coloring of the graph
red = r8
yellow = r9

Heuristics for graph coloring come down to order in which you color nodes

- Linear Scan: Order of definitions in program
- *Simplification*: (Roughly) color high degree nodes first

Coloring by simplification

1. **Build** Interference Graph
2. **Simplify** the graph by removing nodes one at a time, putting them on a stack
3. **Select** colors for nodes in order of the stack

We don’t want to treat move instructions as conflicts/interference

```
%a = inttoptr i32* %aptr to i32
%b = add i32 %a 8
%bptr = ptrtoint i32 %b to i32*
%c = load i32, i32* %aptr
%d = load i32, i32* %bptr
```

%a and %aptr are live at the same time, but can (and should) be in the same register

Steps for a simple graph-coloring allocator

1. **Build** interference graph
2. **Simplify** graph by removing nodes one at a time, pusing them on a stack, until all nodes are on stack
3. As we simplify, identify nodes to potentially **spill**
4. **Select** colors/registers for nodes (in reverse order they were pushed to the stack)

Build interference graph

- For each instruction:
  - If the inst defines a variable a, with $b_1, \ldots, b_n$ live-out:
  - If the instruction is not a move, add edges $(a, b_1), \ldots, (a, b_n)$
  - If the instruction is a move a = c, add edges $\{(a, b_i) \mid b_i \neq c\}$

Coloring by simplification: Simplify/Spill

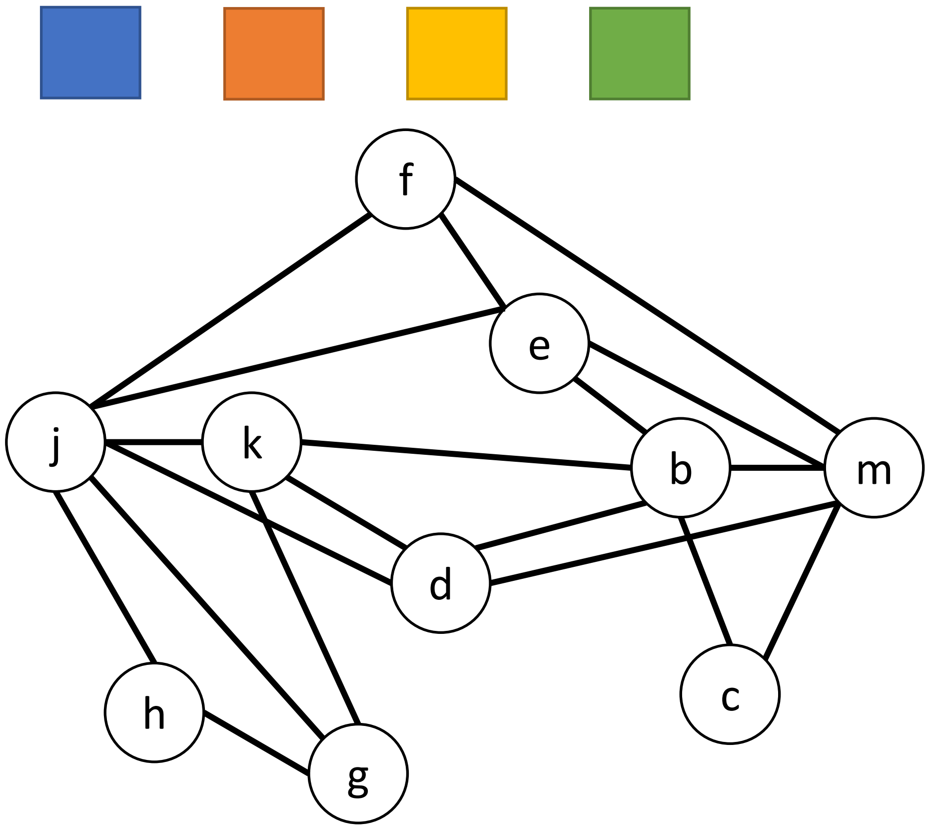
- Let K = number of registers
- Let S = empty stack
- While graph not empty:
  - If there exists a node m with fewer than K neighbors:
    - Remove m from the graph, push it on S
    - Guaranteed that we will be able to find a color for m
  - Otherwise:
    - Pick a node m, remove it from the graph, push it on S (we may end up spilling it)

Coloring by simplification: **Select**

- While S not empty:
  - Pop m from S
  - If there is a color (register) available for m:
    - Choose an available color (register) for m and add it back to the graph
  - Otherwise:
    - Spill m – put it in the next stack slot

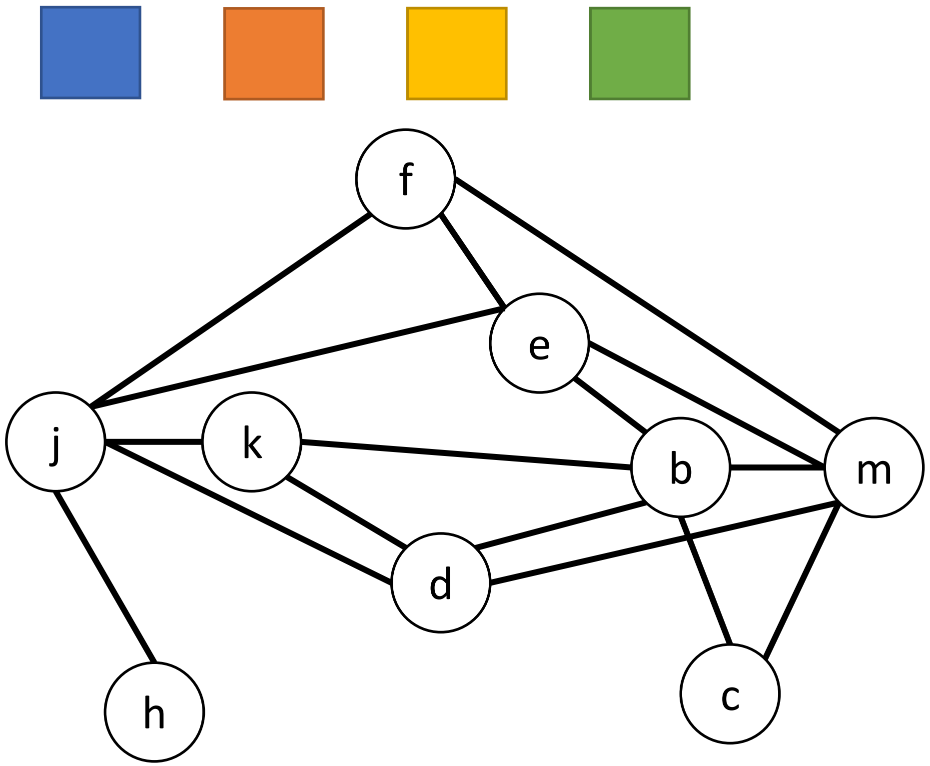
Graph Coloring Example (Appel)

```
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       % Move
k = m + 4
j = b       % Move
            % d, k, j live-out
```



Graph Coloring Example (Appel)

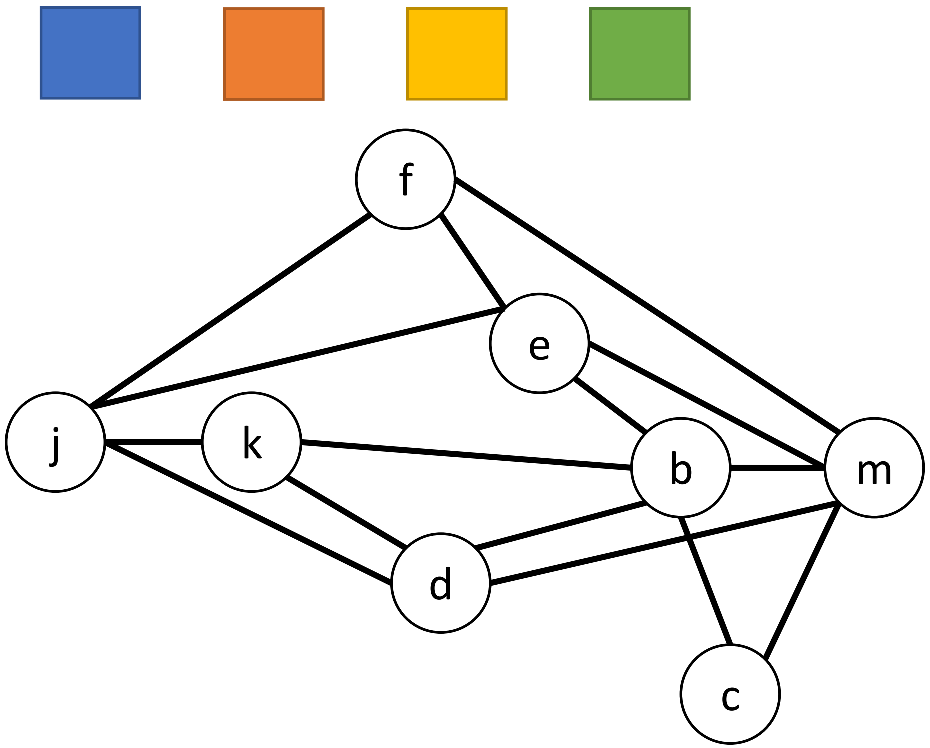
g



Graph Coloring Example (Appel)

h

g

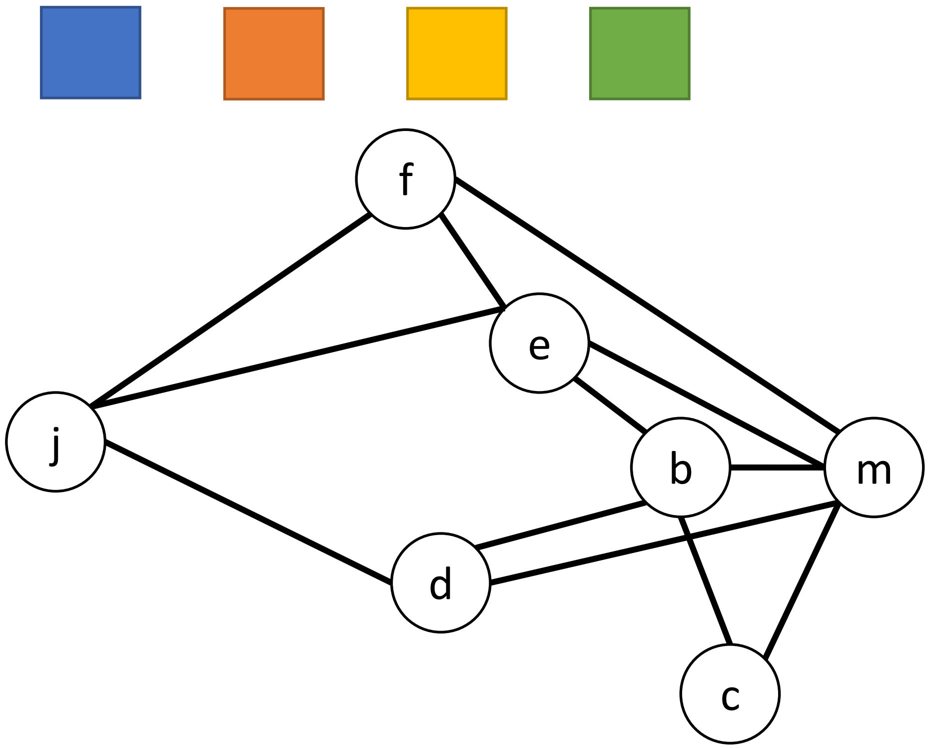


Graph Coloring Example (Appel)

k

h

g



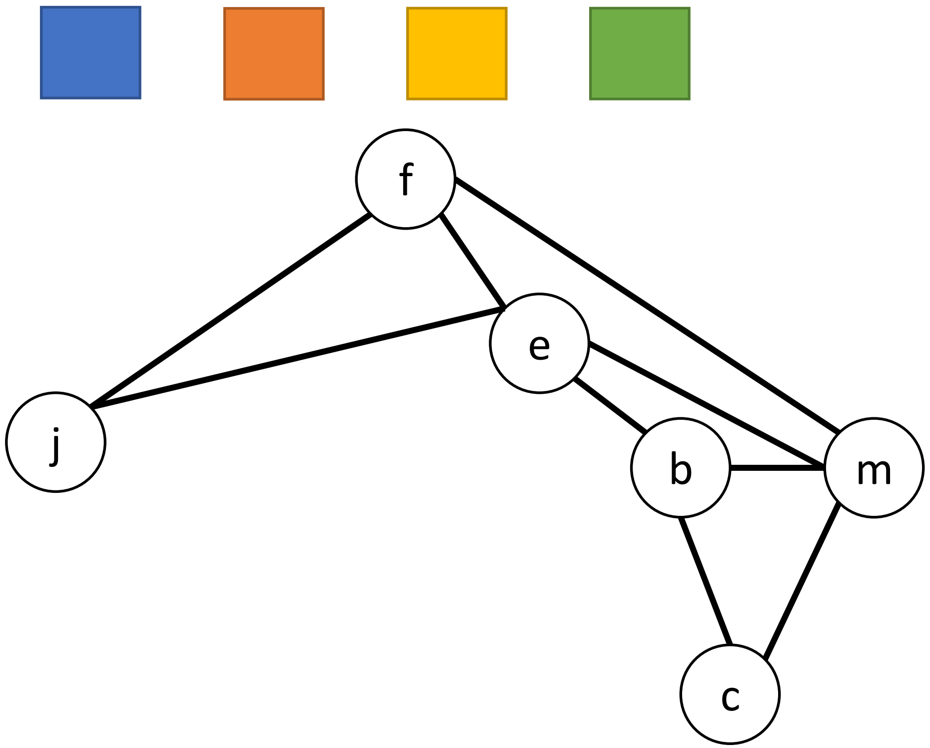
Graph Coloring Example (Appel)

d

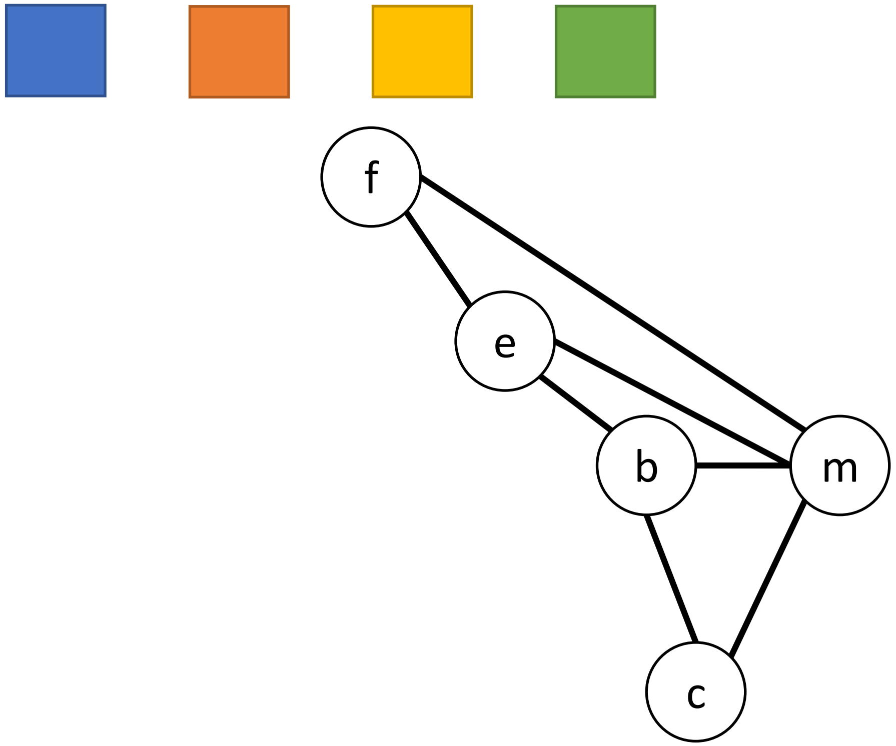
k

h

g



Graph Coloring Example (Appel)




```
j
d
k
h
g
```

Graph Coloring Example (Appel)

e

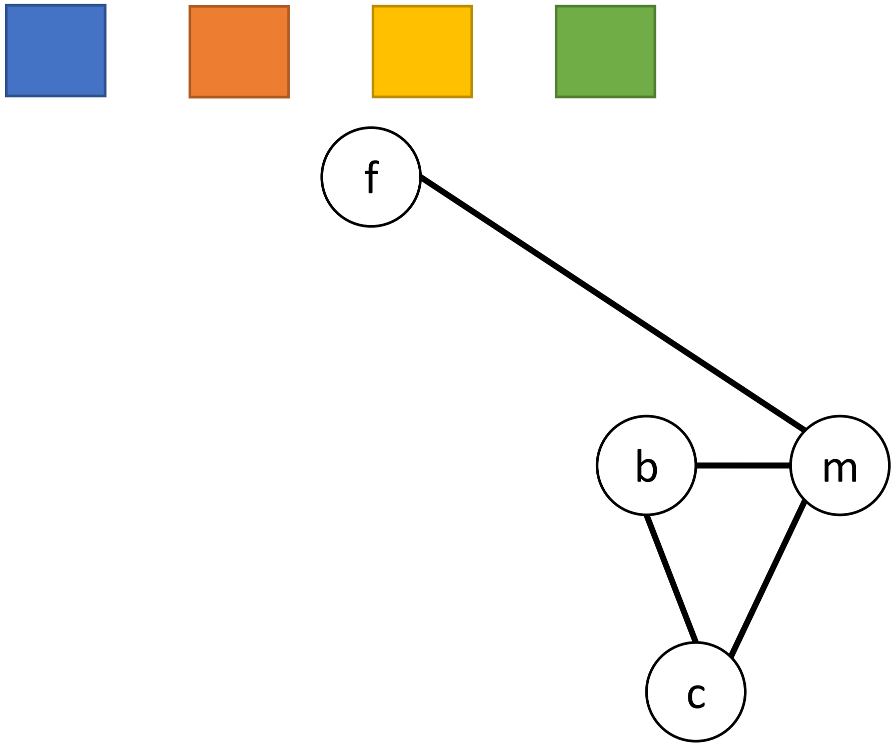
j

d

k

h

g



Graph Coloring Example (Appel)

f

e

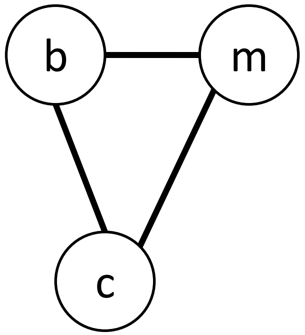
j

d

k

h

g



Graph Coloring Example (Appel)

b

f

e

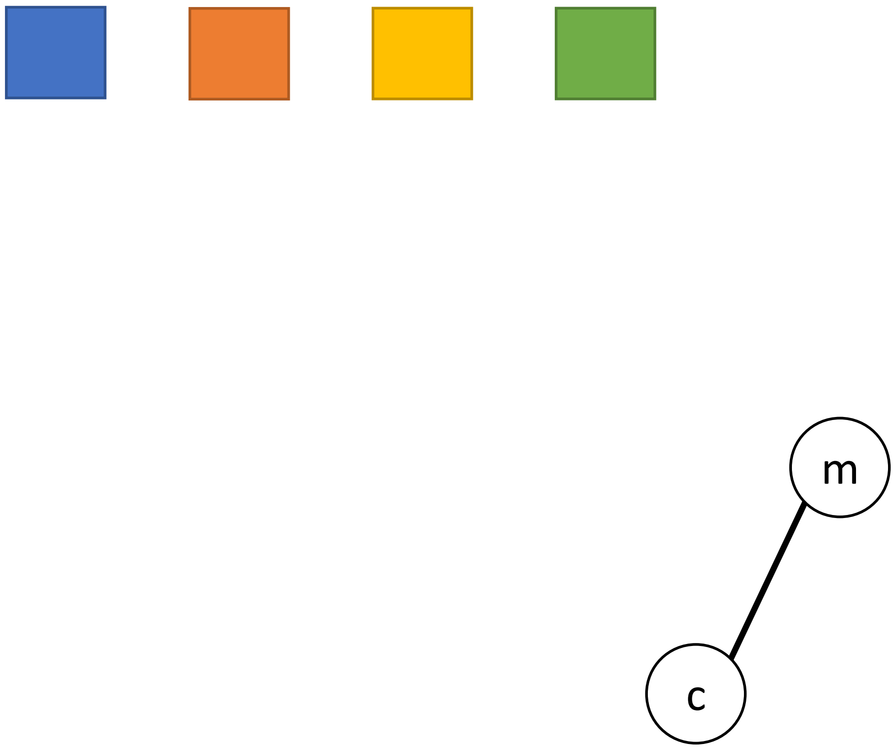
j

d

k

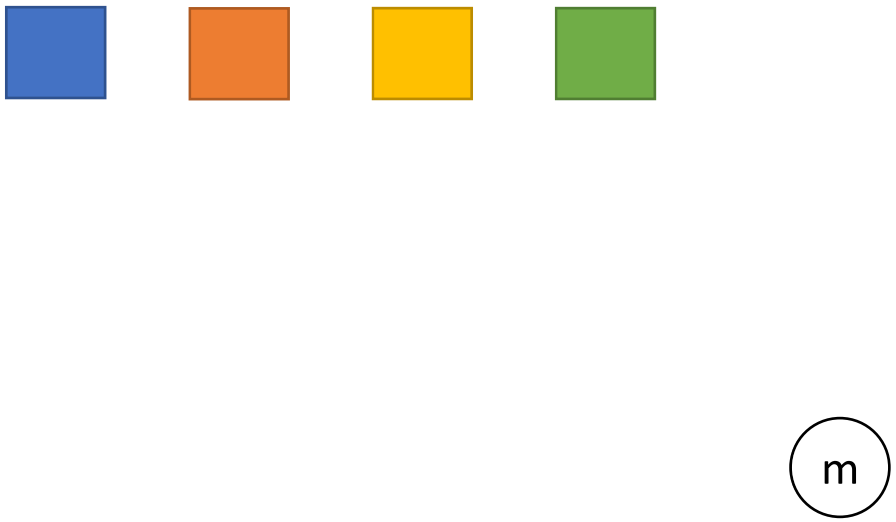
h

g



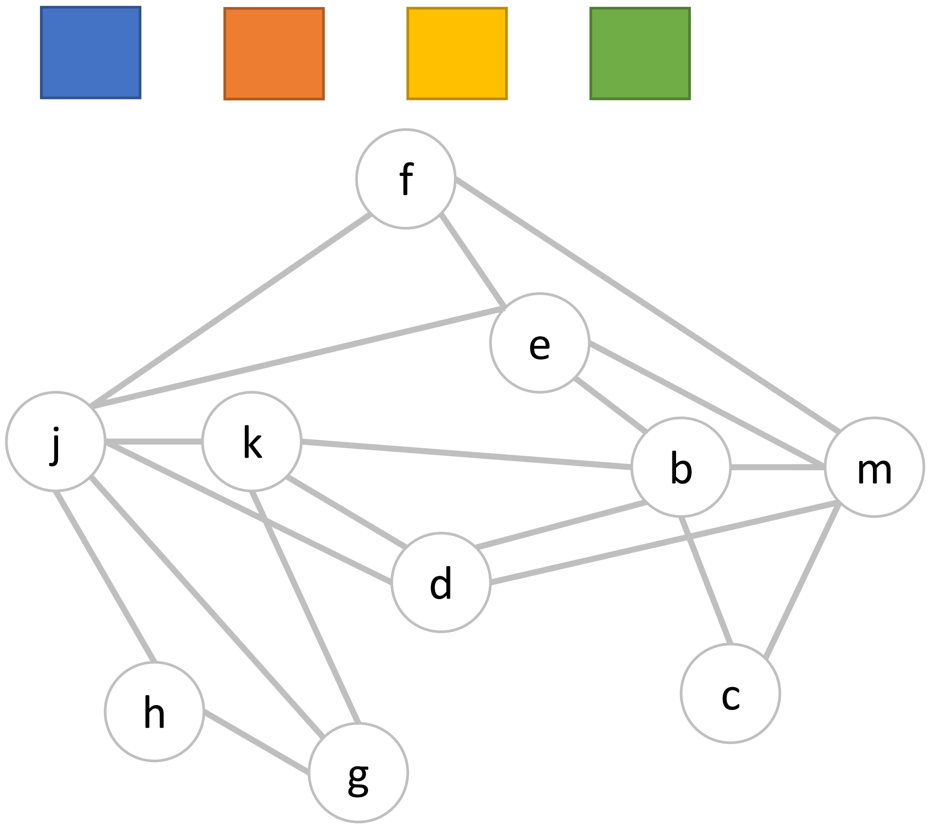
Graph Coloring Example (Appel)

c
b
f
e
j
d
k
h
g



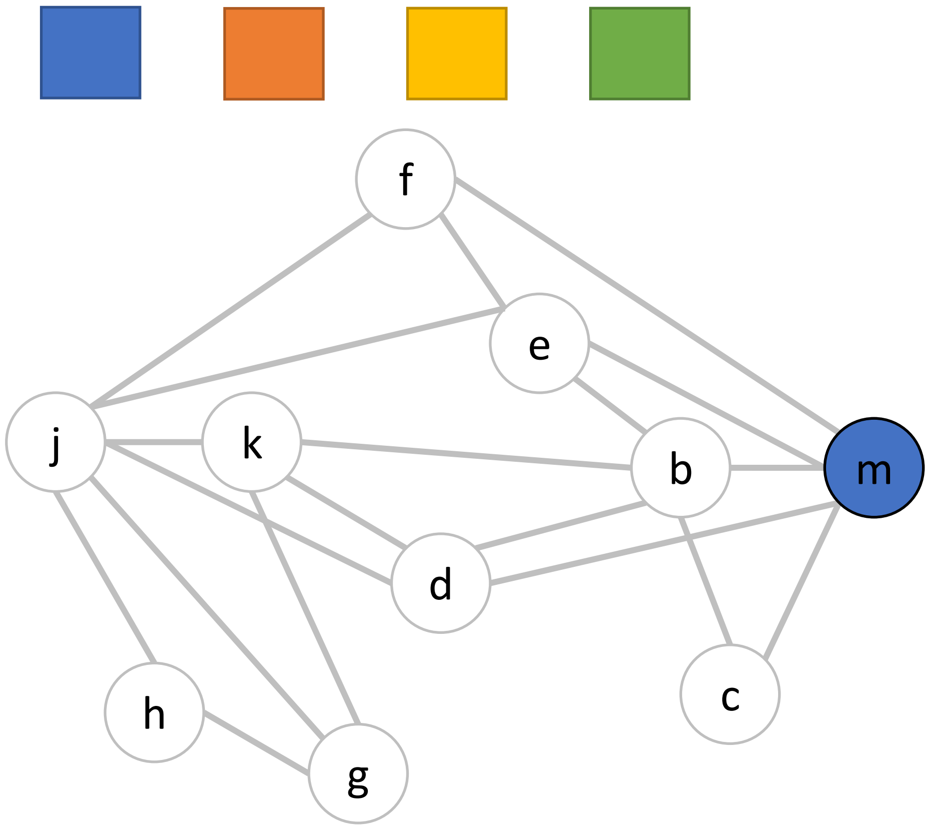
Graph Coloring Example (Appel)

m
c
b
f
e
j
d
k
h
g



Graph Coloring Example (Appel)

c
b
f
e
j
d
k
h
g



Graph Coloring Example (Appel)

b

f

e

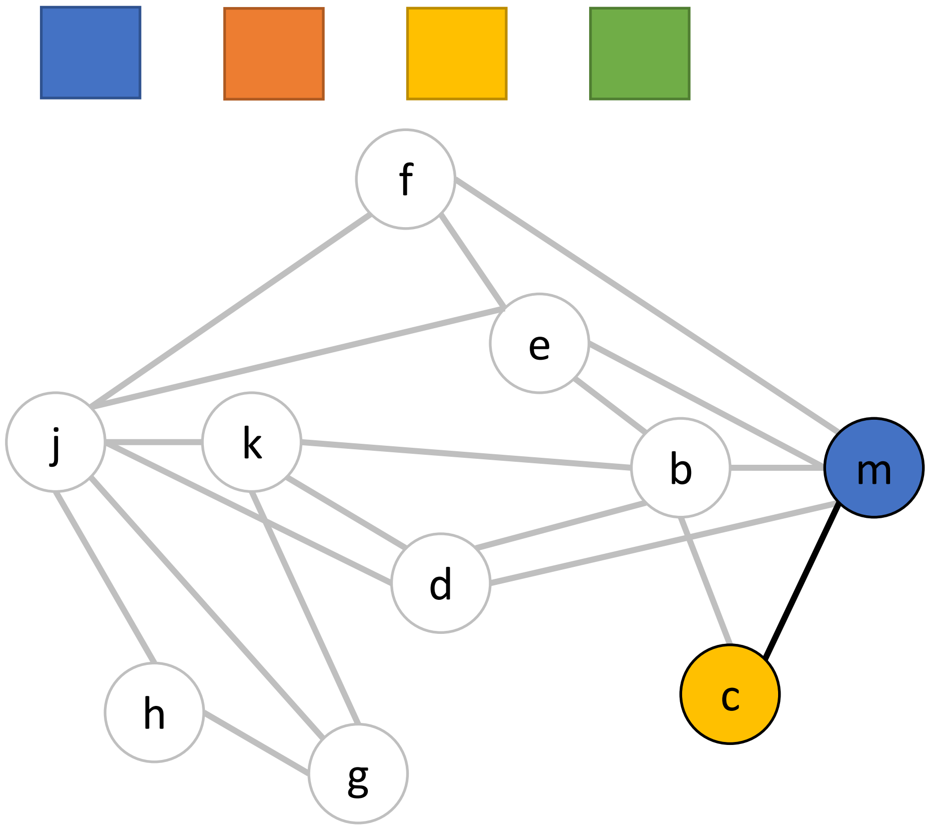
j

d

k

h

g



Graph Coloring Example (Appel)

f

e

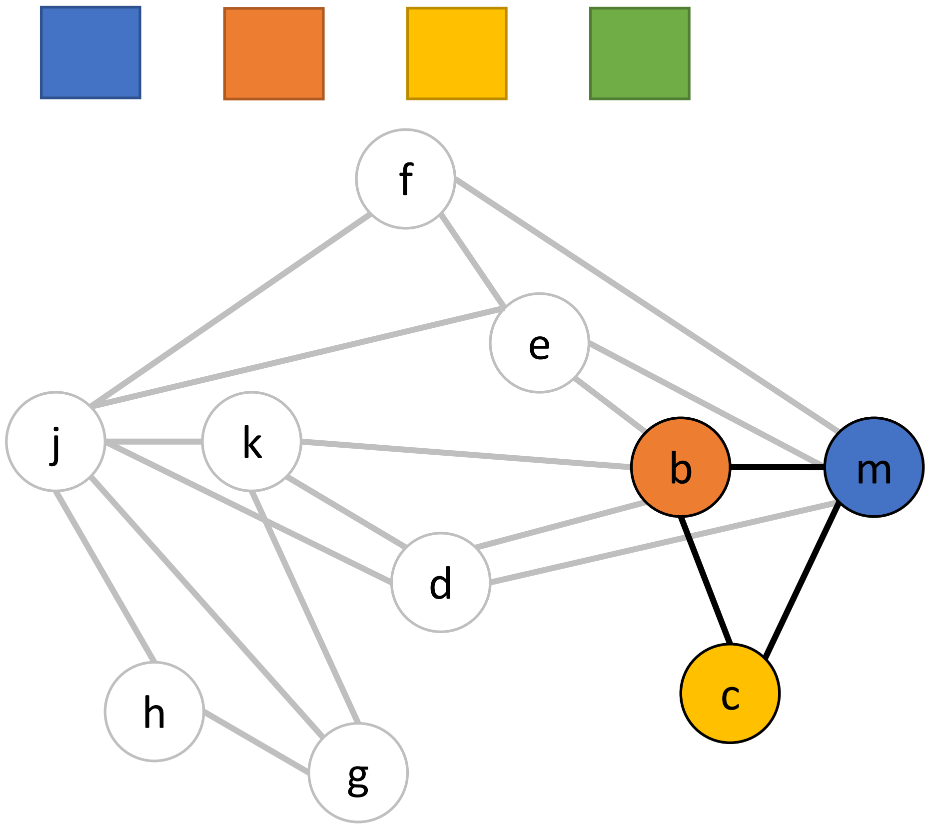
j

d

k

h

g



Graph Coloring Example (Appel)

e

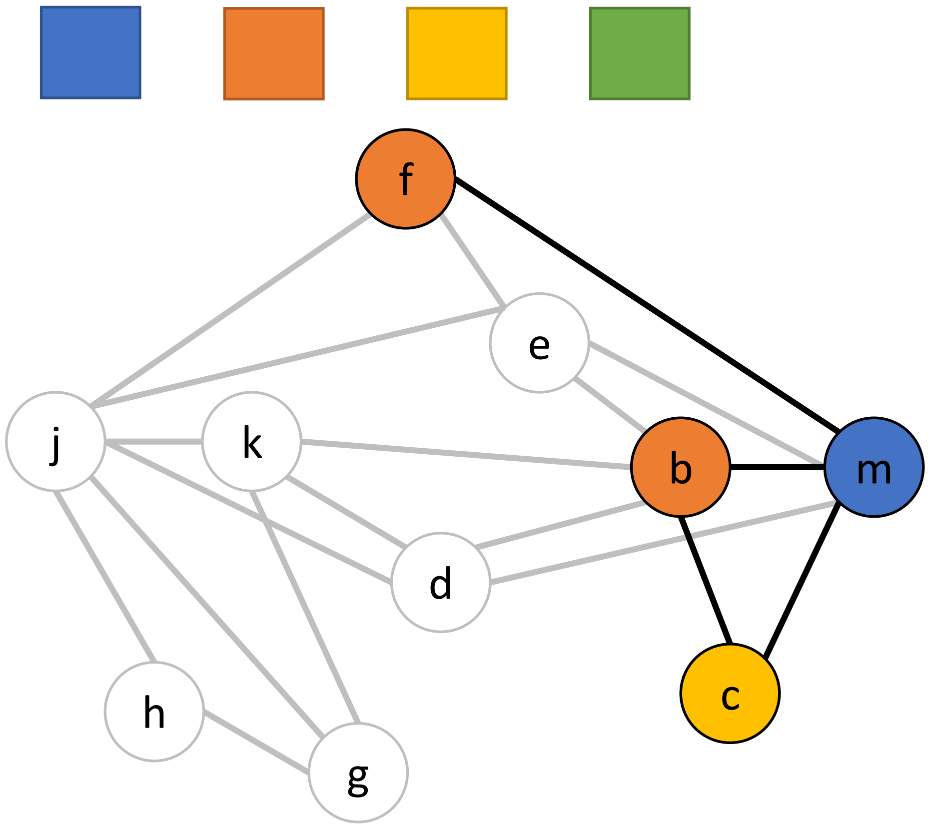
j

d

k

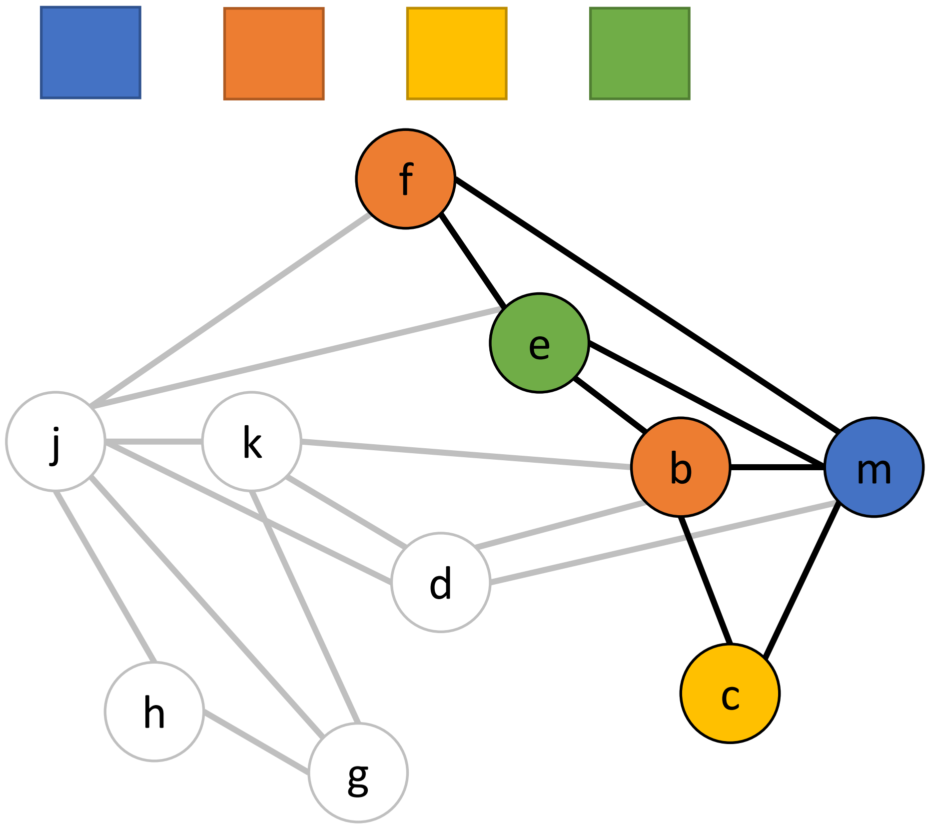
h

g



Graph Coloring Example (Appel)

j
d
k
h
g



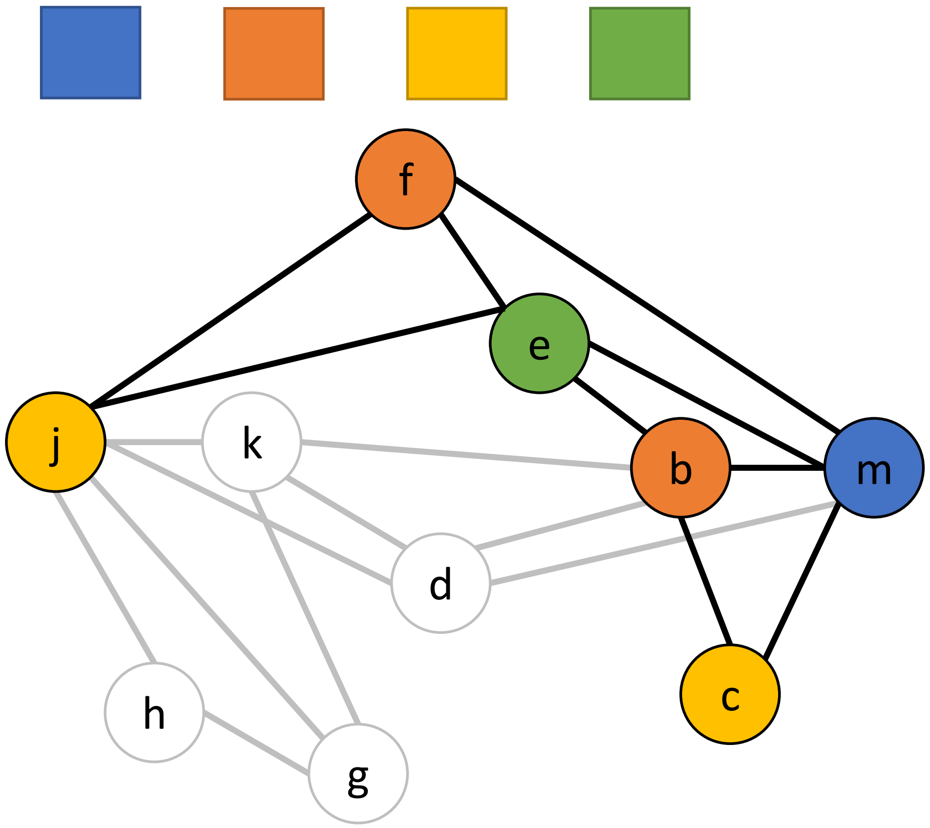
Graph Coloring Example (Appel)

d

k

h

g

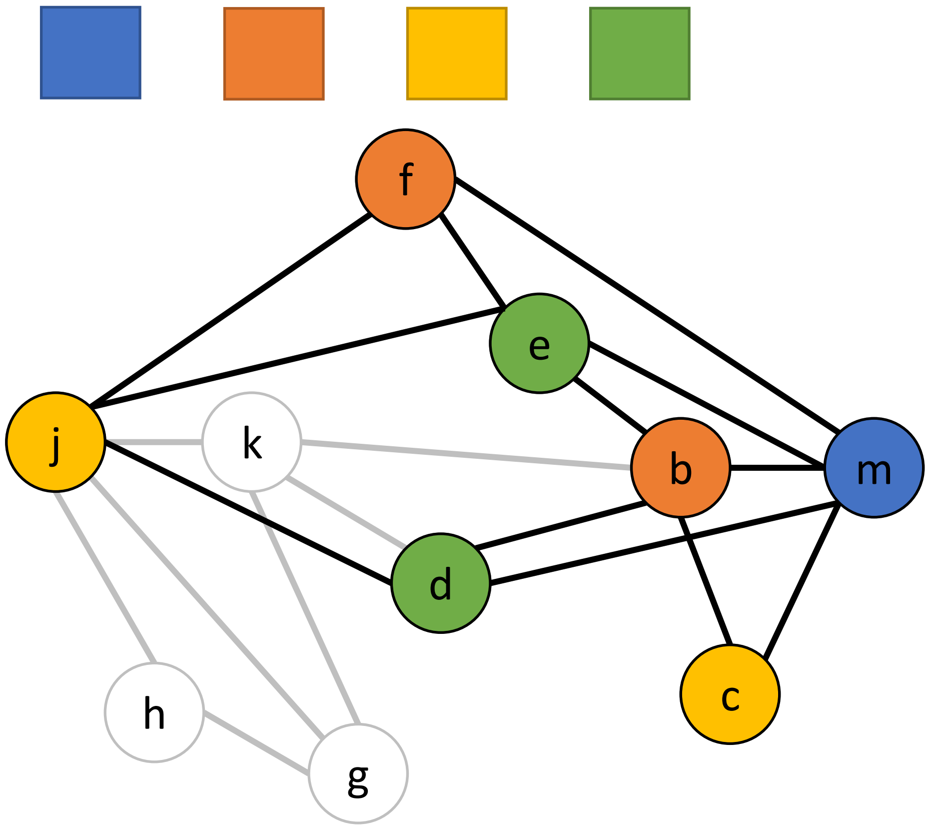


Graph Coloring Example (Appel)

k

h

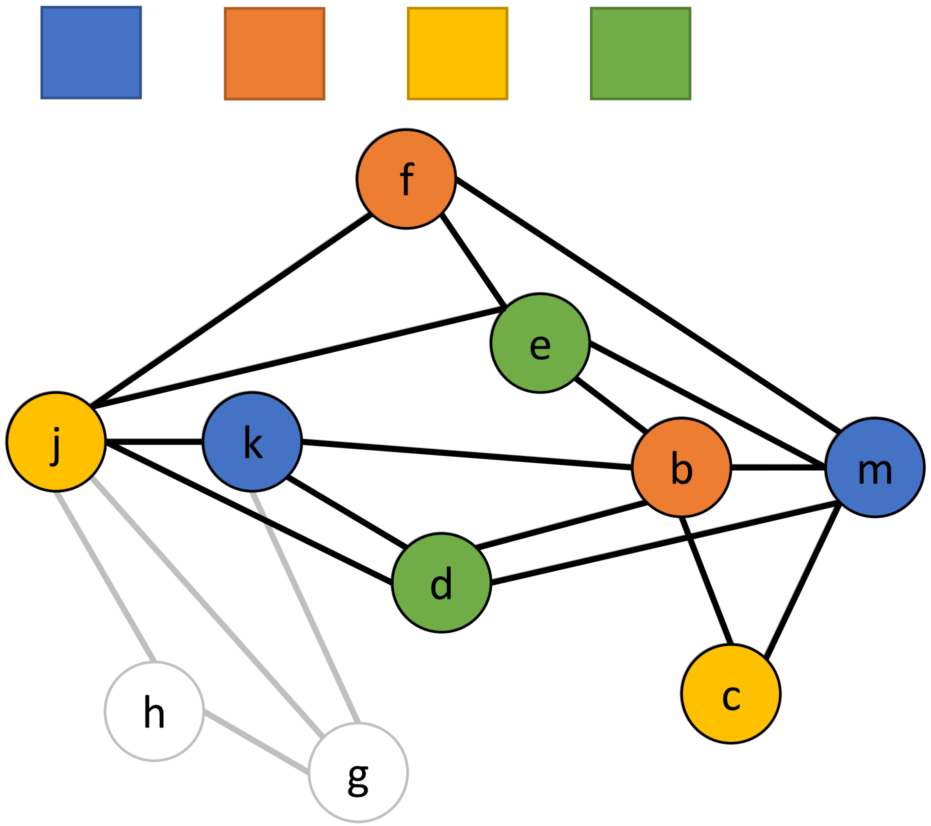
g



Graph Coloring Example (Appel)

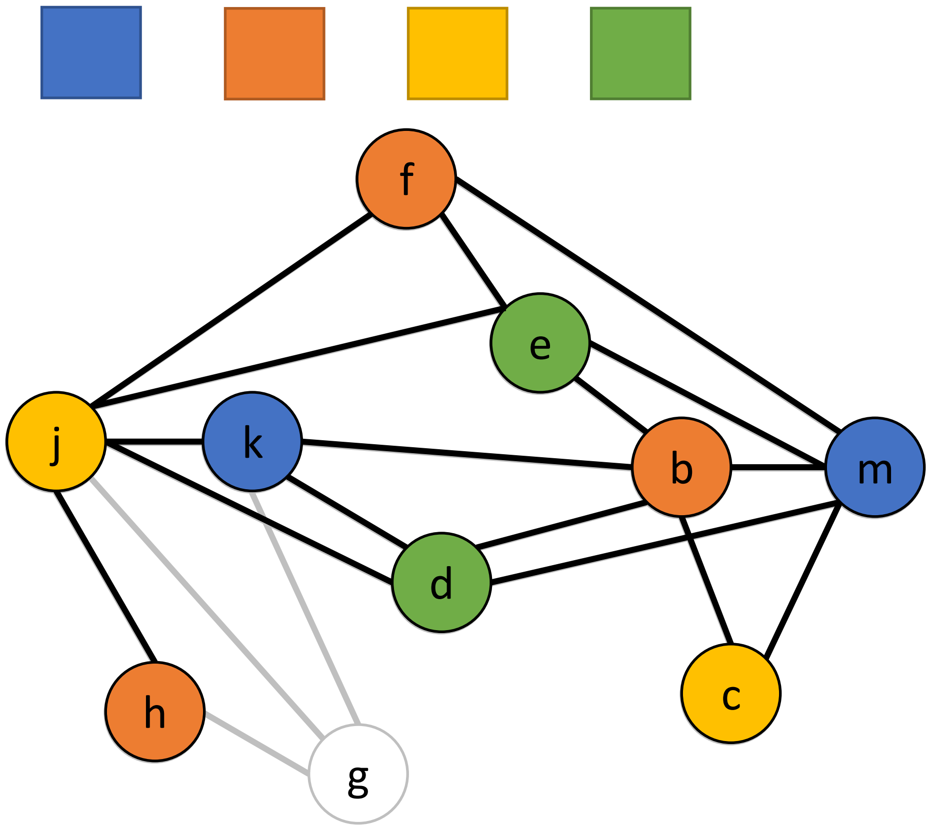
h

g

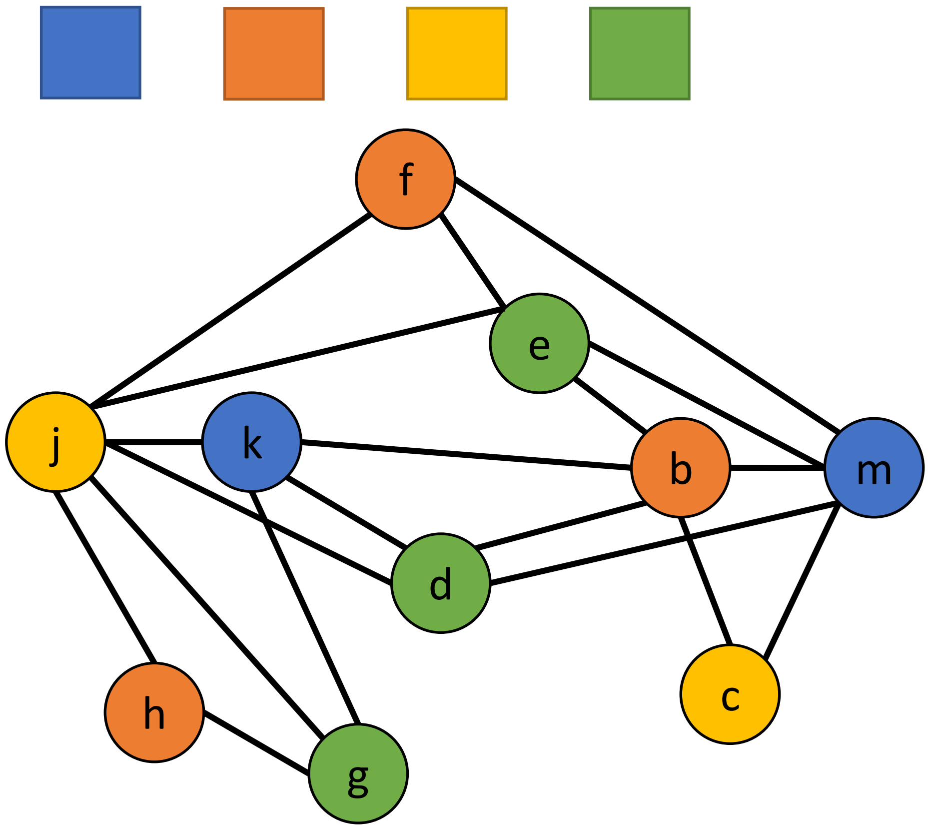


Graph Coloring Example (Appel)

g

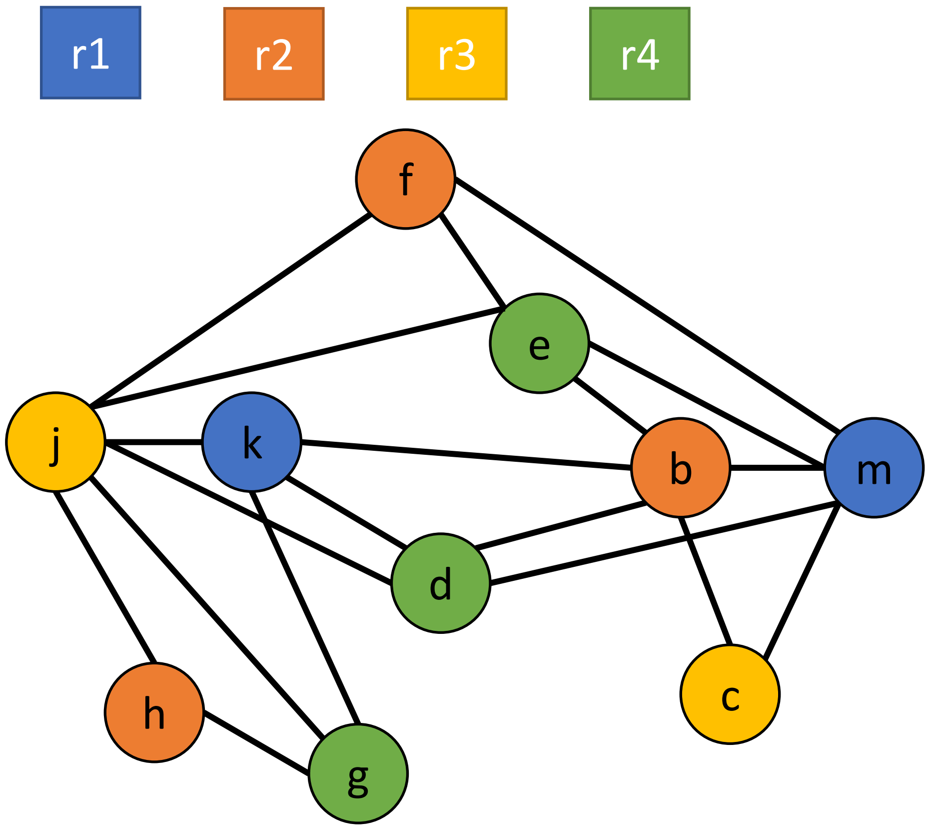


Graph Coloring Example (Appel)



Graph Coloring Example (Appel)

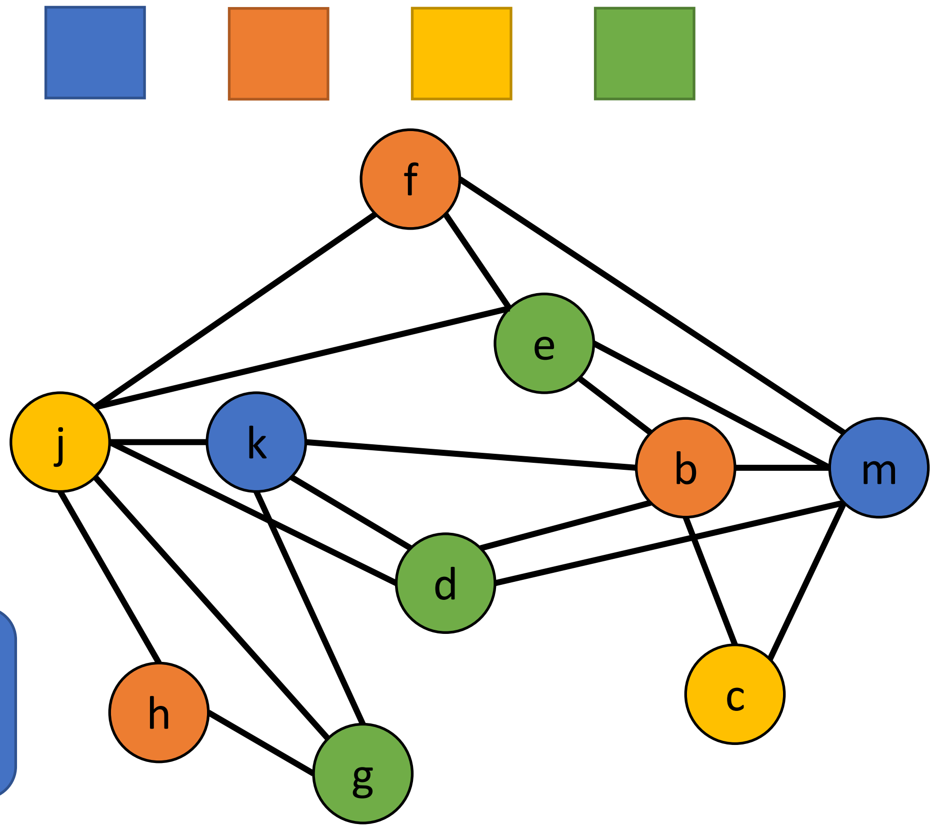
```
g = mem[j + 12]
h = k - 1
f = g * h
e = mem[j + 8]
m = mem[j + 16]
b = mem[f]
c = e + 8
d = c
k = m + 4
j = b
```



Graph Coloring Example (Appel)

```
r4 = mem[r3 + 12]
r2 = r1 - 1
r2 = r4 * r2
r4 = mem[r3 + 8]
r1 = mem[r3 + 16]
r2 = mem[r2]
r3 = e + 8
r4 = r3
r1 = r1 + 4
r3 = r2
```

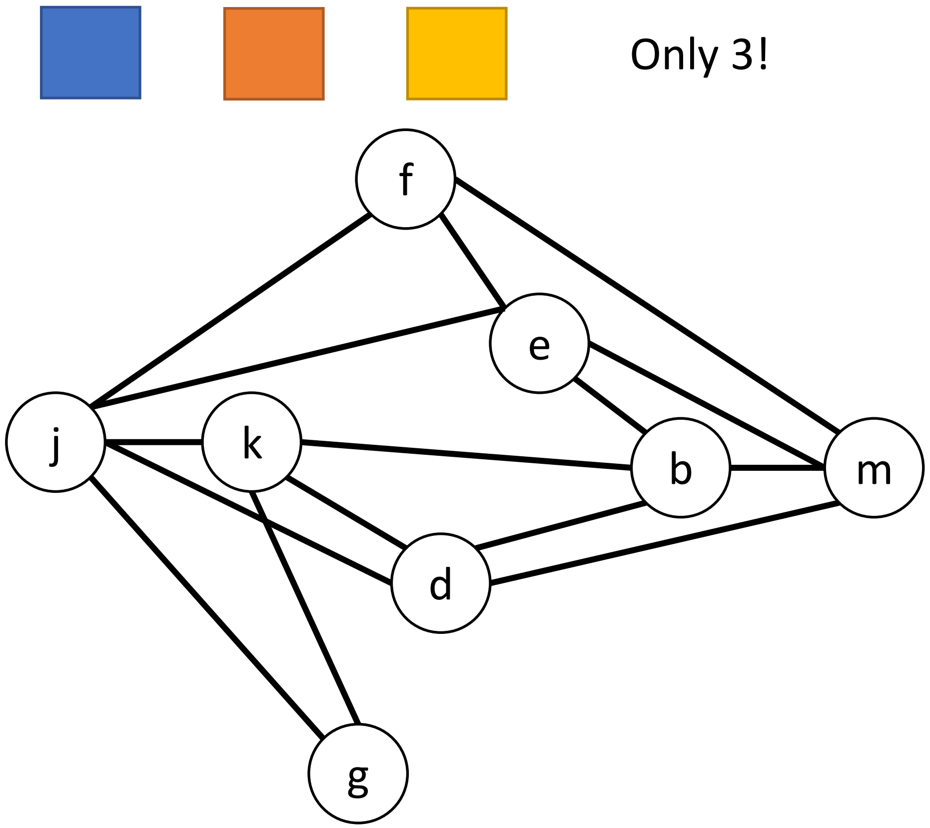
Next time:
Avoid these



Graph Coloring Example (Appel)

c

h

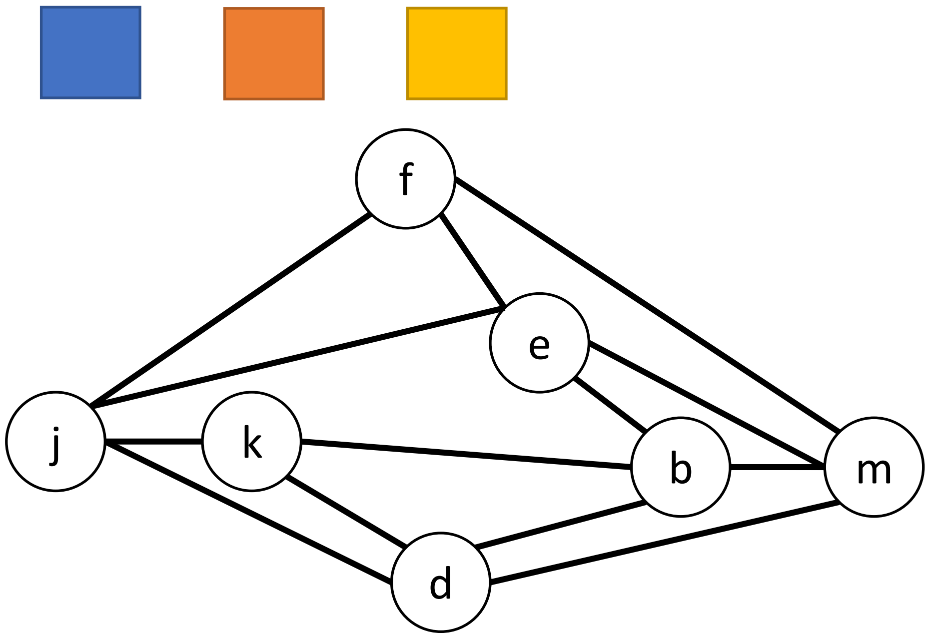


Graph Coloring Example (Appel)

g

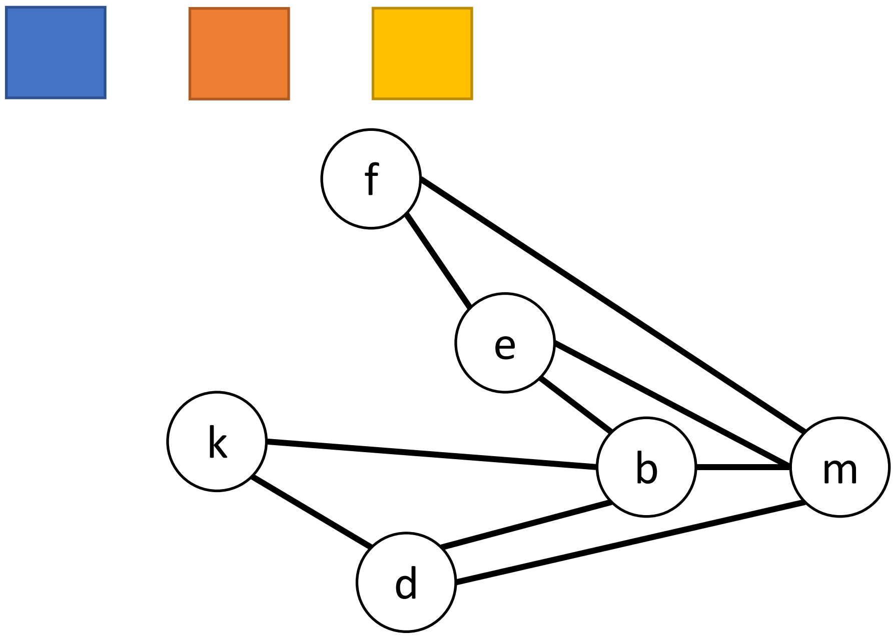
c

h



Graph Coloring Example (Appel)

j
g
c
h



Graph Coloring Example (Appel)

f

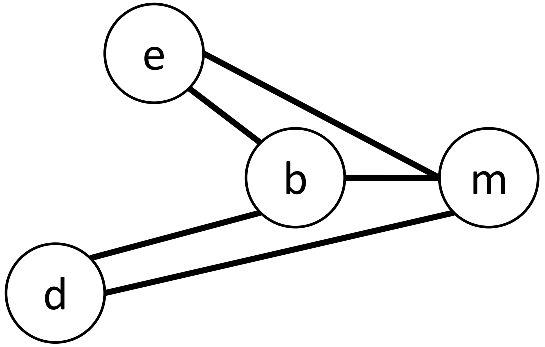
k

j

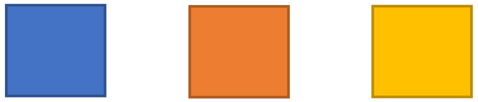
g

c

h



Graph Coloring Example (Appel)



e

d

f

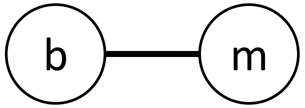
k

j

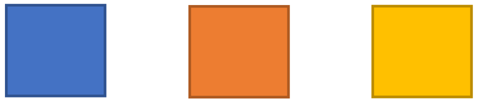
g

c

h



Graph Coloring Example (Appel)

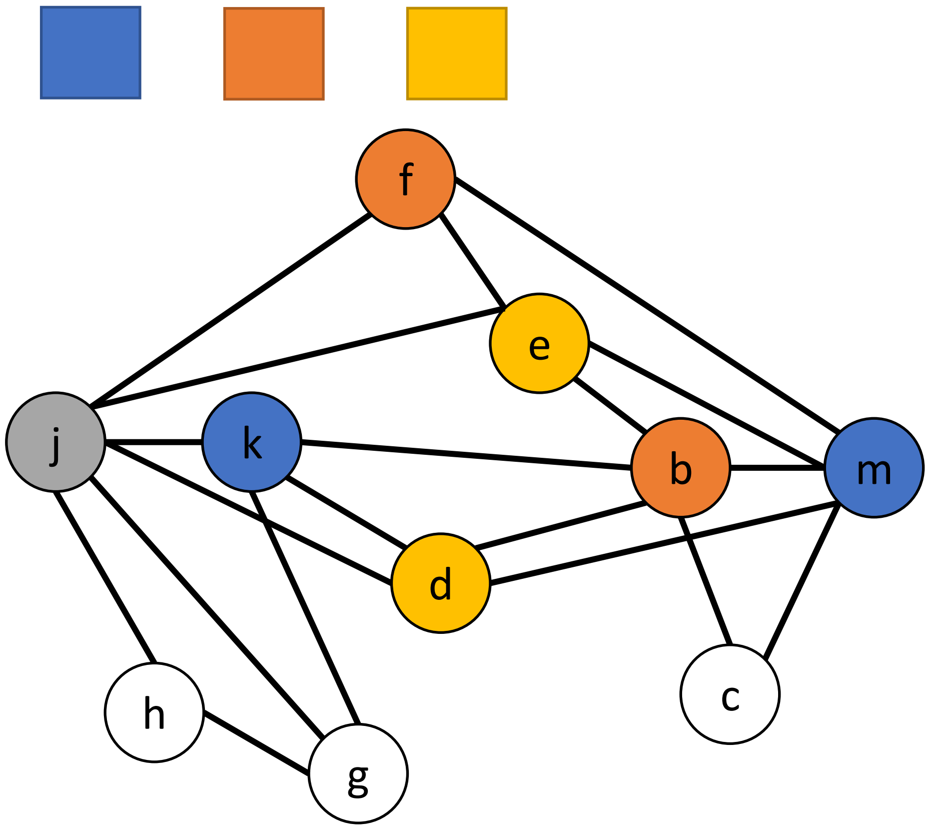


b
m
e
d
f
k
j
g
c
h

Graph Coloring Example (Appel)

j
g
c
h

Well, gotta spill (we might have gotten lucky and still found a color)

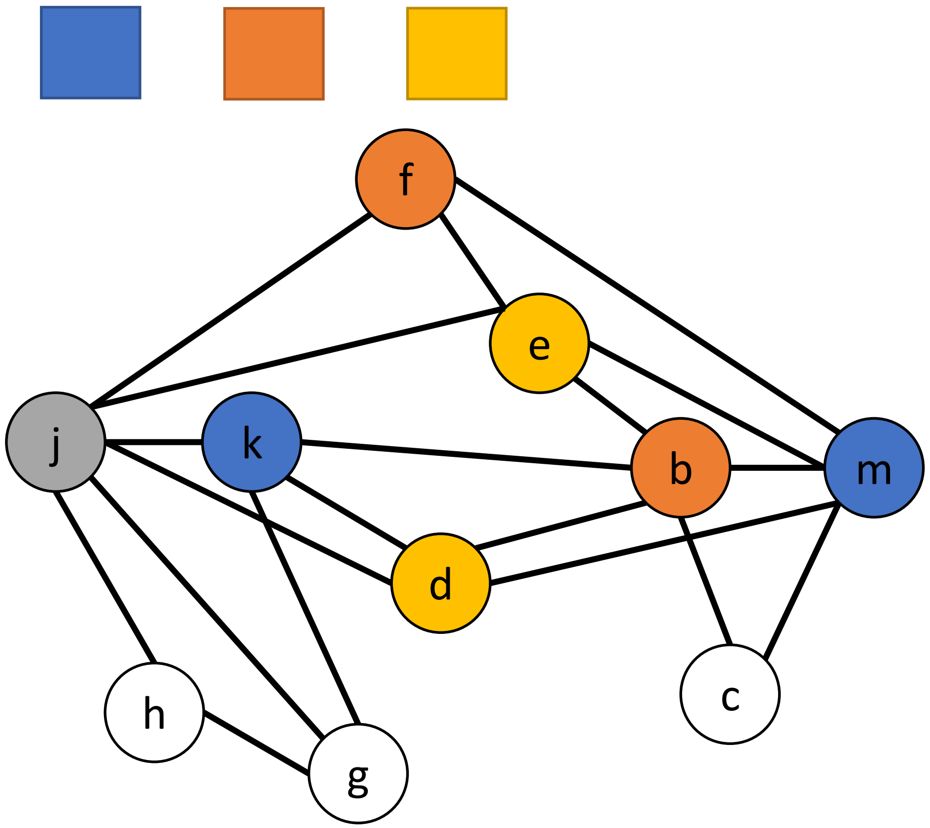


Graph Coloring Example (Appel)

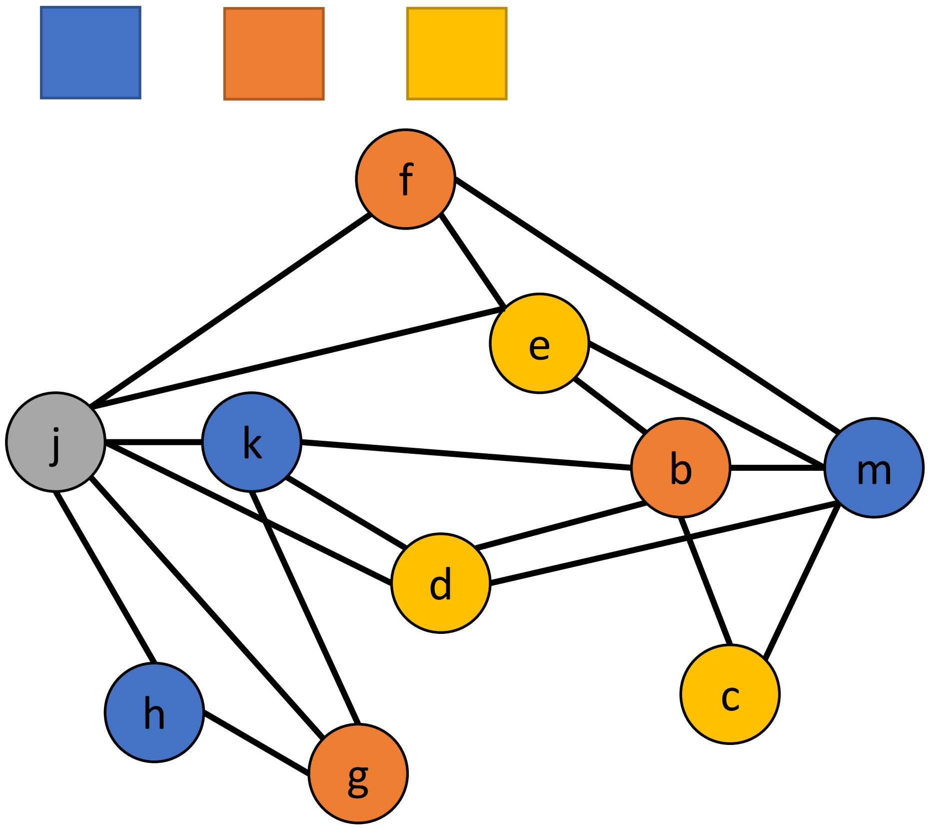
g

c

h

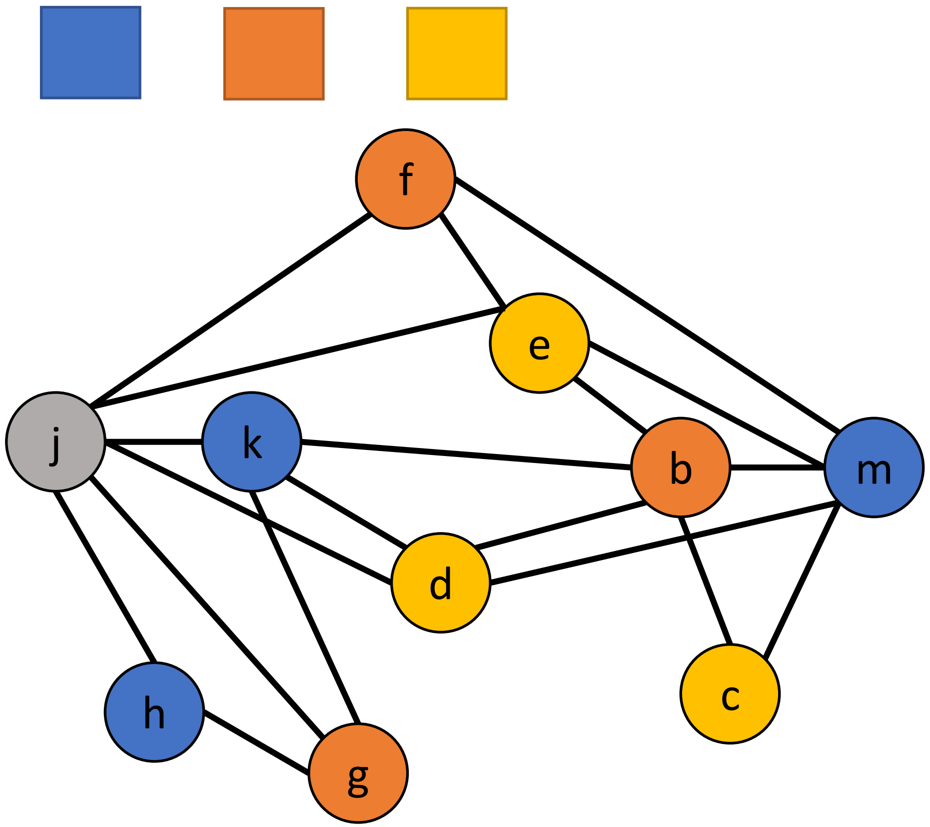


Graph Coloring Example (Appel)



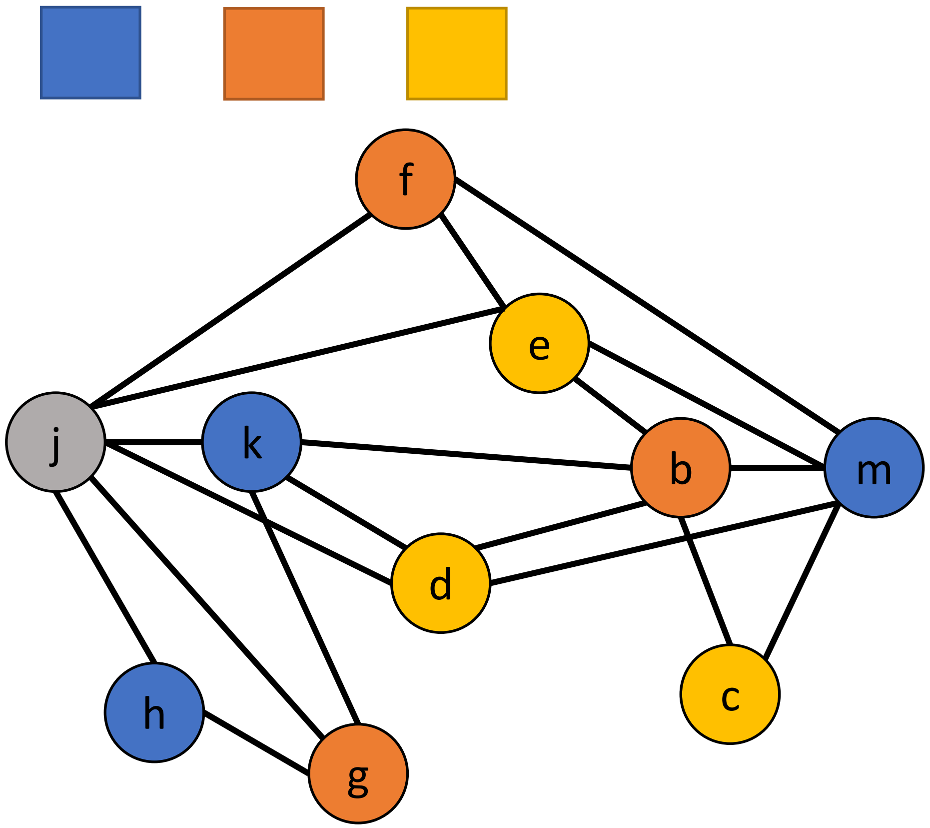
We need to load j from memory… into what?

```
r2 = mem[j + 12]
r1 = r1 – 1
r2 = r2 * r1
r3 = mem[j + 8]
r1 = mem[j + 16]
r2 = mem[r2]
r3 = r3 + 8
r3 = r3
r1 = r1 + 4
j = r2
```



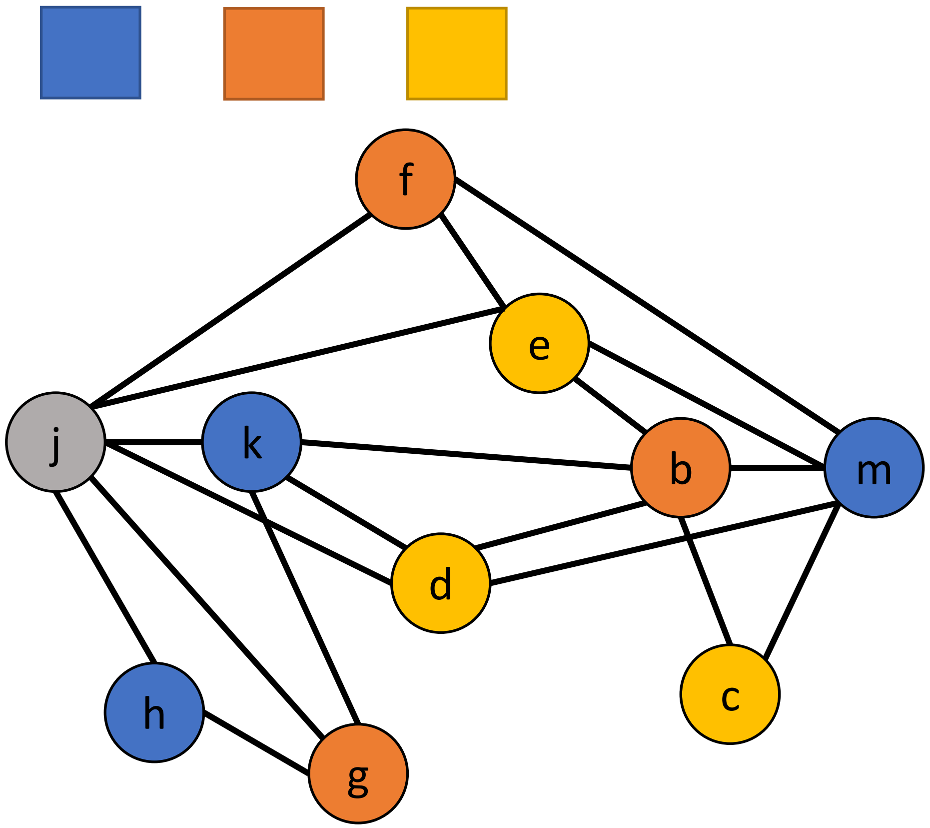
Option 1: Move to a temp, do reg alloc again

```
temp1 = stack[0]
r2 = mem[temp1 + 12]
r1 = r1 – 1
r2 = r2 * r1
temp1 = stack[0]
r3 = mem[temp1 + 8]
temp1 = stack[0]
r1 = mem[temp1 + 16]
r2 = mem[r2]
r3 = r3 + 8
r3 = r3
r1 = r1 + 4
temp1 = r2
stack[0] = temp1
```



Option 2: Reserve a register or two for this

```
r4 = stack[0]
r2 = mem[r4 + 12]
r1 = r1 – 1
r2 = r2 * r1
r4 = stack[0]
r3 = mem[r4 + 8]
r4 = stack[0]
r1 = mem[r4 + 16]
r2 = mem[r2]
r3 = r3 + 8
r3 = r3
r1 = r1 + 4
r4 = r2
stack[0] = r4
```

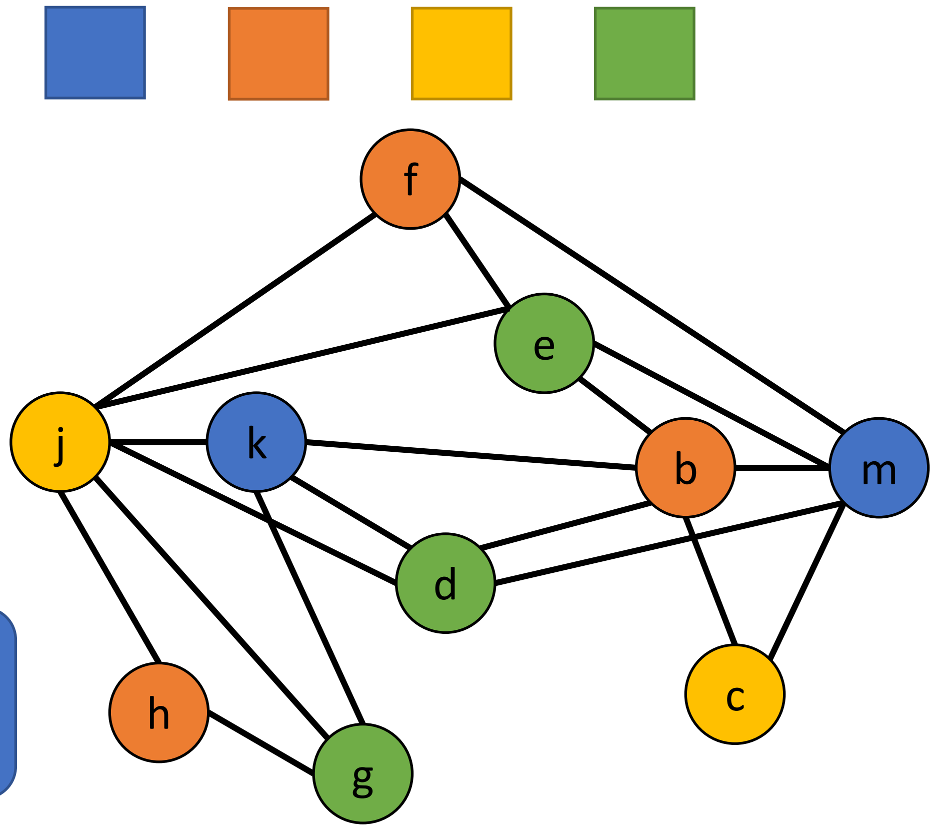


Graph Coloring Example (Appel)

```
r4 = mem[r3 + 12]
r2 = r1 - 1
r2 = r4 * r2
r4 = mem[r3 + 8]
r1 = mem[r3 + 16]
r2 = mem[r2]
r3 = e + 8
r4 = r3
r1 = r1 + 4
r3 = r2
```

This

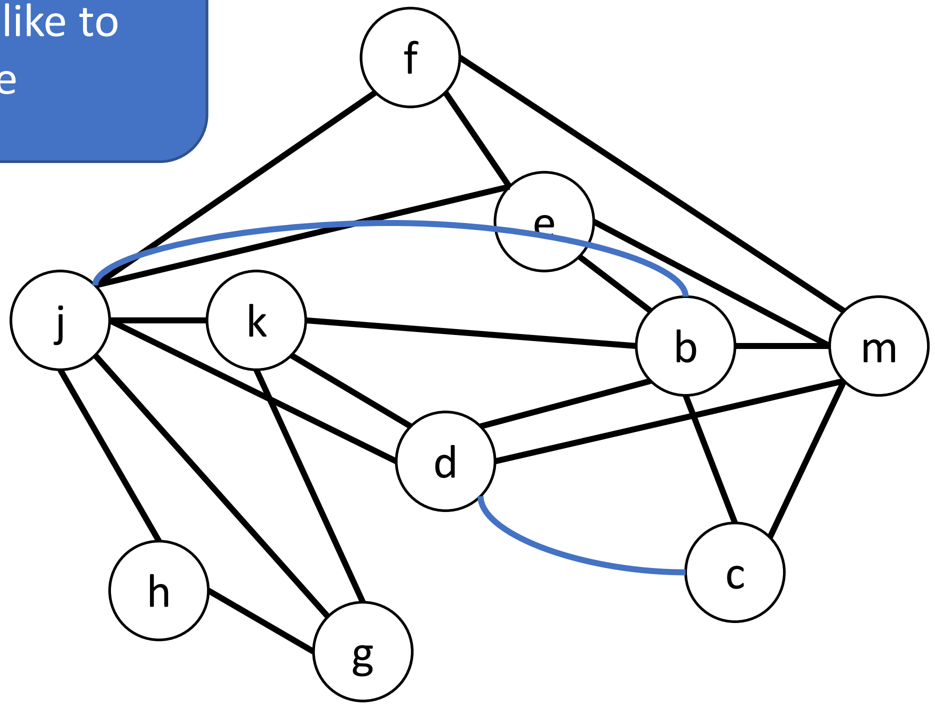
~~Next time:~~
Avoid these



Coalescing: Combining nodes to eliminate moves

```
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
```

Blue edge + no black edge: would like to coalesce

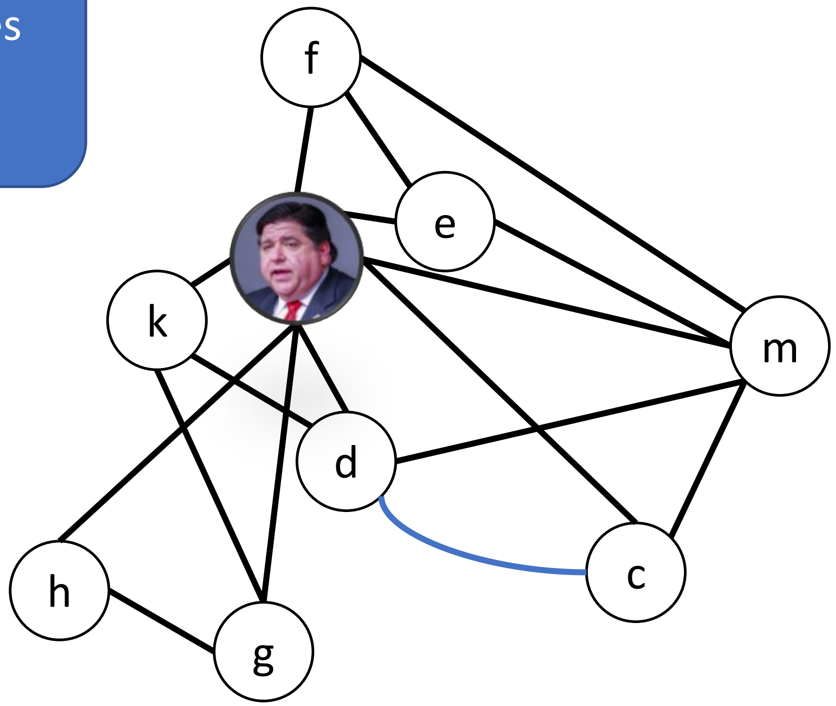


Coalescing unsafely can make a graph uncolorable

```
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~~

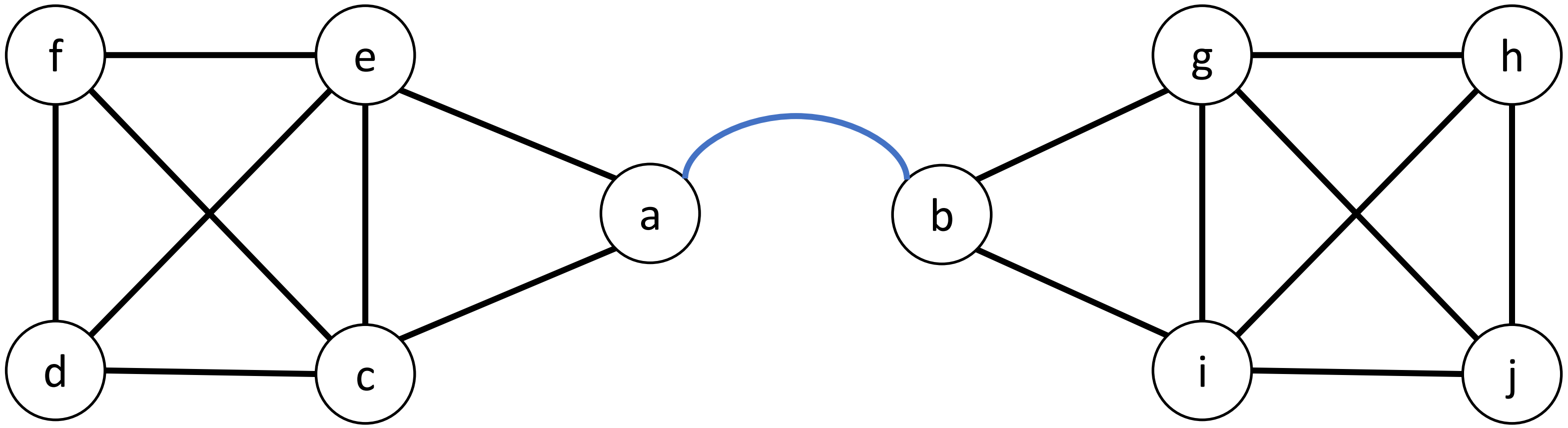
jb has all the edges from j and b!



We'd rather move than spill

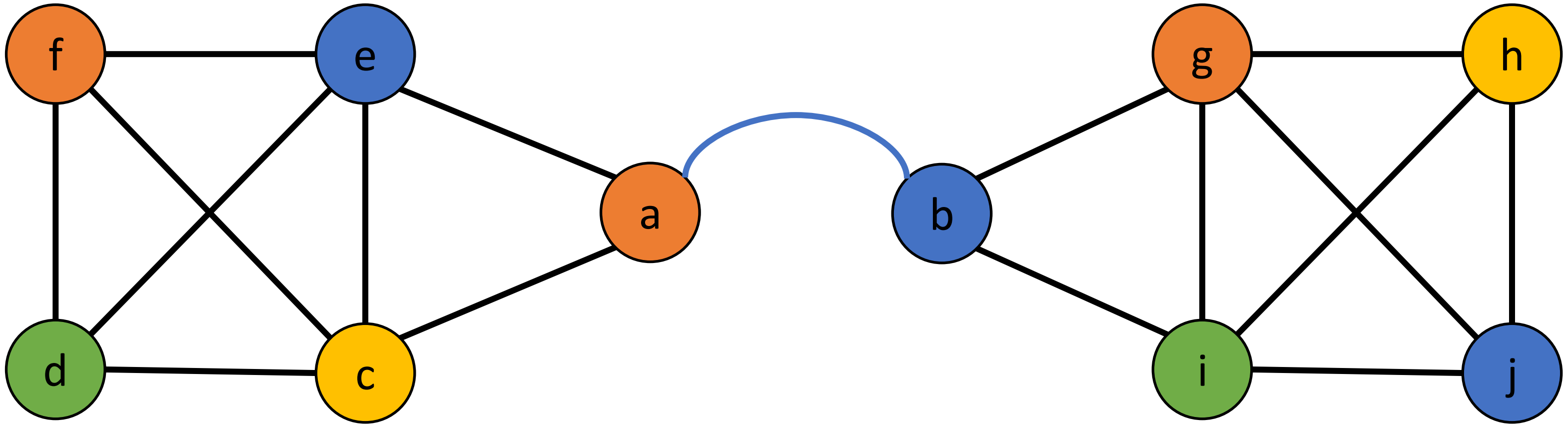
Conservative coalescing strategies will always keep a graph colorable

- Briggs: *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of degree >= K
  - (Recall: K = number registers/colors)



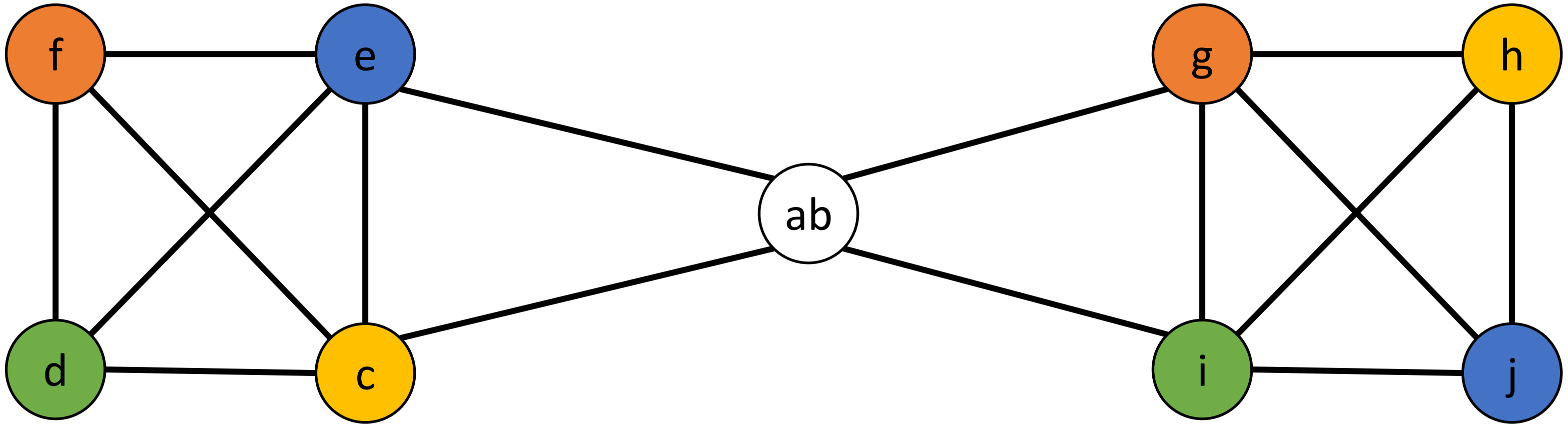
Conservative coalescing strategies will always keep a graph colorable

- Briggs: *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of degree >= K
  - (Recall: K = number registers/colors)



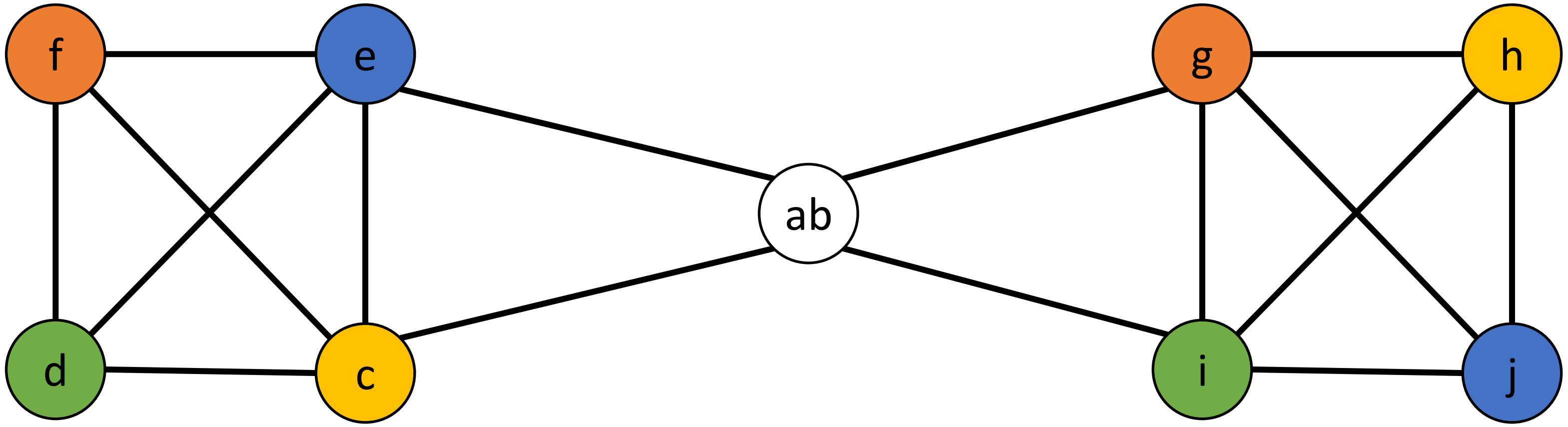
Conservative coalescing strategies will always keep a graph colorable

- Briggs: *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of degree >= K
  - (Recall: K = number registers/colors)



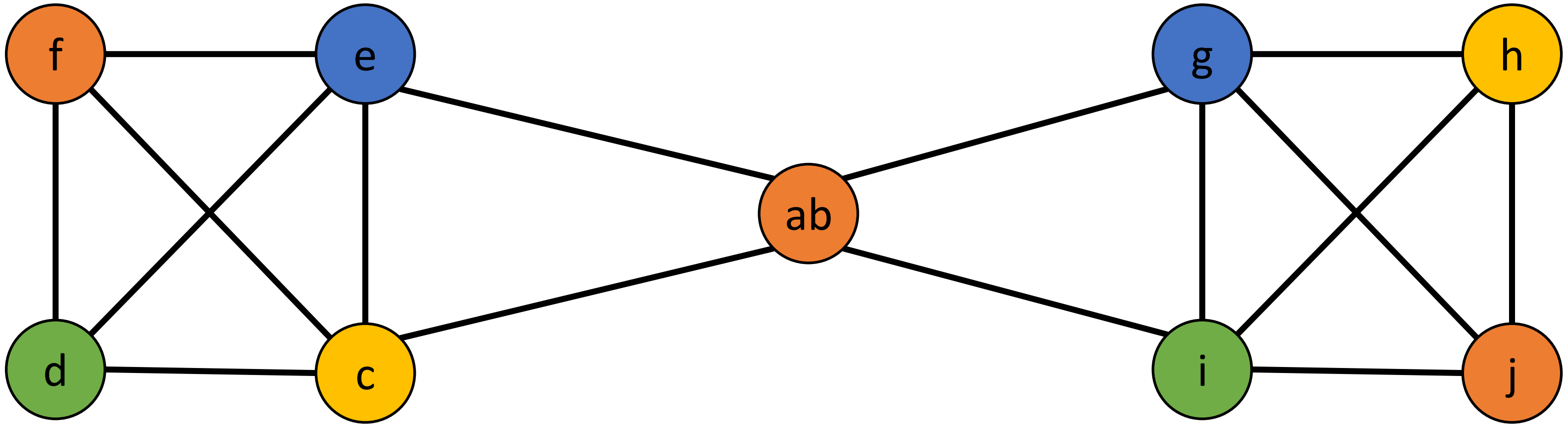
Conservative coalescing strategies will always keep a graph colorable

- Briggs is *conservative*:
  - Coalescing nodes following Briggs is guaranteed not to make a graph uncolorable
  - Briggs might miss nodes that could still be safely coalesced



Conservative coalescing strategies will always keep a graph colorable

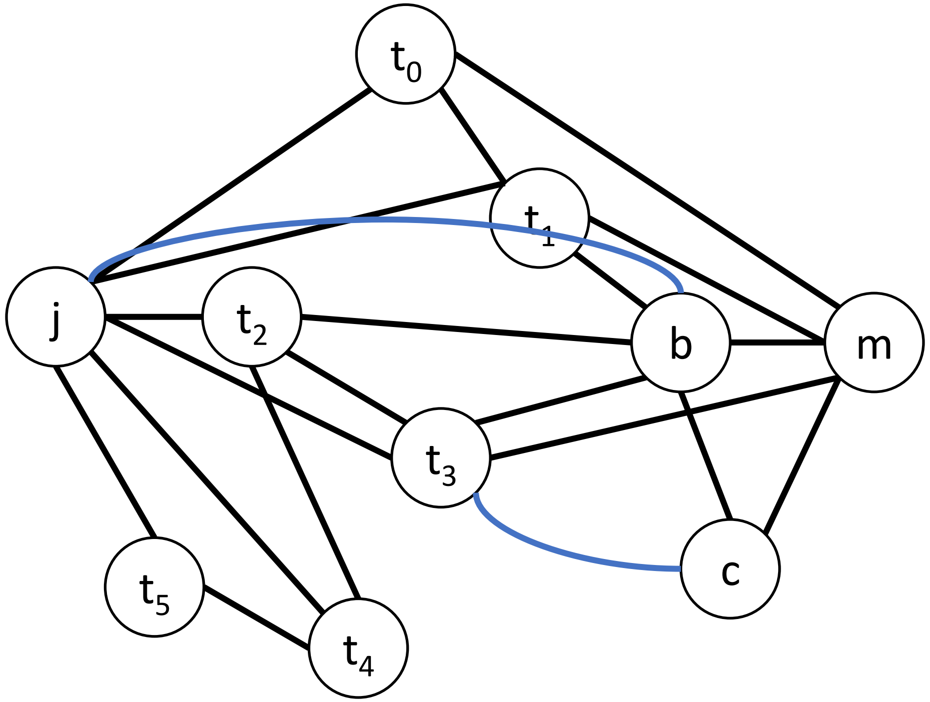
- Briggs is *conservative*:
  - Coalescing nodes following Briggs is guaranteed not to make a graph uncolorable
  - Briggs might miss nodes that could still be safely coalesced



Conservative coalescing strategies will always keep a graph colorable

- George: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either:
  - *t* already interferes with *b* or
  - *t* has degree < K

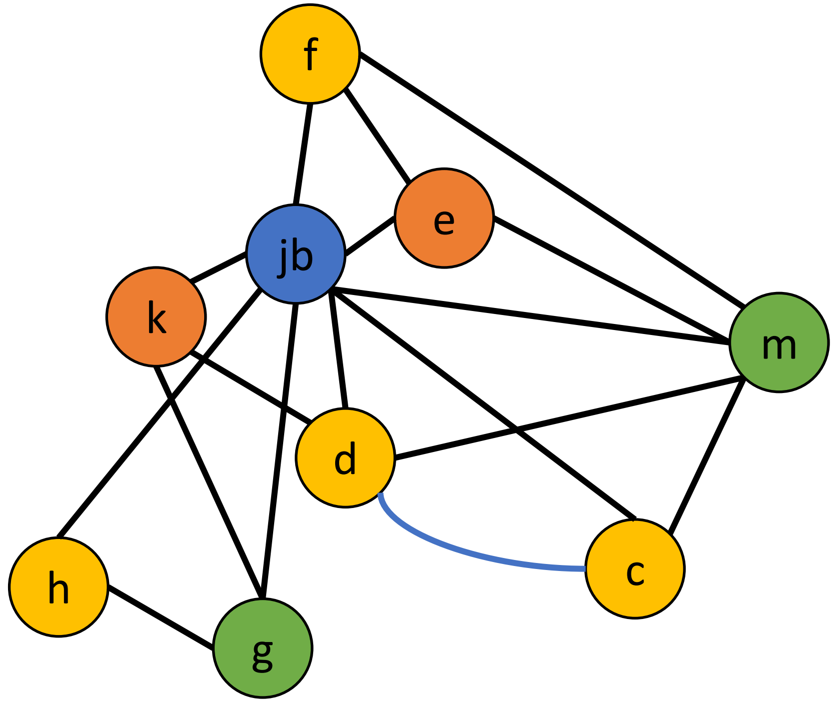
j and b can be coalesced for K=4, not K=3



Conservative coalescing strategies will always keep a graph colorable

- George: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either:
  - *t* already interferes with *b* or
  - *t* has degree < K

j and b can be coalesced for K=4, not K=3

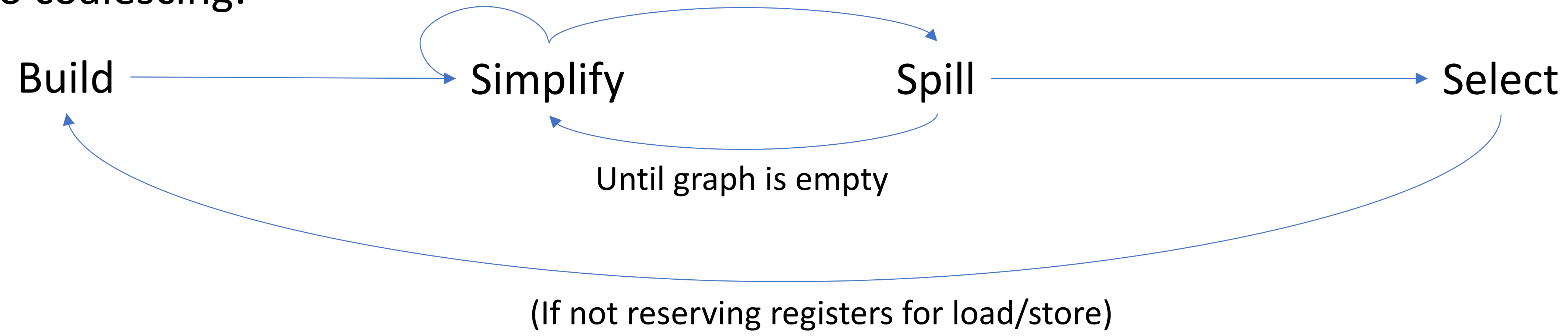


(and the graph is *not* 3-colorable!)

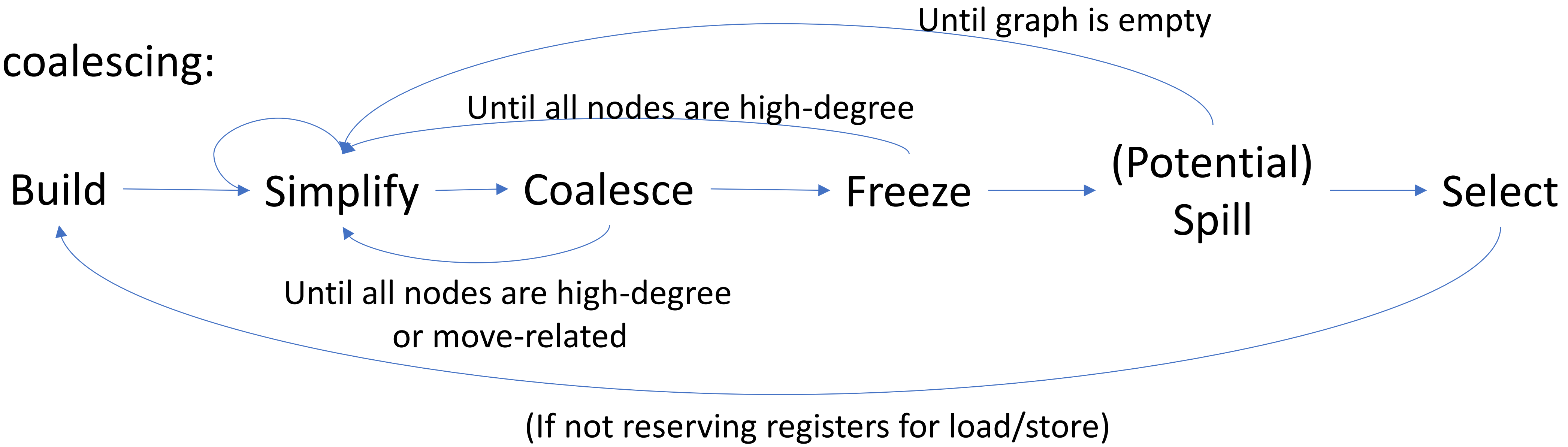
Graph coloring with coalescing

1. **Build** interference graph and classify nodes as move-related or non-move-related
2. **Simplify**, only removing non-move-related nodes of degree < K
3. **Coalesce** move-related nodes using a conservative heuristic
4. **Freeze** move-related nodes (give up trying to coalesce them) if can't simplify or coalesce
5. **Spill** (potentially) a node w/ degree >= K, removing it from the graph and pushing it on the stack
6. **Select** colors for nodes in stack order

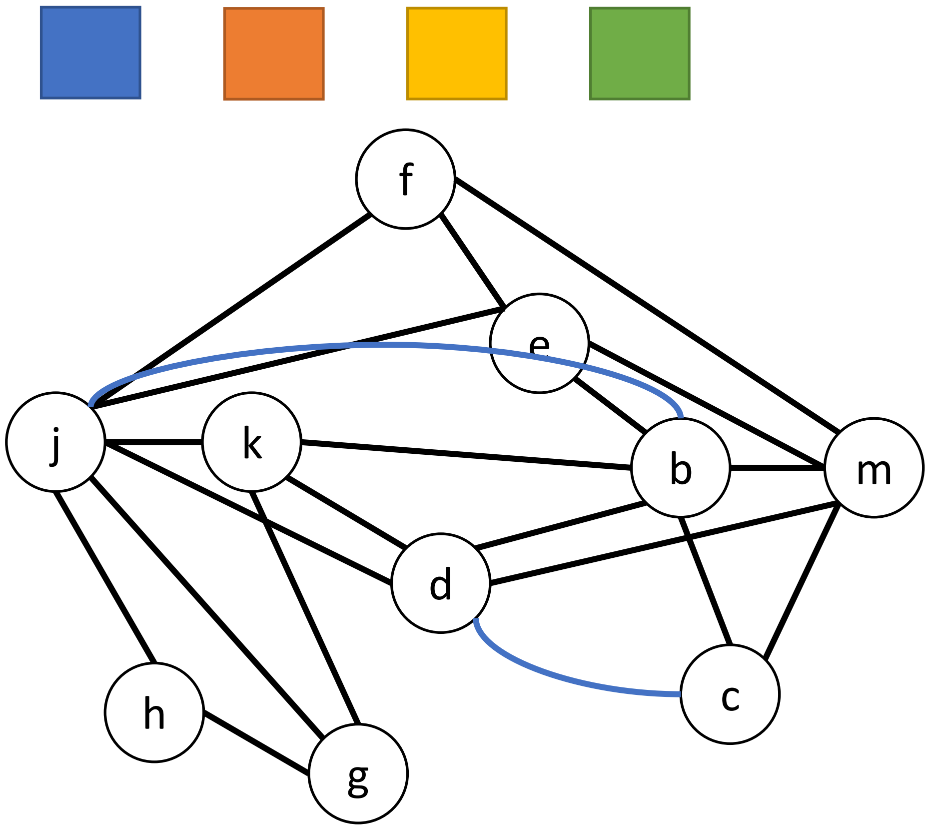
w/o coalescing:



w/ coalescing:



Coalescing Example (Appel)

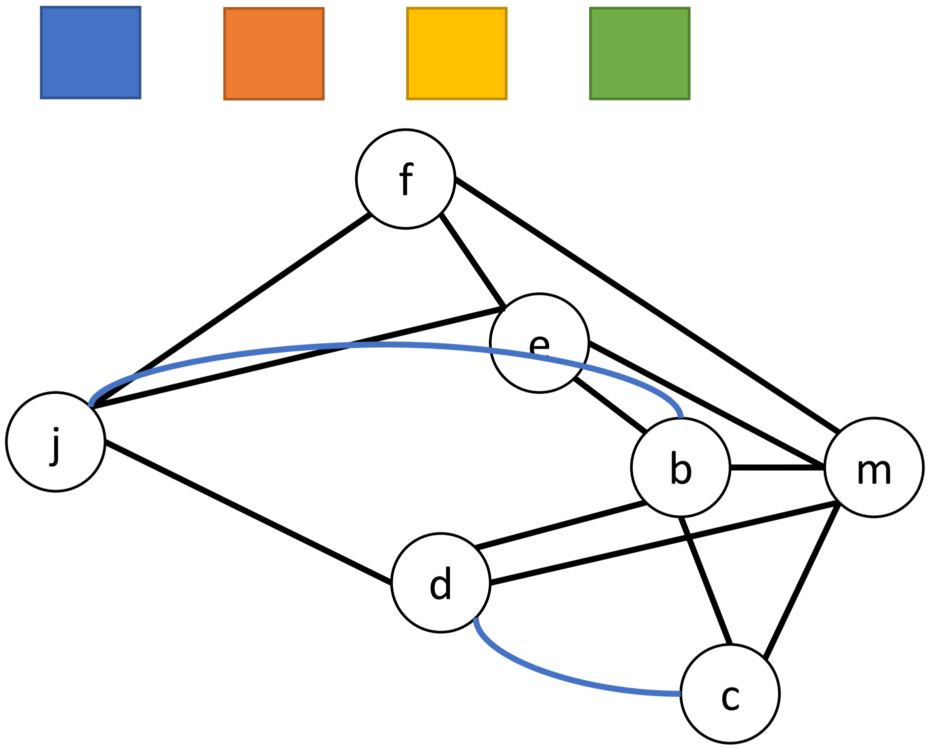


Coalescing Example (Appel)

k

h

g

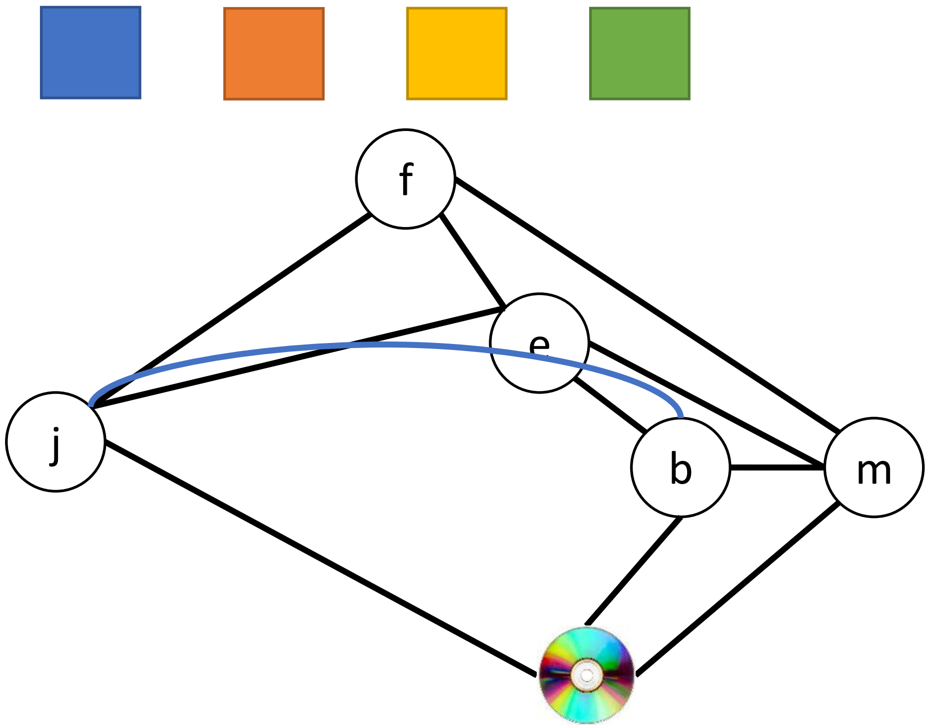


Coalescing Example (Appel)

k

h

g

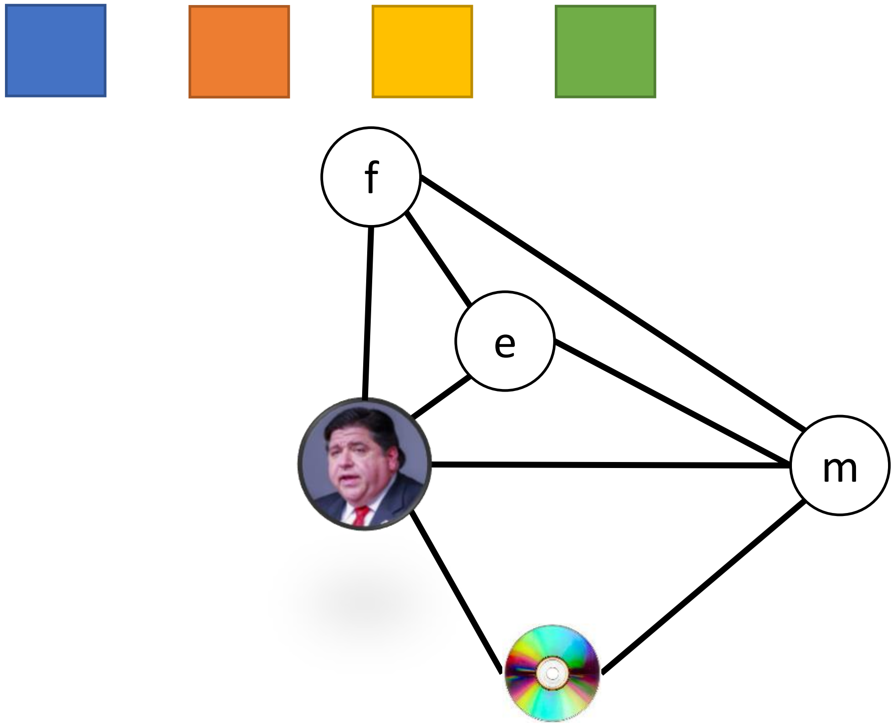


Coalescing Example (Appel)

k

h

g



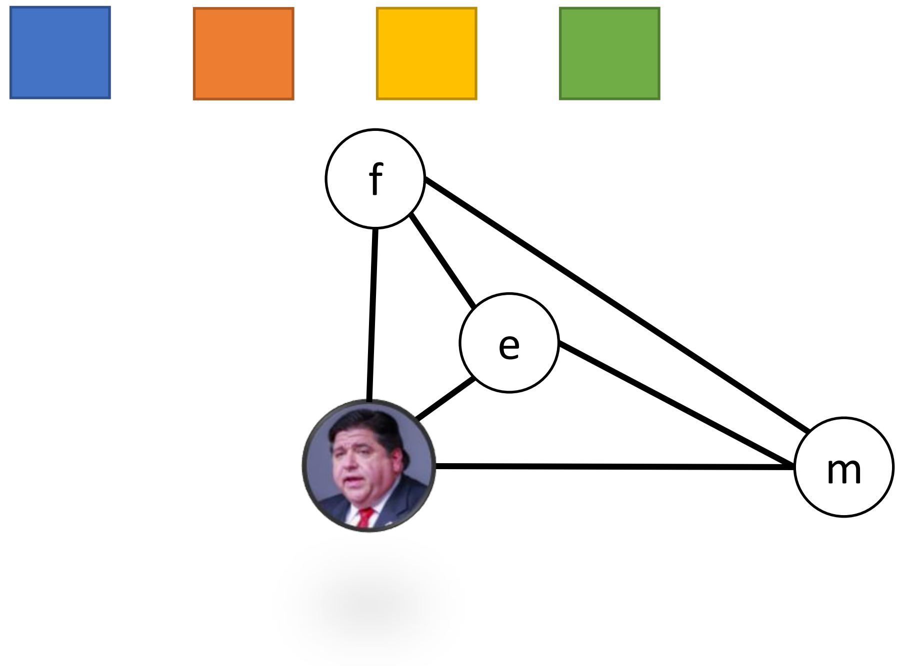
Coalescing Example (Appel)

cd

k

h

g



Coalescing Example (Appel)

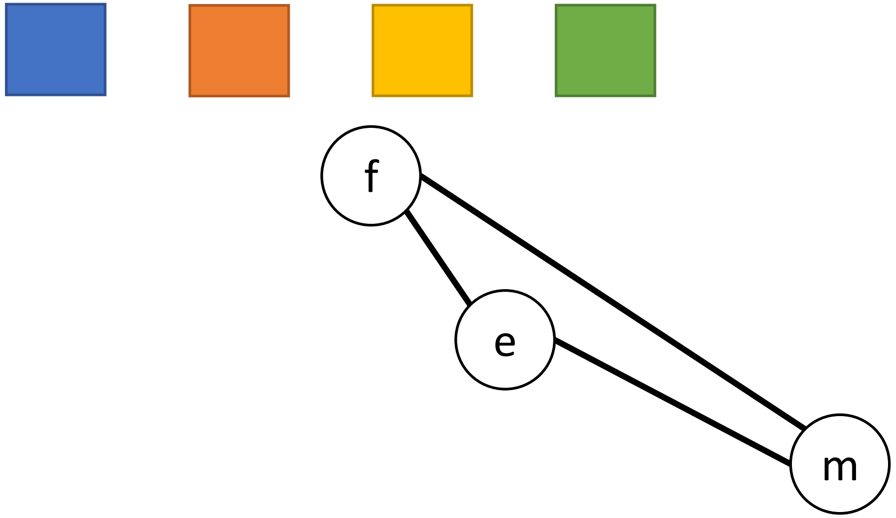
jb

cd

k

h

g



Coalescing Example (Appel)

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

Coalescing Example (Appel)

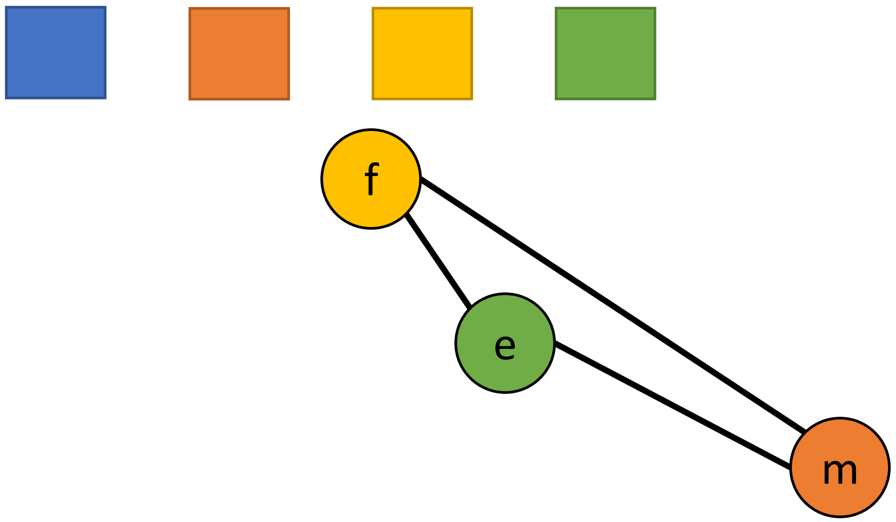
jb

cd

k

h

g

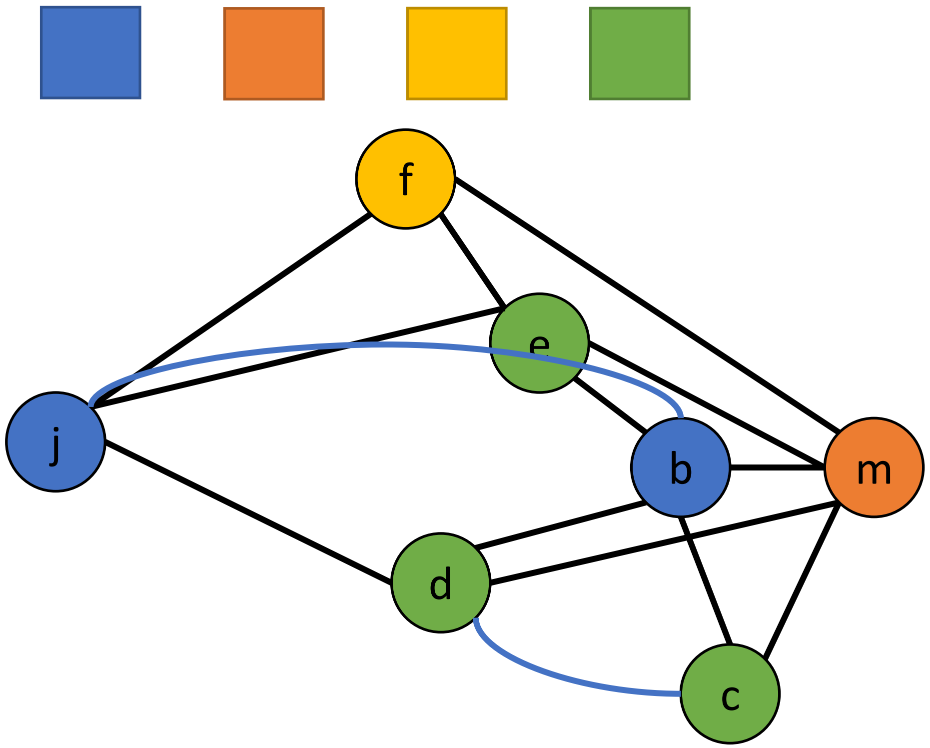


Coalescing Example (Appel)

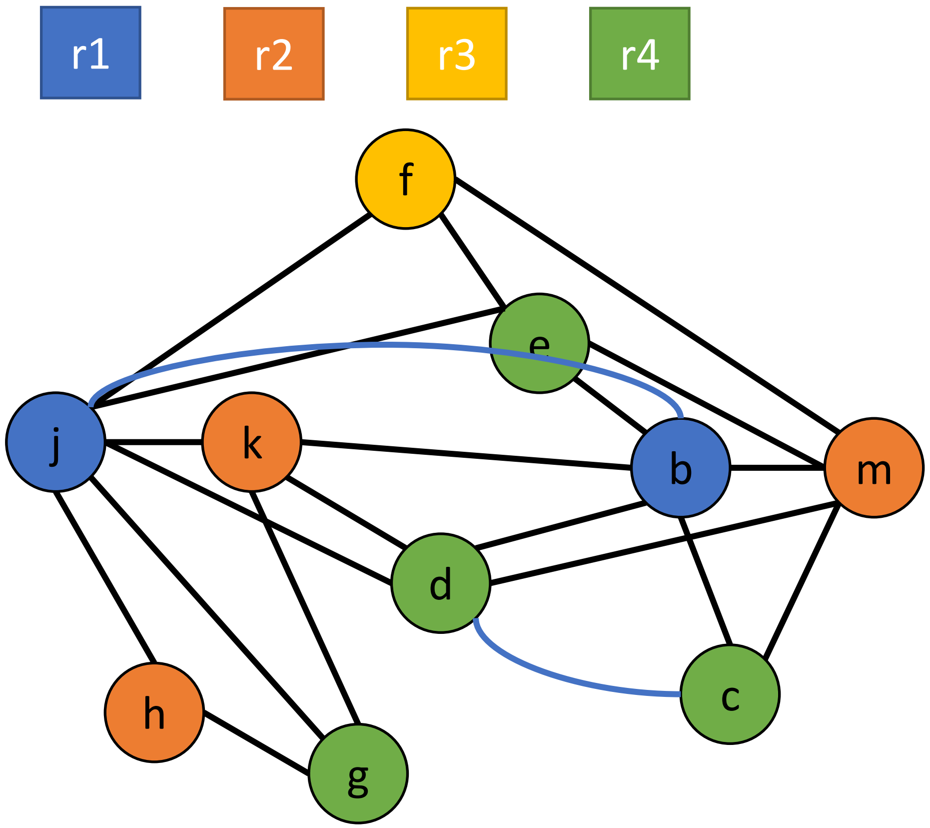
k

h

g

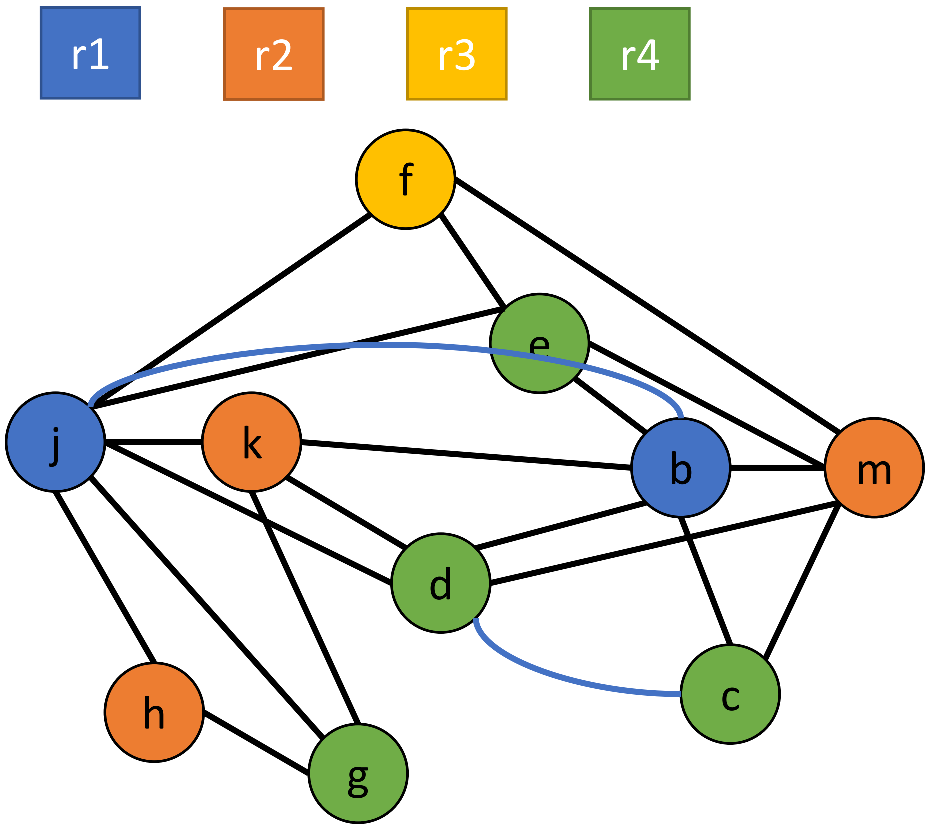


Coalescing Example (Appel)



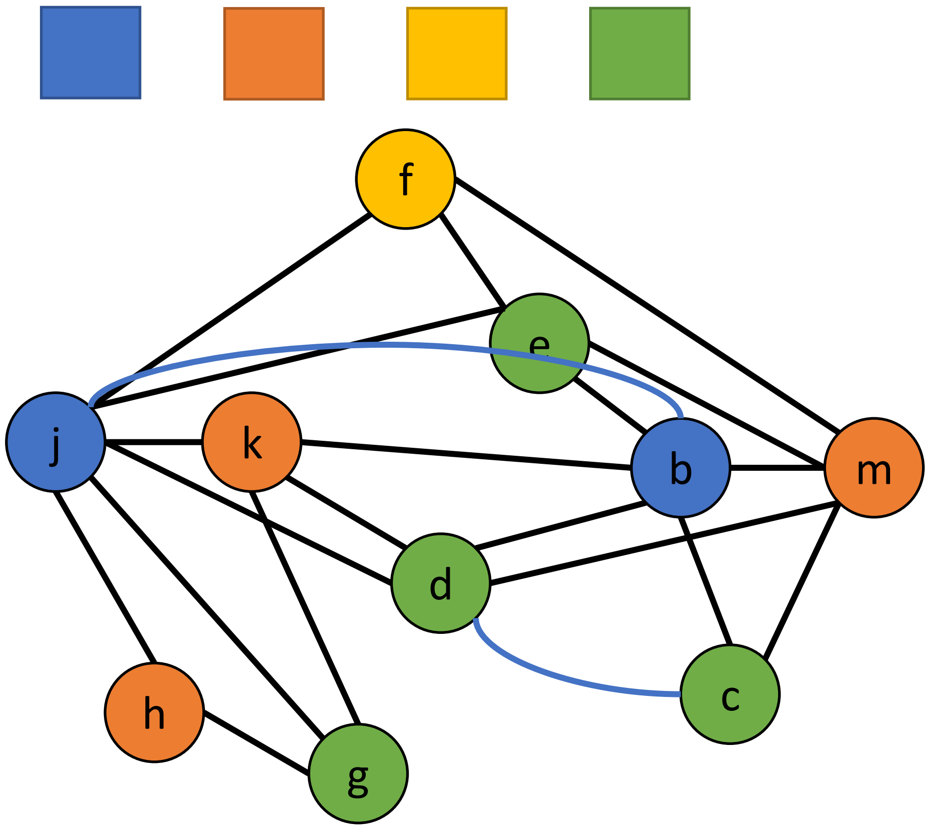
Coalescing Example (Appel)

```
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
```



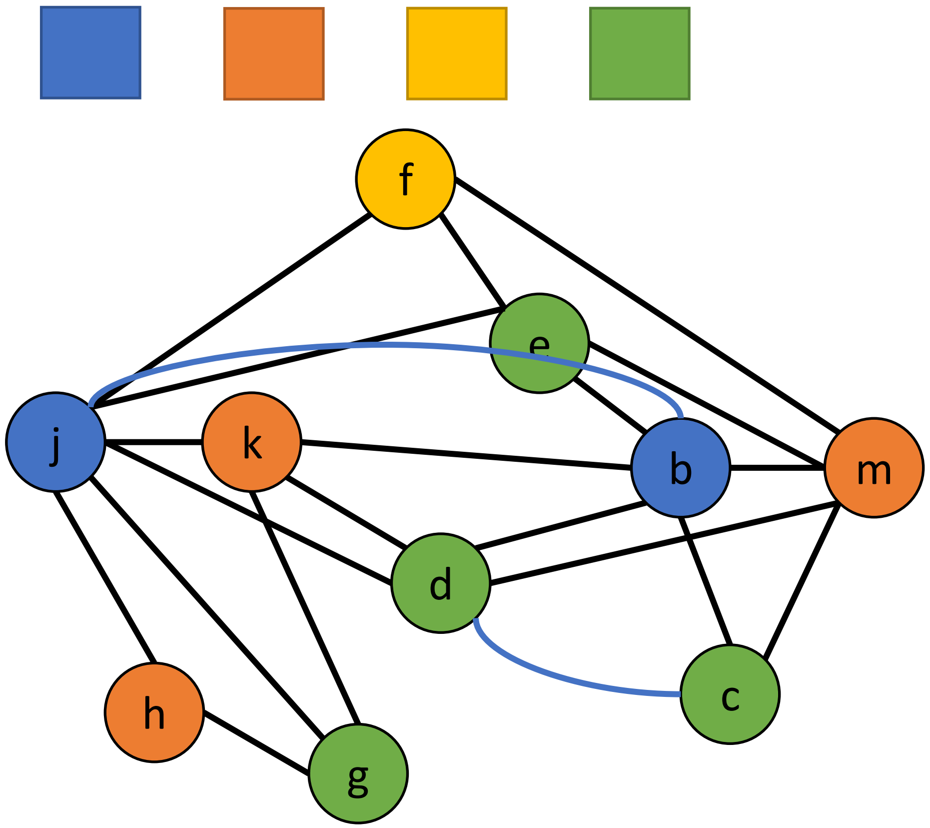
Coalescing Example (Appel)

```
r4 = mem[r1 + 12]
r2 = r2 - 1
r3 = r4 * r2
r4 = mem[r1 + 8]
r2 = mem[r1 + 16]
r1 = mem[r3]
r4 = r4 + 8
r4 = r4
r2 = m + 4
r1 = r1
```



Coalescing Example (Appel)

```
r4 = mem[r1 + 12]
r2 = r2 - 1
r3 = r4 * r2
r4 = mem[r1 + 8]
r2 = mem[r1 + 16]
r1 = mem[r3]
r4 = r4 + 8
r2 = m + 4
```



Another example