CMPUT680 - Winter 2001

Topic 6: Register Allocation and Instruction Scheduling

José Nelson Amaral

http://www.cs.ualberta.ca/~amaral/courses/680

Reading List

- Tiger book: chapter 10 and 11
- Dragon book: chapter 10
- Other papers as assigned in class or homeworks

Register Allocation

- Motivation
- Live ranges and interference graphs
- Problem formulation
- Solution methods

Goals of Optimized Register Allocation

- To assign registers to variables that are more profitable to keep in registers.
- Use the same register for multiple variables when it is legal to do so.

Liveness

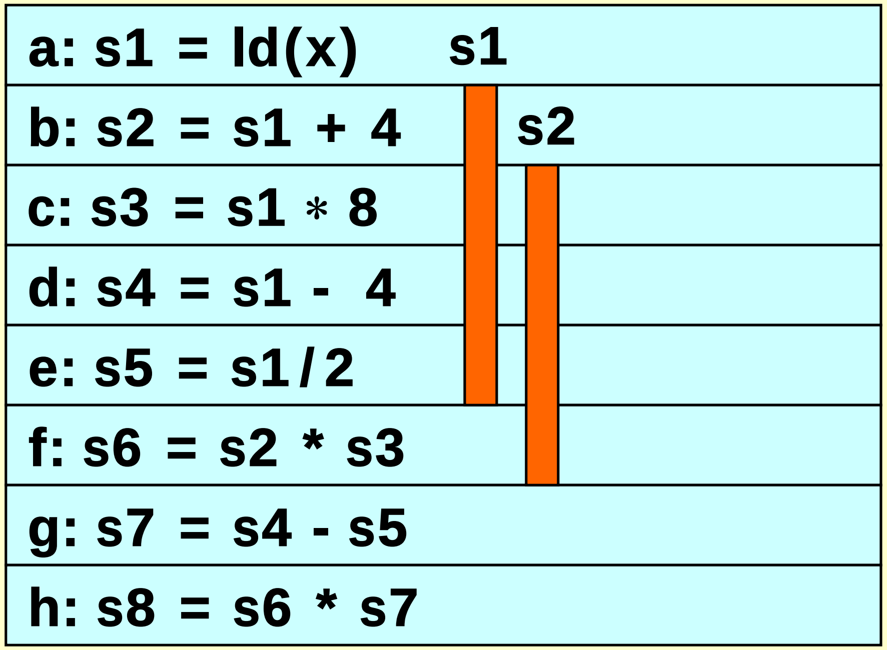
Intuitively a variable **v** is live if it holds a value that may be needed in the future. In other words, **v** is live at a point p_i if:

(i) **v** has been defined in a statement that precedes p_i in any path, and

(ii) **v** may be used by a statement s_j, and there is a path from p_i to s_j.

(iii) **v** is not killed between p_i and s_j.

Live Variables



```
a: s1 = ld(x)
b: s2 = s1 + 4
c: s3 = s1 * 8
d: s4 = s1 -  4
e: s5 = s1/2
f: s6 = s2 * s3
g: s7 = s4 - s5
h: s8 = s6 * s7
```

A variable **v** is live between the point **p_i** immediately after its definition and the point **p_j** immediately after its last use.

The interval **[p_i, p_j]** is the **live range** of the variable **v**.

Which variables have the longest live range in the example?

Variables **s1** and **s2** have a live range of four statements.

Register Allocation

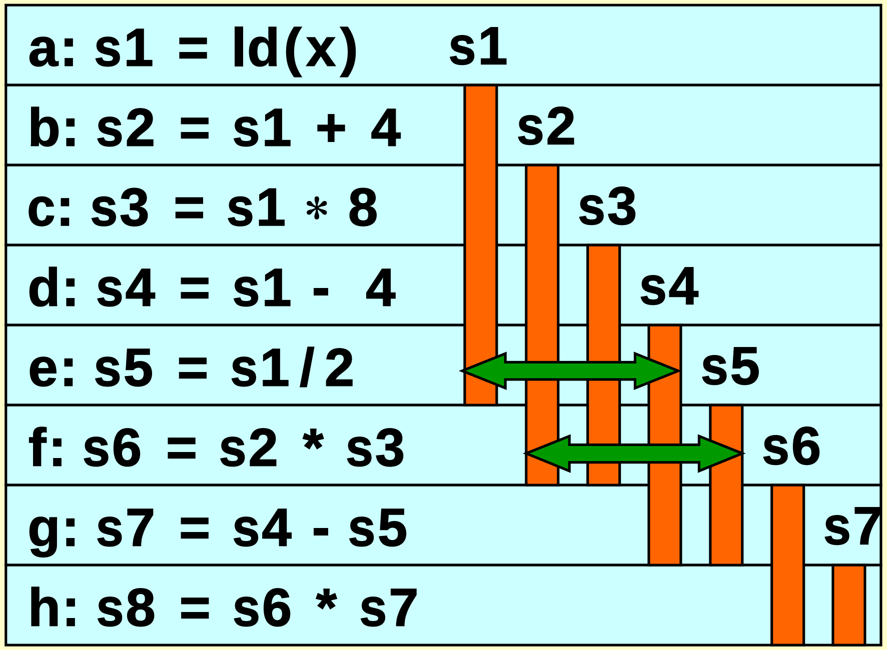
a: s1 = ld(x)
b: s2 = s1 + 4
c: s3 = s1 ∗ 8
d: s4 = s1 - 4
e: s5 = s1/2
f: s6 = s2 * s3
g: s7 = s4 - s5
h: s8 = s6 * s7

How can we find out what is the minimum number of registers required by this basic block to avoid spilling values to memory?

We have to compute the live range of all variables and find the **"fatest"** statement.

Which statements have the most variables live simultaneously?

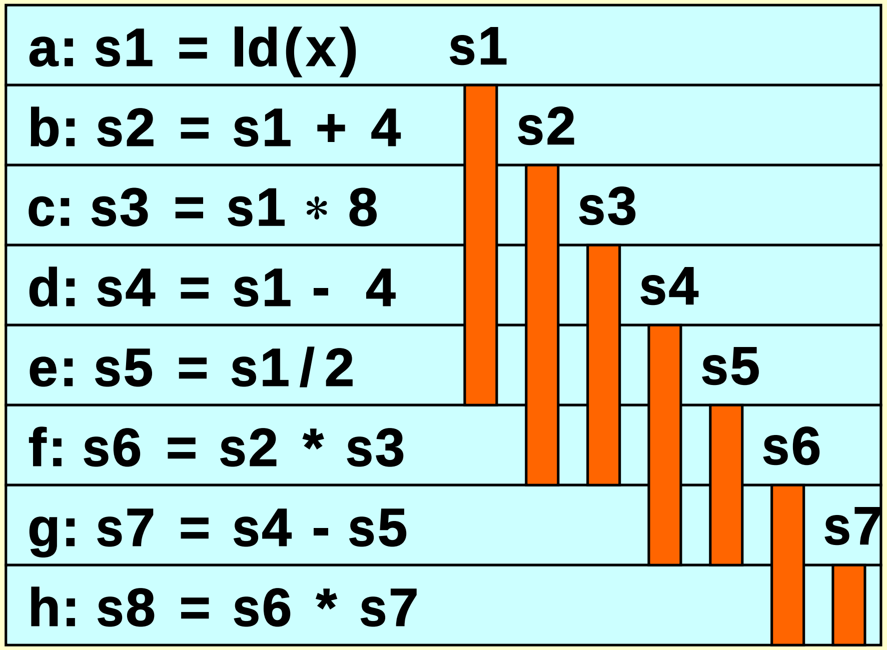
Register Allocation



At statement **d** variables **s1**, **s2**, **s3**, and **s4** are live, and during statement **e** variables **s2**, **s3**, **s4**, and **s5** are live.

But we have to use some math: our choice is **liveness analysis**.

Live-in and Live-out

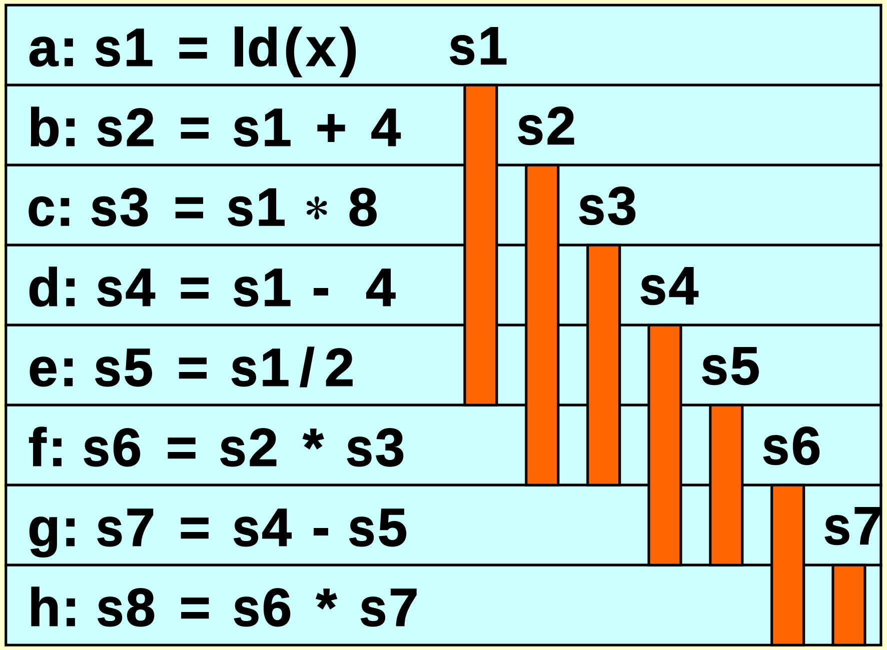


```
a: s1 = ld(x)
b: s2 = s1 + 4
c: s3 = s1 * 8
d: s4 = s1 - 4
e: s5 = s1/2
f: s6 = s2 * s3
g: s7 = s4 - s5
h: s8 = s6 * s7
```

live-in(r): set of variables that are live at the point immediately before statement **r**.

live-out(r): set of variables that are live at the point immediately after statement **r**.

Live-in and Live-out: Program Example



What are live-in(e) and live-out(e)?

live-in(e) = {s1, s2, s3, s4} live-out(e) = {s2, s3, s4, s5}

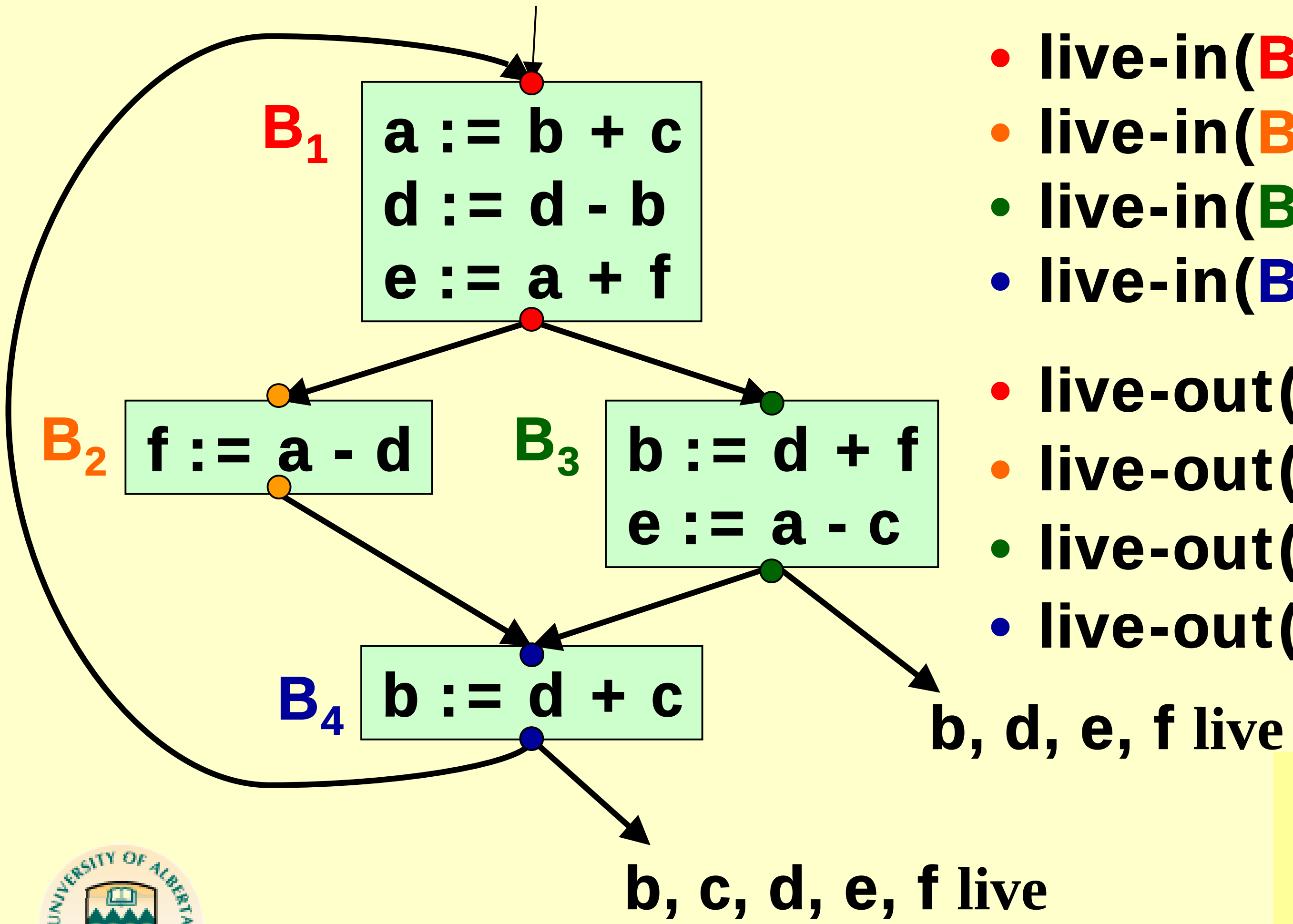
Live-in and Live-out in Control Flow Graphs

The entry point of a basic block B is the point before its first statement. The exit point is the point after its last statement.

live-in(B): set of variables that are live at the entry point of the basic block B.

live-out(B): set of variables that are live at the exit point of the basic block B.

Live-in and Live-out of basic blocks



- live-in(B_1)={b,c,d,f}
- live-in(B_2)={a,c,d,e}
- live-in(B_3)={a,c,d,f}
- live-in(B_4)={c,d,e,f}

- live-out(B_1)={a,c,d,e,f}
- live-out(B_2)={c,d,e,f}
- live-out(B_3)={b,c,d,e,f}
- live-out(B_4)={b,c,d,e,f}

Compute live-in and live-out for each basic block

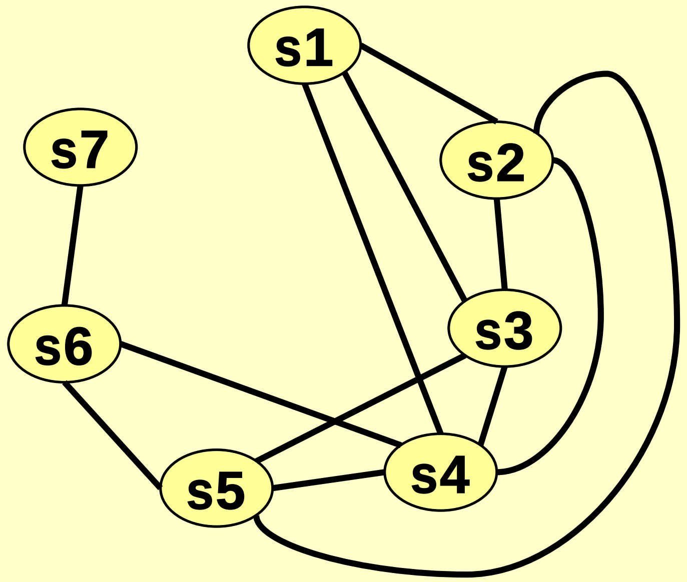
(Aho-Sethi-Ullman, pp. 544)

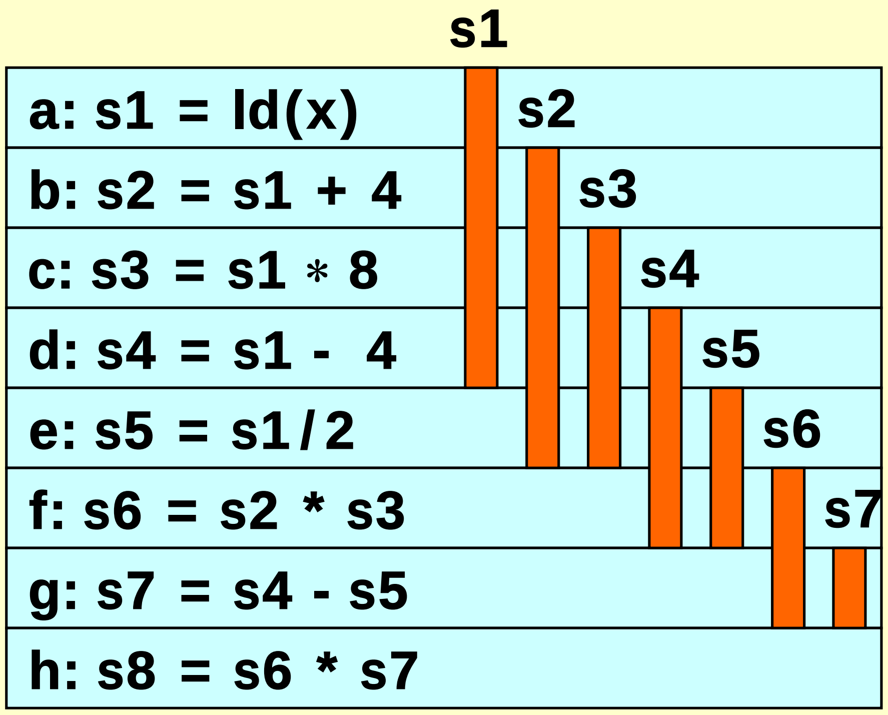
Register-Interference Graph

A ***register-interference graph*** is an **undirected** graph that summarizes live analysis at the variable level as follows:

- A node is a variable/temporary that is a candidate for register allocation.
- An edge connects nodes **v1** and **v2** if there is some statement in the program where variables **v1** and **v2** are live simultaneously. (Variables **v1** and **v2** are said to interfere, in this case).

Register Interference Graph: Program Example





```
a: s1 = ld(x)
b: s2 = s1 + 4
c: s3 = s1 ∗ 8
d: s4 = s1 -  4
e: s5 = s1 / 2
f: s6 = s2 * s3
g: s7 = s4 - s5
h: s8 = s6 * s7
```

Register Allocation by Graph Coloring

Background: A graph is **k**-colorable if each node has been assigned one of **k** colors in such a way that no two adjacent nodes have the same color.

Basic idea: A **k**-coloring of the interference graph can be directly mapped to a legal register allocation by mapping each color to a distinct register. The coloring property ensures that no two variables that interfere with each other are assigned the same register.

Register Allocation by Graph Coloring

The basic idea behind register allocation by graph coloring is to

1. Build the register interference graph,
2. Attempt to find a k-coloring for the interference graph.

Complexity of the Graph Coloring Problem

- The problem of determining if an undirected graph is k-colorable is NP-hard for $k \geq 3$.
- It is also hard to find approximate solutions to the graph coloring problem.

Register Allocation

Question: What to do if a register-interference graph is not k-colorable? Or if the compiler cannot efficiently find a k-coloring even if the graph is k-colorable?

Answer: Repeatedly select less profitable variables for “spilling” (i.e. not to be assigned to registers) and remove them from the interference graph till the graph becomes k-colorable.

Estimating Register Profitability

The register profitability of variable v is estimated by:

$$profitability(v) = \sum_i freq(i) \times savings(v, i)$$

$freq(i)$: estimated execution frequency of basic block i (obtained by profiling or by static analysis),

$savings(v, i)$: estimated number of processor cycles that would be saved due to a reduced number of load and store instructions in basic block i, if a register was assigned to variable v.

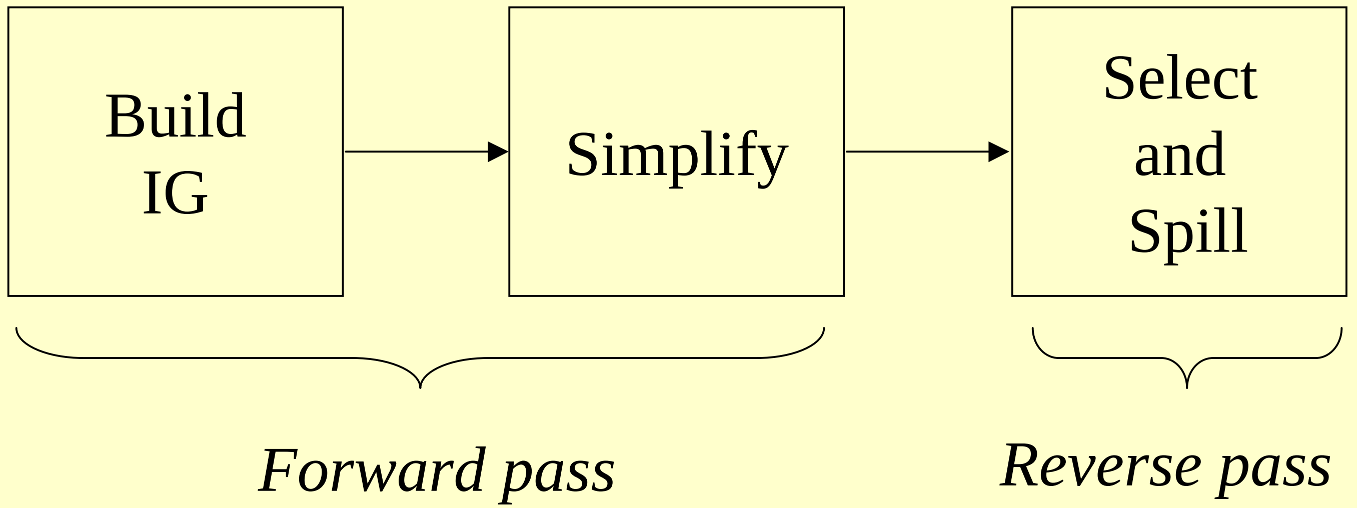
Heuristic Solution for Graph Coloring

Key observation:

$$\begin{tikzcd} G \arrow[rr, "\text{Remove a node x with degree} < k", "\text{from G, and all associated edges}"'] & & G' \end{tikzcd}$$

Then G is k-colorable if G’ is k-colorable.

A 2-Phase Register Allocation Algorithm



Heuristic "Optimistic" Algorithm

```
/* neighbor(v) contains a list of
   the neighbors of v. */
/* Build step */
Build the register-interference
          graph, G;

/* Forward pass */
Initialize an empty stack;
repeat
    while G has a node v such that
        |neighbor(v)| < k do
        /* Simplify step */
        Push (v, neighbors(v), no-spill)
        Delete v and its  edges from G
    end while
    if G is non-empty then
        /* Spill step */
        Choose "least profitable" node v
                as a  potential spill node;
        Push (v, neighbors(v), may-spill)
        Delete v and its edges from G
    end if
until G is an empty graph;
```

Heuristic "Optimistic" Algorithm

```
/* Reverse Pass */
while the stack is non-empty do
    Pop (v, neighbors(v), tag)
    N := set of nodes in neighbors(v);
    if (tag = no-spill) then
        /* Select step */
        Select a register R for v such that
            R is not assigned to nodes in N;
        Insert v as a new node in G;
        Insert an edge in G
            from v to each node in N;
    else /* tag = may-spill */
        if v can be assigned a register R
            such that R is not assigned
            to nodes in N then
            /* Optimism paid off: need not spill */
            Assign register R to v;
            Insert v as a new node in G;
            Insert an edge in G from
                from v to each node in N;
        else
            /* Need to spill v */
            Mark v as not being allocate a register
        end if
    end if
end while
```

Remarks

The above register allocation algorithm based on graph coloring is both efficient (linear time) and effective.

It has been used in many industry-strength compilers to obtain significant improvements over simpler register allocation heuristics.

Extensions

- Coalescing
- Live range splitting

Coalescing

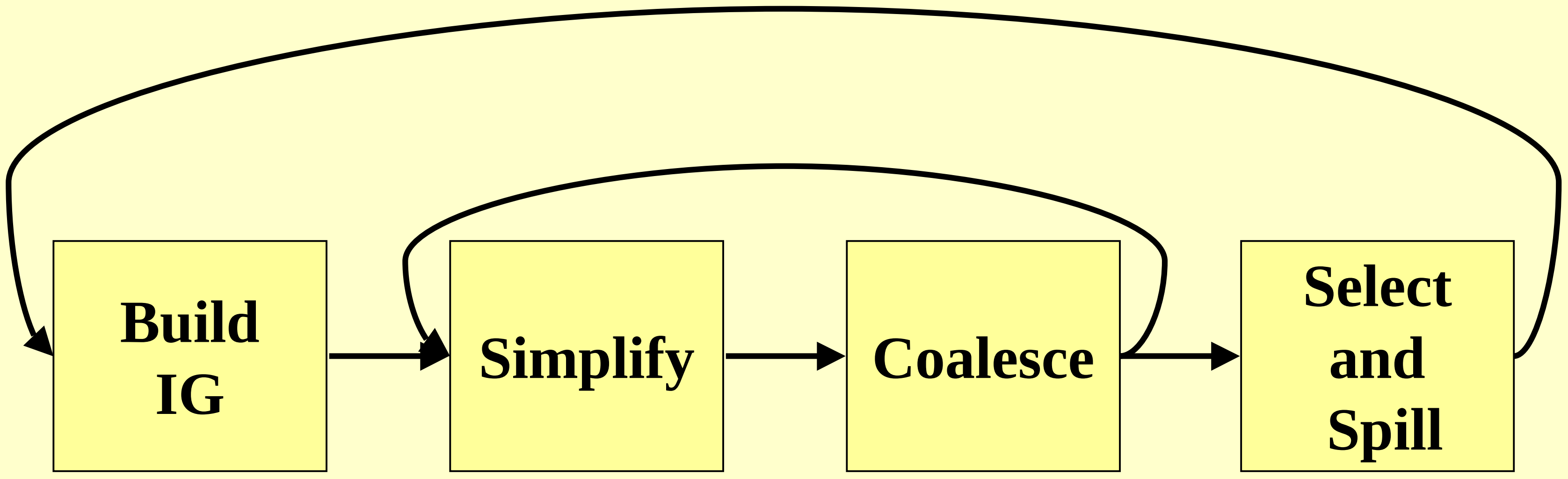
In the sequence of intermediate level instructions with a copy statement below, assume that registers are allocated to both variables **x** and **y**.

```
x   := ...
. . .
y   :=  x
. . .
...  :=  y
```

There is an opportunity for further optimization by eliminating the copy statement if **x** and **y** are assigned the same register.

The constraint that **x** and **y** receive the same register can be modeled by coalescing the nodes for **x** and **y** in the interference graph i.e., by treating them as the same variable.

An Extension with Coalesce



Register Allocation with Coalescing

1. Build: build the register interference graph **G** and categorize nodes as *move-related* or *non-move-related.*

2. Simplify: one at a time, remove non-move-related nodes of low (< **K**) degree from **G**.

3. Coalesce: conservatively coalesce **G**: only coalesce nodes **a** and **b** if the resulting **a-b** node has less than **K** neighbors.

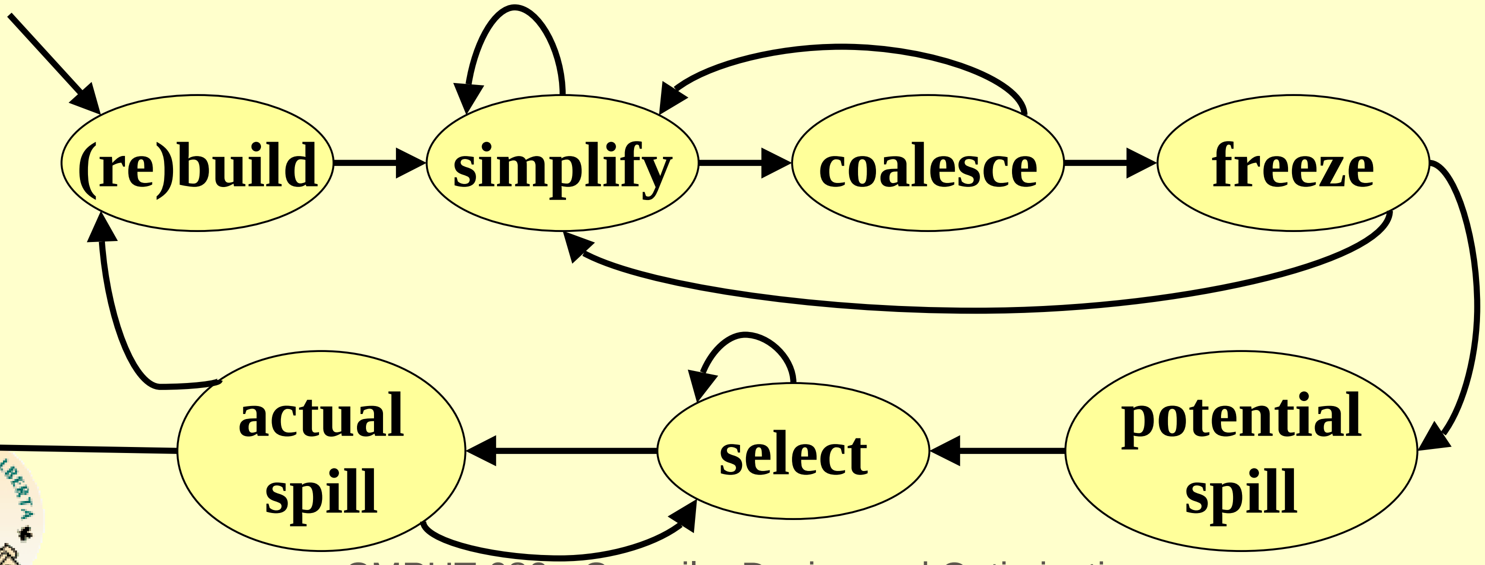
4. Freeze: If neither coalesce nor simplify works, freeze a move-related node of low degree, making it non-move-related and available for simplify.

(Appel, pp. 240)

Register Allocation with Coalescing

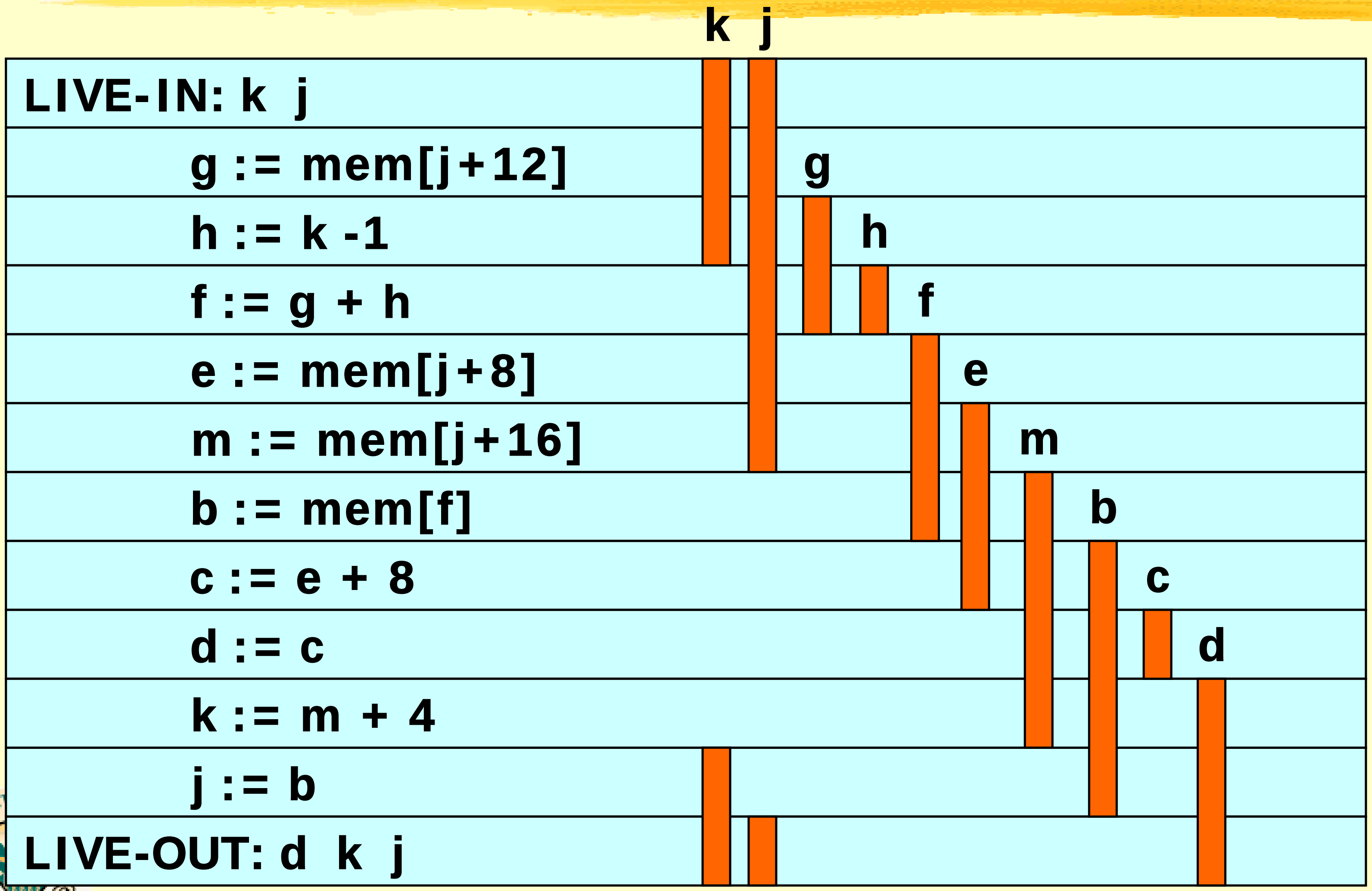
5. Spill: if there are no low-degree nodes, select a node for potential spilling.

6. Select: pop each element of the stack assigning colors.



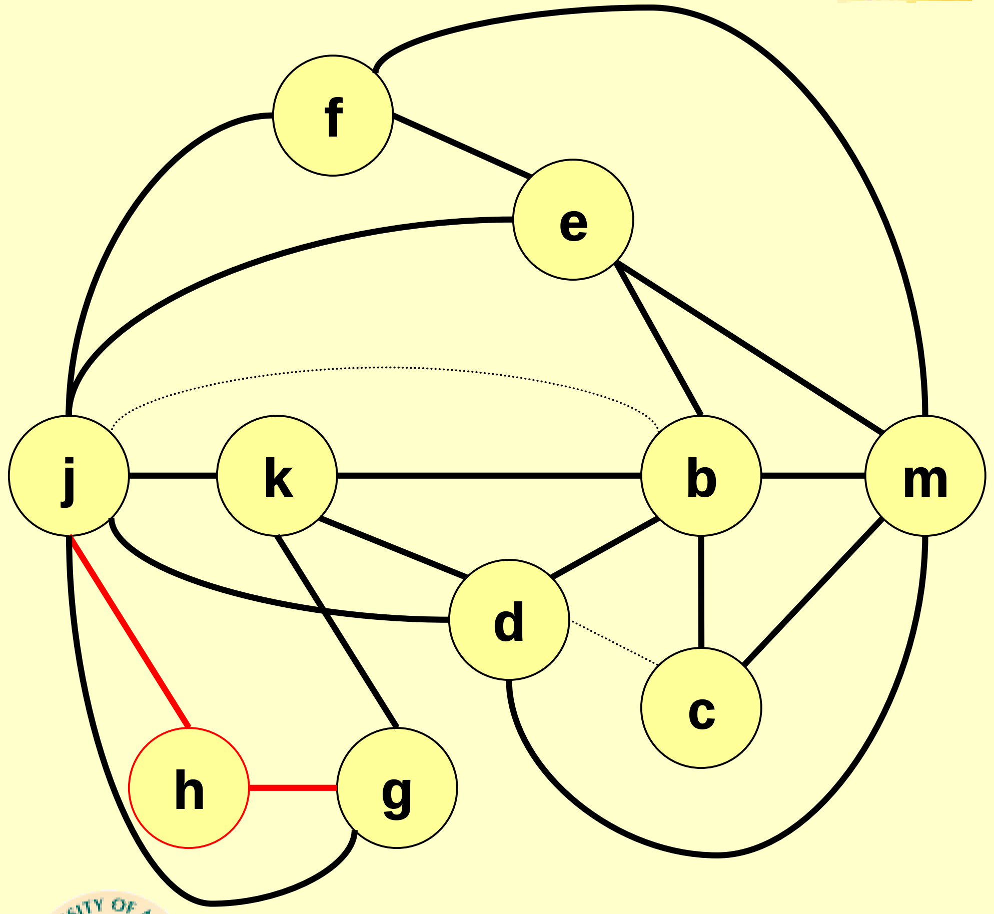
(Appel, pp. 240)

Example: Step 1: Compute Live Ranges



```
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
```

Example: Step 3: Simplify (K=4)

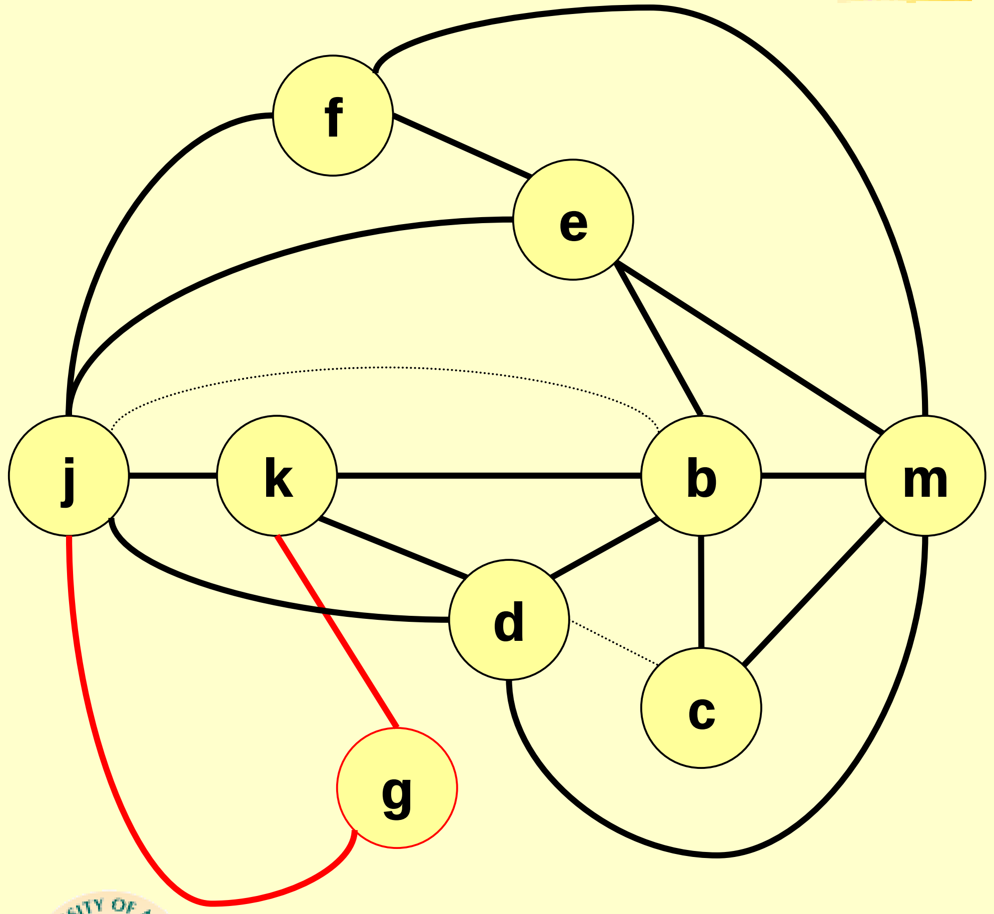


stack

(h,no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=4)

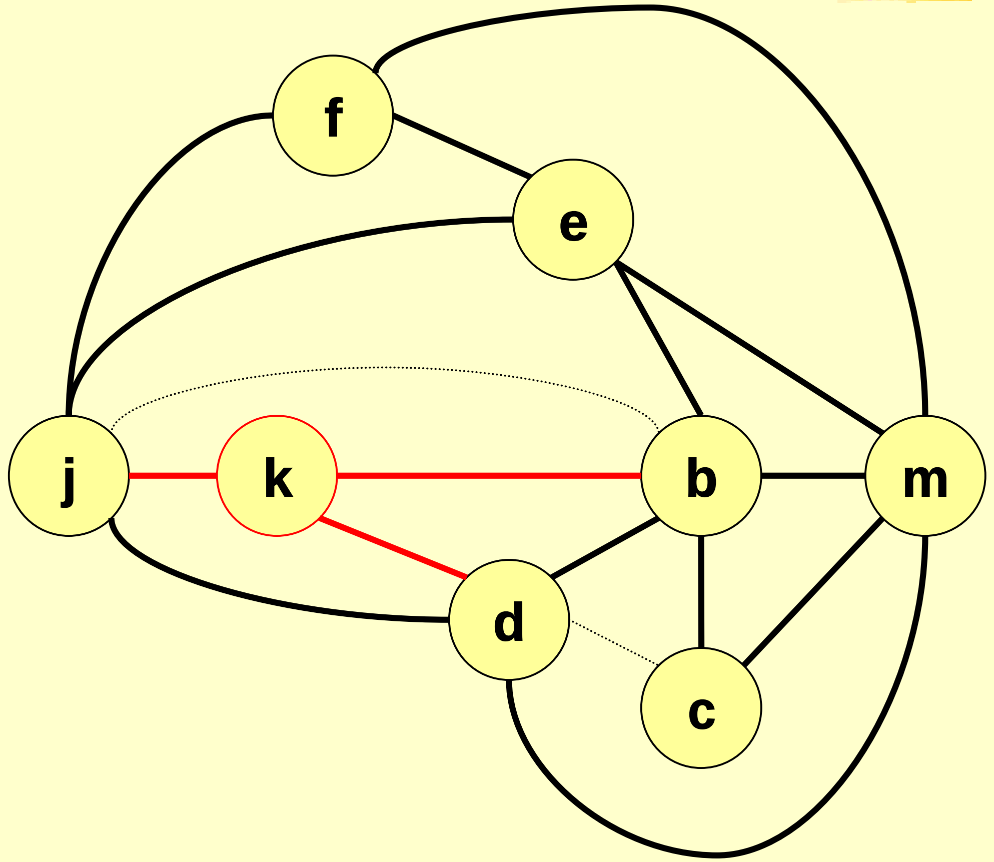


stack

(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=4)

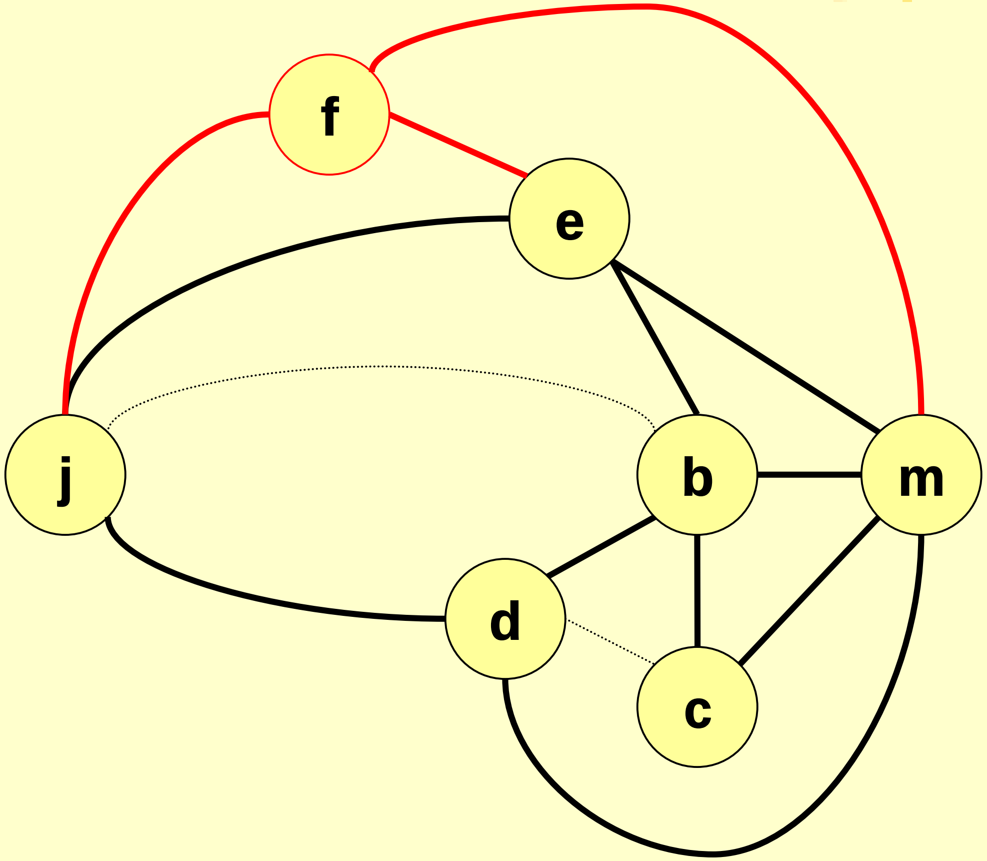


stack

(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=4)

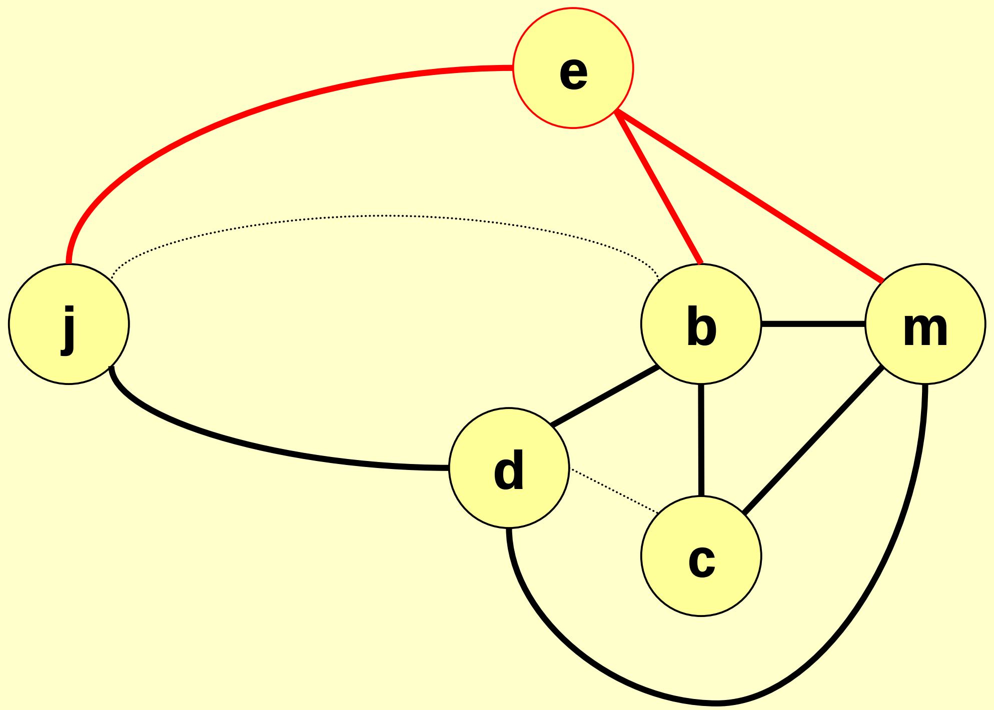


stack

(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example: Step 3: Simplify (K=4)



stack

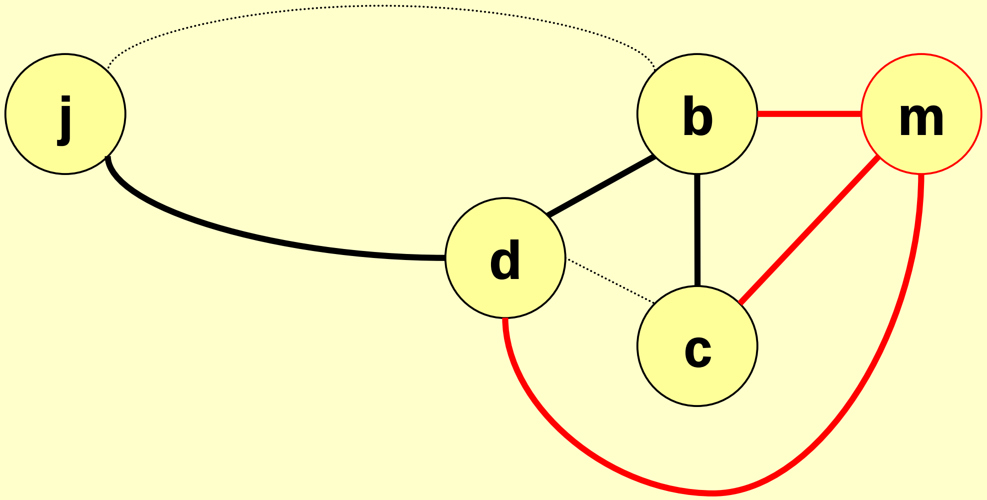
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=4)

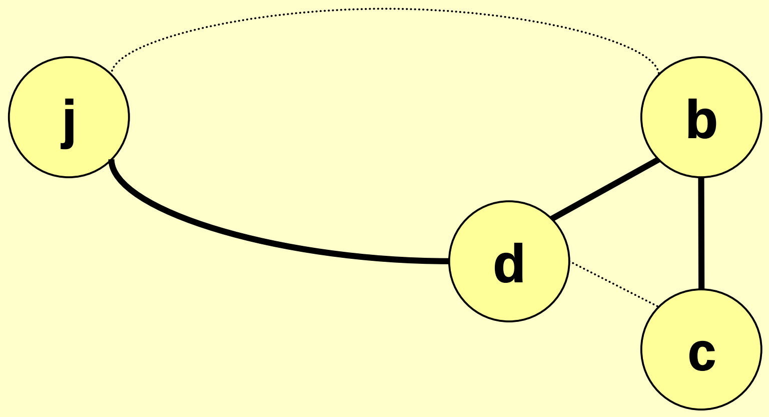
stack

(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)



(Appel, pp. 237)

Example:
Step 3: Coalesce (K=4)



stack

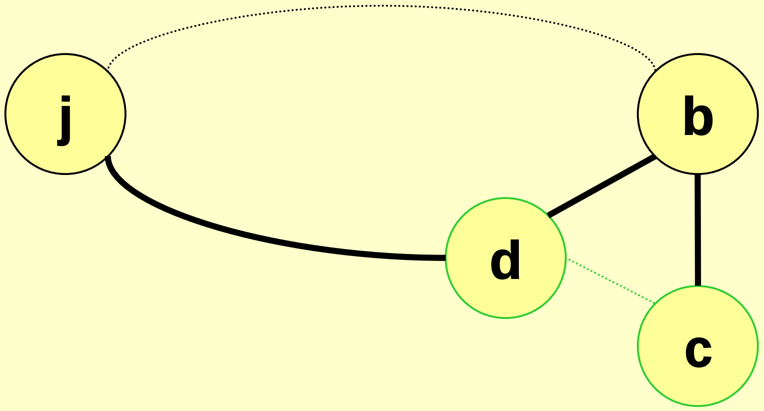
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

Why we cannot simplify?

Cannot simplify move-related nodes.

(Appel, pp. 237)

Example:
Step 3: Coalesce (K=4)

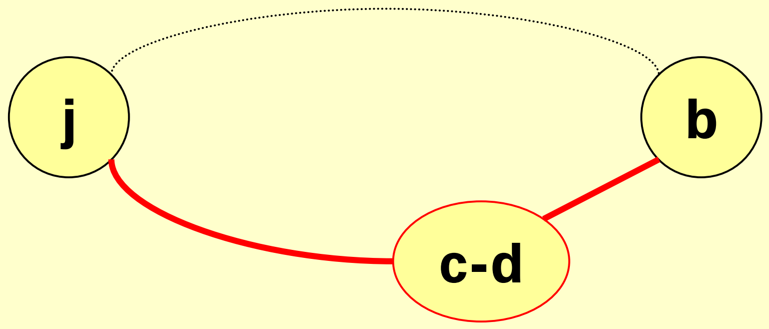


stack

(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=4)

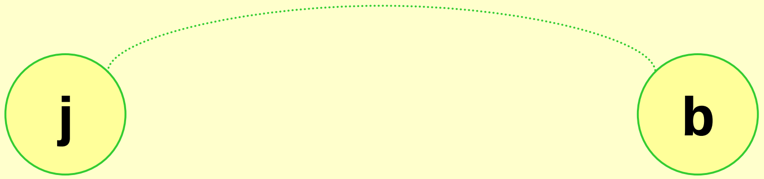


stack

(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Coalesce (K=4)

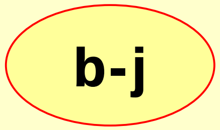


stack

(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=4)

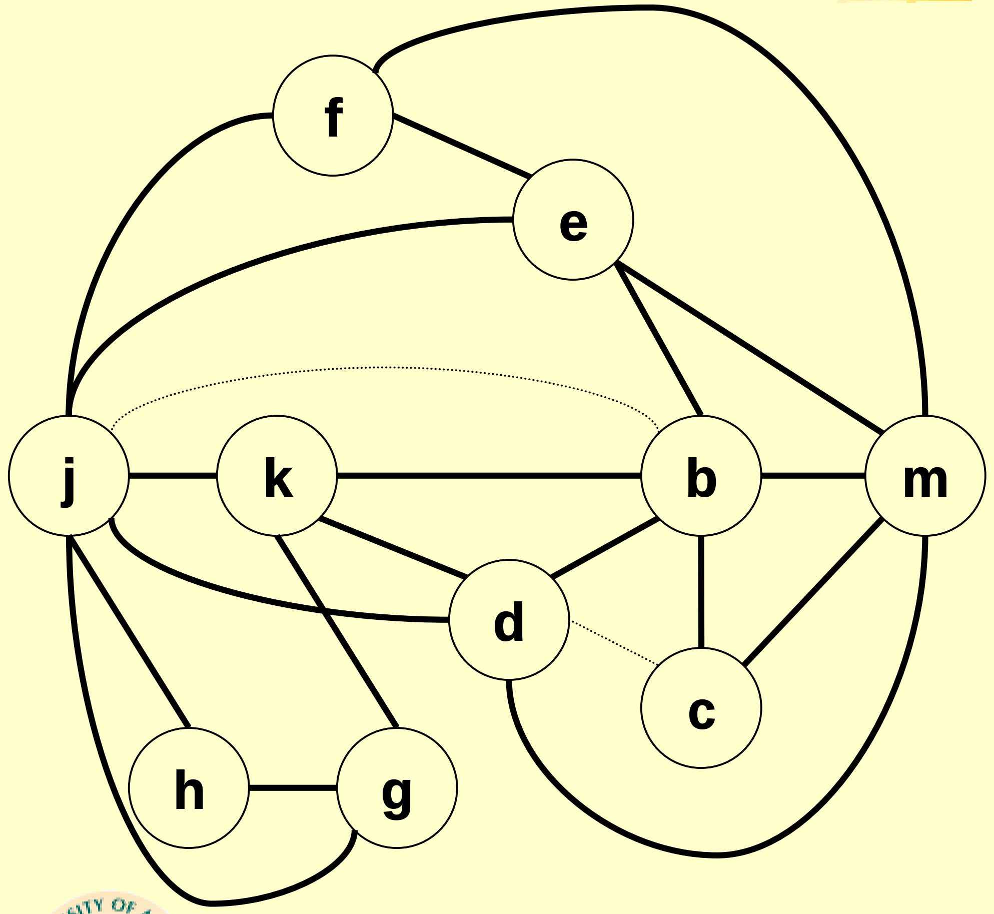


stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example: Step 3: Select (K=4)



stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

R1

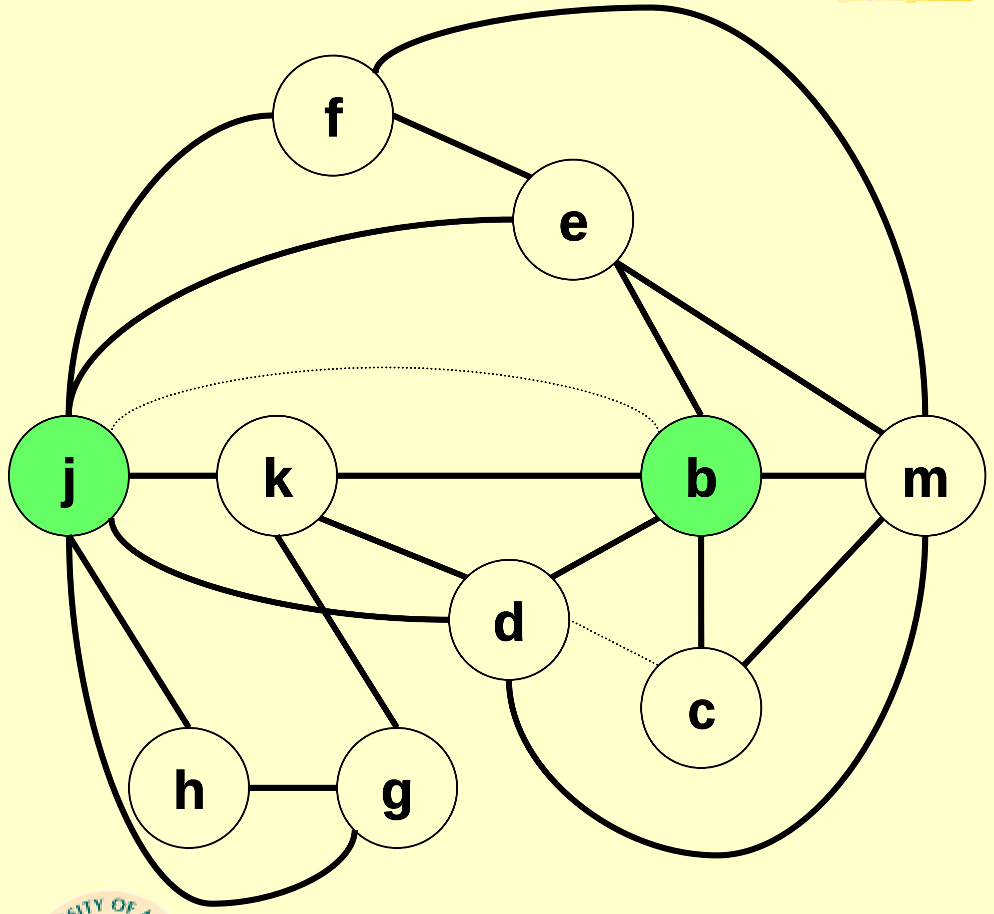
R2

R3

R4

(Appel, pp. 237)

Example:
Step 3: Select (K=4)



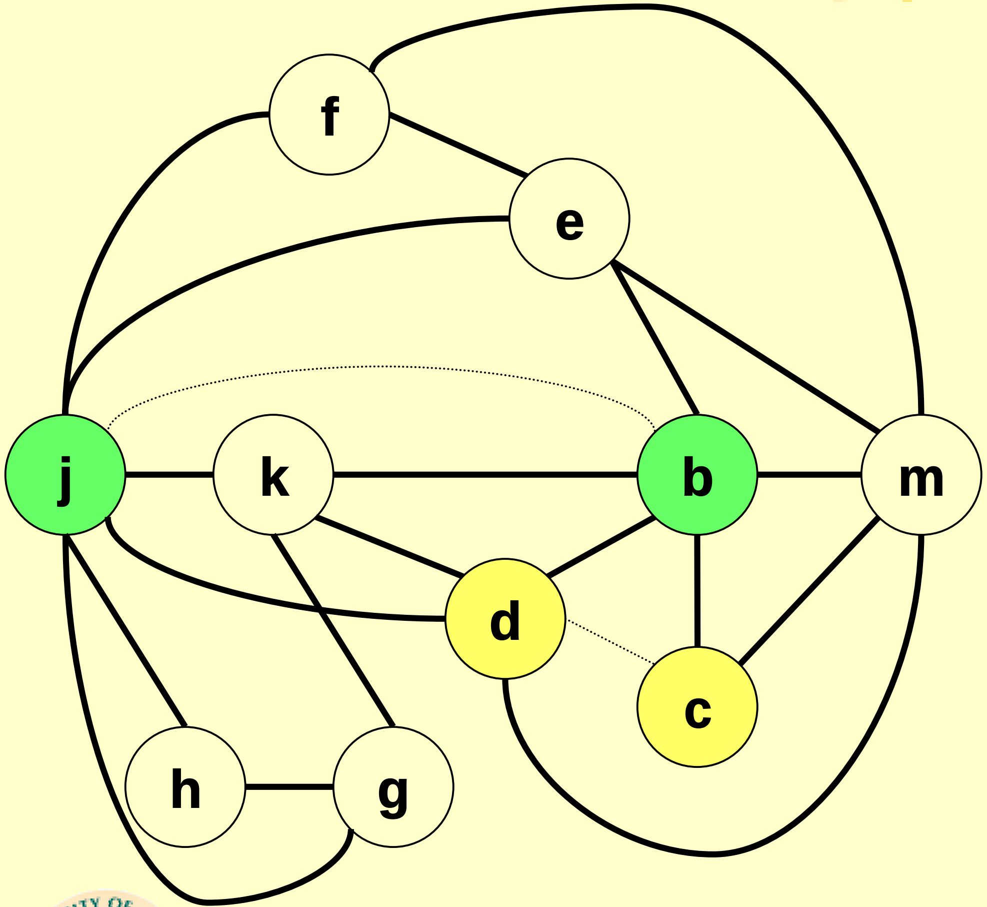
stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

R1 🟩
R2 🟨
R3 🟦
R4 🟥

(Appel, pp. 237)

Example:
Step 3: Select (K=4)



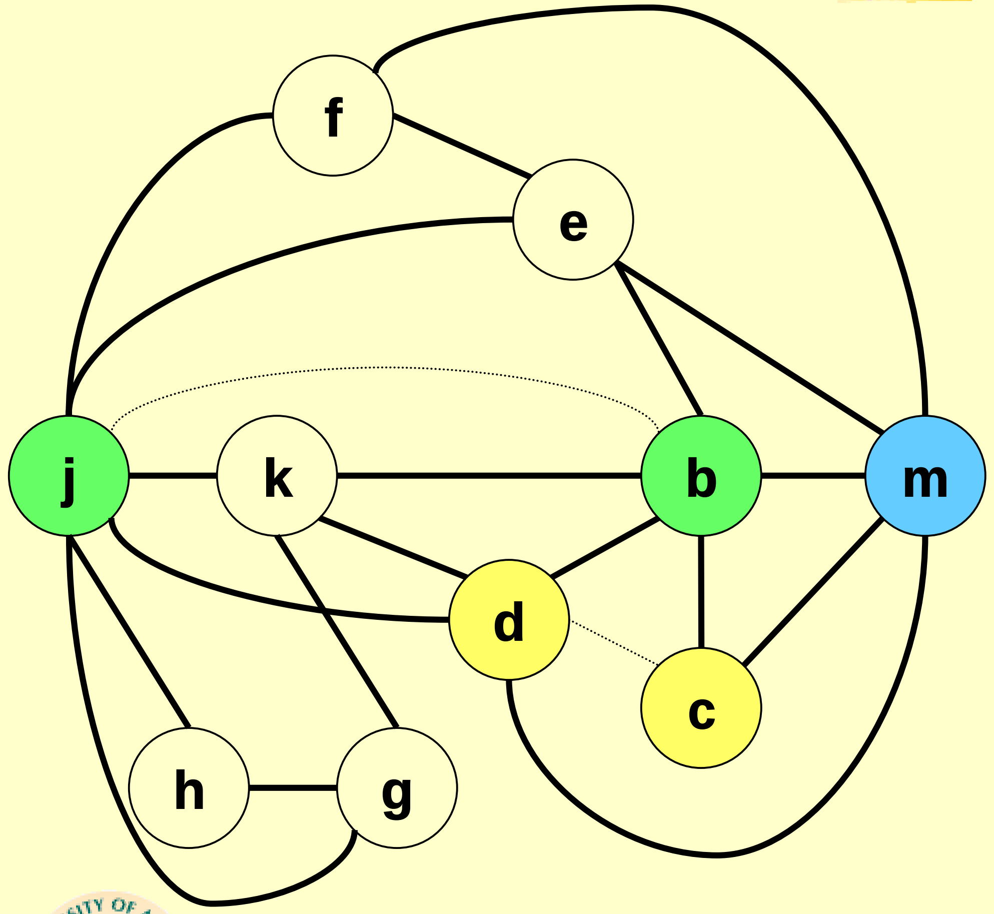
stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

🟩 **R1**
🟨 **R2**
🟦 **R3**
🟥 **R4**

(Appel, pp. 237)

Example:
Step 3: Select (K=4)



stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

🟩 **R1**

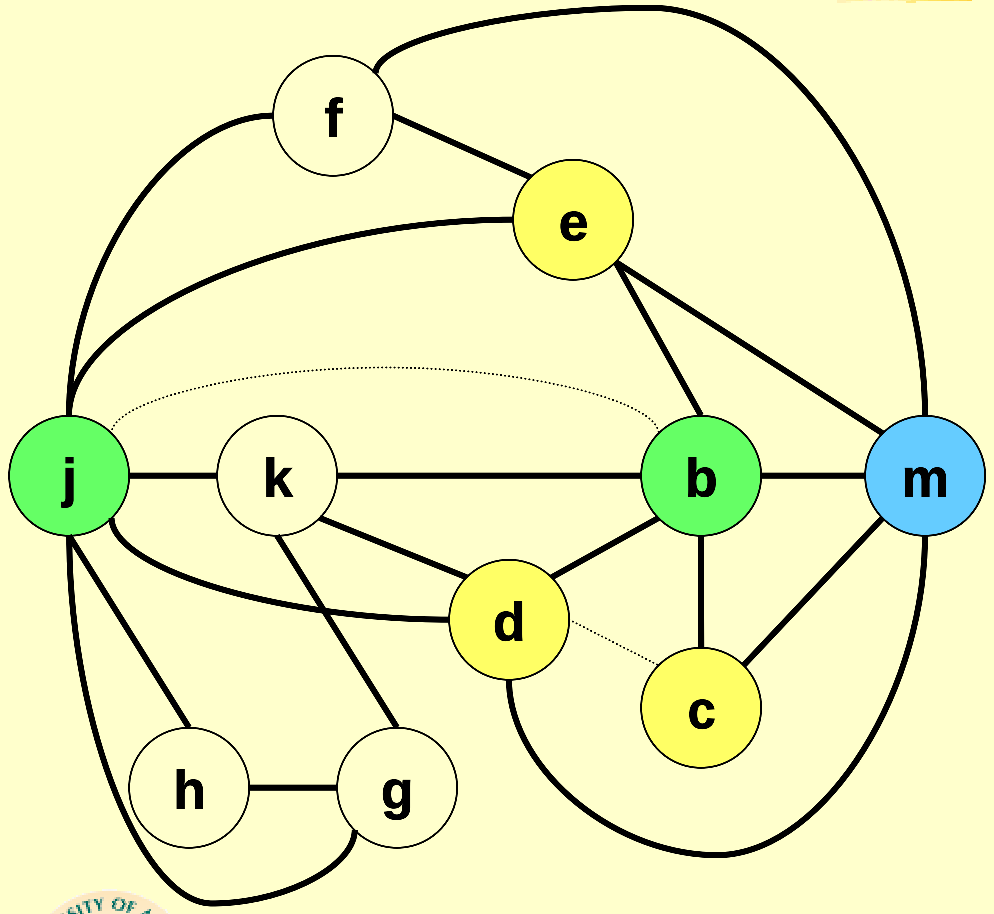
🟨 **R2**

🟦 **R3**

🟥 **R4**

(Appel, pp. 237)

Example: Step 3: Select (K=4)



stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

R1

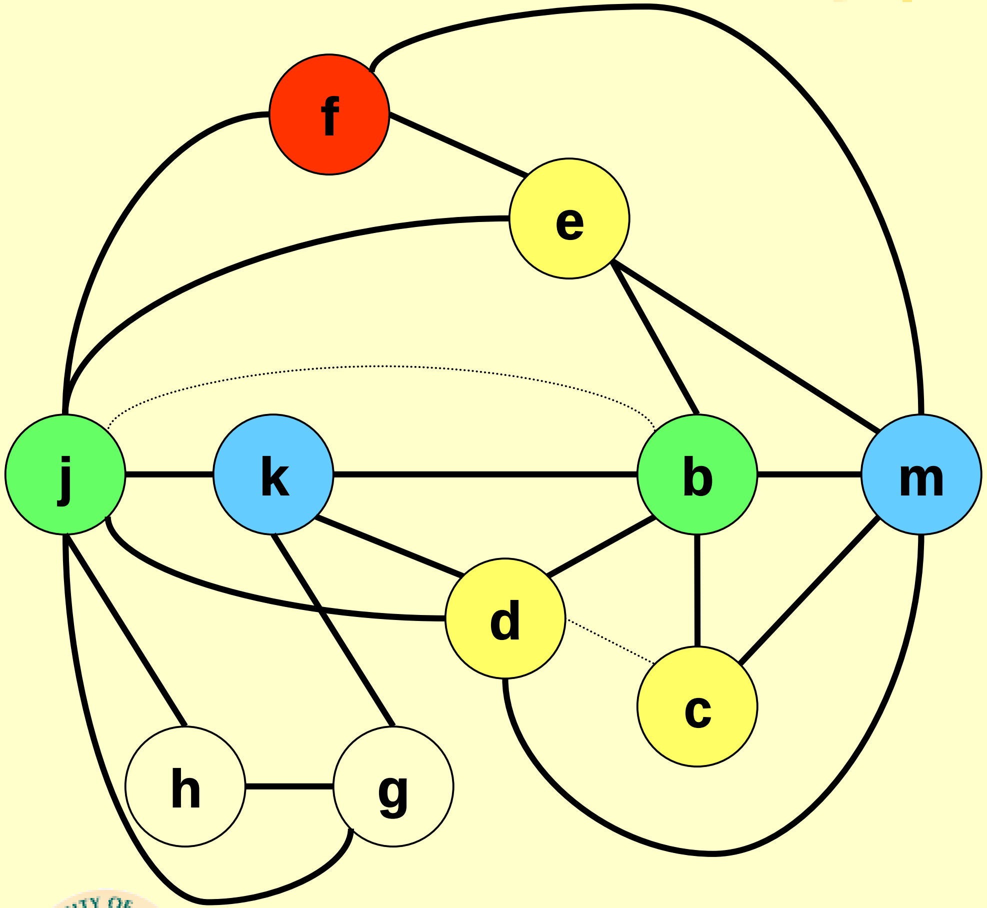
R2

R3

R4

(Appel, pp. 237)

Example:
Step 3: Select (K=4)



stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

🟩 R1

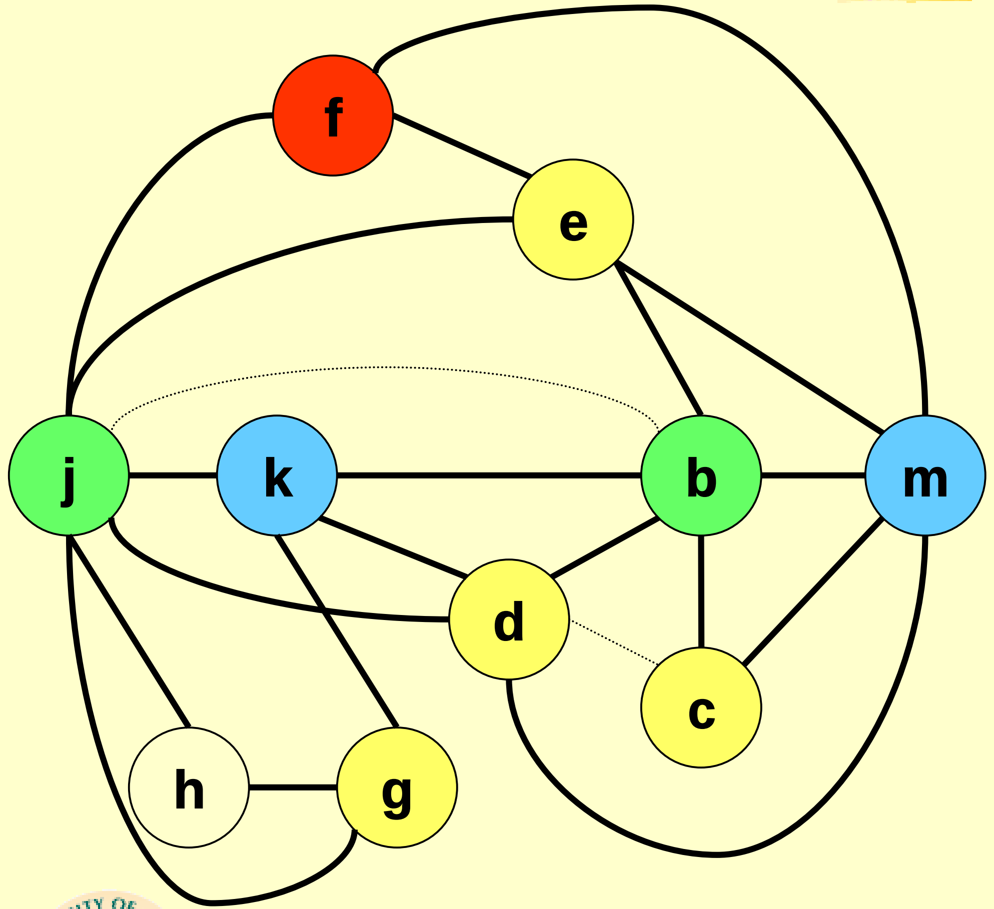
🟨 R2

🟦 R3

🟥 R4

(Appel, pp. 237)

Example: Step 3: Select (K=4)



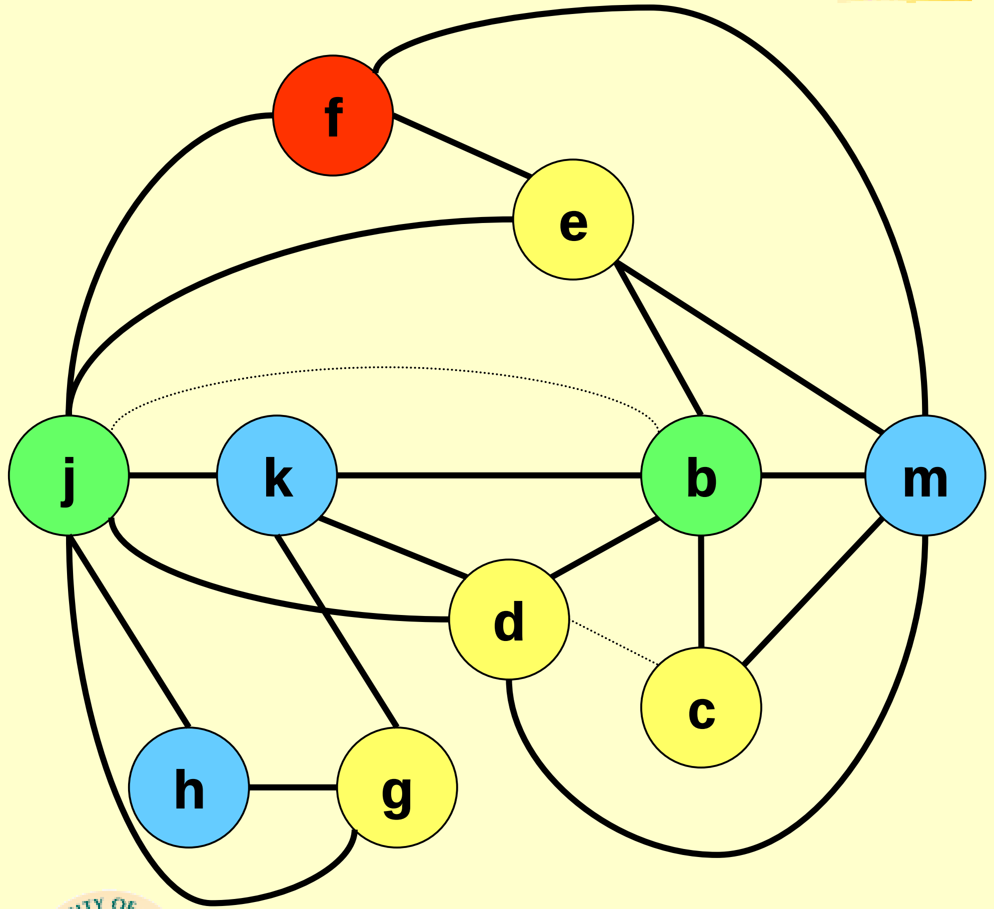
stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

🟩 R1
🟨 R2
🟦 R3
🟥 R4

(Appel, pp. 237)

Example: Step 3: Select (K=4)



stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

R1

R2

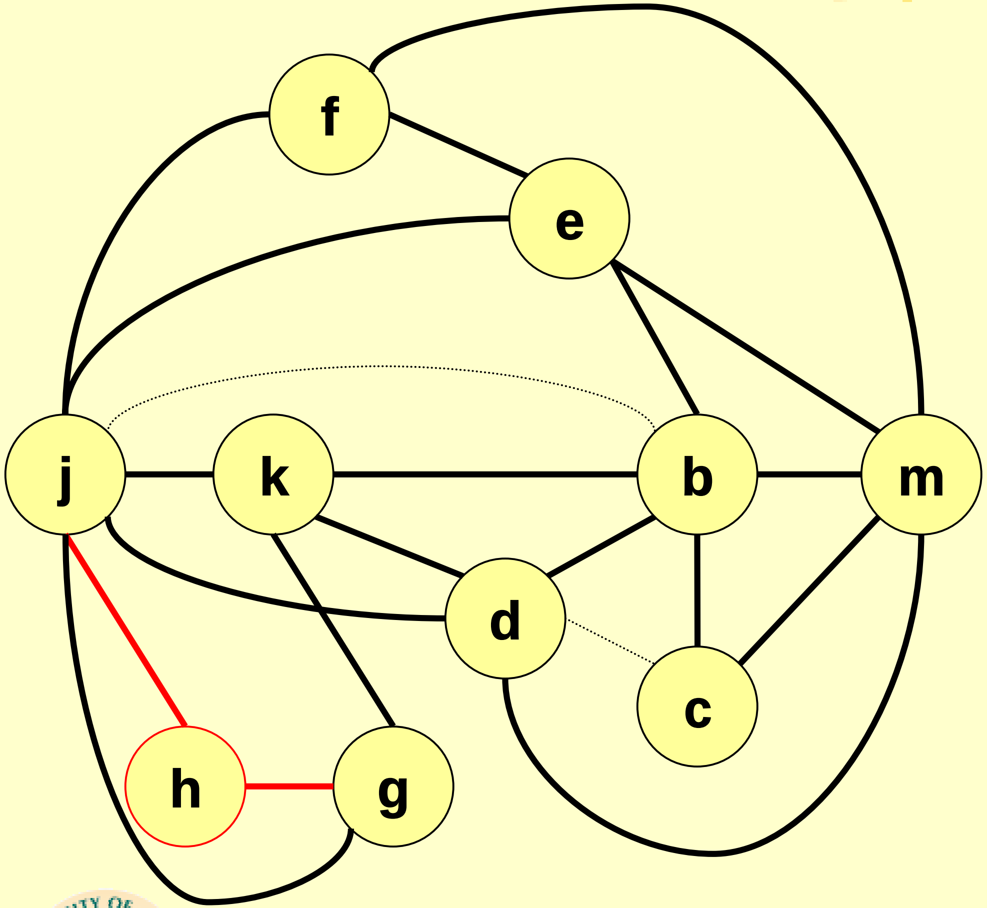
R3

R4

(Appel, pp. 237)

Could we do the allocation in the previous example with 3 registers?

Example:
Step 3: Simplify (K=3)

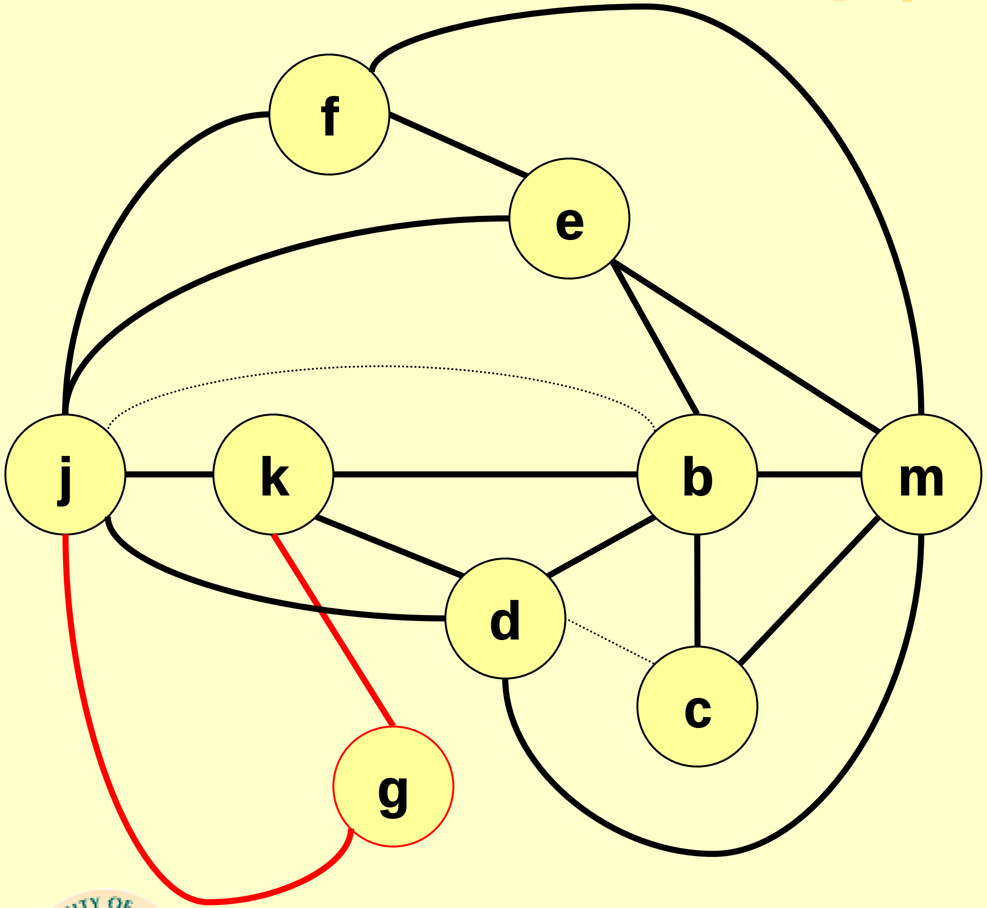


stack

(h,no-spill)

(Appel, pp. 237)

Example: Step 3: Simplify (K=3)

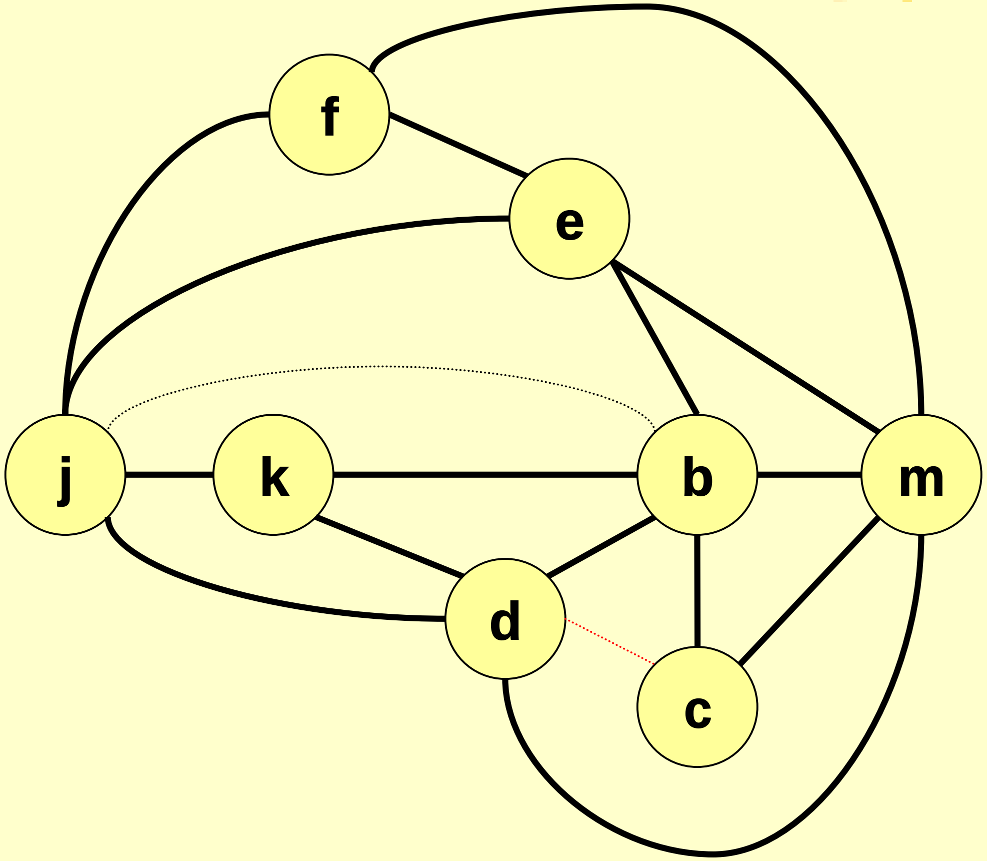


stack

(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 5: Freeze (K=3)



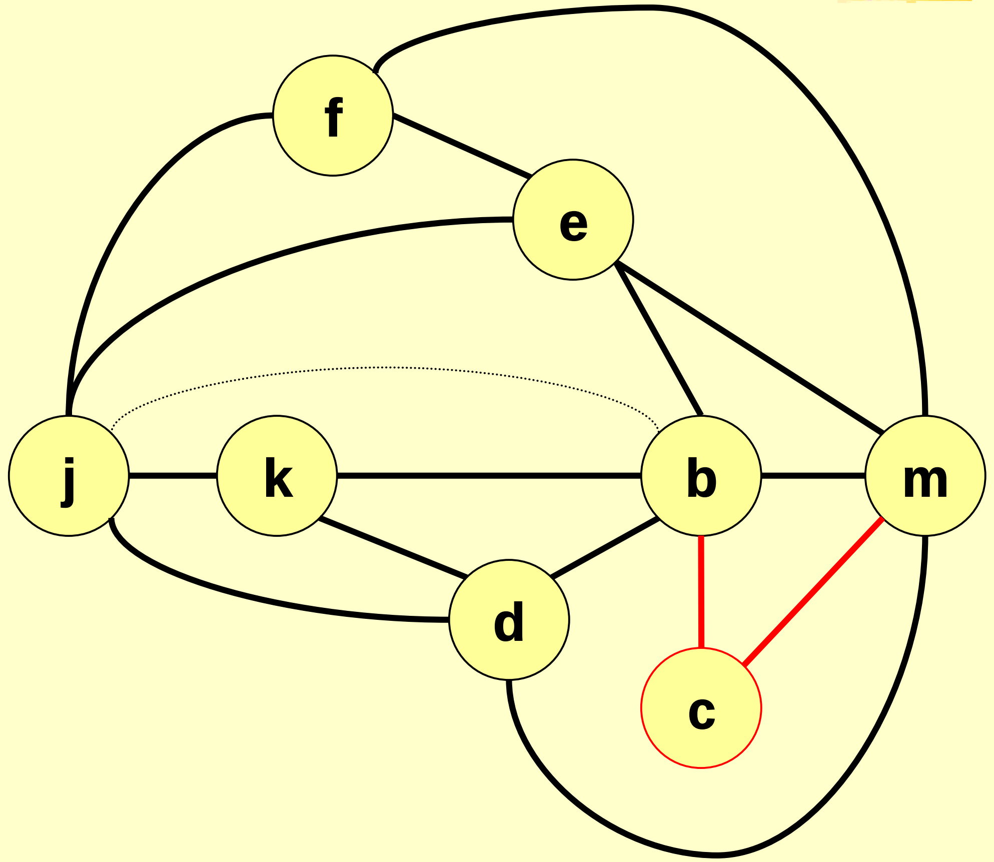
stack

(g, no-spill)
(h, no-spill)

Coalescing would make things worse.
We can freeze the move d-c.

(Appel, pp. 237)

Example: Step 3: Simplify (K=3)

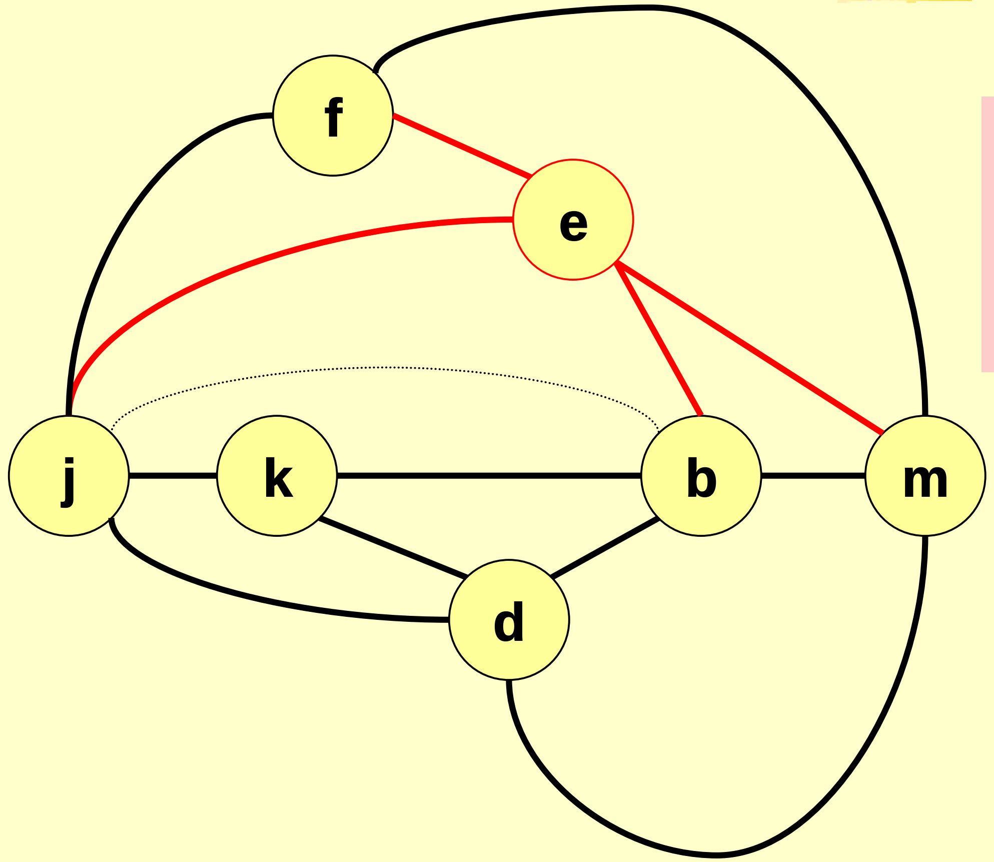


stack

(c, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 6: Spill (K=3)



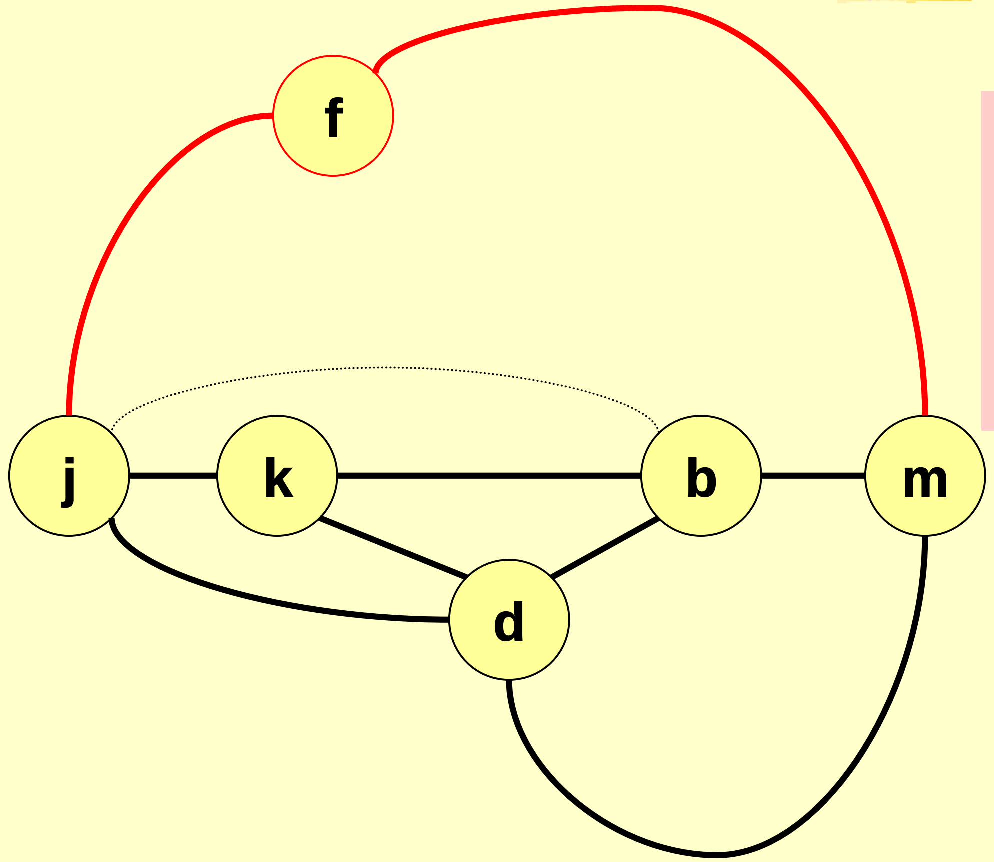
stack

(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

Neither coalescing nor freezing help us.
At this point we should use some profitability analysis to choose a node as *may-spill*.

(Appel, pp. 237)

Example: Step 3: Simplify (K=3)



stack

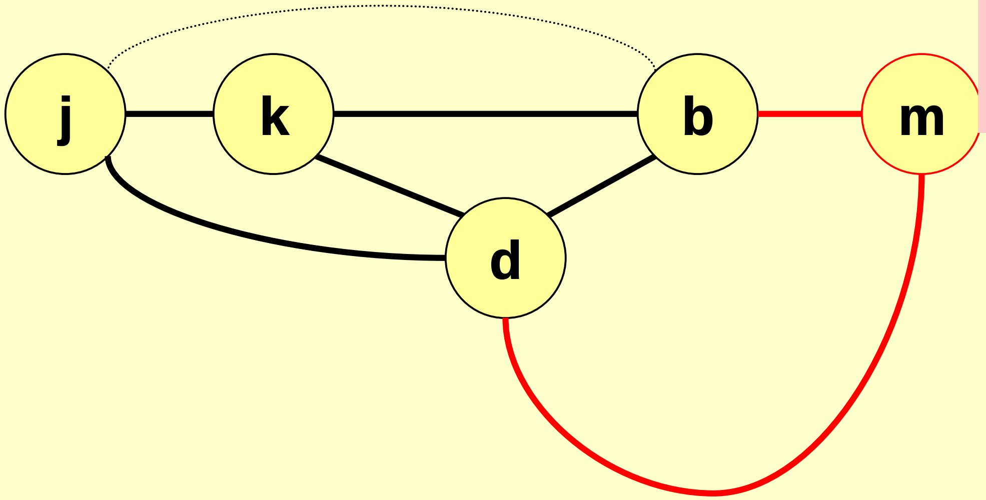
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Simplify (K=3)

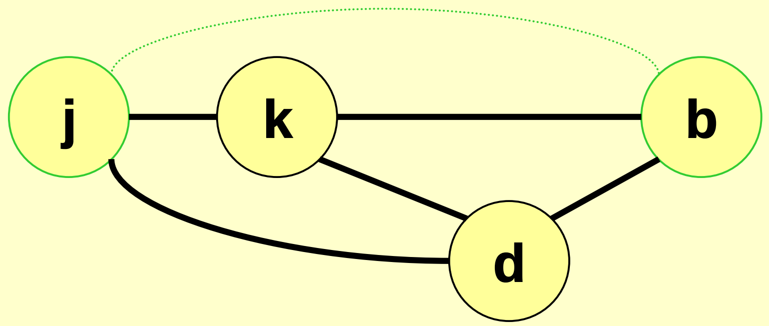
stack

(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)



(Appel, pp. 237)

Example:
Step 3: Coalesce (K=3)

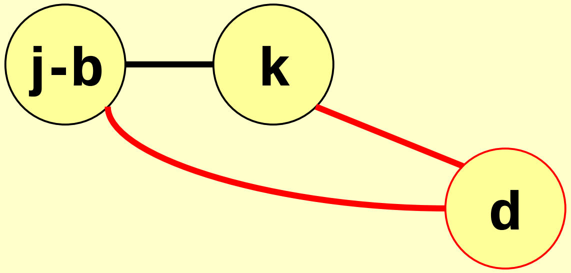


stack

(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example: Step 3: Coalesce (K=3)

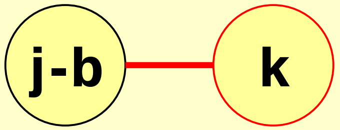


stack

(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Coalesce (K=3)

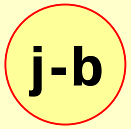


stack

(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example: Step 3: Coalesce (K=3)

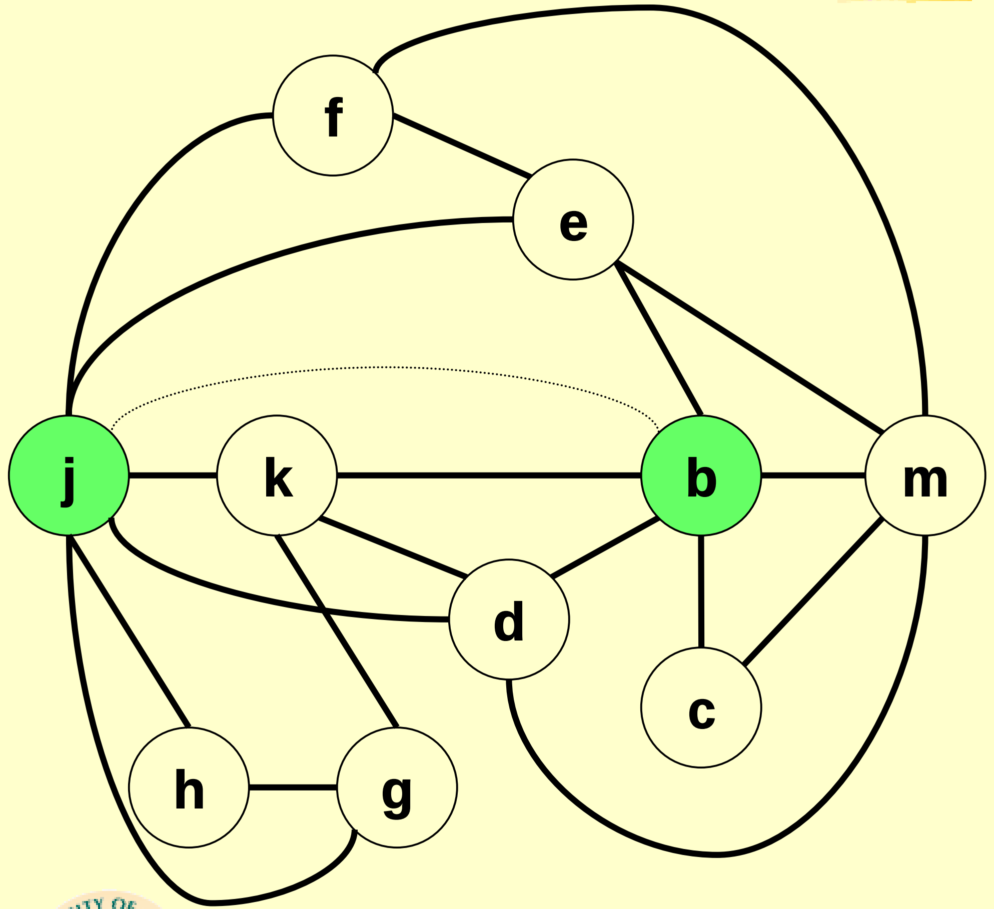


stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)

Example:
Step 3: Select (K=3)



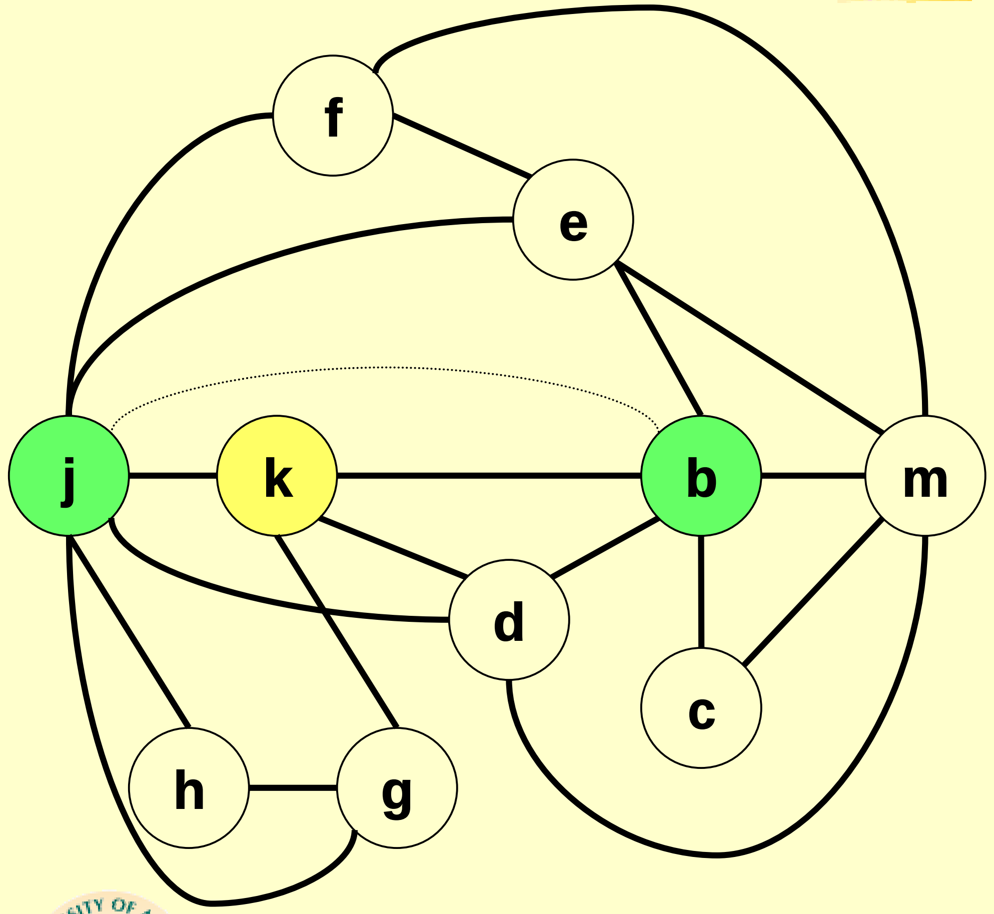
stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

🟩 **R1**
🟨 **R2**
🟥 **R3**

(Appel, pp. 237)

Example: Step 3: Select (K=3)



stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

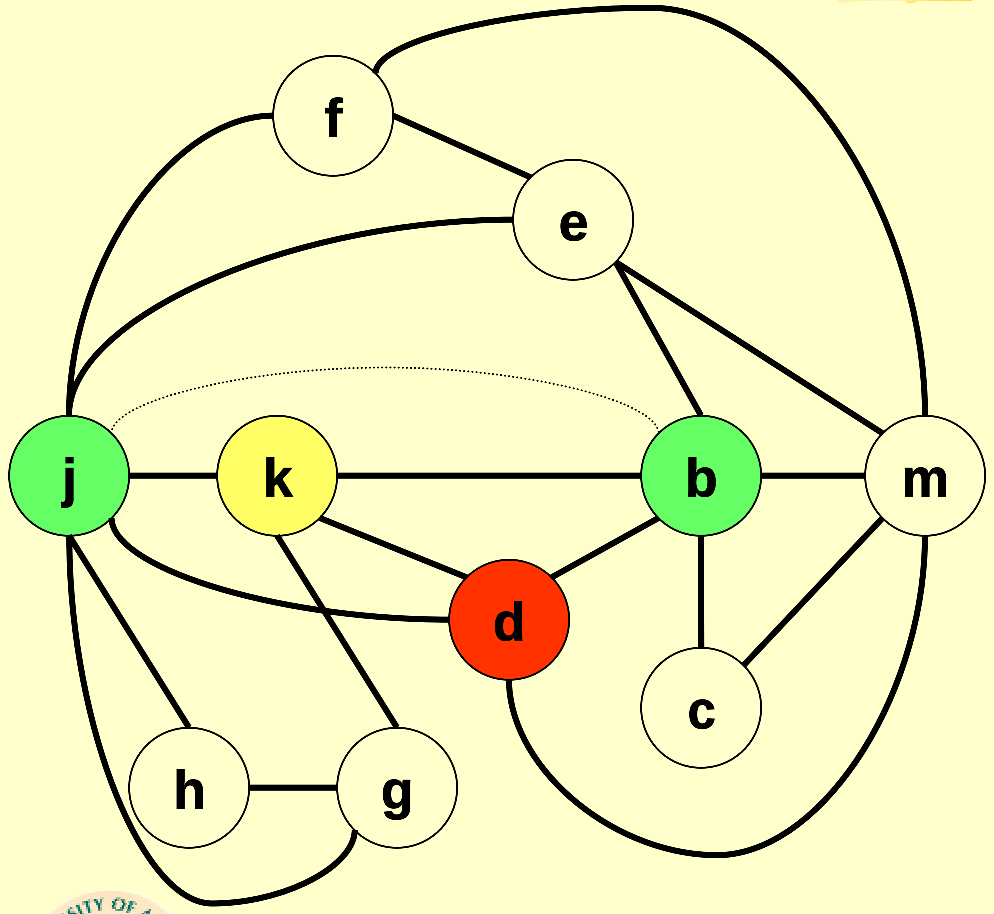
R1

R2

R3

(Appel, pp. 237)

Example:
Step 3: Select (K=3)



stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

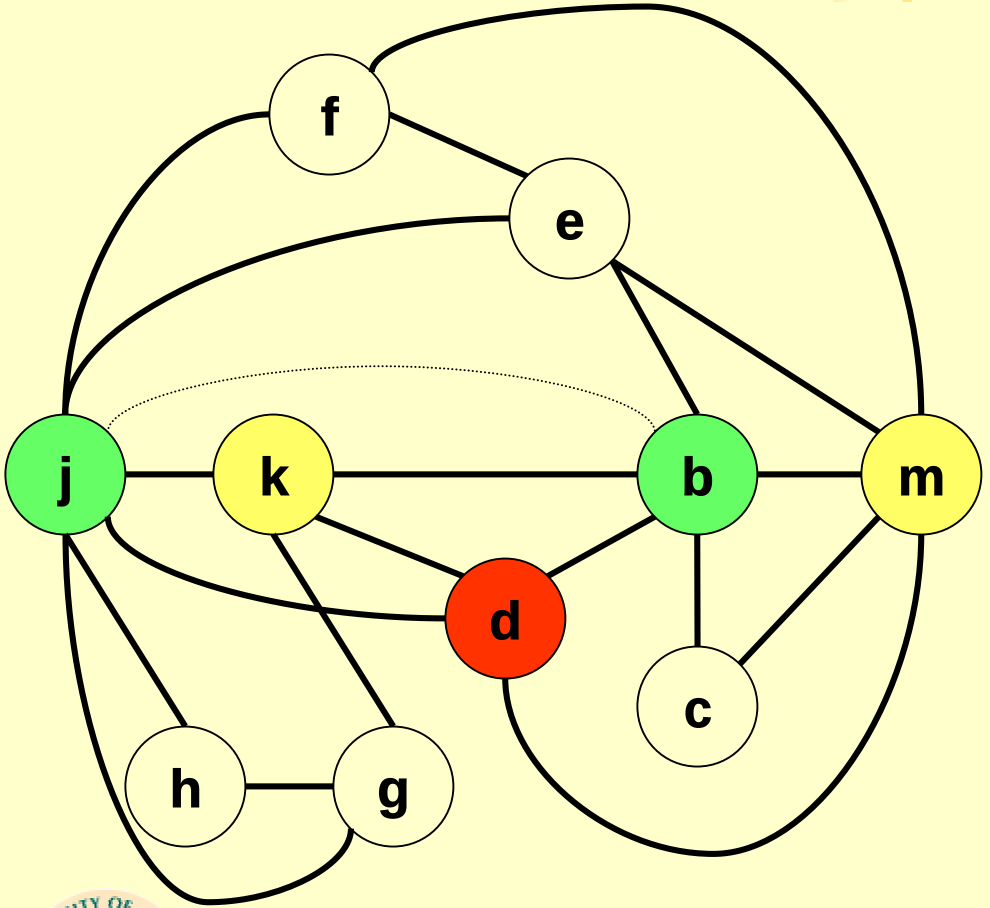
🟩 R1

🟨 R2

🟥 R3

(Appel, pp. 237)

Example:
Step 3: Select (K=3)



stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

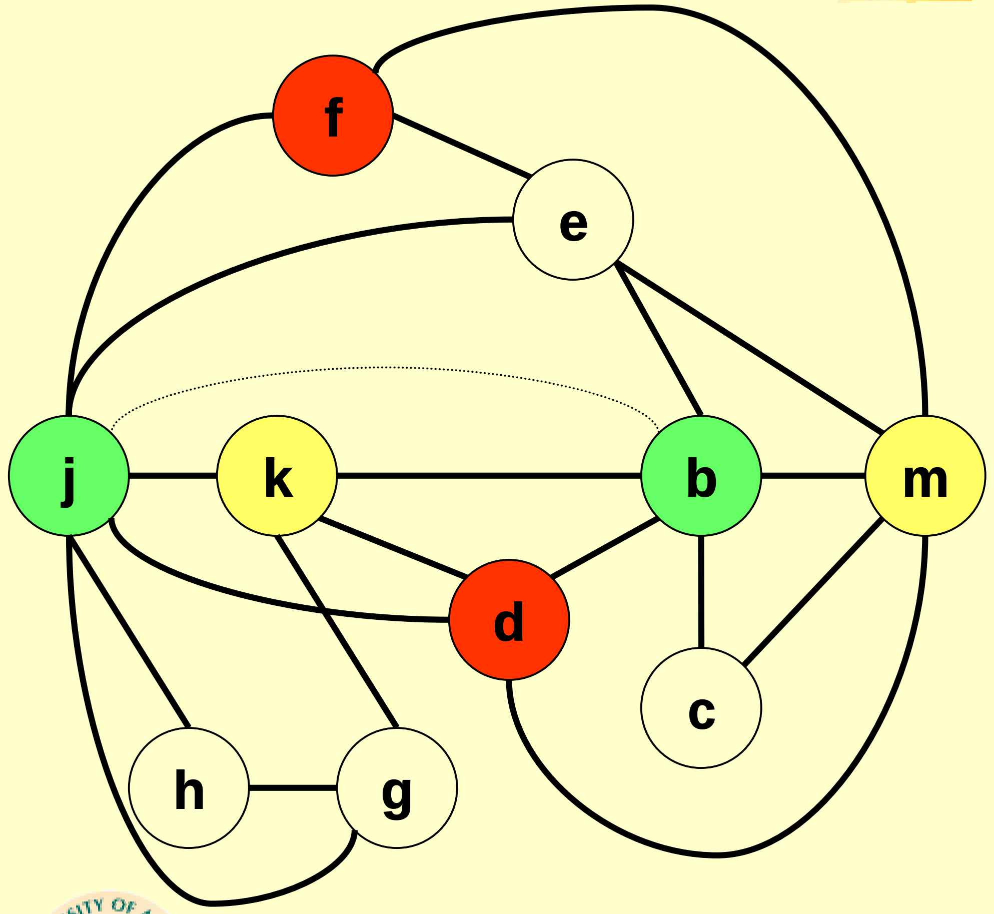
R1

R2

R3

(Appel, pp. 237)

Example: Step 3: Select (K=3)



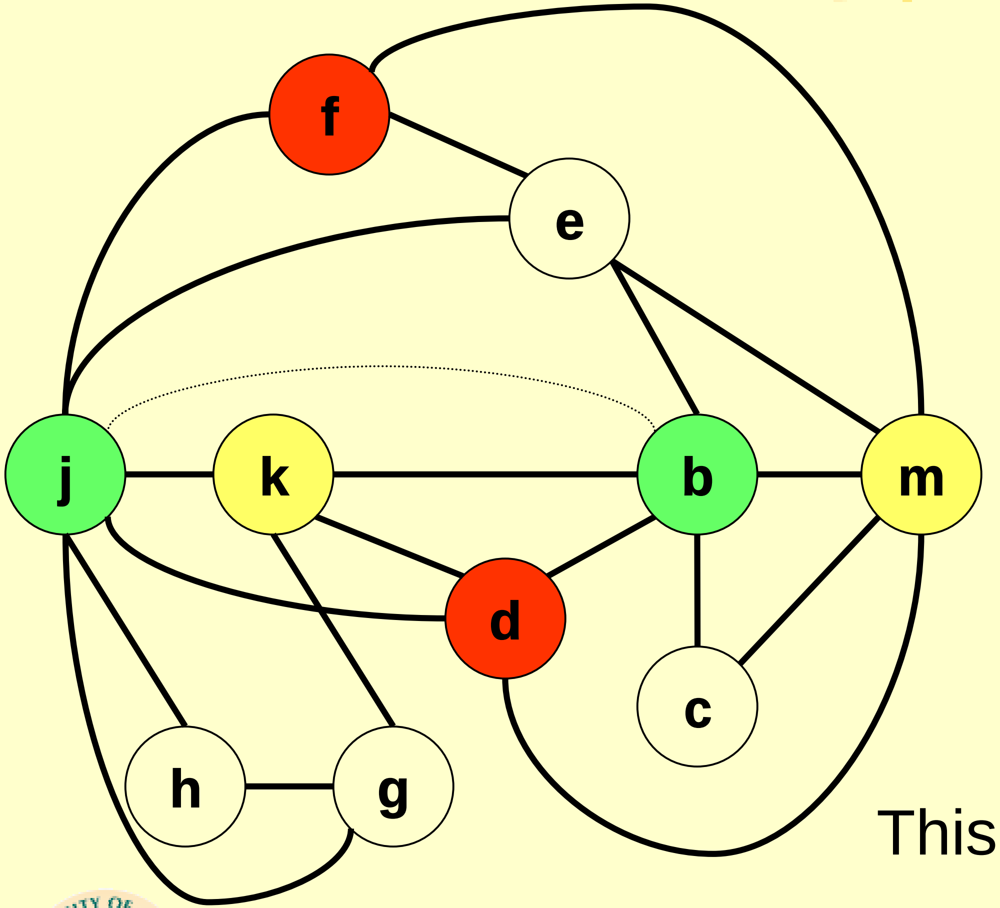
stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

R1
R2
R3

(Appel, pp. 237)

Example: Step 3: Select (K=3)



stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

R1

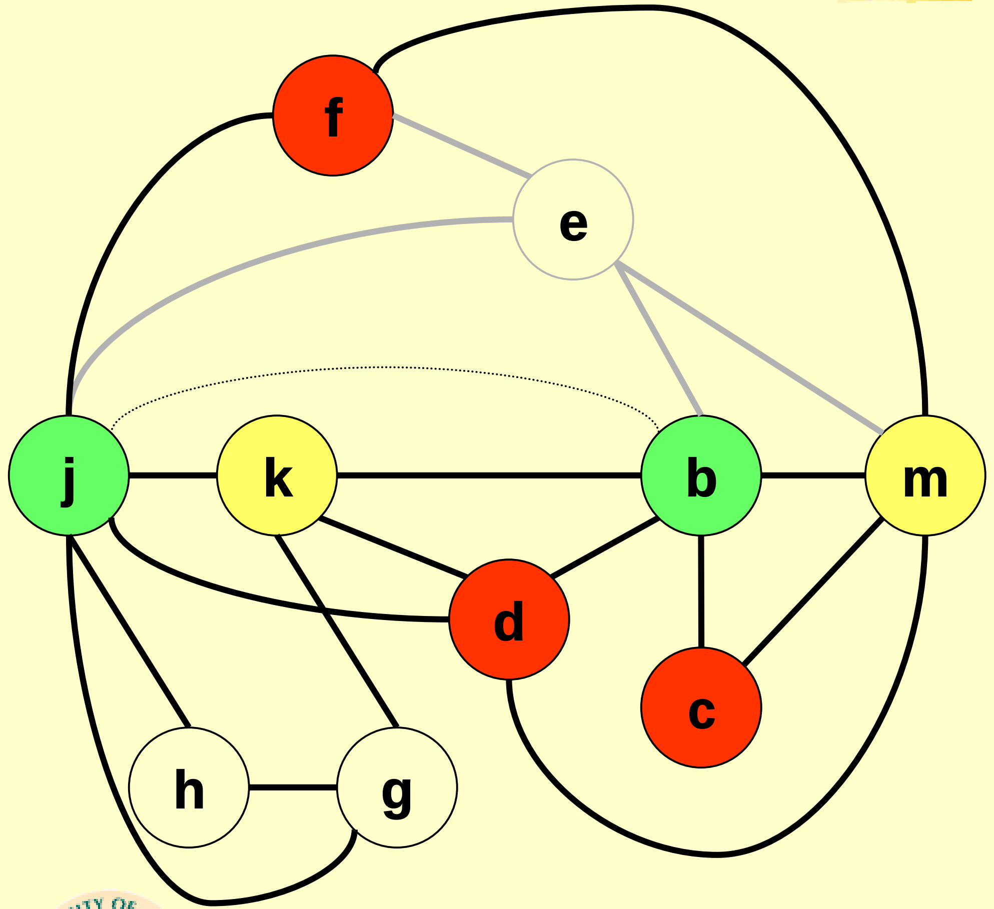
R2

R3

This is when our optimism could have paid off.

(Appel, pp. 237)

Example:
Step 3: Select (K=3)



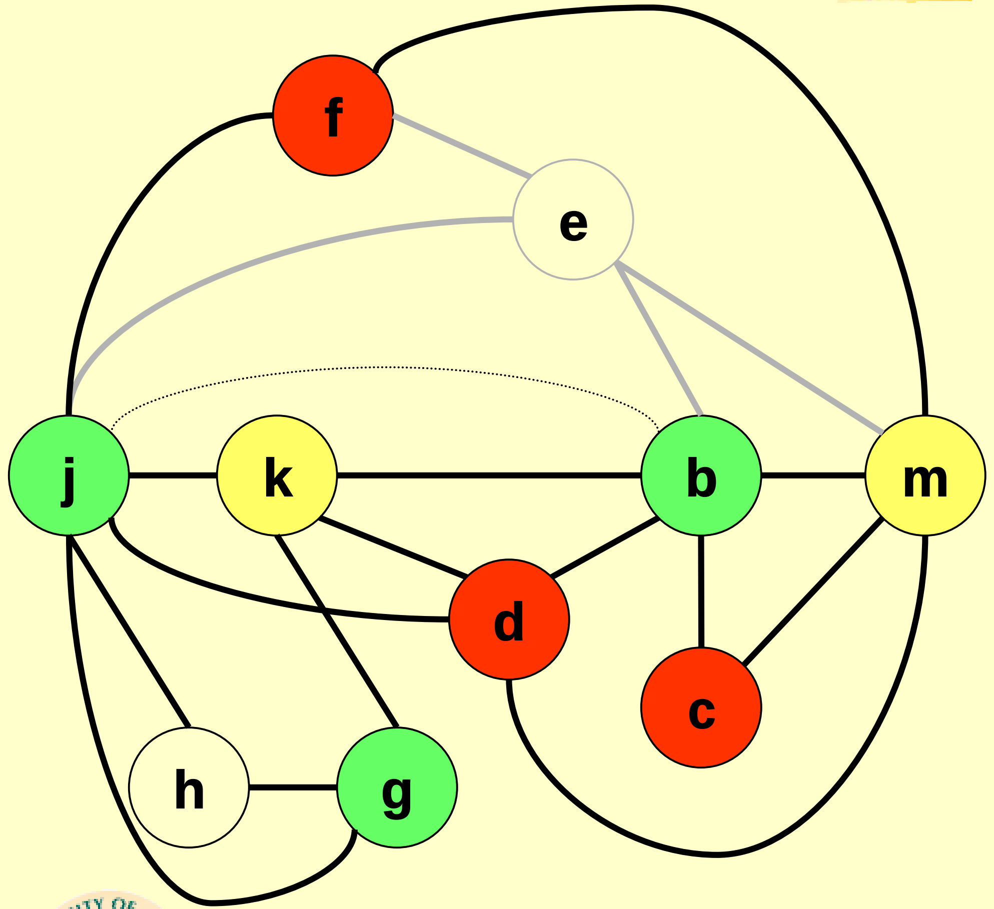
stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

🟩 R1
🟨 R2
🟥 R3

(Appel, pp. 237)

Example: Step 3: Select (K=3)



stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

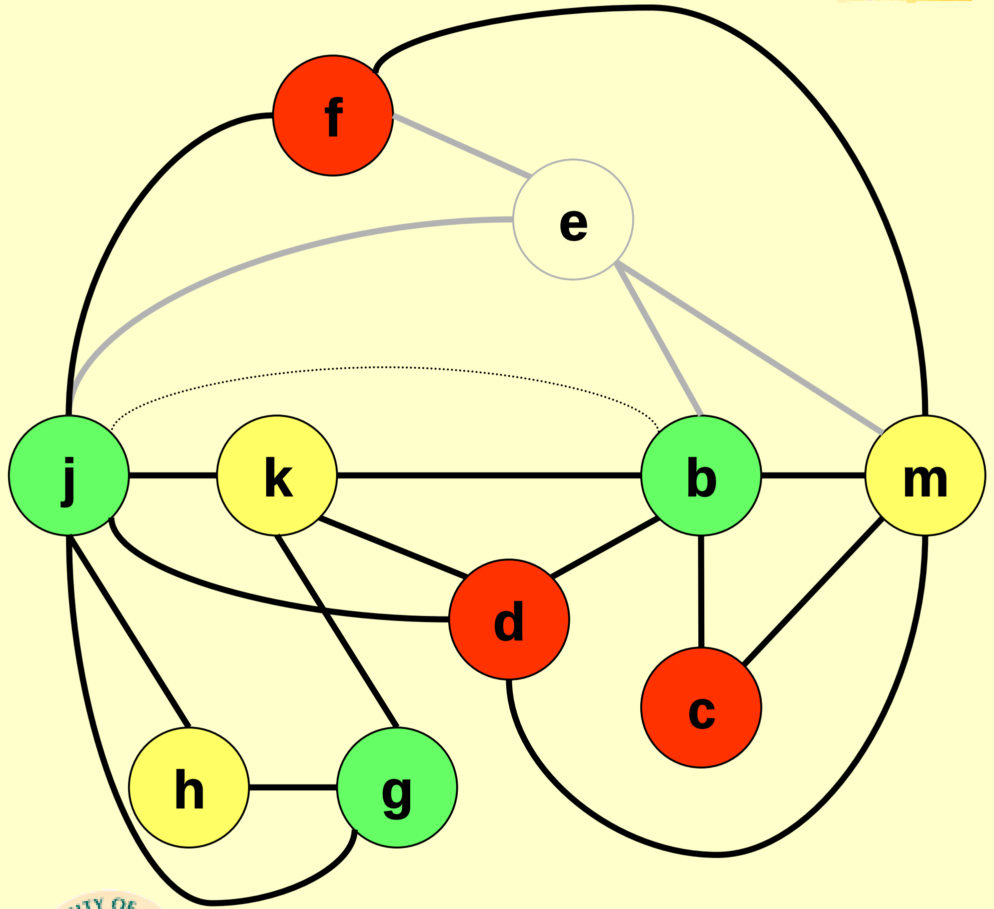
R1

R2

R3

(Appel, pp. 237)

Example: Step 3: Select (K=3)



stack

(j-b, no-spill)
(k, no-spill)
(d, no-spill)
(m, no-spill)
(f, no-spill)
(e, may-spill)
(c, no-spill)
(g, no-spill)
(h, no-spill)

🟩 **R1**

🟨 **R2**

🟥 **R3**

(Appel, pp. 237)

Live Range Splitting

The basic coloring algorithm does not consider cases in which a variable can be allocated to a register for part of its live range.

Some compilers deal with this by splitting live ranges within the iteration structure of the coloring algorithm i.e., by pretending to split a variable into two new variables, one of which might be profitably assigned to a register and one of which might not.

Length of Live Ranges

The interference graph does not contain information of where in the CFG variables interfere and what the lenght of a variable's live range is. For example, if we only had few available registers in the following intermediate-code example, the right choice would be to spill variable w because it has the longest live range:

```
x  =  w  +  1
c  =   a   -  2
y  =   x  *  3
z  =  w  +  y
```

Effect of Instruction Reordering on Register Pressure

The coloring algorithm does not take into account the fact that reordering IL instructions can reduce interference. Consider the following example:

Original Ordering (needs 3 registers)	Optimized Ordering (needs 2 registers)
t_1 := A[i]	t_2 ;= A[j]
t_2 := A[j]	t_3 := A[k]
t_3 := A[k]	t_4 := t_2 * t_3
t_4 := t_2 * t_3	t_1 := A[i]
t_5 := t_1 + t_4	t_5 := t_1 + t_4

Brief History of Register Allocation

Chaitin: ACM SIGPLAN Notices 1982	Use the simple stack heuristic for register allocation. Spill/no-spill decisions are made during the stack construction phase of the algorithm
Briggs: PLDI 1989	Finds out that Chaitin’s algorithm spills even when there are available registers. Solution: the optimistic approach: may-spill during stack construction, decide at spilling time.

Brief History of Register Allocation

Chow-Hennessy: Priority-based coloring.
SIGPLAN Integrate spilling decisions in the
1984 coloring decisions: spill a variable
ASPLOS for a limited life range.
1990 Favor dense over sparse use regions.
Consider parameter passing convention.

Callahan: Hierarchical Coloring Graph,
PLDI register preferencing,
1991 profitability of spilling.