Topic 12: Register Allocation

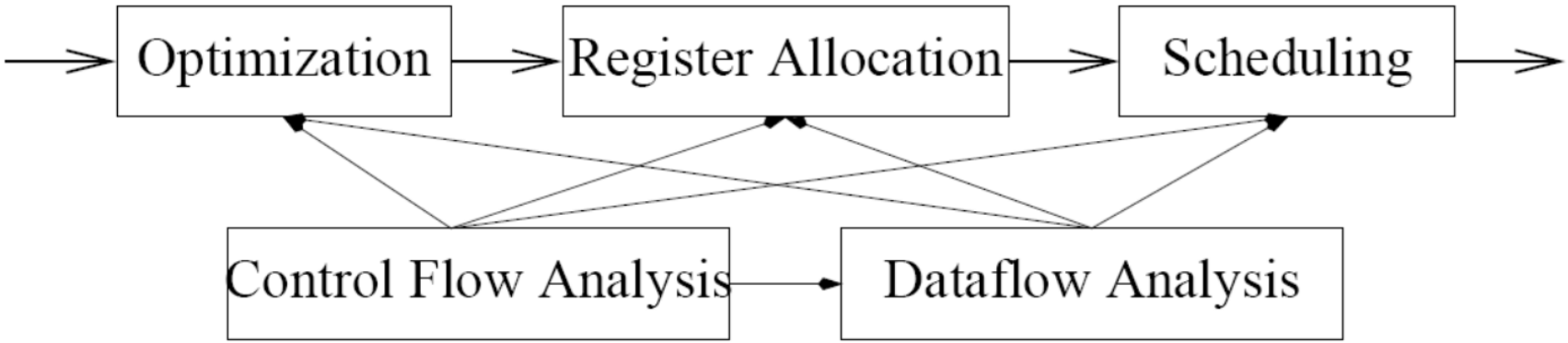
COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Structure of backend



Register allocation

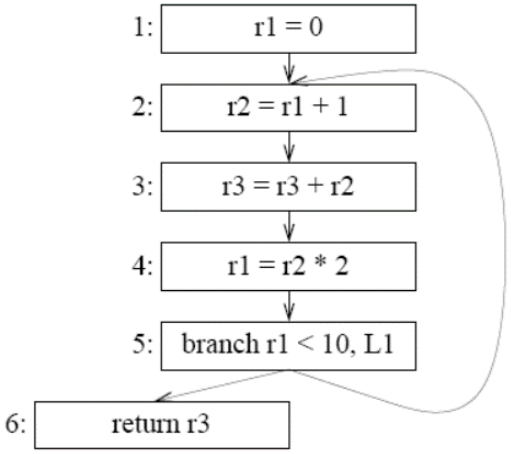
- assigns machine registers (finite supply!) to virtual registers
- based on liveness analysis: interference graph
- primary approach: graph coloring
- spilling
  - needed in case of insufficient supply of machine registers
  - idea: hold values in memory (stack frame)
  - transfer to/from registers to perform arithmetic ops, conditional branches, …
- architecture-specific requirements:
  - caller/callee-save
  - floating point vs integer, …

Recap: interference graph

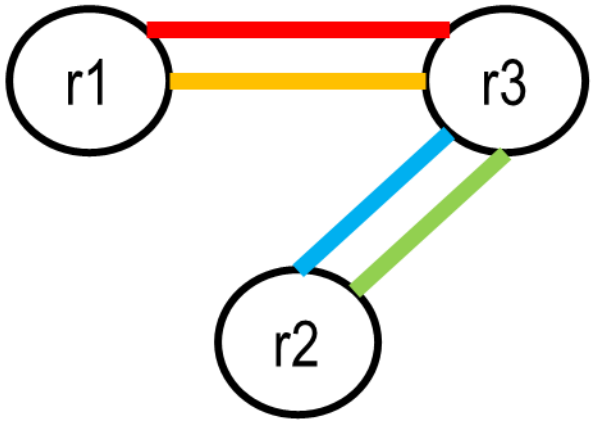
Liveness conflicts: virtual registers **x** and **y** **interfere** if there is a node **n** in the CFG such that **x** and **y** are both LiveOut at **n**.

Representation: conflict/interference graph:

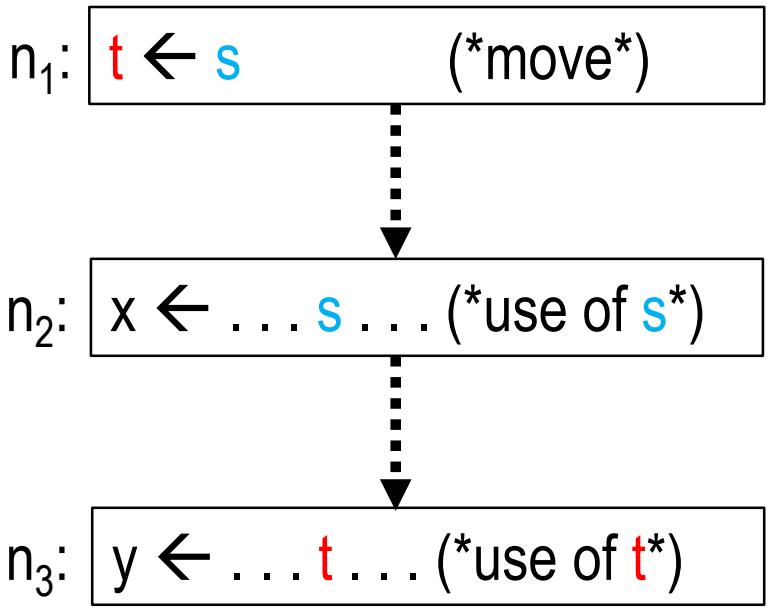
- each virtual register represented by one node
- Interference between **x** and **y**: undirected edge between nodes **x** and **y**



Node	*DEF*	OUT	IN
1	r1	r1,r3	r3
2	r2	r2,r3	r1,r3
3	r3	r2,r3	r2,r3
4	r1	r1,r3	r2,r3
5	-	r1, r3	r1,r3
6	-		r3



Interference graph: optimization for MOVEs



Virtual registers s and t

- are both live-out at n_1
- hence interfere formally

- will hence be assigned different registers

But: we'd like them to share a register, and to eliminate the move instruction!

Solution: treat move instructions differently during interference analysis

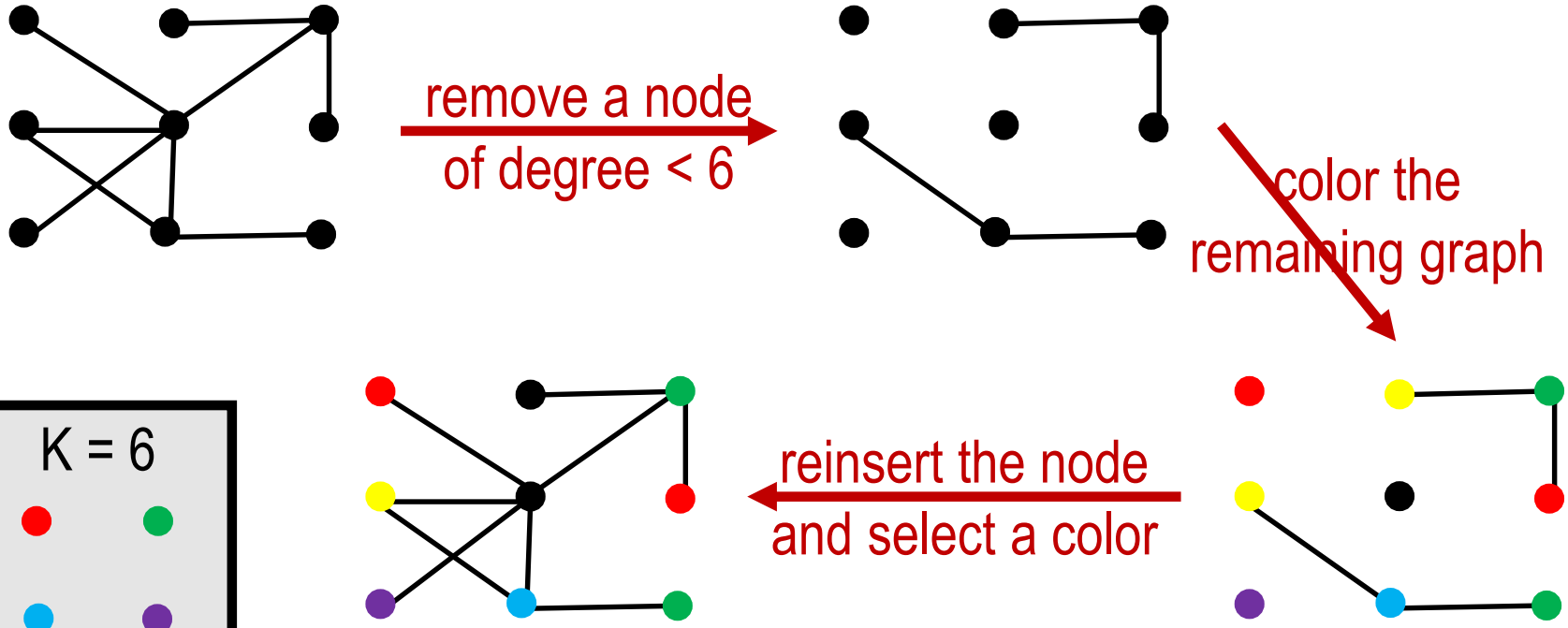
For n: a ← c (*move*) and liveOut(n) = $\{b_1, \ldots, b_k\}$, only add edges

for those b_i that are different from c.

Graph coloring using Kempe's heuristics (1879)

Observation:

- suppose G has a node m with < K neighbors
- if G – {m} can be K-1 colored, G can be K-colored:
  - m's neighbors use at most K-1 colors in G - {m}
  - so can reinsert m into G and select a color

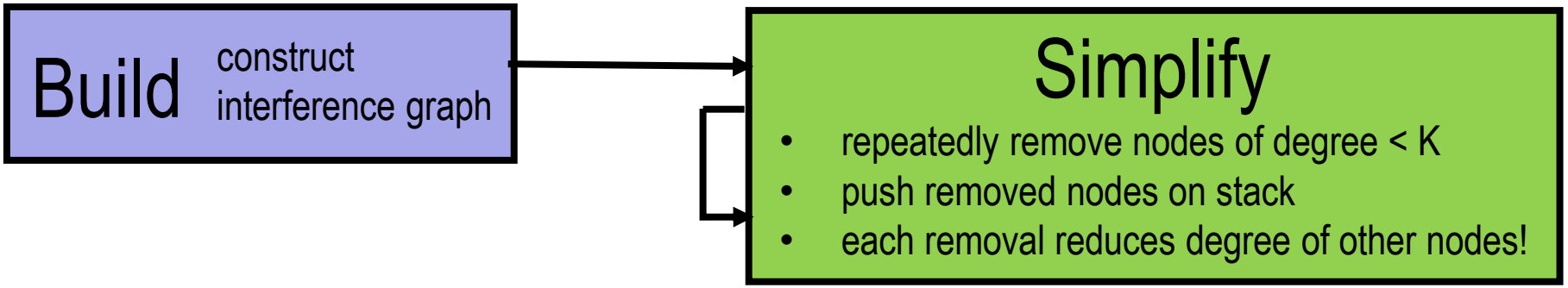


➔ recursive (stack-based) algorithm

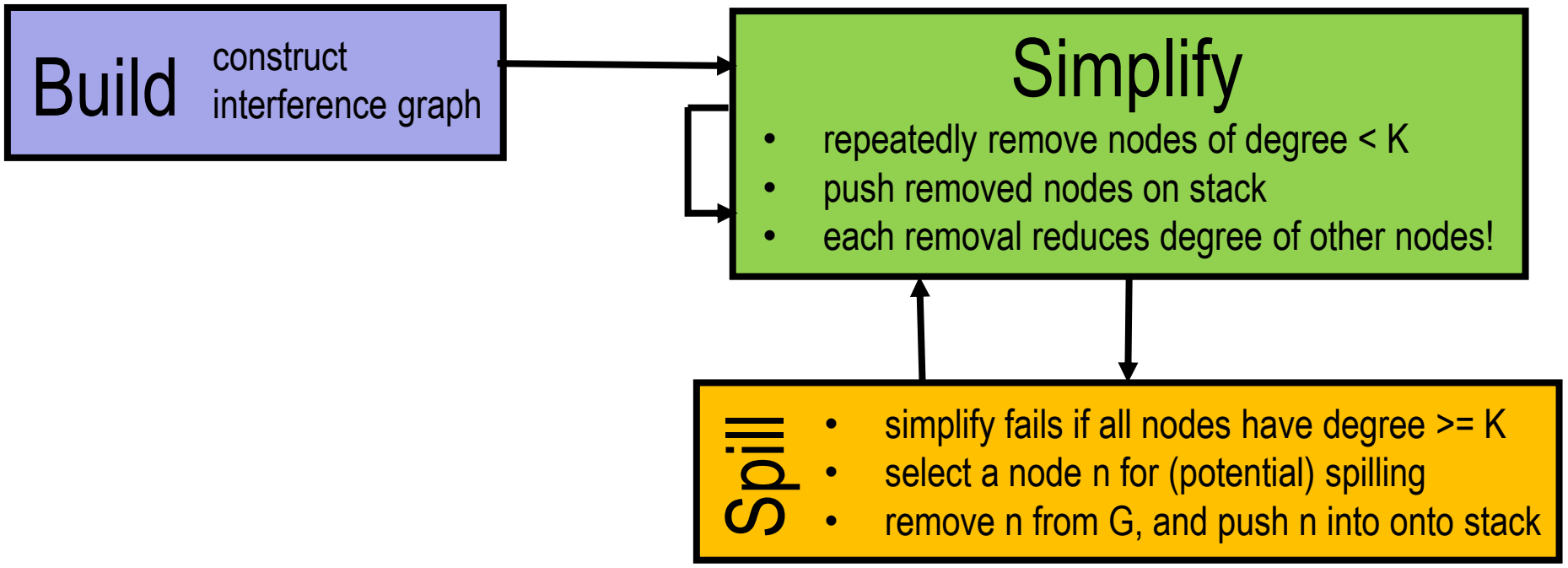
Optimistic coloring using Kempe

Build construct interference graph

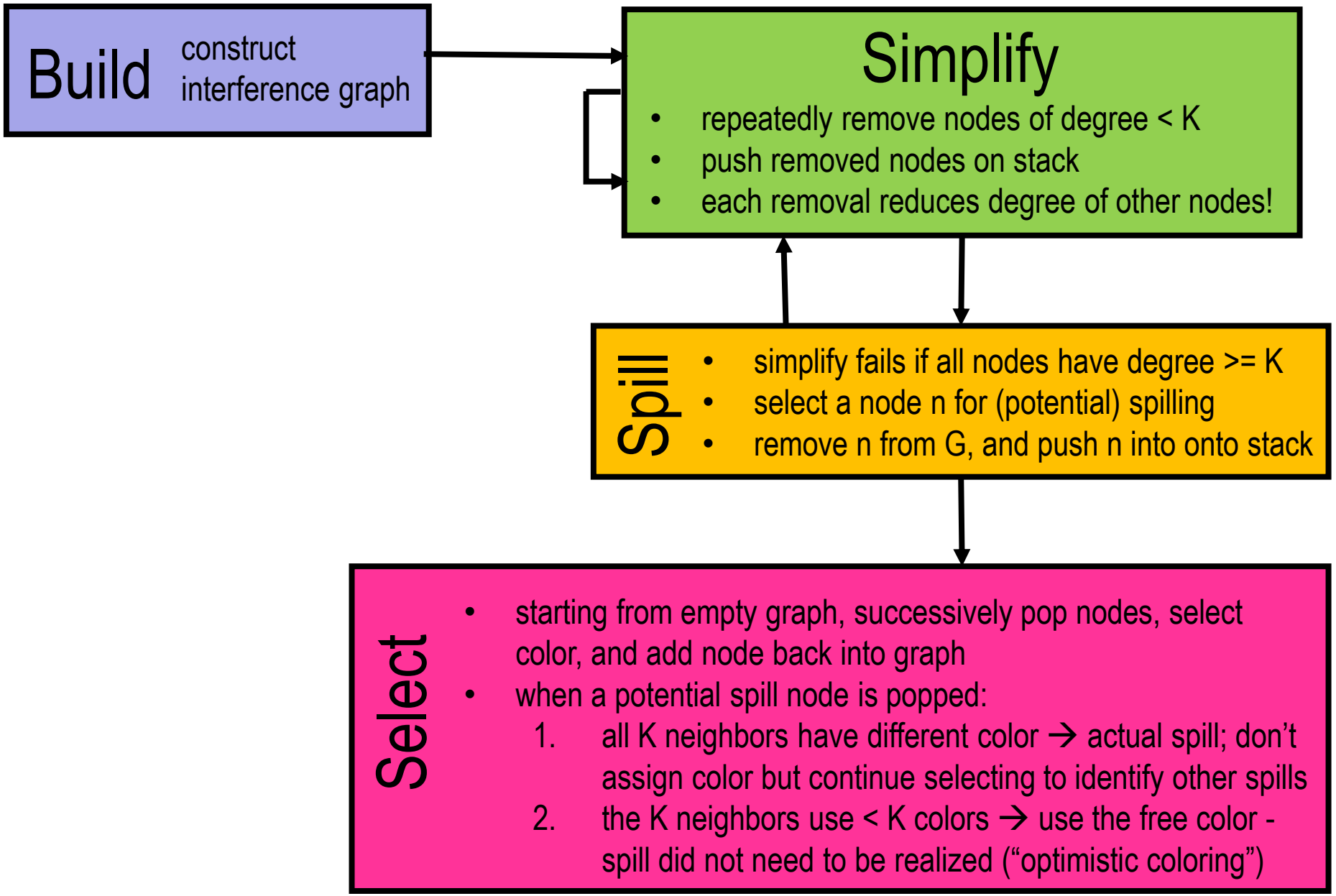
Optimistic coloring using Kempe



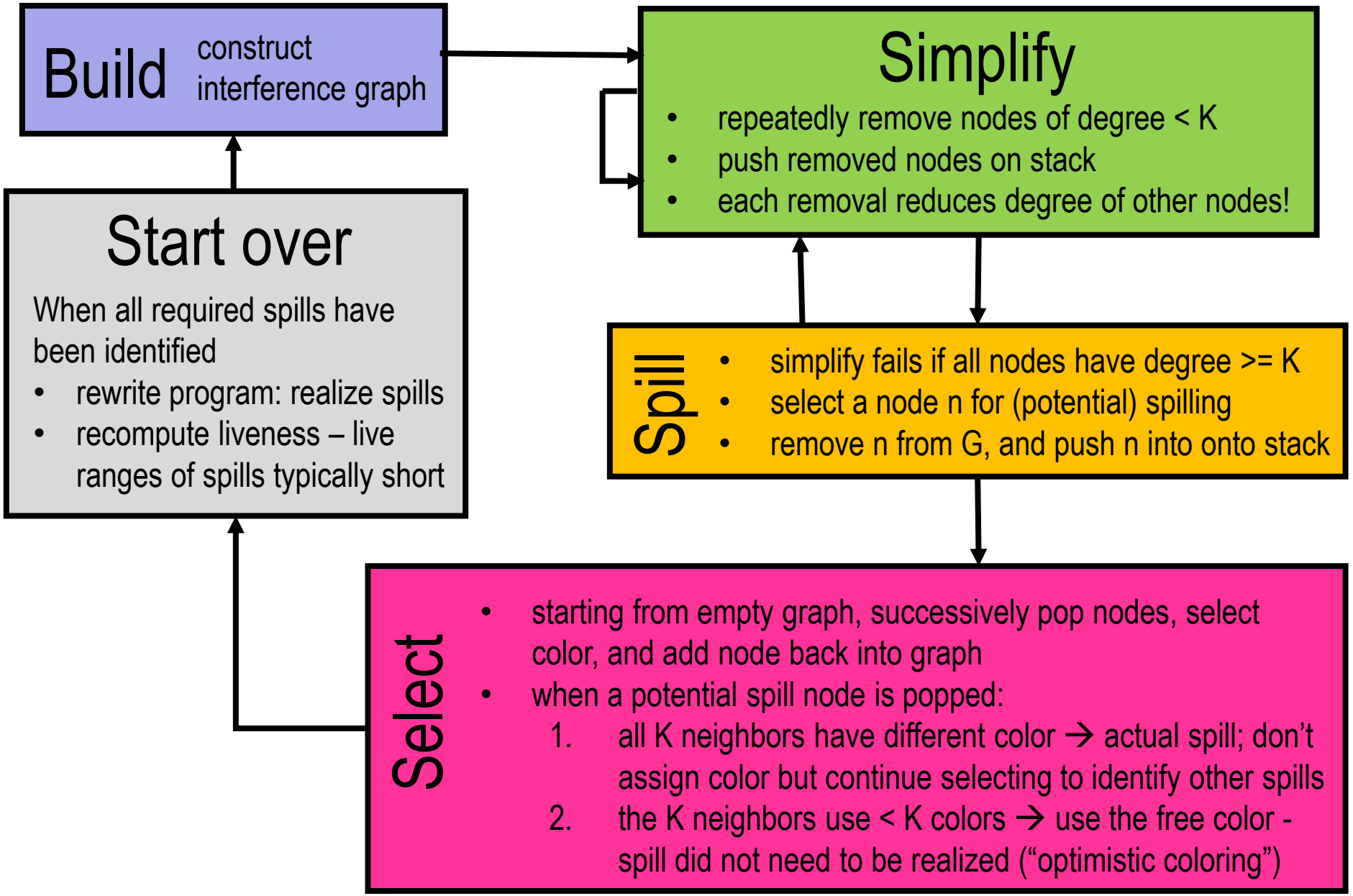
Optimistic coloring using Kempe



Optimistic coloring using Kempe

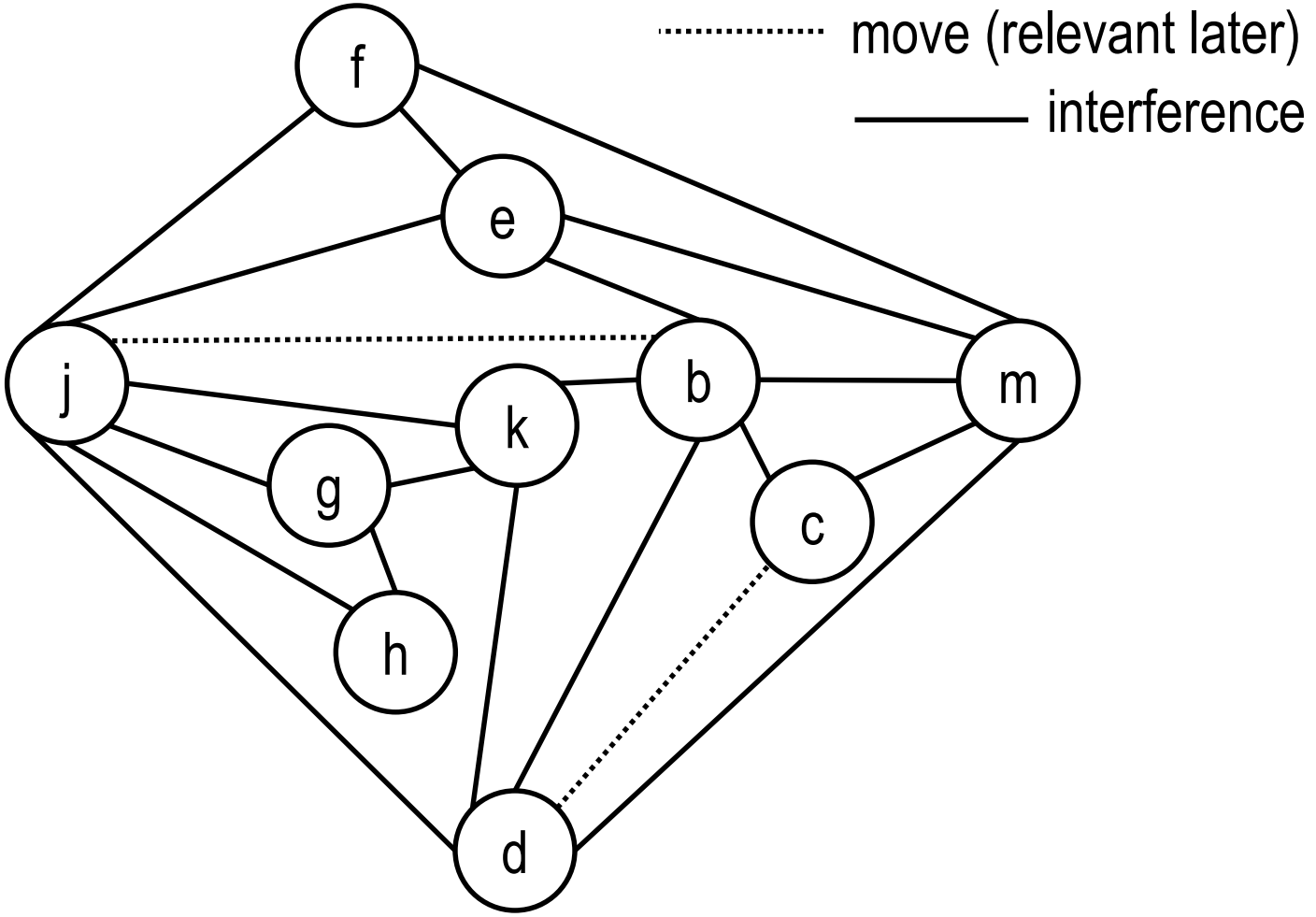


Optimistic coloring using Kempe



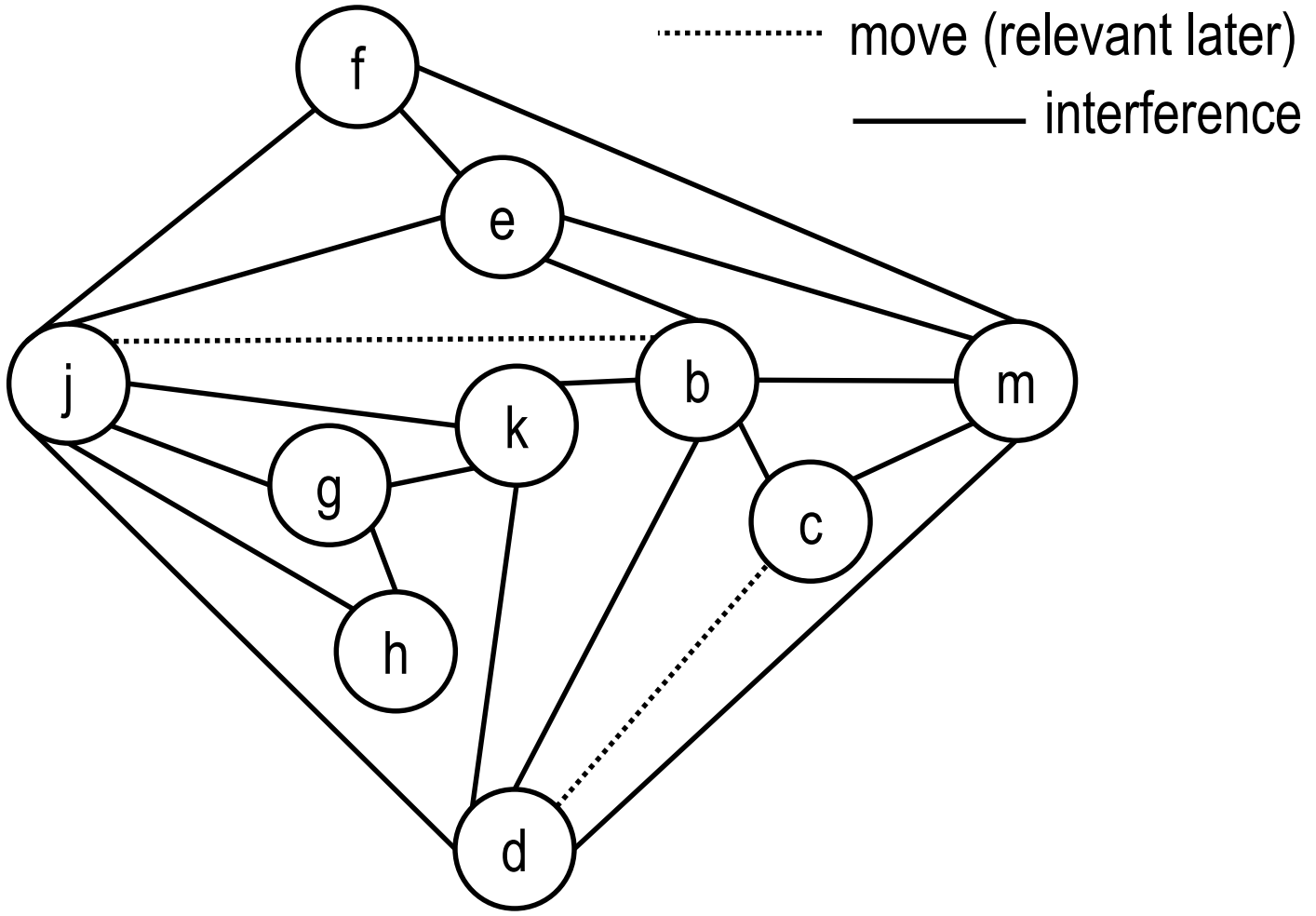
Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```

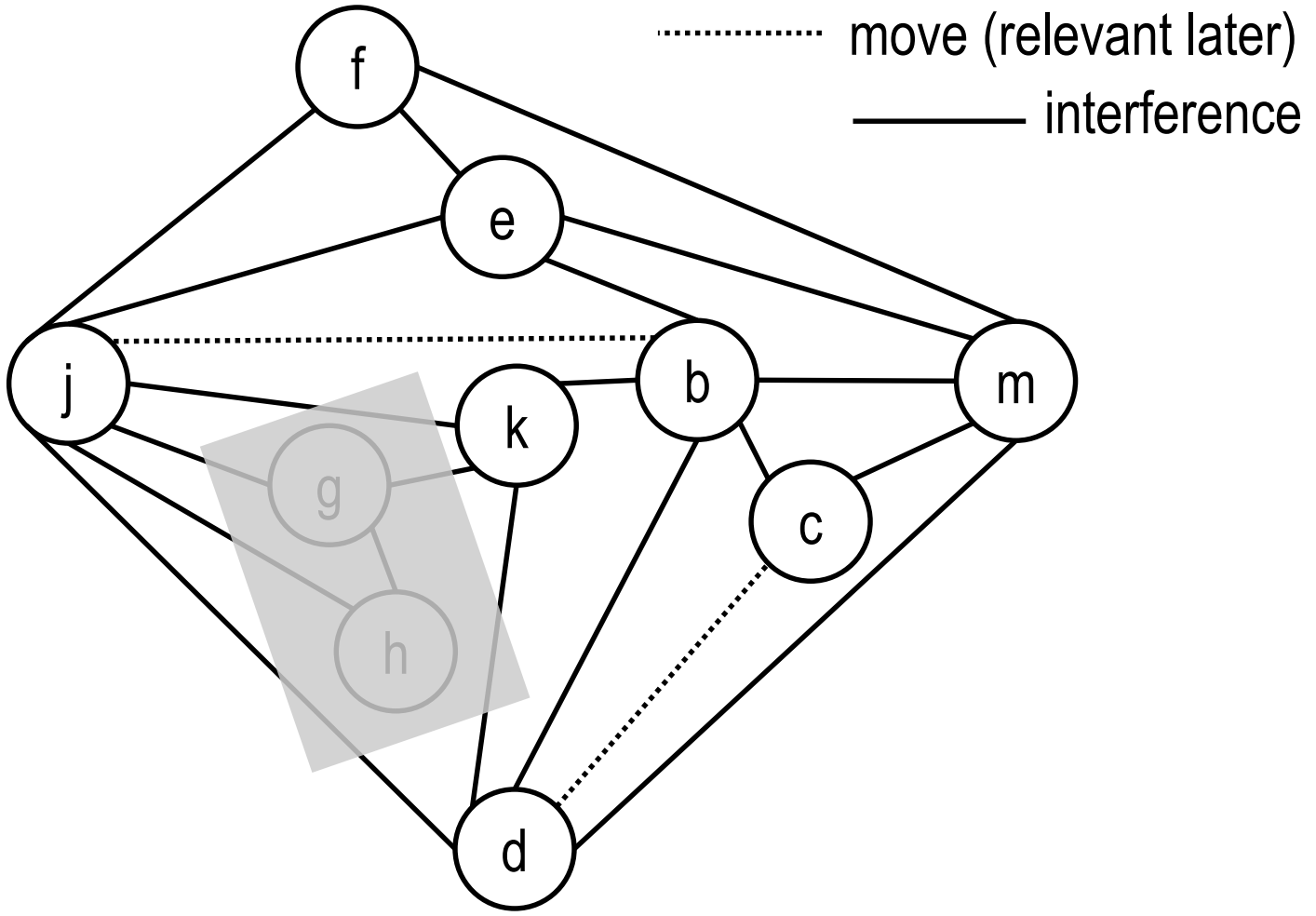


Nodes of degree < K: c, g, h, f

→ push g, h

Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



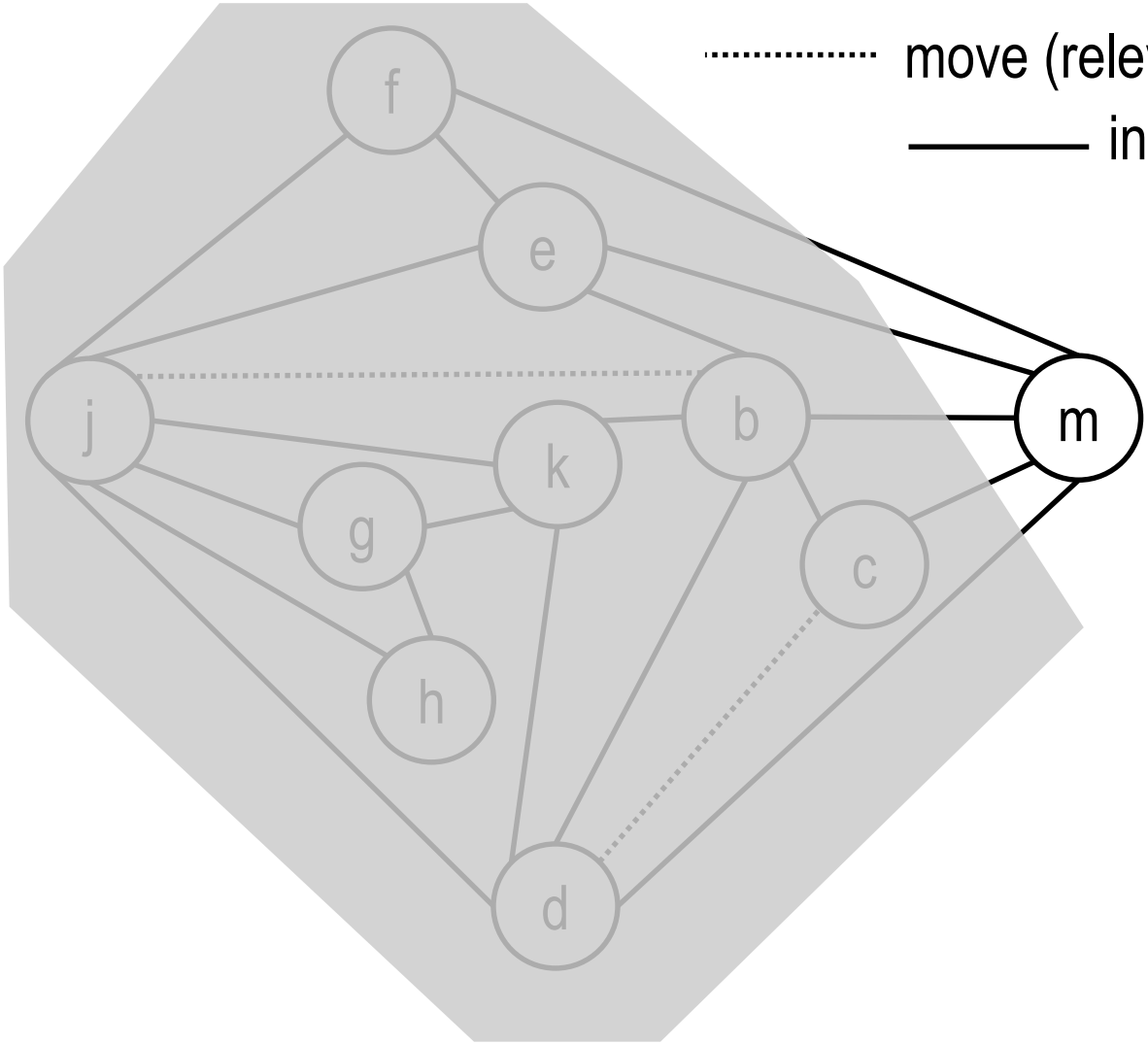
Nodes of degree < K: c, f

→ push g, h

Next: **push** k, d, j, e, f, b, c

Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



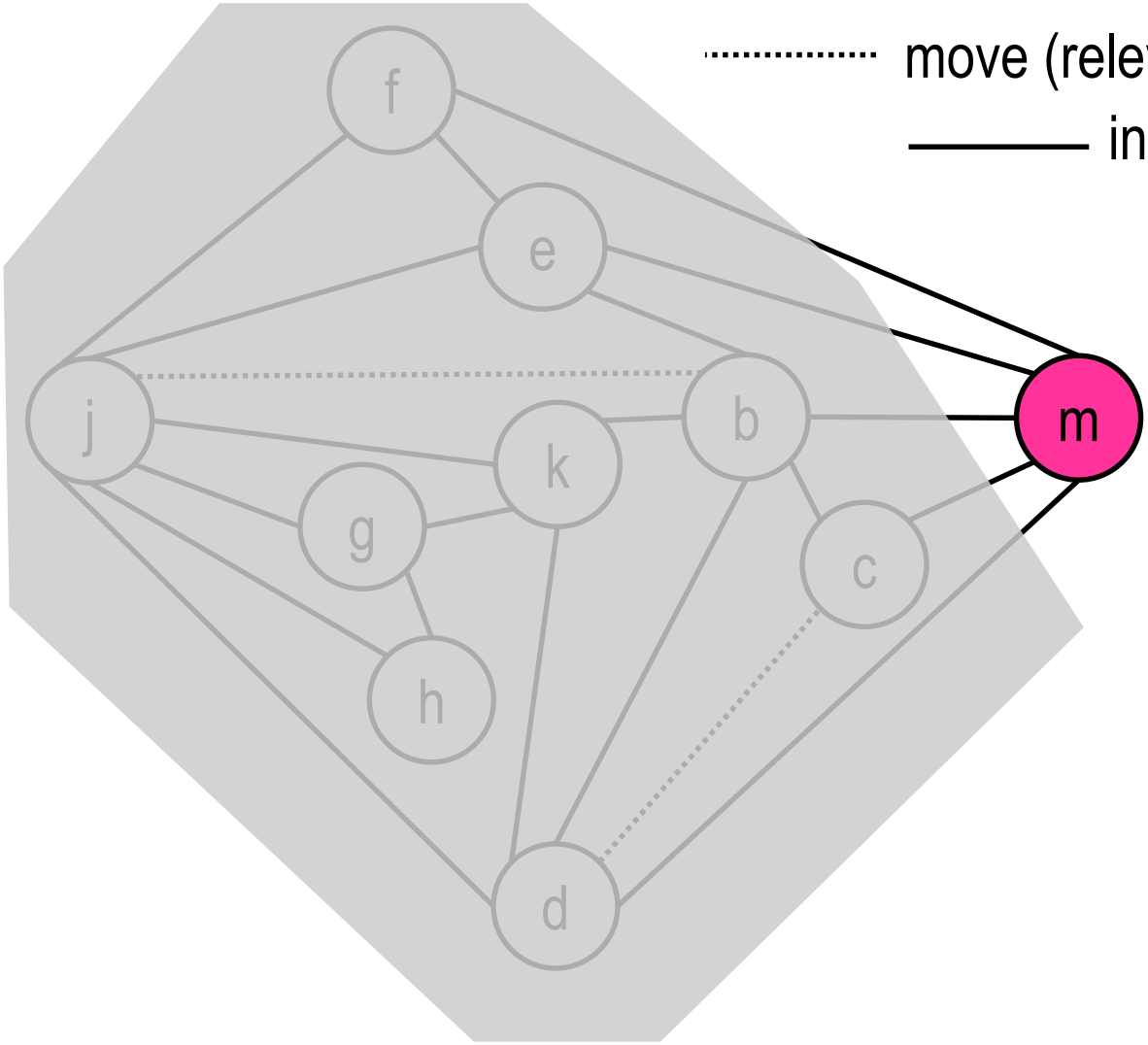
Nodes of degree < K: m

→ push g, h, k, d, j, e, f, b, c

Next: **push** m, **pop** m

Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



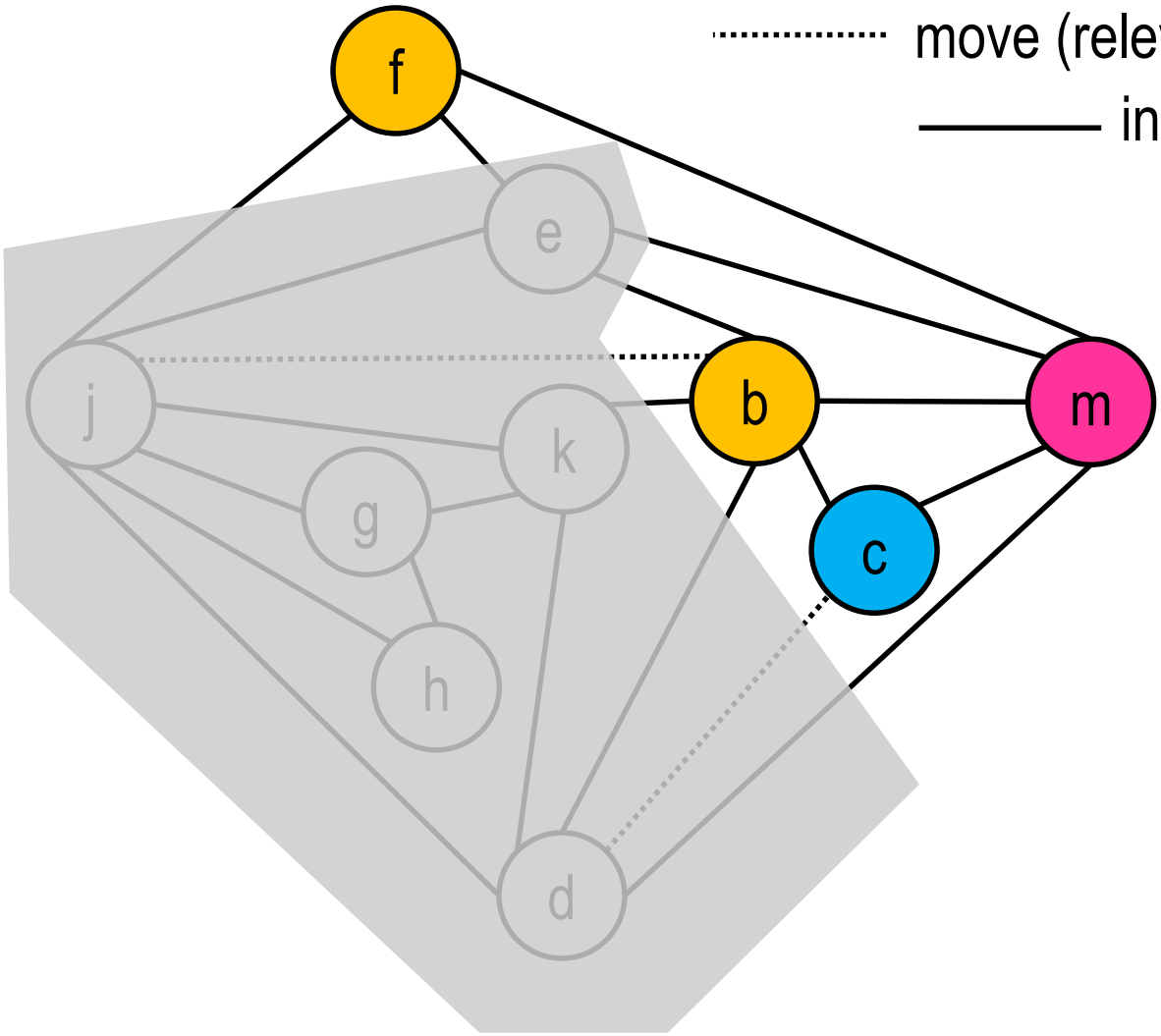
Nodes of degree < K:

→ push g, h, k, d, j, e, f, b, c

Next: **pop** c, b, f

Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



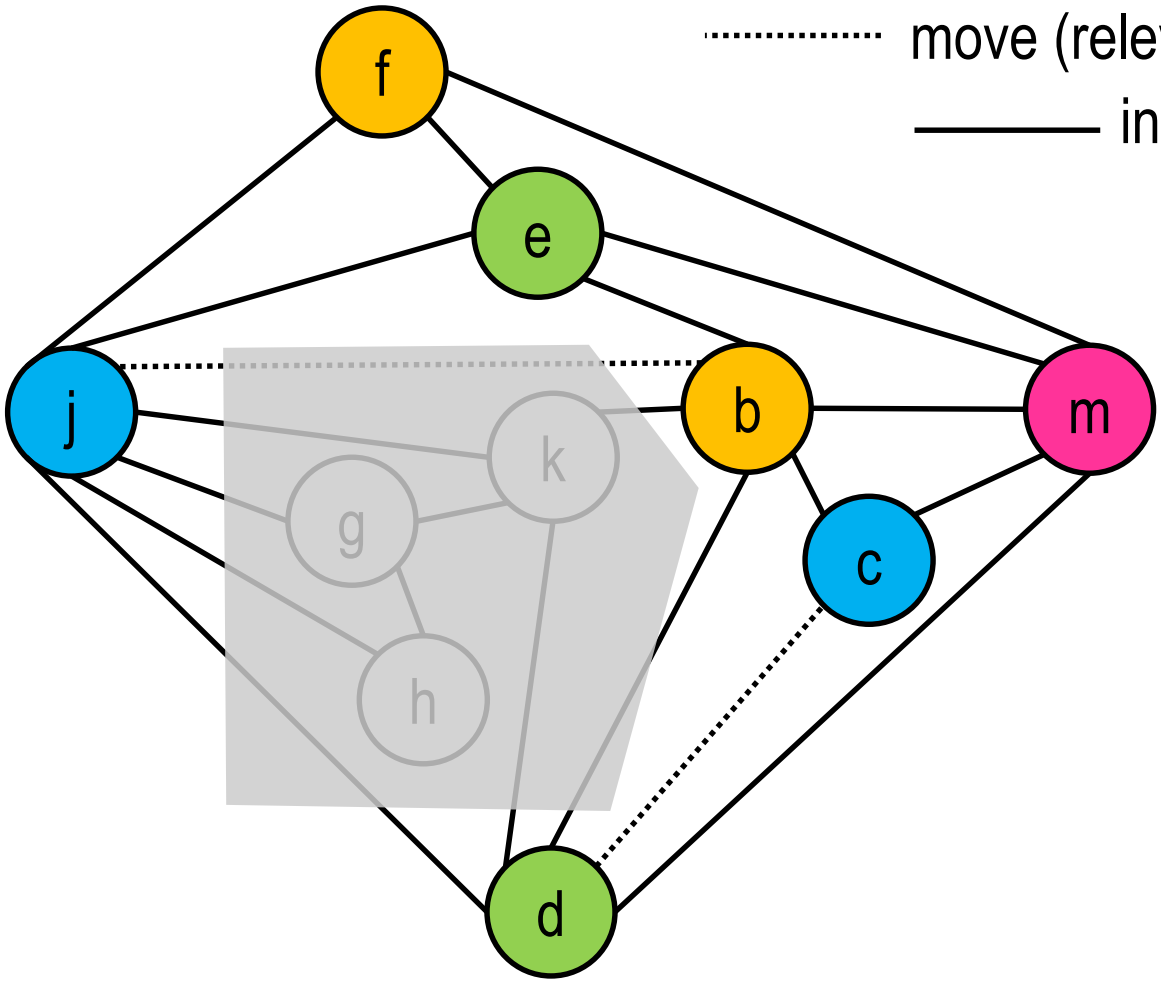
Nodes of degree < K:

→ push g, h, k, d, j, e

Next: **pop** e, j, d

Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



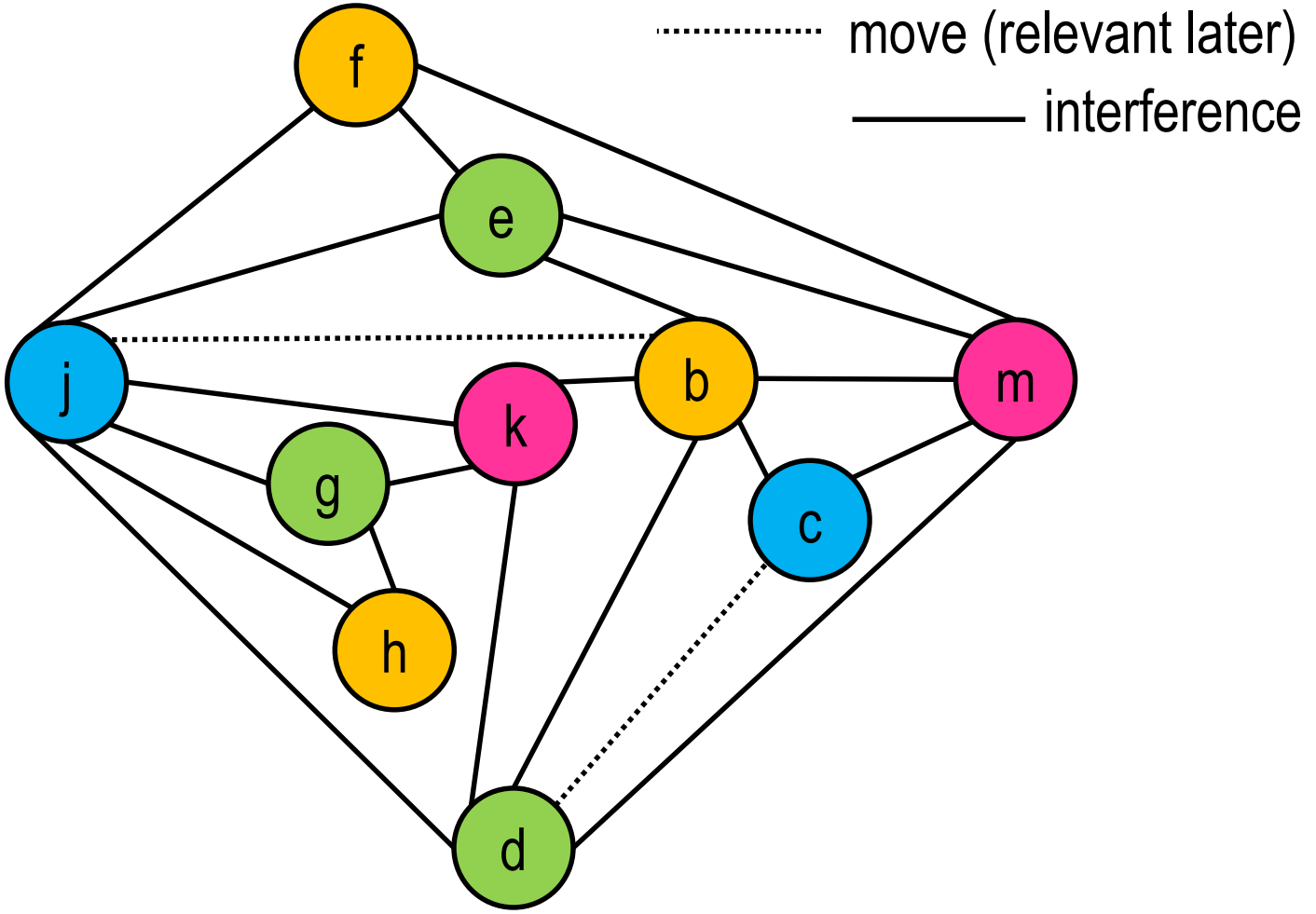
Nodes of degree < K:

→ push g, h, k

Next: **pop** k, h, g

Basic coloring: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



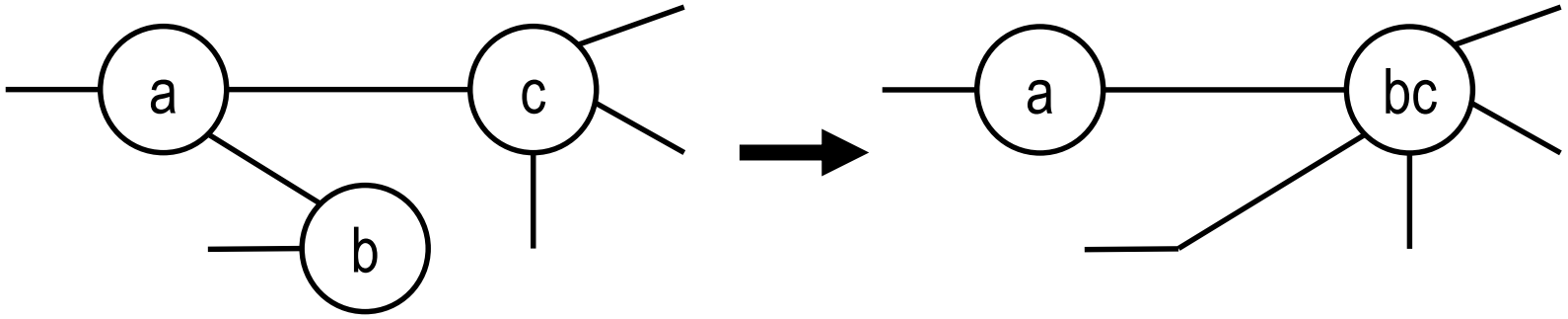
Nodes of degree < K:

→ Stack empty

Done – no spilling needed

Register coalescing

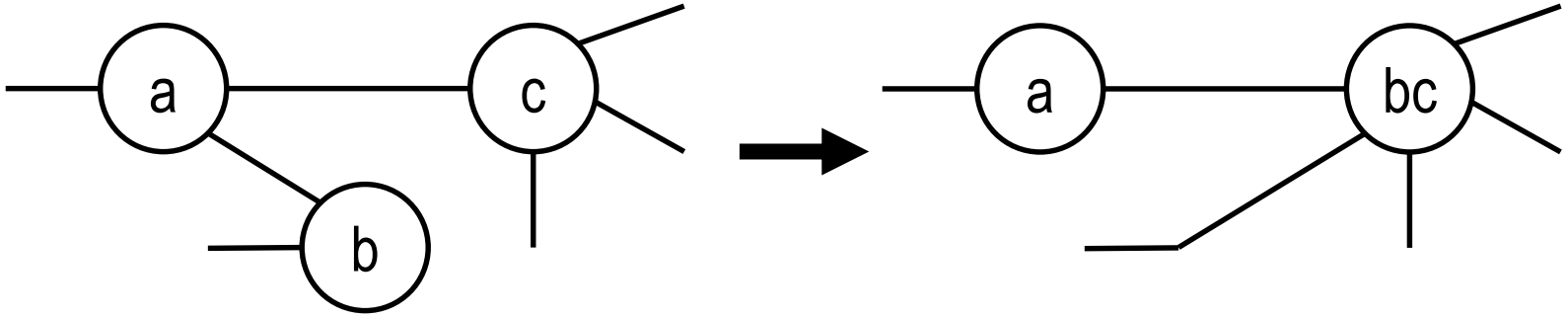
Nodes in the conflict graph can be coalesced, provided that they don't interfere; edges of coalesced node = union of edges associated with original nodes



In particular: if source and dest of a move don't interfere, coalescing allows one to eliminate the move instruction.

Register coalescing

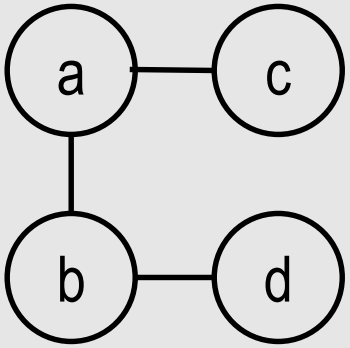
Nodes in the conflict graph can be coalesced, provided that they don't interfere; edges of coalesced node = union of edges associated with original nodes



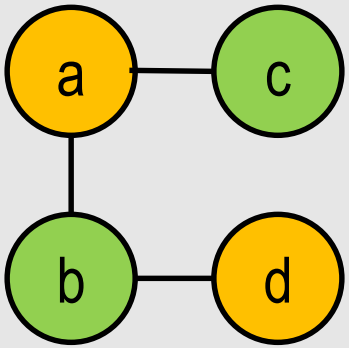
In particular: if source and dest of a move don't interfere, coalescing allows one to eliminate the move instruction.

But: coalescing before coloring may make graph not colorable!

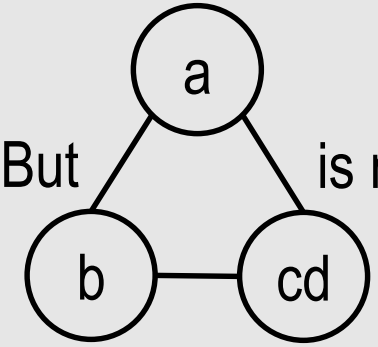
Example:



is 2-colorable:



. But

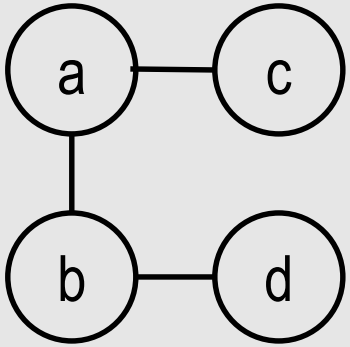


is not.

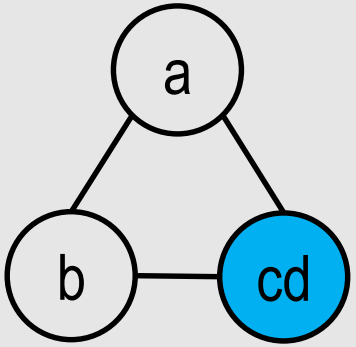
Safe coalescing heuristics: Briggs

Coalesce nodes that don't interfere, provided that the resulting merged node has less than K neighbors of degree ≥ K.

Example:



Don't merge c with d, since deg(a)=deg(b) = 2 in

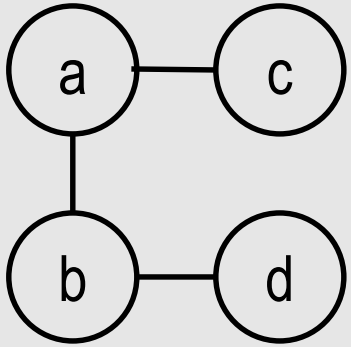


.

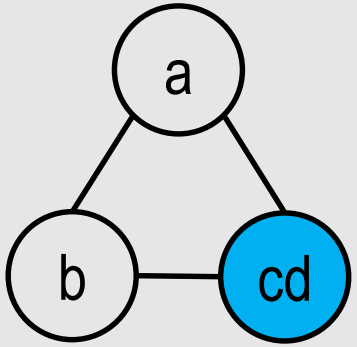
Safe coalescing heuristics: Briggs

Coalesce nodes that don't interfere, provided that the resulting merged node has less than K neighbors of degree ≥ K.

Example:



Don't merge c with d, since deg(a)=deg(b) = 2 in



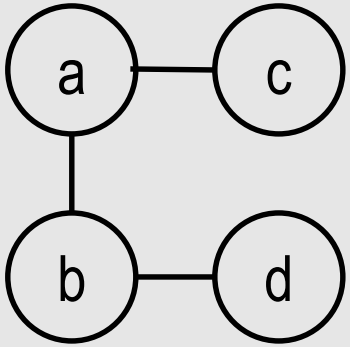
.

Why is this safe?

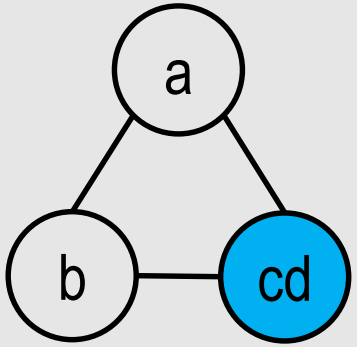
Safe coalescing heuristics: Briggs

Coalesce nodes that don't interfere, provided that the resulting merged node has less than K neighbors of degree ≥ K.

Example:



Don't merge c with d, since deg(a)=deg(b) = 2 in



.

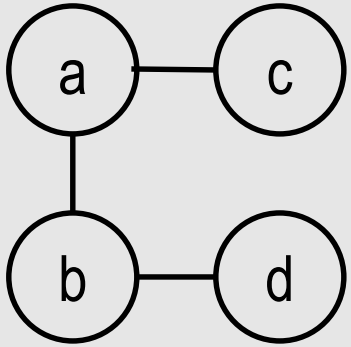
Why is this safe?

- after simplification, all nodes of degree < K have been eliminated

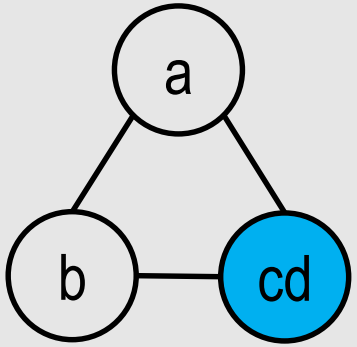
Safe coalescing heuristics: Briggs

Coalesce nodes that don't interfere, provided that the resulting merged node has less than K neighbors of degree ≥ K.

Example:



Don't merge c with d, since deg(a)=deg(b) = 2 in



.

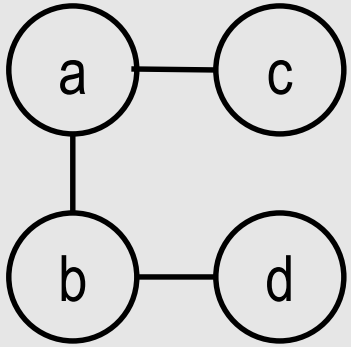
Why is this safe?

- after simplification, all nodes of degree < K have been eliminated
- so only high-degree neighbors of merge remain

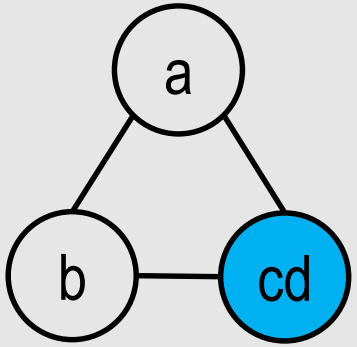
Safe coalescing heuristics: Briggs

Coalesce nodes that don't interfere, provided that the resulting merged node has less than K neighbors of degree ≥ K.

Example:



Don't merge c with d, since deg(a)=deg(b) = 2 in



.

Why is this safe?

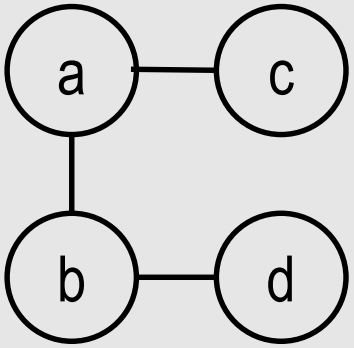
- after simplification, all nodes of degree < K have been eliminated
- so only high-degree neighbors of merge remain
- if there are < K of such neighbors, the degree of the merge is < K, so we can simplify merge

Hence, merging does not render a colorable graph incolorable.

Safe coalescing heuristics: George

Coalesce noninterfering nodes x and y only if every neighbor t of x already interferes with y or is of degree < K.

Example:

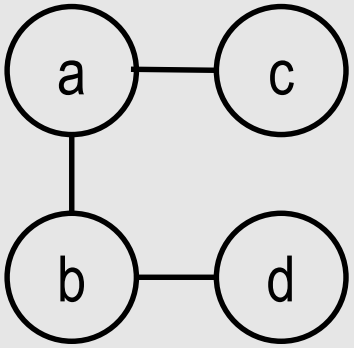


Don't merge c with d, since deg(a)= 2 and a does not yet interfere with d.
Similarly, don't merge d with c, since . . .

Safe coalescing heuristics: George

Coalesce noninterfering nodes x and y only if every neighbor t of x already interferes with y or is of degree < K.

Example:



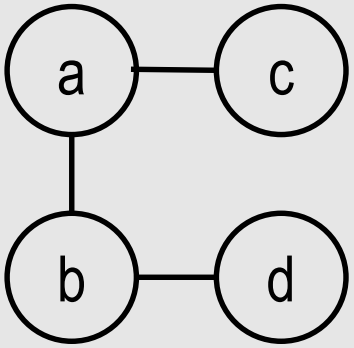
Don't merge c with d, since deg(a)= 2 and a does not yet interfere with d.
Similarly, don't merge d with c, since . . .

Why is this safe?

Safe coalescing heuristics: George

Coalesce noninterfering nodes x and y only if every neighbor t of x already interferes with y or is of degree < K.

Example:



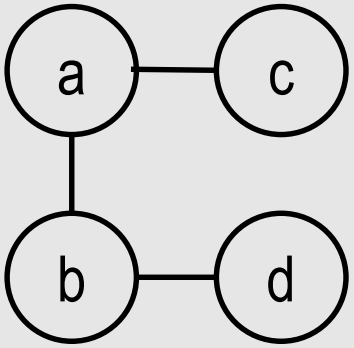
Don't merge c with d, since deg(a)= 2 and a does not yet interfere with d.
Similarly, don't merge d with c, since . . .

Why is this safe? • let S be the set of neighbors of x in G that have degree < K

Safe coalescing heuristics: George

Coalesce noninterfering nodes x and y only if every neighbor t of x already interferes with y or is of degree < K.

Example:



Don't merge c with d, since deg(a)= 2 and a does not yet interfere with d.
Similarly, don't merge d with c, since . . .

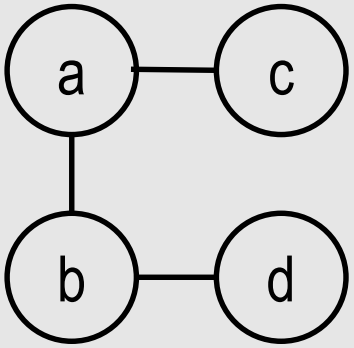
Why is this safe?

- let S be the set of neighbors of x in G that have degree < K
- if coalescing is **not** performed, all nodes in S simplify, leaving a reduced graph G_1

Safe coalescing heuristics: George

Coalesce noninterfering nodes x and y only if every neighbor t of x already interferes with y or is of degree < K.

Example:



Don't merge c with d, since deg(a)= 2 and a does not yet interfere with d.
Similarly, don't merge d with c, since . . .

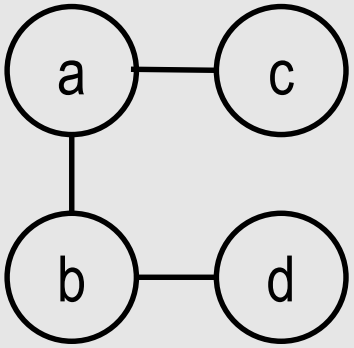
Why is this safe?

- let S be the set of neighbors of x in G that have degree < K
- if coalescing is **not** performed, all nodes in S simplify, leaving a reduced graph G_1
- if coalescing **is** performed, simplify also removes all nodes in S: each s ϵ S is of degree < K or is already adjacent to both x and y in G, so still simplifies after merging of x and y

Safe coalescing heuristics: George

Coalesce noninterfering nodes x and y only if every neighbor t of x already interferes with y or is of degree < K.

Example:



Don't merge c with d, since deg(a)= 2 and a does not yet interfere with d.
Similarly, don't merge d with c, since . . .

Why is this safe?

- let S be the set of neighbors of x in G that have degree < K
- if coalescing is **not** performed, all nodes in S simplify, leaving a reduced graph G_1
- if coalescing **is** performed, simplify also removes all nodes in S: each s ϵ S is of degree < K or is already adjacent to both x and y in G, so still simplifies after merging of x and y
- the resulting G_2 is a subgraph of G_1 ("merge" in G_2 corresponds to y in G_1), so if G_1 can be colored, so can G_2

Again, merging does not render a colorable graph incolorable.

Safe coalescing heuristics: Briggs, George

Both heuristics are conservative:

- we may miss some opportunities to coalesce (HW: example?)
- specifically, we may fail to eliminate some move instructions
- but that's preferable to not coalescing at all, which results in more spills; spills significantly more expensive (time: load+store versus move; space)

Safe coalescing heuristics: Briggs, George

Both heuristics are conservative:

- we may miss some opportunities to coalesce (HW: example?)
- specifically, we may fail to eliminate some move instructions
- but that's preferable to not coalescing at all, which results in more spills; spills significantly more expensive (time: load+store versus move; space)

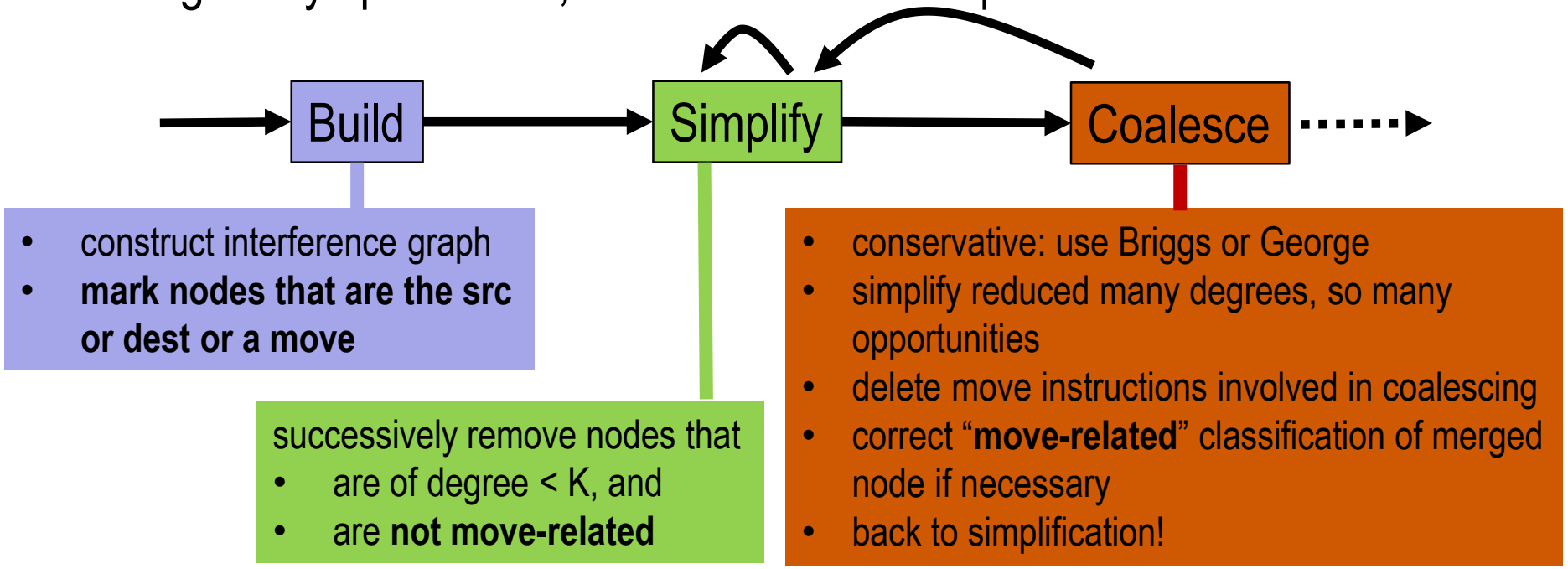
→ interleaving simplify with coalescing eliminates many moves, while still avoiding many spills. Thus, refine our allocation procedure:

Safe coalescing heuristics: Briggs, George

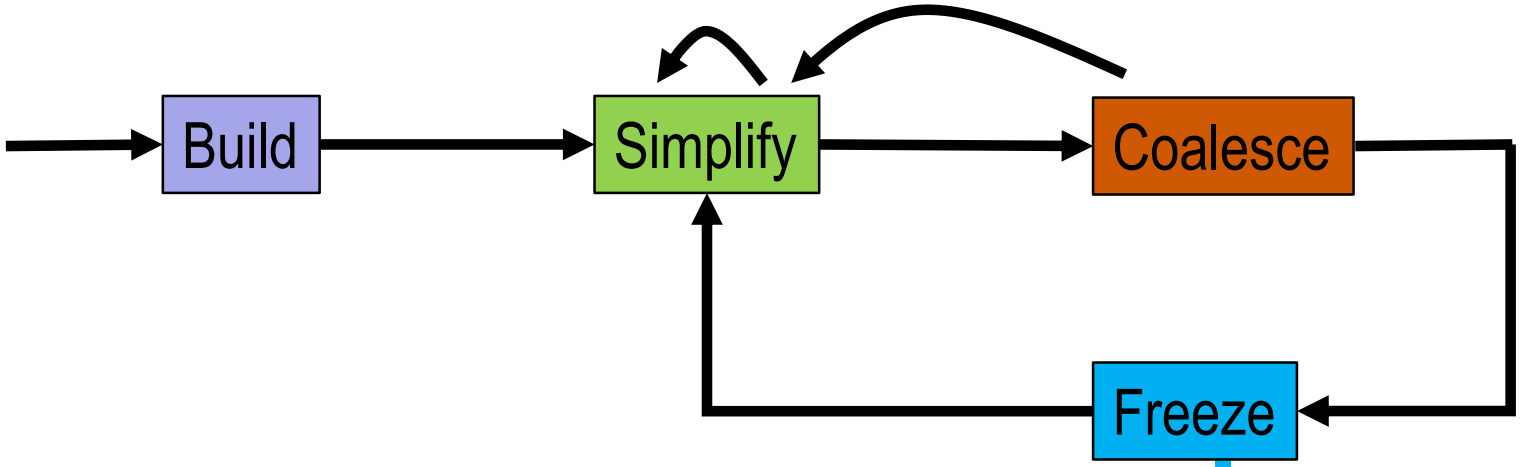
Both heuristics are conservative:

- we may miss some opportunities to coalesce (HW: example?)
- specifically, we may fail to eliminate some move instructions
- but that's preferable to not coalescing at all, which results in more spills; spills significantly more expensive (time: load+store versus move; space)

→ interleaving simplify with coalescing eliminates many moves, while still avoiding many spills. Thus, refine our allocation procedure:



Allocation with coalescing: freezing

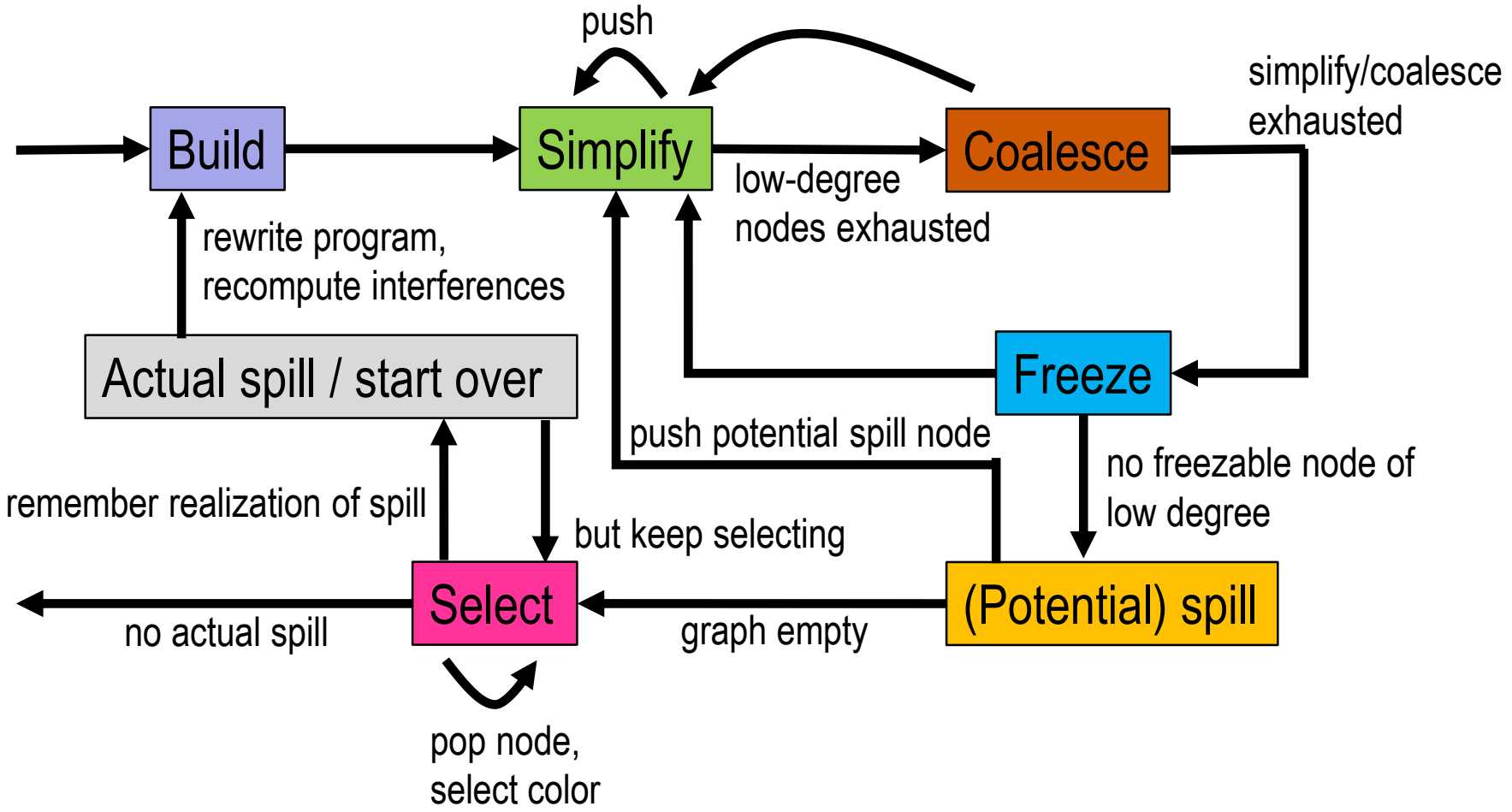


NEW PHASE:

- select a low-degree node **n** that is marked **move-related**
- mark it **non-move-related**
  - "give up hope to ever coalesce it"
  - also mark **n**'s move-partner **non-move-related**, unless it participates in some other move(s)
- back to simplify: at least the now unmarked nodes can be simplified

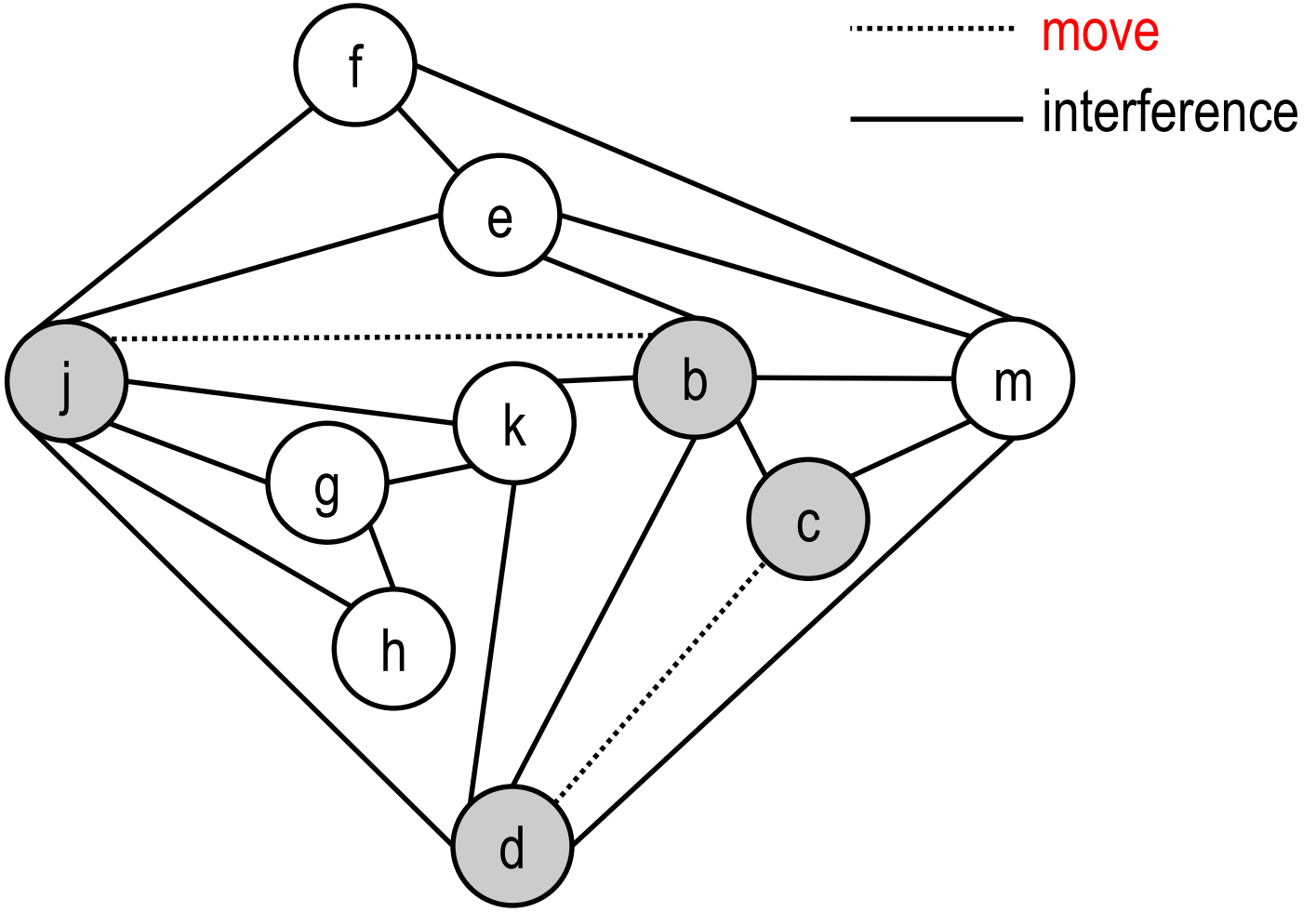
Allocation with coalescing: completing the algorithm

Remaining phases as before:



Coloring with coalescing: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```

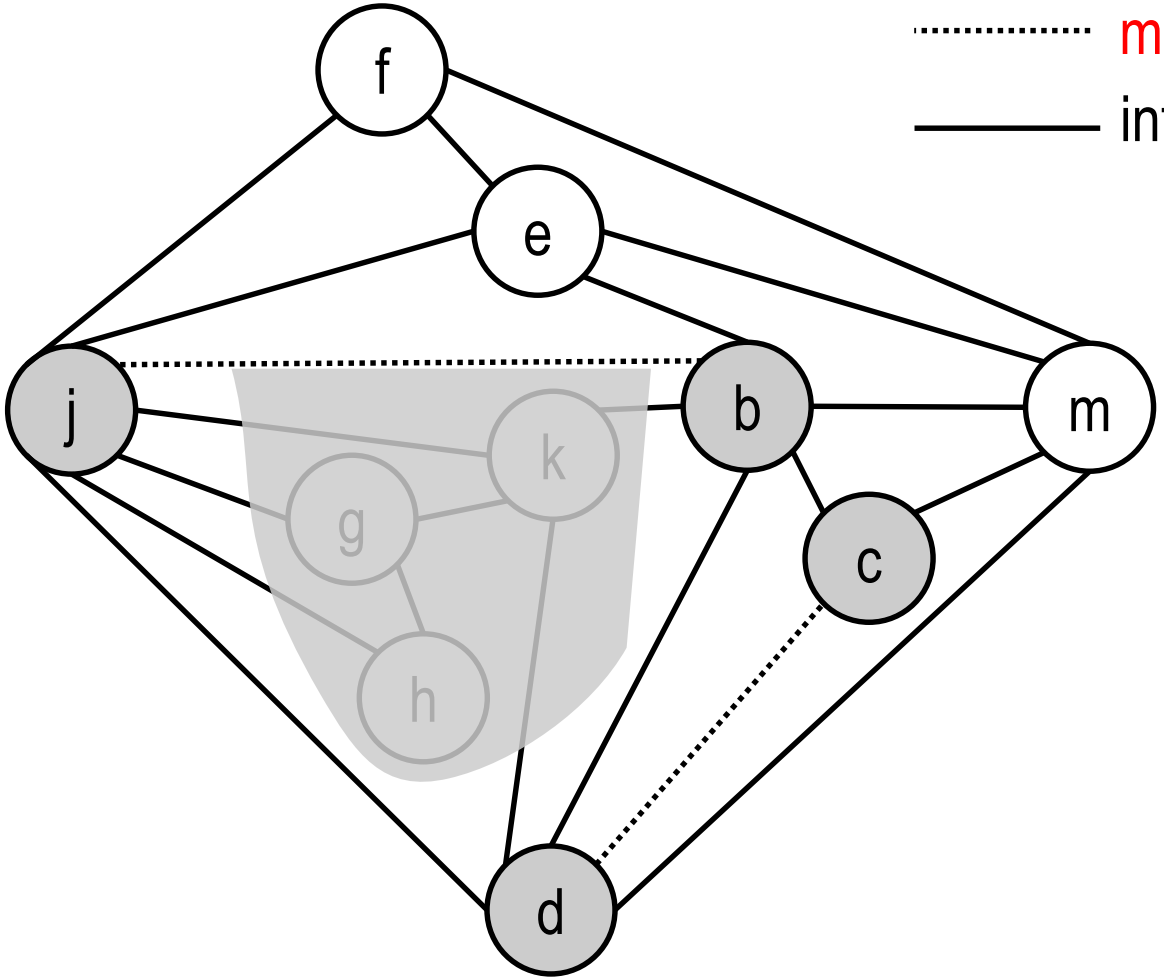


Non-marked nodes of degree < K: g, h, f

→ push g, h, k

Coloring with coalescing: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



Non-marked nodes of degree < K: f

→ push g, h, k

could still simplify f instead!

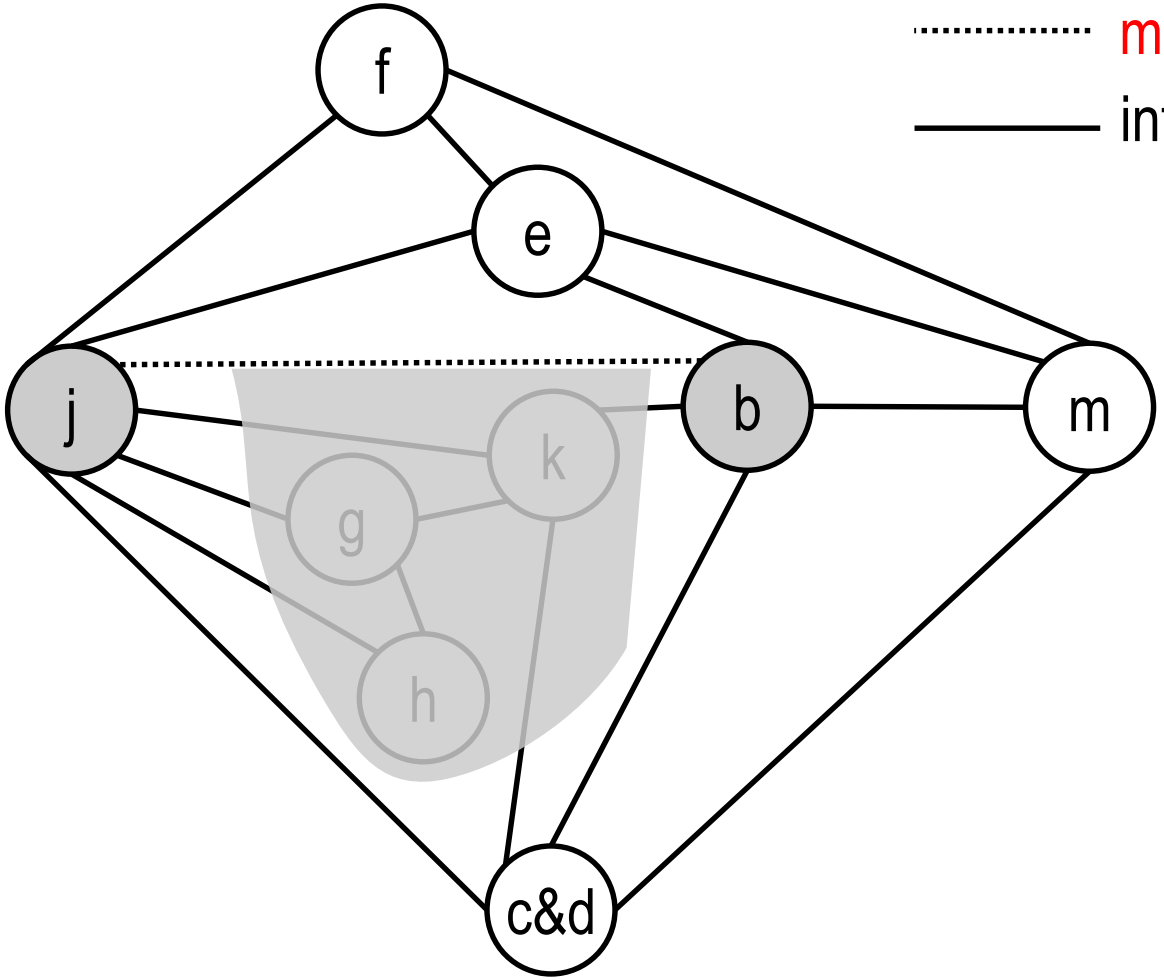
Next: **coalesce** c & d

George: all neighbors of c already interfere with d

Briggs: merged node has < K neighbors of degree ≥ K

Coloring with coalescing: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



could still simplify f instead!

Non-marked nodes of degree < K: f

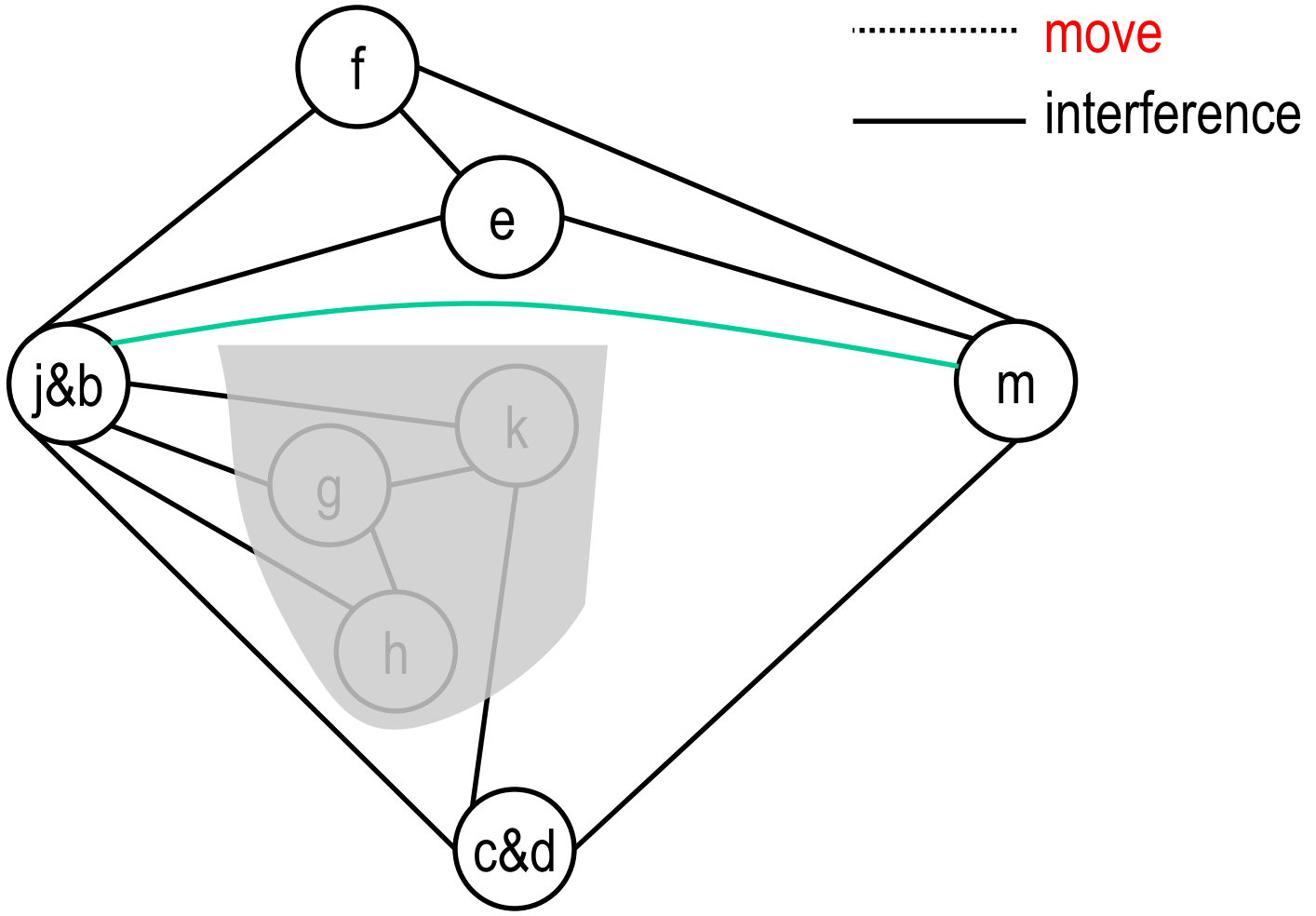
→ push g, h, k

Next: **coalesce** j & b

Briggs: merged node has < K neighbors of degree ≥ K

Coloring with coalescing: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



Non-marked nodes of degree < K: f, e, c&d

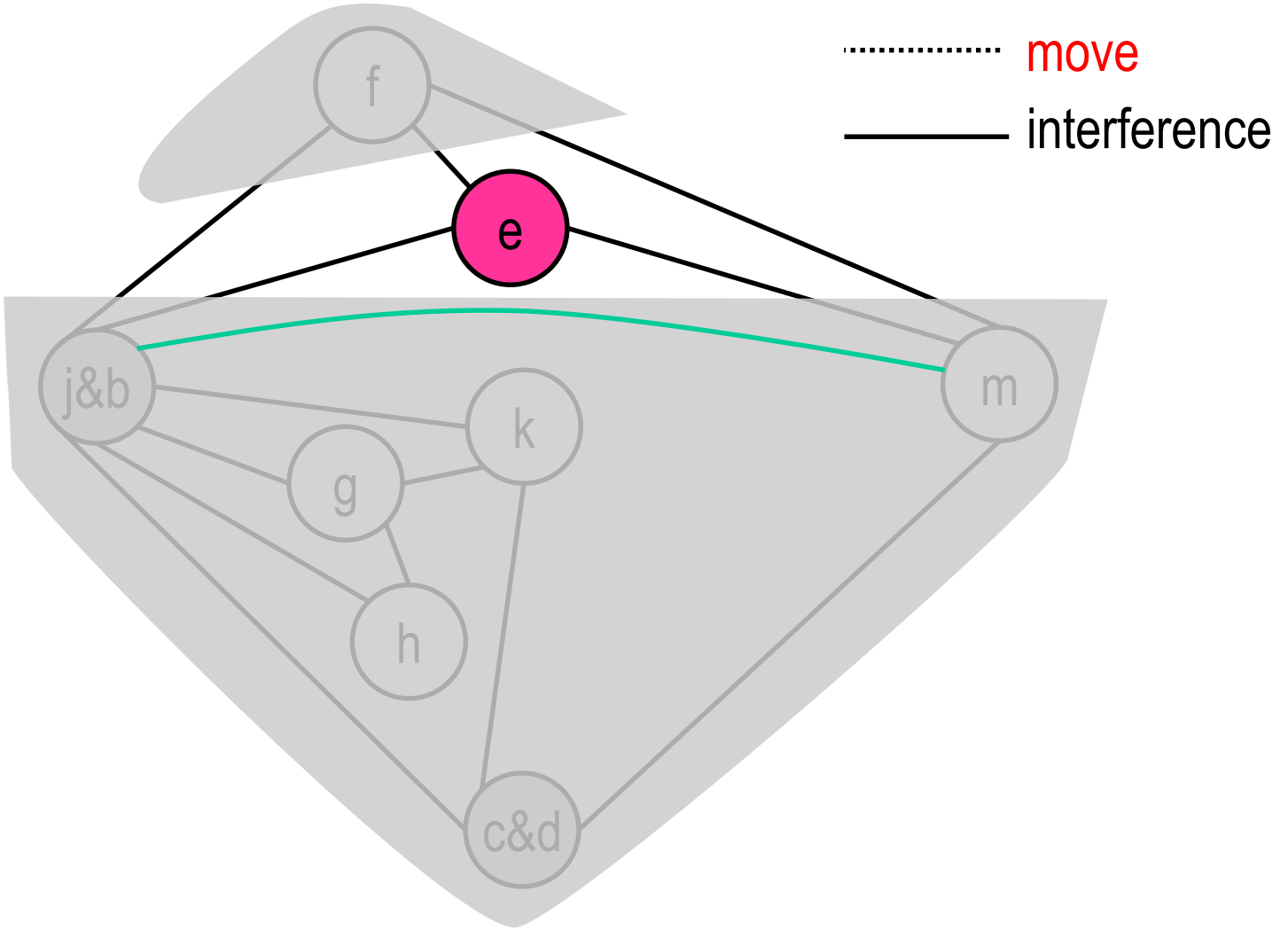
→ push g, h, k

Next: **push** c&d, j&b, f, m, e

pop e

Coloring with coalescing: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



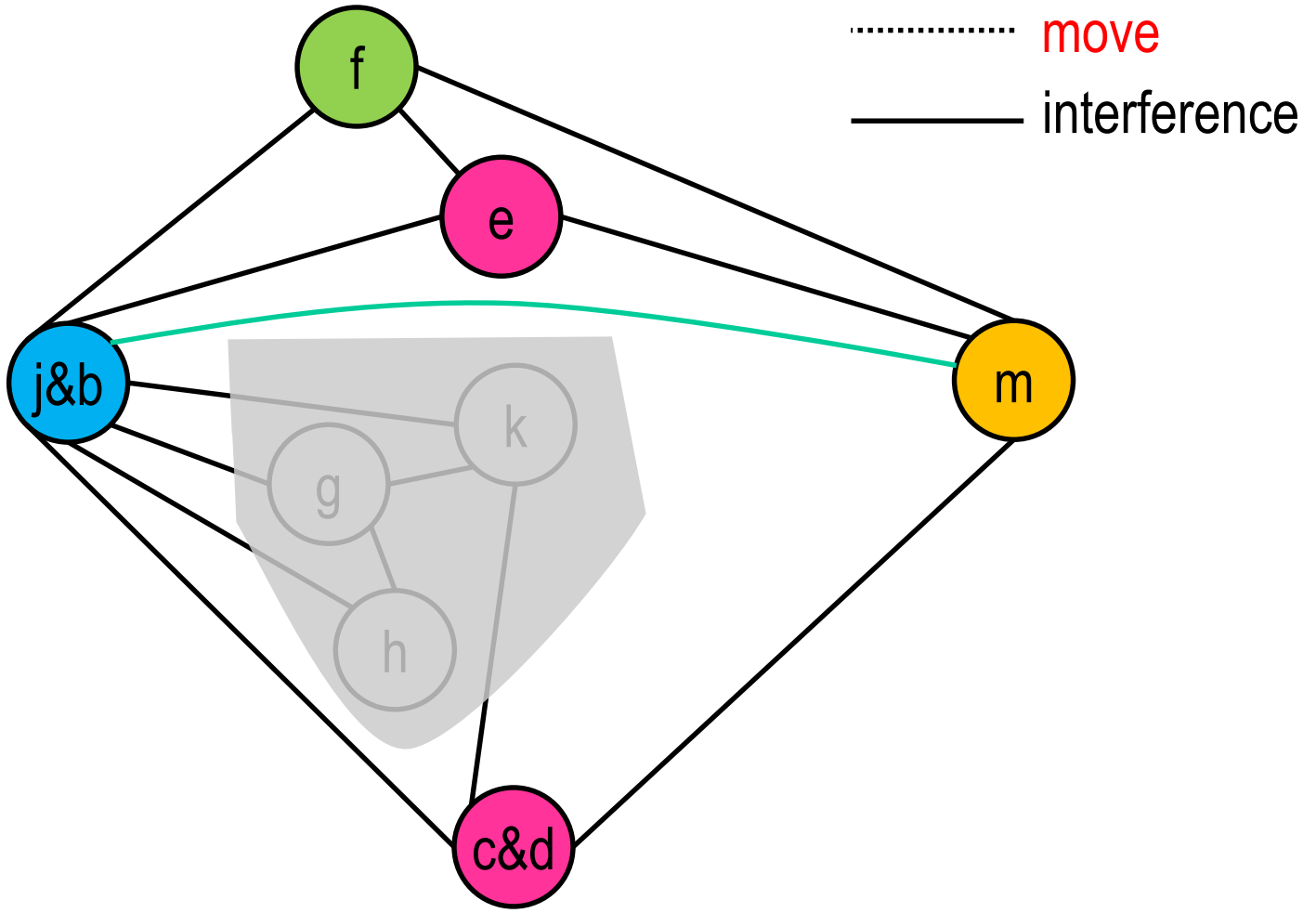
Non-marked nodes of degree < K:

→ push g, h, k, c&d, j&b, f, m

Next: **pop** m, f, j&b, c&d

Coloring with coalescing: example (K = 4)

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



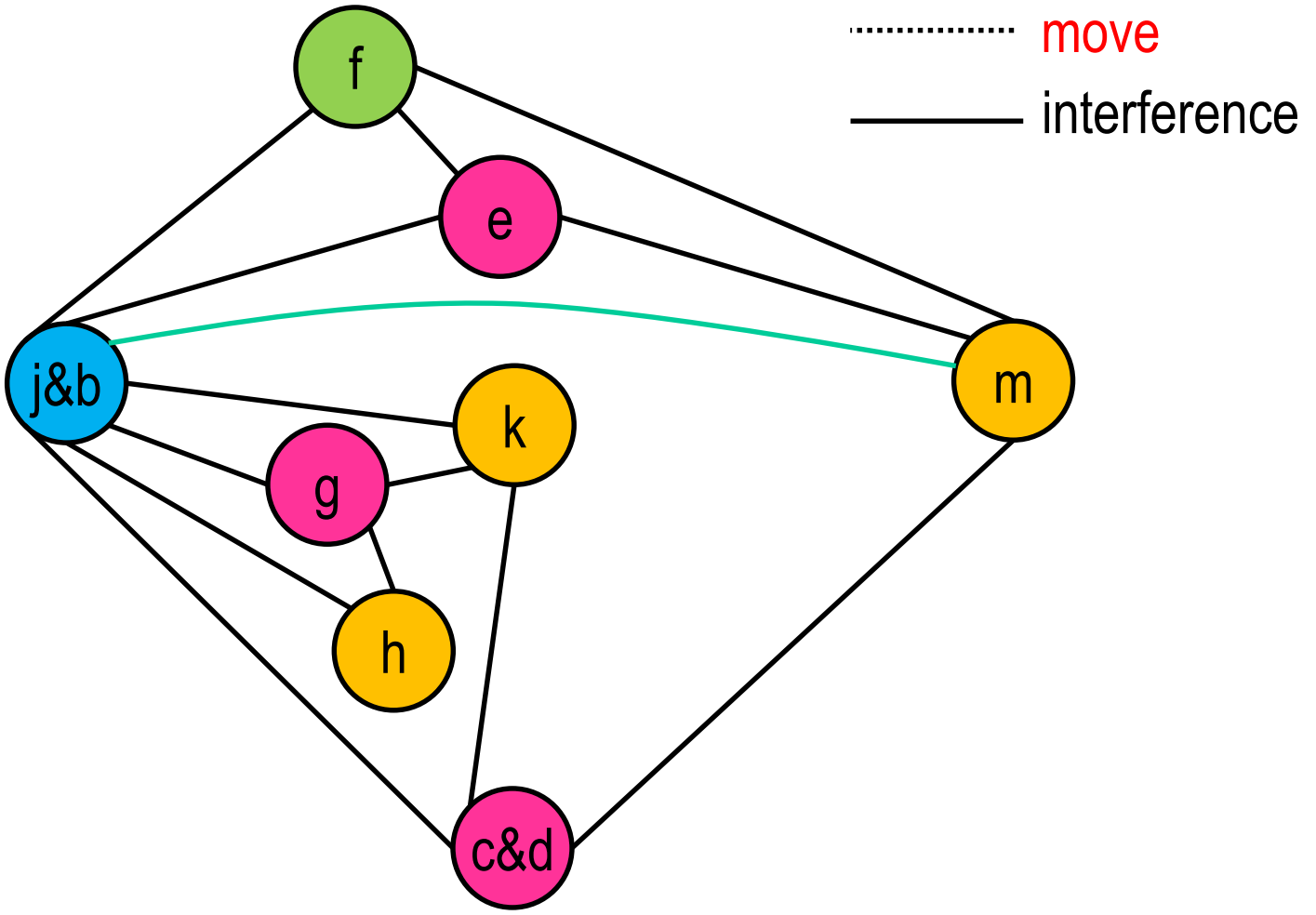
Non-marked nodes of degree < K:

→ push g, h, k

Next: **pop** k, h, g

Coloring with coalescing: example (K = 4)

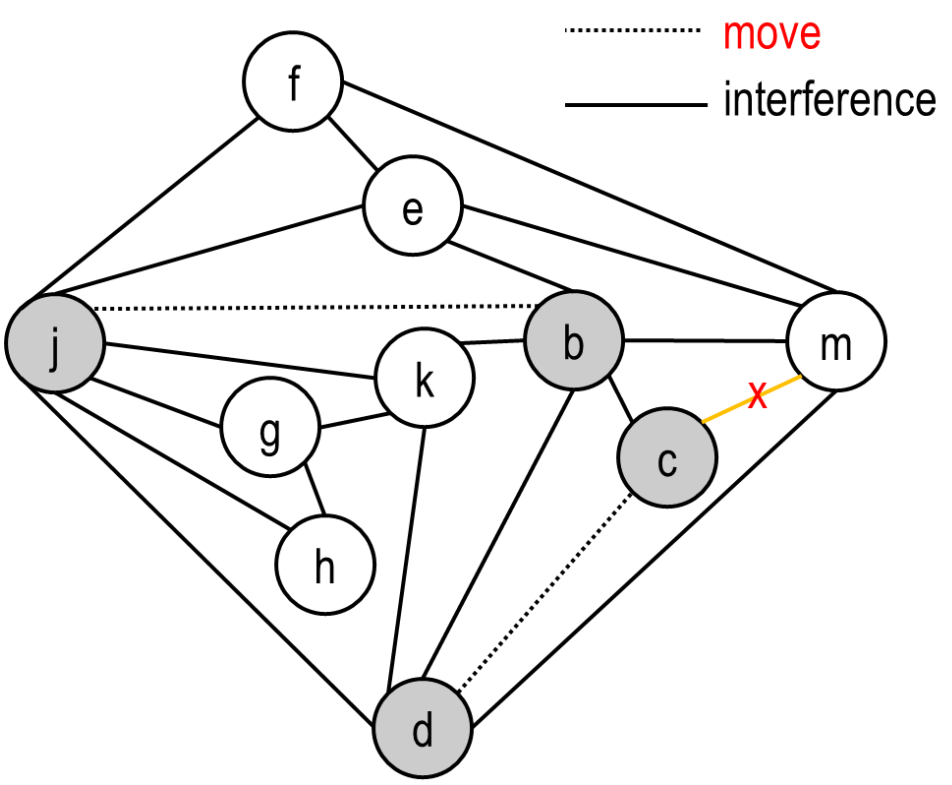
```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
b := M [ f ]
c := e + 8
d := c
k := m + 4
j := b
// liveOut d k j
```



Done

Spilling heuristics

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
M [ m_loc ] := m
b := M [ f ]
c := e + 8
d := c
m := M [ m_loc ]
k := m + 4
j := b
// liveOut d k j
```



SPILL m

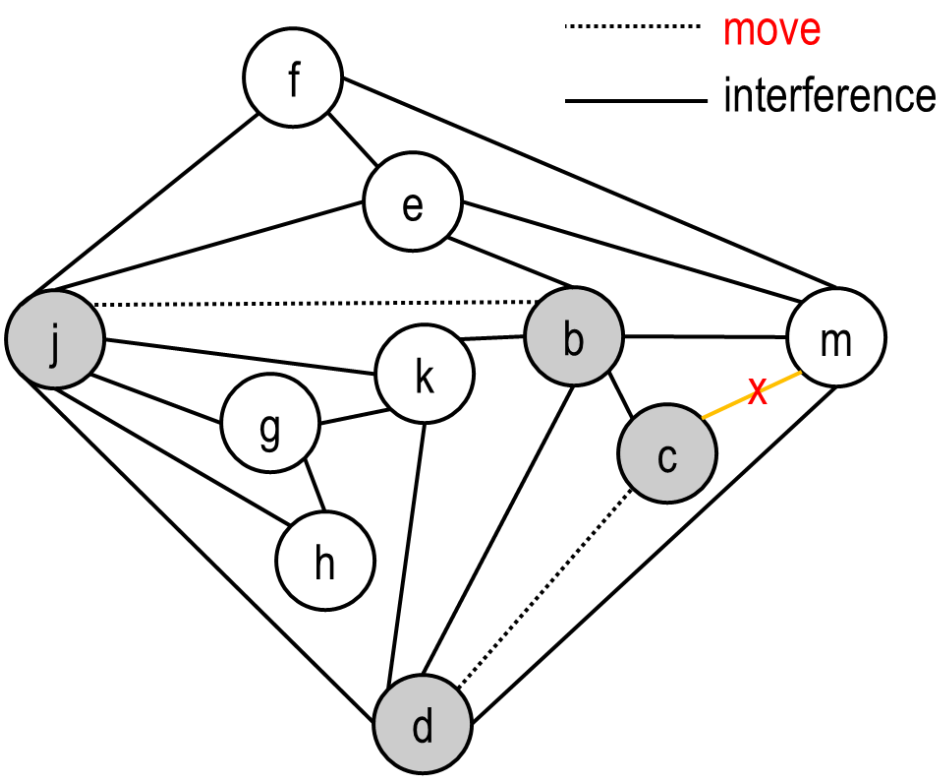
- splits single large liveness range of m into two short liveness ranges
- eliminates interference c ←→ m

General heuristics: spill nodes that

- have high degree, but few uses
- particularly if the live-range is long but sparse

Spilling heuristics

```
// liveIn: k, j
g := M [ j+12 ]
h := k – 1
f = g * h
e := M [ j + 8 ]
m := M [ j + 16 ]
M [ m_loc ] := m
b := M [ f ]
c := e + 8
d := c
m := M [ m_loc ]
k := m + 4
j := b
// liveOut d k j
```



SPILL m

- splits single large liveness range of m into two short liveness ranges
- eliminates interference c ←→ m

Spilling heuristics

Naïve spilling: when rewriting program, undo all register coalescing
Improvement: remember all coalescing done before the first potential spill was discovered – they will tend to be rediscovered -- but undo the later coalescings.

Spilling heuristics

Naïve spilling: when rewriting program, undo all register coalescing
Improvement: remember all coalescing done before the first potential spill was discovered – they will tend to be rediscovered -- but undo the later coalescings.

Coalescing spills:

- many spill locations → large stack frames
- don't need to keep spill locations apart if their virtual registers don't interfere!
- further benefit: eliminate spill-to-spill-moves:
  a ←b when both a and b are spilled:
  $t \leftarrow M[b_{loc}]; M[a_{loc}] \leftarrow t$ (typo in MCIL here – see errata list!)

Hence, can use coloring to minimize spill locations:

- infinitely many colors: no bound on size of frame
- liveness info yields interference between spilled nodes
- first, coalesce all spill nodes related by moves
- then, simplify and select (try to reuse colors)
- resulting # colors is # spill locations

All done during "Start Over", before spill code is generated and new register interference is computed

Precolored temporaries / nodes

- some temporaries correspond directly to machine registers: stack / frame pointer, standard argument registers 1 & 2, …
- these special temporaries implicitly interfere with each other
- but: ordinary temporaries **can** share color with precolored node (see example below)

Precolored temporaries / nodes

- some temporaries correspond directly to machine registers: stack / frame pointer, standard argument registers 1 & 2, …
- these special temporaries implicitly interfere with each other
- but: ordinary temporaries **can** share color with precolored node (see example below)

K-register machine:

- introduce precolored K nodes, all interfering with each other
- liveness range of special-purpose registers (frame pointer etc) interfere with all ordinary temporaries that are live
- general-purpose registers have no additional interferences
- precolored nodes can't be simplified (they already have a color!), and can't be spilled (they are registers!)
- hence, consider them to be of infinite degree and start selection phase not from empty graph but graph of precolored nodes
- to keep live ranges of precolored nodes short, front-end can "copy them away", to freshly introduced temps

“Copying away” precolored temporaries

- suppose register r7 is callee-save:
  - considering function entry as definition of r7, and function exit as use ensures it’s live throughout the body, so it will be preserved
  - but: we don’t want to block the callee-save-register color for the entire body

```
entry: def(r7)
         :
exit: use(r7)
```

“Copying away” precolored temporaries

- suppose register r7 is callee-save:
  - considering function entry as definition of r7, and function exit as use ensures it’s live throughout the body, so it will be preserved
  - but: we don’t want to block the callee-save-register color for the entire body
  - so: introduce a new temporary t and insert moves
  - if register pressure is low, allocator will coalesce and eliminate moves
  - if register pressure is high, allocator will spill

```
entry: def(r7)
        :
exit: use(r7)
```

```
entry: def(r7)
       t ← r7
        :
       r7 ← t
exit: use(r7)
```

“Copying away” precolored temporaries

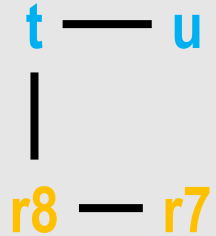
- suppose register r7 is callee-save:
  - considering function entry as definition of r7, and function exit as use ensures it’s live throughout the body, so it will be preserved
  - but: we don’t want to block the callee-save-register color for the entire body
  - so: introduce a new temporary t and insert moves
  - if register pressure is low, allocator will coalesce and eliminate moves
  - if register pressure is high, allocator will spill

```
entry: def(r7)
        :
exit: use(r7)
```

```
entry: def(r7)
       t ← r7
        :
       r7 ← t
exit: use(r7)
```

Note: the thus introduced temps t (one for each callee-save register) interfere with each other, with “later” other callee-save regs, and with most variables defined + used in the body, and are hence of “high degree and low #uses”.

```
entry: def(r7, r8)
       t ← r7
       u ← r8
        :
       r8 ← u
       r7 ← t
exit: use(r7, r8)
```



Liveness-across-call and caller/callee-save preference

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```

Temporary x is not live across the call to f

- allocating x to a callee-save register r will force body of f to store r away to some t (previous slide), and restore r before returning
- but caller does not need x

Liveness-across-call and caller/callee-save preference

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```

Temporary **x** is not live across the call to f

- allocating **x** to a callee-save register **r** will force body of f to store **r** away to some **t** (previous slide), and restore **r** before returning
- but caller does not need **x**
- prefer allocation of **x** to caller-save register **s**:
  - callee f is free to overwrite **s**
  - that's ok: **x** is not used after function return
  - caller even does not even need to store **s** away prior to call – and knows this (liveness info)

Liveness-across-call and caller/callee-save preference

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```

Temporary x is not live across the call to f

- allocating x to a callee-save register r will force body of f to store r away to some t (previous slide), and restore r before returning
- but caller does not need x
- prefer allocation of x to caller-save register s:
  - callee f is free to overwrite s
  - that's ok: x is not used after function return
  - caller even does not even need to store s away prior to call – and knows this (liveness info)

Temps not live across calls should be allocated to caller-save registers.

Liveness-across-call and caller/callee-save preference

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```

Temporary y is live across the call to f

- allocating y to a caller-save register s would mean that f is free to overwrite s
- but caller does need y/s after function return
- so y/s would additionally need to be spilled / copied away prior to call
- we don't want to spill all variables that are live across calls!

Liveness-across-call and caller/callee-save preference

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```

Temporary y is live across the call to f

- allocating y to a caller-save register s would mean that f is free to overwrite s
- but caller does need y/s after function return
- so y/s would additionally need to be spilled / copied away prior to call
- we don't want to spill all variables that are live across calls!
- prefer allocation of y to callee-save register r:
  - callee f copies r away to some t (coalesce if possible) and will restore r prior to return
  - no additional work needed on caller side

Liveness-across-call and caller/callee-save preference

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```

Temporary y is live across the call to f

- allocating y to a caller-save register s would mean that f is free to overwrite s
- but caller does need y/s after function return
- so y/s would additionally need to be spilled / copied away prior to call
- we don't want to spill all variables that are live across calls!
- prefer allocation of y to callee-save register r:
  - callee f copies r away to some t (coalesce if possible) and will restore r prior to return
  - no additional work needed on caller side

Temps live across calls should be allocated to callee-save registers.

Liveness-across-call and caller/callee-save preference

Temps not live across calls should be allocated to caller-save registers.

Temps live across calls should be allocated to callee-save registers.

How can we nudge the allocator to do this?

```
Body of g():
    :
x := 5
y := x + 1
z := f ()
return z + y
```

Liveness-across-call and caller/callee-save preference

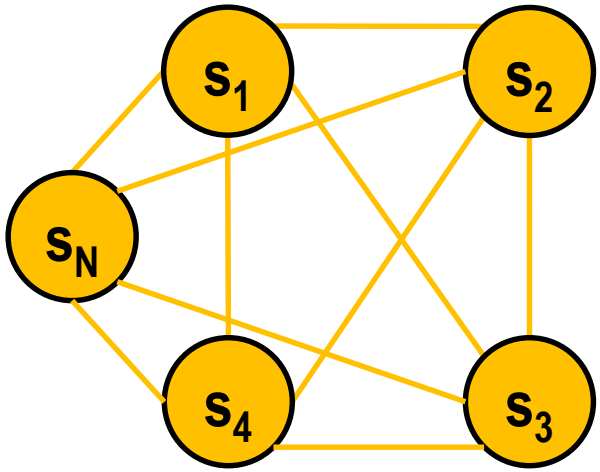
Temps not live across calls should be allocated to caller-save registers.

Temps live across calls should be allocated to callee-save registers.

How can we nudge the allocator to do this?

In **CALL** instruction, understand all **N caller-save registers** to be defined/live-out. They interfere with **each other**

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```



Liveness-across-call and caller/callee-save preference

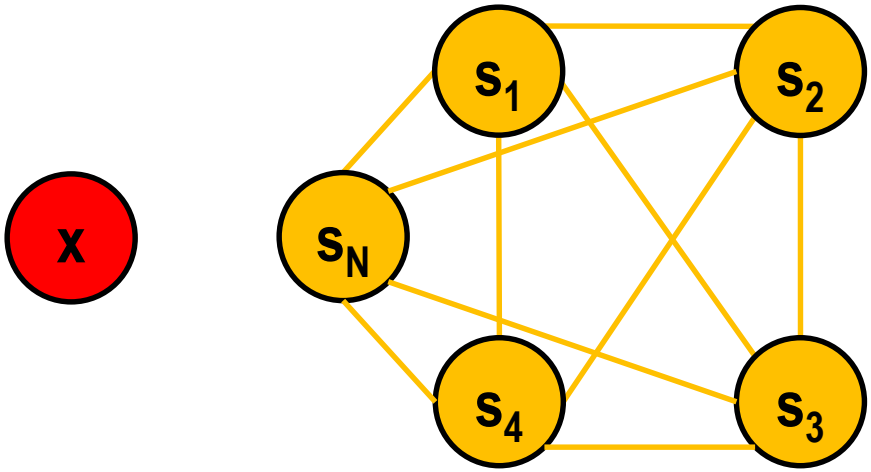
Temps not live across calls should be allocated to caller-save registers.

Temps live across calls should be allocated to callee-save registers.

How can we nudge the allocator to do this?

In **CALL** instruction, understand all **N caller-save registers** to be defined/live-out. They interfere with **each other but not with x**, so a good allocator will tend to assign **x** to the precolor of one of the s_i.

```
Body of g():
    :
  x := 5
  y := x + 1
  z := f ()
  return z + y
```



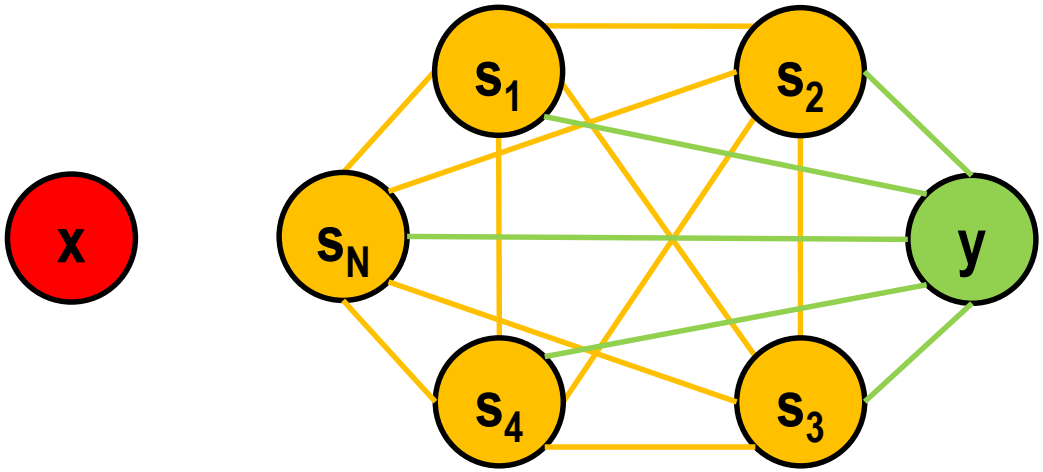
Liveness-across-call and caller/callee-save preference

Temps live across calls should be allocated to callee-save registers.

How can we nudge the allocator to do this?

In **CALL** instruction, understand all **N caller-save registers** to be defined/live-out. They interfere with **each other and also with y.**

```
Body of g():
    :
x := 5
y := x + 1
z := f ()
return z + y
```



Liveness-across-call and caller/callee-save preference

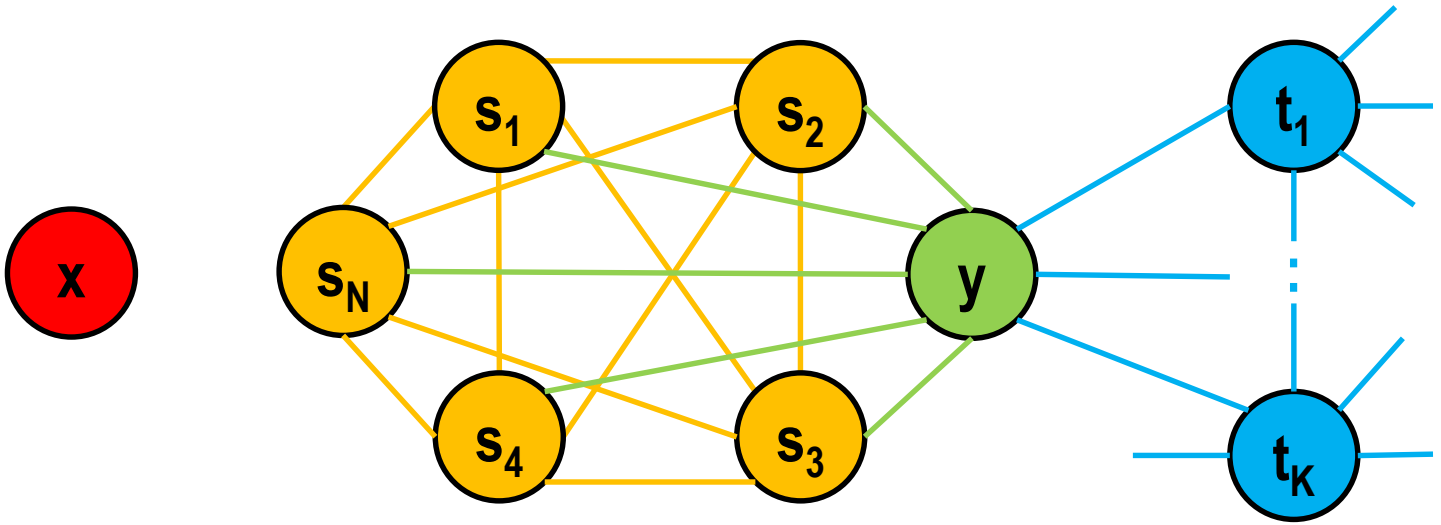
Temps live across calls should be allocated to callee-save registers.

How can we nudge the allocator to do this?

In **CALL** instruction, understand all **N caller-save registers** to be defined/live-out. They interfere with **each other and also with y.** But **y** also interferes with the t_i created by the front-end in the body of g.

Body of g():

```
    :
x := 5
y := x + 1
z := f ()
return z + y
```



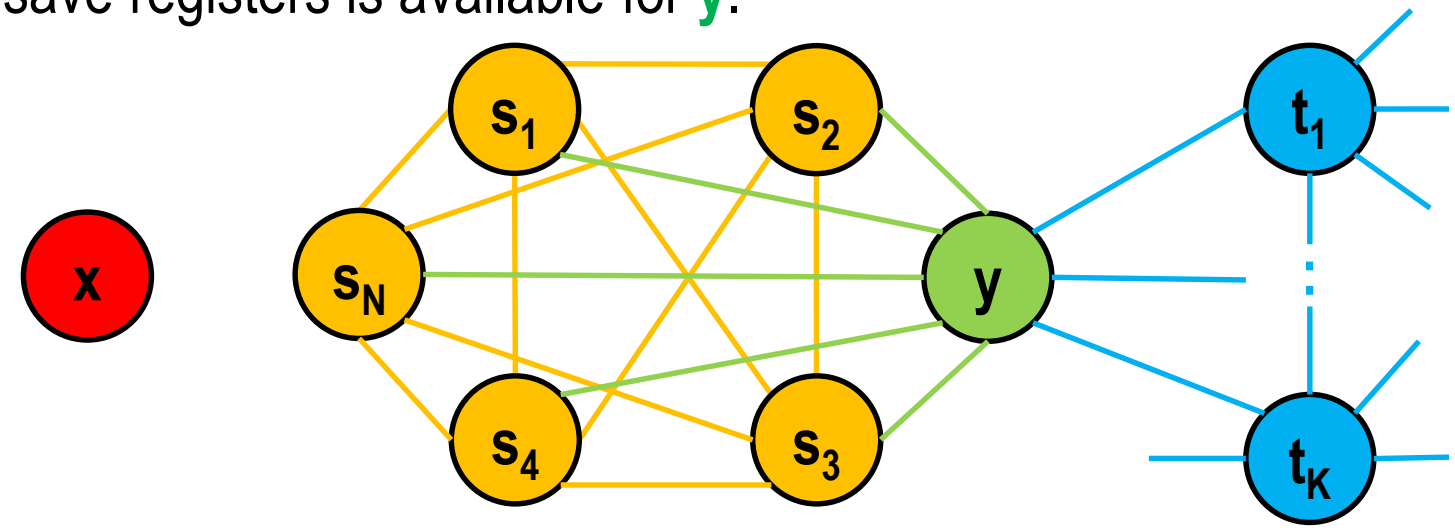
Liveness-across-call and caller/callee-save preference

Temps live across calls should be allocated to callee-save registers.

How can we nudge the allocator to do this?

In **CALL** instruction, understand all **N caller-save registers** to be defined/live-out. They interfere with **each other and also with y.** But **y** also interferes with the t_i created by the front-end in the body of g. So a spill is likely. Since the t_i are "high degree, **low use**", they are more likely to be selected for spill. So, the color of one callee-save registers is available for **y**.

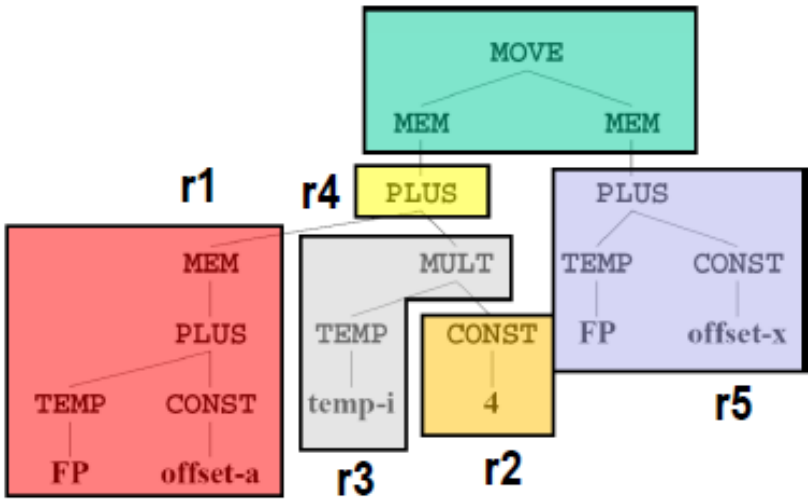
```
Body of g():
    :
x := 5
y := x + 1
z := f ()
return z + y
```



Register allocation for expression trees

Can avoid liveness calculation, interference graph construction, coloring.

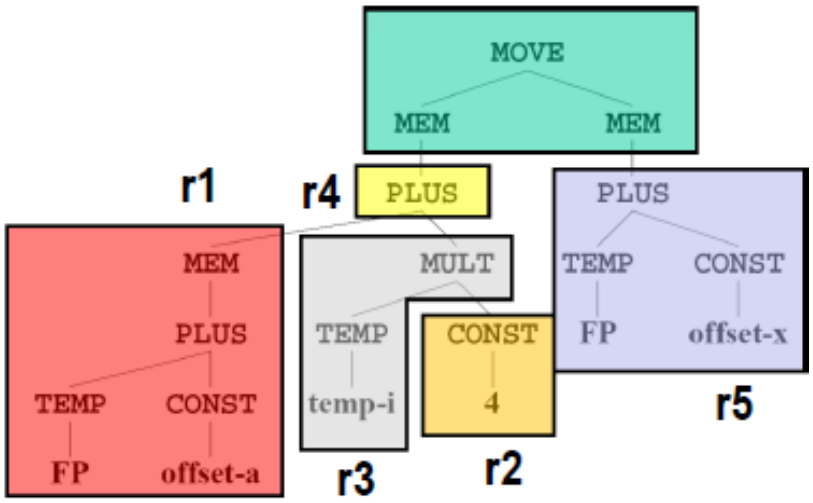
Flashback to instruction selection: “tiling”, ie covering the tree with patterns corresponding to machine instructions.



In IR phase, had suggested use of separate (virtual) registers for each tile.

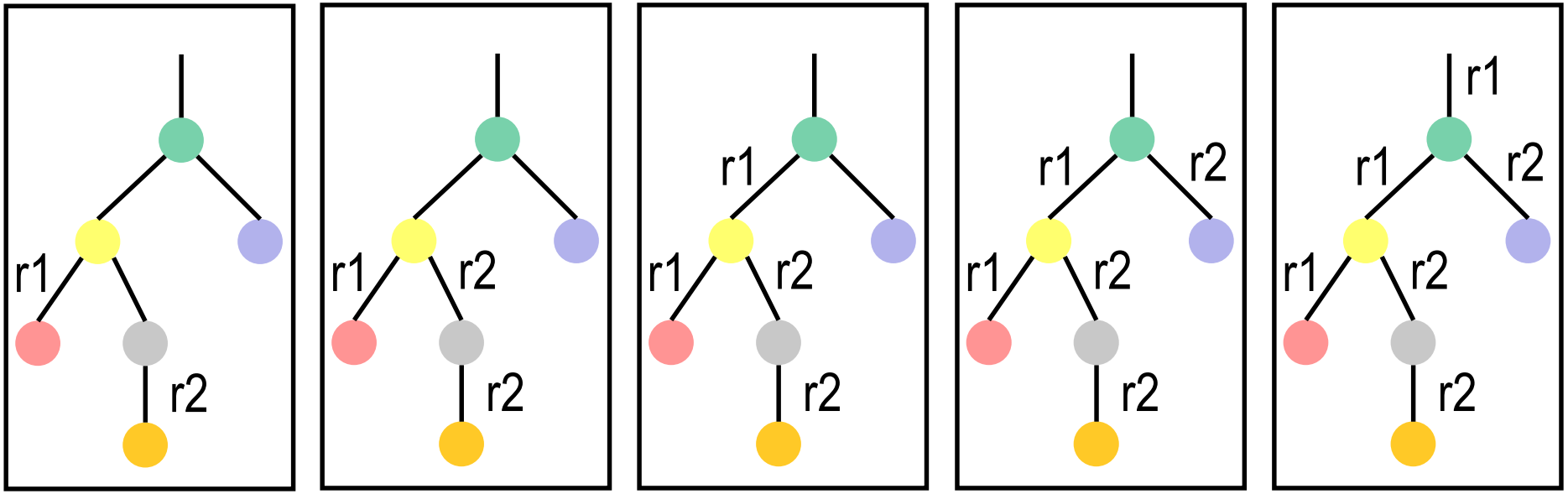
Clearly, can do better…

Register allocation for expression trees

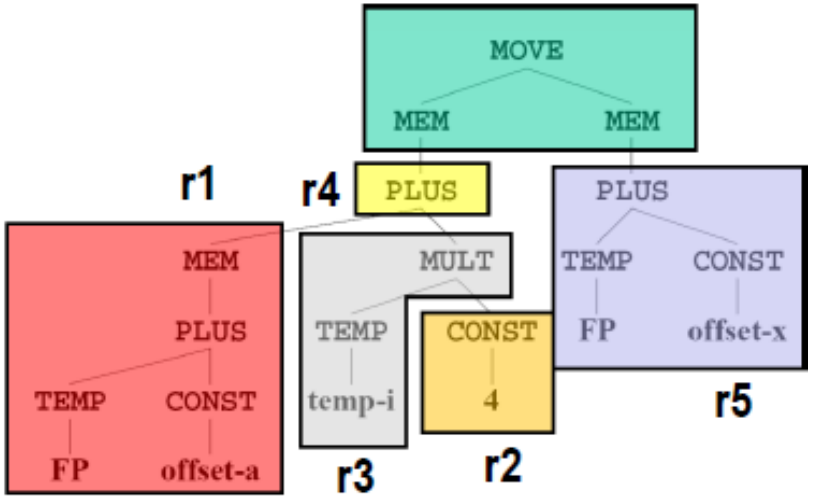


Algorithm 1:

- simple postorder traversal
- can be combined with maximal munch (optimal but not optimum)



Register allocation for expression trees



Algorithm 2:

- dynamic programming
- label each tile with number of registers needed for its evaluation
- when visiting node u with children u_{left} u_{right}, with needs n_l and n_r, respectively:
  - evaluating left child; hold result while evaluating right child: cost = $\max(u_{left}, 1 + u_{right})$
  - evaluating right child; hold result while evaluating left child: cost = $\max(1 + u_{left}, u_{right})$
  - choose cheaper evaluation order