Code Generation - 2

- **Global register allocation through graph coloring**
  - **Live ranges**
  - **Interference graph**
  - **Coloring algorithm**

Global Register Allocation

- **Picks values to store in registers across groups of basic blocks in the control flow graph**
  - **Choose using estimates of profitability (saved loads and stores) and availability of registers**
- **Calculate *live ranges* (regions -- set of *traces*-- in which a value will stay in a register)**
  - ***Live range* may not be entire program**
  - **A value can be in a register and then in memory or vice versa**

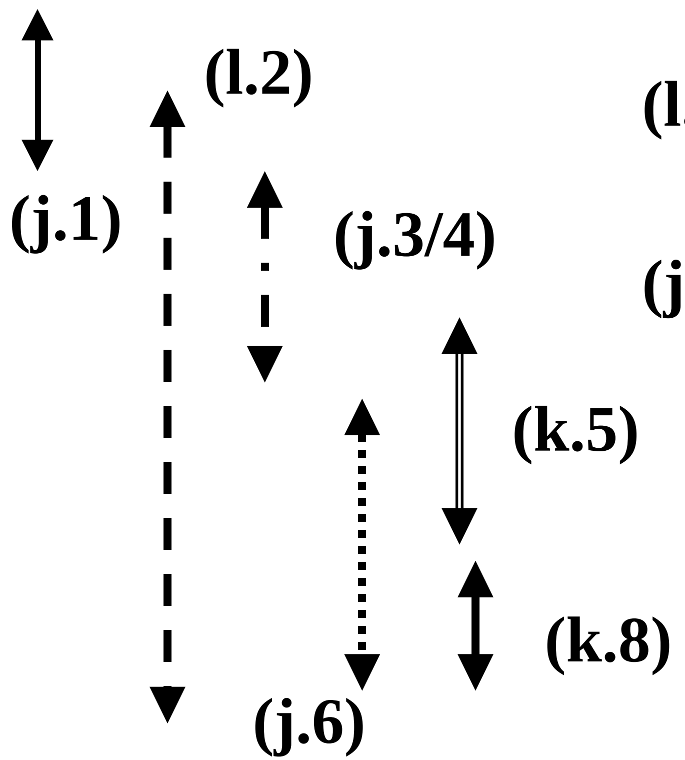
Global Register Allocation

- **Map interference between live ranges**
  - **Each live range is a node in an undirected graph**
  - **Two live ranges overlapping is shown by placing an edge between their corresponding nodes**
  - **For *k* global registers, add a *k-clique* to the graph**
    - ***k-clique: k* nodes all connected to each other**
- **Use graph coloring to map registers to ranges where each color represents a register**
  - **Try to obtain a *k*-coloring of this graph**
    - ***Legal coloring: assign colors to each node in the graph such that no adjacent nodes are the same color.***

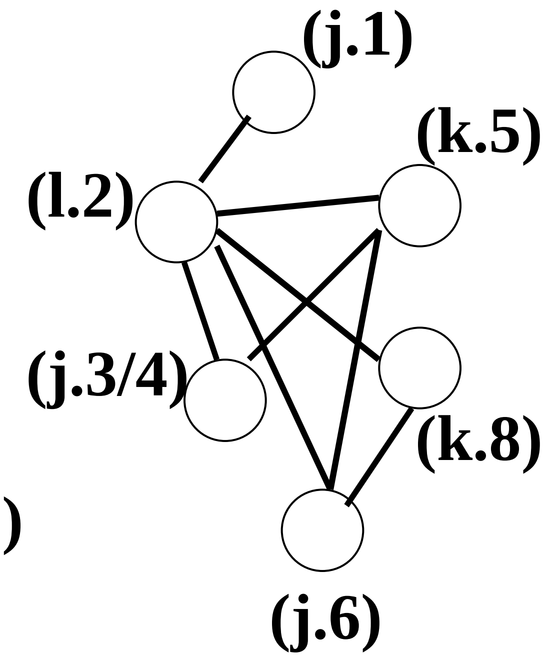
Example

```
1. j :=
2. l :=  ...j...
3. if( ) then { j :=  }
4.       else {j := }
5. k := j …
6. j :=
7.  := j…k
8. k:=
9.  := ...j…l..k...
```

Code

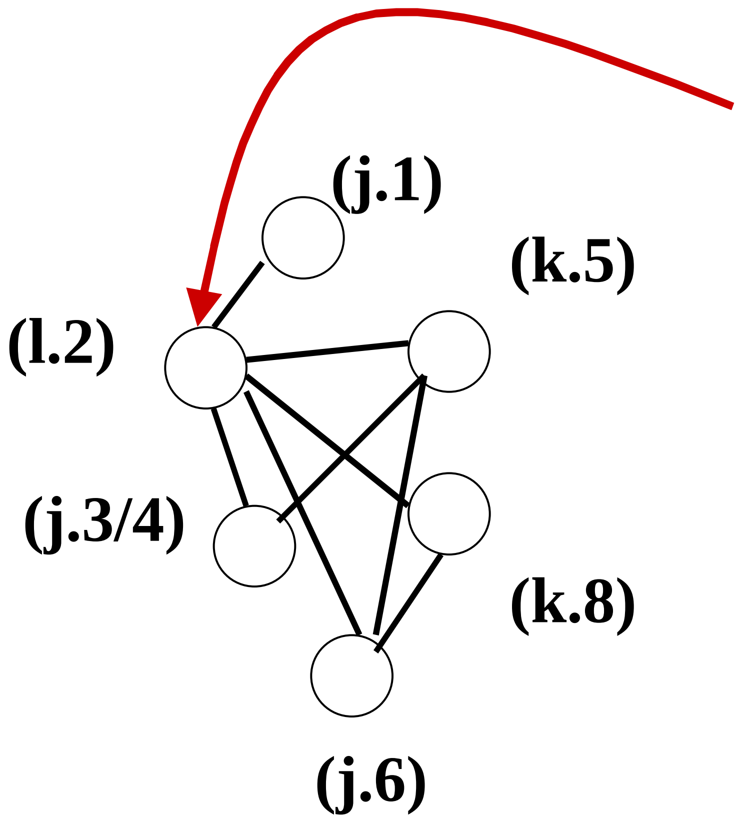


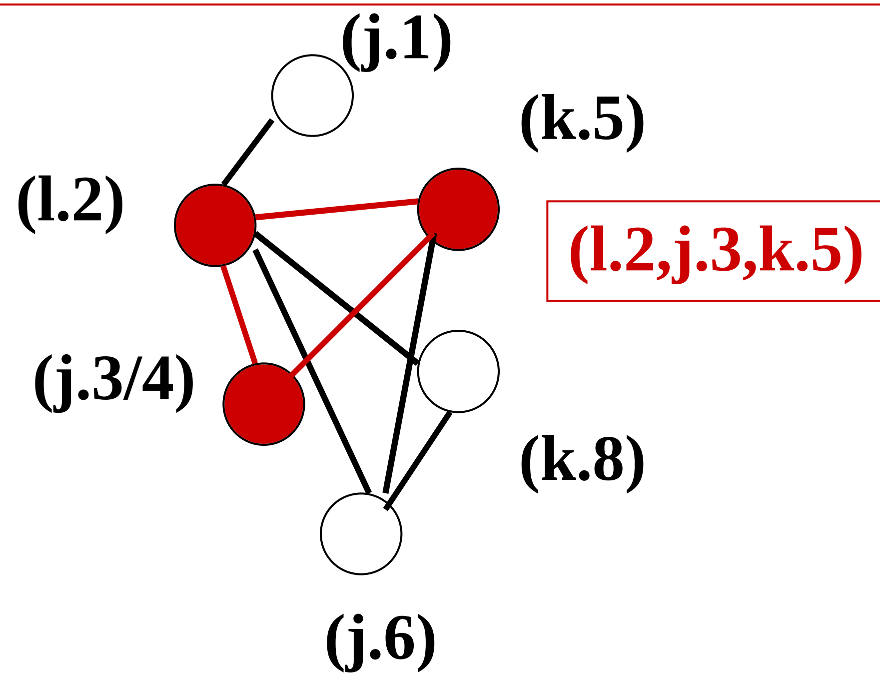
Live Ranges



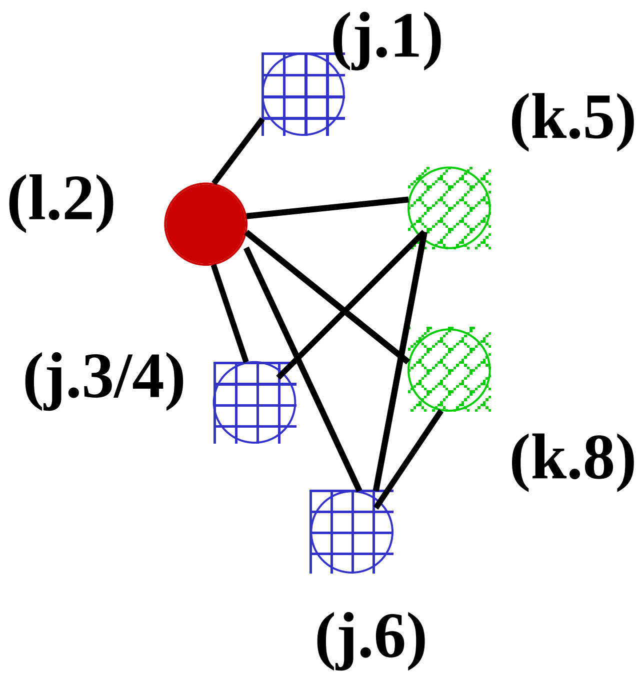
Interference Graph

With one node having 5 neighbors, we may need lots of colors, if neighbors are interconnected.
Find 3-cliques: (l.2,j.3/4,k.5) (l.2,k.5,j.6) (l.2,j.6,k.8)
so need at least 3 registers to color this graph!



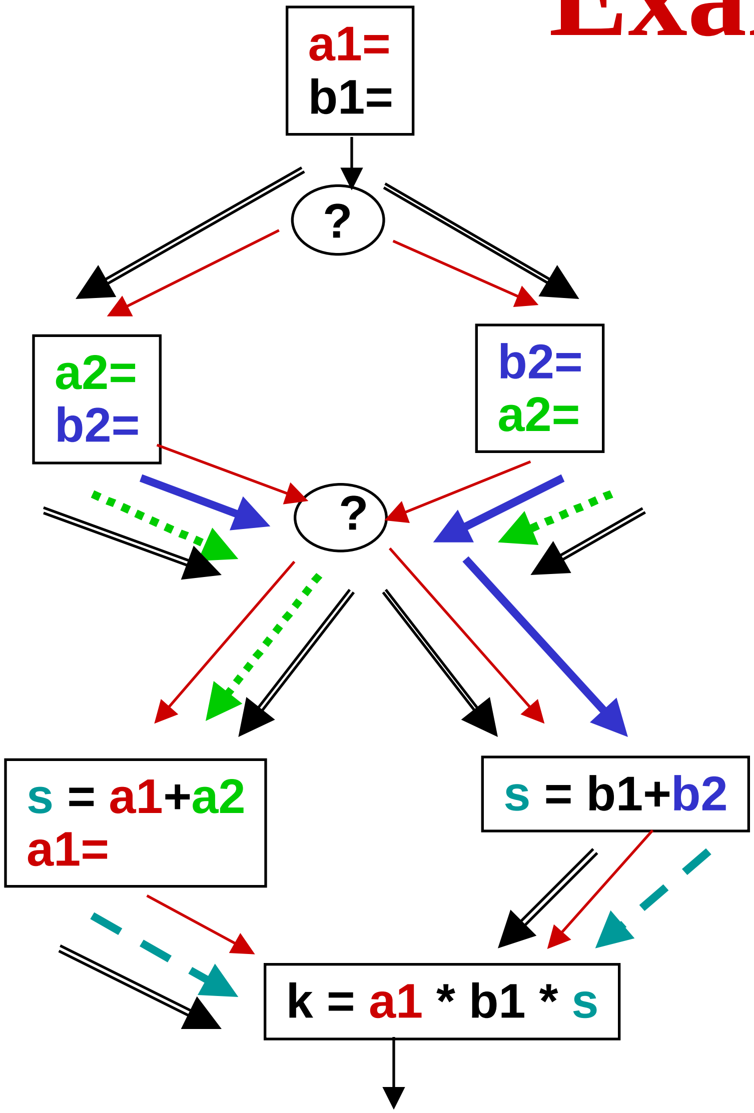


Example



A possible 3 coloring.

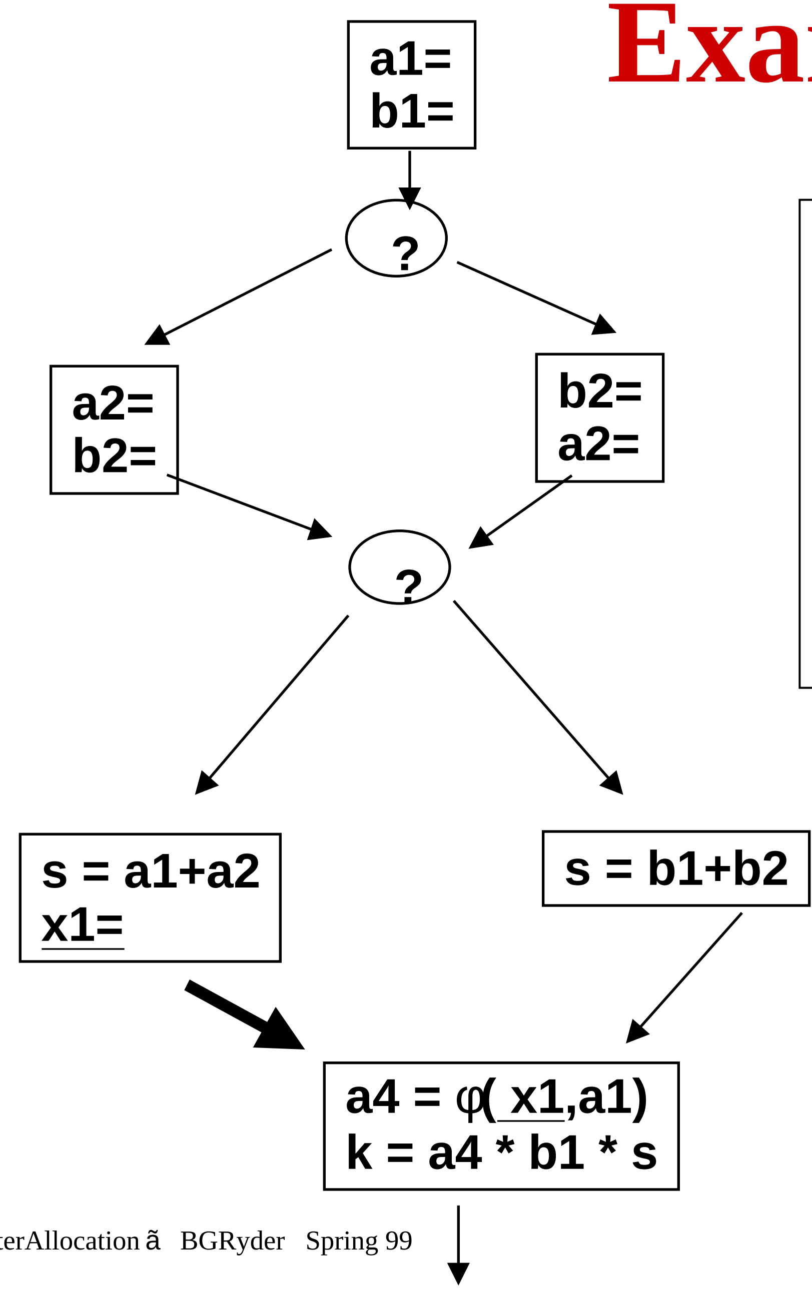
Example



live range for a1
live range for b2
live range for a2
live range for b1
live range for s

Interference shown by overlap of live ranges in shared nodes and/or edges in graph.

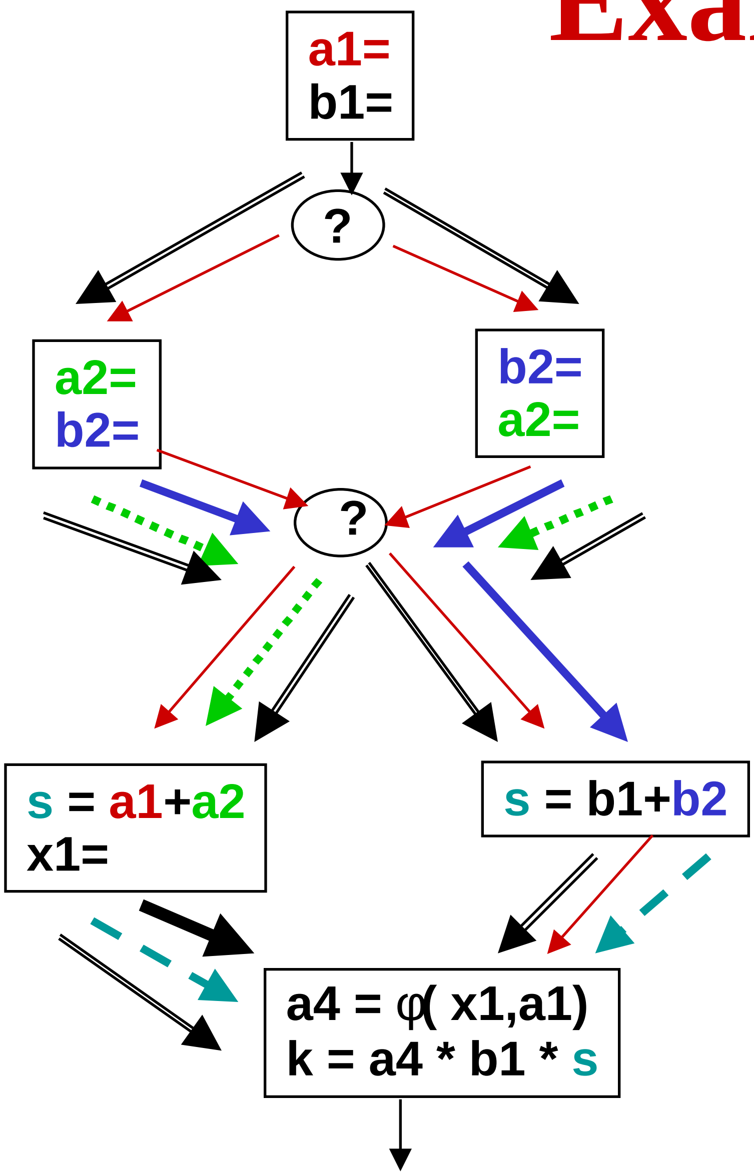
Example

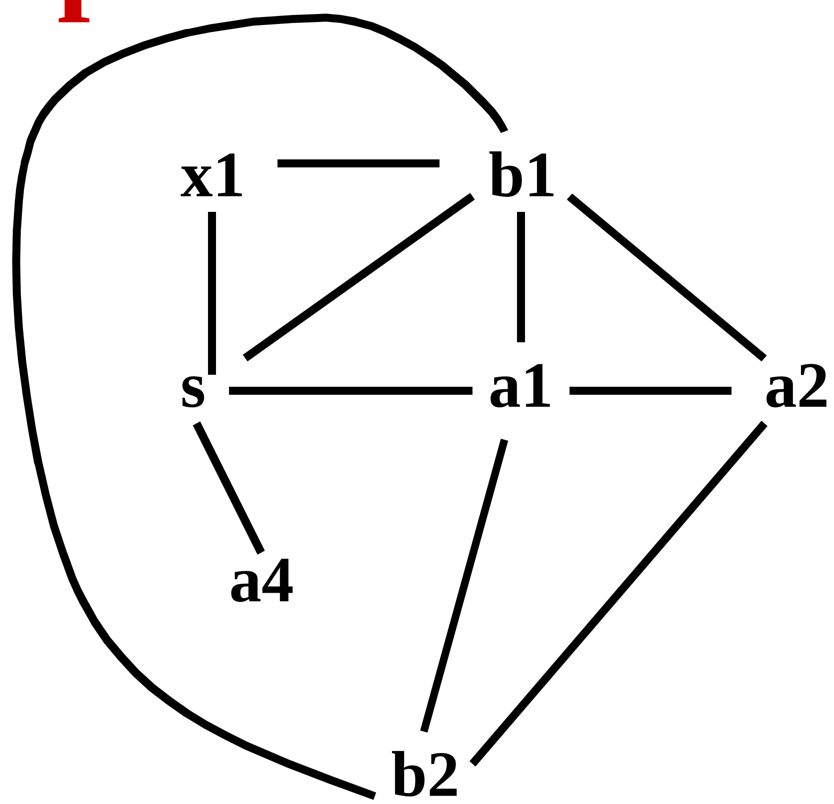


Variable renaming may let us eliminate some interferences. This graph has no interference between x1 and a1.

SSA-form: each variable use is reached by only one definition.

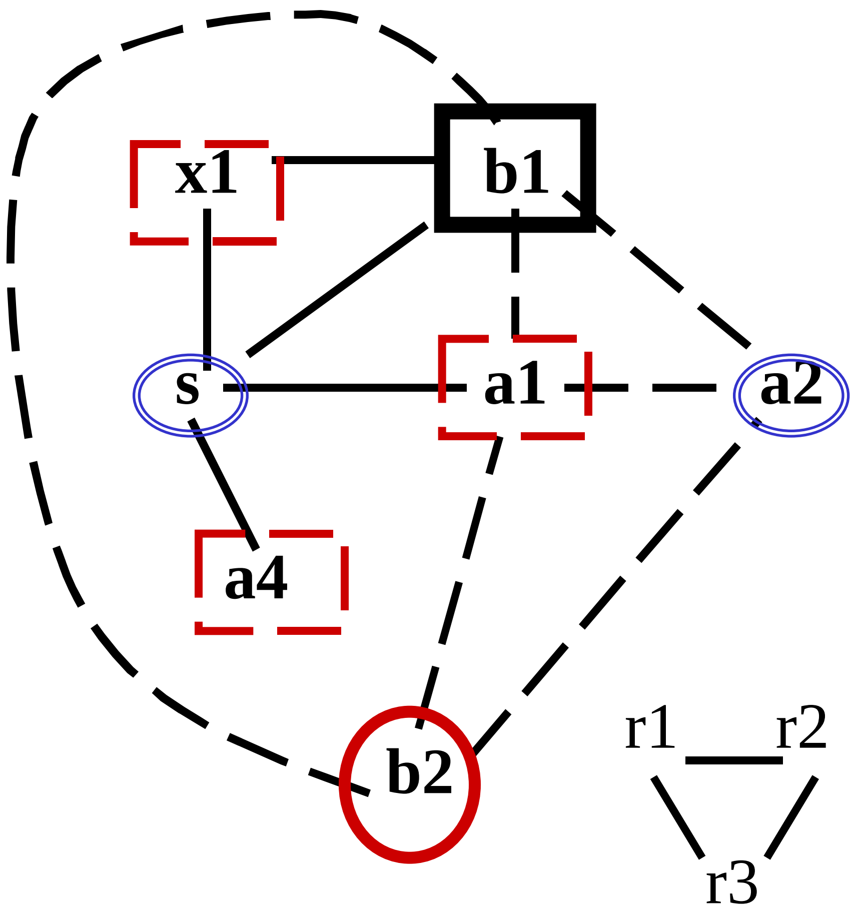
Example





Interference graph

Example



Interference graph

- **Interference graph contains a 4-clique so can't find a 3 coloring**
- **Means have to spill something to find a 3 coloring**
- **But a 4 coloring is possible as shown**

Optimistic Register Allocation

Chaitin et.al. 1982

- ***Build*** **interference graph**
- ***Simplify*** **by removal of easy-to-color nodes to a stack (degree < *k*)**
- ***Spill*** **some value when reach state where all nodes left have degree ≥ *k (potential spill)***
- ***Select*** **color for each node in stack order**
- ***Restart*** **after removal of spilled node and its adjacent edges (*actual spill*)**

Optimistic Register Allocation

- **Usually 1-2 iterations work in practice**
- **Heuristics**
  - **Which node to spill? try to estimate which node is inhibiting the coloring the most**
    - **E.g., minimize estimated spill cost per neighbor of current node where spill cost = #def points +#use points weighted by execution frequency**
  - **Which node to color next during algorithm?**
    - **Try to have coloring *fail* as early as possible. Why?**
    - ***Color urgency* - #uncolored neighbors/#possible colors left**

Example

coloring with k=3.

<u>stack</u> simplify nodes with 1,2 neighbors

x1

a4

s   ***assume can color s; now delete edges so that b1 (deg 4), a1 (deg 3)***

a1   ***assume can color a1; now delete edges so that a2,b2 (deg 2), b1(deg3)***

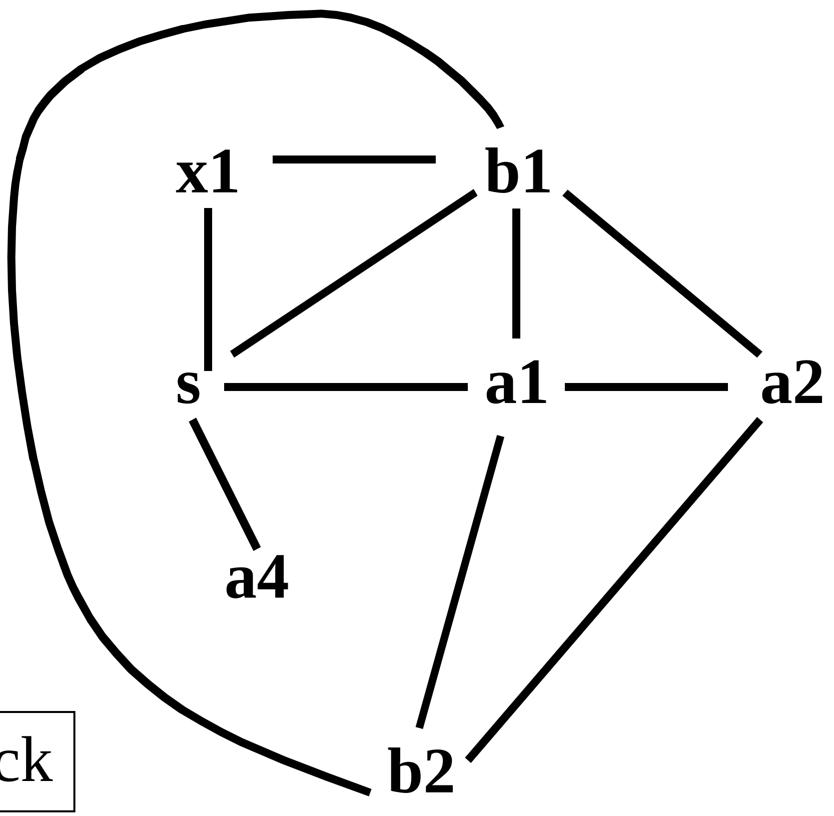
b1   ***assume can color b1; now delete edges so that a2,b2 (deg 1)***

a2

b2

DONE

3 potential spills in stack



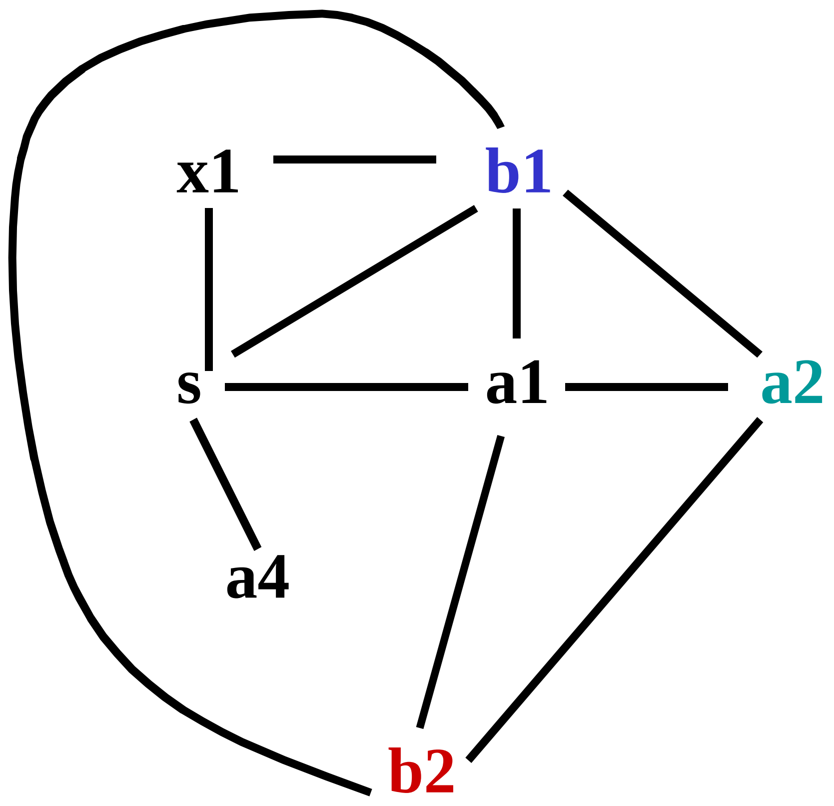
Example

Colors: 1, 2, 3

<u>Assignments</u>

b2	**1**
a2	**2**
b1	**3**
a1	**can’t color so must be actual spill**

Redraw interference graph and start over.



Example

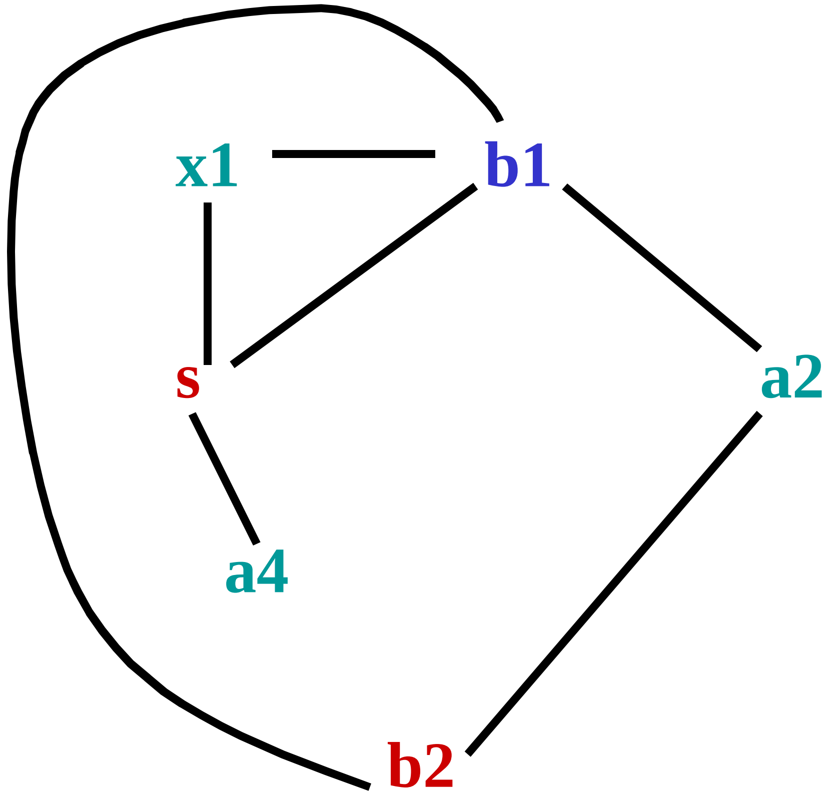
Restart

Colors: 1, 2, 3

Assignments

b2	**1**
a2	**2**
b1	**3**
s	**1**
a4	**2**
x1	**2**

SUCCESS



Improved Register Allocation

Briggs, et. al. (1994), George+Appel (1996)

- ***Coalescing* - try to combine live ranges when can avoid register copies Rs ← Rt; check if can do calculation in Rs.**
  - **Improvement on earlier algorithm**
  - **When coalesce 2 nodes, get a new node with union of the edges of the 2 previous nodes**
  - **If overdo coalescing, then can create too many interferences, but how to tell?**

When to coalesce?

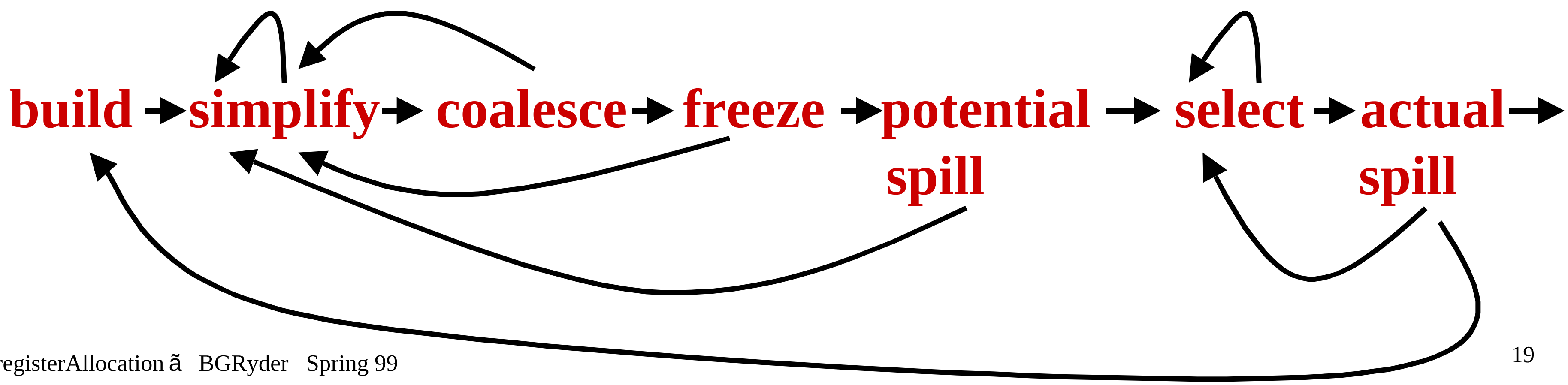
- **Need heuristic to guide coalescing decisions**
- **Briggs: Coalesce nodes *a* and *b* if resulting node will have fewer neighbors of degree ≥ *k***
- **George: Coalesce nodes *a* and *b* if for every neighbor *t* of *a*, *t* already interferes with *b* or *t* is of degree < *k*.**
- **Both strategies are *safe*, but *conservative***

Improved Algorithm

- ***Build*** **interference graph categorizing nodes as *move-related* or not**
- ***Simplify*** **by removal of non-move-related easy-to-color nodes to a stack (degree < k)**
- ***Coalesce*** **conservatively on simplified graph; restart simplify**
- ***Freeze*** **some move-related node of low degree and make it non-move-related; restart simplify**

Improved Algorithm, cont

- ***Spill*** **some value when reach state where all nodes left have degree ≥ *k (potential spill)***
- ***Select*** **color for each node in stack order**
- ***Restart*** **after removal of spilled node and its adjacent edges (*actual spill*)**



Improvements

- **If must spill, may need to undo coalescing**
  - **Simple: undo all coalesces and rebuild graph**
  - **Complex: undo all coalesces AFTER the first potential spill was identified**
- ***Precolored nodes* - values that must be in specific registers (e.g., for parameter passing)**
  - **Can color other nodes with those colors, as long as they don't interfere**
  - **Can't simplify or spill such nodes so want to keep their live ranges short!**

Improvements

- **Can sometimes coalesce spilled values if can prove their live ranges do not interfere (reuse storage)**
  - **Get interference graph for spilled nodes**
  - **Coalesce any non-interfering spilled nodes connected by a move**
  - **Use simplify and select to color the graph with each color corresponding to shared frame locations**

Improvements

- **Rematerialization**
  - **Look for never-killed calculations that are cheaply redone (in 1 instruction) instead of saved in a register**
    - **E.g., immediate loads of an integer constant, computing a constant offset from a frame pointer**

Register Saving and Allocation

- **Local variable or compiler temporary should be allocated to caller-save register to avoid saves**
- **Values live across several levels of procedure call, should be put in callee-save registers since then only 1 save is necessary**
  - **Can force this by making such nodes interfere with all precolored caller-save registers**

Observations

- **Empirical data**
  - **Optimistic coloring gains -2%-48% execution time improvement**
- **Interference graphs in practice aren't big**
- **Compilers should make constants recognizable by register allocator for rematerialization**
- **Order of coalescing seems significant**
  - **Better to do from inner loops to outer**

Observations

- **Limited backtracking in the coloring may be useful**
- **Can also split live ranges to decrease the number of nodes with ≥ *k* edges, however too much splitting makes it harder to select spills**
- **NP-noise explains anomalous behavior in heuristic solution of NP-complete problem**