HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 21:
Register Allocation

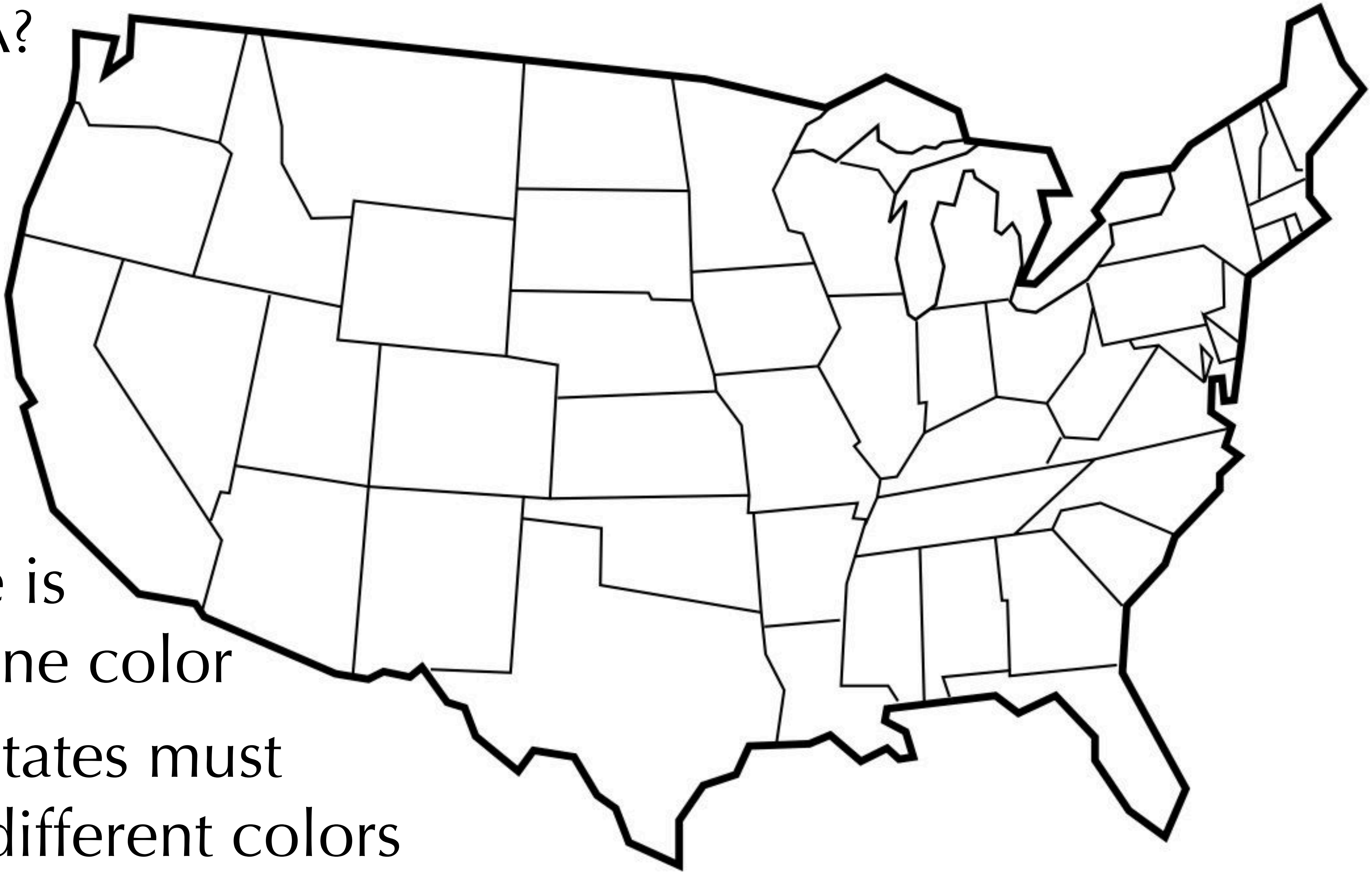
Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

Pre-class Puzzle

- What's the minimum number of colors needed to color a map of the USA?



- Every state is assigned one color
- Adjacent states must be given different colors

https://printable-maps.blogspot.com/2011/12/blank-map-of-united-states.html

Pre-class Puzzle Answer

- 4
- Four-color theorem says ≤4
- Must be at least 4:
  - Suppose we had only 3 colors
  - Pick some colors for CA and OR (Red and Green)
  - NV must be Blue
  - ID must be Red
  - AZ must be Green
  - UT!!!!!!

Announcements

- HW5: Oat v.2 out
  - Due in one week: Tue Nov 19
- HW6 released today
  - Due in 3 weeks: Tue Dec 3

Today

- HW6 overview
- Register allocation
  - Graph coloring by simplification
  - Coalescing
  - Coloring with coalescing
    - Pre-colored nodes to handle callee-save, caller-save, and special purpose registers

HW6

- Analysis and optimization
  - Implement generic iterative dataflow
  - Implement Alias Analysis
  - Implement Dead-Code Elimination
  - Implement Constant Propagation
- Register allocation
- Performance
  - Post a test case
- Optional: participate in leaderboard
  - More optimizations…
- Extra credit available!
  - Sophisticated register allocation, additional optimizations

Yet More General Dataflow Analysis

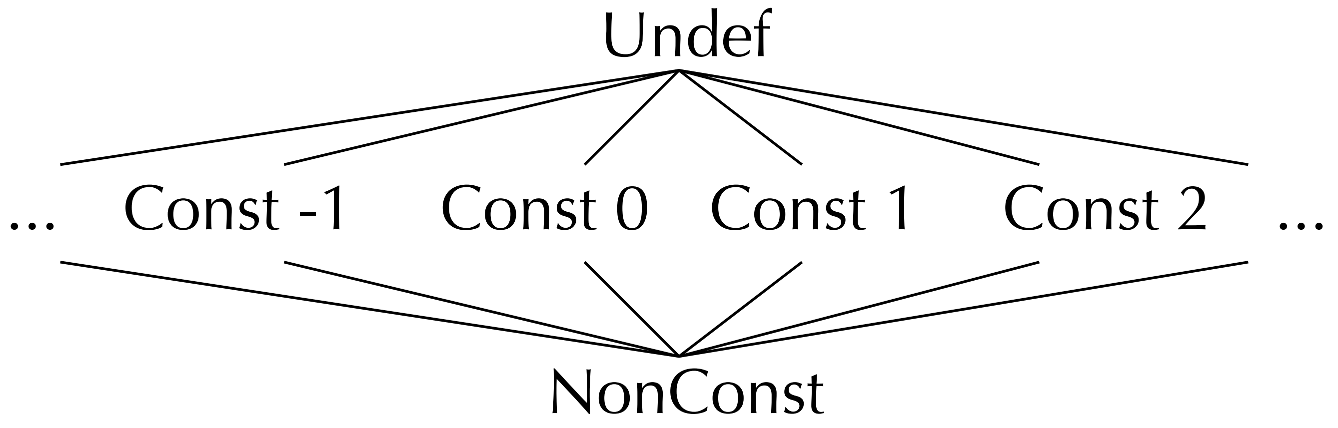
- Gen-kill framework suits many dataflow analyses
  - Forward: out[n] := gen[n] ∪ (in[n] – kill[n])
  - Backward: in[n] := gen[n] ∪ (out[n] – kill[n])
- But some analyses can't be phrased in this way
  - E.g., constant propagation, alias analysis
- Instead, characterize a (forward) dataflow analysis by:
  - Domain of dataflow values $\mathcal{L}$
    - The info we are computing
    - E.g., for reaching definitions, $\mathcal{L}$ is the set of definitions
  - Flow function for instruction n, $F_n : \mathcal{L} \to \mathcal{L}$
    - For gen-kill analyses, $F_n(\ell) = gen[n] \cup (\ell - kill[n])$
  - Combining operator ⊓ : $\mathcal{L} \to \mathcal{L}$
    - "If either ℓ_1 or ℓ_2 holds just before node n, we know at most $\ell_1 \sqcap \ell_2$
    - $in[n] = \sqcap_{n' \in pred[n]} out[n']$
    - E.g., for may analyses ⊓ is ∪ (set union), for must analyses ⊓ is ∩ (set intersection)
  - (Backwards analysis is similar)

Generic Iterative Forward Analysis

```
for all n, in[n] := ⊤, out[n] := ⊤
repeat until no change in ‘in’ and ‘out’
    for all n
        in[n] := $\sqcap_{n' \in succ[n]}$ out[n′]
        out[n] := $F_n$(in[n])
    end
end
```

- ⊤ is the “top element” of $\mathcal{L}$, typically the “maximum” amount of information
    - Having “more” information enables more optimizations
    - “Maximum” information could be inconsistent with the constraints
    - Iteration refines the answer, eliminating inconsistencies

Constant Propagation

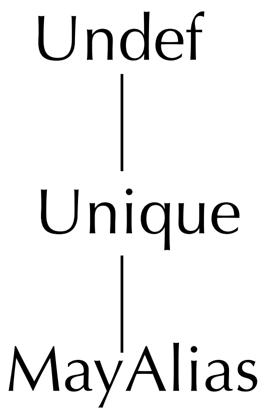

Undef

… Const -1 Const 0 Const 1 Const 2 …

NonConst

- Domain
  - $\mathcal{L}$ = uid → SymConst
  - SymConst = NonConst | Const i | Undef
- Flow function:
  - $F_{\text{uid = ins}}$(m) = m[uid ↦ ⟦ins⟧m]
  - ⟦o1 + o2⟧m =
    - NonConst if ⟦o1⟧m = NonConst or ⟦o2⟧m = NonConst
    - Undef if ⟦o1⟧m = Undef or ⟦o2⟧m = Undef
    - Const k where k = i + j and ⟦o1⟧m = Const i and ⟦o2⟧m = Const j
  - ⟦Null⟧m = NonConst
  - ⟦k⟧m = Const k (i.e., a constant integer operand
  - ⟦%u⟧m = m(u)
  - …
- Combining operator:
  - m1, m2 : uid → SymConst
  - (m1 ⊓ m2)(%u) = m1(%u) ⊓ m2(%u)

Alias Analysis



Undef

Unique

MayAlias

- Domain
  - $\mathcal{L}$ = uid → SymPtr
  - SymPtr = MayAlias | Unique | Undef
- Flow function:
  - $F_{\text{uid = ins}}$ (m) = F(uid = ins, m)
  - F(%s = alloca …, m) = m[%s $\mapsto$ Unique]
  - F(%s = load …, m) = m[%s $\mapsto$ MayAlias]
  - F(%s = store t %t o, m) = m[%t $\mapsto$ MayAlias]
    - (i.e., %t was stored in a location, and so it may no longer be a unique pointer)
  - …
- Combining operator:
  - m1, m2 : uid → SymPtr
  - (m1 $\sqcap$ m2)(%u) = m1(%u) $\sqcap$ m2(%u)

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs
- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e., behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed
- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

- Simple, greedy register-allocation strategy:
- 1. Compute liveness information: `live_out`(`x`)
    - recall: `live_out(x)` is the set of uids that are live immediately after `x`'s definition
- 2. Let `pal` be the set of usable registers
    - usually reserve a couple for spill code [our implementation uses `rax`,`rcx`]
- 3. Maintain "layout" `uid_loc` that maps uids to locations
    - locations include registers and stack slots n, starting at n=0
- 4. Scan through the program. For each instruction that defines a uid `x`
    - `used` = {`r` | `reg r` = `uid_loc`(`y`) s.t. y ∈ `live_out`(`x`)}
    - `available` = `pal` - `used`
    - If available is empty: // no registers available, spill
      `uid_loc`(`x`) := slot n ; n = n + 1
    - Otherwise, pick `r` in `available`: // choose an available register
      `uid_loc`(`x`) := `reg r`

For HW6

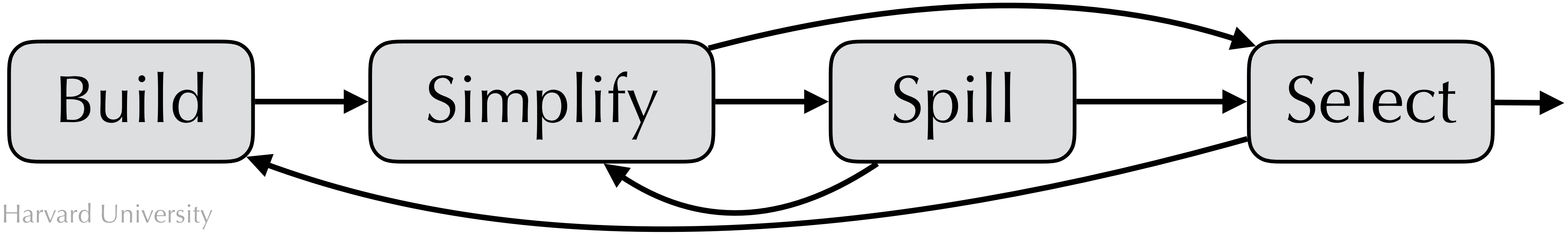
- HW 6 implements two naive register allocation strategies:
  - `no_reg_layout`: spill all registers
  - `greedy_layout`: assign registers greedily using linear scan
- Your job: do "better" than these.
- Quality Metric:
  - registers other than `rbp` count positively
  - `rbp` counts negatively (it is used for spilling)
  - shorter code is better
- Linear scan register allocation should suffice
  - But… can we do better?

Register Allocation

- Register allocation is in generally an NP-complete problem
  - Can we allocate all these *n* temporaries to *k* registers?
- But we have a heuristic that is linear in practice!
  - Based on **graph coloring**
    - Given a graph, can we assign one of *k* colors to each node such that connected nodes have different colors?
  - Here, nodes are temp variables, an edge between `t1` and `t2` means that `t1` and `t2` are live at the same time. Colors are registers.
- But graph coloring is also NP-complete! How does that work?

Coloring by Simplification

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

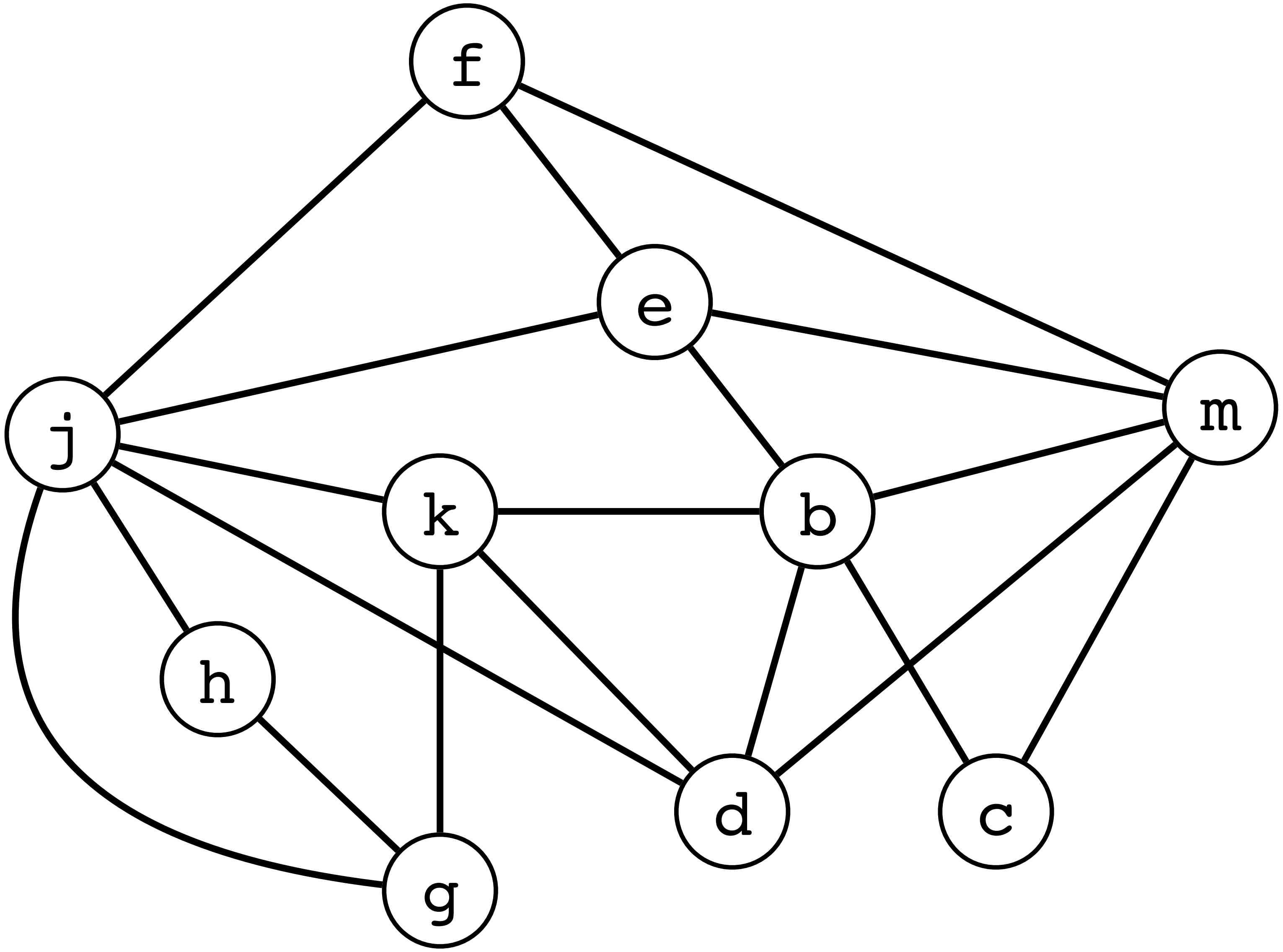


Example

From Appel

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

Interference graph

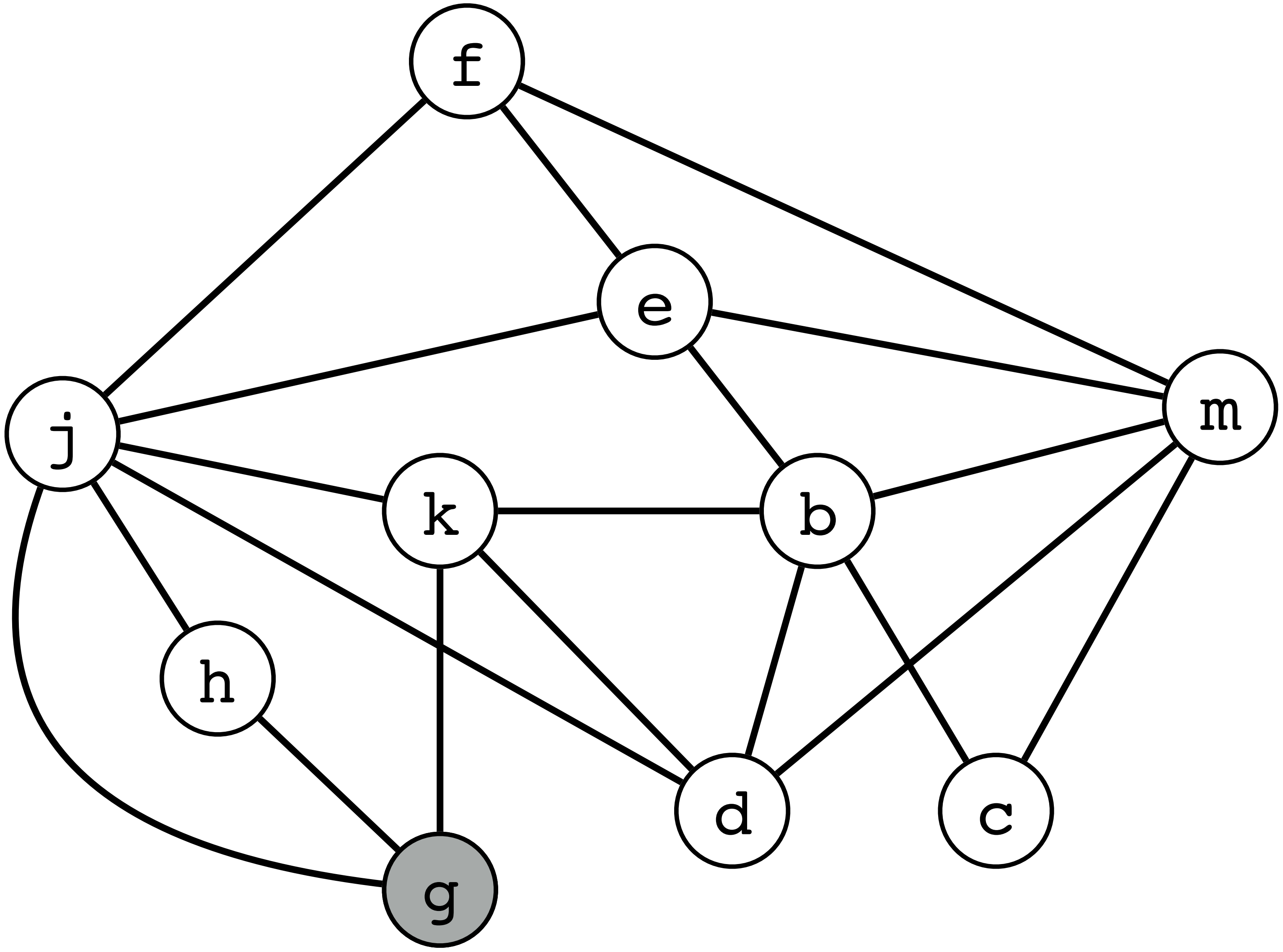


Simplification (4 registers)

Choose any node with degree <4

Stack:

g



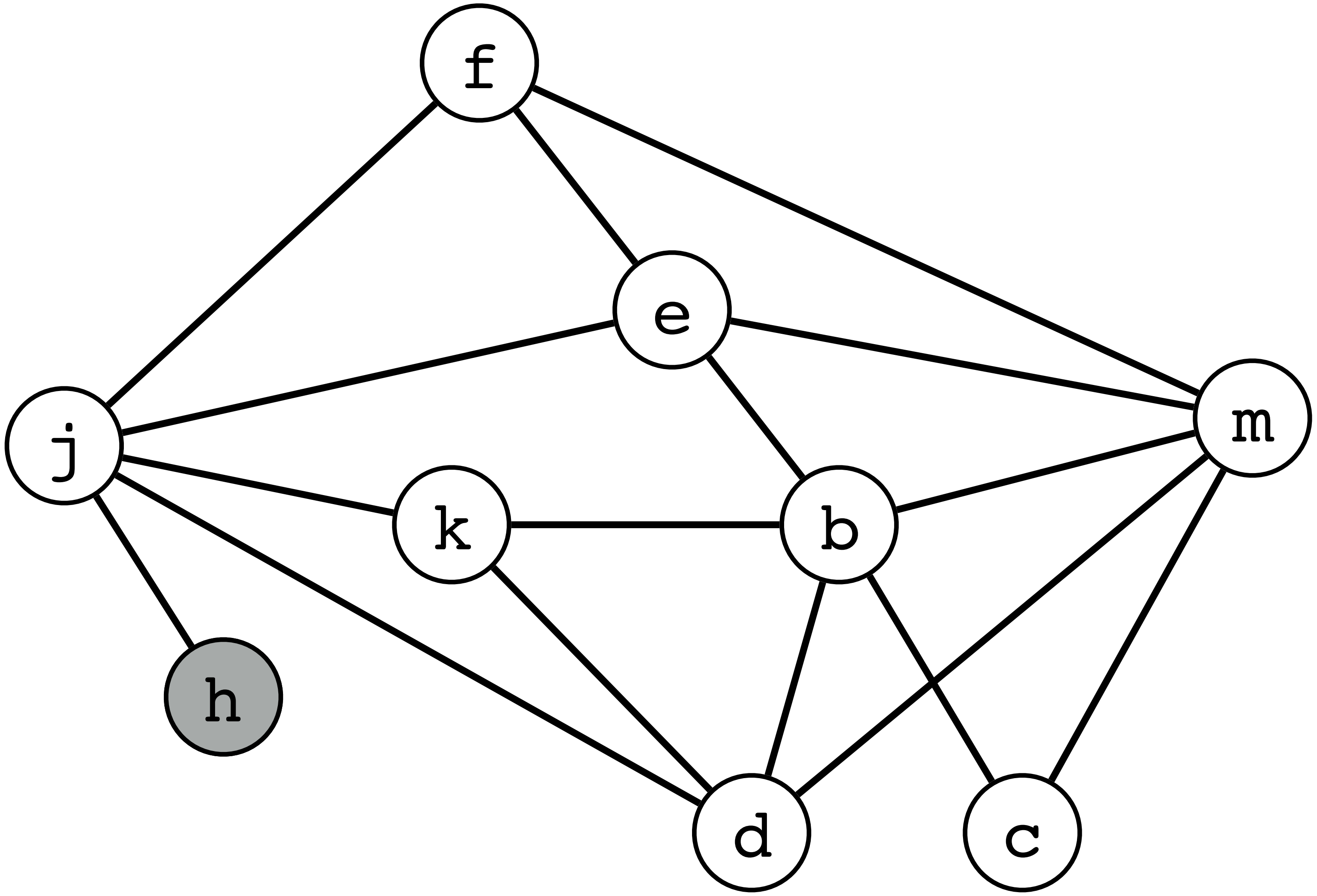
Simplification (4 registers)

Choose any node with degree <4

Stack:

g

h



Simplification (4 registers)

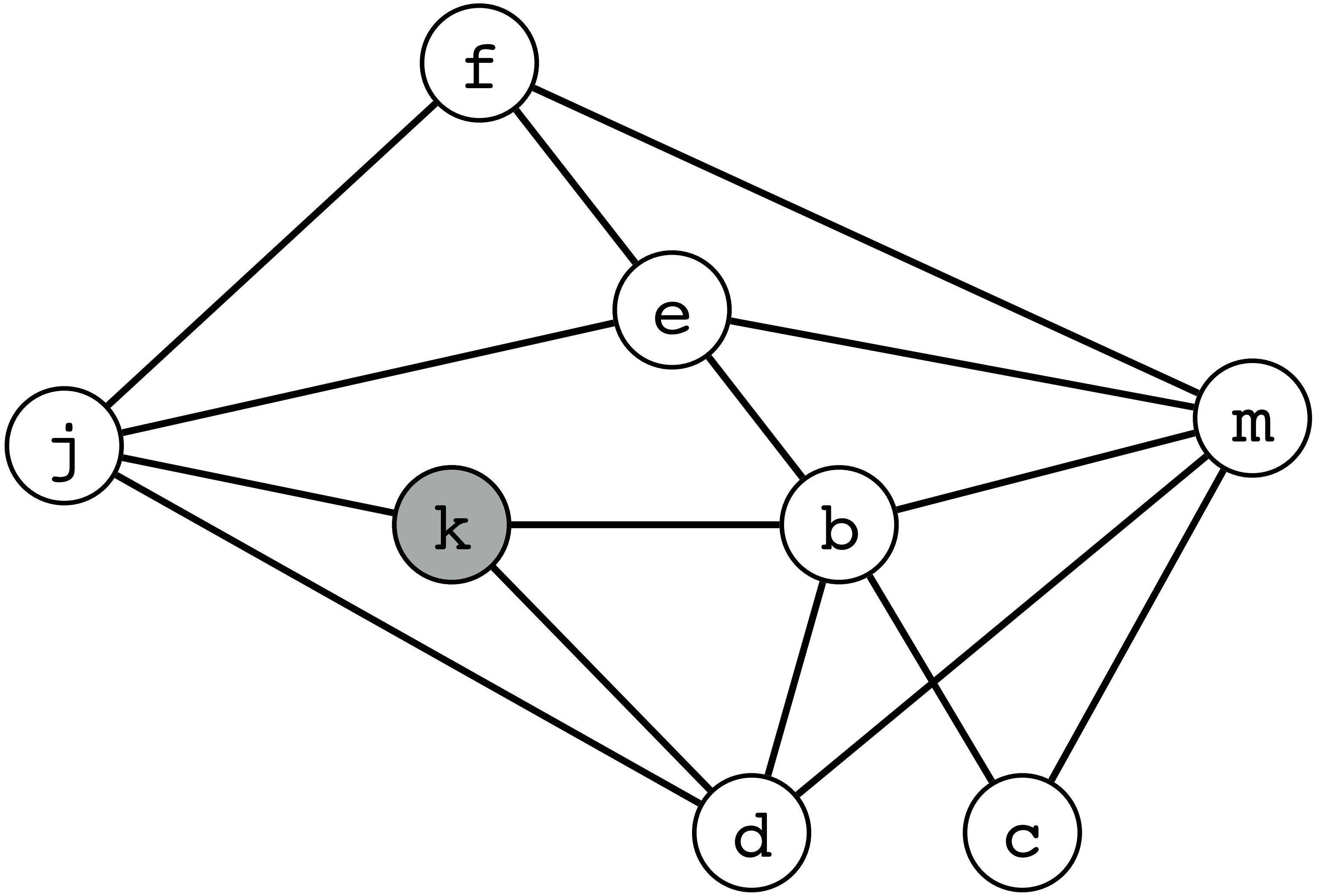
Choose any node with degree <4

Stack:

`g`

`h`

`k`



Simplification (4 registers)

Choose any node with degree <4

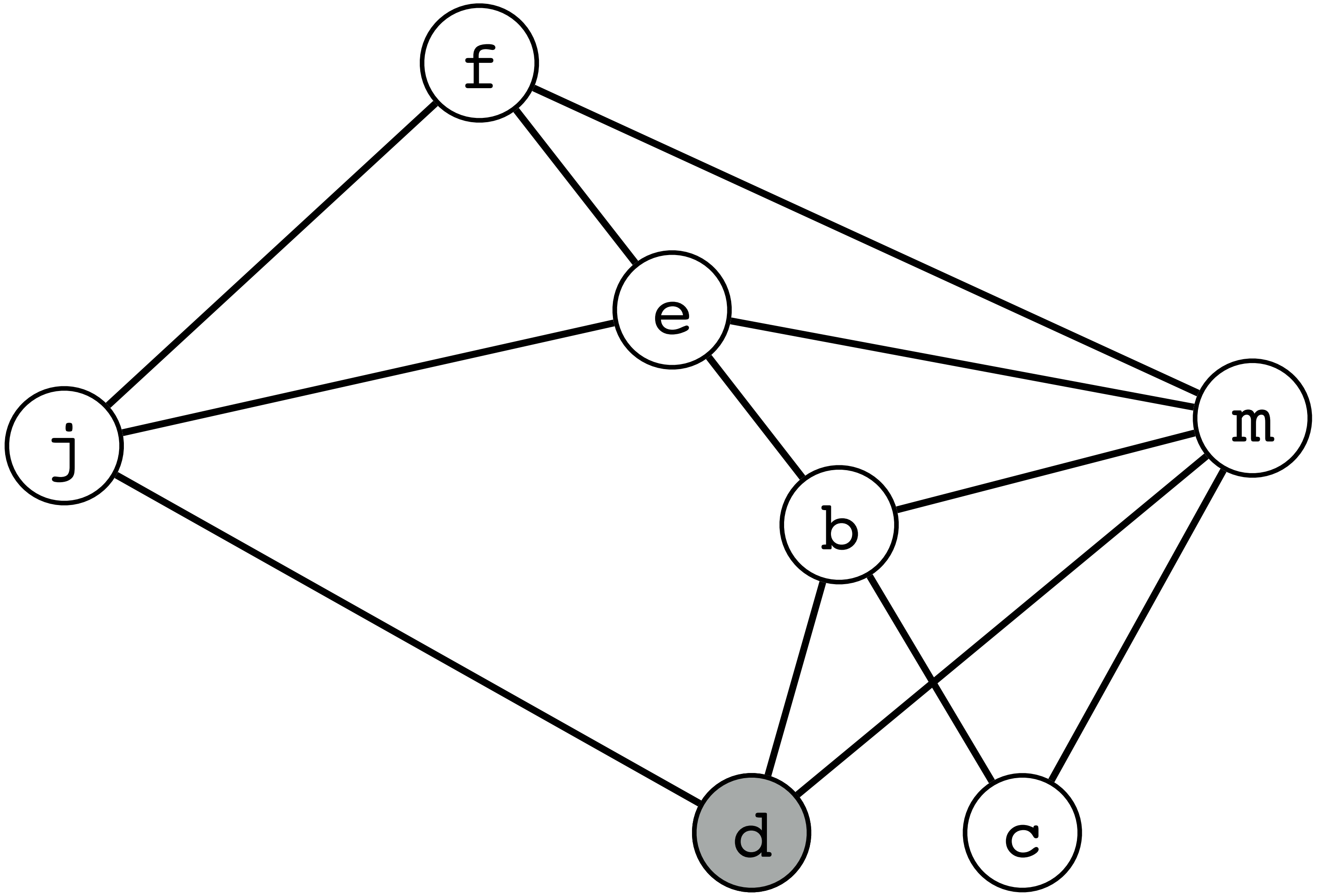
Stack:

`g`

`h`

`k`

`d`



Simplification (4 registers)

Choose any node with degree <4

Stack:

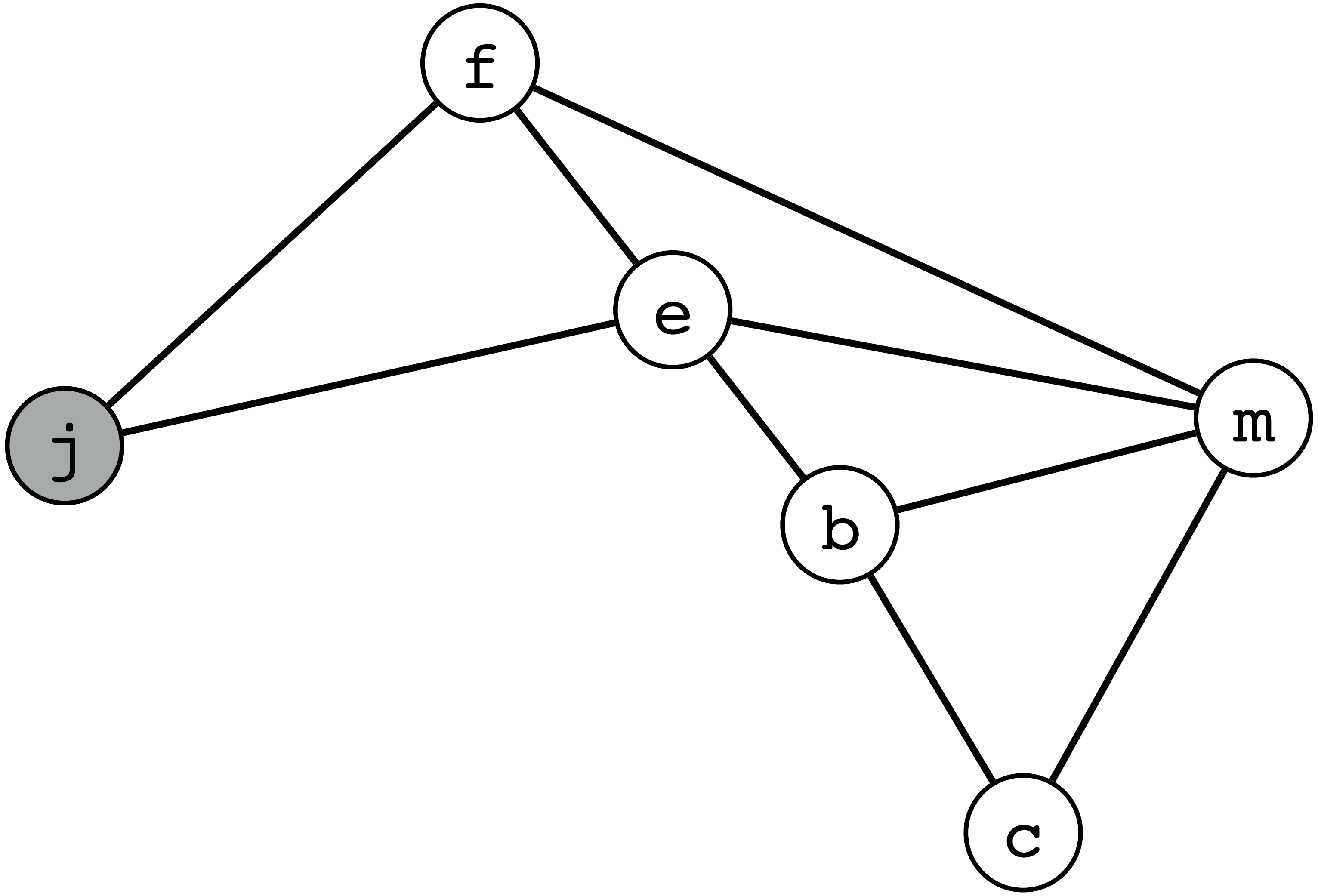
g

h

k

d

j

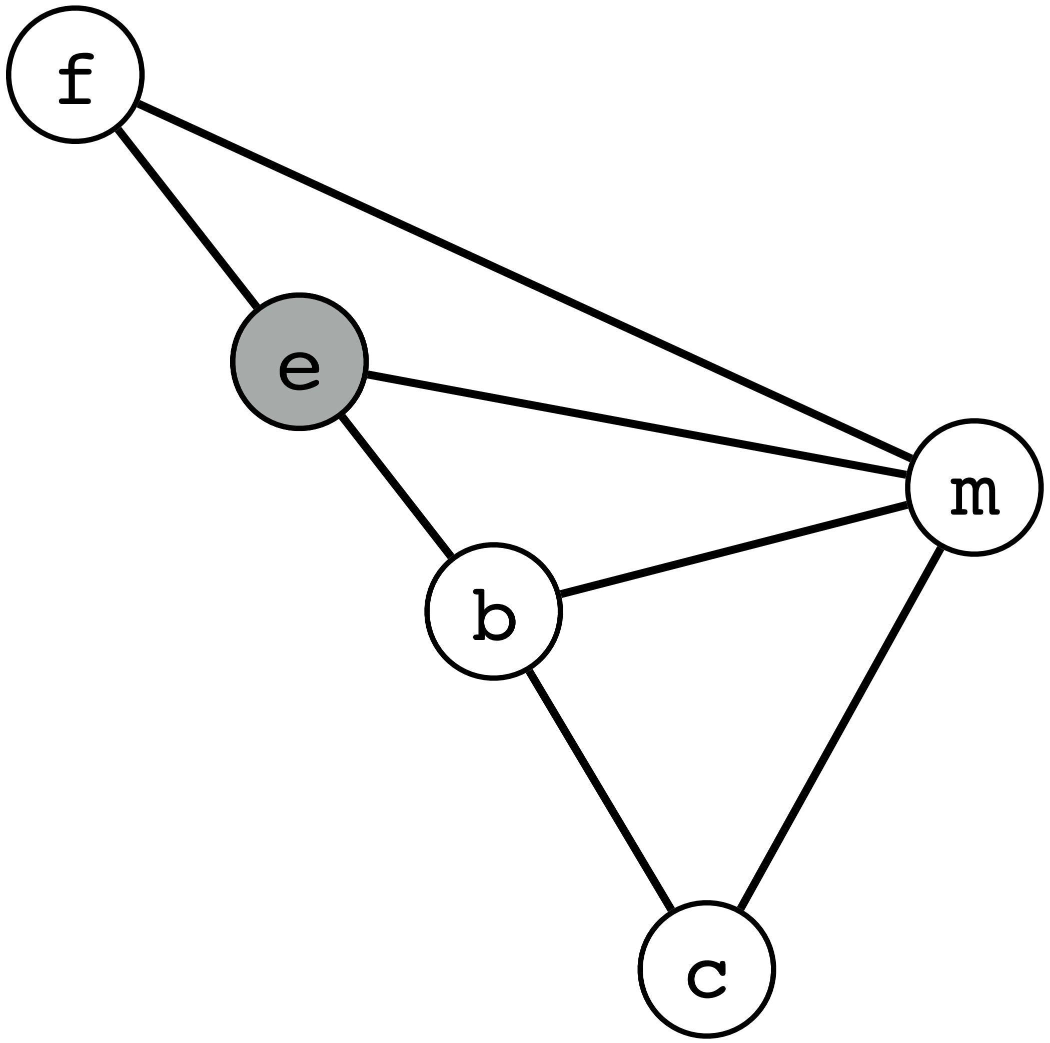


Simplification (4 registers)

Choose any node with degree <4

Stack:

`g`
`h`
`k`
`d`
`j`
`e`

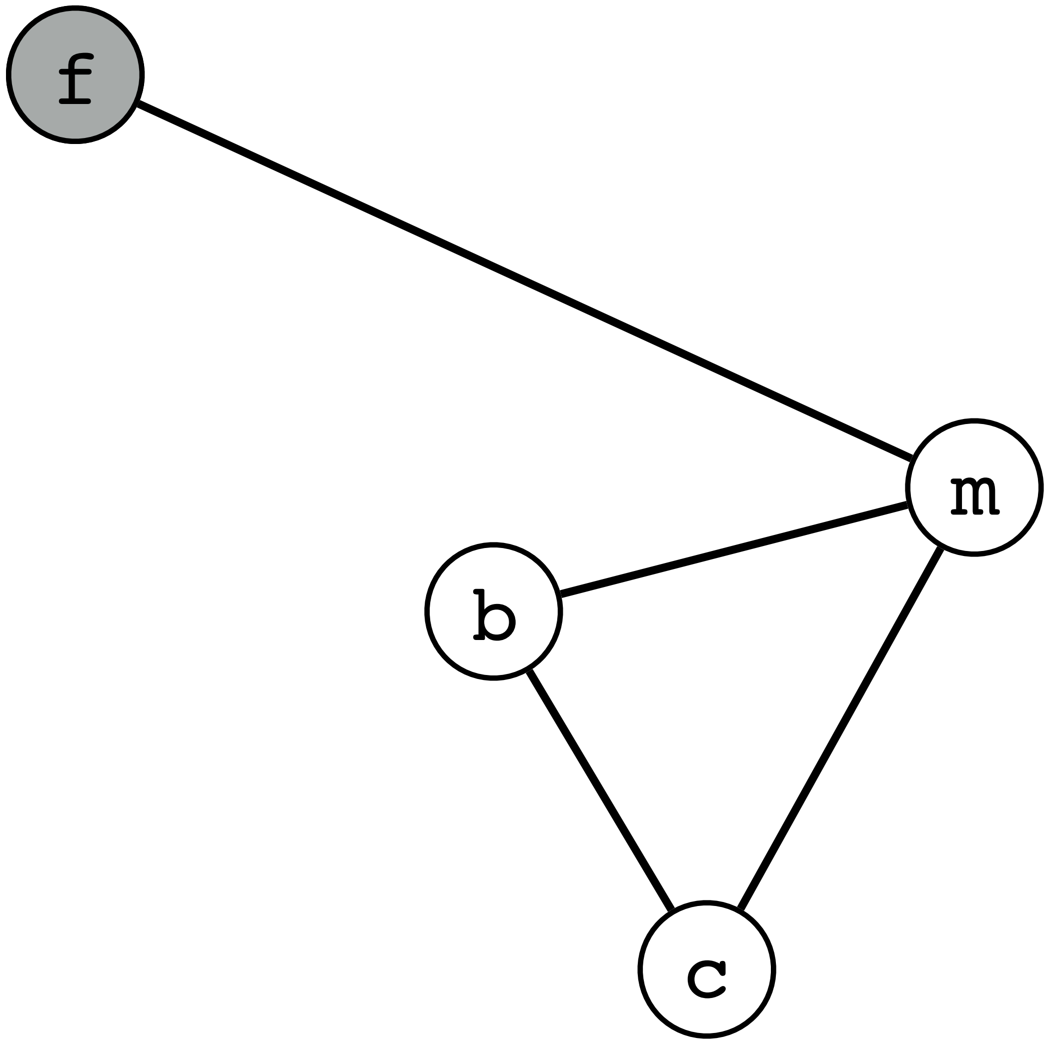


Simplification (4 registers)

Choose any node with degree <4

Stack:

`g`
`h`
`k`
`d`
`j`
`e`
`f`

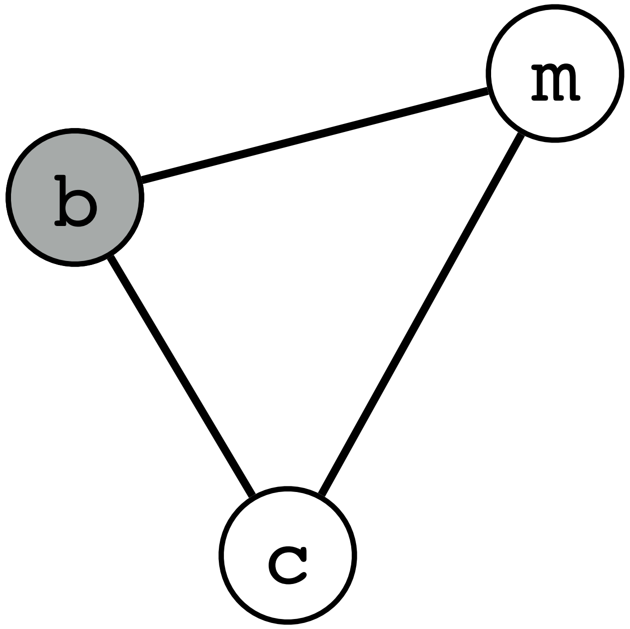


Simplification (4 registers)

Choose any node with degree <4

Stack:

`g`
`h`
`k`
`d`
`j`
`e`
`f`
`b`

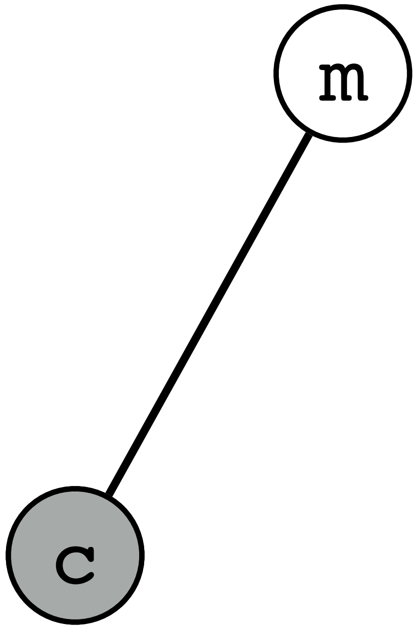


Simplification (4 registers)

Choose any node with degree <4

Stack:

g
h
k
d
j
e
f
b
c

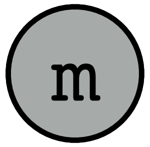


Simplification (4 registers)

Choose any node with degree <4

Stack:

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



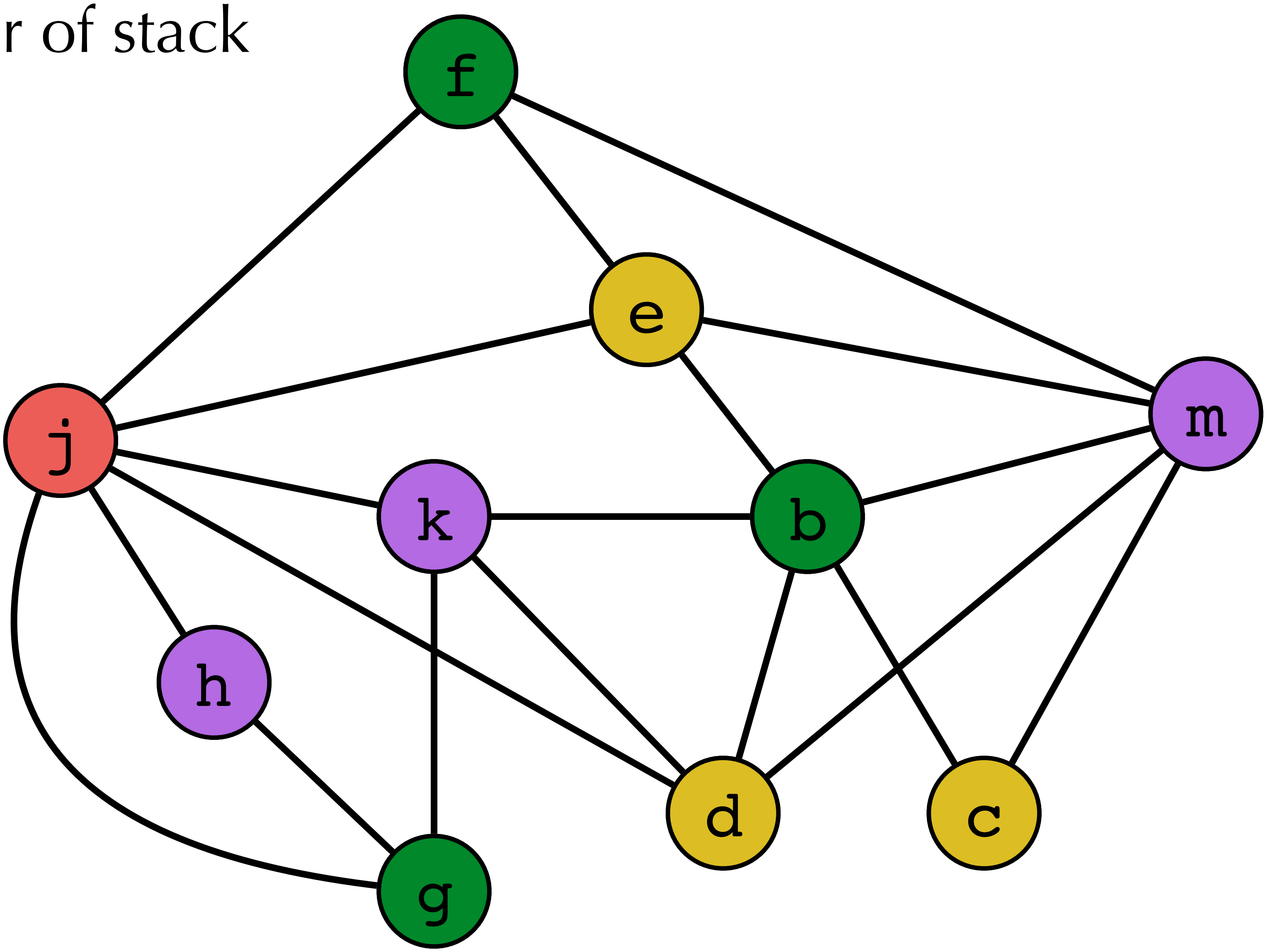
Select (4 registers)

Graph is now empty!

Color nodes in order of stack

Stack:

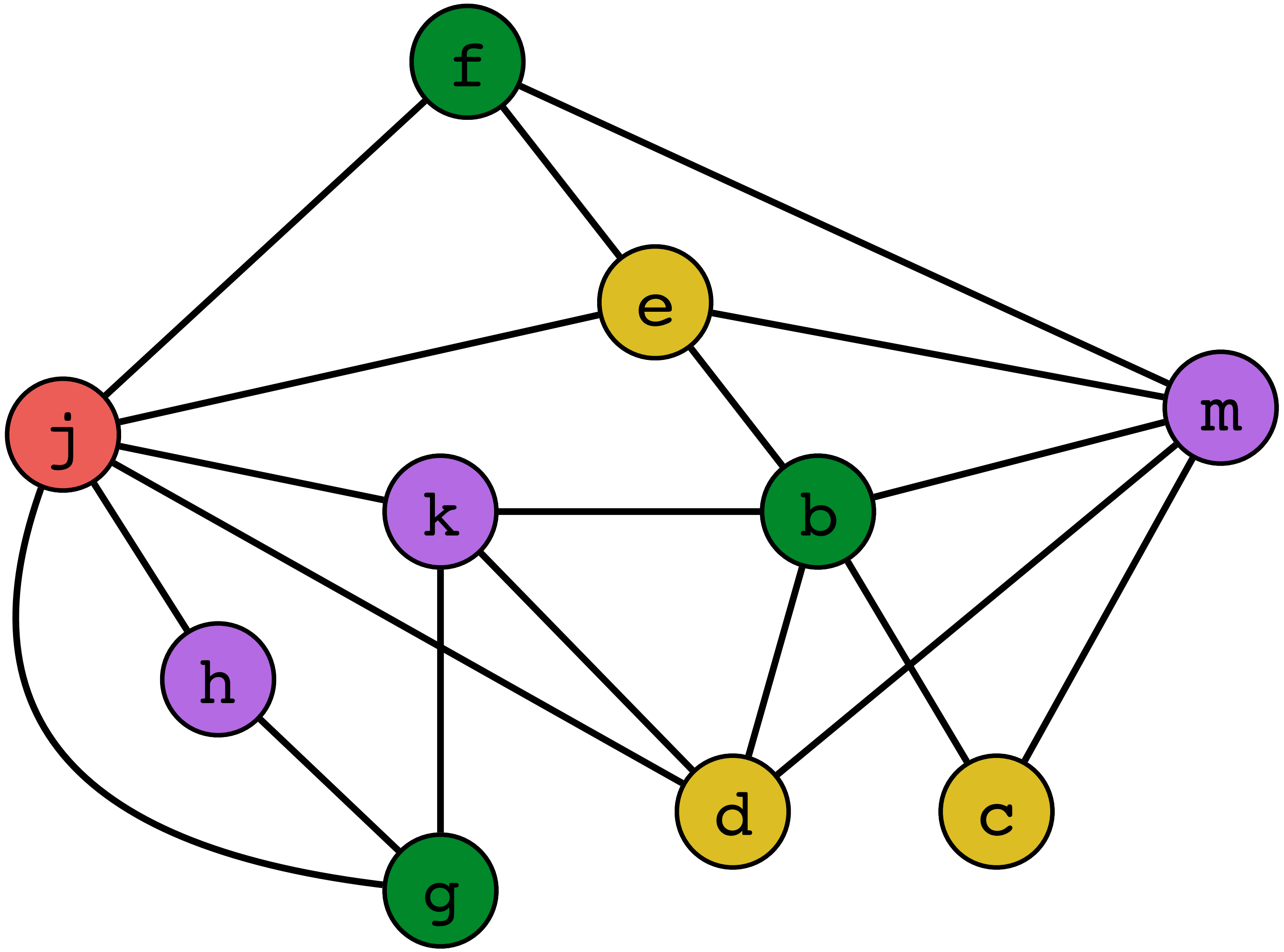
- g
- h
- k
- d
- j
- e
- f
- b
- c
- m



🟣=t1 🟢=t2 🟡=t3 🔴=t4

Select (4 registers)

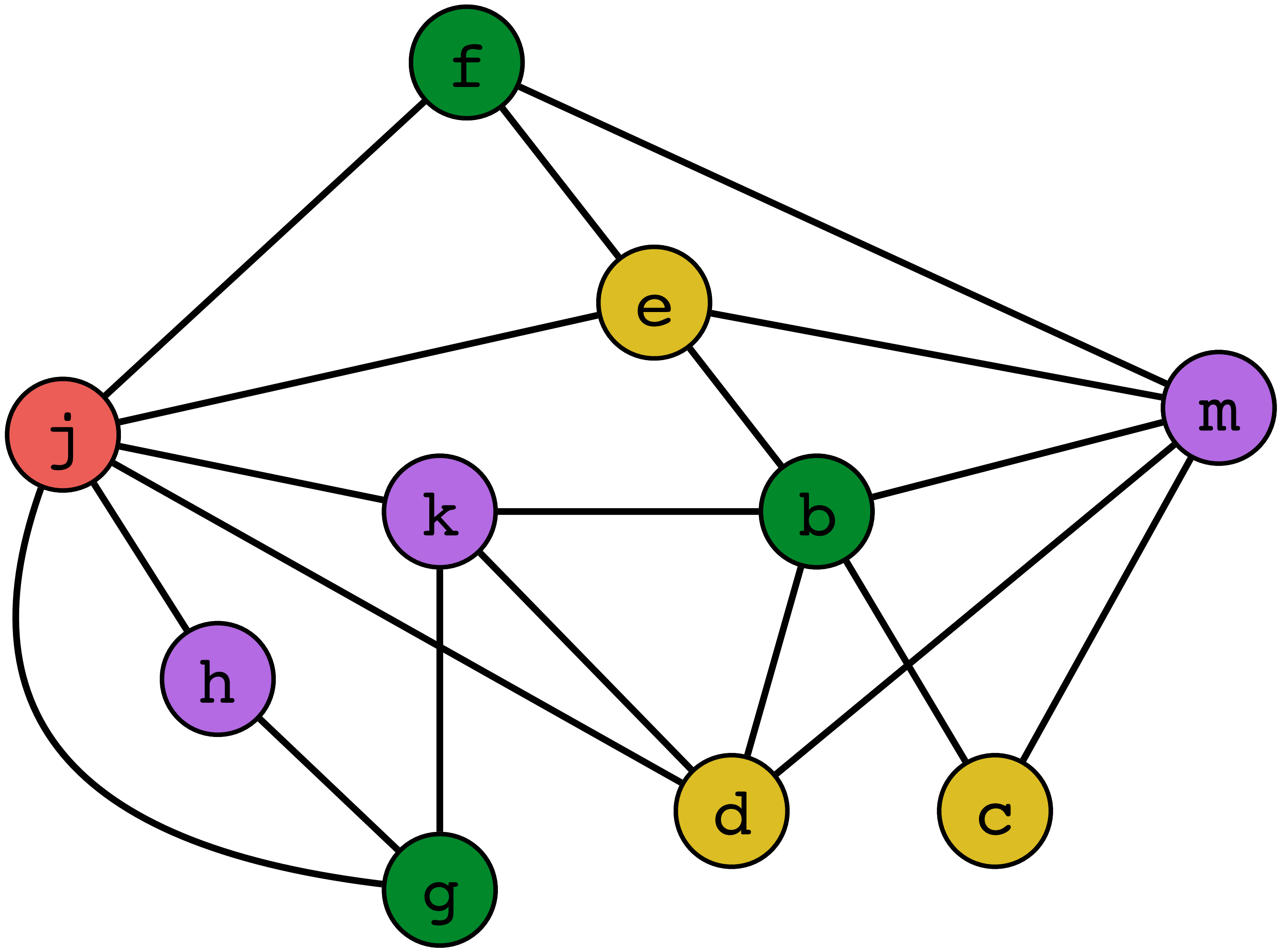
```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
```



🟣=t1 🟢=t2 🟡=t3 🔴=t4

Select (4 registers)

```
$t2 := *(t4+12)
$t1 := $t1 - 1
$t2 := $t2 * $t1
$t3 := *($t4+8)
$t1 := *($t4+16)
$t2 := *($t2+0)
$t3 := $t3 + 8
$t3 := $t3
$t1 := $t1 + 4
$t4 := $t2
```



Some moves might subsequently be simplified...

🟣=t1 🟢=t2 🟡=t3 🔴=t4

Spilling

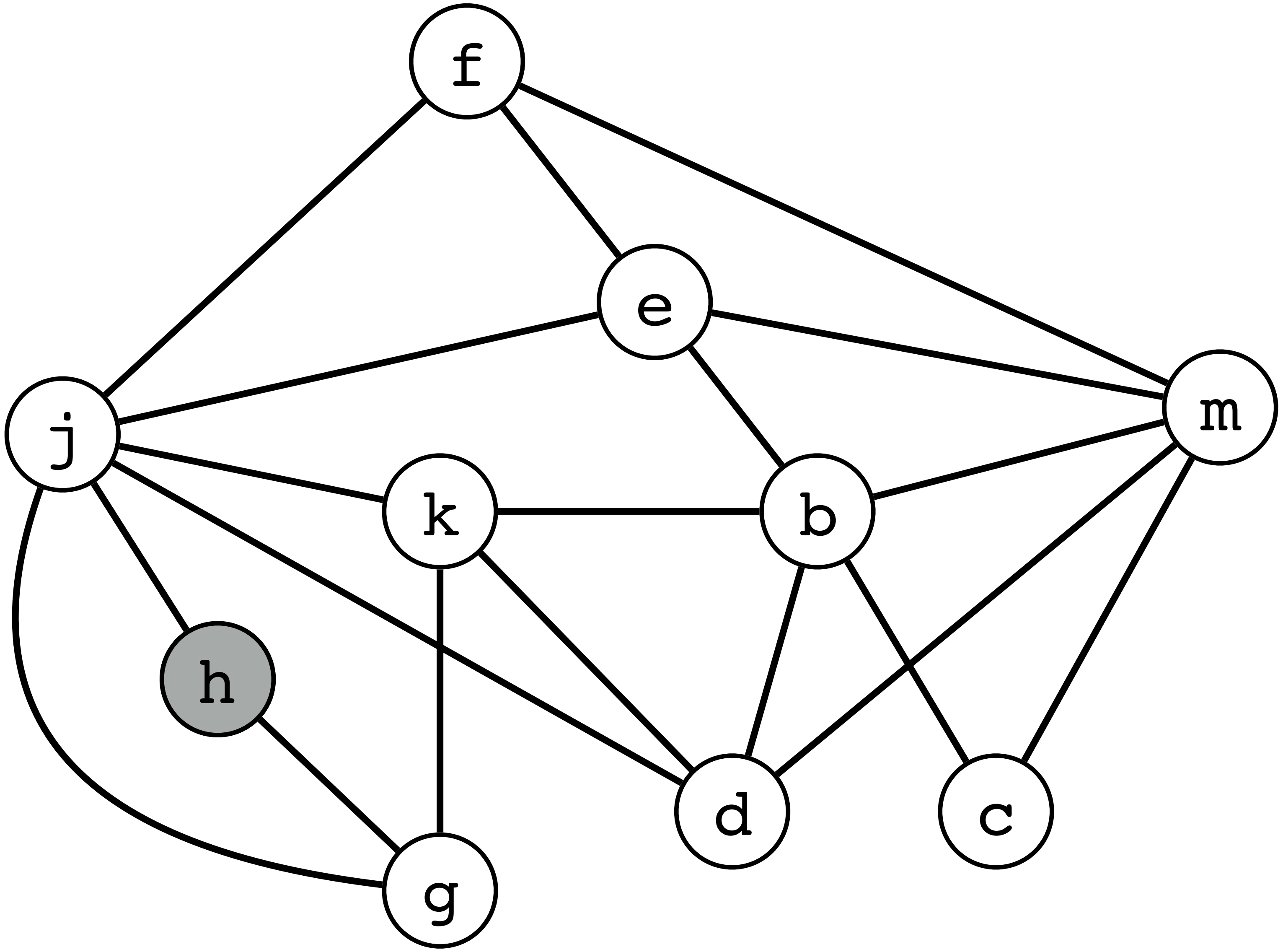
- This example worked out nicely!
- Always had nodes with degree $<k$
- Let's try again, but now with only 3 registers...

Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`

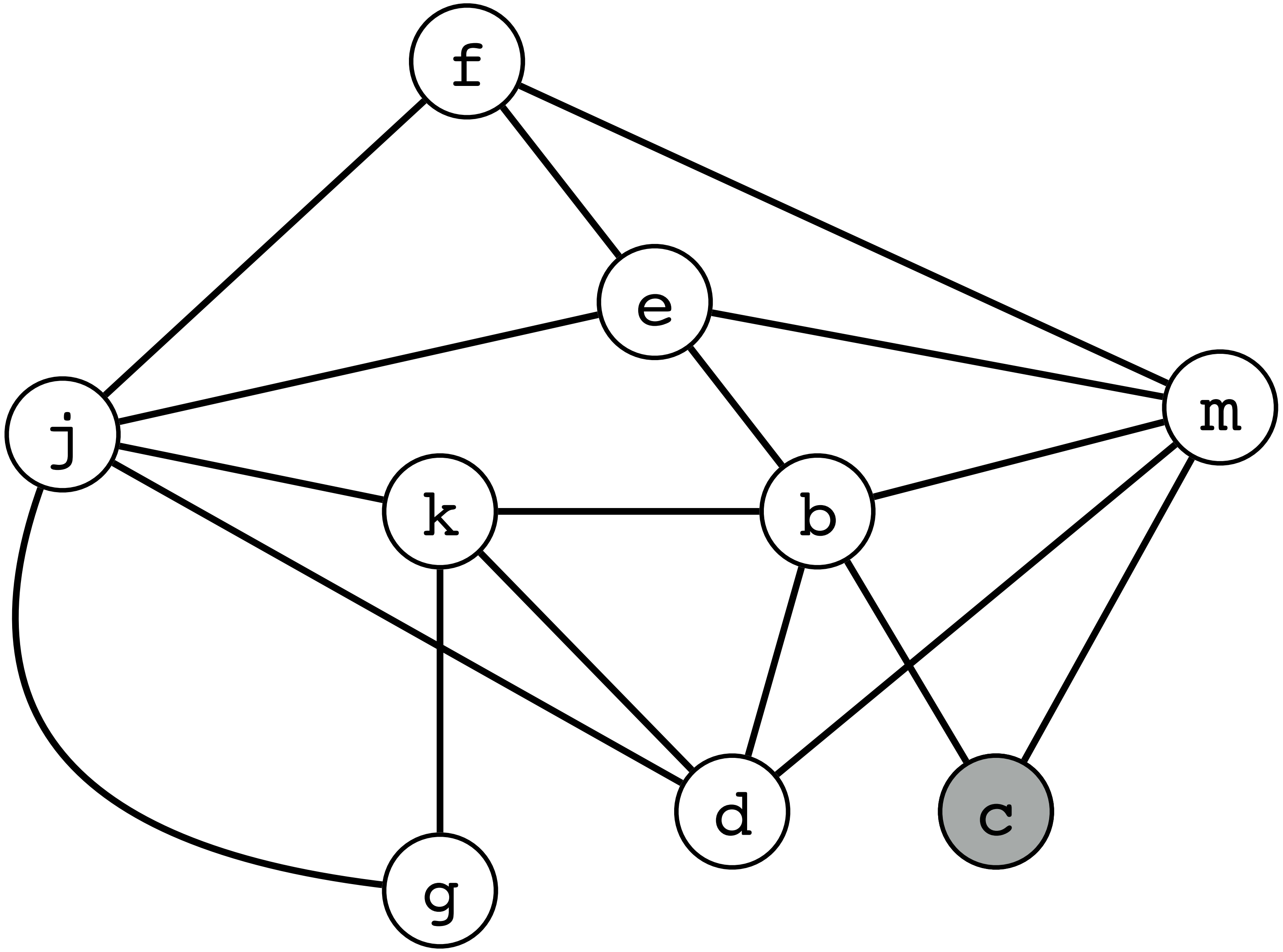


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`

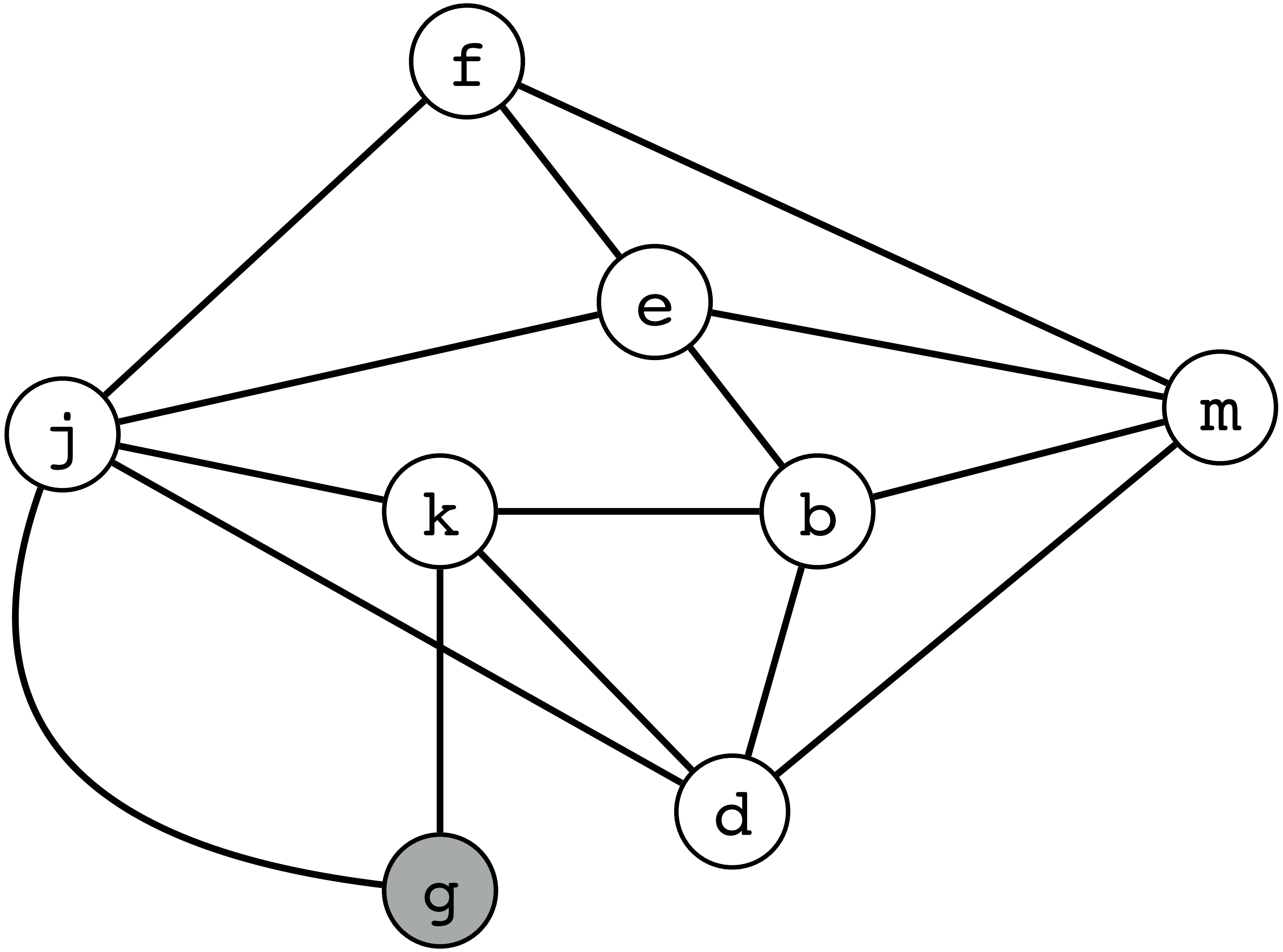


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g

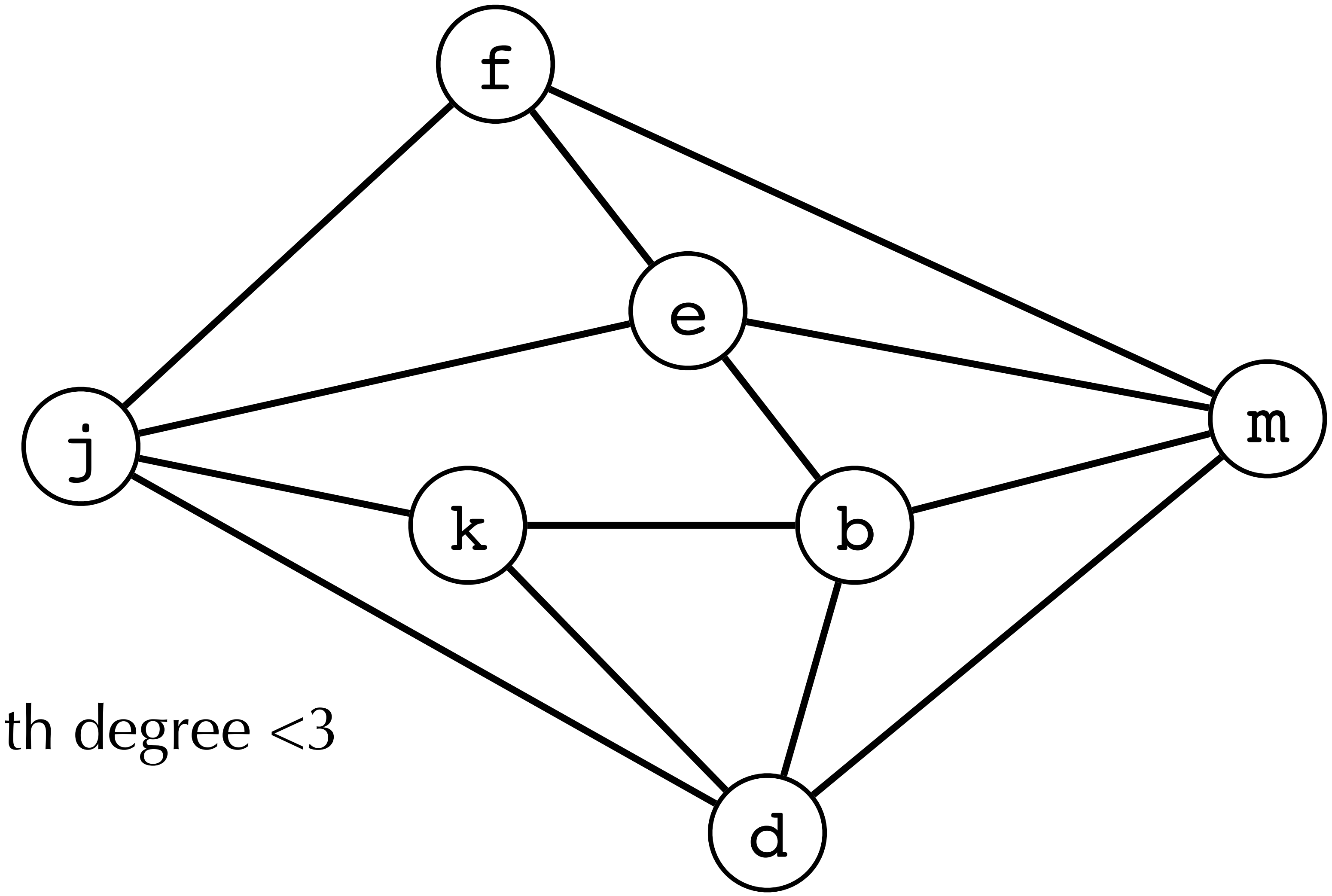


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`
`g`

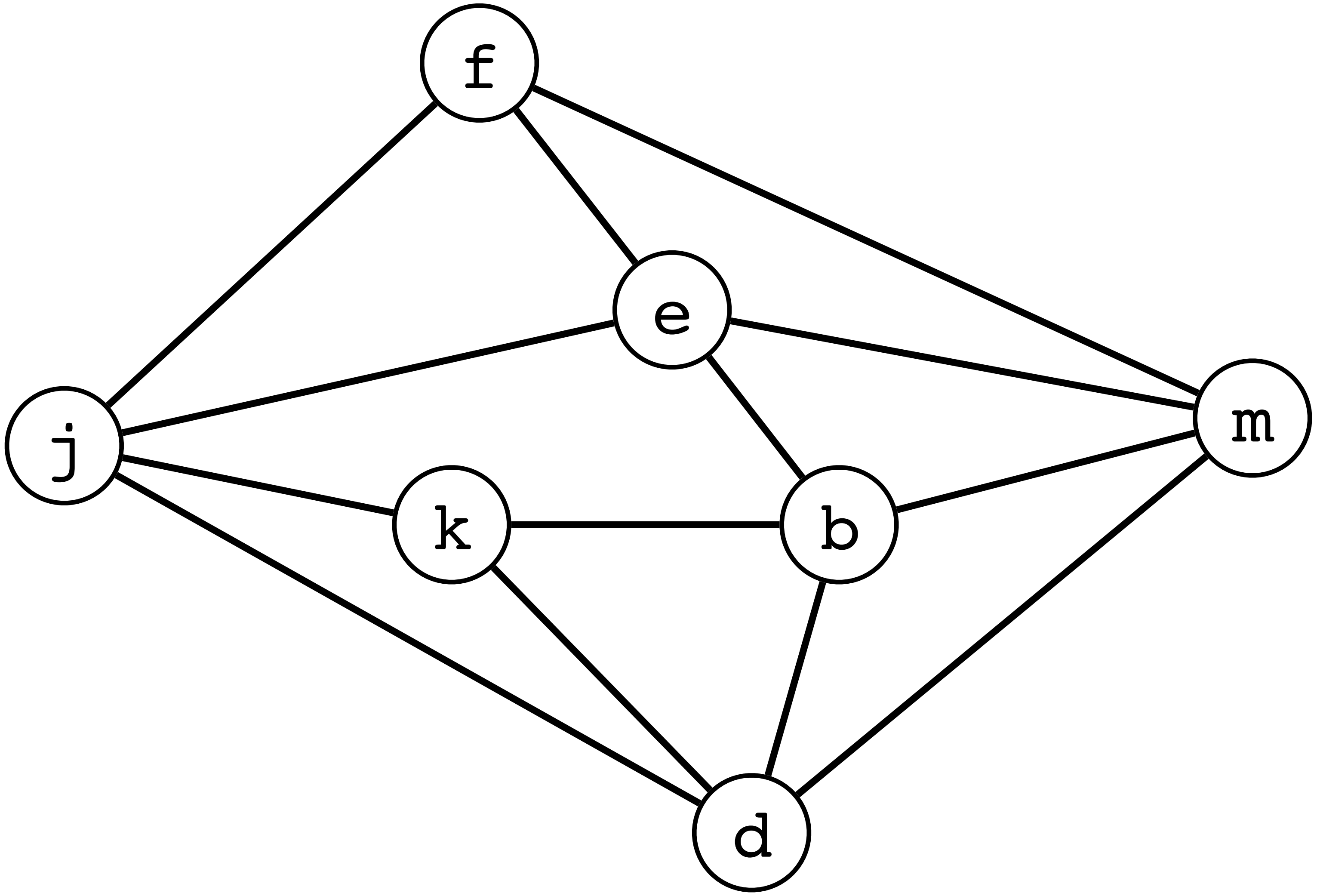


Now we are stuck! No nodes with degree <3

Pick a node to potentially spill

Which Node to Spill?

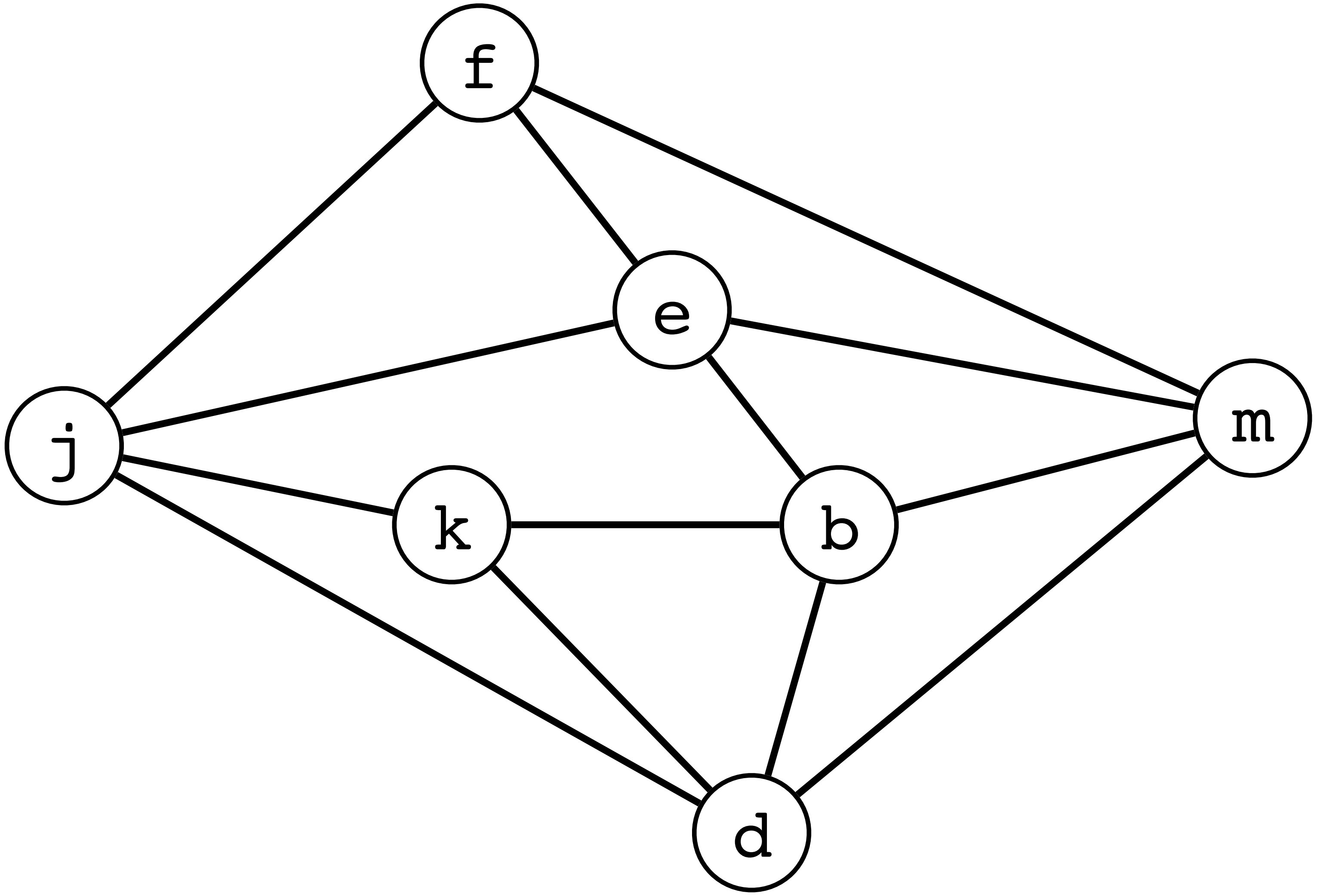
- Want to pick a node (i.e., temp variable) that will make it likely we’ll be able to *k* color graph
  - High degree (≈ live at many program points)
  - Not used/defined very often (so we don’t need to access stack very often)
- E.g., compute **spill priority** of node



$$\frac{\textit{Uses+defs outside loop} + \textit{Uses+defs in loop} \times 10}{\textit{degree of node}}$$

Which Node to Spill?

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



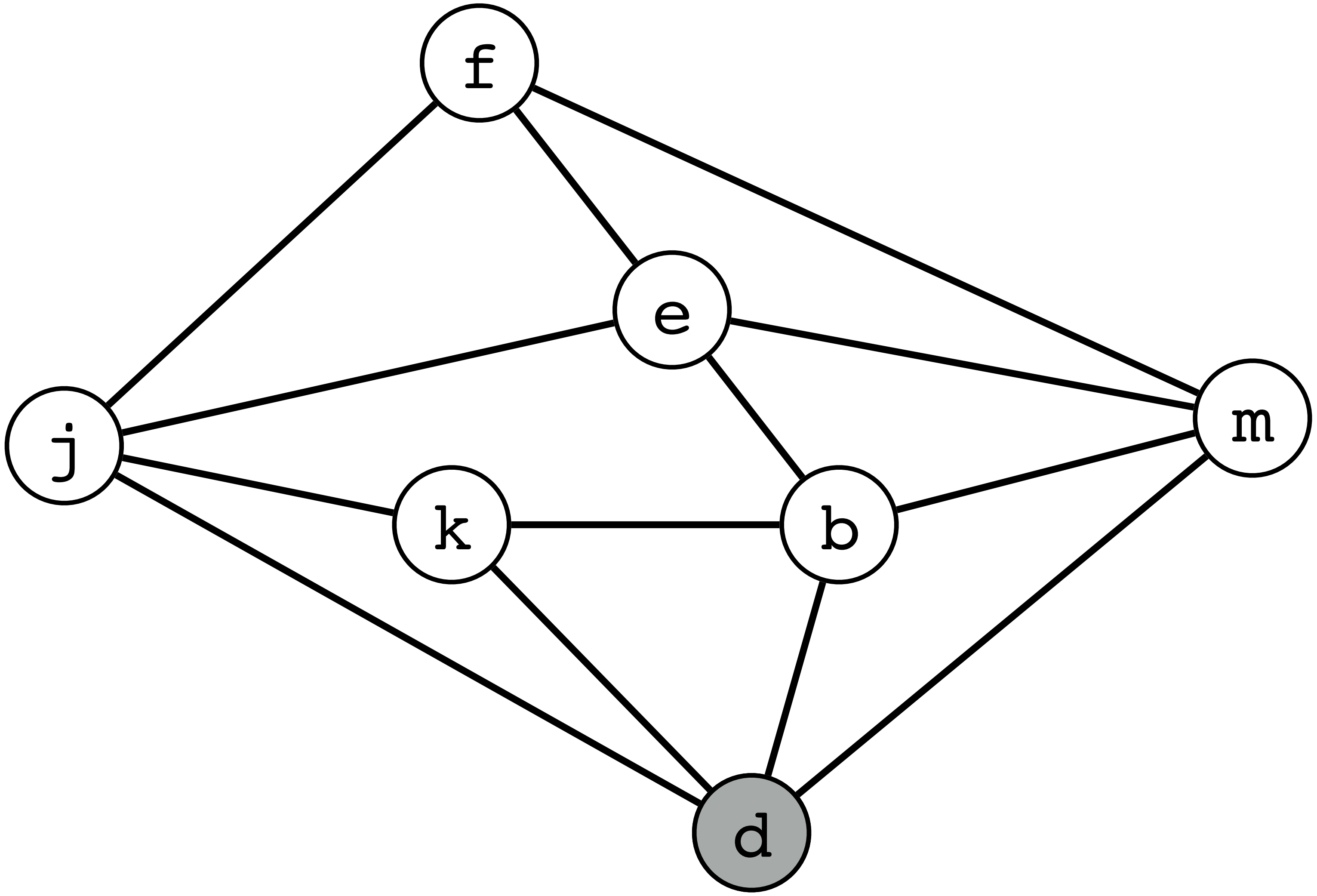
$$\textit{Spill priority} = \frac{\textit{Uses+defs outside loop} + \textit{Uses+defs in loop} \times 10}{\textit{degree of node}}$$

Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`
`g`
`d` *spill?*



Pick a node with small spill priority degree to potentially spill

Simplification (3 registers)

Choose any node with degree <3

Stack:

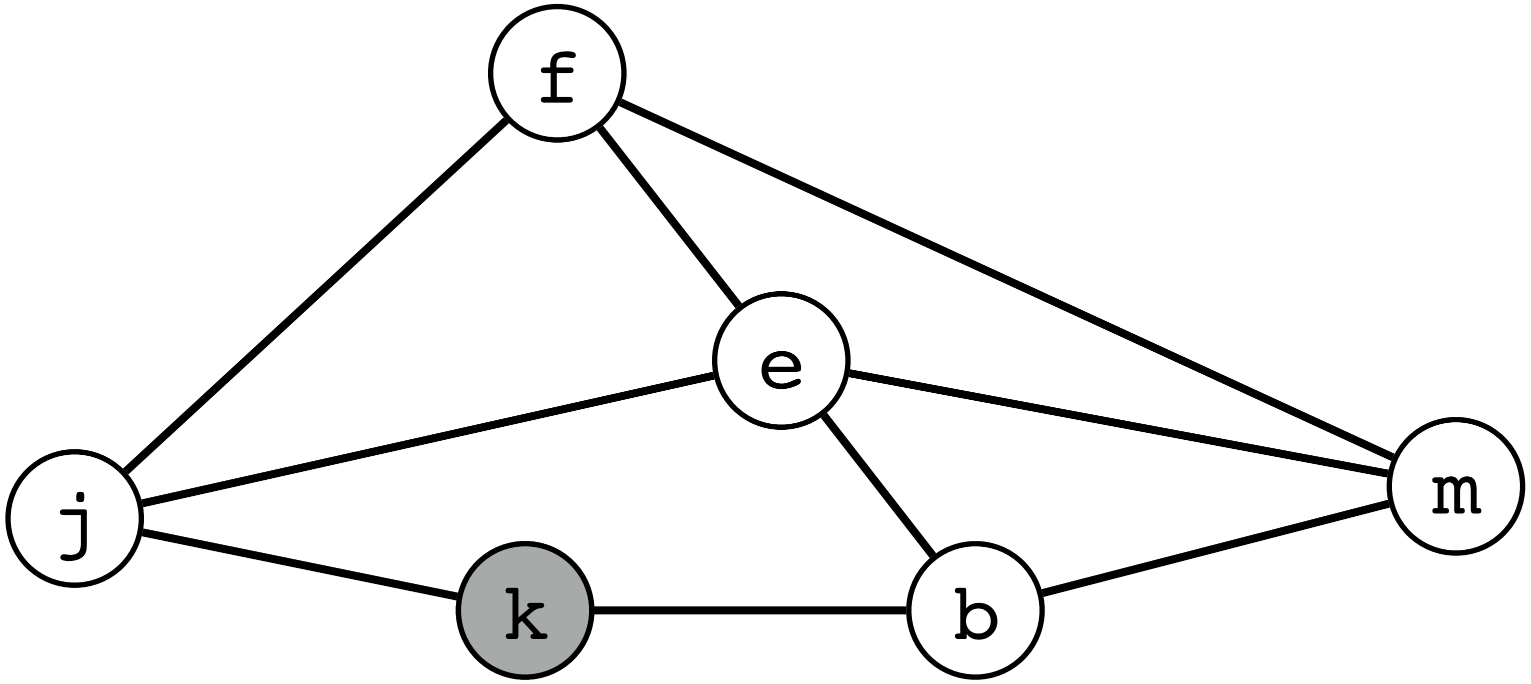
`h`

`c`

`g`

`d` *spill?*

`k`

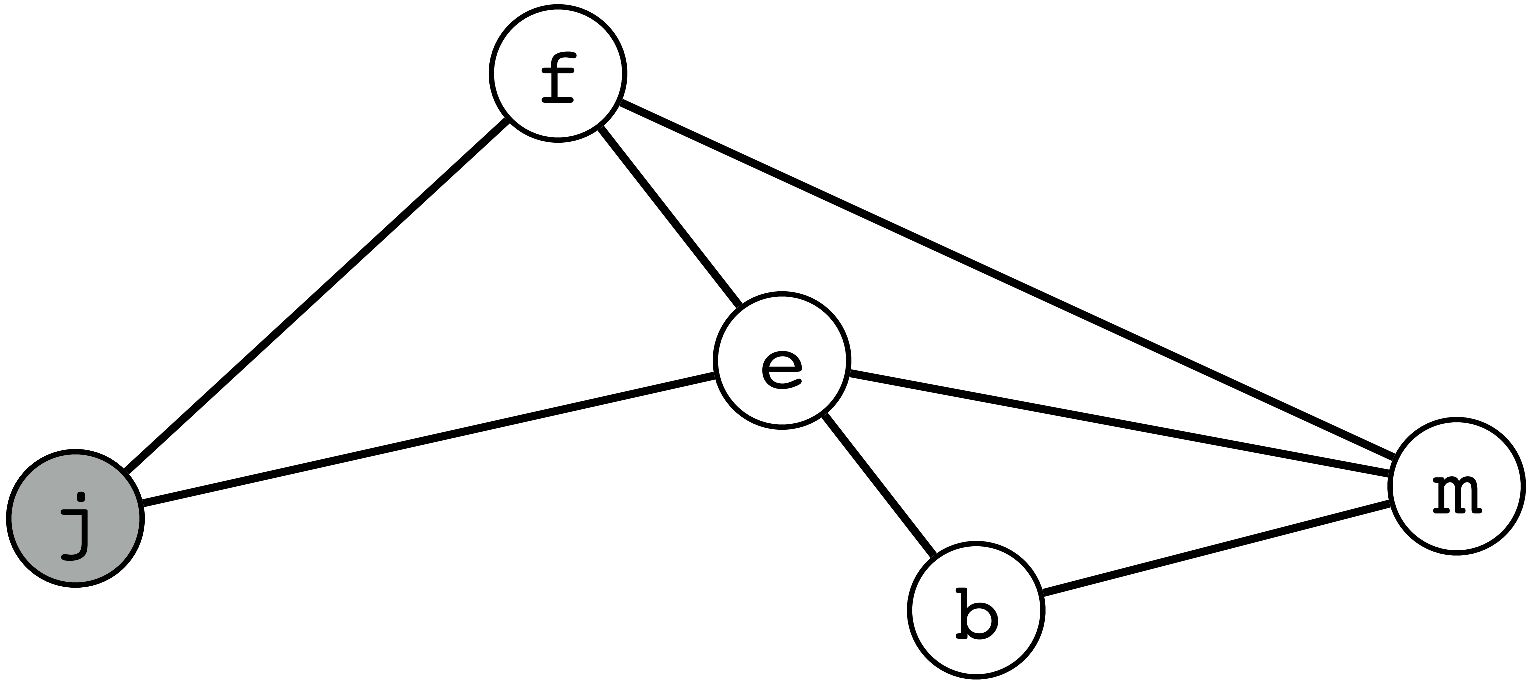


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`
`g`
`d` *spill?*
`k`
`j`



Simplification (3 registers)

Choose any node with degree <3

Stack:

h

c

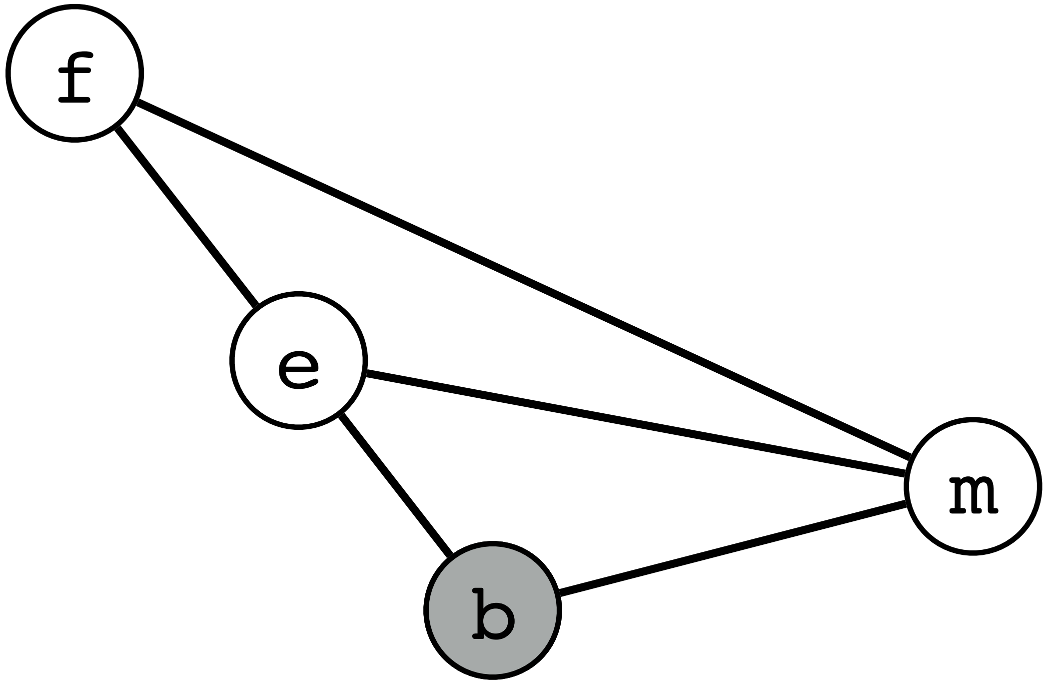
g

d *spill?*

k

j

b

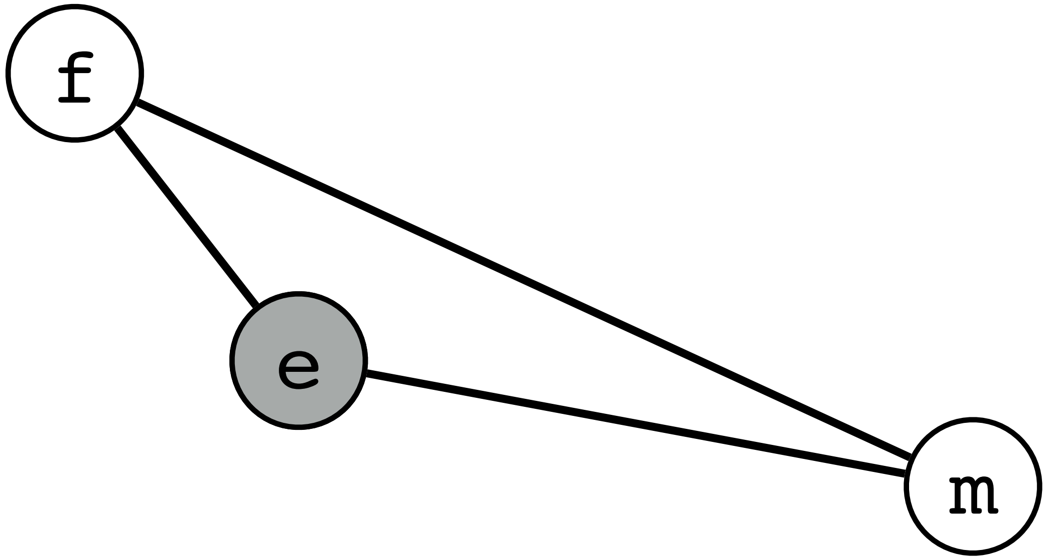


Simplification (3 registers)

Choose any node with degree <3

Stack:

- `h`
- `c`
- `g`
- `d` *spill?*
- `k`
- `j`
- `b`
- `e`

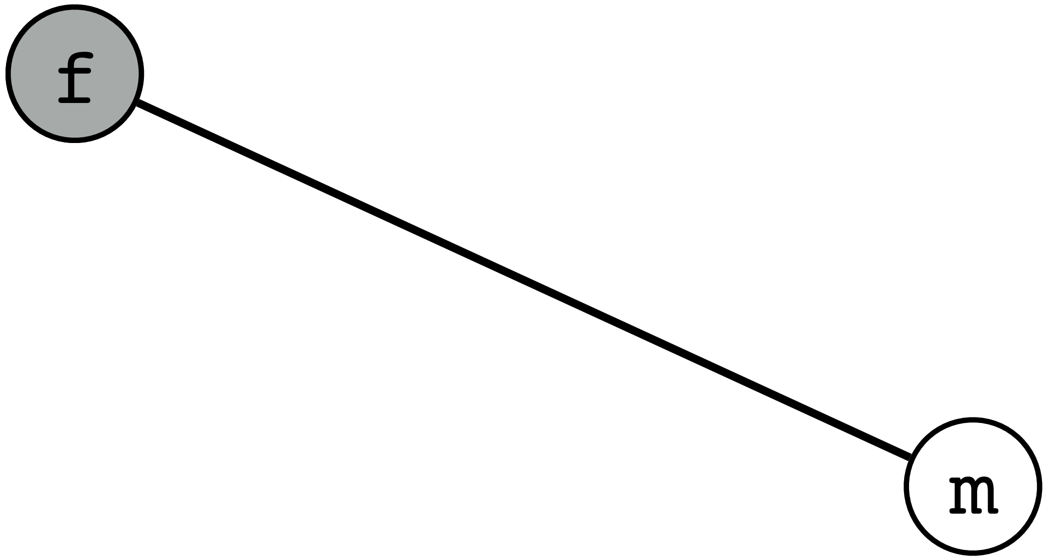


Simplification (3 registers)

Choose any node with degree <3

Stack:

- `h`
- `c`
- `g`
- `d` *spill?*
- `k`
- `j`
- `b`
- `e`
- `f`

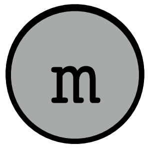


Simplification (3 registers)

Choose any node with degree <3

Stack:

`h`
`c`
`g`
`d` *spill?*
`k`
`j`
`b`
`e`
`f`
`m`



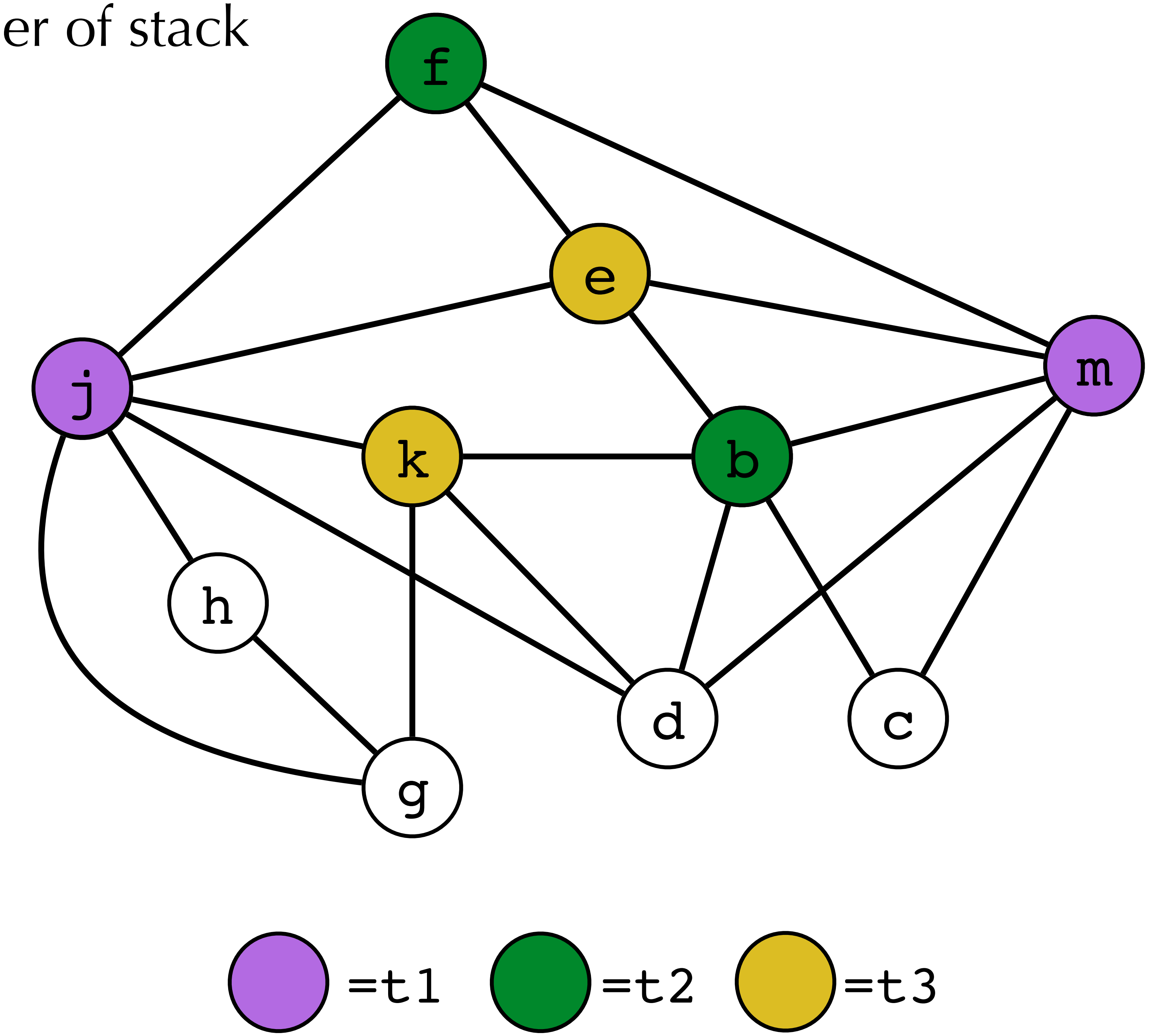
Select (3 registers)

Graph is now empty!

Color nodes in order of stack

Stack:

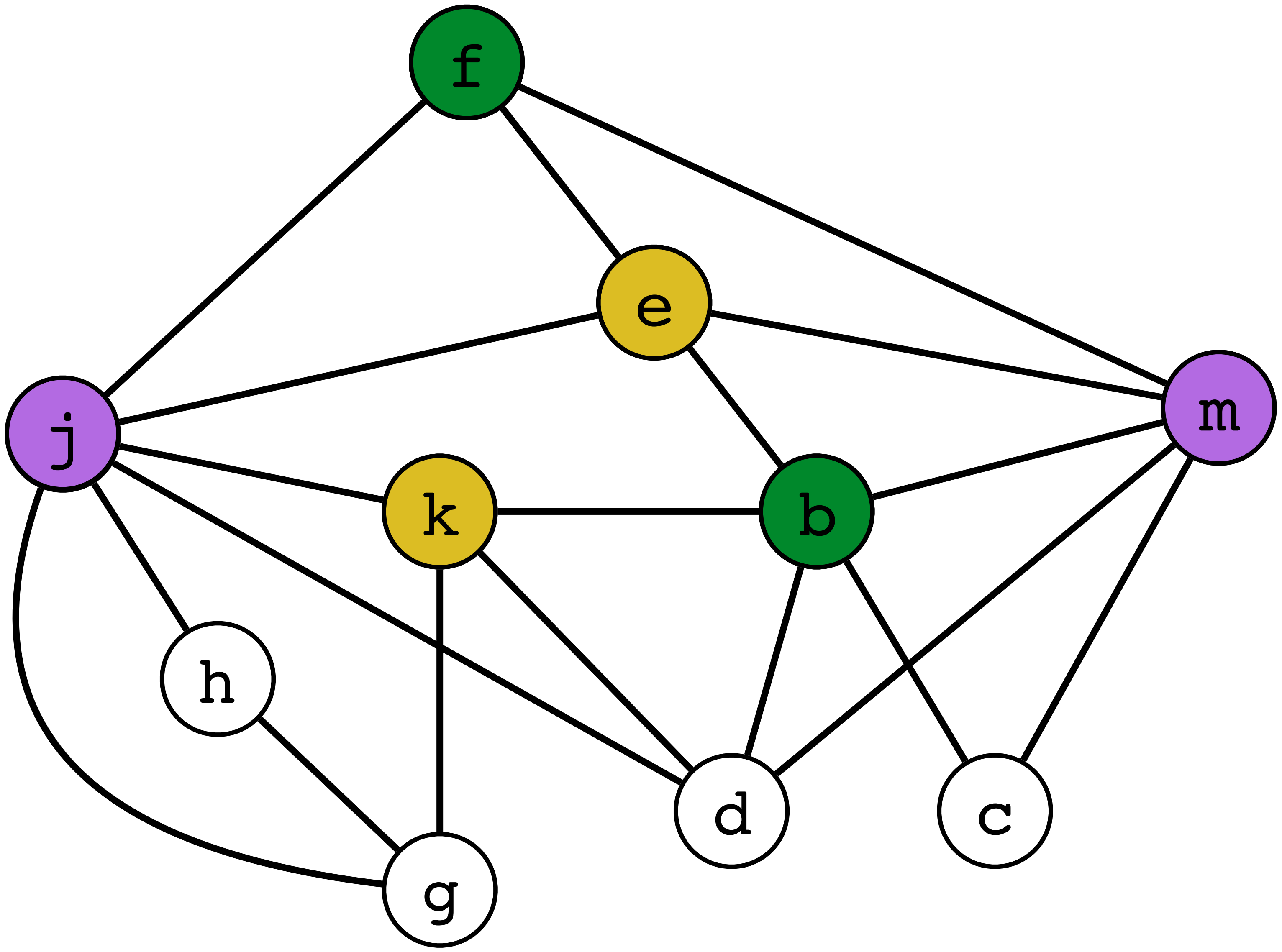
h
c
g
d *spill?*
k
j
b
e
f
m



Select (3 registers)

Stack:

`h`
`c`
`g`
`d` *spill?*



We got unlucky!

In some cases a potential spill node is still colorable, and the Select phase can continue.

But in this case, we need to rewrite...

🟣 `=t1` 🟢 `=t2` 🟡 `=t3`

Select (3 registers)

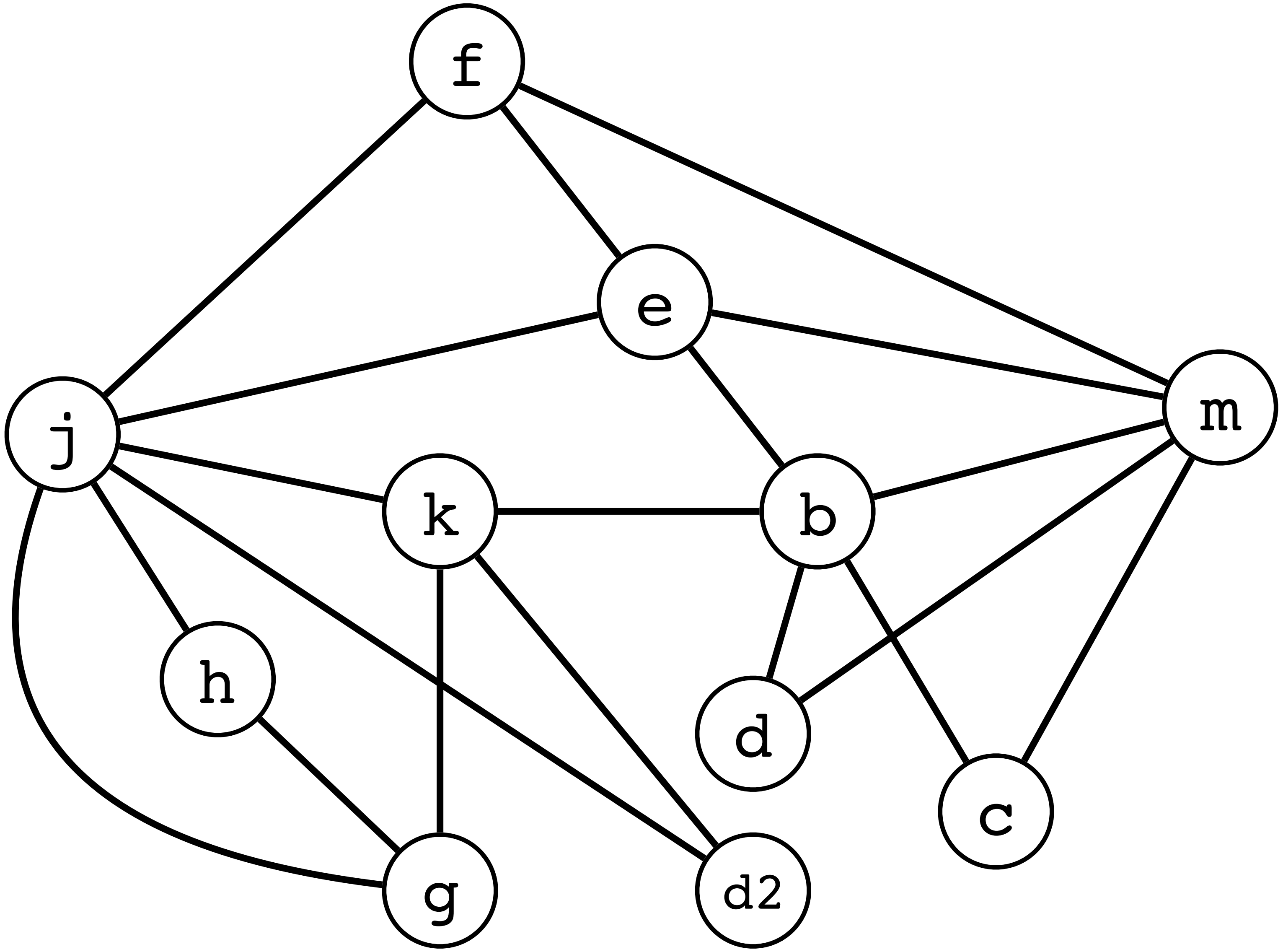
- Spill d

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

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
*<fp+doff>:=d
k := m + 4
j := b
d2:=*<fp+doff>
{live-out: d2,j,k}
```

Build

```
{live-in: j, k}
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
*<fp+doff>:=d
k := m + 4
j := b
d2:=*<fp+doff>
{live-out: d2,j,k}
```

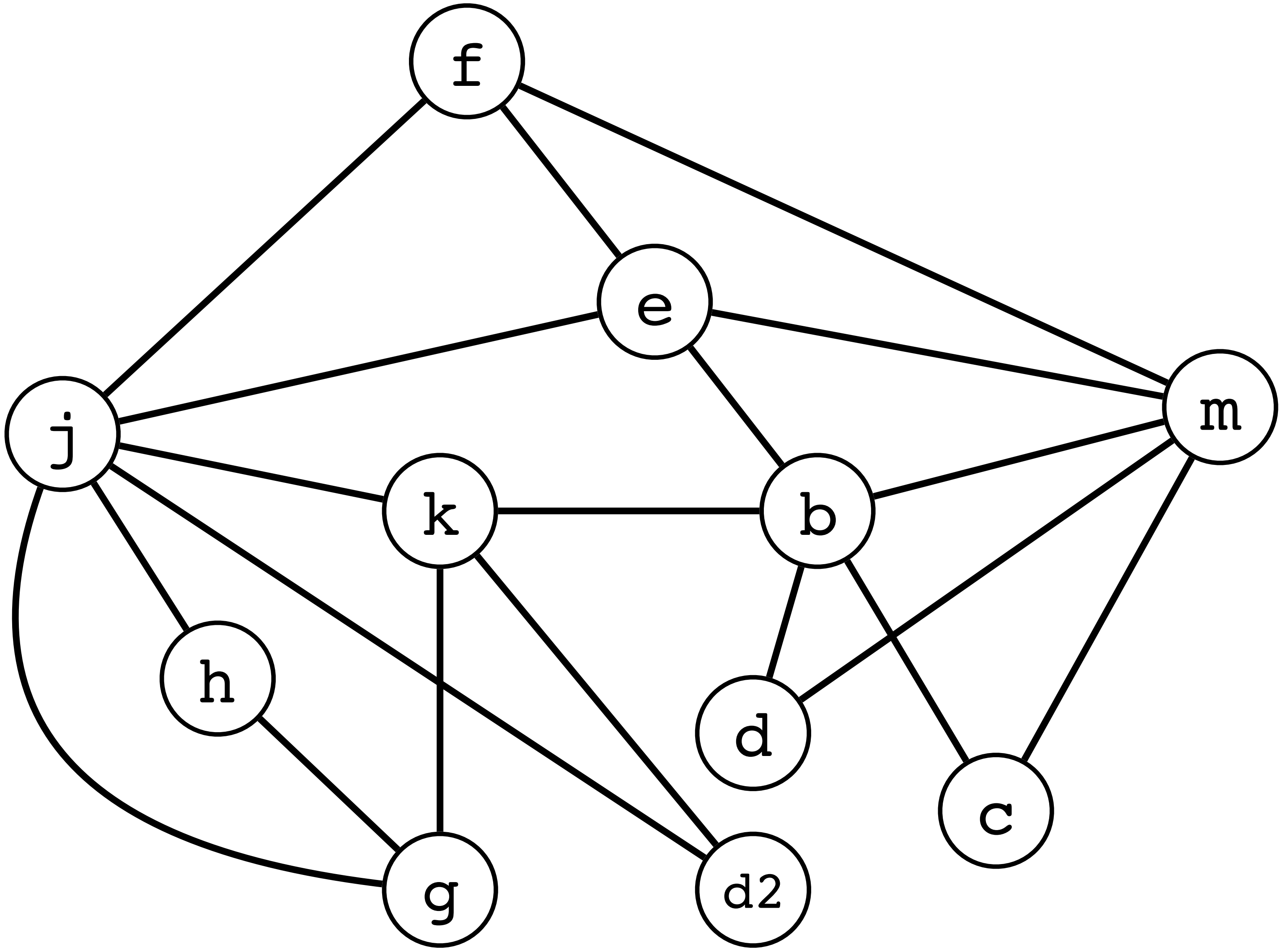


Simplification (3 registers)

Choose any node with degree <3

Stack:

h
c
g
d
d2
k
b
m
e
f
j



This time we succeed and
will be able to complete Select phase successfully!