Compilation 2024

Register Allocation

Aslan Askarov

Based on slides by E. Ernst

Register Allocation

- Recall: Interference graph
  - Node: temporary
  - Edge: interference (cannot unify end points)
  - Undirected
- Unification of temporaries: graph coloring
  - Neighbors ⇒ different colors
  - *K*-coloring of interference graph = register allocation
  - If no *K*-coloring exists: spill and repeat
- Basic parameter: We have *K* registers
- Useful concept: *Significant degree*: degree(*n*) ≥ *K*
- Convenient words for it: A *heavy node* (vs *light*)

Graph Coloring

- The basic problem is NP-complete (polynomial to verify, exponential to compute)

- We are lucky: Good approximation linear!

- Algorithm:
  - build interference graph G
  - simplify G
  - spill some nodes
  - select colors

Graph Coloring: Build

- Build the interference graph
- Recall how:
  - build data flow graph
  - compute use/def locally, then live-in/live-out via iteration
  - create interference graph node per temp, edge per pair of temps with overlapping live ranges

Graph Coloring: Simplify

- Reducing the graph G, preserving colorability
- Algorithm:
  - repeat { find light node n; remove n from G; push n }
- For each step we have: Graph K-colorable *after* removal $\Rightarrow$ also K-colorable before removal
- Reason: Node n is light, i.e., at most K-1 colors used
- Stopping: Every node is heavy

Graph Coloring: Spill

- Remove one node from the graph G, marking it as a ‘potential spill’

- Algorithm:
  - find heavy node n; remove n; mark n ‘spill’; push n

- Got here because Simplify stopped
  - If G non-empty: all nodes heavy, proceed, go to Simplify
  - If G empty: go to Select

Graph Coloring: Select

- Pop and re-insert all nodes from the stack
- Algorithm:
  - repeat { n=pop; add n with edges to G; color n }
- Cases
  - n was light: reinsert/color always works
  - n was heavy (marked ‘spill’): go ahead and try! ;-)
    - it worked — continue
    - it failed — insert w/o color, continue, noting failure
- Stopping: stack empty

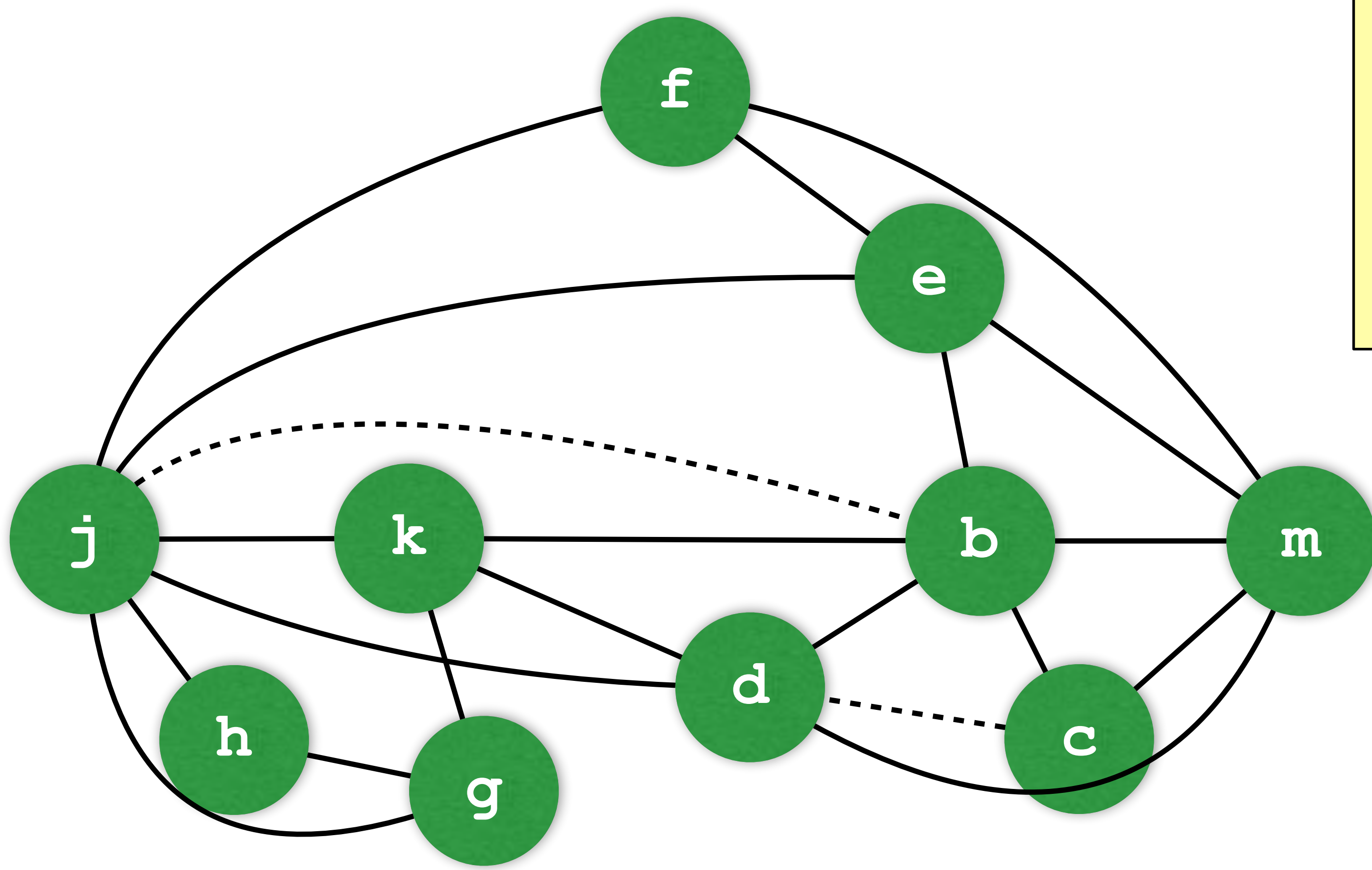
Graph Coloring: Start over

- Perform spills, if any
- Algorithm:
  - rewrite program: add load/store of temp at use/def, using new, short-lived temporaries
- Changed program ⇒ recompute all (go to build)
- Note: entire algorithm typically repeats only 1-2 times

Graph Coloring Example

- Example program, similar to 3-address assembly
- *K* = 4
- Build interference graph

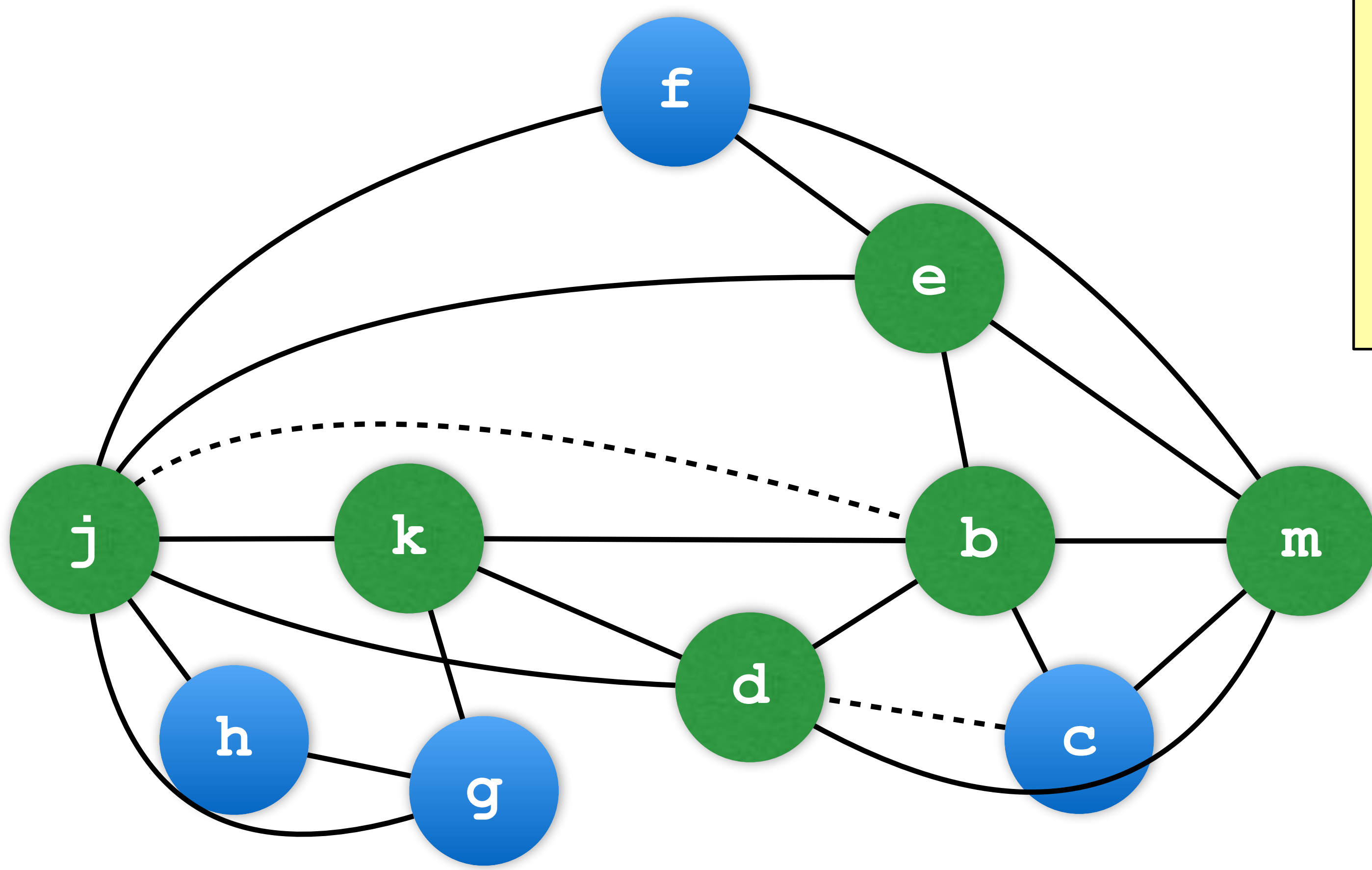
```
live-in: k j
        g := mem[j+12]
        h := k - 1
        f := g * h
        e := mem[j+8]
        m := mem[j+16]
        b := mem[f]
        c := e + 8
        d := c
        k := m + 4
        j := b
live-out: d k j
```



Graph Coloring Example

- Simplify:
  - g, h, c, f are *light*, less than *K* neighbors
  - choose g, h, then k for removal

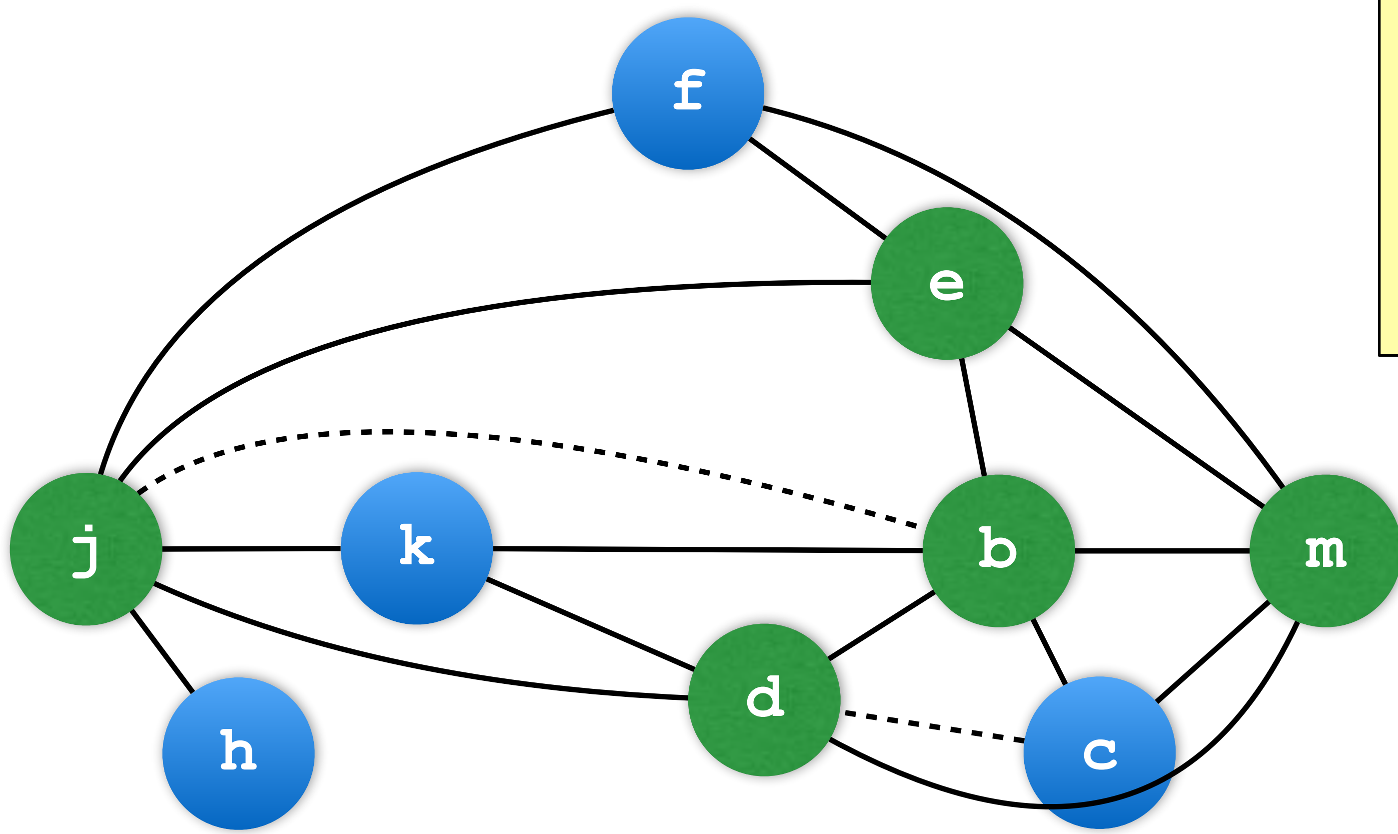
```
live-in: k j
        g := mem[j+12]
        h := k - 1
        f := g * h
        e := mem[j+8]
        m := mem[j+16]
        b := mem[f]
        c := e + 8
        d := c
        k := m + 4
        j := b
live-out: d k j
```



Graph Coloring Example

- Simplify:
  - g, h, c, f are *light*, less than *K* neighbors
  - choose g, h, then k for removal

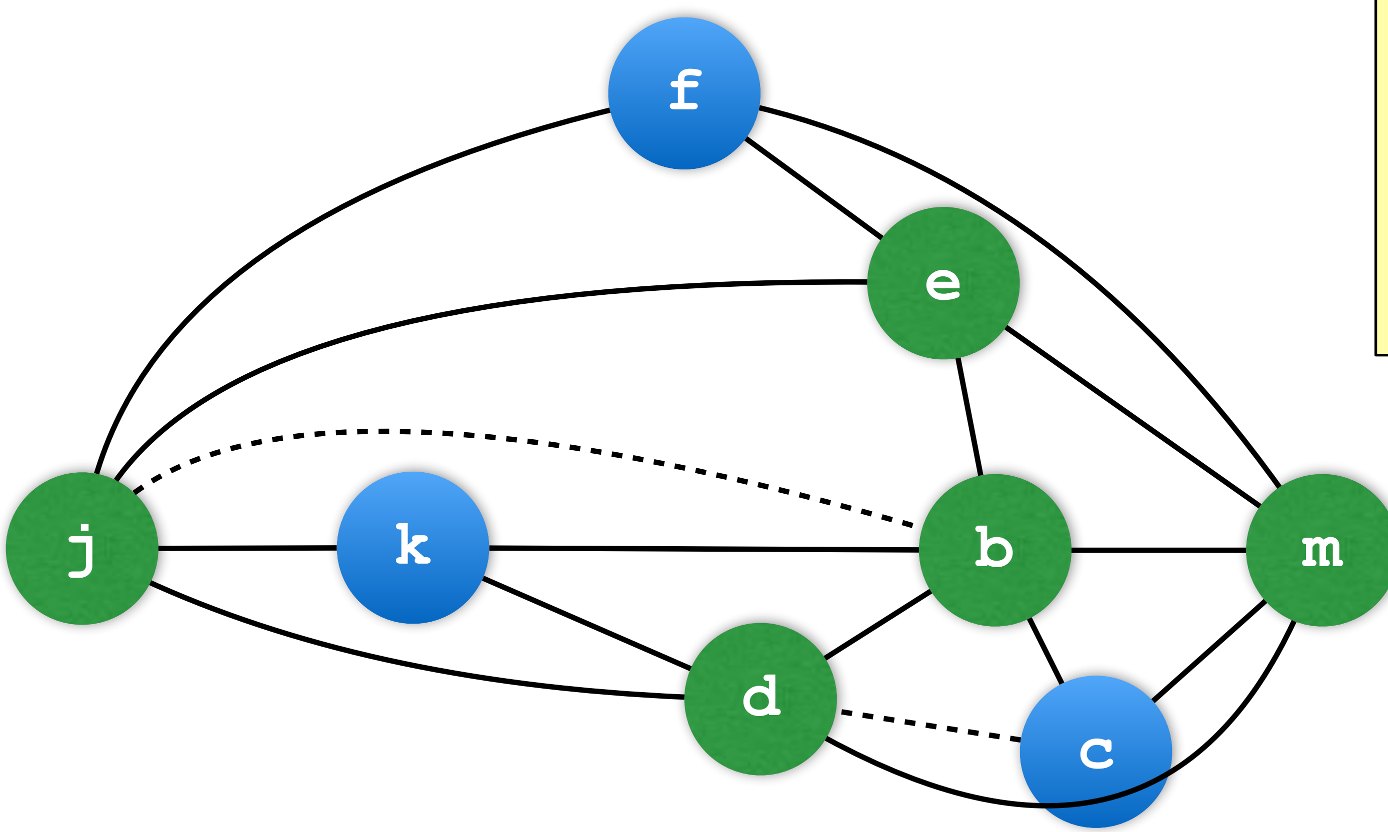
```
live-in: k j
      g := mem[j+12]
      h := k - 1
      f := g * h
      e := mem[j+8]
      m := mem[j+16]
      b := mem[f]
      c := e + 8
      d := c
      k := m + 4
      j := b
live-out: d k j
```



Graph Coloring Example

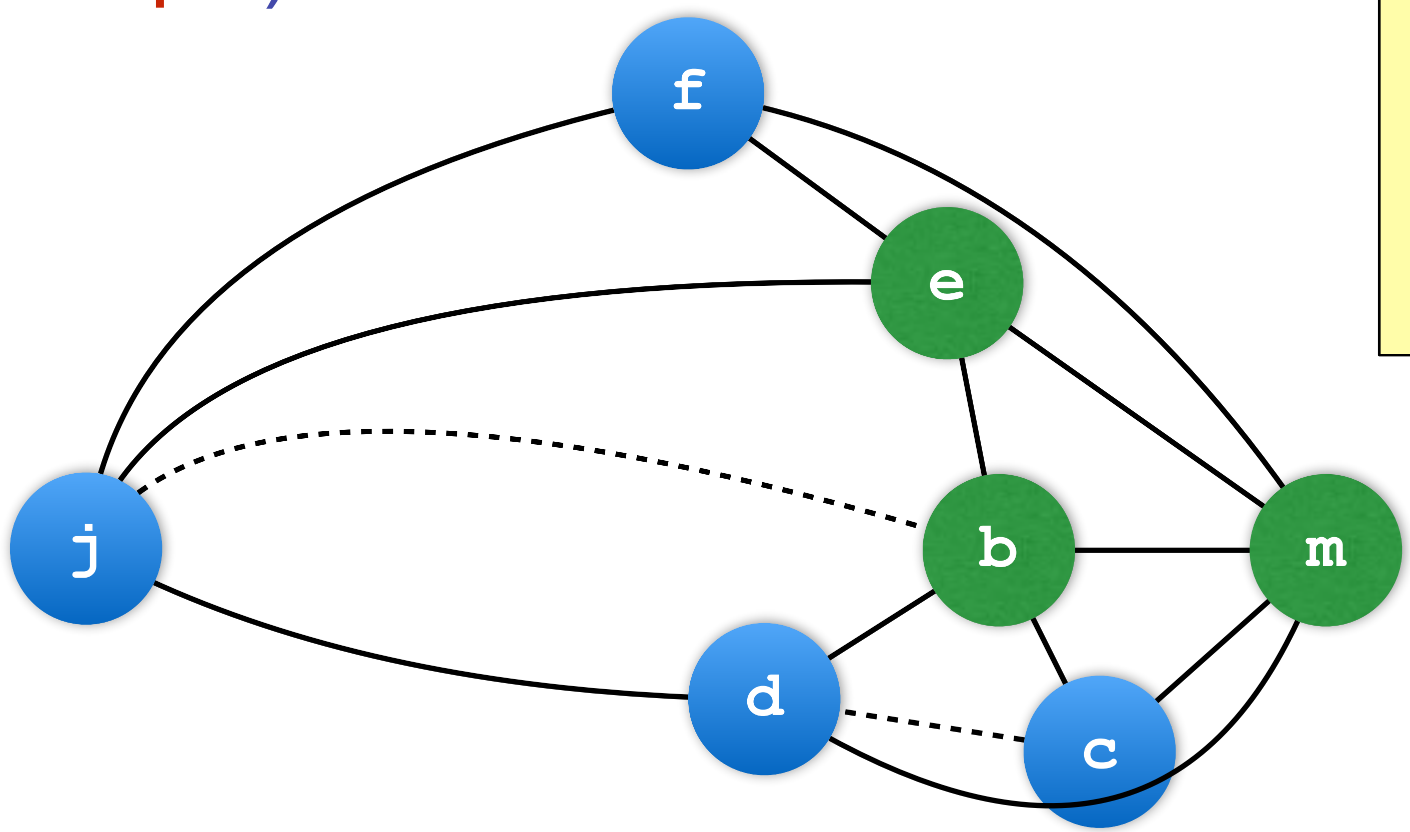
- Simplify:
  - g, h, c, f are *light*, less than *K* neighbors
  - choose g, h, then k for removal

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



Graph Coloring Example

- Simplify:
  - result after removal of g, h, k
  - then continue to produce stack
  - run Select for coloring
  - (no Spill)




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

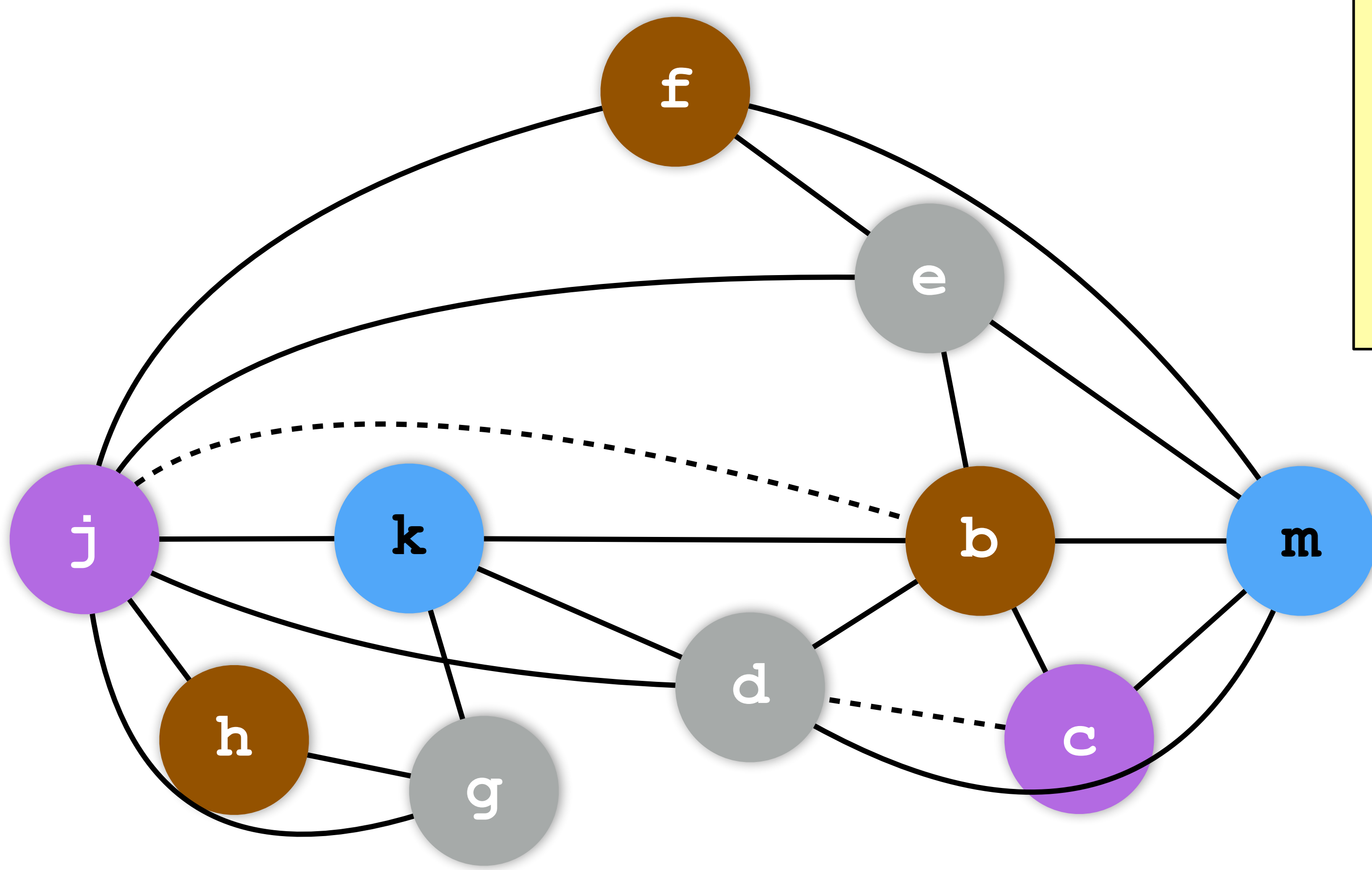
At end:

```
STACK:
  m c b f e j d k h g
COLORING:
  1 3 2 2 4 3 4 1 2 4
```

Graph Coloring Example

- Note important property:
  - choice required during Select
  - NP-completeness: it's hard!

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

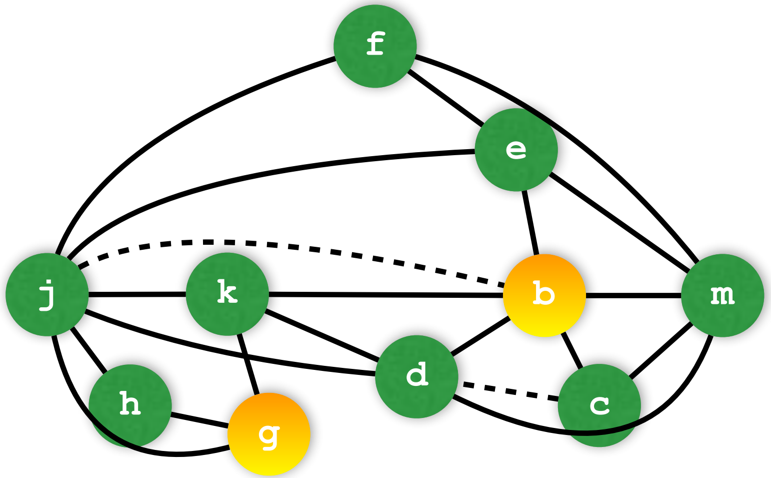


At end:

```
STACK:
  m c b f e j d k h g
COLORING:
  1 3 2 2 4 3 4 1 2 4
```

Coalescing moves

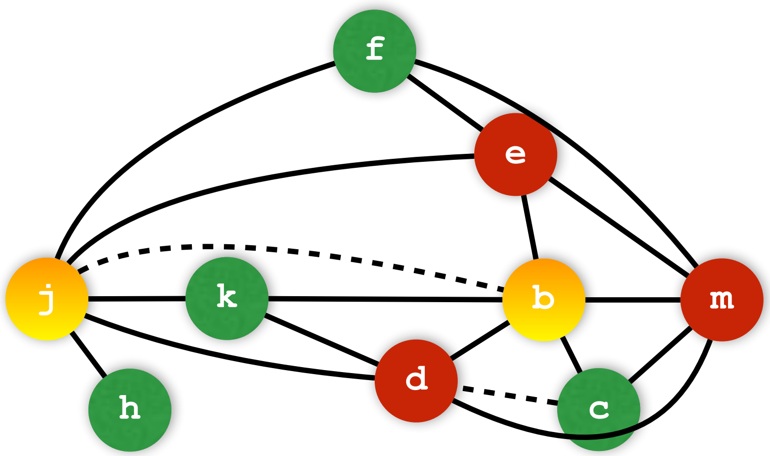
- Basic idea: If two move-related nodes do not interfere, they could be the same color (register)



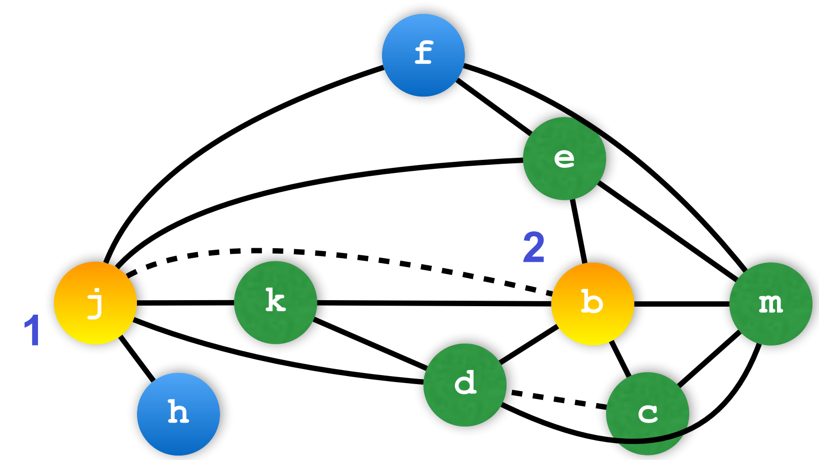
- Problem: Merging two nodes can add a heavy node from two light ones
- Problem: Making a heavy node heavier could prevent it becoming light enough during Simplify

Coalescing Criteria

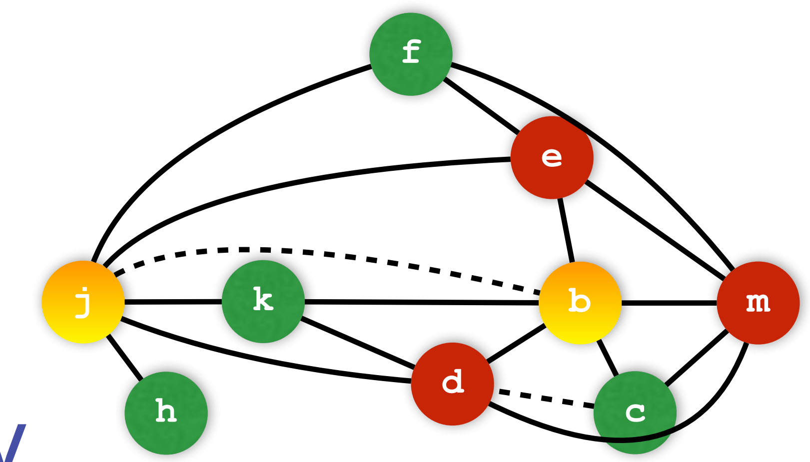
- Solution: Criterion that ensures *K*-colorability preservation



- Briggs: ensure merged node has <*K* heavy neighbors
- George: ensure *first* node to merge has only light exclusive neighbors (i.e., not neighbors of *second* node to merge)



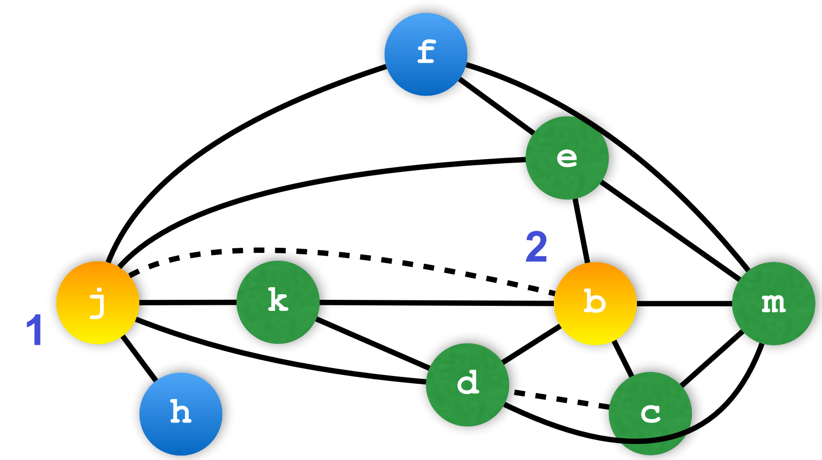
Briggs Correctness



- Briggs: ensure merged node has <*K* heavy neighbors
- Let G *K*-colorable, *j*,*b* have *K'*<*K* heavy neighbors, G' is G with merged node *jb*. Then G' is *K*-colorable

PROOF (sketch)

George Correctness

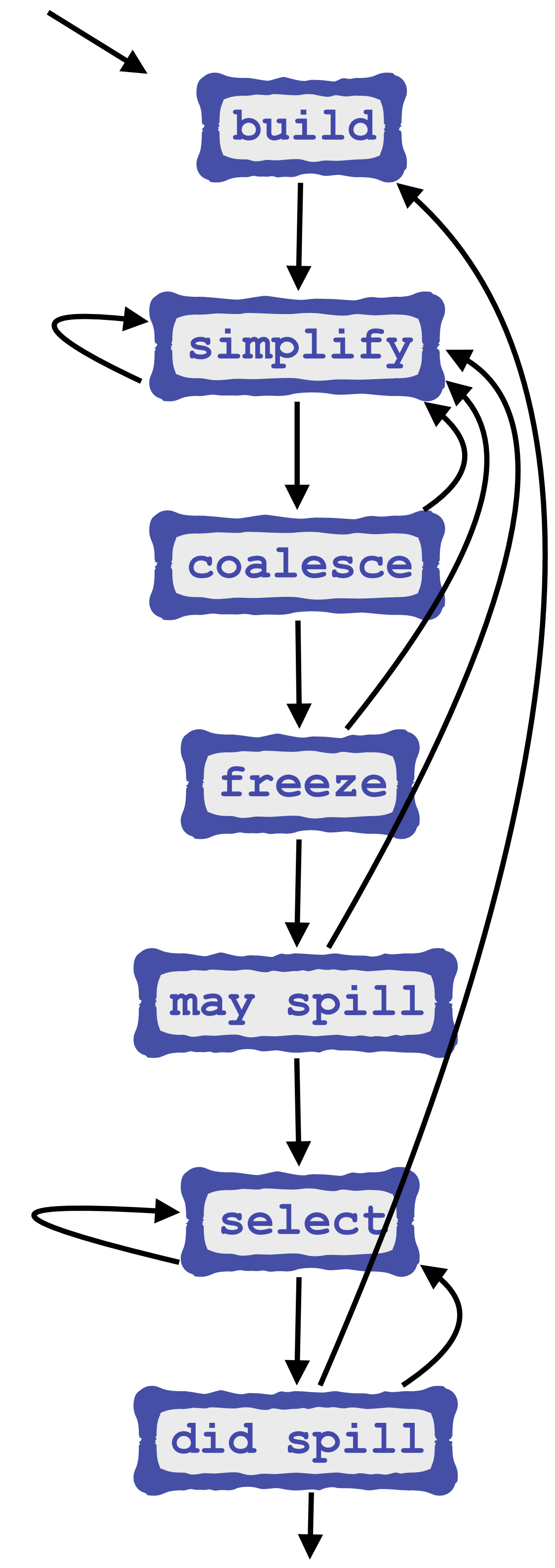


- George: ensure *first* node to merge has only light exclusive neighbors (i.e., not neighbors of *second* node to merge)
- Let G *K*-colorable, *j,b* merging, all exclusive neighbors of *j* light. Let G’ be G with merged node *jb.* *T*hen G' is K-colorable

PROOF (sketch)

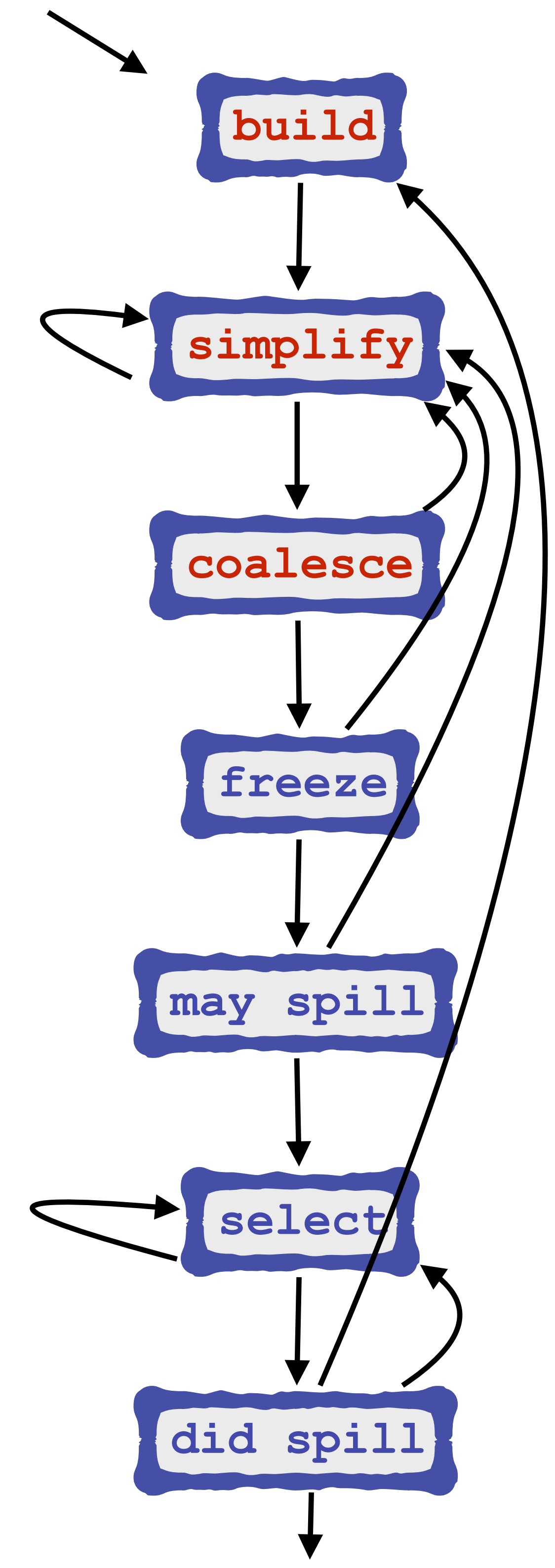
Graph Coloring with Coalescing

- Extend previous algorithm with extra phases
- Simplify modified
- Core addition: coalesce
- Needed: freeze, to give up



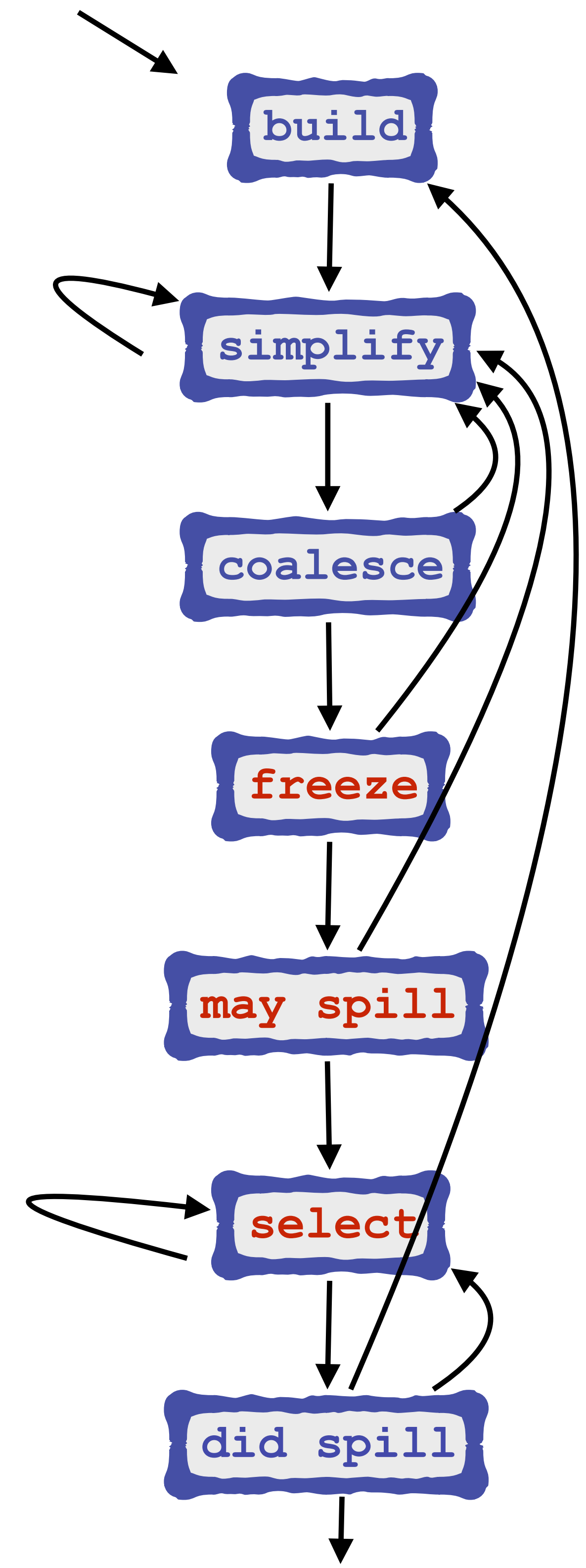
Graph Coloring with Coalescing

- Build: as before, *but* mark end-points of move edges as move related ('moving')
- Simplify: remove light nodes *if* not move related
- *Coalesce*: enforce Briggs or George criterion; repeat until all nodes heavy or moving



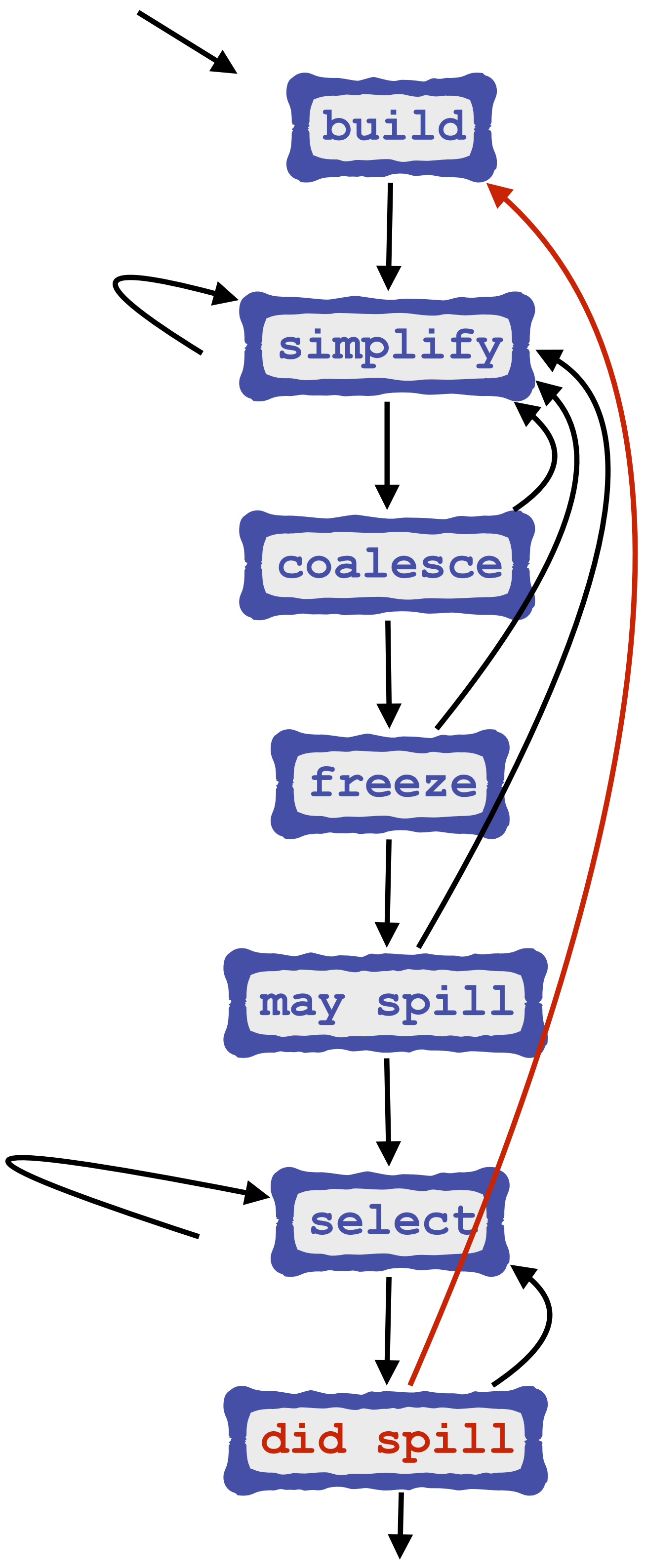
Graph Coloring with Coalescing

- Freeze: unmark one low degree moving node, enabling new simplifications
- Spill: preferring low degree node, select and push
- Select: pop all, assign colors for each reinsertion



Graph Coloring with Coalescing

- Do spill: change program as before
- When actual spill occurred, rebuild graph
- Extra corner case: *constrained move*, where pair has both move and interference (remove ‘moving’ mark)



Spilling — déjà vu

- When rerunning build, we can preserve coalescing nodes created *before* first spill was discovered

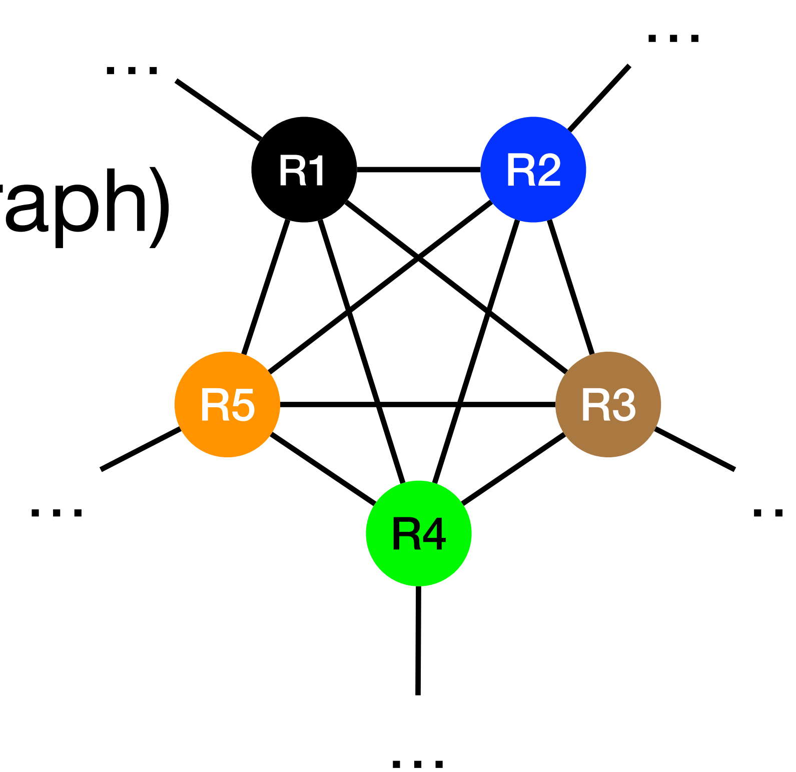
- Stack frame can grow wildly due to spilled temps
- May well have disjoint live ranges: Use graph coloring with coalescing!

- NB: no limit on stack frame size, as if $K = \infty$, just coalesce aggressively (no criteria)

Registers and graph coloring

- Not all registers are interchangeable: e.g., `imull` instruction
- Registers must exist in interference graph

- We add one "*pre-colored*" node per register
- Forced properties on register nodes
  - all register nodes pairs interfere (full subgraph)
  - each register *R* has a unique color *c*
  - color *c* must map to *R*
    - other temps may have color *c*, too
  - cannot spill/freeze register nodes
  - cannot coalesce two register nodes
- Simple rule: "Treat register node as if they had infinite degree"
- Calling conventions (caller/ee-save registers) is relevant



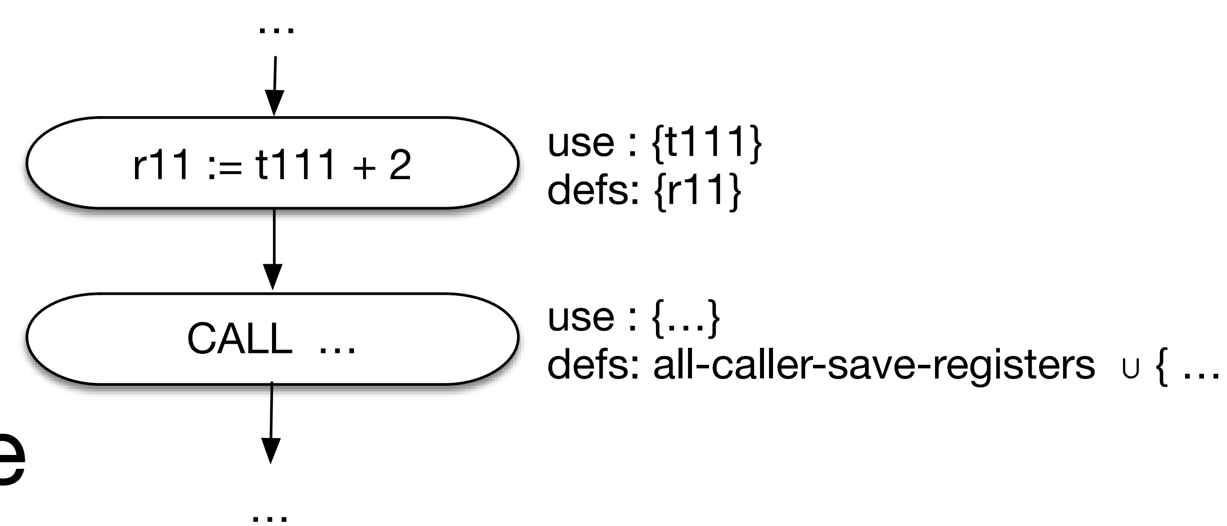
Strategies for dealing w/ callee-save register nodes

- “Don’t-spill-callee-save-register-nodes” rule is needed but it backfires by producing long-living registers (bad)
  - interferes with many temps
  - prevents many possibilities
- Can be avoided by tweaking the code-generator:
  - copy callee-save register to fresh temp node and back before use
    - if low register pressure, coalesce will kick-in and eliminate redundant moves
    - OR: temp is just another register, not too bad either
    - OR: if high register pressure, temp will spill
  - Typical use case: callee-save registers

```
live-in: r7
enter:
    # save r7
    t231 := r7
    . . .
    # restore r7
    r7 := t231
exit:
live-out: r7
```

Treatment of caller-save registers

- `call` instruction (re)defines all caller-save registers
    - means that temps that are live across the call interfere with caller-save registers



- Register allocation w/ spilling tends to allocate short-lived temps to caller-save registers

Example: Graph Coloring w/precoloring

- Consider a C function f, with generated pseudo-assembly code

- Platform has $K = 3$
- r_1, r_2 caller-save; r_3 callee-save
- arguments passed in r_1, r_2 (usual trick: copy to fresh temp)

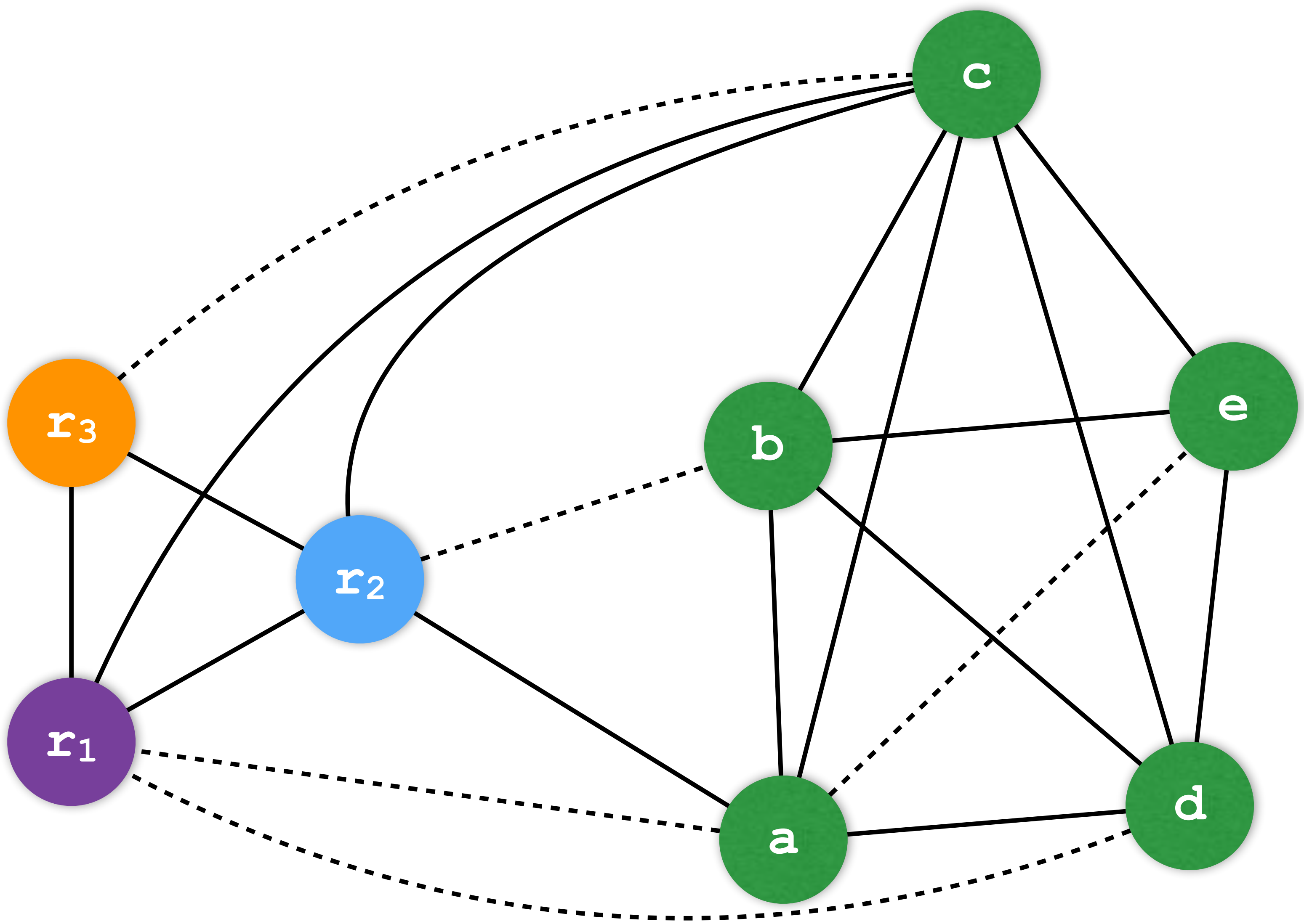
- now compute interference graph (skip control flow graph, boring)

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter: c := r3
       a := r1
       b := r2
       d := 0
       e := a
loop:  d := d + b
       e := e - 1
       if e>0 goto loop
       r1 := d
       r3 := c
live-out: r1 r3
```

Example: Graph Coloring w/ precoloring

- Interference graph

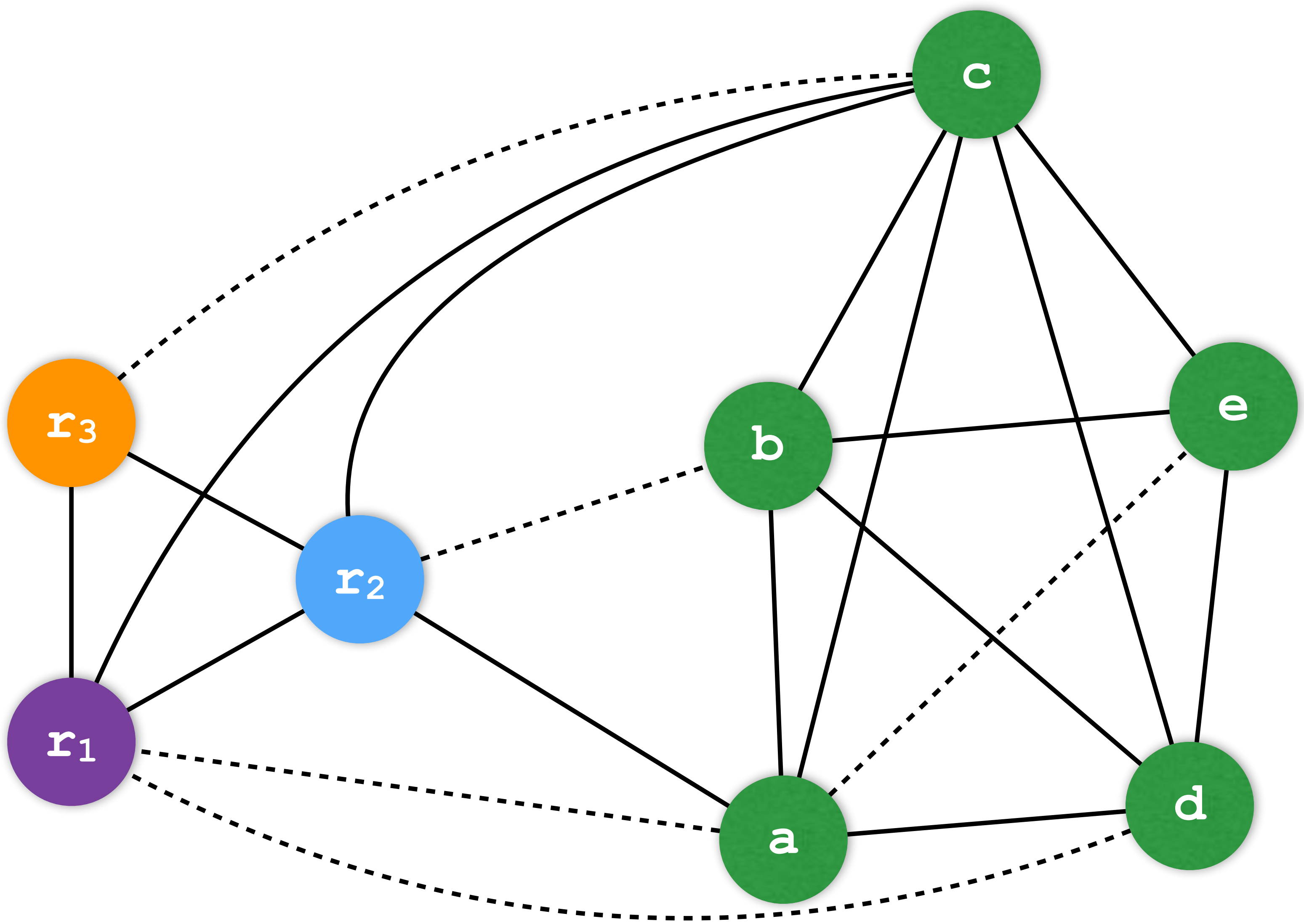



```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter: c := r3
       a := r1
       b := r2
       d := 0
       e := a
loop:  d := d + b
       e := e - 1
       if e>0 goto loop
       r1 := d
       r3 := c
live-out: r1 r3
```

Example

- Cannot simplify, freeze, coalesce, must spill




```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:   d := d + b
        e := e - 1
        if e>0 goto loop
        r1  := d
        r3  := c
live-out: r1 r3
```

Example

- Cannot simplify, freeze, coalesce, must spill — using $(o+10i) / d$

Node	use/def outside loop	use/def in loop	degree	spill priority
a	2	0	4	0,50
b	1	1	4	2,75
c	2	0	6	0,33
d	2	2	4	5,50
e	1	3	3	10,33

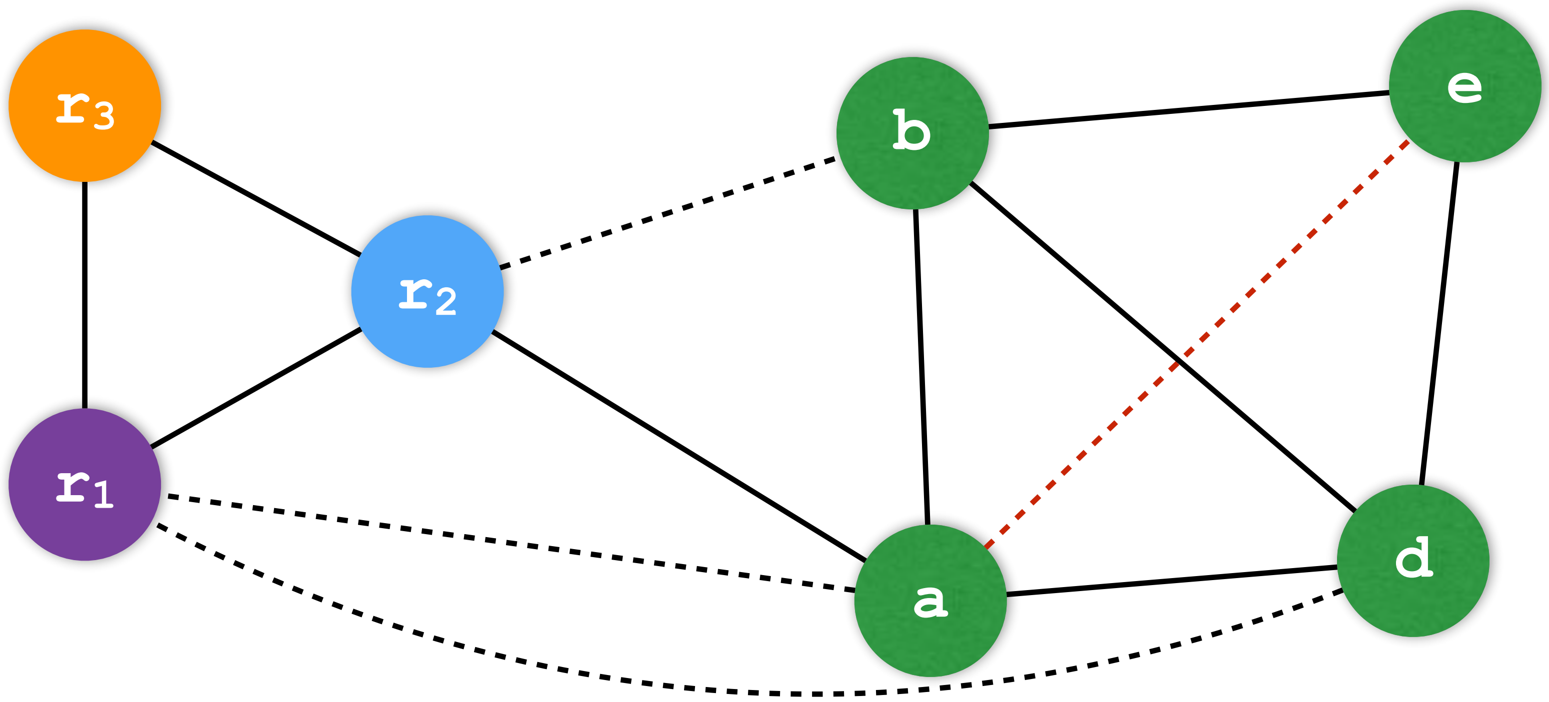
```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter: c := r3
       a := r1
       b := r2
       d := 0
       e := a
loop:  d := d + b
       e := e - 1
       if e>0 goto loop
       r1 := d
       r3 := c
live-out: r1 r3
```

Example

- Spilled c; now coalesce a&e, OK by Briggs

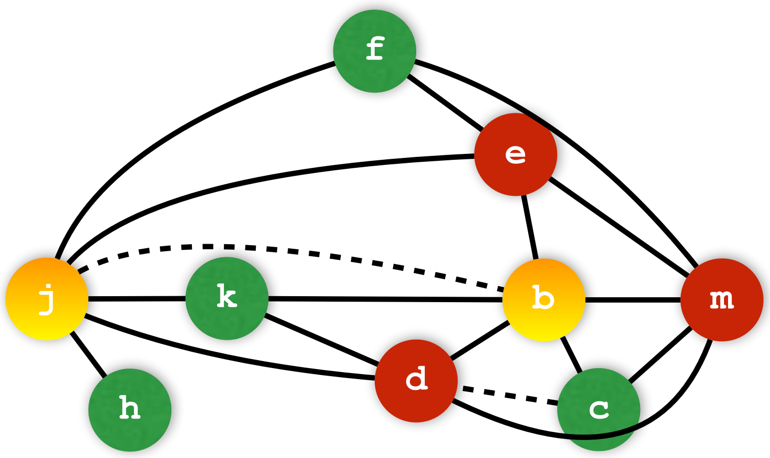
```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```



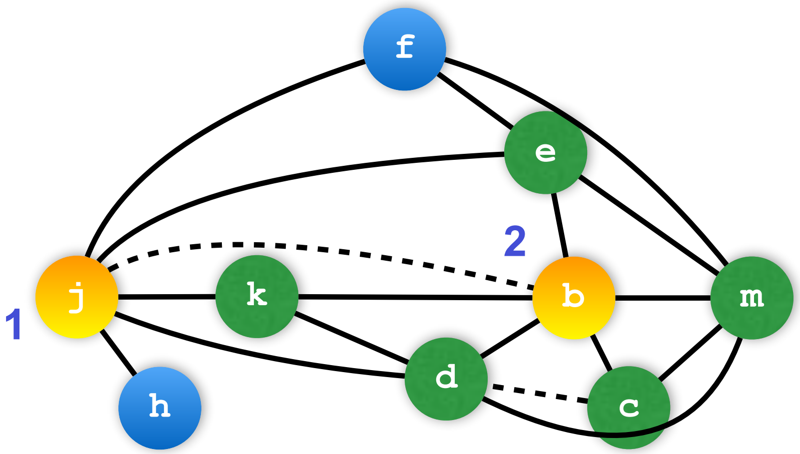

```
live-in: r1 r2 r3
enter:  c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:   d := d + b
        e := e - 1
        if e>0 goto loop
        r1 := d
        r3 := c
live-out: r1 r3
```

Recall: Coalescing Criteria

- Solution: Criterion that ensures *K*-colorability preservation
- Briggs: ensure merged node has <*K* heavy neighbors



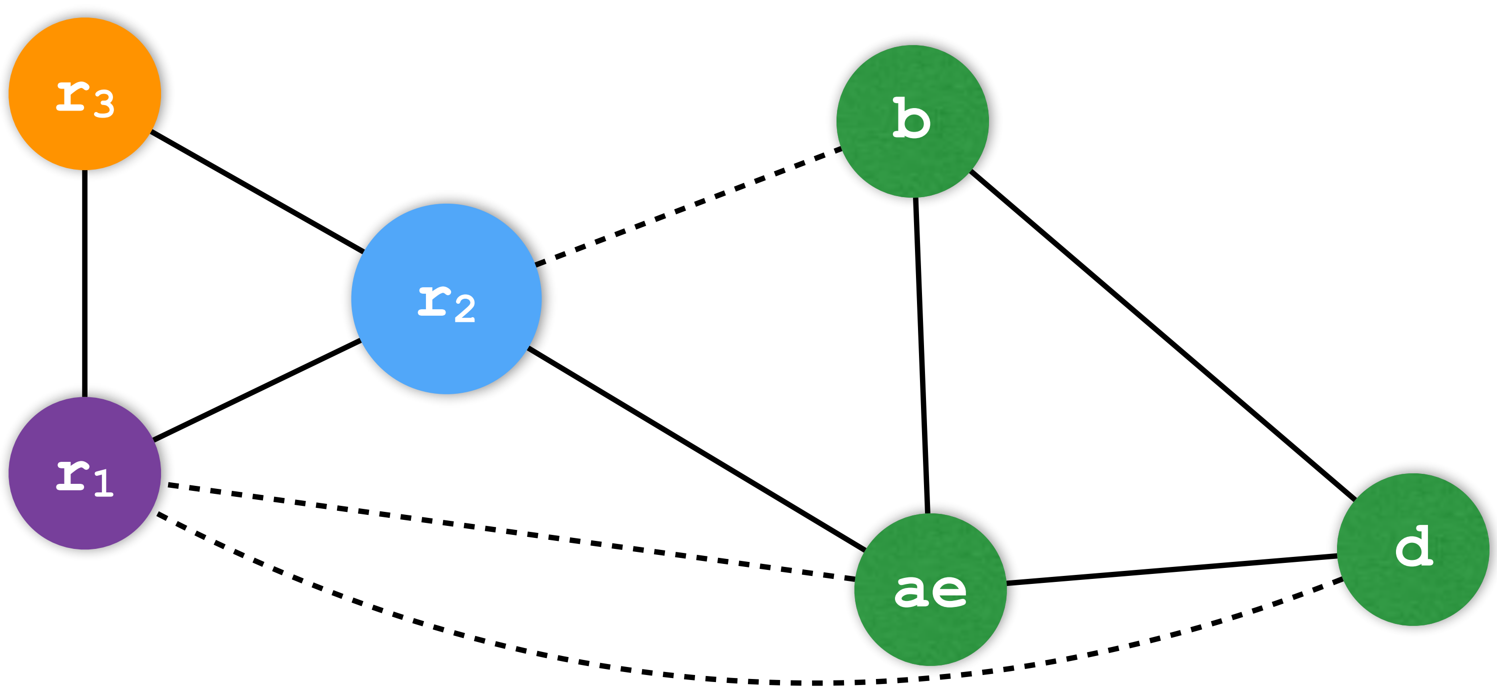
- George: ensure *first* node to merge has only light exclusive neighbors (i.e., not neighbors of *second* node to merge)



Example

- Coalesced a&e; now coalesce ae&r_1 or b&r_2, OK by George
- Q: Why not d&r_1 ?

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```



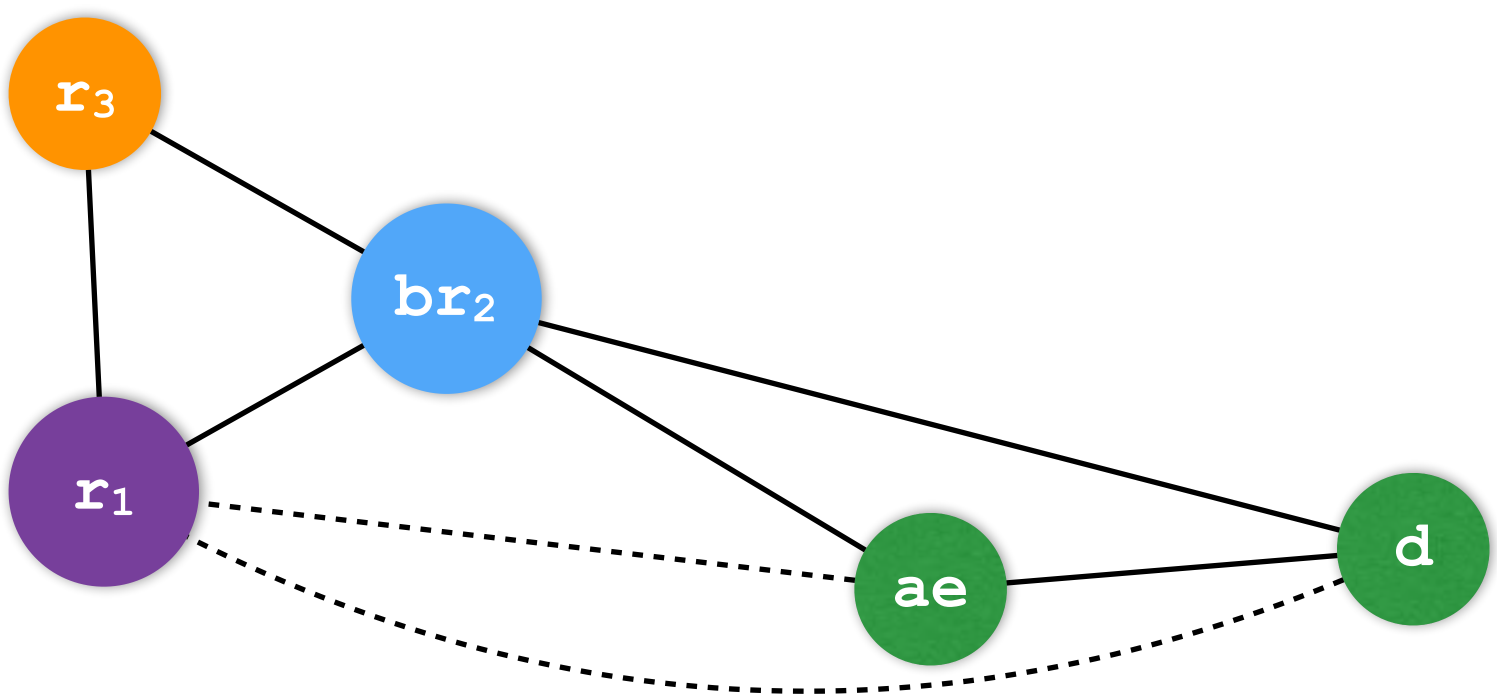

```
live-in: r1 r2 r3
enter: c  := r3
       a  := r1
       b  := r2
       d  := 0
       e  := a
loop:  d  := d + b
       e  := e - 1
       if e>0 goto loop
       r1  := d
       r3  := c
live-out: r1 r3
```

Example

- Coalesced b&r_2; now coalesce ae&r_1 or d&r_1, OK by George
- Q: Why is d&r_1 OK now?

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

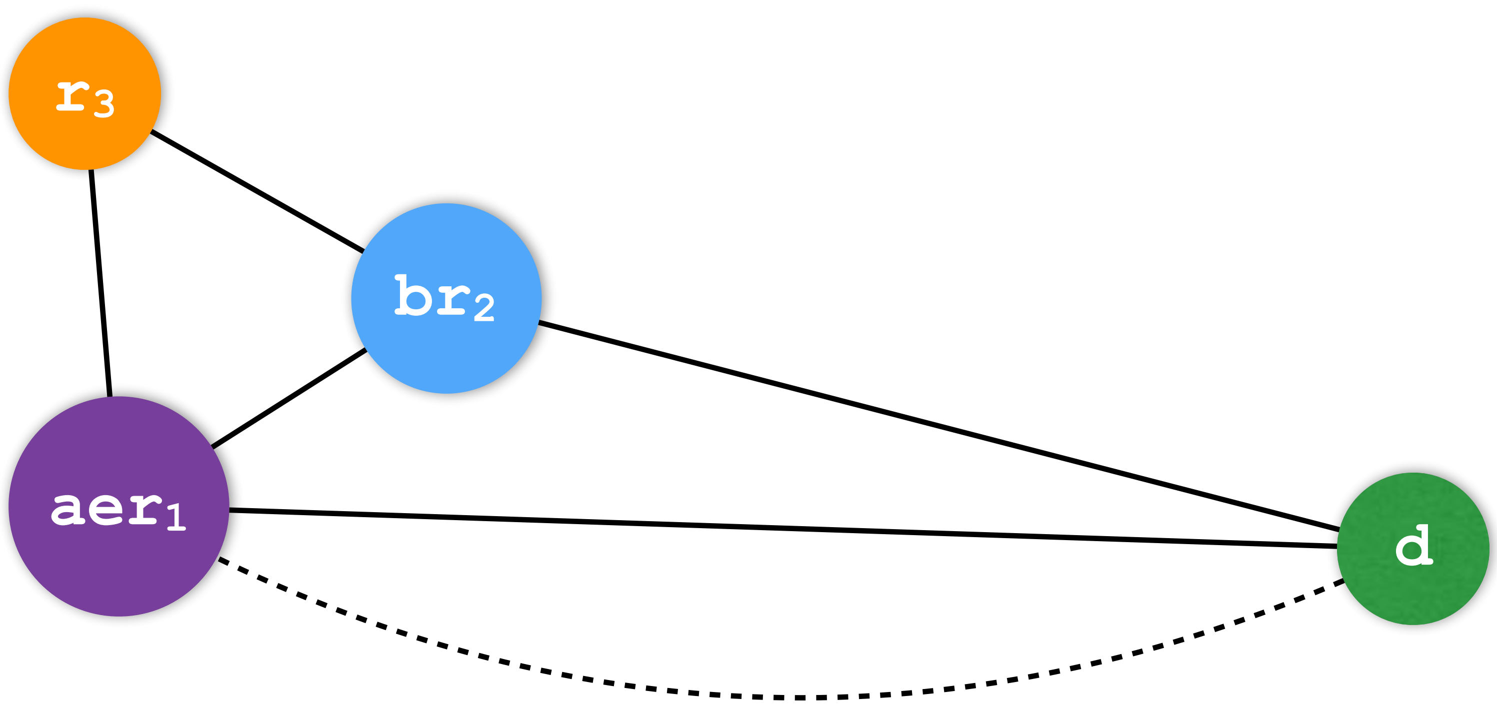
```
live-in: r1 r2 r3
enter: c := r3
       a := r1
       b := r2
       d := 0
       e := a
loop:  d := d + b
       e := e - 1
       if e>0 goto loop
       r1 := d
       r3 := c
live-out: r1 r3
```



Example

- Coalesced ae&r_1; now simplify d
- Q: Why not d&aer$_1$?

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```



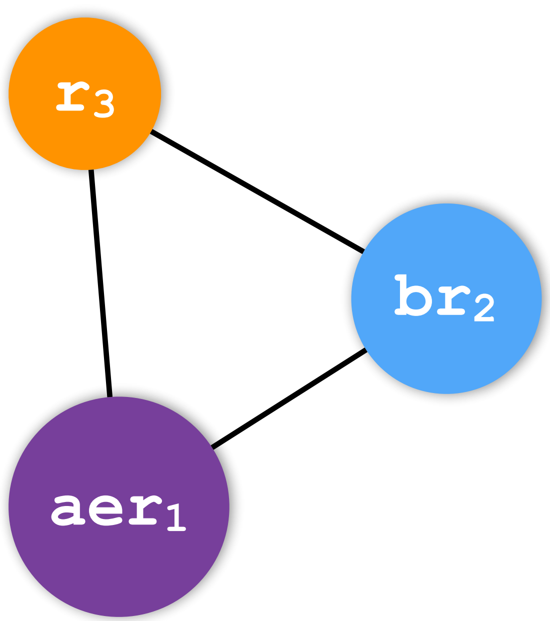

```
live-in: r1 r2 r3
enter: c := r3
       a := r1
       b := r2
       d := 0
       e := a
loop:  d := d + b
       e := e - 1
       if e>0 goto loop
       r1 := d
       r3 := c
live-out: r1 r3
```

Example

- Graph now fully precolored, select:
- d can get color r_3,
- c an actual spill: change program!

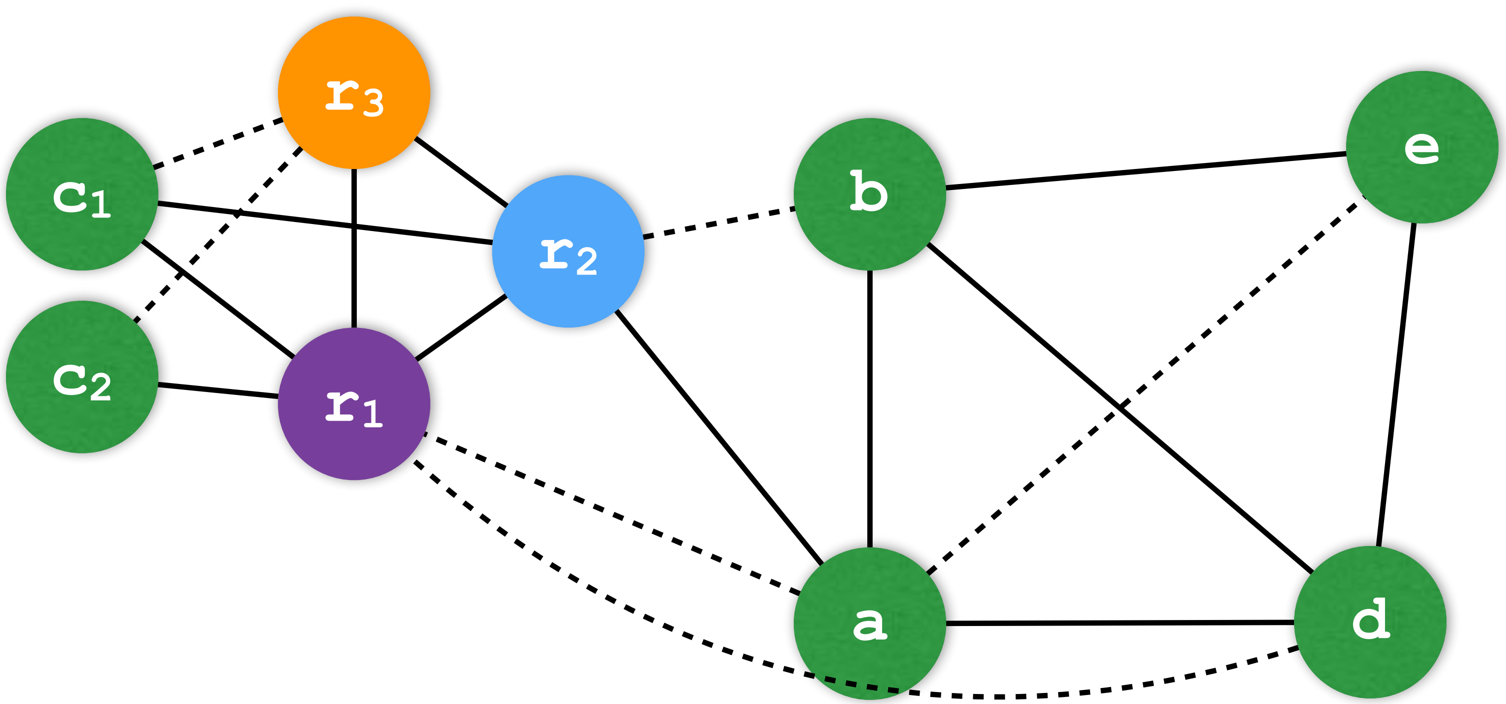
```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  c1  := r3
        M[cloc]  := c1
        a  := r1
        b  := r2
        d  := 0
        e  := a
loop:   d  := d + b
        e  := e - 1
        if e>0 goto loop
        r1  := d
        c2  := M[cloc]
        r3  := c2
live-out: r1 r3
```



Example

- New interference graph

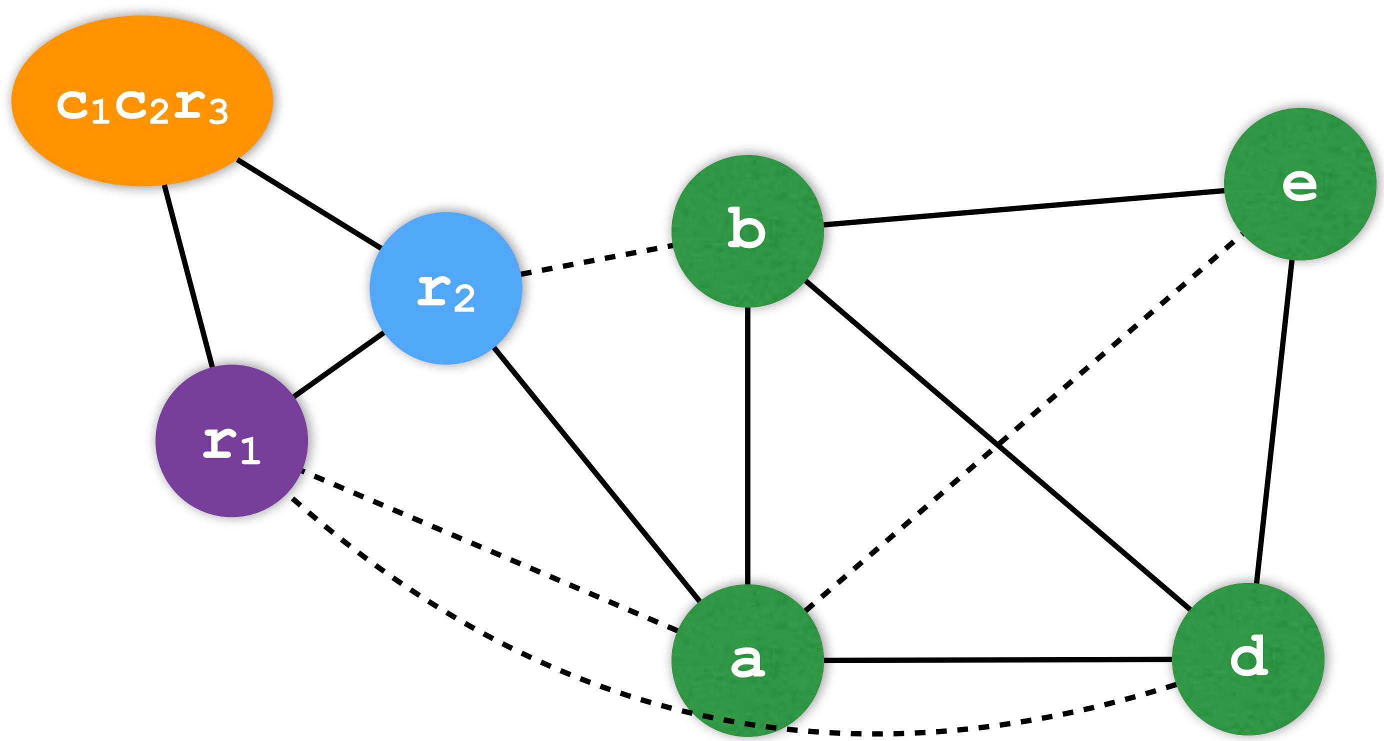



```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  c1  := r3
        M[cloc] := c1
        a  := r1
        b  := r2
        d  := 0
        e  := a
loop:   d  := d + b
        e  := e - 1
        if e>0 goto loop
        r1  := d
        c2  := M[cloc]
        r3  := c2
live-out: r1 r3
```

Example

- Coalesced c_1&r_3, c_2&r_3 (by ..?)

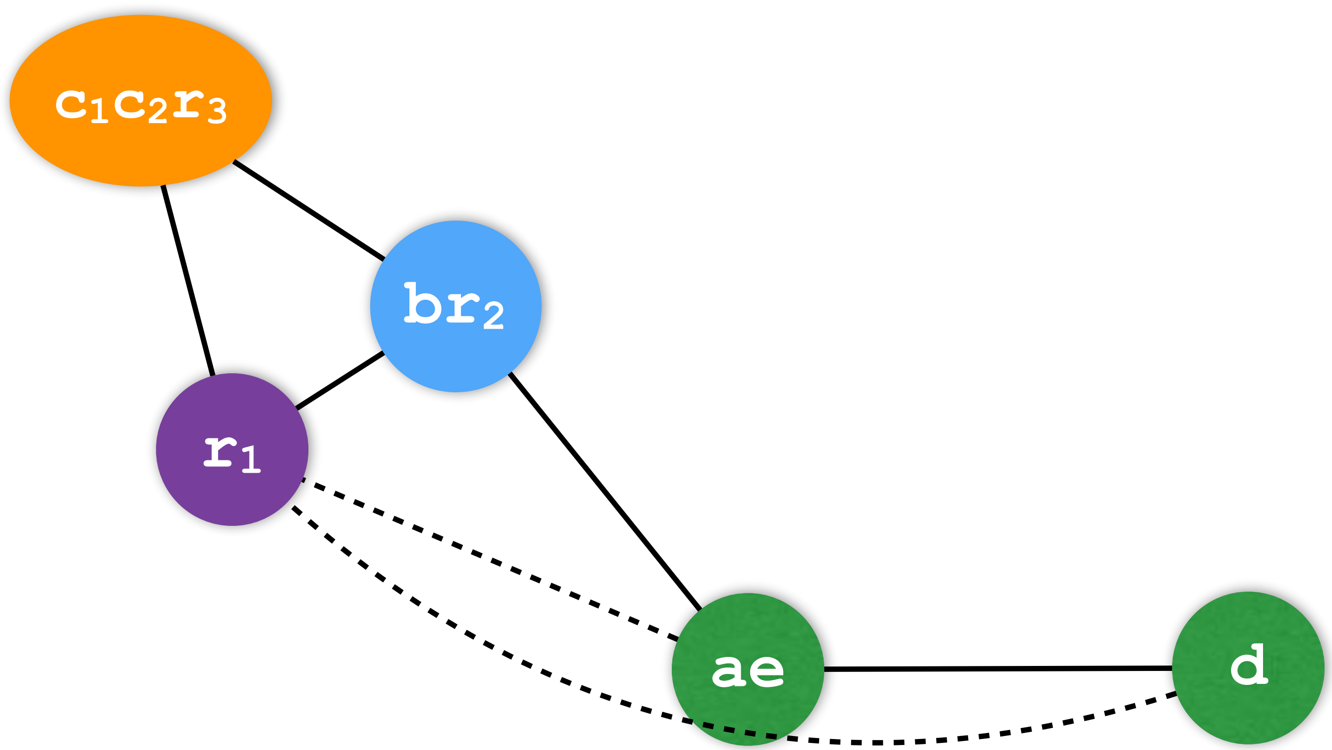



```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  c1 := r3
        M[cloc] := c1
        a := r1
        b := r2
        d := 0
        e := a
loop:   d := d + b
        e := e - 1
        if e>0 goto loop
        r1 := d
        c2 := M[cloc]
        r3 := c2
live-out: r1 r3
```

Example

- Coalesced a&e, b&r_2 (as earlier)

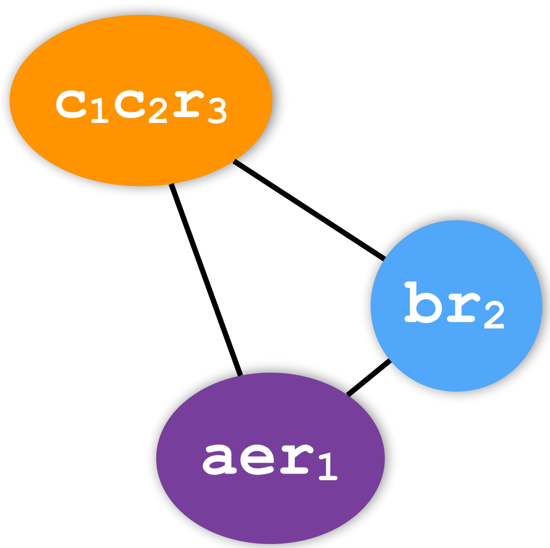



```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  c1  := r3
        M[cloc] := c1
        a  := r1
        b  := r2
        d  := 0
        e  := a
loop:   d  := d + b
        e  := e - 1
        if e>0 goto loop
        r1  := d
        c2  := M[cloc]
        r3  := c2
live-out: r1 r3
```

Example

- Coalesced ae&r_1, simplified d (again, as earlier)




```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  c1  := r3
        M[cloc] := c1
        a  := r1
        b  := r2
        d  := 0
        e  := a
loop:   d  := d + b
        e  := e - 1
        if e>0 goto loop
        r1  := d
        c2  := M[cloc]
        r3  := c2
live-out: r1 r3
```

Example

- Graph precolored, start select:
- only d on stack, gets color r_3

Node/Temp	Color/Register
a	r_1
b	r_2
c	r_3
d	r_3
e	r_1

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  r3  := r3
        M[cloc]  := r3
        r1  := r1
        r2  := r2
        r3  := 0
        r1  := r1
loop:   r3  := r3 + r2
        r1  := r1 - 1
        if r1>0 goto loop
        r1  := r3
        r3  := M[cloc]
        r3  := r3
live-out: r1 r3
```

Example

- Many no-ops detected

Node/Temp	Color/Register
a	r_1
b	r_2
c	r_3
d	r_3
e	r_1

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter:  r3 := r3
        M[cloc] := r3
        r1 := r1
        r2 := r2
        r3 := 0
        r1 := r1
loop:   r3 := r3 + r2
        r1 := r1 - 1
        if r1>0 goto loop
        r1 := r3
        r3 := M[cloc]
        r3 := r3
live-out: r1 r3
```

Example

- Final program
- NB: This is a serious piece of optimization, including many ad-hoc techniques

```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
    d = d+b;
    e = e-1;
  } while (e>0);
  return d;
}
```

```
live-in: r1 r2 r3
enter: M[cloc] := r3
       r3 := 0
loop:  r3 := r3 + r2
       r1 := r1 - 1
       if r1>0 goto loop
       r1 := r3
       r3 := M[cloc]
live-out: r1 r3
```

Register Allocation for Trees

- Could ignore it — just a special case of solution we already have obtained

- However, optimal solution within reach
- ‘Sethi-Ullman’ algorithm includes spilling

- Assumes reordering OK, must know side-effects

Summary 1/2

- Register allocation builds on interference graph
- Idea: colored nodes represent choice of registers
- Graph coloring NP-complete, but linear approx.
- Algorithm: build simplify spill select start_over
- Coalescing: merge two non-interfering nodes
- Problem: creates ‘heavier’ nodes
- Solution: criteria (Briggs, George)
- Enhanced algorithm: build simplify coalesce freeze may_spill select did_spill
- Can use aggressive coalescing on the stack

Summary 2/2

- Registers must exist in interference graph, pre-colored
- Special treatment gathered into ‘infinite degree’
- Useful technique: copy to/from fresh temp
- Ex: allows flexible handling of callee-save register
- Ex: caller-save register gravitates toward short life
- (Example)
- Note special case: Register allocation for trees