# Coloring Heuristic

To R-Color a Graph (where R is the number of registers available)

1. While any node, n, has < R neighbors:
   Remove n from the Graph.
   Push n onto a Stack.

2. If the remaining Graph is non-empty:
   Compute the Cost of each node.
   The Cost of a Node (a Live Range) is the number of extra instructions needed if the Node isn't assigned a register, scaled by $10^{loop_depth}$.
   Let NB(n) =
   Number of Neighbors of n.
   Remove that node n that has the smallest Cost(n)/NB(n) value.
   Push n onto a Stack.
   Return to Step 1.

3. While Stack is non-empty:

Pop n from the Stack.

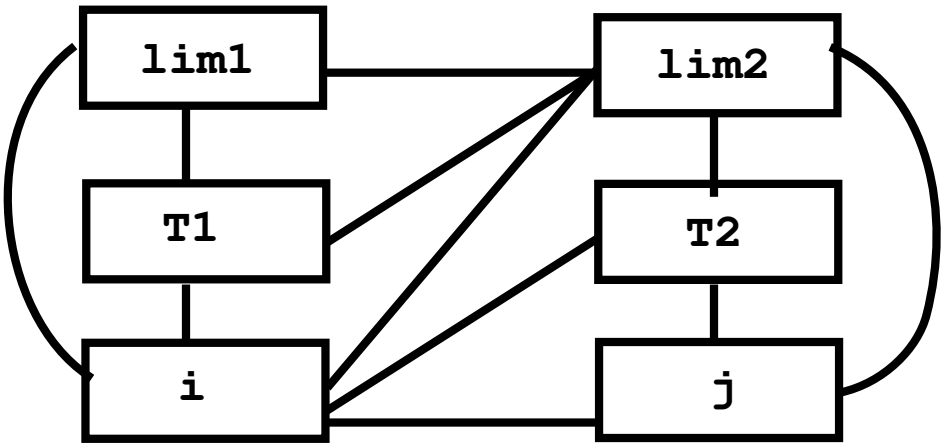
If n's neighbors are assigned fewer than R colors

Then assign n any unassigned color

Else leave n uncolored.

# Example

```
    int p(int lim1, int lim2) {
    int *T1 = &A[0];
    for (i=0; i<lim1 && *(T1+i)>0;i++){}
    int *T2 = &B[0];
    for (j=0; j<lim2 && *(T2+j)>0;j++){}
    return i+j;
}
```



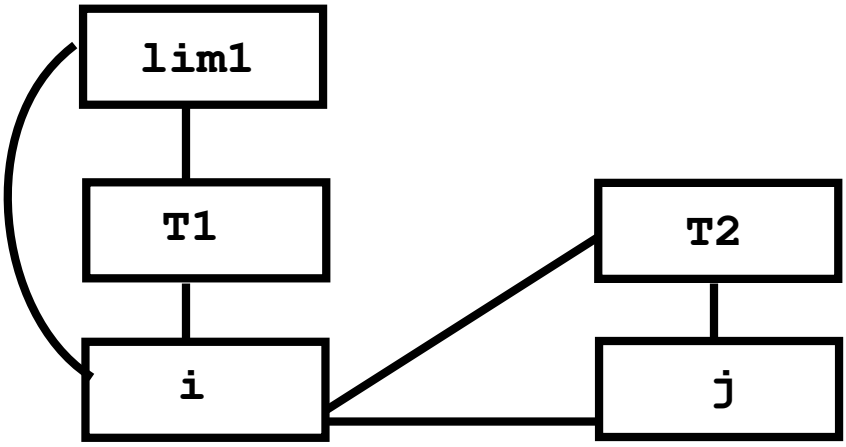

| | lim1 | lim2 | T1 | T2 | i | j |
|---|---|---|---|---|---|---|
| Cost | 11 | 11 | 11 | 11 | 42 | 42 |
| Cost/ Neighbors | 11/3 | 11/5 | 11/3 | 11/3 | 42/5 | 42/3 |

Do a 3 coloring

Since no node has fewer than 3 neighbors, we remove a node based on the minimum Cost/Neighbors value.

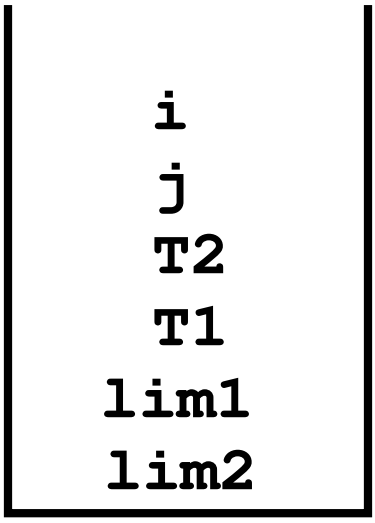
**`lim2`** is chosen.

We now have:



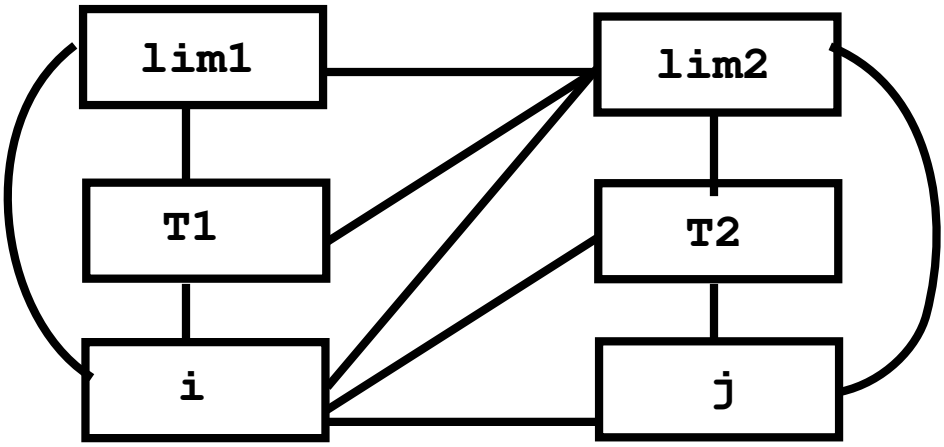

Remove (say) **`lim1`**, then **`T1`**, **`T2`**, **`j`** and **`i`** (order is arbitrary).

The Stack is:




Assuming the colors we have are R1, R2 and R3, the register assignment we choose is

**i**:R1, **j**:R2, **T2**:R3, **T1**:R2, **lim1**:R3, **lim2**:spill



# Color Preferences

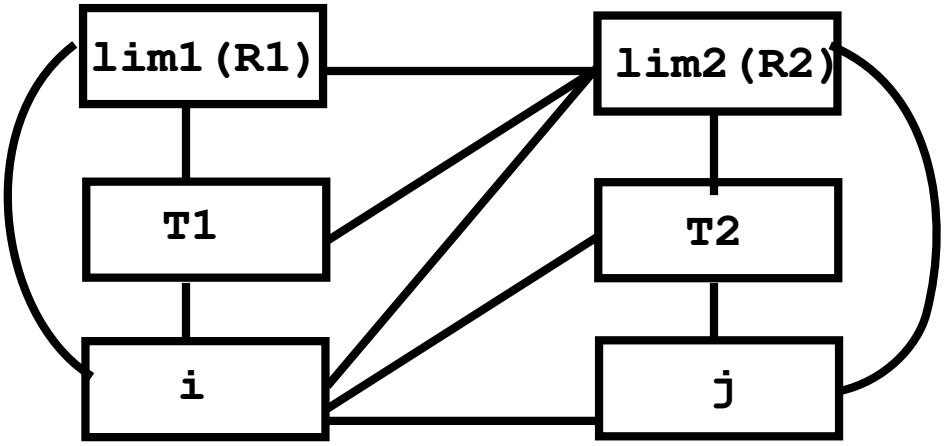
Sometimes we wish to assign a particular register (color) to a selected Live Range (e.g., a parameter or return value) *if possible*.

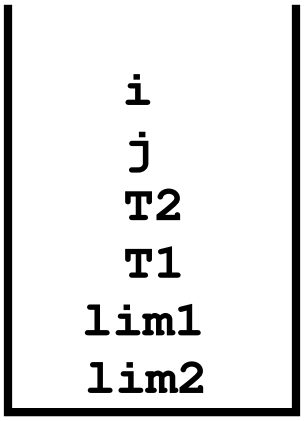
We can mark a node in the Interference Graph with a *Color Preference*.

When we unstack nodes and assign colors, we will avoid choosing color c if an uncolored neighbor has indicted a preference for it. If only color c is left, we take it (and ignore the preference).

# Example

Assume in our previous example that lim1 has requested register R1 and lim2 has requested register R2 (because these are the registers the parameters are passed in).




Now when `i`, `j` and `T1` are unstacked, they respect `lim1`'s and `lim2`'s preferences:

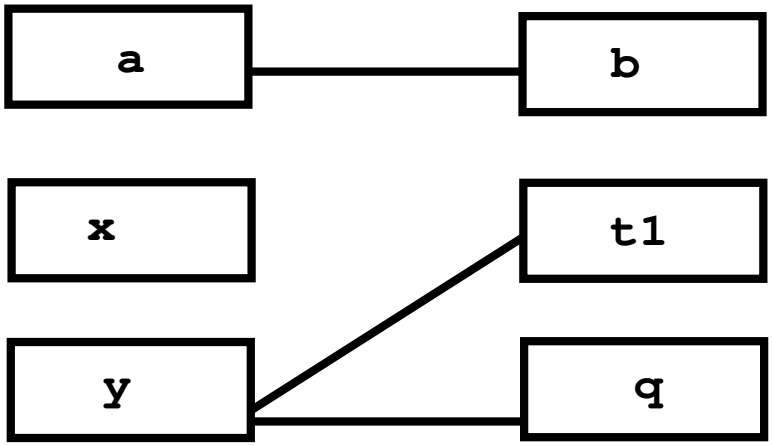
`i`:R3, `j`:R1, `T2`:R2, `T1`:R2, `lim1`:R1, `lim2`:spill

# Using Coloring to Optimize Register Moves

A nice "fringe benefit" of allocating registers via coloring is that we can often *optimize away* register to register moves by giving the source and target the *same color*.

Consider

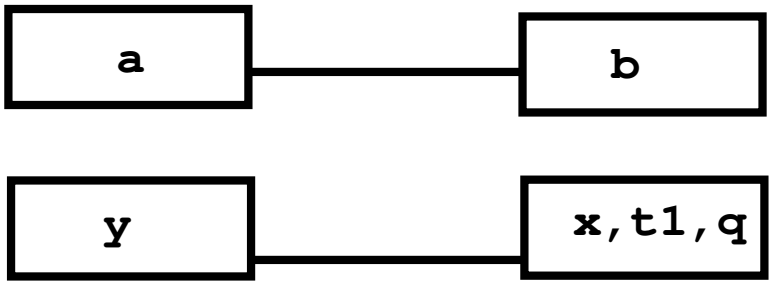
```
Live in: a,b

t1 = a + b

x = t1

y = x + 1

q = t1

Live out: y,q
```




We'd like **x**, **t1** and **q** to get the same color. How do we "force" this?

We can "merge" **x**, **t1** and **q** together:

```
Live in: a,b

t1 = a + b

x = t1

y = x + 1

q = t1

Live out: y,q
```



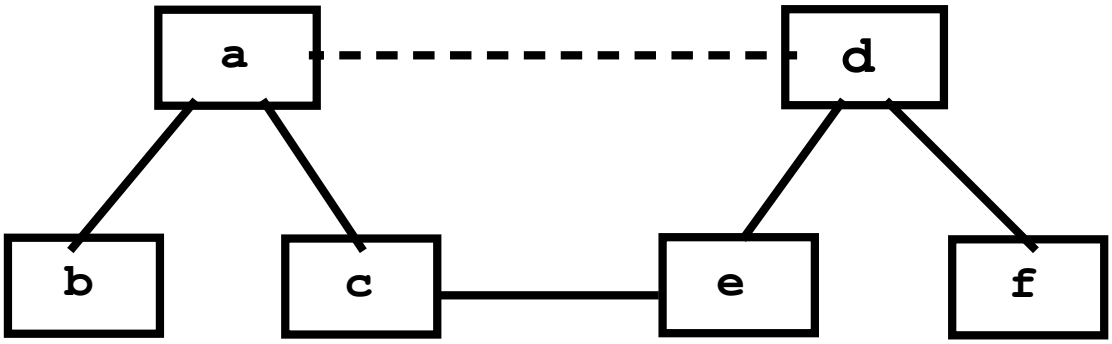

Now a 2-coloring that optimizes away both register to register moves is trivial.

# Reckless Coalescing

Originally, Chaitin suggested merging *all* move-related nodes that don't interfere.

This is *reckless*—the merged node may not be colorable!

(Is it worth a spill to save a move??)




This Graph is 2-colorable before the reckless merge, but *not* after.

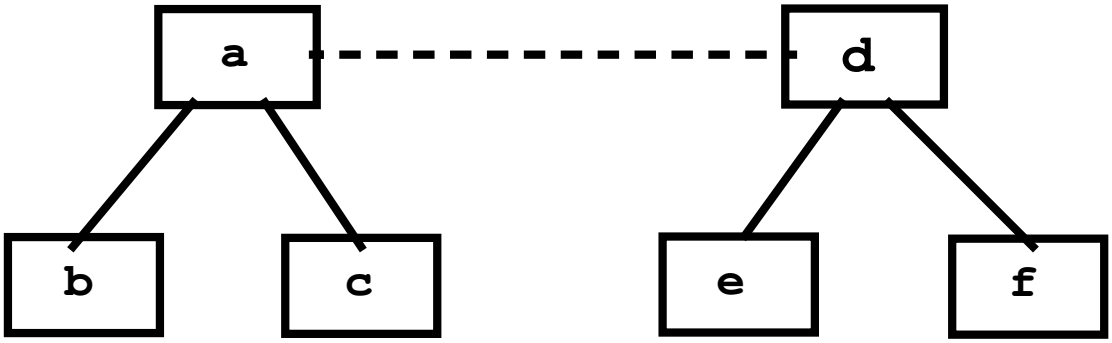
# Conservative Coalescing

In response to Chaitin's reckless coalescing approach, Briggs suggested a *more conservative* approach.

See "Improvement to Graph Coloring Register Allocation," P. Briggs et. al., ACM Toplas, May 1994.

Briggs suggested that two move-related nodes should be merged *only if* the combined source and target node has fewer than R neighbors.

This *guarantees* that the combined node will be colorable, but may miss some optimization opportunities.




After a merge of nodes **a** and **d**, there will be four neighbors, but a 2-coloring is still possible.

# Iterated Coalescing

This is an intermediate approach, that seeks to be safer than reckless coalescing and more effective than conservative coalescing. It was proposed by George and Appel.

1. Build:

   Create an Interference Graph, as usual. Mark source-target pairs with a special move-related arc (denoted as a dashed line).

2. Simplify:

   Remove and stack non-move-related nodes with < R neighbors.

3. Coalesce:

   Combine move-related pairs that will have < R neighbors after coalescing.

Repeat steps 2 and 3 until only nodes with R or more neighbors or move-related nodes remain or the graph is empty.

4. Freeze:

If the Interference Graph is non-empty:

Then If there exists a move-related node with < R neighbors

Then: "Freeze in" the move and make the node non-move-related.

Return to Steps 2 and 3.

Else: Use Chaitin's Cost/Neighbors criterion to remove and stack a node.

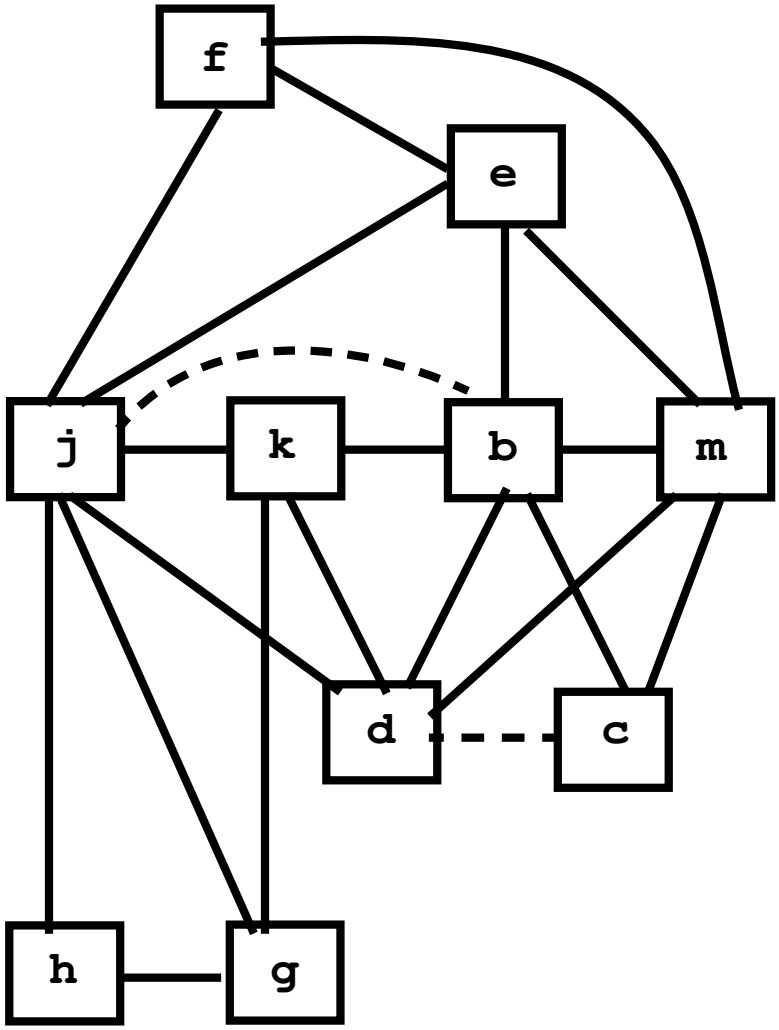
Return to Steps 2 and 3.

5. Unstack:

Color nodes as they are unstacked as per Chaitin and Briggs.

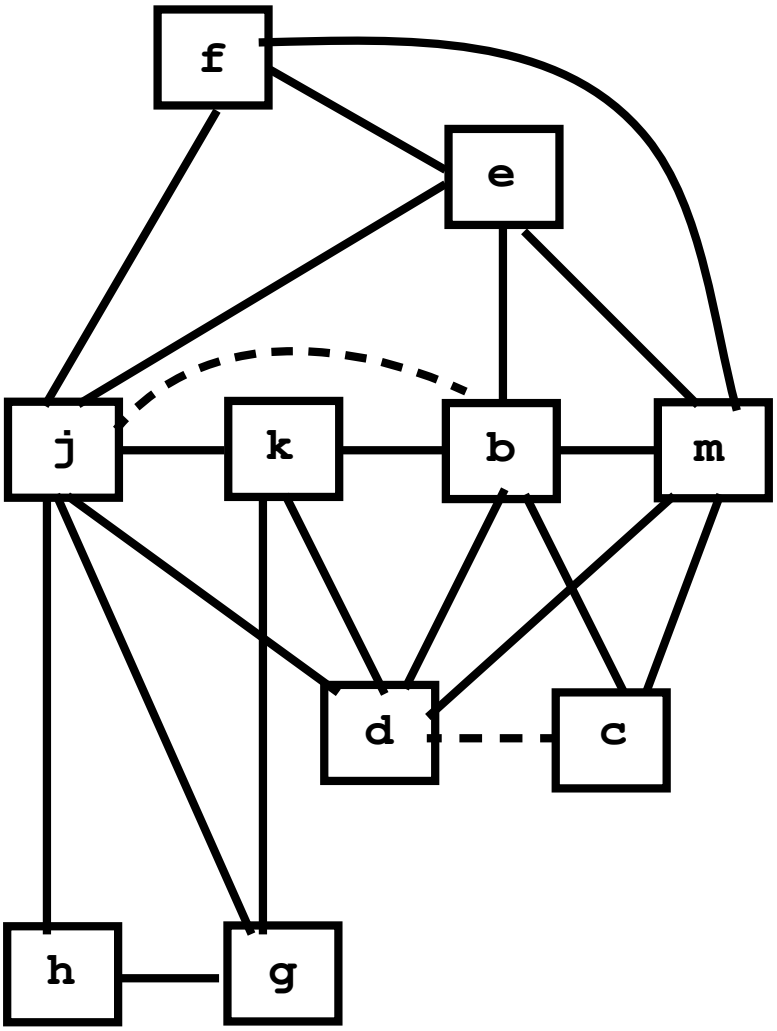
# Example

```
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
goto d
Live out: d,k,j
```




Assume we want a 4-coloring.

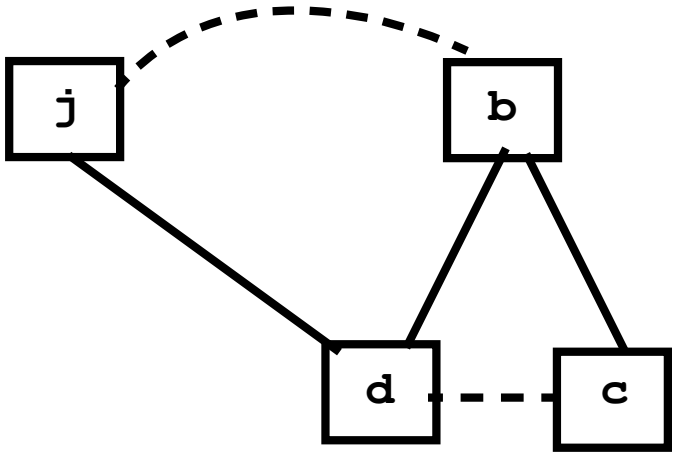
Note that neither **j&b** nor **d&c** can be conservatively colored.



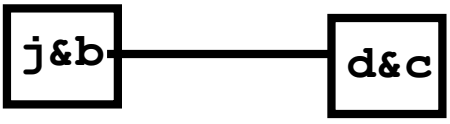

We simplify by removing nodes with fewer than 4 neighbors.

We remove and stack: **g, h, k, f, e, m**

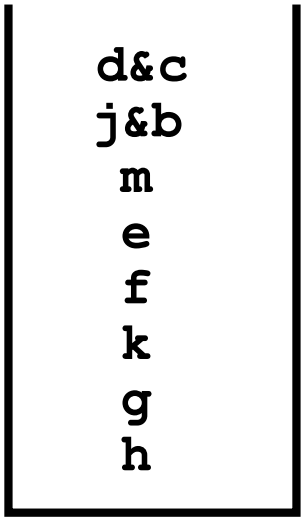
The remaining Interference Graph is




We can now conservatively coalesce the move-related pairs to obtain




These remaining nodes can now be removed and stacked.




We can now unstack and color:

**d&c**:R1, **j&b**:R2, **m**:R3, **e**:R4, **f**:R1, **k**:R3, **h**:R1, **g**:R4

No spills were required and both moves were optimized away.