Register Allocation

We want to store live variables in registers.

Why?

- Accessing a value from a register takes 1 instruction cycle
- Accessing a value from cache takes 3 to 5 cycles
- Accessing a value from RAM takes 20 to 100 cycles.

The OS determines what is in cache and what is in RAM; the compiler determines what is in a register.

Here is the idea of register allocation: suppose the runtime environment has R available registers and a particular function uses D <= R variables, including parameters, local variables and temporaries. We can speed up the program by

a) assigning each variable to a register
b) at the start of the function load each variable into its register
c) throughout the function use only the registers
d) at the end of the function "spill" all of the registers back to the memory locations associated with their variables.

This is sweet, only there are problems:

- All bets are off if the body of the function calls another function (which might trash the registers).
- If a variable is only referenced once, this wastes time writing it into a register and then writing it back to memory.
- There are usually more variables than registers.

- We can't help the problem with calling other functions. Dataflow analysis is only applied to the portions of code between calls. Some heuristics are applied to determine whether any such optimization is worthwhile.
- To avoid the problems with variables that are used only once or twice, it is pretty easy to walk down a flowgraph and count the number of references. Note that anything used in a loop can be assumed to be referenced many times.
- Register allocation algorithms assume these problems have been handled and we have a flow graph where we know the live variables at every point on the graph.

The Register Allocation Problem is determining how to assign variables to the available registers.

The most common approach to register allocation uses a Graph Coloring algorithm due to G.J. Chaitin (1981, plus lots of follow-up papers) Chaitin was a researcher at IBM's TJ Watson research center.

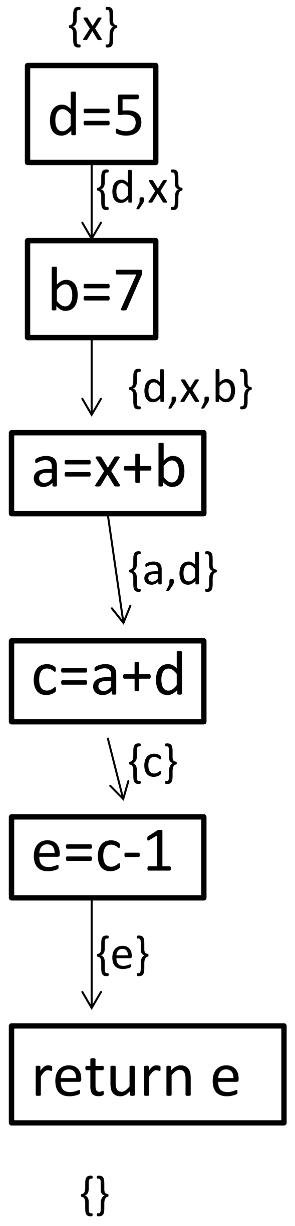
For starters, notice that two variables can use the same register if they are never live at the same time. For example x might be live in statements s1-s7, then y might be live in s10 -s15, then x might be live again in s20-s28. Suppose they cohabit register R. If x comes alive by being assigned the value of 23 in statement s1, we just write that value into R. After statement s7 we write R back to x's location in memory. Similarly, y would come alive in s10 through assignment, which we would make into R

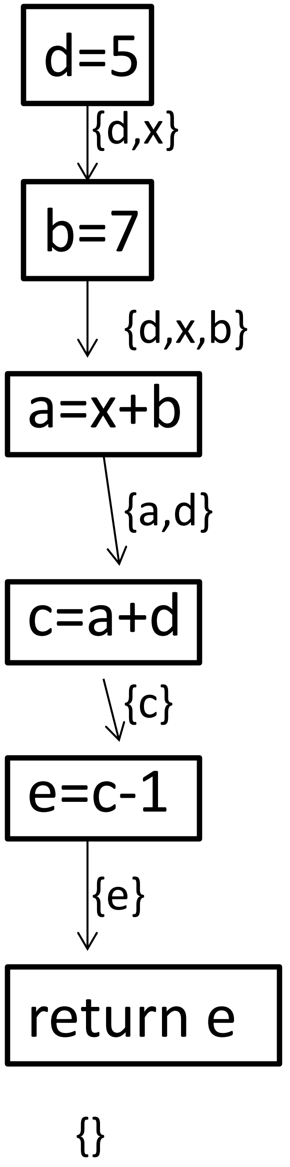
The graph coloring idea is to draw a graph where the nodes are the variables and edges link every pair of variables that are live at the same time. This is called the *Interference Graph.* We color the variables according to the register assignments. In other words, if we can color the variables with {red, green, blue} then we put all of the red variables into one register, all of the green ones into another, and so forth.

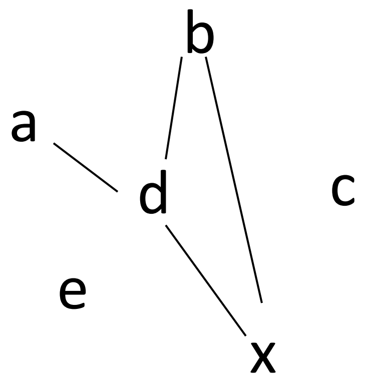
Of course, graph coloring is NP-Hard and we need to do this quickly, but that is what makes life interesting.

Example:

```
int f(x) {
    int a, b, c, d, e;
    d = 5;
    b = 7;
    a = x+b;
    c=a+d;
    e=c-1;
    return e;
}
```





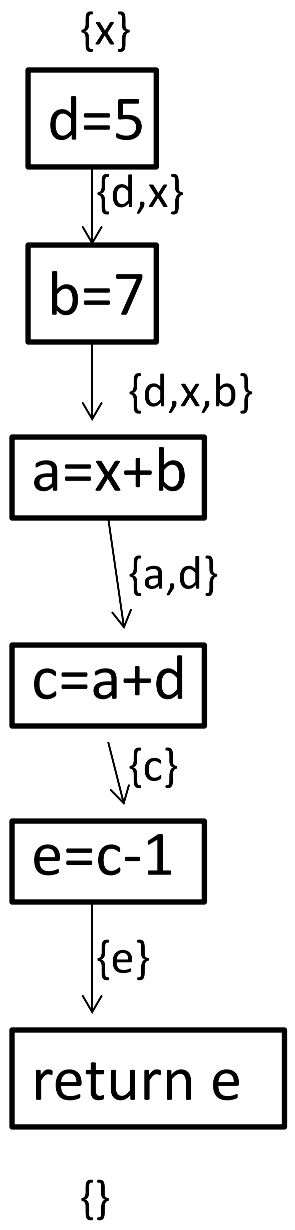


We could do this with just 3 registers:

r1: {x}

r2: {a,c,e,b}

r3: {d}



r1: {x}   r2: {a,c,e,b} r3: {d}

The code becomes:

```
move x to r1
r3=5
r2=7
r2=r1+r2
r2=r2+r3
r2=r2-1
return r2
```

So here's the methodology for register allocation:

- For each function we build a flow graph.
- We do a live variable analysis on the flow graph.
- We build an interference graph for the variables, connecting any pair that are live at the same time.
- We run a graph coloring algorithm on the flow graph. If we can color it with the number of registers we have available, we are golden. Otherwise we choose a subset of the variables to store in registers, the rest will be stored in memory.

Here is a lemma that will help with our graph coloring:

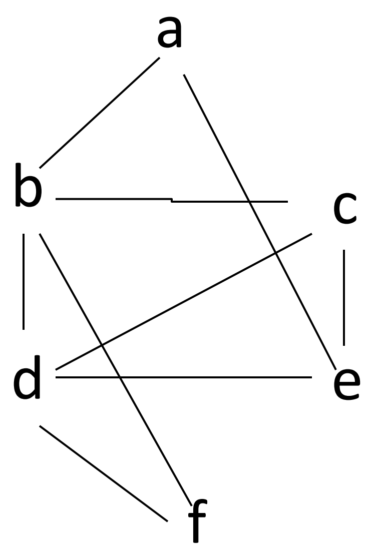
Lemma: Let G be an interference graph. Let P be a node in G with fewer than k neighbors. Let G' be the subgraph of G formed by eliminating P and all of its edges. If G' is k-colorable than so is G.
Proof: Suppose G' is k-colorable. Let $c_1...c_n$ be the neighbors of P in G, where $n<k$. These neighbors use at most n colors in G'. Let *col* be a color from G' not used by $c_1...c_n$ We can return node P to G with color *col* and give the rest of the nodes in G their colors in G'. This provides a k-coloring of G.

Now, to k-color an interference graph, we repeatedly apply this lemma. Repeat the following until the graph has k or fewer nodes:

- Pick a node t with fewer than k neighbors in the graph. Hopefully there is one.
- Eliminate t and its edges from the graph.

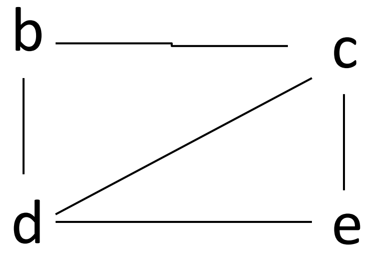
When the graph has k or fewer nodes it must have a k-coloring. We can add the removed nodes back, in the reverse of the order in which they were removed (last removed node is added first) while preserving the k-coloring.

For example, suppose we want a 3-coloring of the following graph:

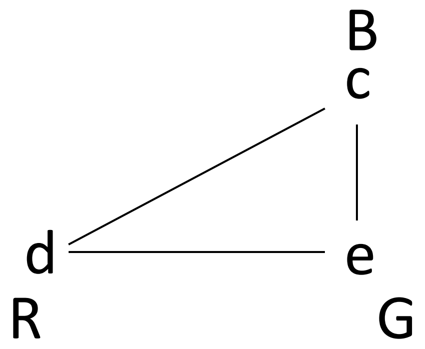


We can remove node a then f, because each has only 2 neighbors in the graph.

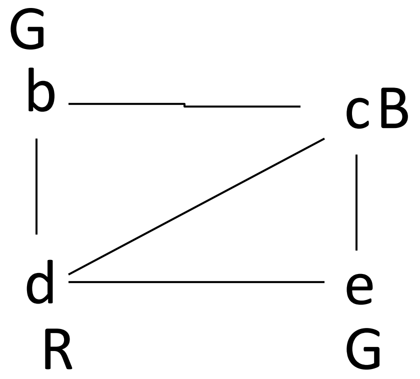
This leaves us with



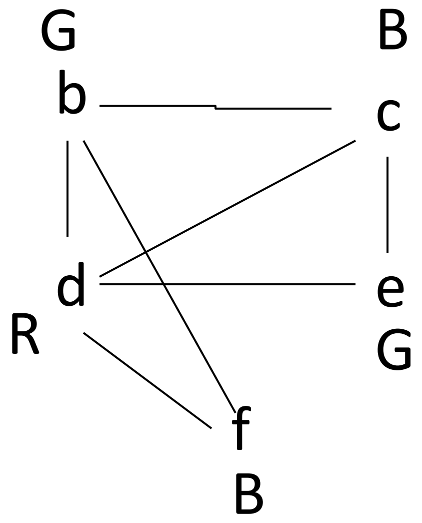
Node b now has only 2 neighbors; it can be removed, leaving us with 3 nodes that can be colored with 3 colors:



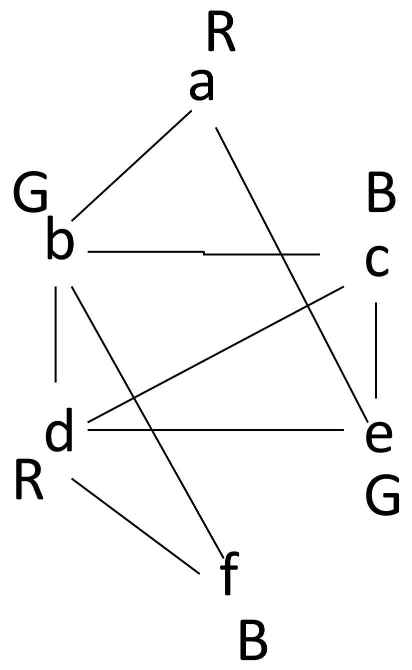
We add back b and its edges, with the only color it can be:



Then f:



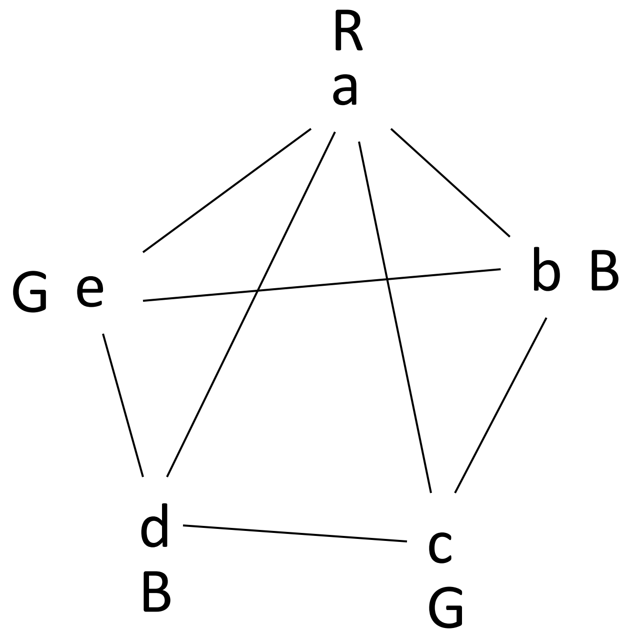
Finally, we add back in a, the first node we removed:



Of course, there is no guarantee that there will be a node in the graph with fewer than k neighbors. In this case all we can do is to choose a node that we will *not* store in a register, remove this and its edges from the graph, and continue with the algorithm. Possible heuristics for choosing a node to remove are

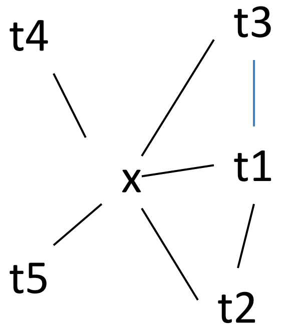
a) Nodes with many neighbors, to try to help the rest of the algorithm work
b) Nodes with few uses, so there is less cost in not storoing them in registers

We know the coloring problem is NP-hard and this is a polynomial-time algorithm for finding a k-coloring. There has to be a catch. The catch is that it doesn't always work. Here is a graph with a 3-coloring where every node has at least 3 edges:



Here is a simple example. Consider the following program which evaluates the polynomial $f(x)=x^2+3x-5$ over 10 values of x between 0.0 and 1.0:

```
x = 0.0                         {x}
while (x<1.0) {
                                {x}
        t1 = x*x
                                {x,t1}
        t2 = 3*x
                                {x,t1,t2}
        t3 = t2-5
                                {x,t1,t3}
        t4 = t1+t3
                                {x,t4}
        print(t4)
                                {x}
        t5 = x+0.1
                                {x,t5}
        x = t5
                                {x}
}
```



If we have 3 registers available our algorithm will find a 3-coloring for this graph.

Making the assignments:

x in R1

t1 in R2

all other variables in R3 results in the following code:

```
R1 = 0.0
while (R1<1.0) {
        R2 = R1*R1
        R3 = 3*R1
        R3 = R3-5
        R3 = R2+R3
        print(R3)
        R3 =R1+0.1
        R1 = R3
}
```

One more example:

```
b=2
c=3
f=5      {b,c,f}
while (b<100) {
                 {b,c,f}
        a=b+c
                 {a,c,f}
        d=a
                 {c,d,f}
        e=d+f
                 {c,d,e,f}
        if (e < 10)
                f=2*e
        else
                b=d+e
        b=f+c
}                {b,c,e,f}
e=e-1
```