Compiler Design

Spring 2018

9 Register allocation

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland

Outline

- **9.1 Introduction**
    - Live range
    - Interference graph
- **9.2 Graph coloring**
- **9.3 Live range spilling**
- **9.4 Live range splitting**

9.1 Register allocation

- **IA32 demands that (at least) one operand of an instruction is in a register**
    - Other machines (RISC architectures like MIPS, Power, SPARC, …) demand that *all* operands reside in registers
- **There is a finite number of registers**
    - Given an expression tree, choice of evaluation order may help (reduce register demands)
    - Some expression trees require more registers than provided by the target architecture
- **Compiler must manage**
    - Which operand resides in a register
    - Which register is used to hold operand

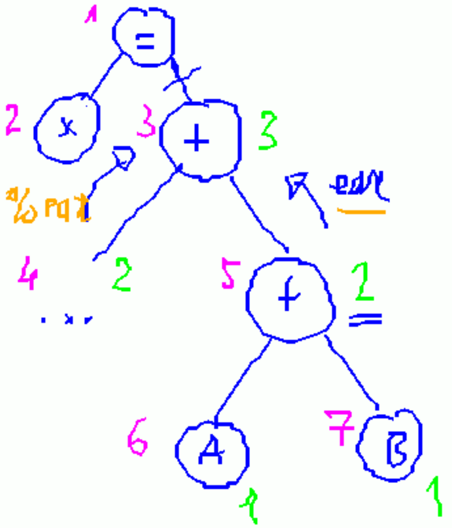
Register allocation

- **Many approaches, many papers…**
- **Interesting problem: compiler must manage a limited resource**
  - Try to do a good job
  - Finding a perfect (optimal) solution not practical

Recall: Code generation for operand accesses

- **(Back in lecture ”7.0 Code generation”)**
- **Approach: produce code (select instructions) and *assume unlimited number of registers***
  - “Virtual registers”
  - Later phase maps virtual registers to real registers
- **(Recommended) alternative for Homework 4:
Handle register shortage “on the fly”**
  - Need a register? Free a register
  - Save register contents onto stack

```
X = … + A + B
reg_pool = {%~~eax~~,%~~ebx~~}
```



```
cg(3)
  gen(4)
    ...
    movl ..., %eax
  push %eax.
  gen(5)  // oops: we have 1 register
                   2 registers are needed
             Idea: free one register
                   (result from left subtree)
    get(6)
       movl a, %eax
    gen(7)
       movl b, %ebx
    addl %ebx, %eax
  pop %ebx
  addl %eax, %ebx
```

Register allocation: Problem statement

- **(Let's assume code generator uses virtual registers)**
  - HotSpot C1 compiler also uses this approach
    - High-level IR – uses virtual registers
    - Low-level IR– uses real registers
- **Given an IR program with virtual registers v_1, v_2, …**
- **Decide when a virtual register is *assigned* to a real (physical) register**
  - Like `%eax`, `%ebx`, …

Register allocation: Problem statement

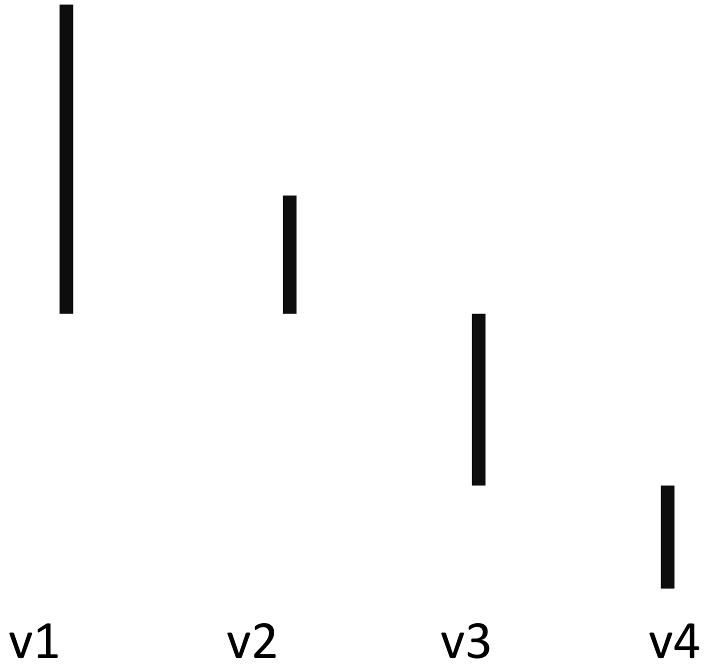
- **If no physical register is available then store virtual register in memory**
  - Retrieve and store as needed

- **Start: find out where virtual registers are live**
  - Two virtual registers cannot be given same register if alive at the same point P in the program
    - “live simultaneously”

Live ranges

- **Range where a { virtual register | variable } is live**
  - Range: a sequence of instructions

```
v1 = a + b
c = v1 + k
v2 = b * 2
v3 = v1 + v2
d = v3 * j
v4 = v3 + 1
e = v4 * 2
```



Computing live ranges

- **Virtual register live if there is another use**
- **Idea: treat virtual registers like variables in global dataflow**
  - Compute liveness information
  - Set L_P: virtual registers live at point P

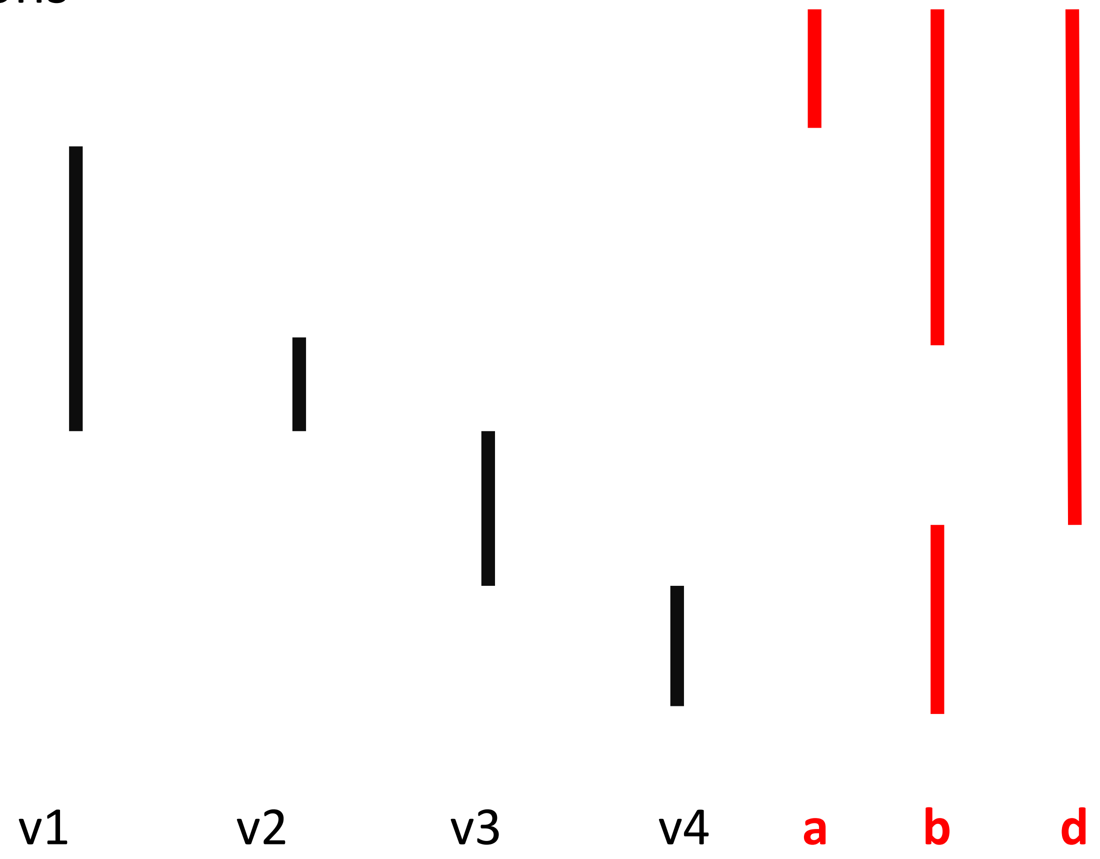
Live ranges

- **Range where a virtual register is live**
  - Range: a sequence of instructions

```
                              L=∅
v1 = a + b                    L={v1}
c = v1 + k                    L={v1}
v2 = b * 2                    L={v1,v2}
v3 = v1 + v2                  L={v3}
d = v3 * j                    L={v3}
v4 = v3 + 1                   L={v4}
e = v4 * 2
```

- **What about *normal variables*? Consider**
  - ISlightly) different instructions

```
v1 = a + b
c = v1 + d
v2 = b * 2
v3 = v1 + v2
b = v3 * d
v4 = v3 + 1
e = v4 * b
```



Live ranges

- **Range where a { *virtual register* | *variable* } is live**

```
v1 = a + b                L={a,b,d}
c = v1 + d                L={v1,b,d}
v2 = b * 2                L={v1,b,d}
v3 = v1 + v2              L={v1,v2,d}
b = v3 * d                L={v3,d}
v4 = v3 + 1               L={v3,b}
e = v4 * b                L={v4,b}
```

Computing live ranges

- **Virtual register live if there is another use**
- **Idea: treat virtual registers like variables in global dataflow**
  - Compute liveness information
  - Compute reaching definitions
  - Set L_P: virtual registers live at point P

Live ranges

- **Range where a { virtual register | variable } is live**
  - Range: a sequence of instructions

D_b

`v1 = a + b`	L={a,b,d}	R={D_a,D_b,D_d}
`c = v1 + d`	L={v1,b,d}	R={D_{v1},D_b,D_d}
`v2 = b * 2`	L={v1,b,d}	R={D_{v1},D_b,D_c,D_d}
`v3 = v1 + v2`	L={v1,v2,d}	R={D_{v1},D_{v2},D_b,D_d,D_c}
D'_b `b = v3 * d`	L={v3,d}	R={D_{v1},D_{v2},D_{v3},D_b,D_d,D_c}
`v4 = v3 + 1`	L={v3,b}	R={D_{v1},D_{v2},D_{v3},D'_b,D_d,D_c}
`e = v4 * b`	L={v4,b}	R={D_{v1},D_{v2},D_{v3},D_{v4},D'_b,D_d,D_c}

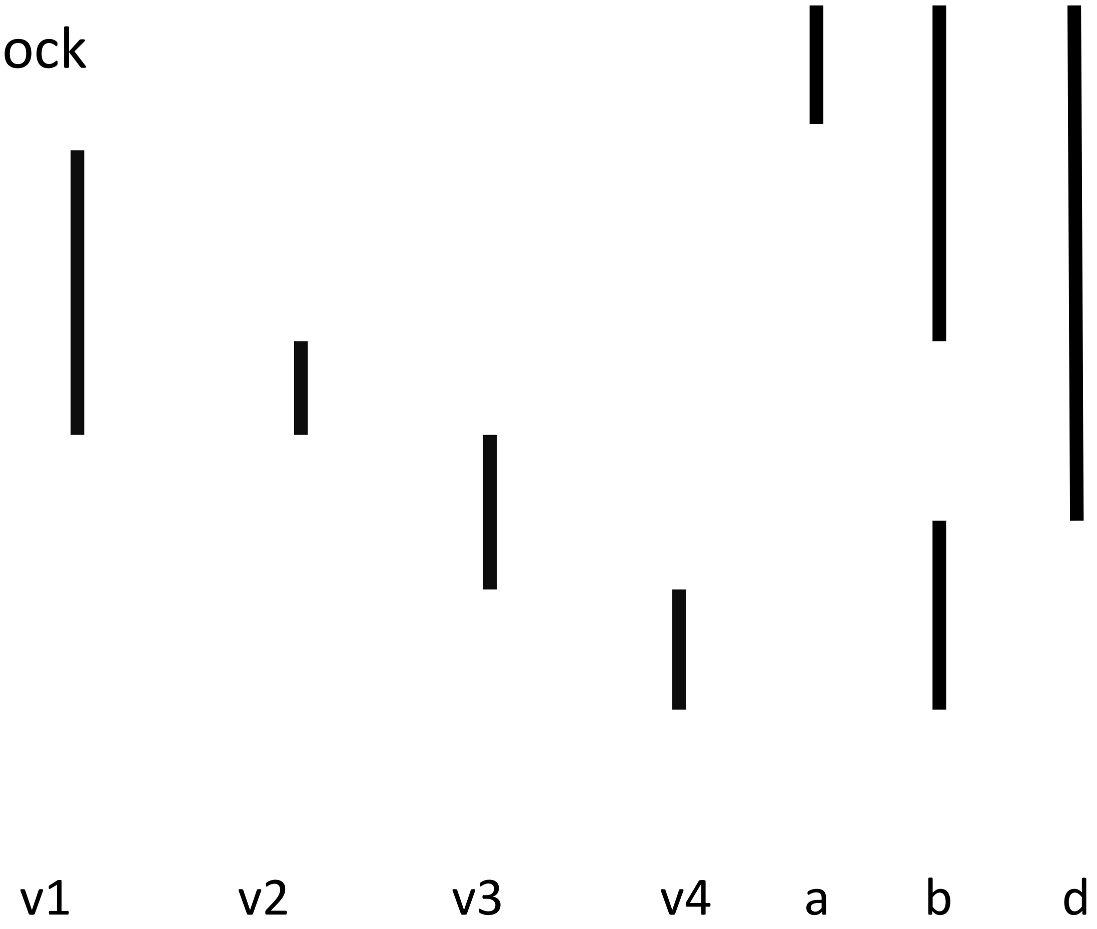
- **Live range: intersection of instructions where a definition reaches with instructions where a variable is live**

Live ranges

- **One possible understanding: live range ends “on the right hand side” of a statement, starts on “the left hand side”**
  - Allows us to realize that a register freed by an operand can be used for the result
- **Many compilers do *not* work with such a fine-grained model**
  - Live range extends till the end of the statement, live range includes complete statement
- **Model of live range can be extended to basic blocks**
  - Live range of a variable or virtual register v is the set of basic blocks B_i such that v ‘s live range includes a statement S from B_i.

- **Compiler computes live ranges –** ***here shown for statements***
  - Live ranges inside a basic block

```
v1 = a + b
c = v1 + d
v2 = b * 2
v3 = v1 + v2
b = v3 * d
v4 = v3 + 1
e = v4 * b
```

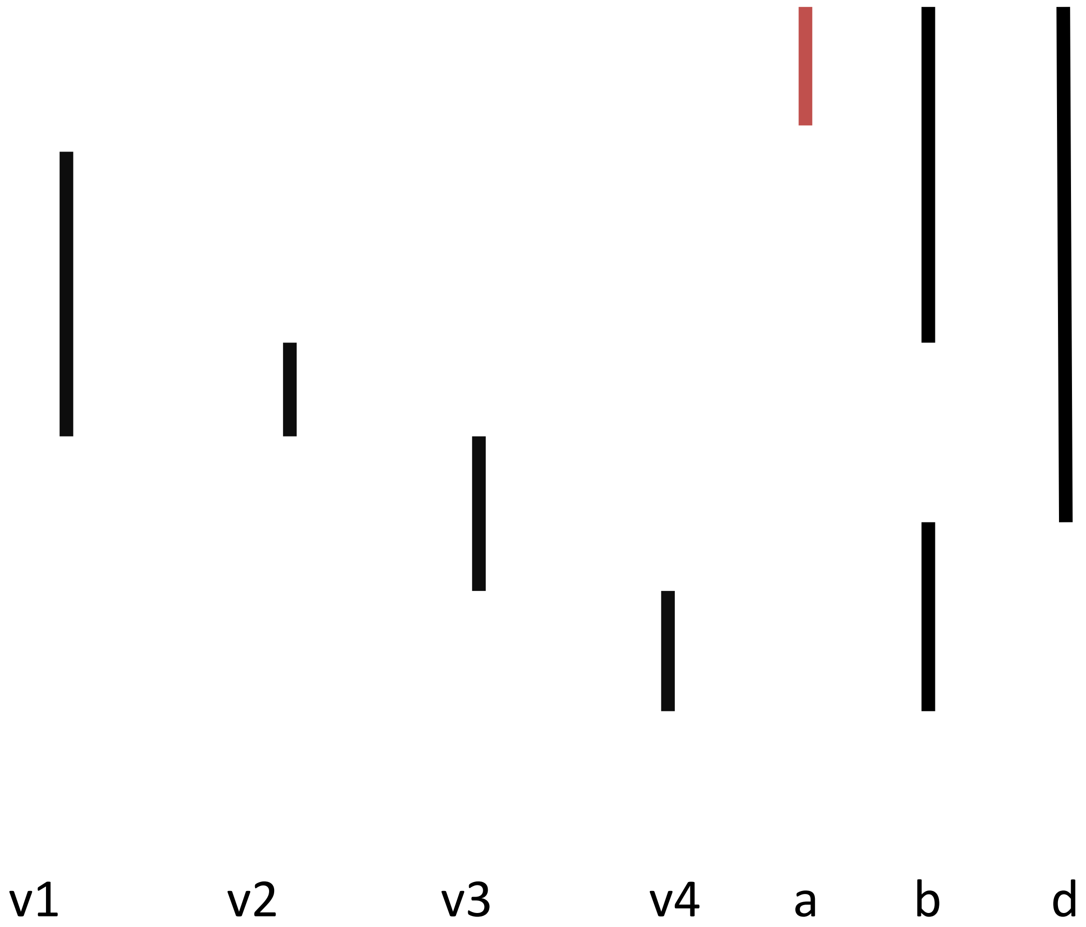


Recap: Problem statement

- **Decide when a virtual register is assigned to a real (physical) register**
  - Like `%eax`, `%ebx`, …
- **If no physical register is available then store virtual register in memory**
  - Retrieve and store as needed
- **Start: find out where virtual registers are live**
  - Two virtual registers cannot be given same register if alive at the same point P in the program
  - Note: virtual registers and program variables are treated the same
  - "Live range"

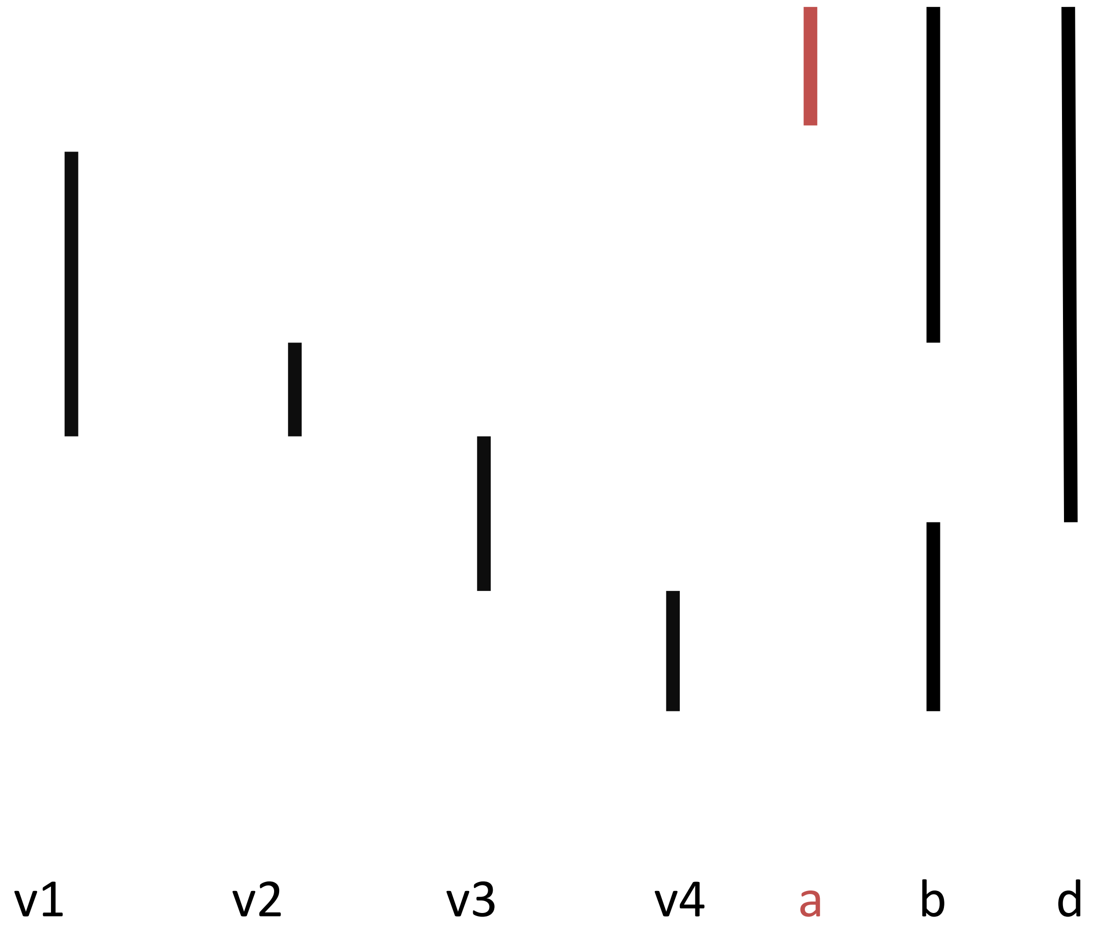
Simplification of example

```
v1 = a + b
c = v1 + d
v2 = b * 2
v3 = v1 + v2
b = v3 * d
v4 = v3 + 1
e = v4 * b
```



Simplification of example

```
v1 = 1 + b
c = v1 + d
v2 = b * 2
v3 = v1 + v2
b = v3 * d
v4 = v3 + 1
e = v4 * b
```



Register allocation: Problem statement

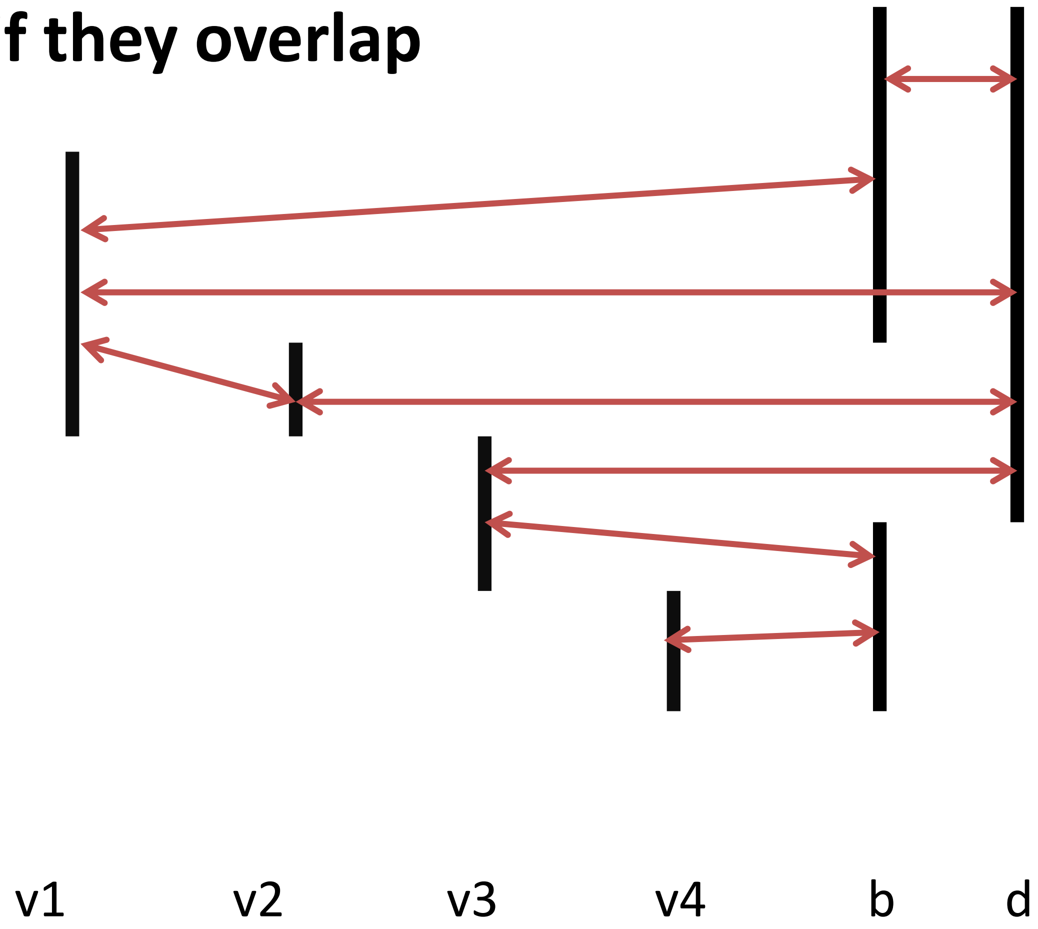
- If no physical register is available then store virtual register in memory
  - Retrieve and store as needed

- **Start: find out where virtual registers are live**
  - Two virtual registers cannot be given same register if alive at the same point P in the program
    - “live simultaneously” -- live ranges of virtual registers overlap

Interference

- Two live ranges *interfere* if they overlap

```
v1 = 1 + b
c = v1 + d
v2 = b * 2
v3 = v1 + v2
b = v3 * d
v4 = v3 + 1
e = v4 * b
```



Interference graph

- **Nodes of the graph: live ranges**
  - Labelled with name of { virtual register | variable }
    - Note: for variables *subscript* distinguishes between different live ranges for same variable
    - Remember: There are multiple definitions for the same variable
- **Edges indicate if the live ranges interfere**

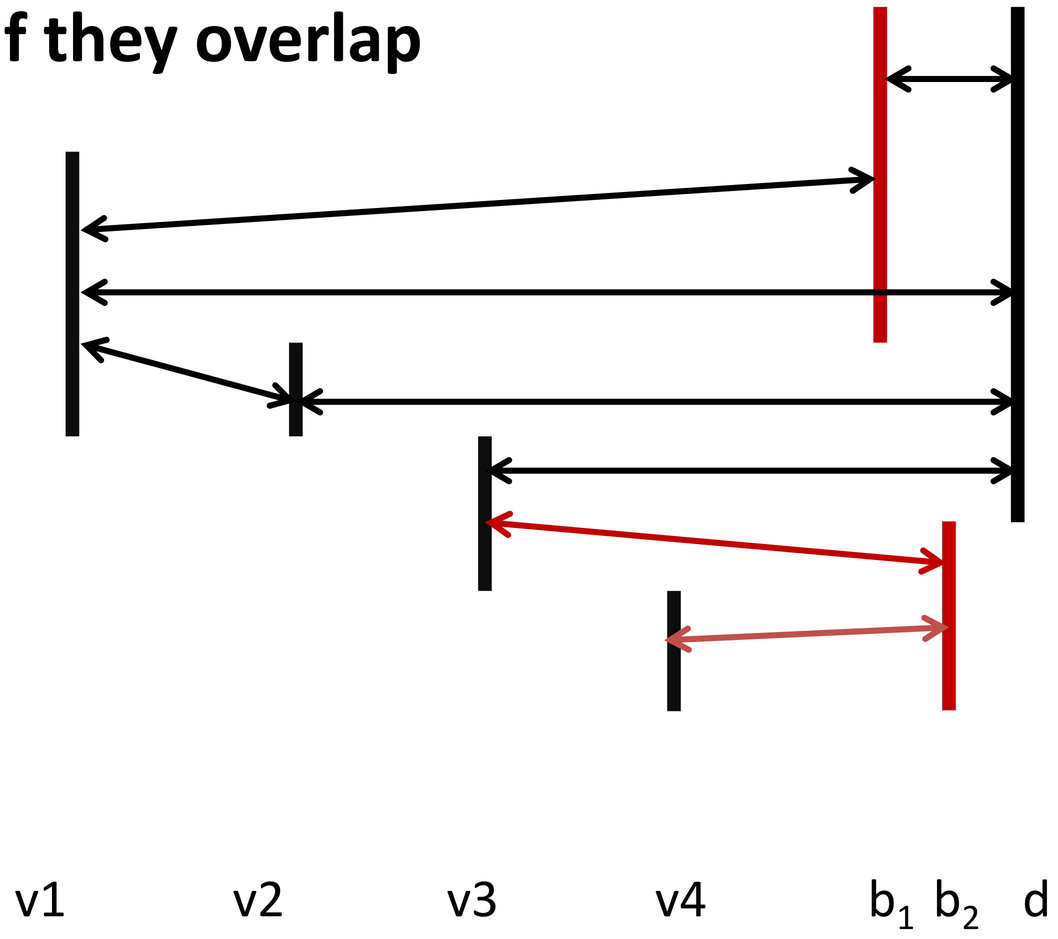
Interference graph

- Nodes of the graph: live ranges
- Edges indicate if the live ranges interfere

Interference – precise view

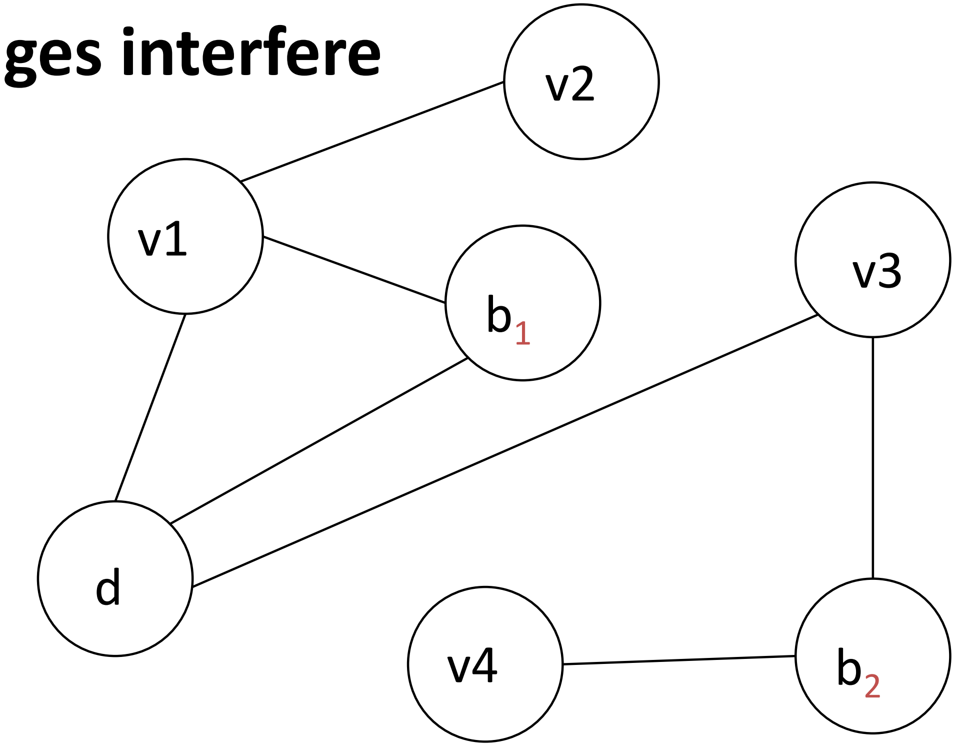
- **Two live ranges *interfere* if they overlap**

```
v1 = 1 + b
c = v1 + d
v2 = b * 2
v3 = v1 + v2
b = v3 * d
v4 = v3 + 1
e = v4 * b
```



Interference graph

- Nodes of the graph: live ranges
- Edges indicate if the live ranges interfere



Observation

- **We assigned one node in the graph *for every definition* of a variable v**
  - Remember: Live range is *the intersection of* instructions where a definition of variable v *reaches* with instructions where variable v is *live*
- **What would happen if we used only liveness information?**

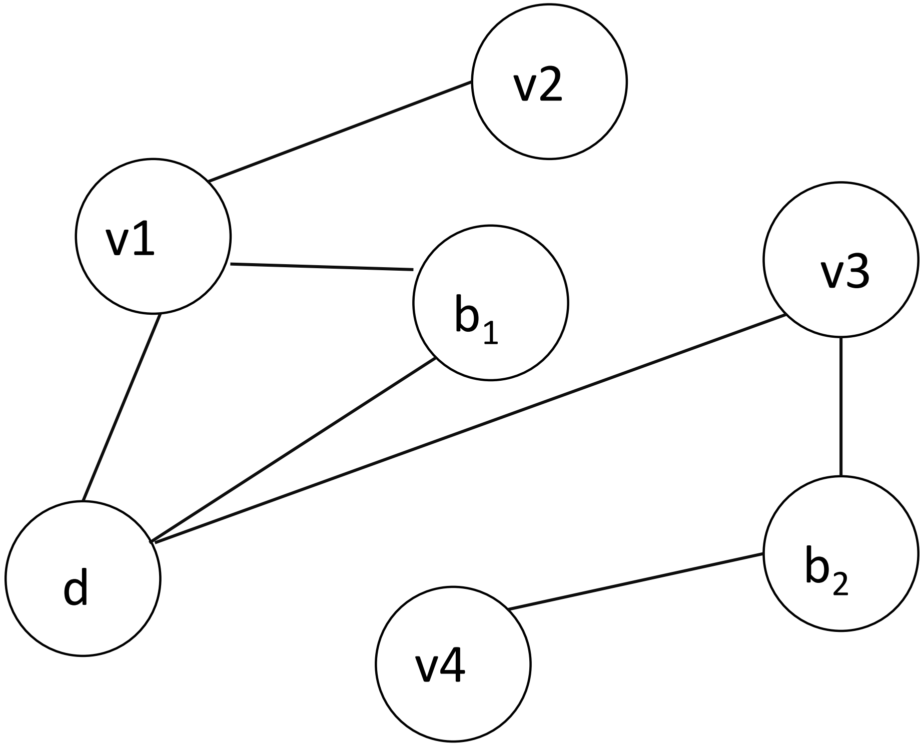
Live ranges

- **Range where a { virtual register | variable } is live**

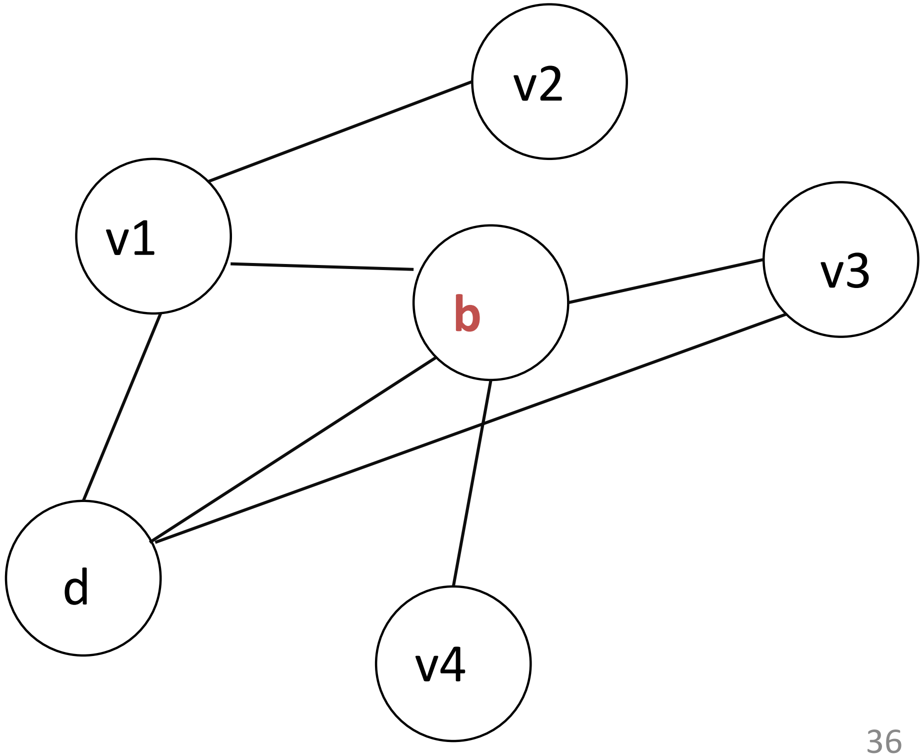
`v1 = a + b`	L={a,b,d}	R={D_a,D_b,D_d}
`c = v1 + d`	L={v1,b,d}	R={D_{v1},D_b,D_d}
`v2 = b * 2`	L={v1,b,d}	R={D_{v1},D_b,D_c,D_d}
`v3 = v1 + v2`	L={v1,v2,d}	R={D_{v1},D_{v2},D_b,D_d,D_c}
`b = v3 * d`	L={v3,d}	R={D_{v1},D_{v2},D_{v3},D_b,D_d,D_c}
`v4 = v3 + 1`	L={v3,b}	R={D_{v1},D_{v2},D_{v3},D'_b,D_d,D_c}
`e = v4 * b`	L={v4,b}	R={D_{v1},D_{v2},D_{v3},D_{v4},D'_b,D_d,D_c}

Interference graph

- **w/ reaching definitions**



- **w/o reaching definitions**



Observation

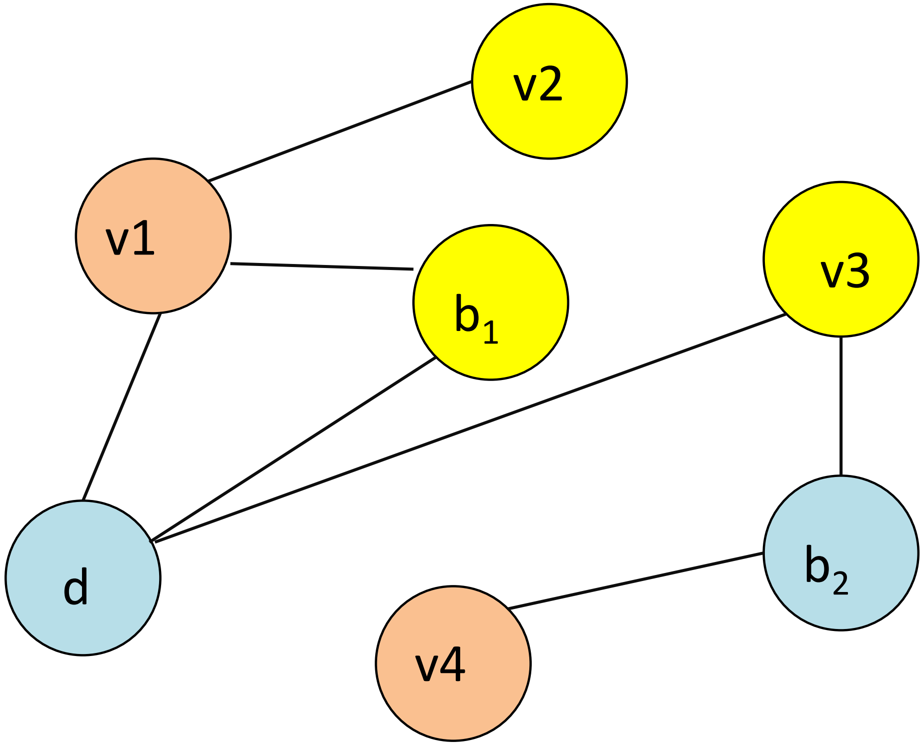
- **We assigned one node in the graph *for every definition* of a variable v**
  - Remember: Live range is *the intersection of* instructions where a definition of variable v *reaches* with instructions where variable v is *live*
- **What would happen if we used only liveness information?**
- **Unnecessary restriction: Both “versions” of variable b must be kept in the same register**
  - Also, we have one node the interferes with four others

9.2 Register allocation and graph coloring

- **Register allocation problem modeled as graph coloring problem**
- **Given *K* colors, determine colors for the nodes of the interference graph so that nodes connected by an edge have different colors**
  - If possible we say the graph is K-colorable
- **If live ranges are simultaneous (there is an edge in the graph) they have different colors (reside in different registers)**

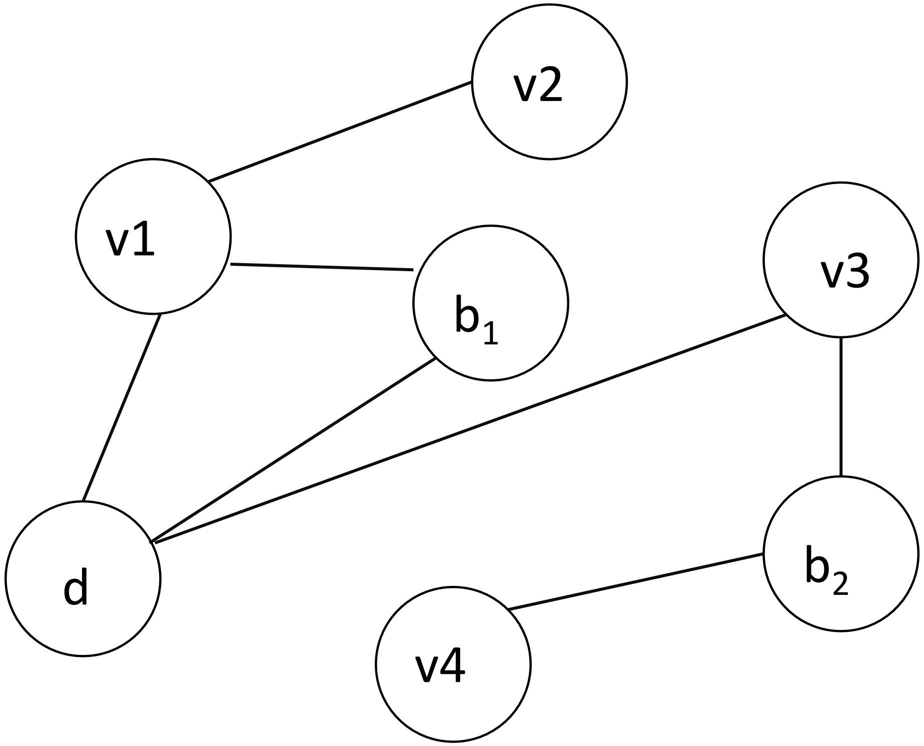
Interference graph

- **Assume three colors**
  - EAX 🟧
  - EBX 🟦
  - EDX 🟨



Interference graph

- **Assume two colors**



Graph coloring

- **Can we *efficiently find a coloring* for a given graph?**
- **Can we compute the *minimal number of colors* required to color a given graph?**
- **What can we do if *there are not enough registers*?**
  - I.e., for some number K we cannot find a coloring with K colors, and we cannot increase K

Graph coloring

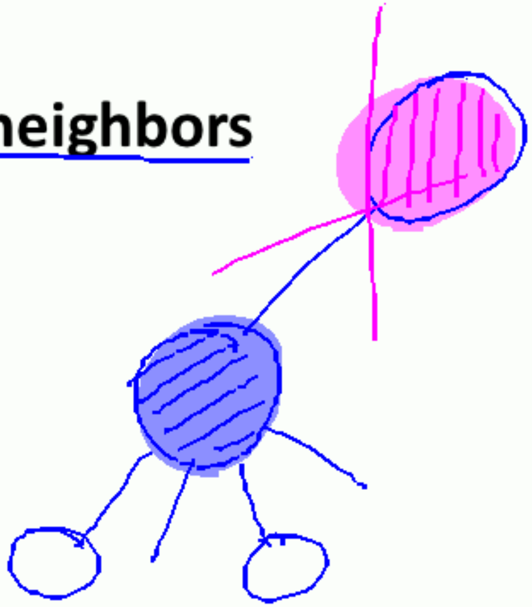
- **Unfortunately a hard problem**
  - K=2 special case…
  - For K > 2
  - Is a graph G K-colorable? NP-complete
- **Better bounds for special graphs – but interference graphs rarely have these special properties**
  - Cycles, chordal graphs, ladders, …

Graph coloring

- **For K > 2**
- **Kempe's algorithm (1879)**
- **Phase 1: Remove a node if it has K-1 or fewer neighbors**
  - Such nodes can later be colored w/o problems
  - Push on a stack when removing
  - Remove edges connected to node
  - Remove …

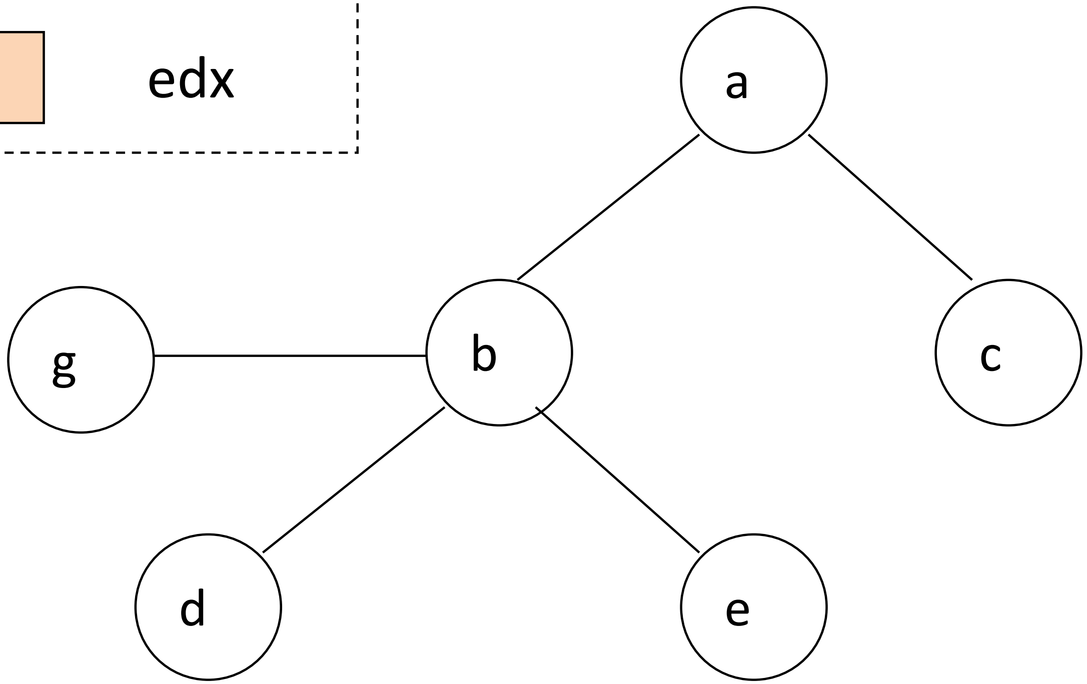
… until there are K nodes (hope)

  - Then each node has < *K* neighbors
  - (alternative formulation): … until there is one node
- **Phase 2: Color graph**
  - Look at neighbors – find free color



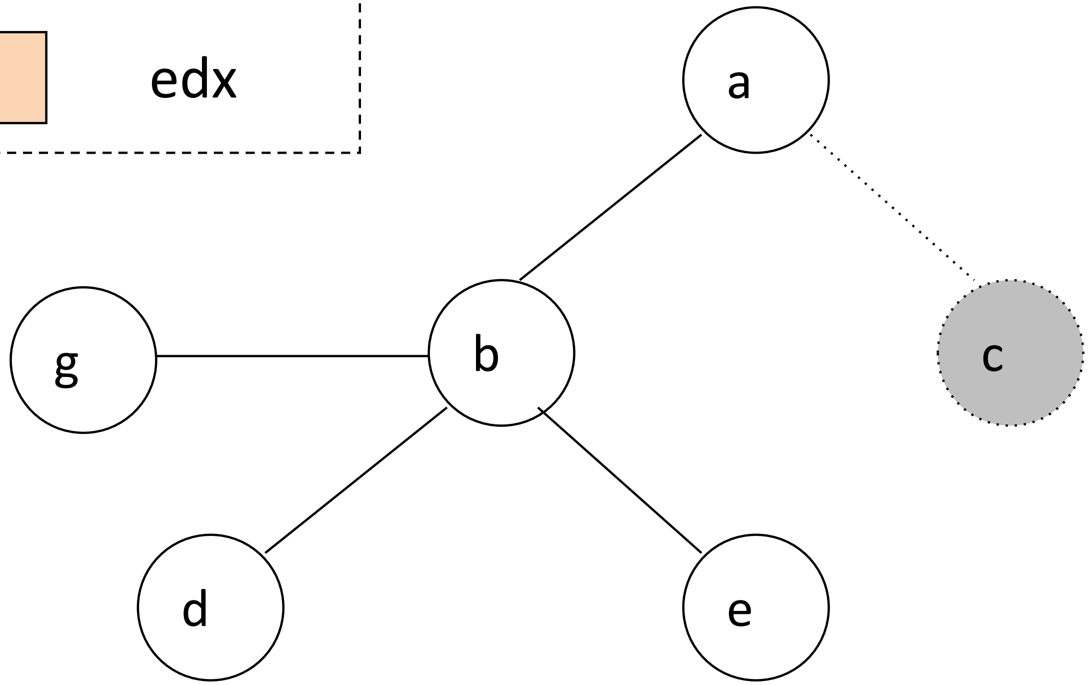
Example

Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

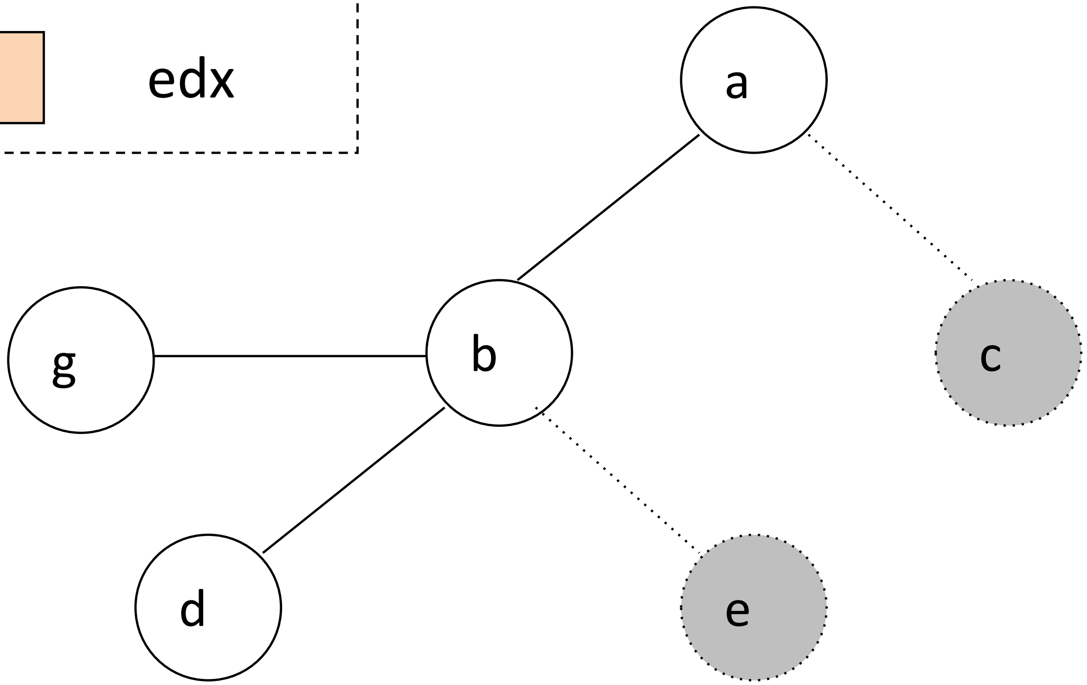
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

c

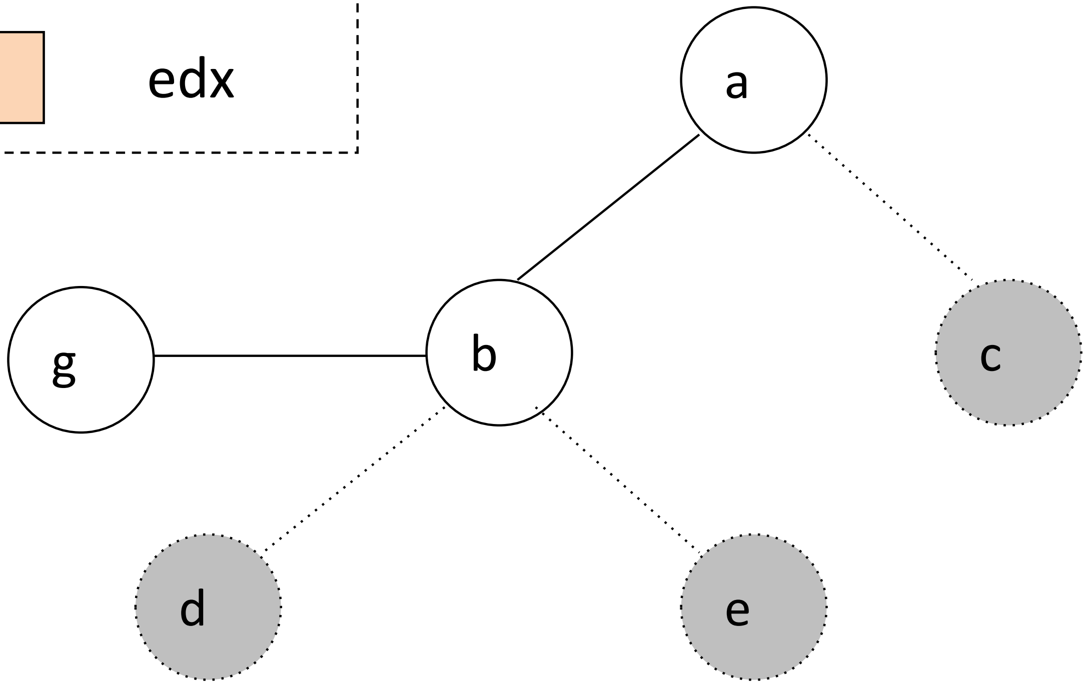
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

e
c

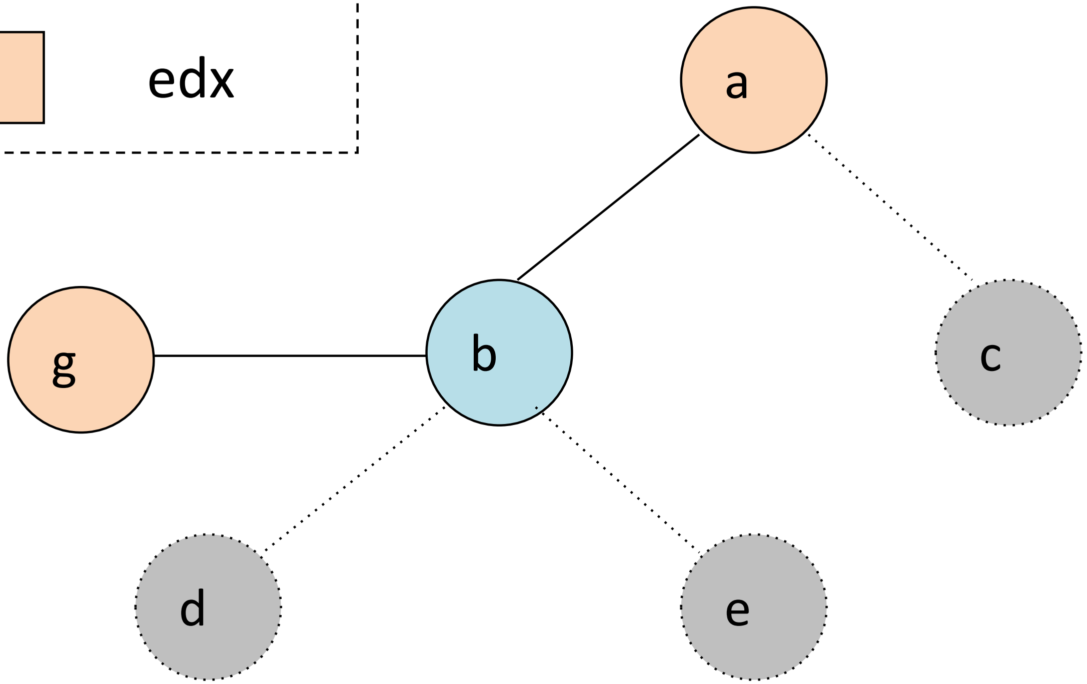
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

d
e
c

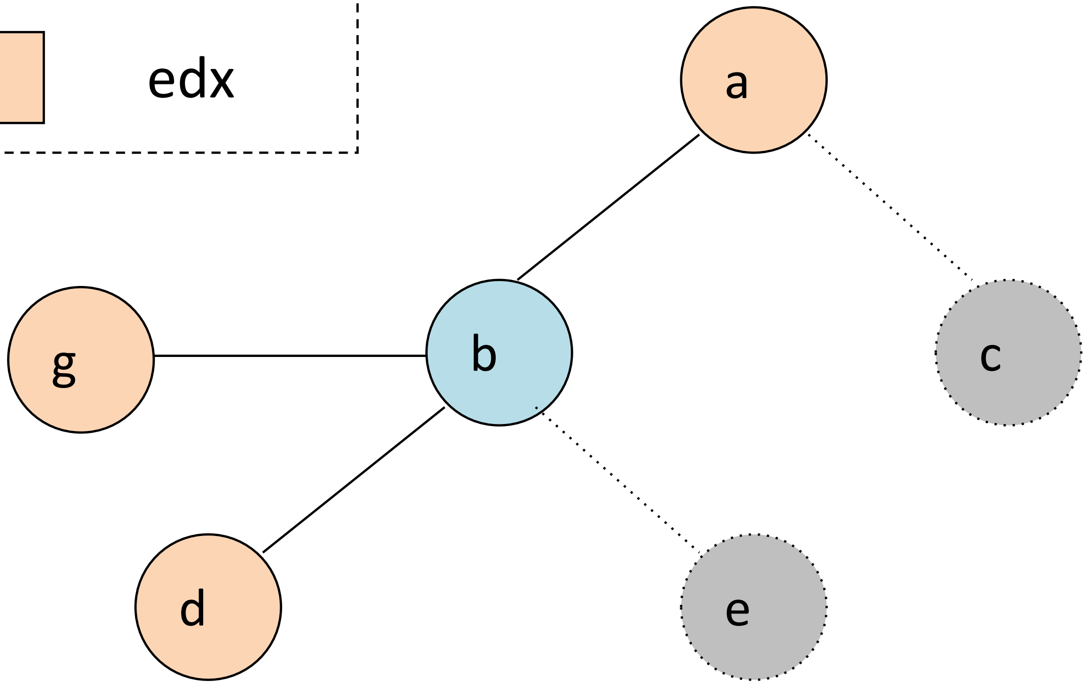
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

d
e
c

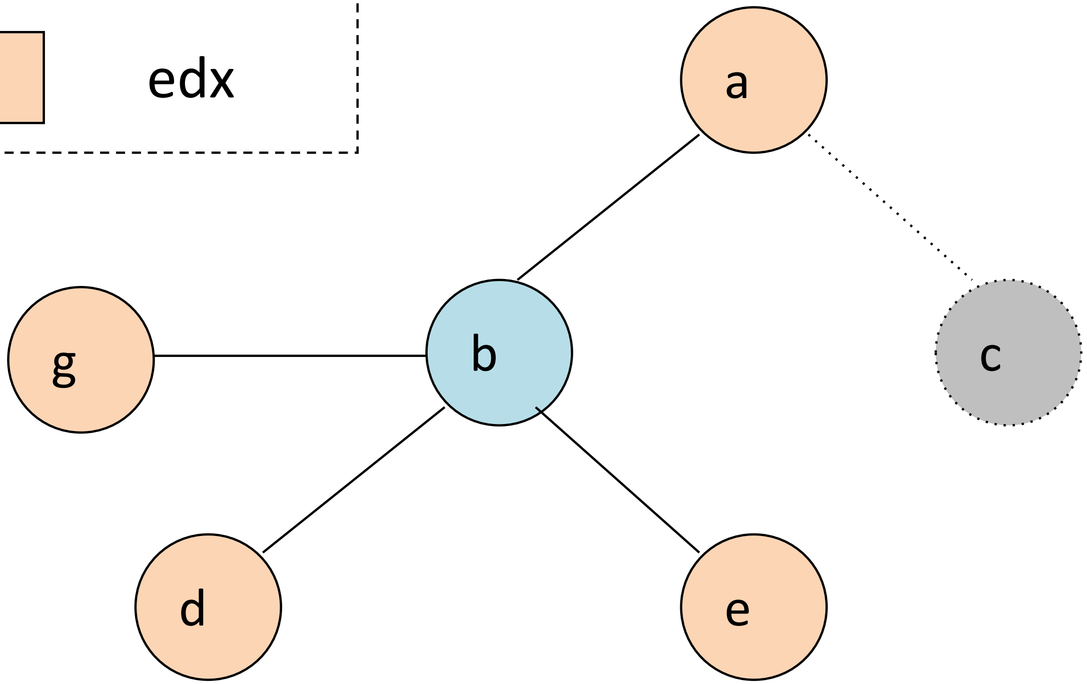
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

e
c

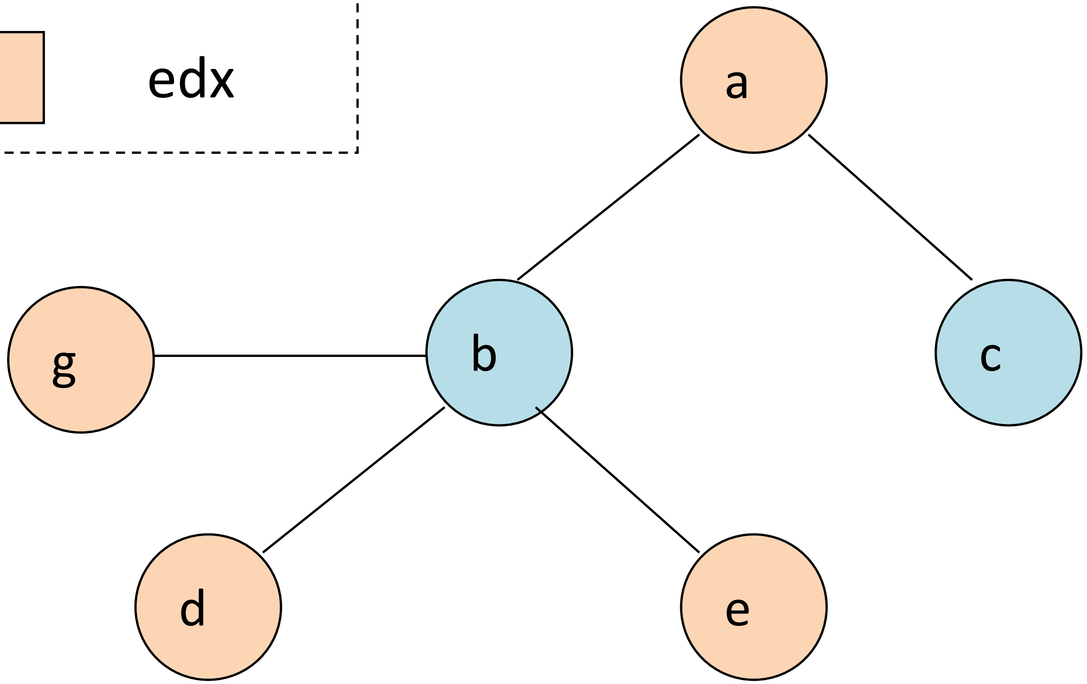
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

c

Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

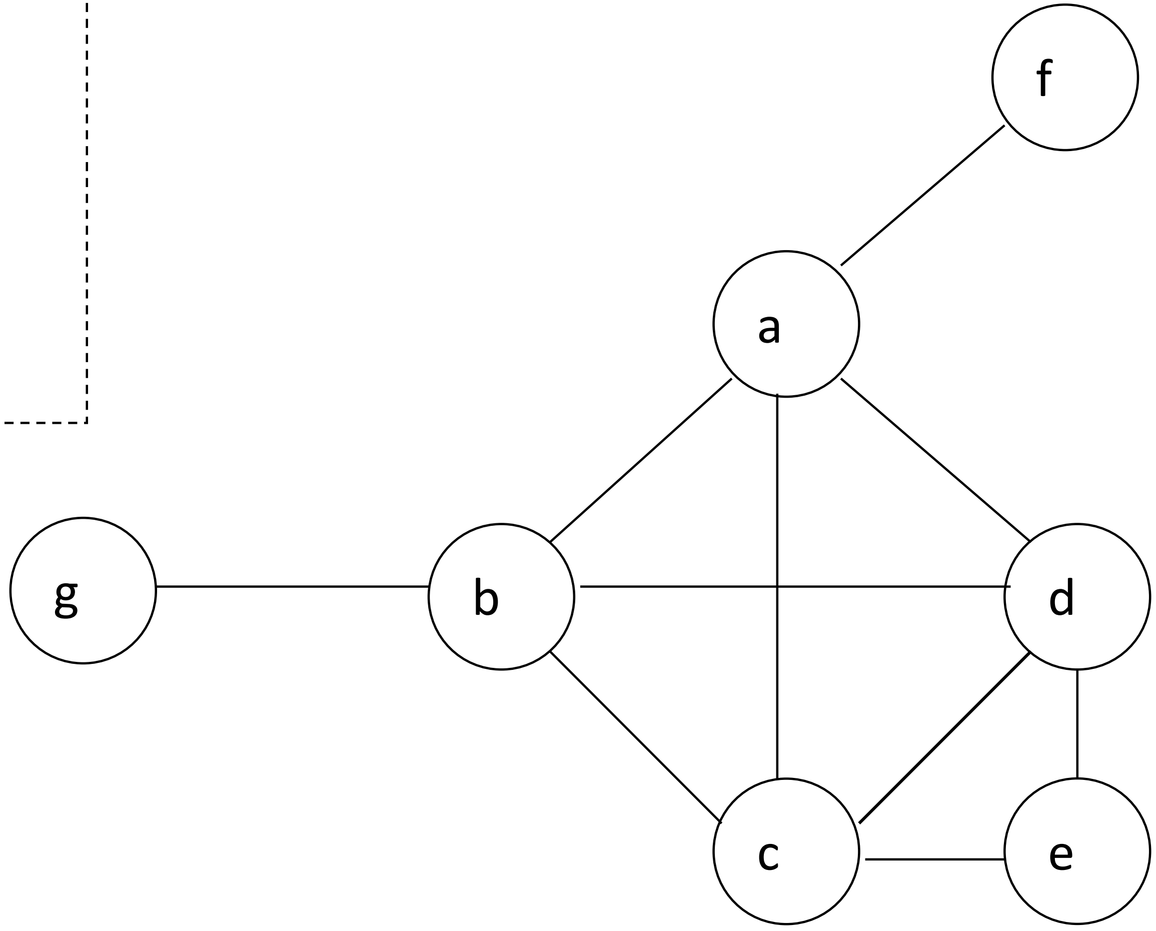
Graph coloring

- **Kempe's algorithm (1879), for K > 2**
- **Phase 1: Remove a node if it has K-1 or fewer neighbors**
  - Such nodes can later be colored w/o problems
  - Push on a stack when removing
  - Remove edges connected to node
  - Remove …

… until there are K nodes – optimistic

  - Not guaranteed to succeed
  - Can also stop with a graph such that each node has ≥ K neighbors

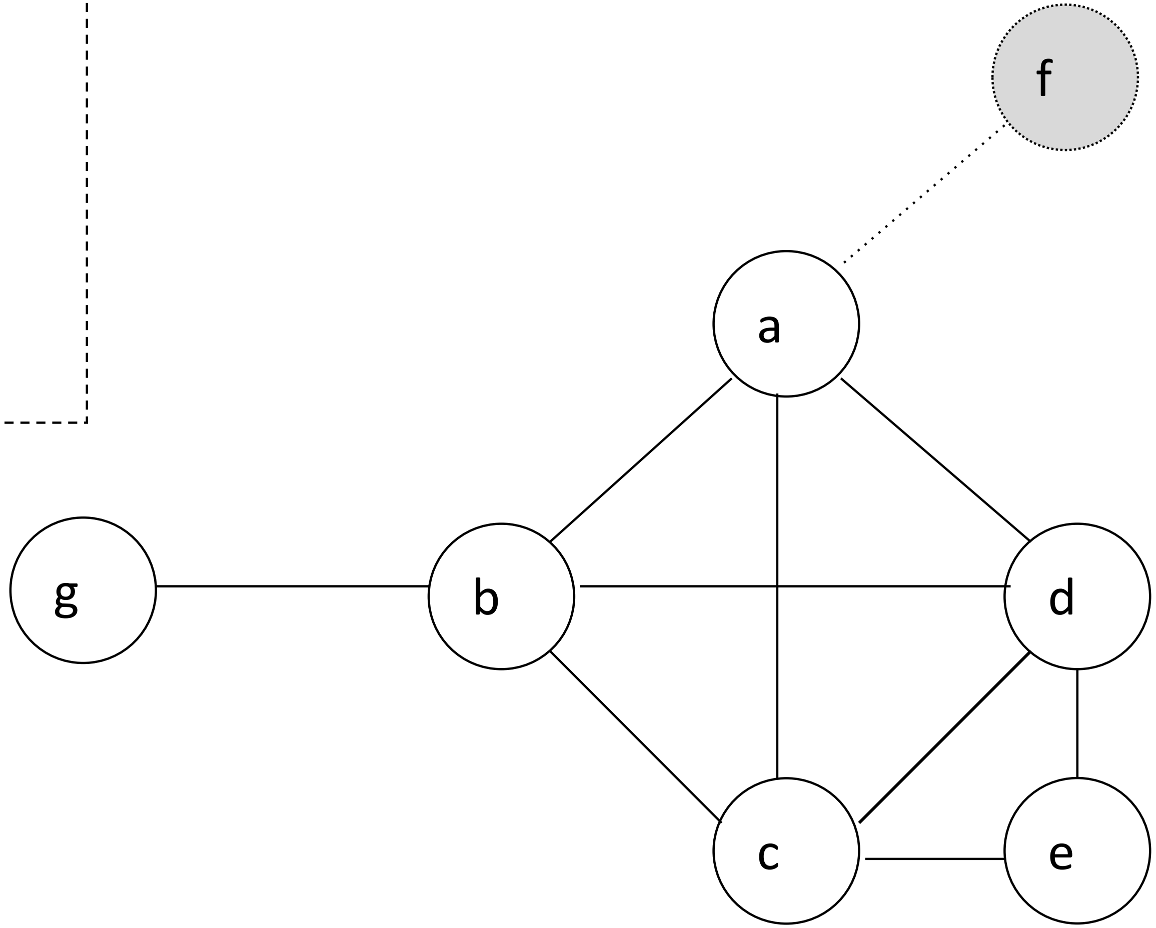
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

f

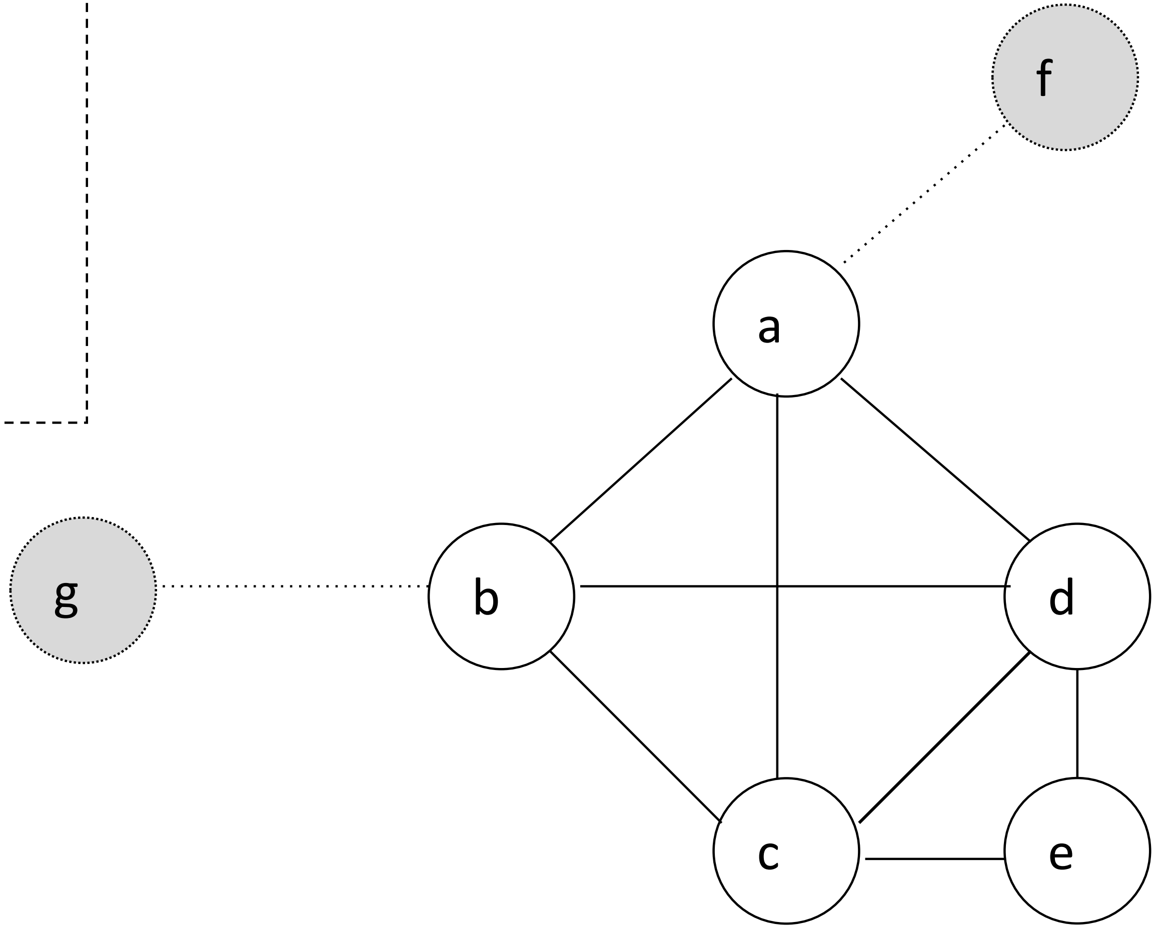
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

f

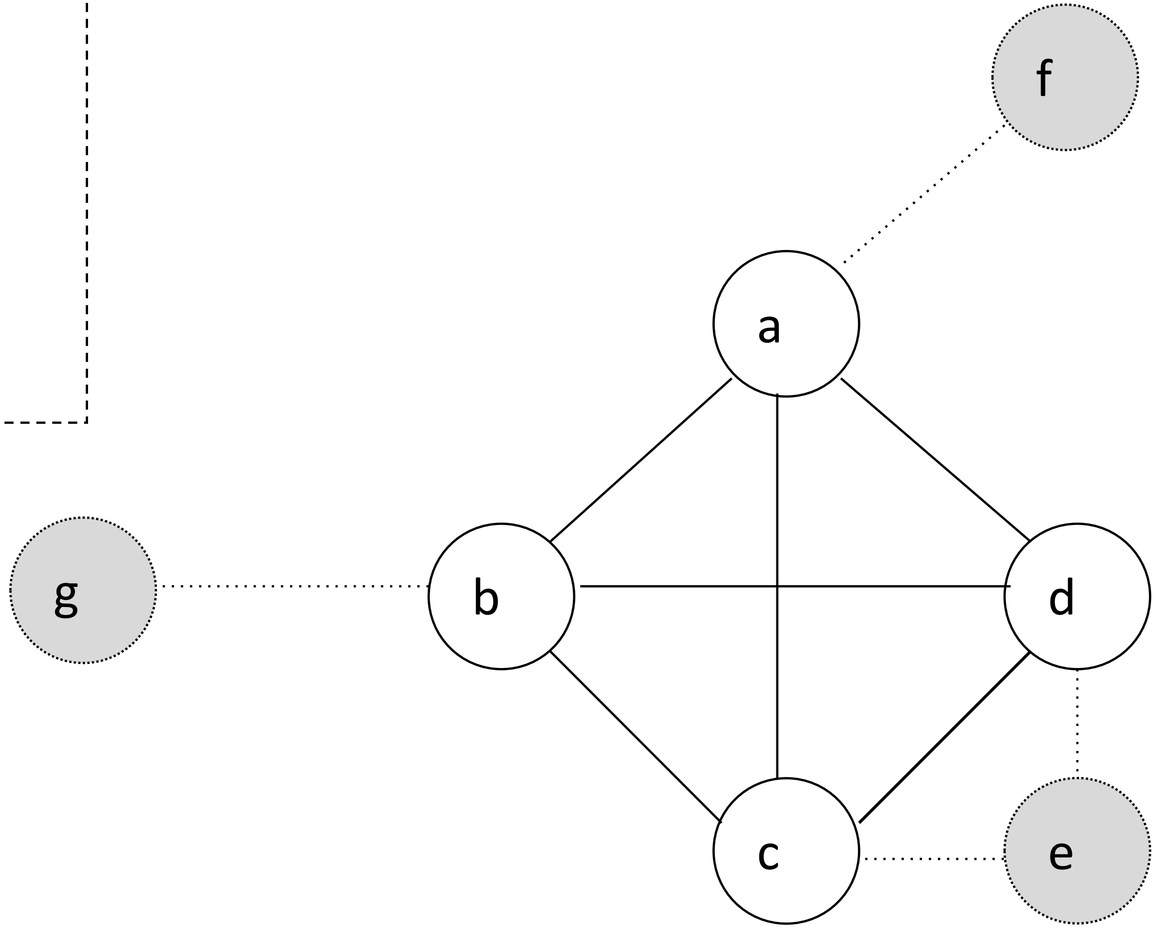
Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

g
f

Color	Register
🟩	eax
🟦	ebx
🟧	edx



Stack:

e
g
f

Graph coloring

- **Kempe's algorithm removes nodes with < K edges**
  - This step is called *simplification*
- **Simplification either ends with an empty graph or a graph such that each node has ≥ K edges**
  - Now we have to do something
    - Either try out all possible K-colorings
    - Graph surgery

Graph surgery

- **(If all nodes have ≥ K neighbors)**
- **Idea: Pick a node and remove it**
  - We discuss later how to pick a node (heuristics)
  - Node is *spilled*: won't get a register and is assigned to memory
  - Remove until no node has ≥ K neighbors
- **Color (remaining) graph**
  - Color nodes pushed on stack in Phase 1

Outline

- 9.1 Introduction
  - Live range
  - Interference graph
- 9.2 Graph coloring
- 9.3 Live range spilling
- 9.4 Live range splitting

9.3 Spilling

- **Given a graph that has been simplified (but is not empty)**
- **Pick a node and remove this node *and* all its edges from the graph**
  - The live range represented by this node is not allocated a register
  - It is "spilled" – the home location is in memory
- **We discuss later how to pick a node**

Graph coloring, revised

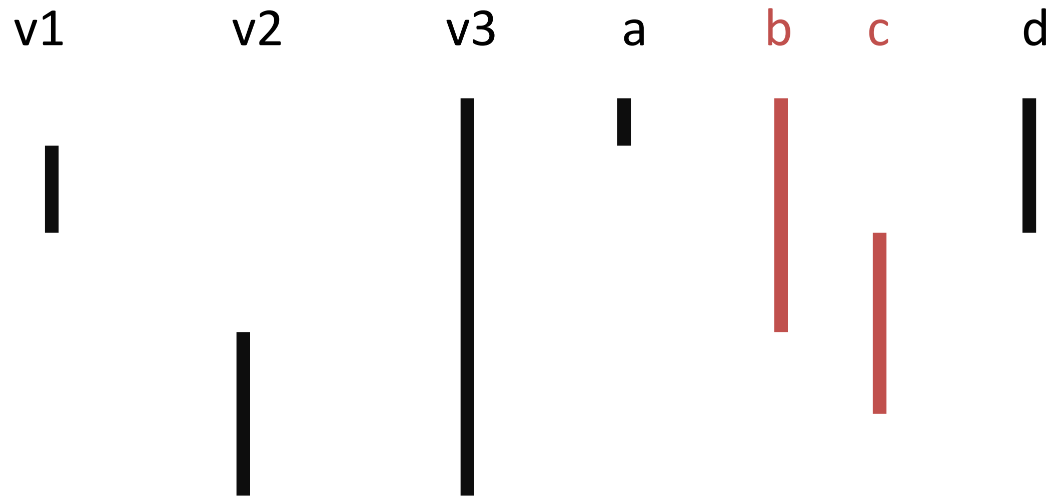
- **Phase 1: Remove a node if it has K-1 or fewer neighbors**
  - Push on a stack when removing
  - Remove … until all nodes have ≥ K neighbors or the graph is empty
- **Phase 2: (If all nodes have ≥ *K* neighbors): Pick a node and remove it with all its edges**
  - Continue simplification
    - Can’t continue as all nodes have ≥ *K* neighbors: Pick a node and remove it
- **Phase 3: (Graph is empty): Color graph**
  - Pop node from stack
  - Assign color

Spilled live ranges

- **A spilled live range resides in memory**
  - Create temporary, usually stored in the activation record

- **What should we do with a spilled live range when generating code?**

```
v1 = a + b
c = v1 + d
v2 = b * 2
v3 = c + 5
```



b, c are spilled

Spilled live ranges

- **Target machine (x86) requires that at least one operand resides in a register**
  - The other one can be supplied by memory

```
mov a, R2
add b, R
```

a be in this register

v1 goes into this register

- **Spilled live range ⇒ operand in memory**
  - `v1 = a + b`: constraint that `b` must be in memory

eax

ebx

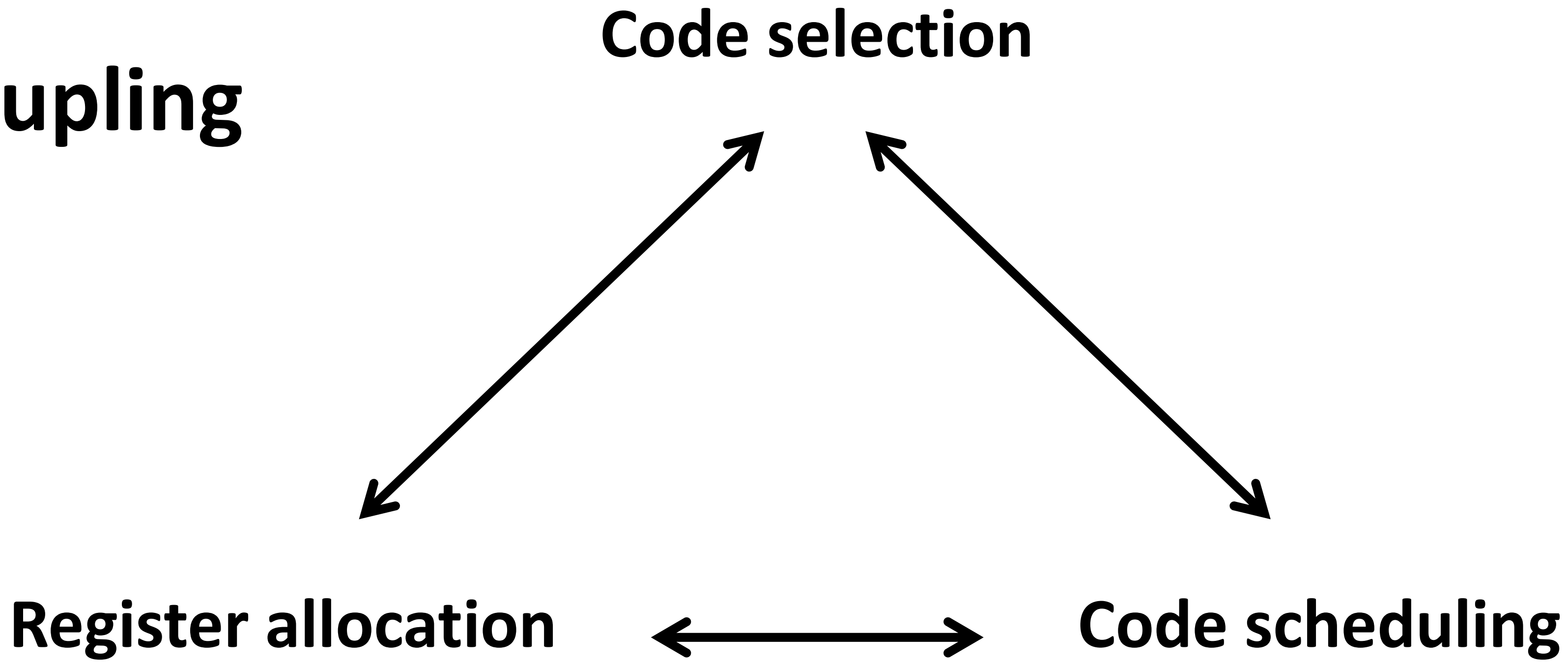
Lucky if a is dead otherwise issue a MOV

Spilled live ranges

- **Target machine (x86) requires that at least one operand resides in a register**
  - The other one can be supplied by memory

- **Spilled live range ⇒ operand in memory**
  - `v1 = a + b` : constraint that `b` must be in memory
  - OUCH
  - Now the register allocator determines instruction selection
    - `a` must reside in register R, R must hold `v1`
    - `a` must be dead or must be copied
  - Must run register allocation prior to instruction selection

Phase coupling



Code selection

Register allocation

Code scheduling

- Code selection depends on code scheduling
- Code scheduling depends on register allocation
- Register allocation depends on code selection
- Close coupling of different code generator phases

Spilled live ranges

- **Target machine (x86) requires that at least one operand resides in a register**
  - The other one can by supplied by memory

- **Spilled live range ⇒ operand in memory**
  - `v1 = a + b` : constraint that `b` must be in memory

  - And what if `a` is spilled as well?
    - Same problem for RISC machine: All operands must be in a register

Spilled live ranges

- **Code generator may need a register for a spilled live range (… or for two live ranges, or for destination if destination live range is spilled)**
- **Option 1: Spare registers**
  - Code generator keeps spare registers that are not allocated by register allocator
    - 1 register enough on IA32, 2 needed on RISC machine
    - Depends… not all registers may be created equal
  - Register allocator finds (K-2)-coloring
    - or (K-1)-coloring
    - Maybe OK on a RISC with 32 or 64 registers

Option 2: More graph surgery

- When spilling a node, introduce a new temporary, rewrite the IR and start over
- Example

```
    v1 = a + b
```

with `b` spilled. Introduce a temporary `temp101`, stored at (say) `ebp+40`

- Rewrite to

```
temp101 = *(ebp + 40)
v1 = a + temp101
```

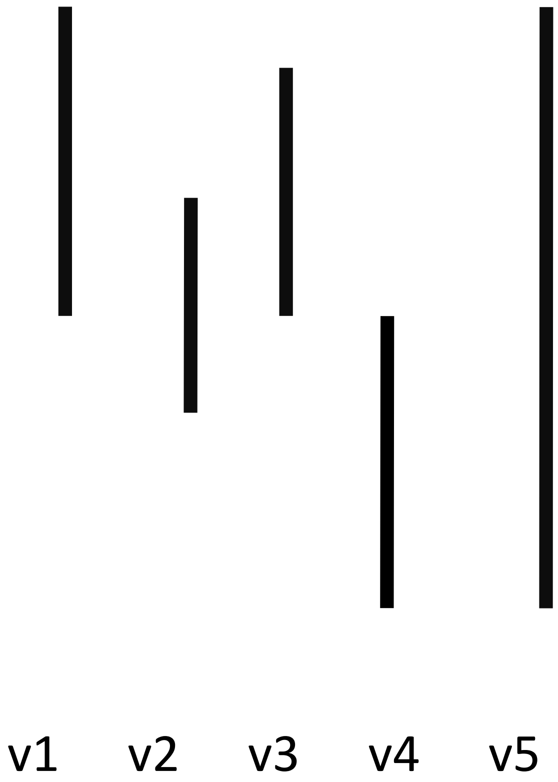
- *(ebp+40): shorthand for “load temporary”

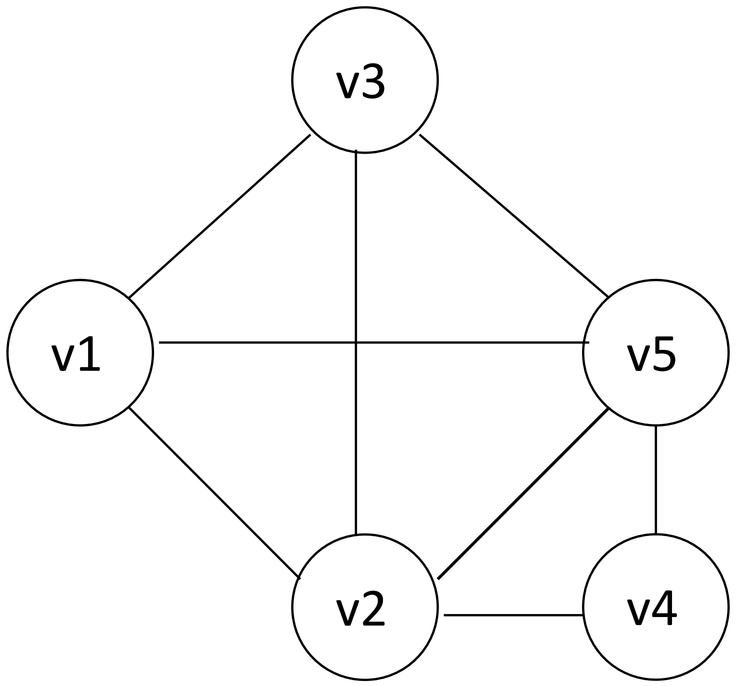
Temporary live ranges

- **Live range of temporaries is very small**
  - Just one instruction
- **Graph should be easier to color**
  - Temporary has smaller number of edges than spilled live range
  - A different temporary is used for each use of the spilled variable
- **Rebuild interference graph and start over**
  - And if the graph still cannot be K-colored: Pick another node for spilling
  - As long as number of registers > number of (asm) operands the process terminates with a legal K-coloring

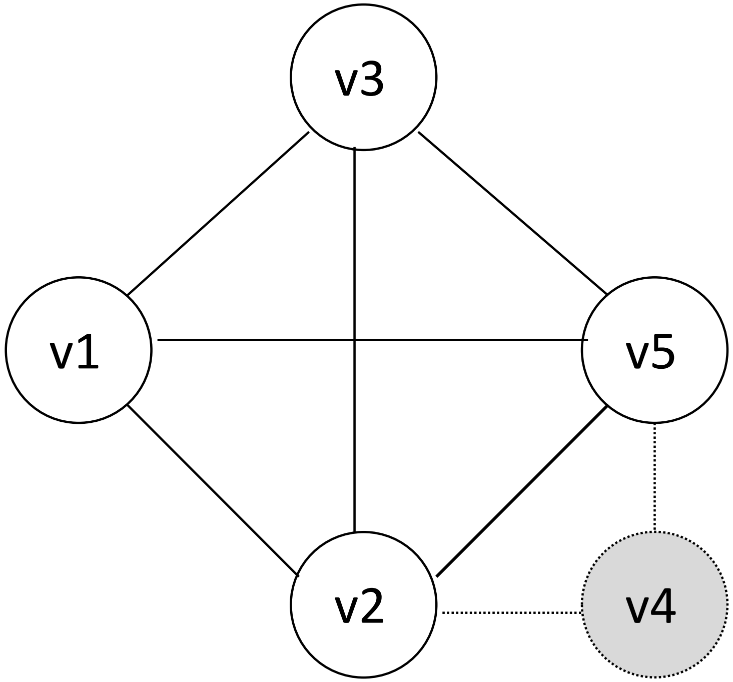
Example

- **Consider an interference graph with 5 variables**





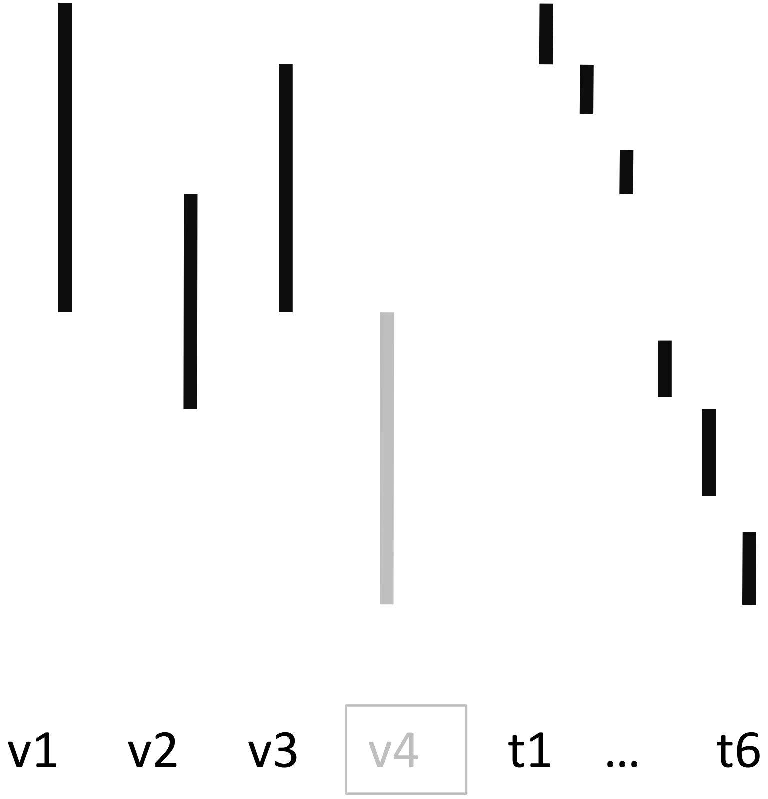
Example with 3 registers

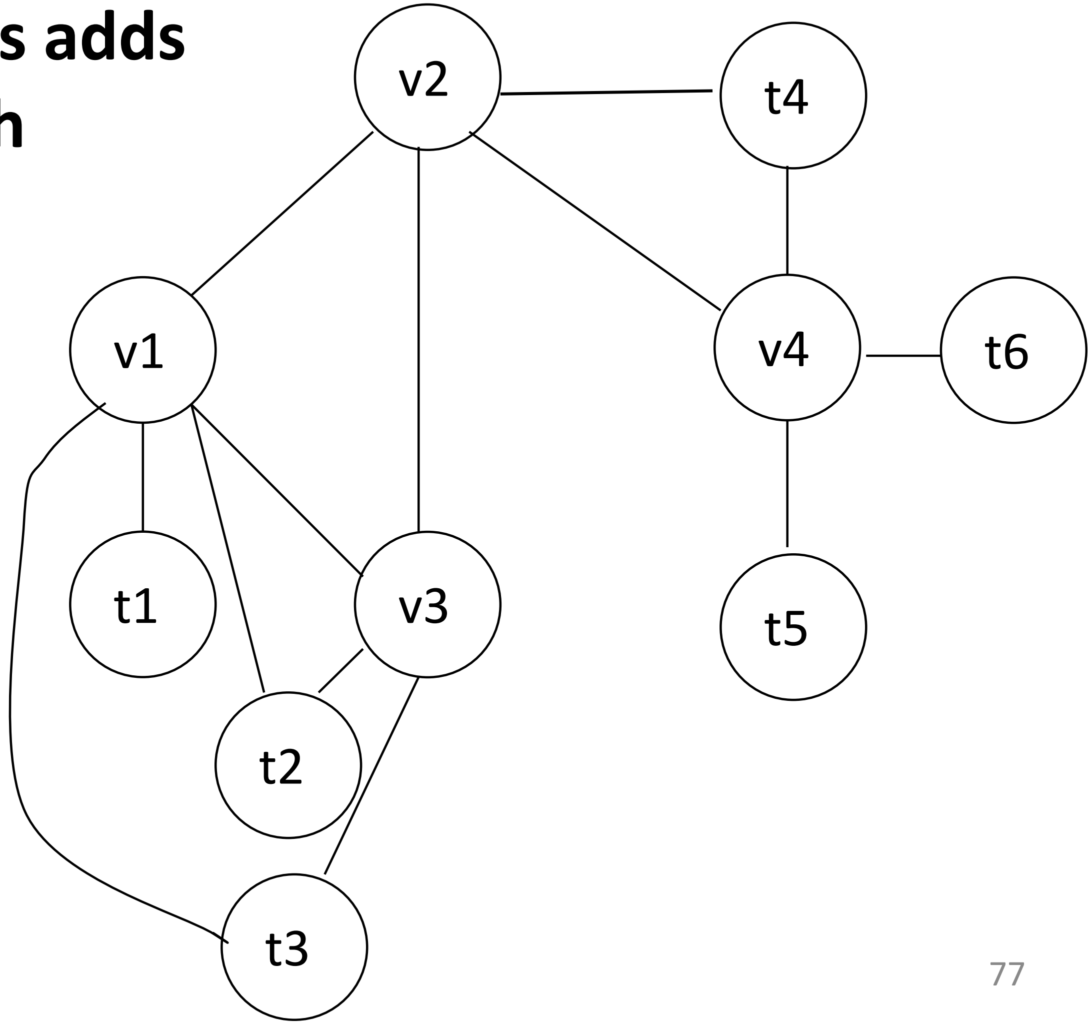


- **v4 is removed by simplification**
- **All remaining nodes ≥ 3 edges**
- **Let v5 be spilled**

Interference graph reconstruction

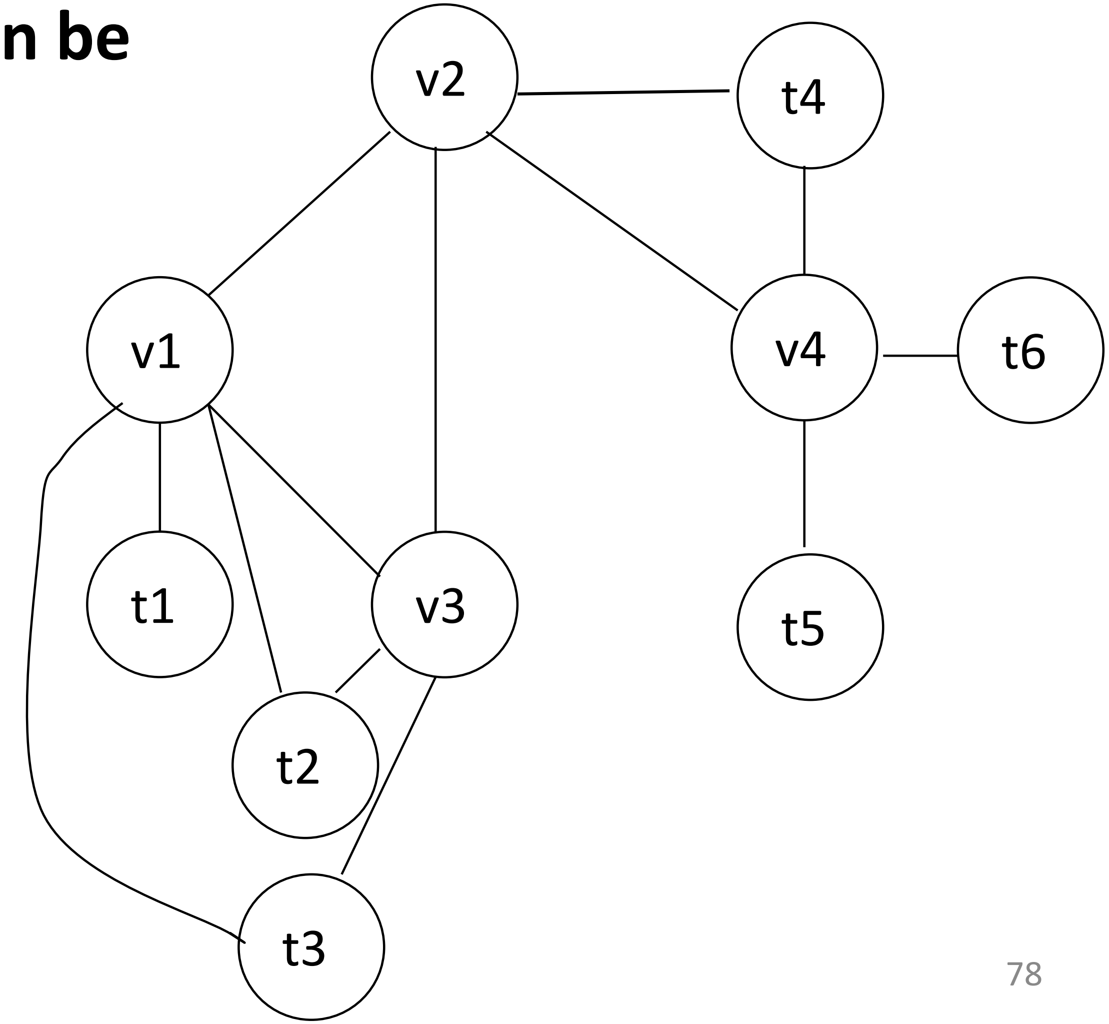
- **Introduction of temporaries adds nodes to interference graph**





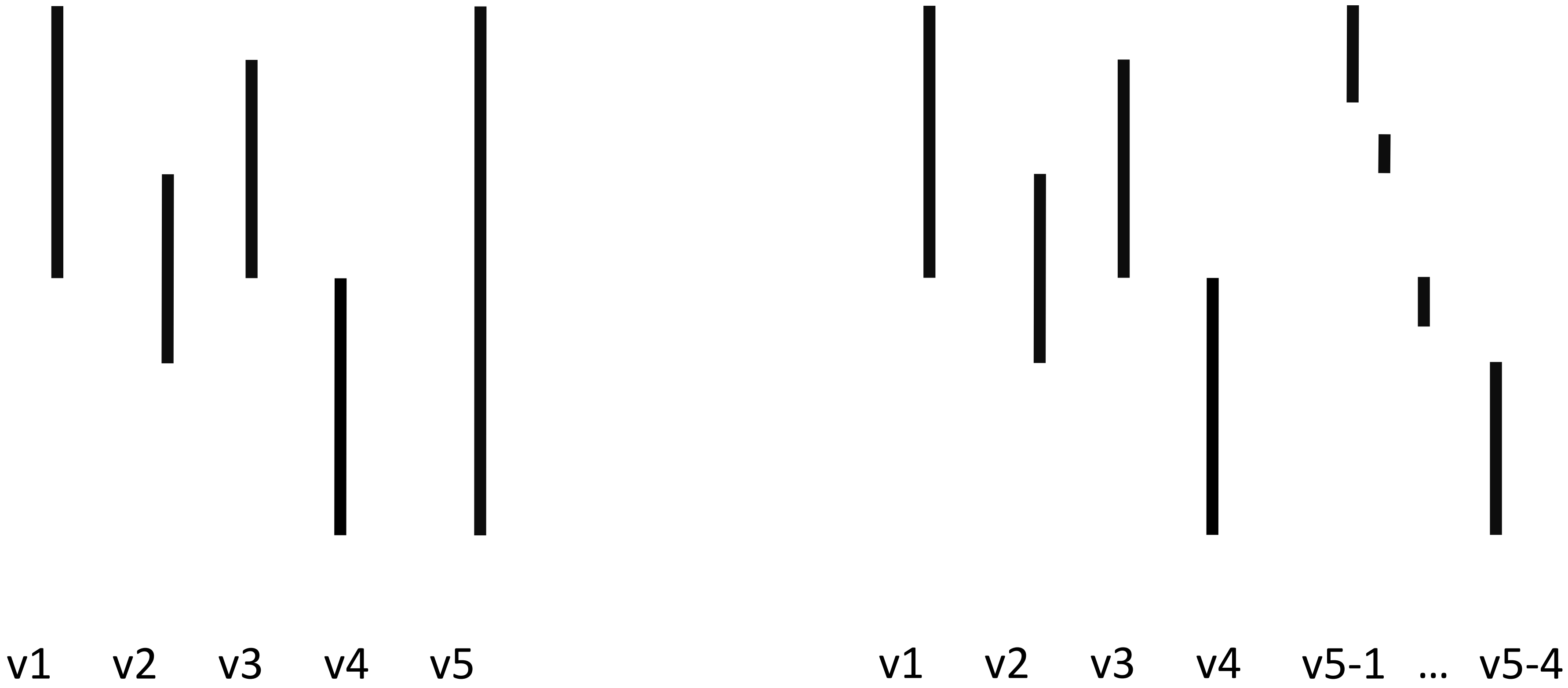
Another attempt to color

- **New interference graph can be colored (K=3)**

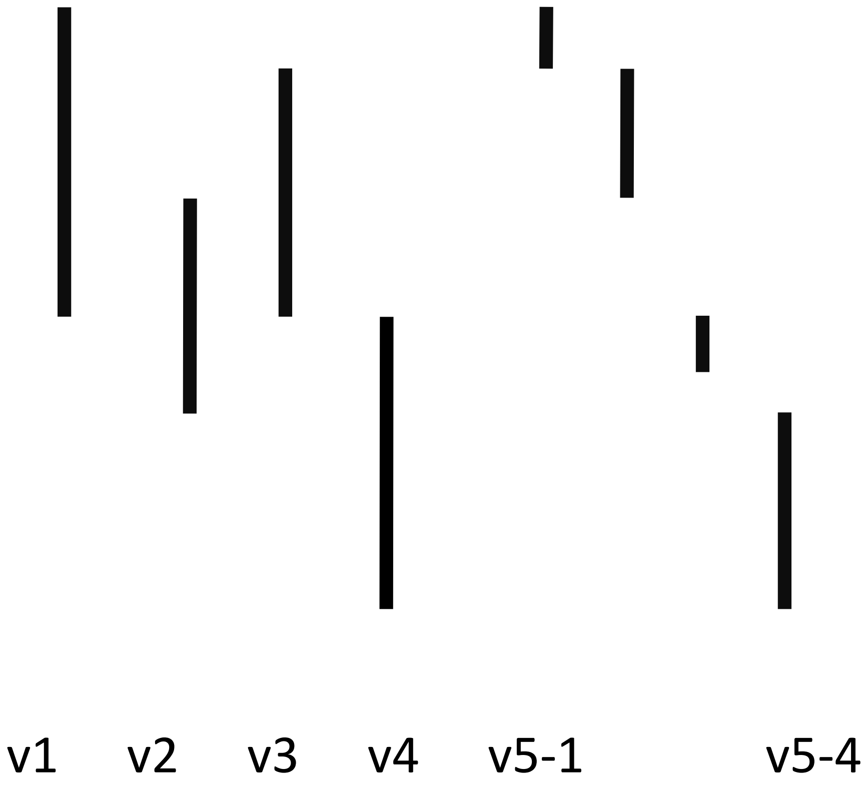


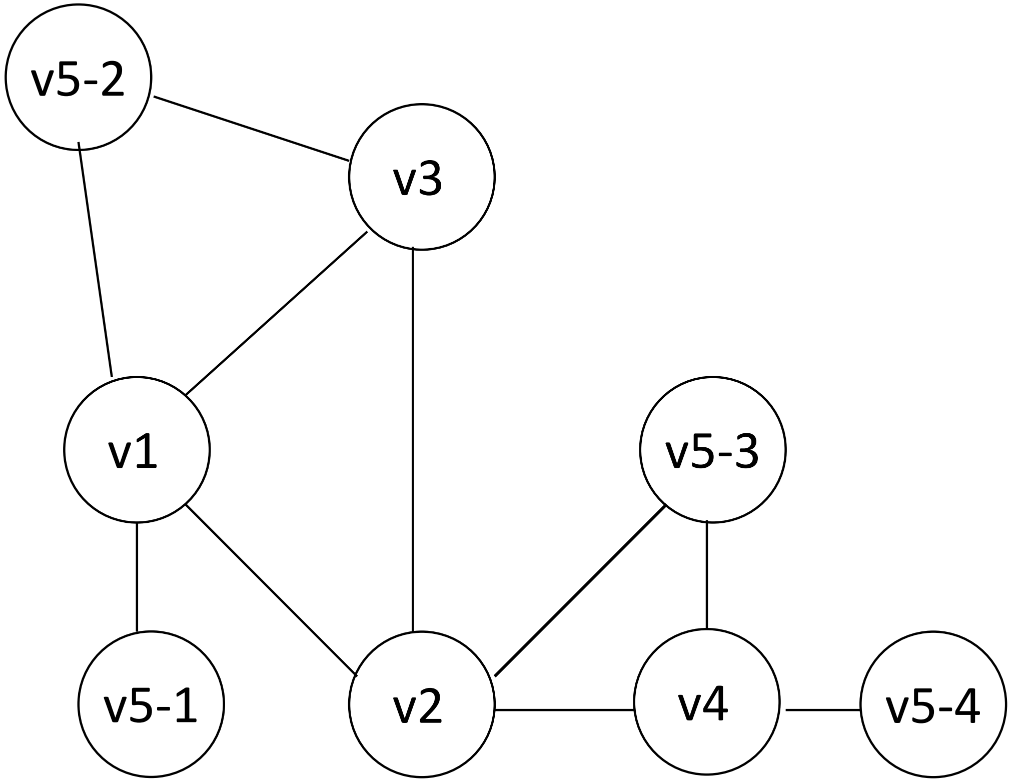
More graph surgery

- A (better?) approach is to *split* the live range



A new interference graph





9.4 Splitting

- **Splitting reduces number of instructions that are needed to load (store) “temporary” variables**
  - Variables that are spilled to memory
- **Which live ranges to split?**
- **Where to split them?**

Spilling and splitting

- **Two techniques to reduce register pressure**
- **Could be done in either order**
  - Splitting in the limit like spilling (separate live range for each use)
- **Need to discuss spilling decisions before splitting**

Graph coloring, revised

- **First: Simplification**
  - (Kempe’s algorithm)
- **(All nodes have ≥ *K* neighbors): Pick a node and remove it with all its edges**
  - Continue simplification
    - Can’t continue as all nodes have ≥ *K* neighbors: Pick a node and remove it
- **(Graph is empty): Color graph**
  - Pop node from stack
  - Assign color

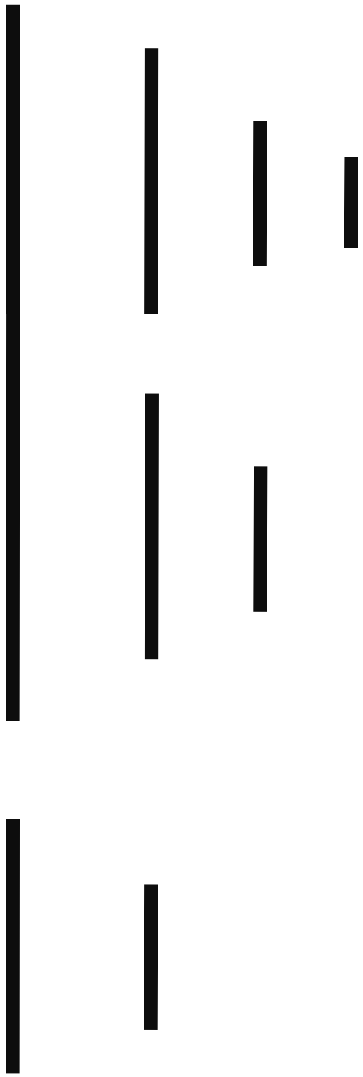
Picking the spill victim

- **A number of heuristics have been tried.**
- **Pick a node at random (Chaitin, 1982)**
- **Pick node with lowest spill cost estimate (Chow, 1983)**
  - How do we estimate spill cost?
- **Pick node with lowest use count**
- **…**

Estimating spill cost

- **Need to estimate how often a basic block is executed**
- **Use profile from past execution of program**
  - Input dependent?
- **Use profile of *current* execution**
  - Can be done in JIT (Just-in-time compiler)
  - Guess: past predicts the future

Estimating spill cost



Consider a well-structured program

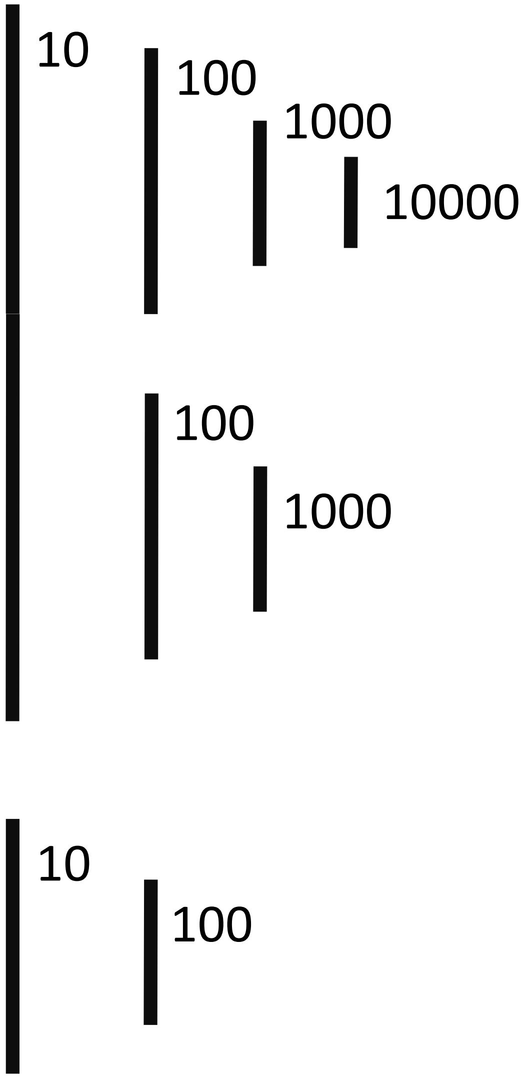
Bars indicate a loop

Profile from past execution may give us "trip count" (number of times a loop body is executed)

Estimating spill cost

- **Need to estimate how often a basic block is executed**
- **Use profile from past execution of program**
  - Input dependent?
- **Use profile of *current* execution**
  - Can be done in JIT (Just-in-time compiler)
  - Guess: past predicts the future
- **Guess by rule-of-ten: loops execute 10 times**

Estimating spill cost



In the absence of profile information we can guess: each loop is executed 10 times.

Extensions

- **Spill cost estimate can be extended to identify splitting candidates**

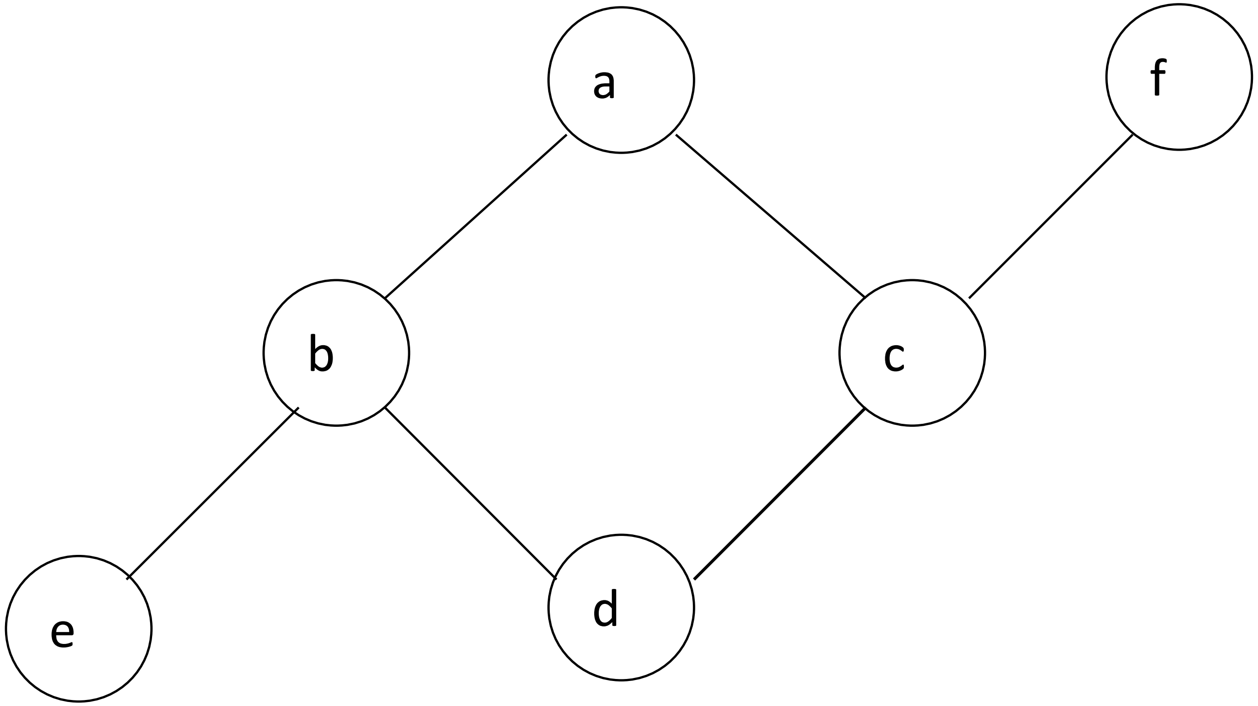
- **Don’t forget: interference graph rebuilt after each split decision**
  - Requires computation of live ranges!

9.5 Comments

- **Sometimes spills may not even be necessary.**

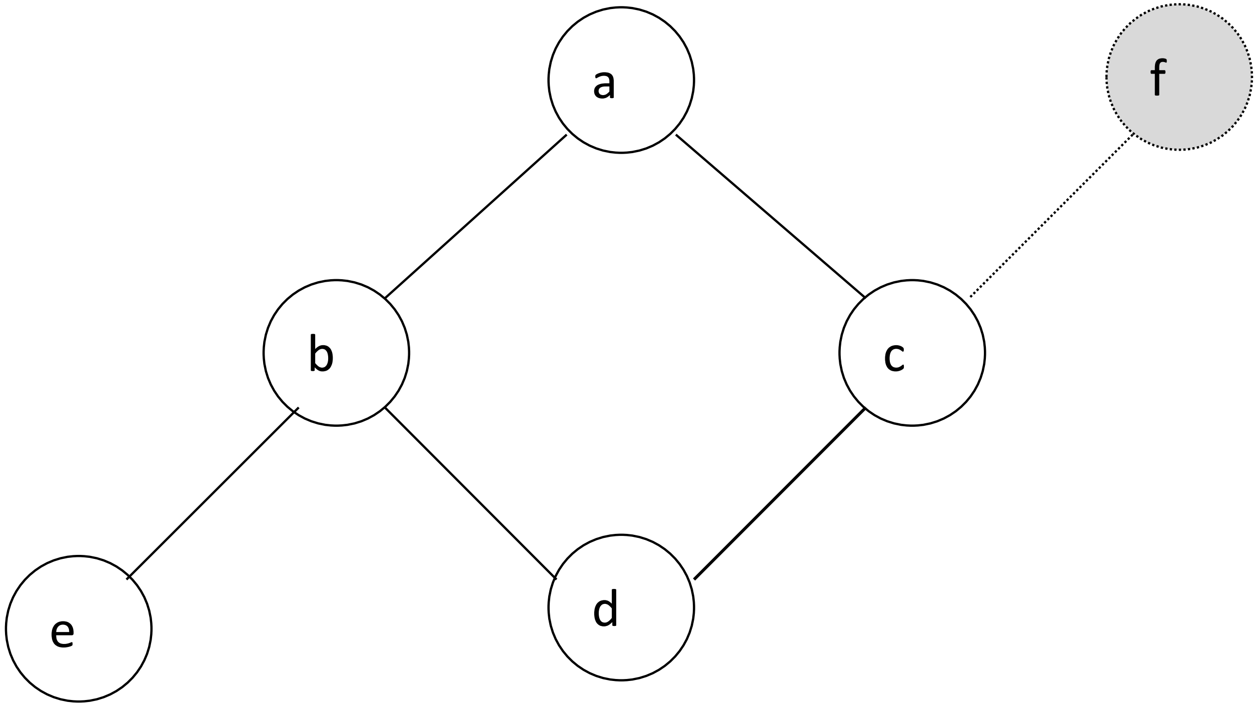
Example – 2 registers

Color	Register
🟦	eax
🟧	ebx



Stack:

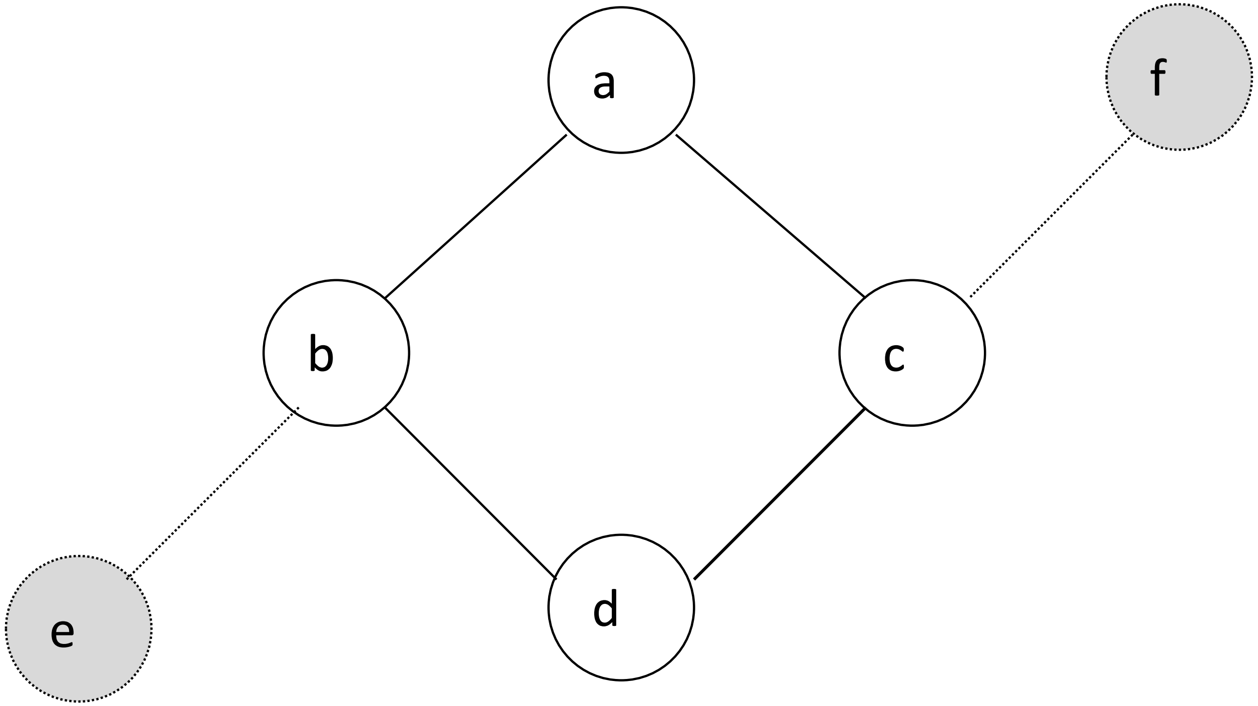
Color	Register
🟦	eax
🟧	ebx



Stack:

f

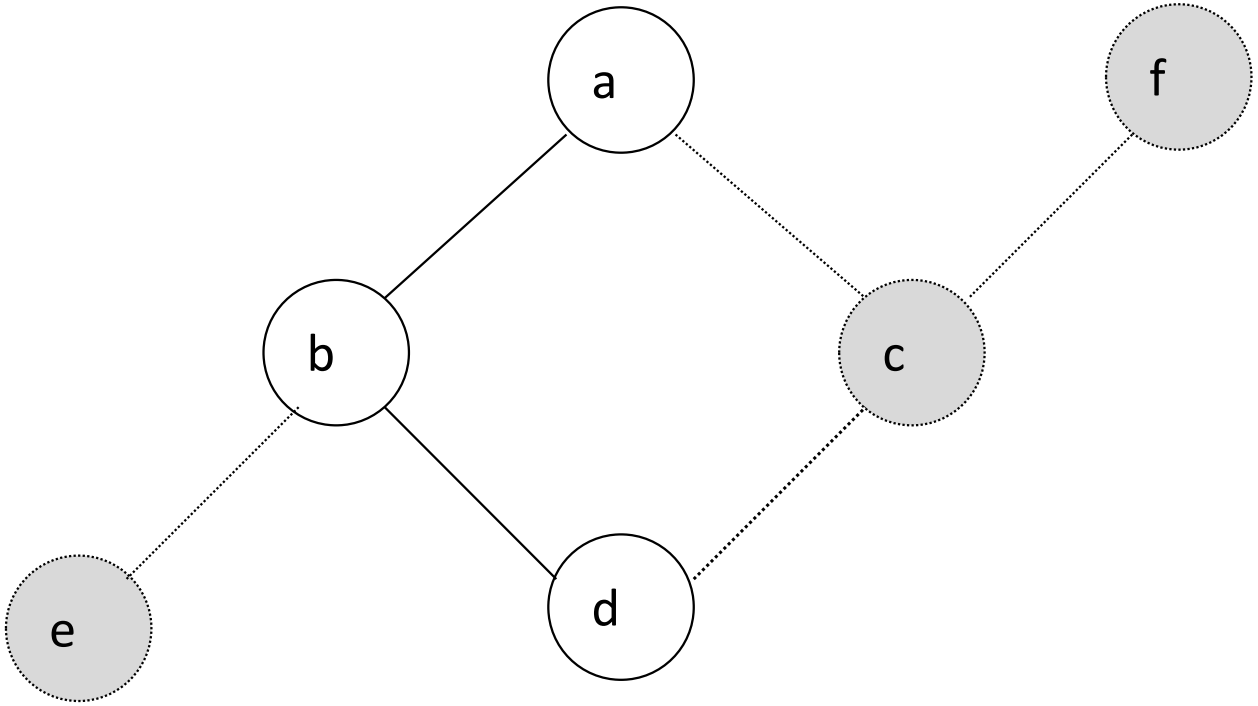
Color	Register
🟦	eax
🟧	ebx



Stack:

e
f

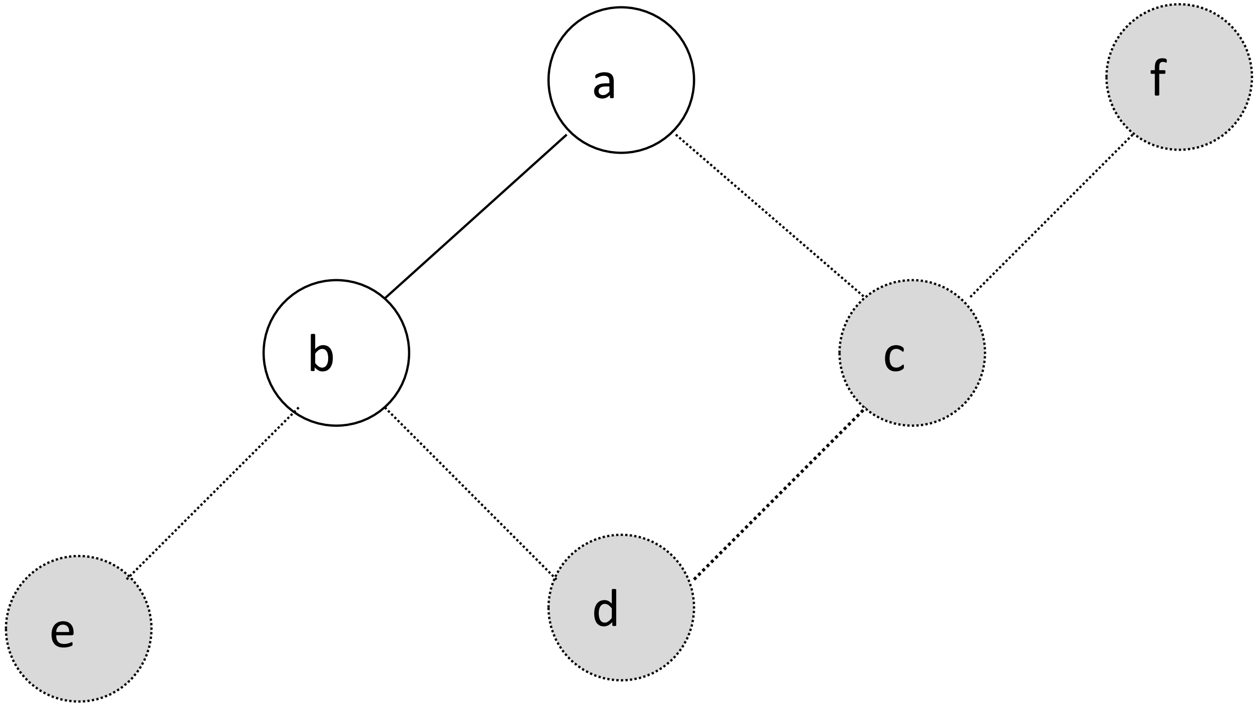
Color	Register
🟦	eax
🟧	ebx



Stack:

c
e
f

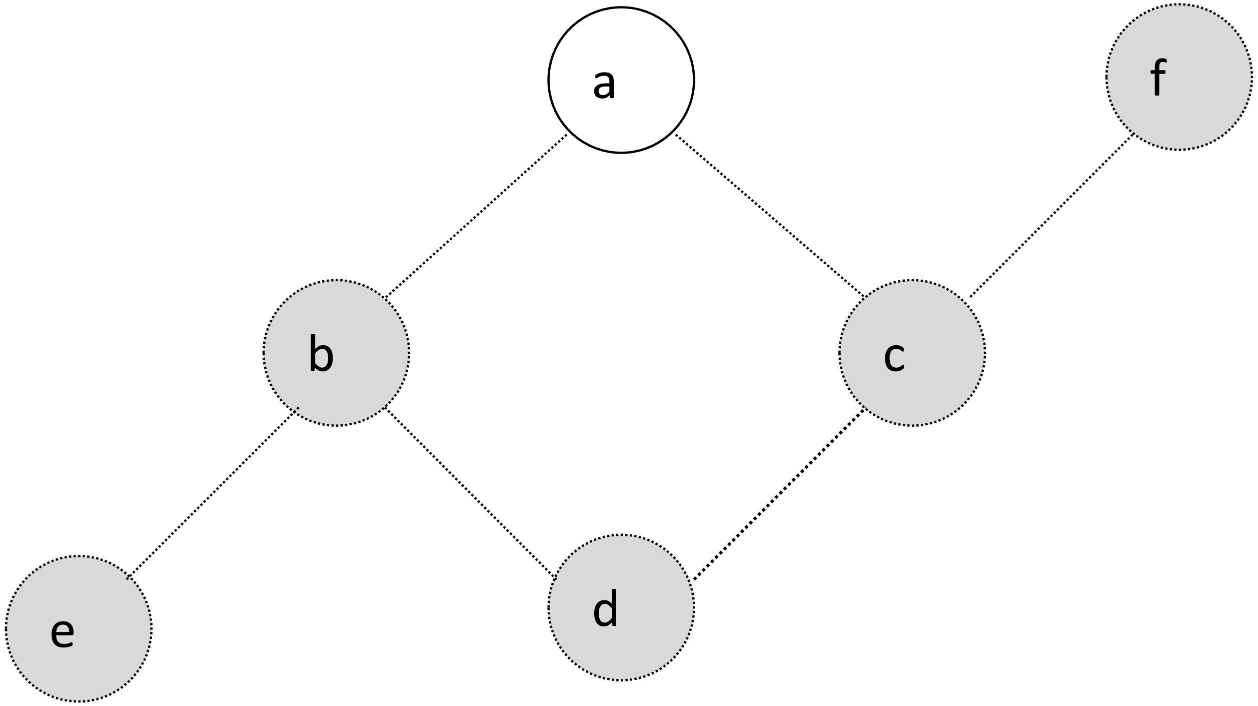
Color	Register
🟦	eax
🟧	ebx



Stack:

d
c
e
f

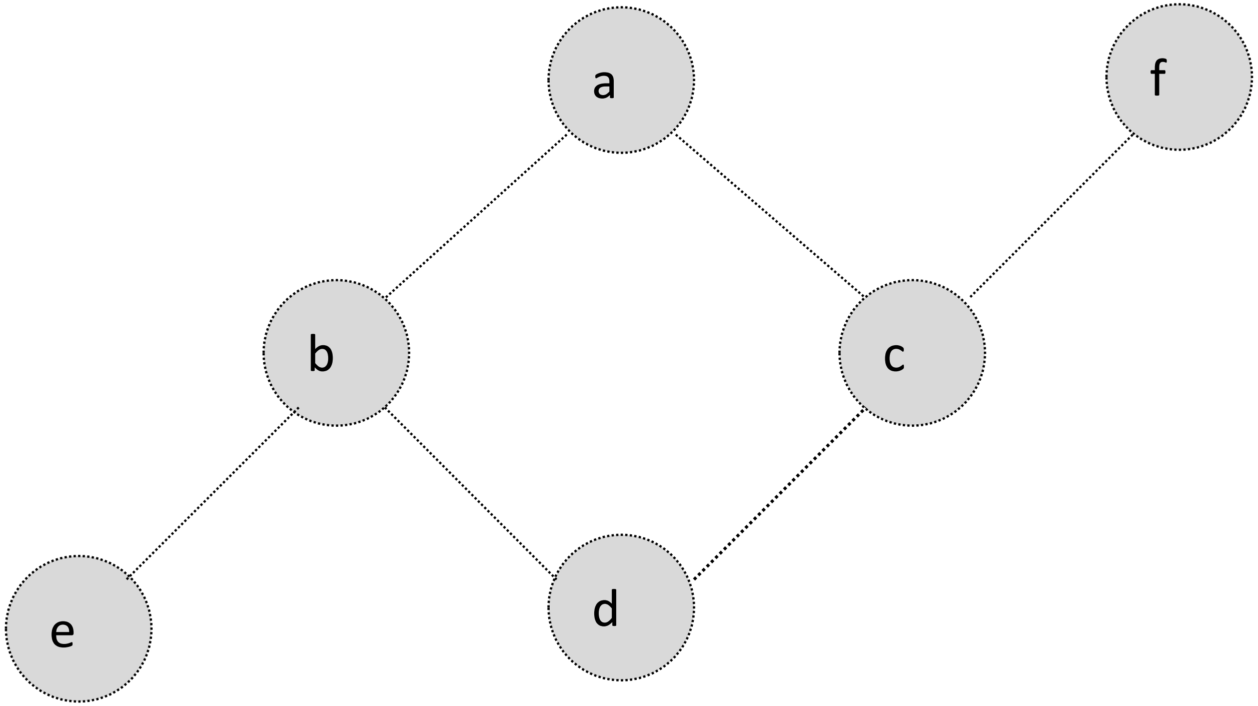
Color	Register
🟦	eax
🟧	ebx



Stack:

b
d
c
e
f

Color	Register
🟦	eax
🟧	ebx



Stack:

a

b

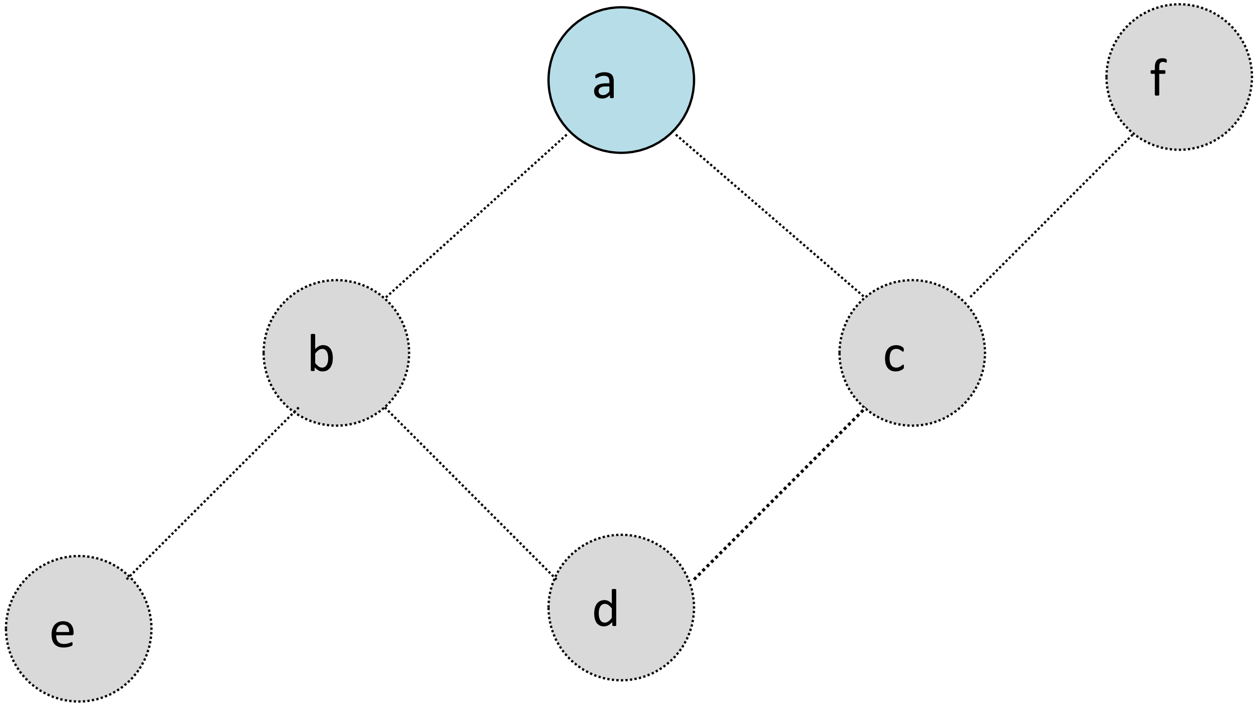
d

c

e

f

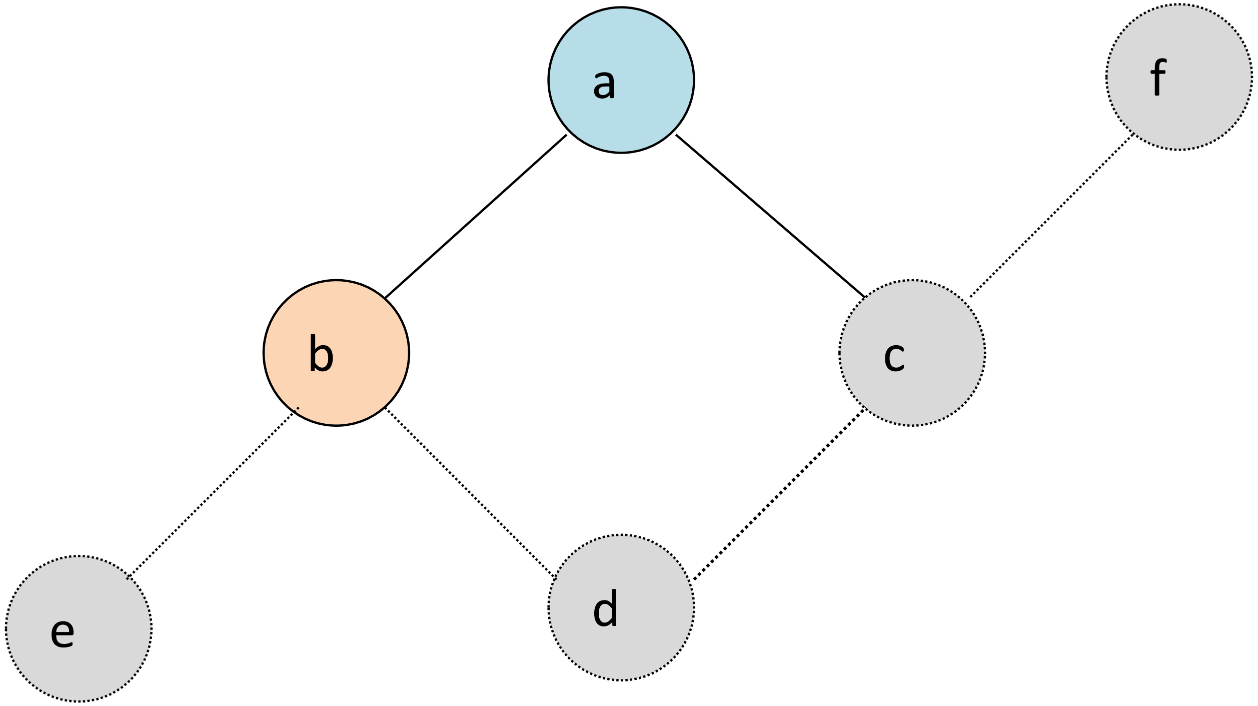
Color	Register
🟦	eax
🟧	ebx



Stack:

b
d
c
e
f

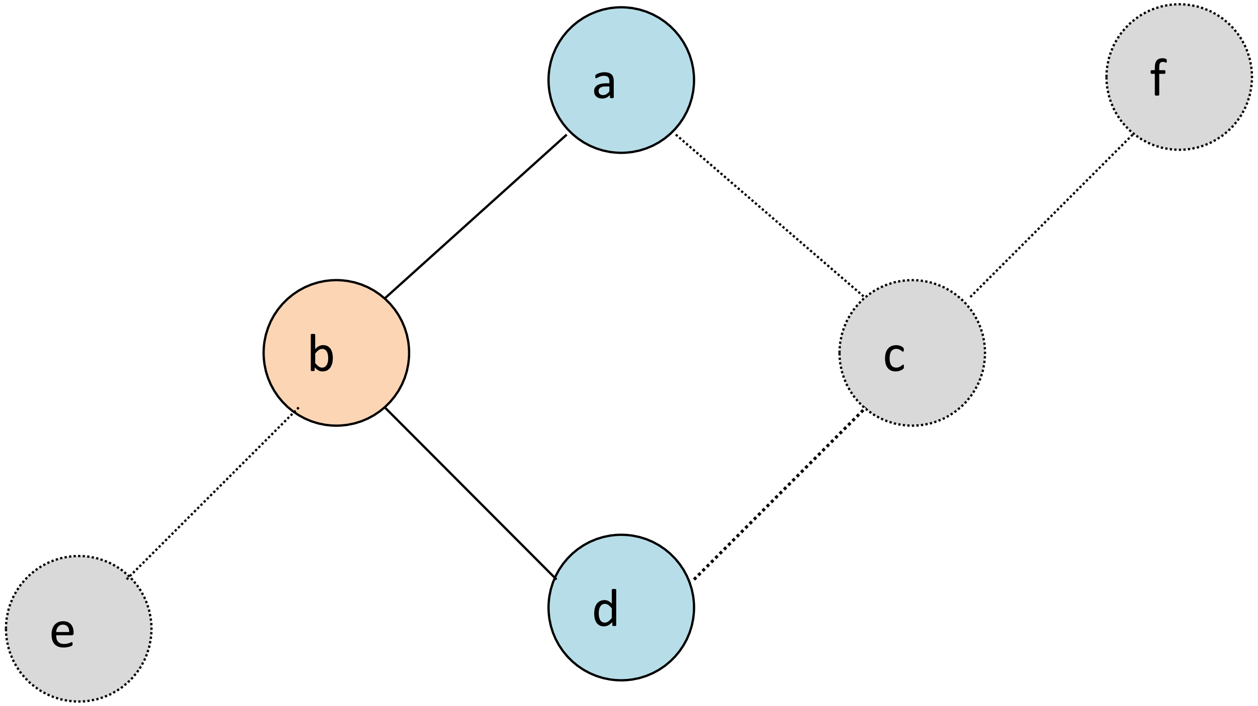
Color	Register
🟦	eax
🟧	ebx



Stack:

d
c
e
f

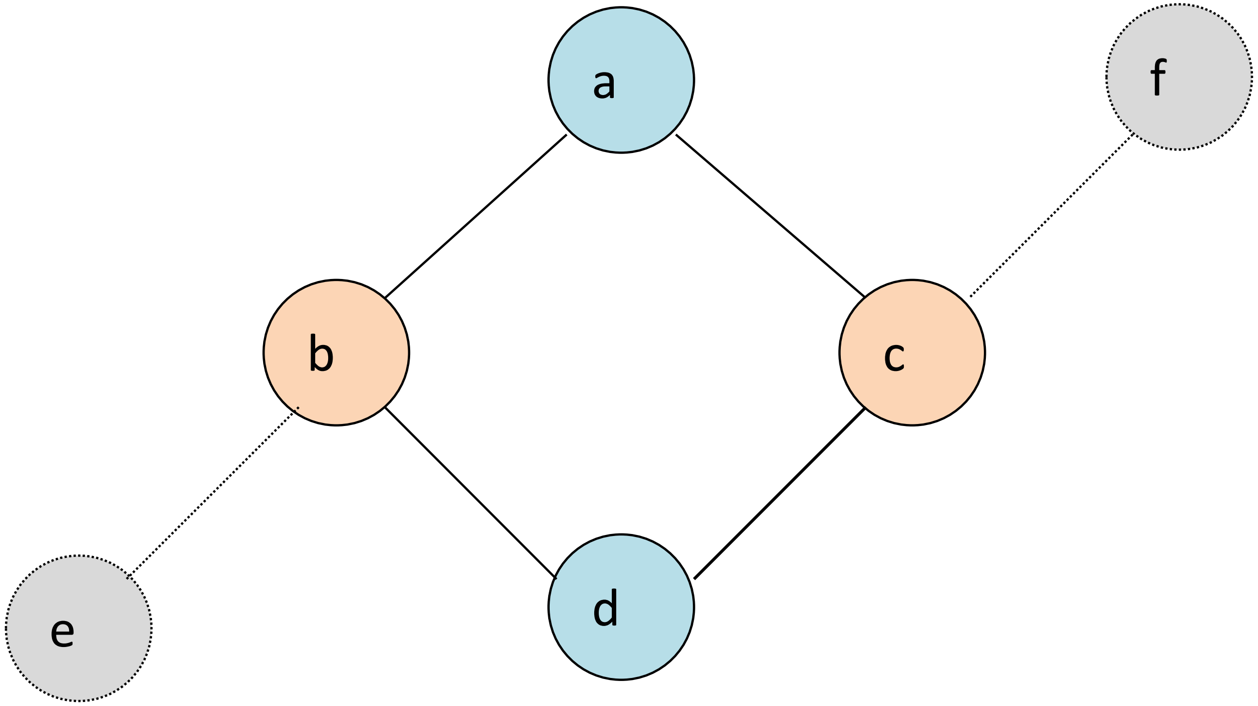
Color	Register
🟦	eax
🟧	ebx



Stack:

c
e
f

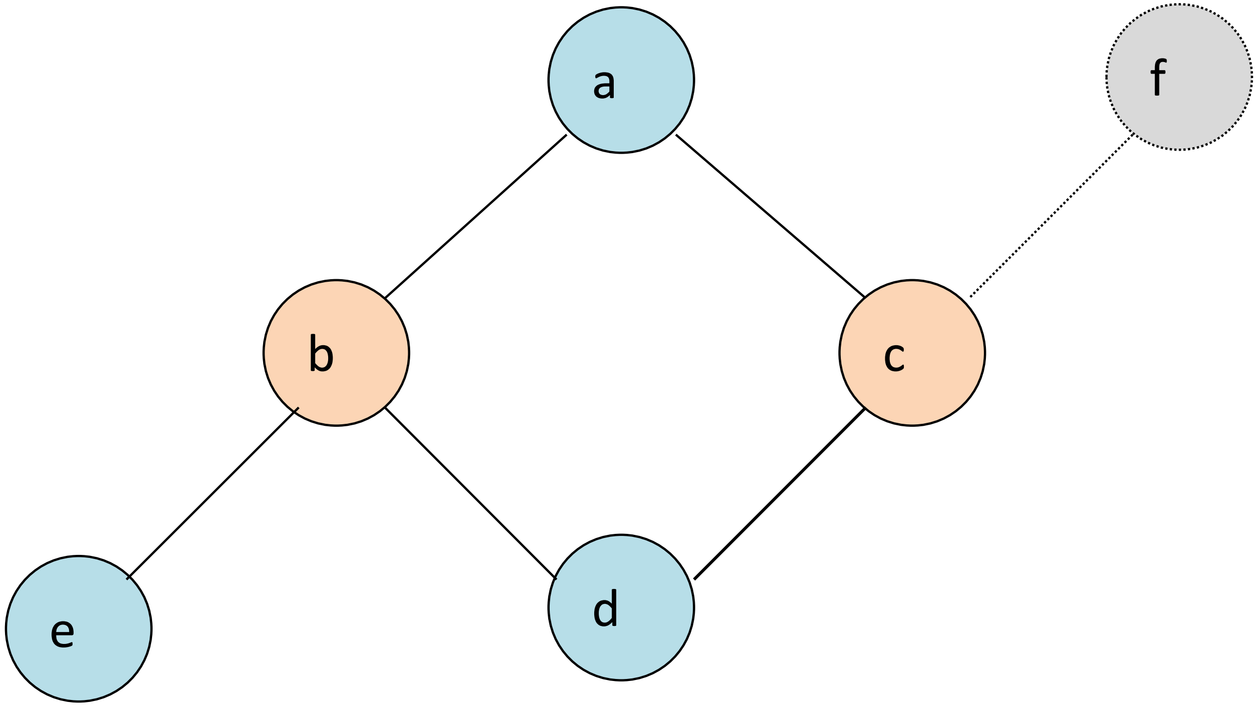
Color	Register
🟦	eax
🟧	ebx



Stack:

e
f

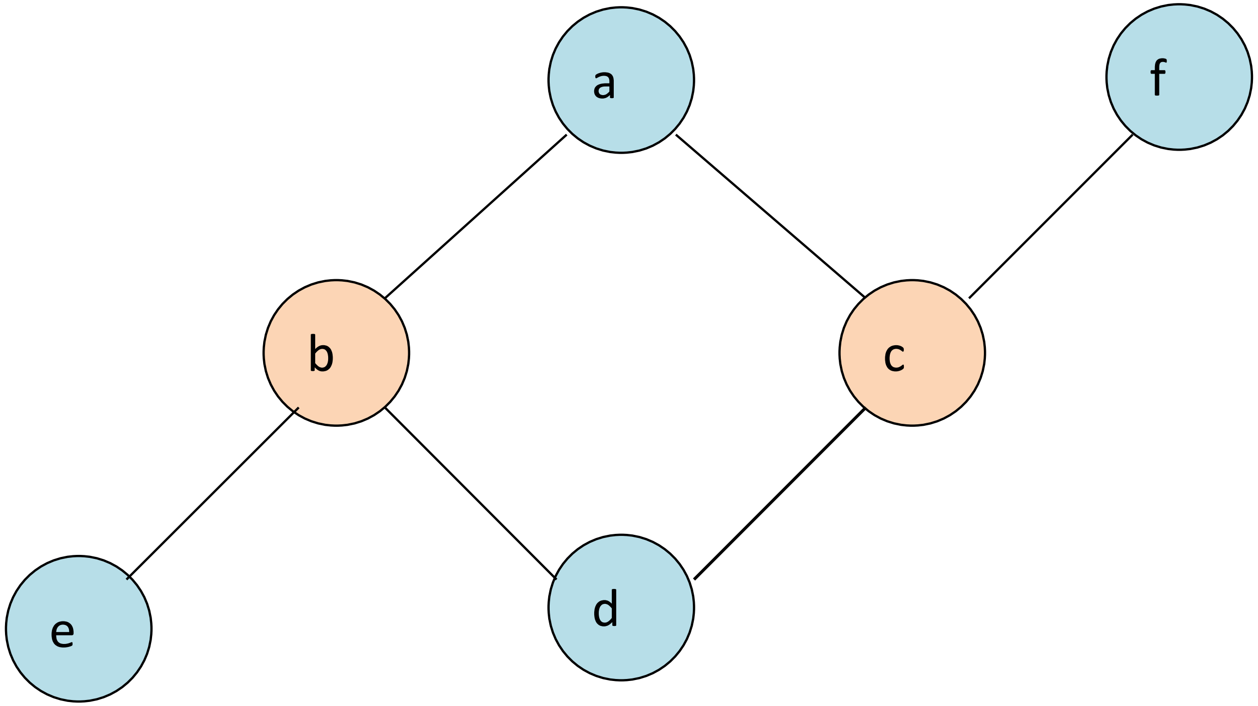
Color	Register
🟦	eax
🟧	ebx



Stack:

f

Color	Register
🟦	eax
🟧	ebx



Stack:

Example

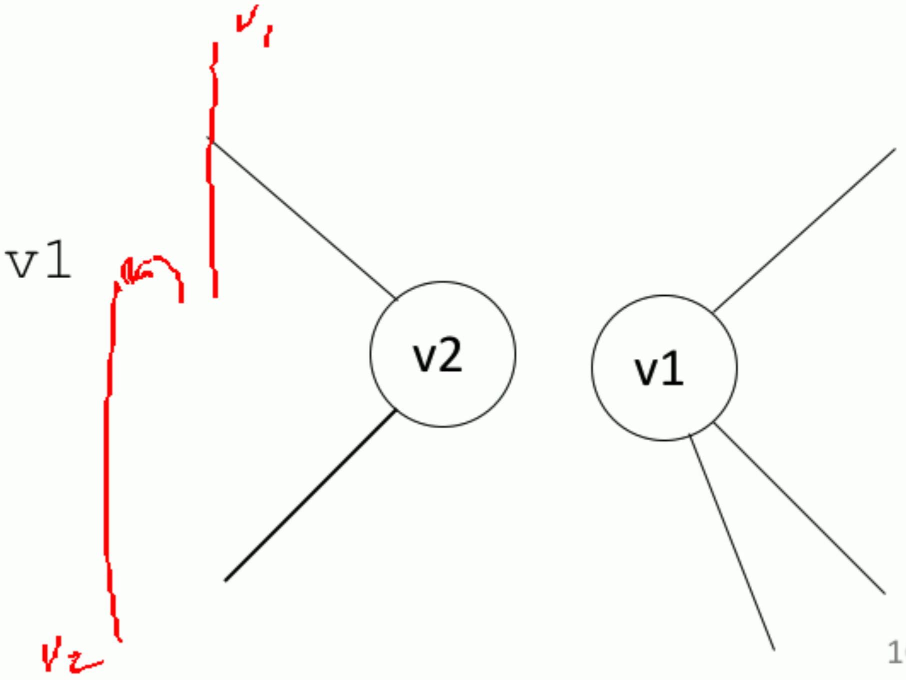
- **Although each node (after removing e, f) has ≥ 2 edges, we find a 2-coloring.**

- **Can we exploit this insight in the register allocator?**

Coalescing

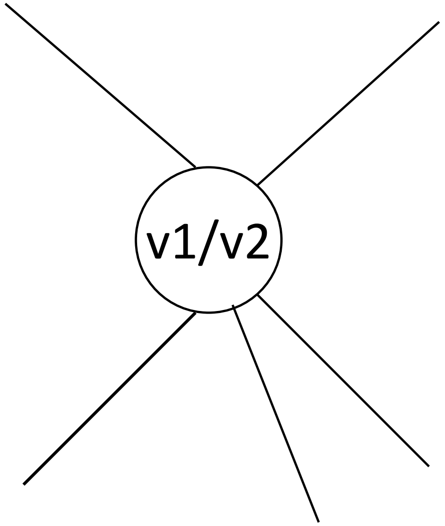
- **Code often contains a number of copy assignments**
  - Despite copy propagation
- **Example**

```
   = v1 + ...
v2 = v1    // last use of v1
   = ... + v2
```



Coalescing (cont'd)

- **We can *coalesce* these live ranges**
  - Removes the need to have a copy assignment
  - May make life harder for register allocator as combined node (v1/v2) *may* not be removed by simplification



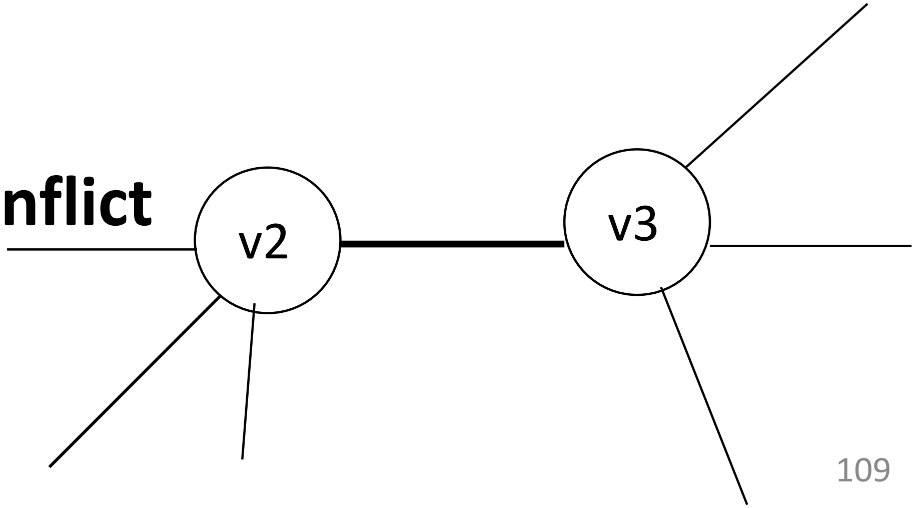
  - Heuristics to decide when to coalesce

Moves, again

- **Another example of a copy**

```
      =  v2 + …
v3    =  v2  // not last use of v2
      =  … + v3
      =  v2
```

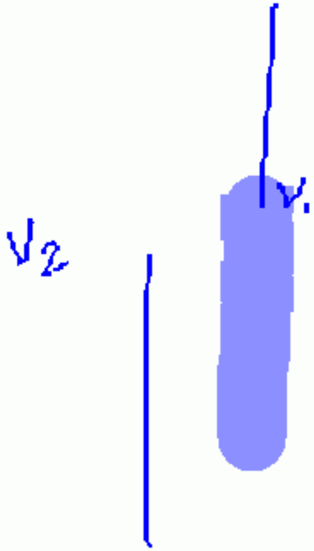
- **Now live ranges of v2 and v3 conflict**



Moves, again

- **Another example of a copy**

```
     =  v2 + …
v3   =  v2   // not last use of v2
     =  … + v3
     =  v2
```

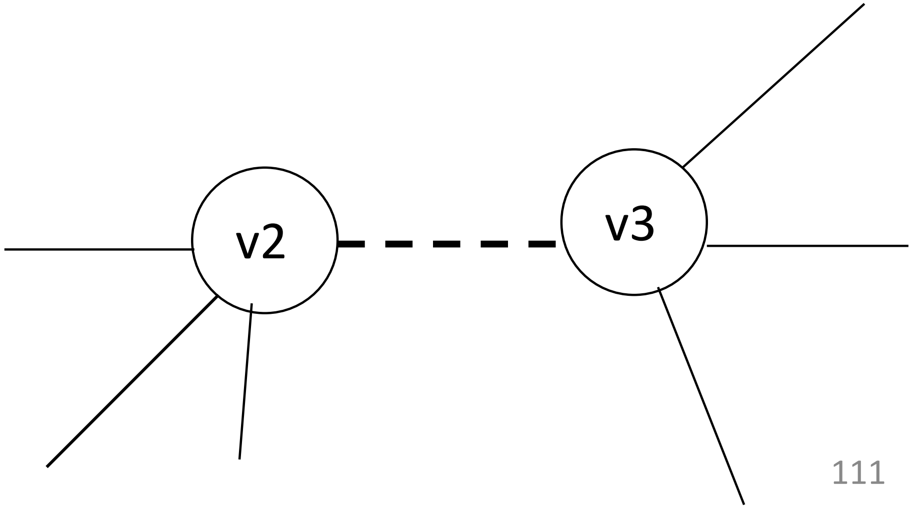


Potential conflicts

- **If one live range duplicates the value of another live range then give special treatment to edges in interference graph**

```
         =  v2 + …
    v3   =  v2  // last use of v2
         =  … + v3
         =  v3
```

- **Edge v2—v3 indicates copy property**
    - Attempt to give these nodes the same color

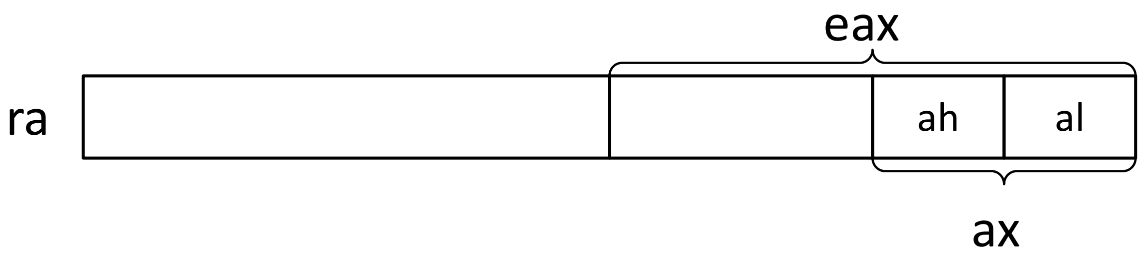


Machine features

- **Some instructions work with specific registers**
  - mul on x86: reads `eax`, defines `eax` and `edx`
- **Must make sure operands are in these registers**
  - Other registers not allowed
- **“Pre-color” these operands**
  - Assures that operand is assigned to this register
  - Color node for operand in interference graph
  - Pre-colored nodes are not removed during simplification
  - Coloring starts when all other nodes are removed

Machine features

- **The interference graph for x86 architectures must reflect that accesses to different parts of the same physical register are possible**
  - Low order bytes and lower half-word have separate names



  - 64bit register space shares resources with 32bit registers (and 16 bit registers (and 8 bit registers))
- **Not a topic for our compiler**

Register allocation…

- **Once considered to be beyond the reach of compilers**
  - Need for expert programmers
- **C programming language contains `register` storage class**
  - Hint to compiler to put variable into a CPU register
  - `register int loopcntr;`

Register allocation…

- **First formulation as coloring problem (paper ~1970s by Cocke, Yershov, Schwartz, first workable implementation published by Chaitin in 1981)**
- **Today: Compiler produces good results in many cases**
  - Some compilers produce multiple color assignments and then pick “the best”
  - Even C compilers ignore the `register` directive

Register allocation…

- **Many iterations may be needed**
  - Various heuristics create many options
- **Major steps**
  - Liveness analysis, interference graph construction
  - Coloring – Simplification
  - Spill/split decisions
    - Rewrite code
  - Actual coloring