Graph-Coloring and Register Allocation

CS132

March 28, 2011

Recall

The most common approach to register allocation is:

- Build an interference graph
- Find a "coloring" of this graph.

For each instruction s:

- If s is a move instruction **c** $\leftarrow$ **a** then add edges between **c** and every member of $liveout(s)$ except **a**.
- Otherwise, add an edge between every node in $def(s)$ and every temporary in $liveout(s)$.

Implementation Difficulties

What if we have k registers but there isn't a k-coloring?

How can we hope to solve an NP-Complete Problem?

Spilling

A temporary in memory (e.g., the stack frame) instead of a register is said to have been *spilled*.

Spilling transformation:

- Given a temporary **`t`** and a memory location M for it.
- Rewrite uses of **`t`** to load from M into a fresh temporary, and to use that
- Rewrite defns of **`t`** to write to a fresh temporary and to store that into M.

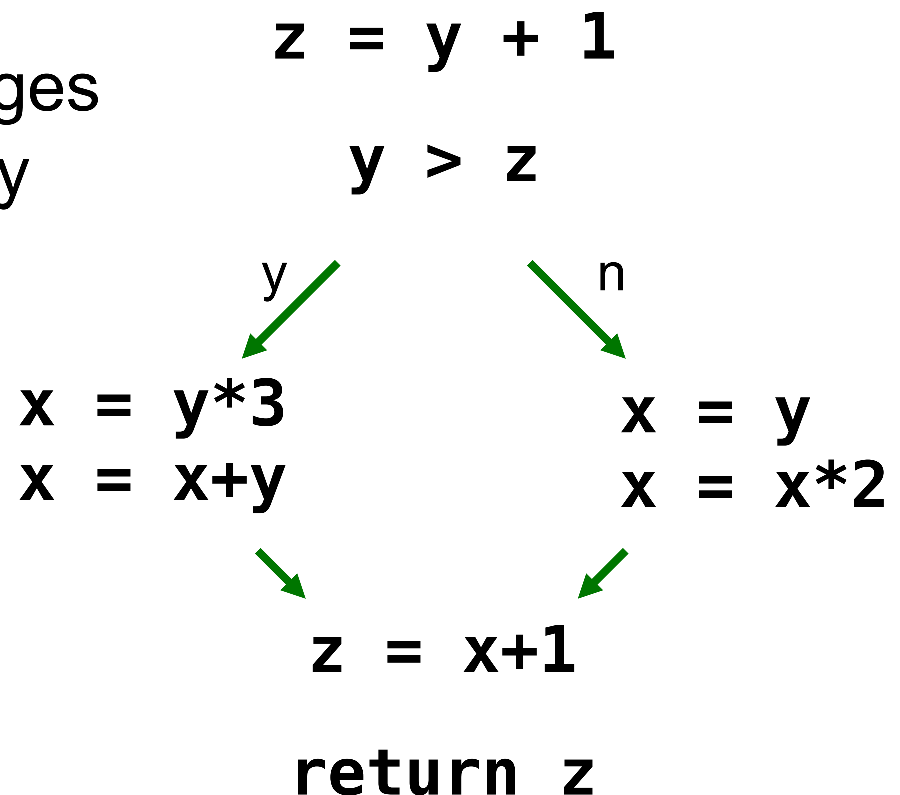
Gets rid of the temporary **`t`**, at the cost of adding new temporaries (with very short live ranges).

Live Ranges

The live range of a temporary is the part of the program where that temporary is live.

Variables with large live ranges tend to interfere with many other variables.

Variables with short live ranges tend to interfere with few other variables.



Spilling Example

```
a ← x+1
b ← x+2
x ← a+x
```

spill **x**

to *(%fp-8)

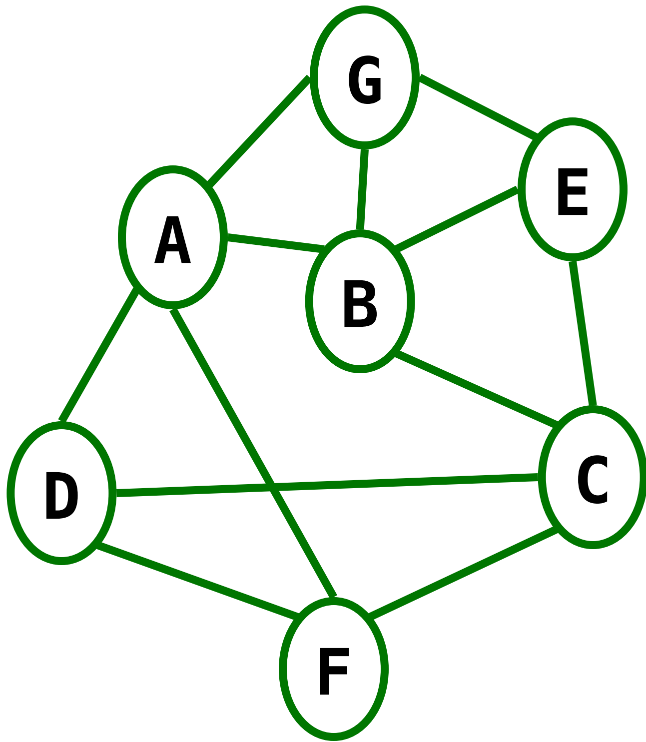
Could we "optimize away" redundant loads?

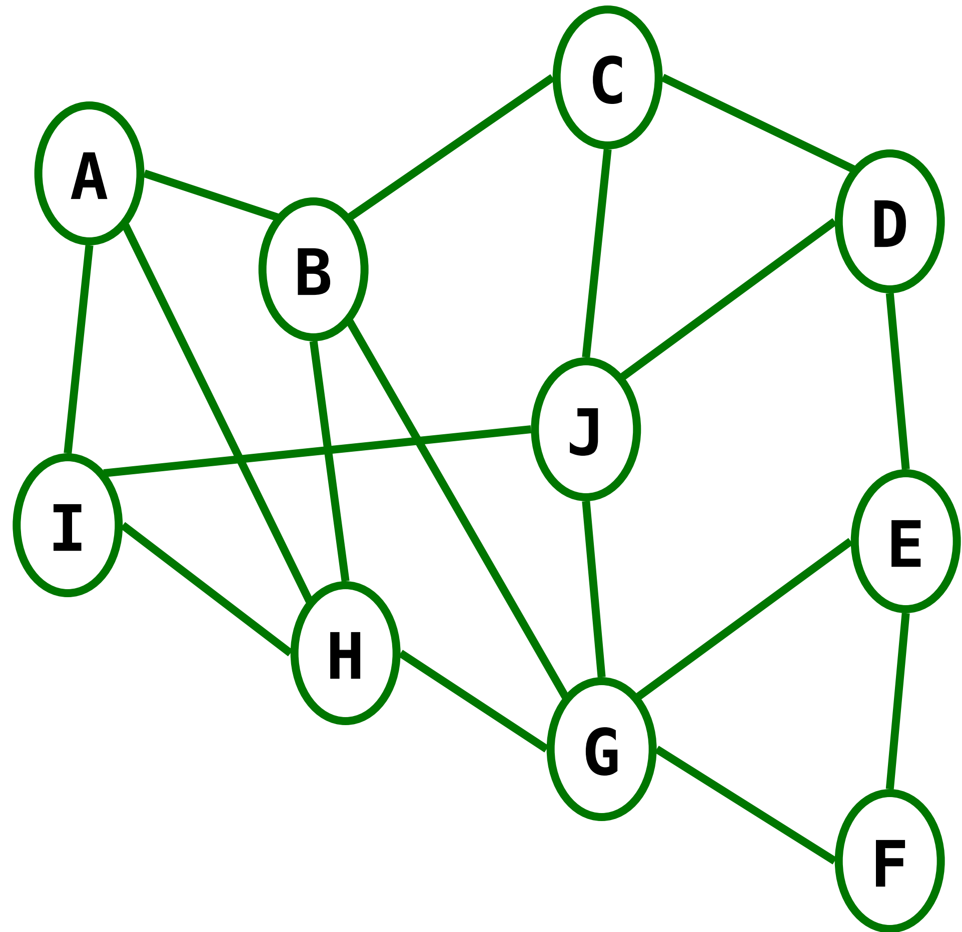
Graph Coloring Algorithm

Naive algorithm for k-coloring a graph:

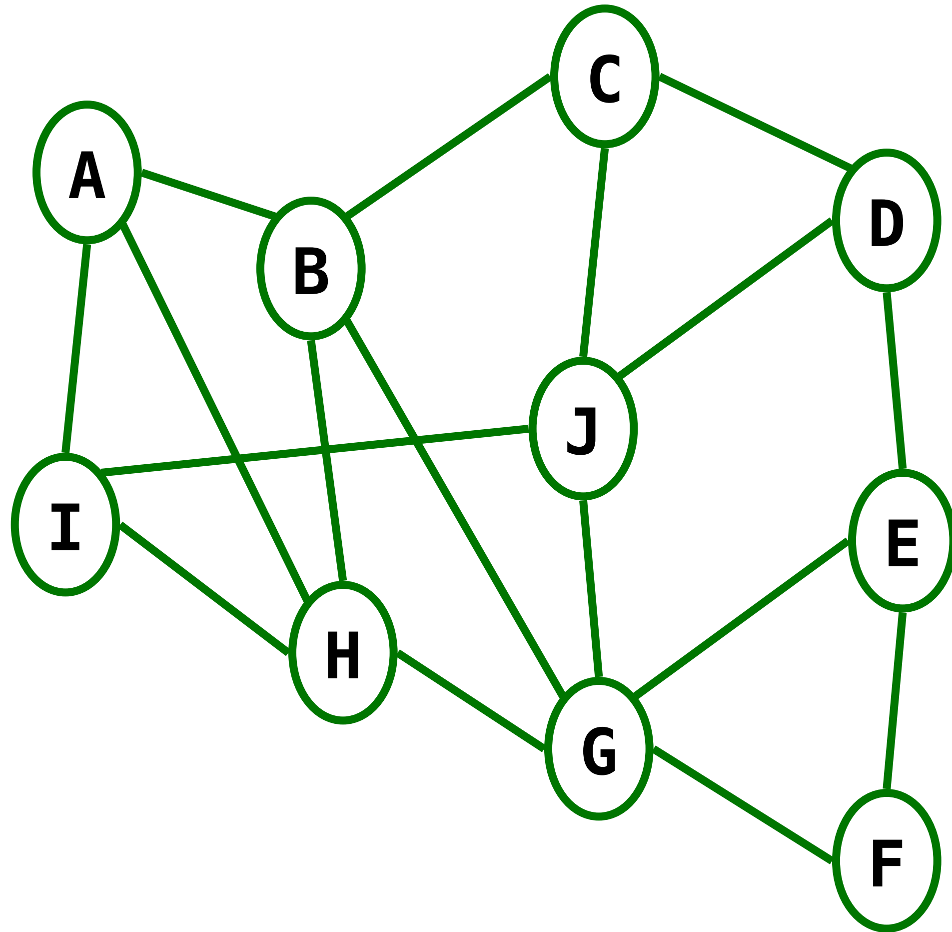
- Put the graph nodes into a sequence
- Iterate through this sequence, assigning each node a color that does not introduce an immediate conflict in the graph.
- Backtrack when we reach a node that cannot be assigned a color

Can These Be 3-Colored?

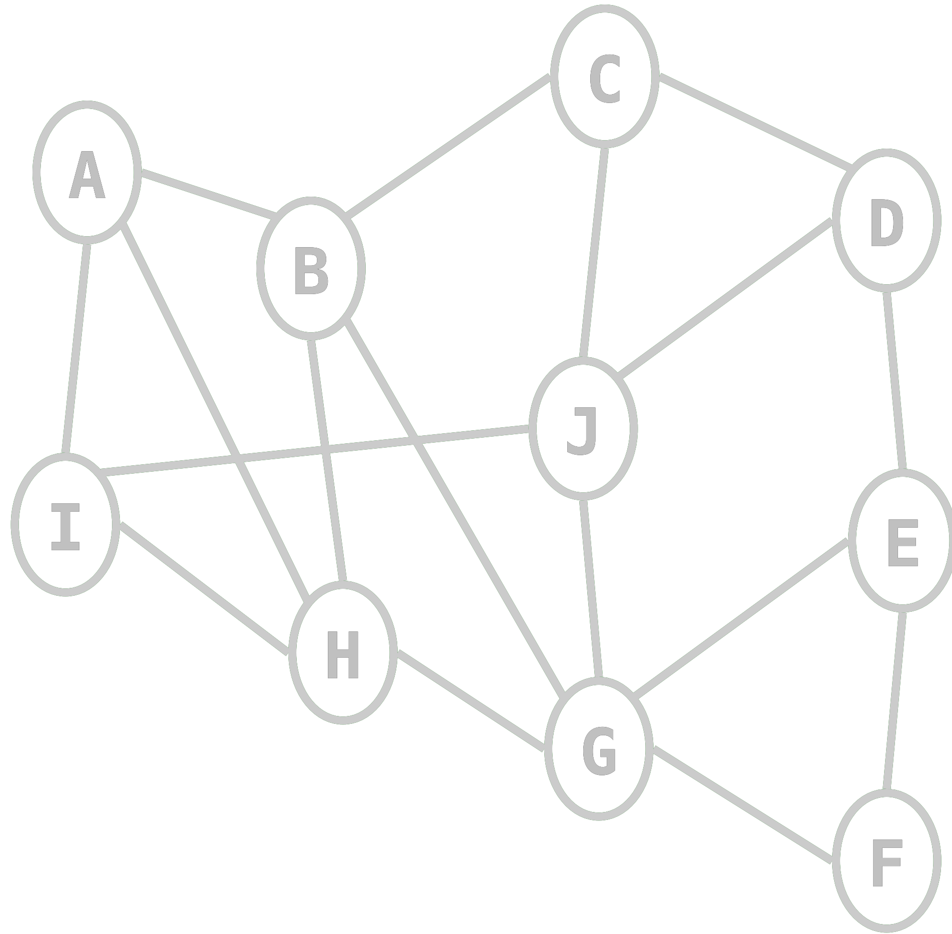




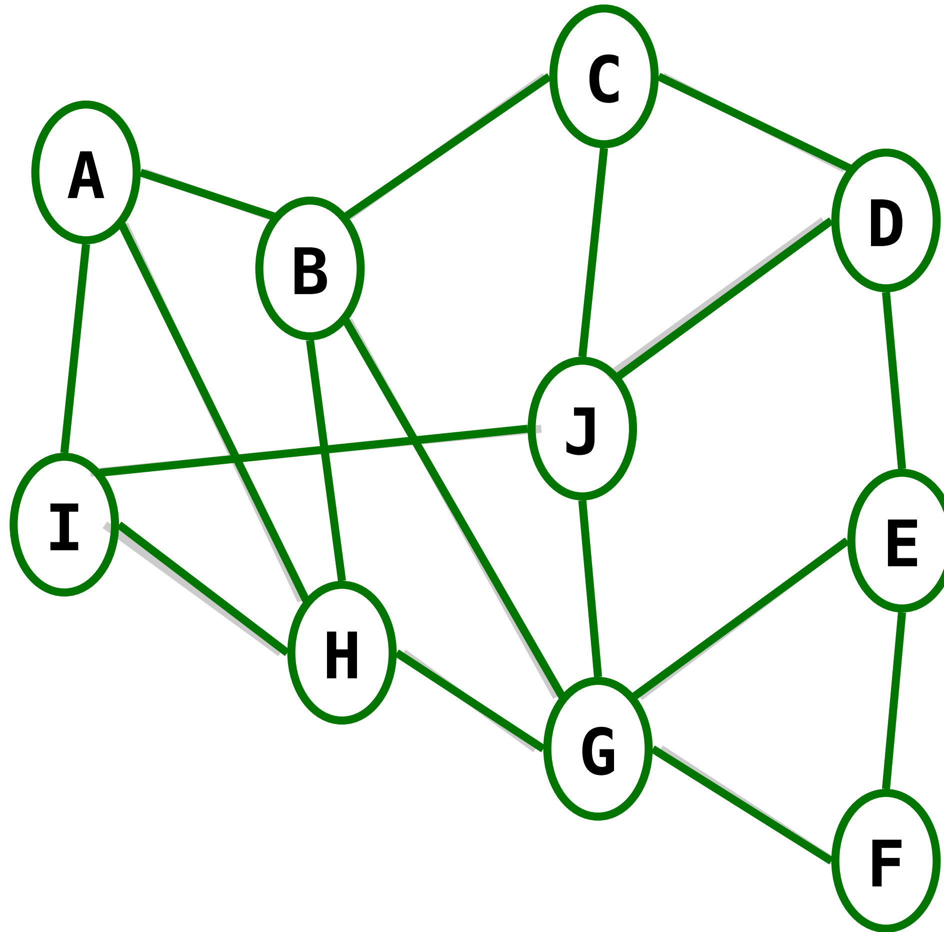
A Clever Observation



Simplify Phase



Select Phase



Chaitin's Heuristic

Given the graph G:

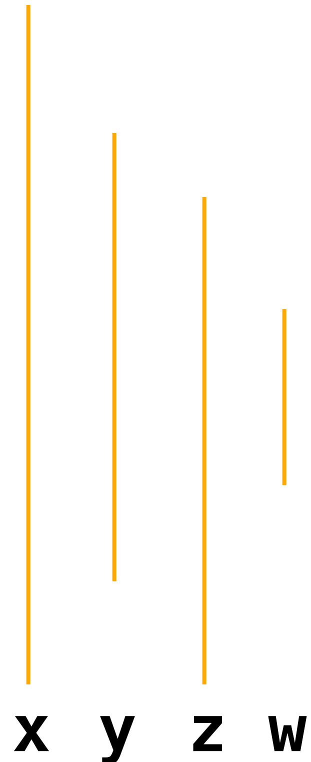
- If G contains a node n of degree < k:
  - Remove it to get G', recursively color G'
  - Find a non-conflicting color for n (always possible).
- If G contains only high-degree nodes:
  - Pick some node n (often of high degree) to be spilled
  - Remove n from the graph and recurse

When the graph is empty:

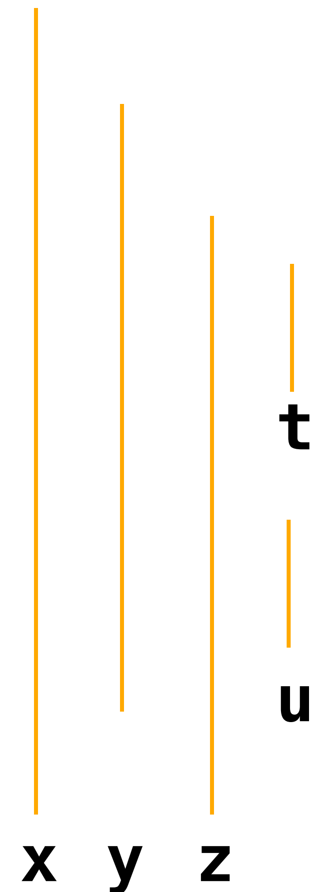
- Add spilling code and re-run register allocation.
- Or, abort allocation as soon as first temporary is spilled; rewrite code and re-run register allocation
- Or, reserve a few registers just for spilling.

Iterative Spilling

```
x = 3
y = 4
z = 7
w = x + y
z = z + 1
x = x + w
z = z + y
return x+z
```



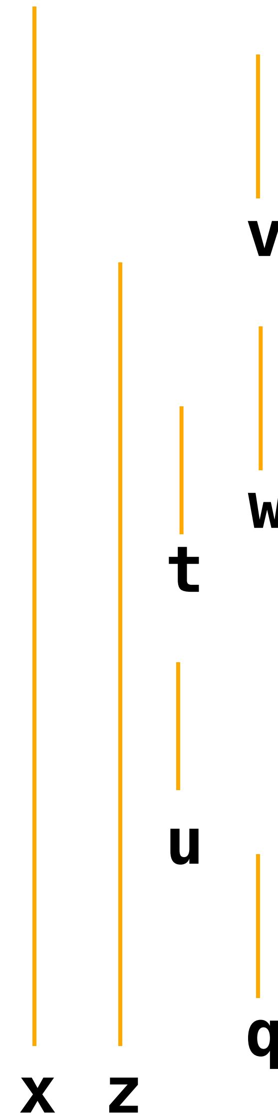

```
x = 3
y = 4
z = 7
t = x + y
*M_w = t
z = z + 1
u = *M_w
x = x + u
z = z + y
return x+z
```



Iterative Spilling

```
x = 3
y = 4
z = 7
t = x + y
*Mw = t
z = z + 1
u = *Mw
x = x + u
z = z + y
return x+z
```

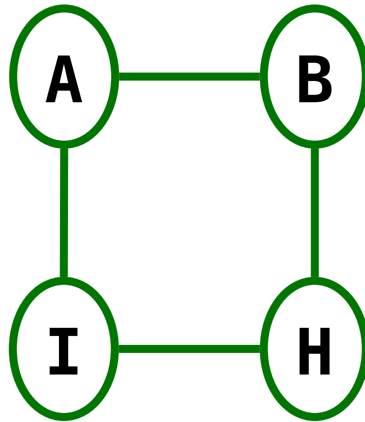
```
x = 3
v = 4
*My = v
z = 7
w = *My
t = x + w
*Mw = t
z = z + 1
u = *Mw
x = x + u
q = *My
z = z + q
return x+z
```



Will this process even terminate?

What Would Chaitin Do?

Assume we need a 2-coloring.



How could we improve the algorithm?

Briggs' Optimistic Coloring

- Simplify routine builds a stack of nodes from graph G.
  - If G contains a node n of degree < k:
    - Put n onto the stack, remove it from the graph, and recurse
  - If G contains only high-degree nodes:
    - Pick high-degree node n (*spill candidate*)
    - Put n onto the stack, remove it from the graph, and recurse
- Select routine colors the nodes on the stack
  - Mark nodes to be spilled only if their neighbors already use all available colors.
- If temporaries were spilled, repeat as before.

Which temps should spill?

To minimize size penalty for spill code?

To minimize the time penalty for spill code?

To minimize the number of spilled temporaries?

NB: Never spill temporaries introduced by spill code!

Improvements

If our code contains the copy instruction

```
movl t1, t2
```

It'd be nice if t1 and t2 were assigned the same register.

We already went to some effort not to forbid this when constructing the interference graph.

But how could we actively encourage it?

Biased Coloring

When choosing a color for a node, look to see if there's one that would eliminate a move operation.

Coalescing

Requiring a and b must be allocated the same register

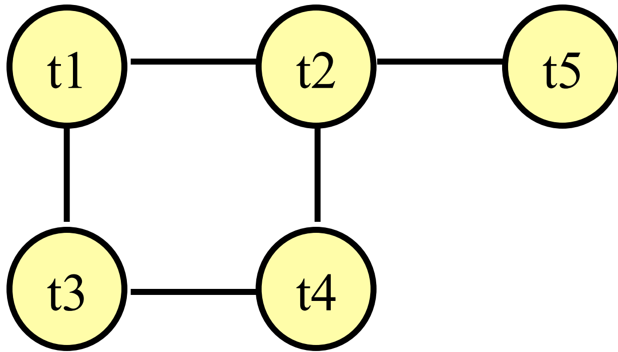
==

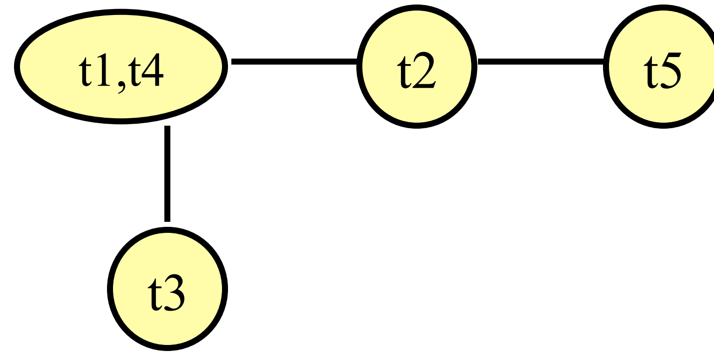
merging nodes for a and b in the graph.

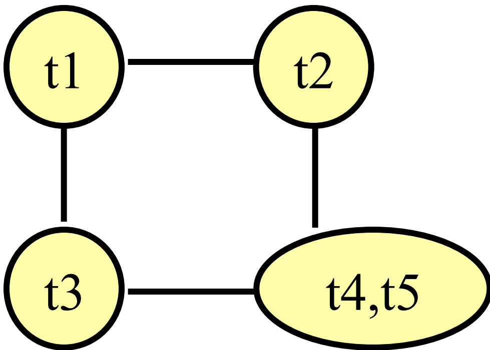
Single "coalesced" node replaces two nodes

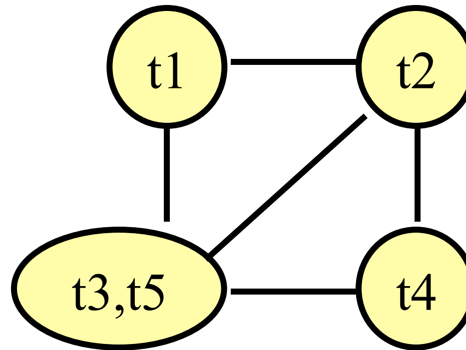
Interferes with any node that either of the two original nodes interfered with.

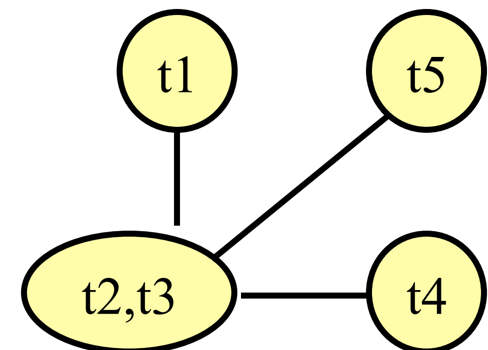
Coalescing Example











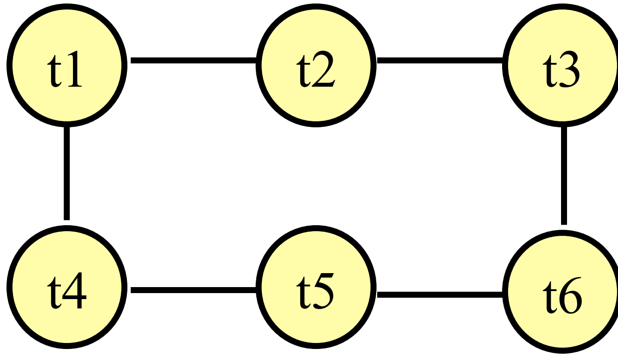
Aggressive Coalescing

Chaitin:

- Before coloring, coalesce the source and destination of every copy (as long as they don't interfere).

But is this a good idea?

movl %t2, %t5



Coalescing Strategies

Spill code is usually more expensive than a copy, because it involves memory access

Hence many allocators are very conservative:

Only coalesce if we can guarantee that it doesn't increase the number of colors required.

Conservative Heuristics

Briggs:

Two nodes can be coalesced if the resulting node has < k neighbors with ≥ k edges.

George/Appel

Two nodes **`a`** and **`b`** can be coalesced if

for every neighbor **`t`** of **`a`**,

either **`t`** is a neighbor of **`b`**, or else **`t`** has degree < k.

Opt. Graph Coloring with Traditional Coalescing

1. Build the interference graph.
2. Apply conservative coalescing to move-related nodes.
3. Repeatedly remove nodes from the graph (preferentially removing nodes of degree $< k$).
4. Color the simplified nodes in reverse order.
5. Insert spill code and repeat, if necessary.
6. Replace references to temporaries with references to registers.

Iterated Coalescing

George and Appel:

It's safe to do coalescing even after some of the graph nodes have been removed.

So?

Iterated Coalescing

When simplifying the graph (in order of preference)

- Remove a non move-related node of degree <k
- Or, conservatively coalesce pair of move-related nodes
- Or, remove a move-related node of degree <k
  - (Give up on trying to coalesce it)
- Pull out a high-degree node
  - (As a spill candidate)

Note: Coalescing may create non move-related nodes with low degree, etc.

Live Range Splitting

So far we have required that a variable has a permanent "home" --- either a register or memory.

Live range splitting breaks the lifetime of a variable into separate pieces, each of which can be separately allocated.

Keep the temporary in a register when it's used a lot

Keep it in memory between times.

Live Range Splitting

Easy case: a variable has disconnected live ranges.

- Chaitin's "right number of names"

```
int i;
for (i = 1; i < 10; ++i)
  f(i);
for (i = 20; i < 30; ++i)
  g(i);
```

Otherwise it's much harder, and still appears to be an open problem how to do well (and quickly).

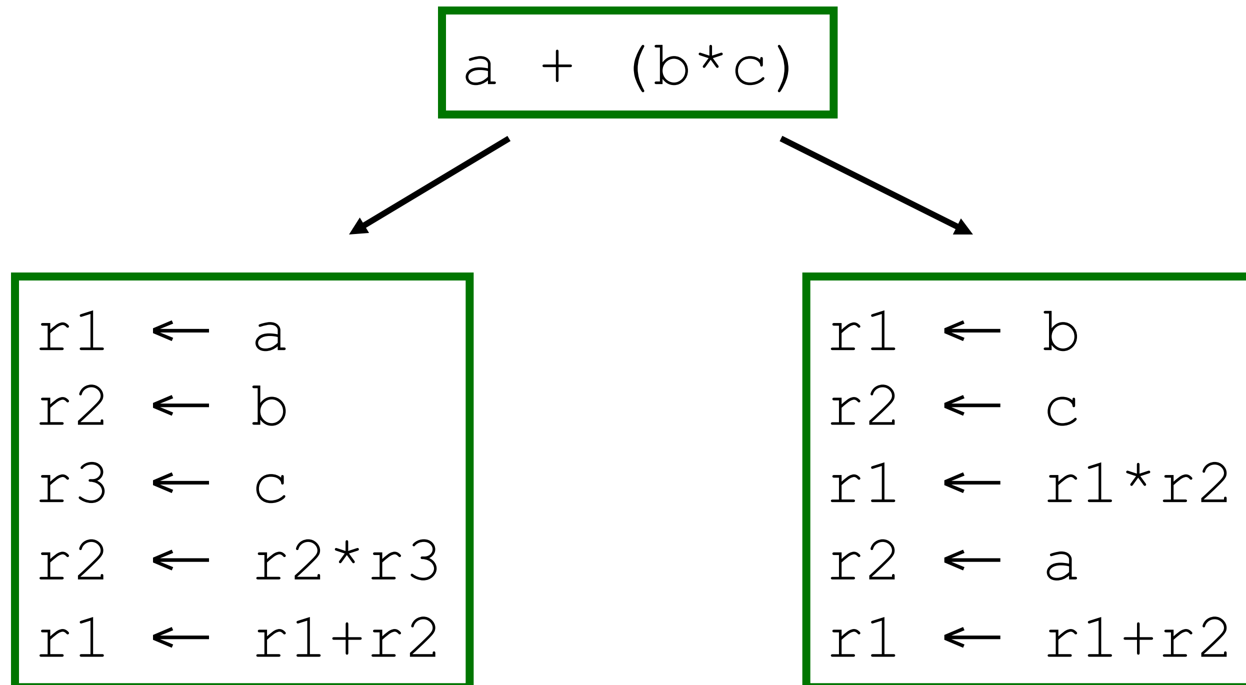
Sethi-Ullman Numbering

- Assume that all variables in a complex expression are stored in memory.

```
a + (b*c)
```

How many registers are required to evaluate it (assuming no side effects)?

Sethi-Ullman Example



Stepping Back

Where are we actually applying register allocation?

- A single expression
- A single basic block: *local* allocation.
- An entire procedure: *global* allocation.
- Many procedures: *interprocedural* allocation.