How many variables?

x,y,z,xy,xz,res1

Do we need 7 distinct registers if we wish to avoid load and stores?

7 variables: x,y,z,xy,yz,xz,res1

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
xz = x*z
res1 = xy + yz
m[3] = res1 + xz
```

can do it with 5 only!

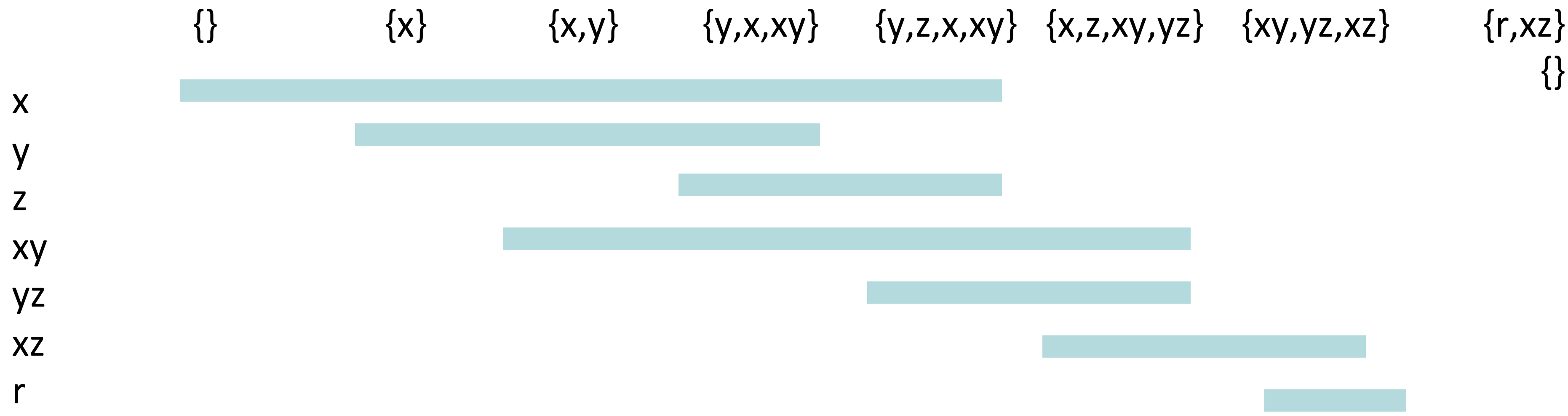
```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
y = x*z          // reuse y
x = xy + yz      // reuse x
m[3] = x + y
```

Idea of Register Allocation

program:

x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

live variable analysis result:

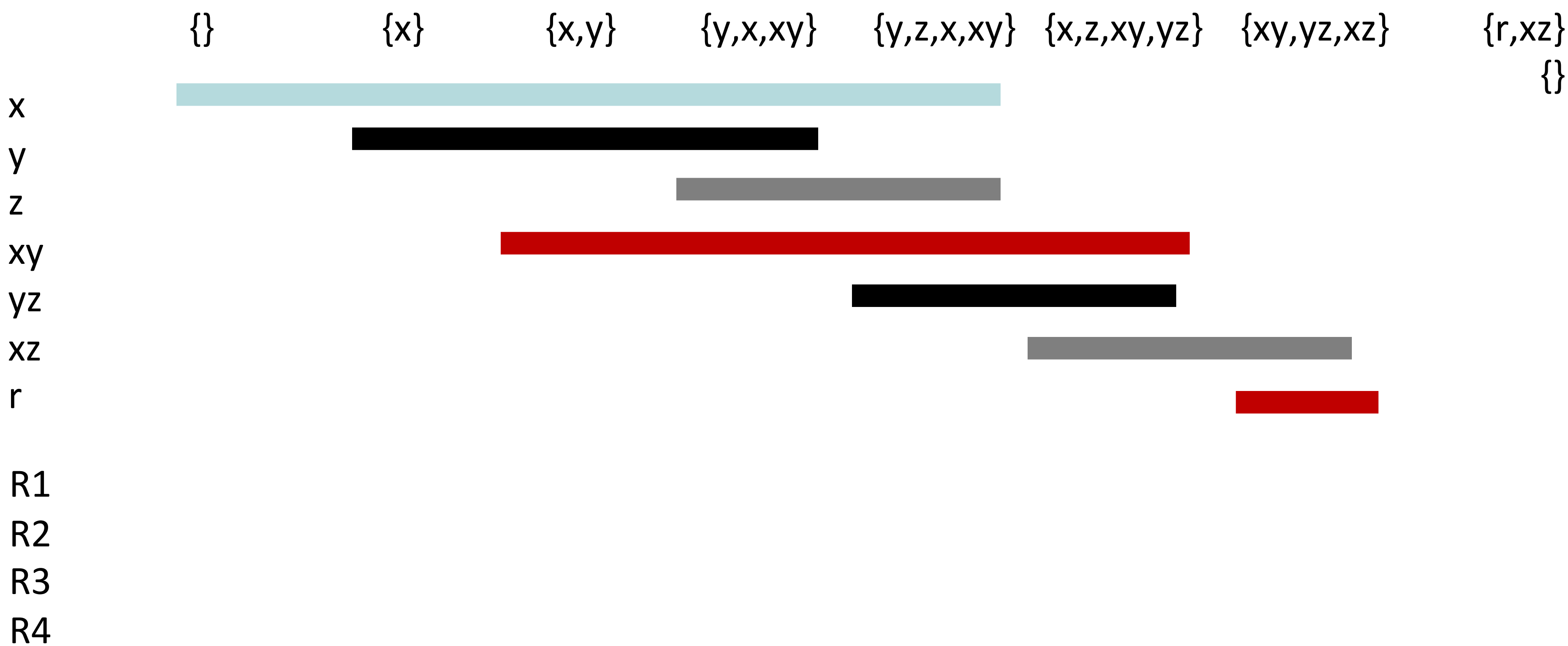


Color Variables
Avoid Overlap of Same Colors

program:

x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

live variable analysis result:



Each color denotes a register

4 registers are enough for this program

Color Variables
Avoid Overlap of Same Colors

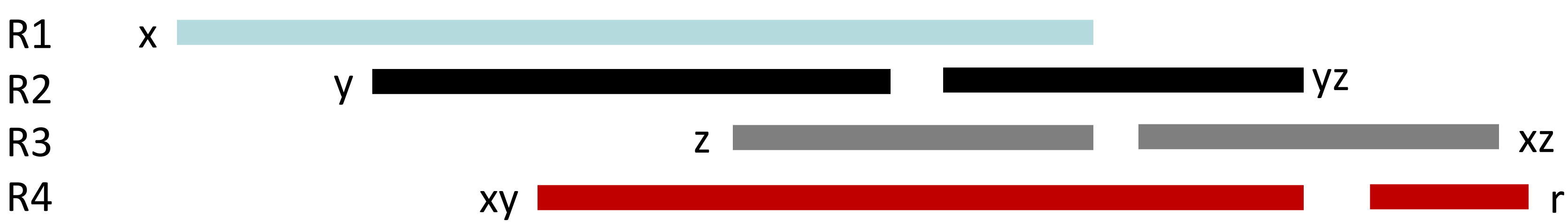
program:

x = m[0];   y = m[1];   xy = x*y;   z = m[2];   yz = y*z;   xz = x*z;   r = xy + yz;   m[3] = r + xz

live variable analysis result:

{}   {x}   {x,y}   {y,x,xy}   {y,z,x,xy}   {x,z,xy,yz}   {xy,yz,xz}   {r,xz}
{}

x
y
z
xy
yz
xz
r



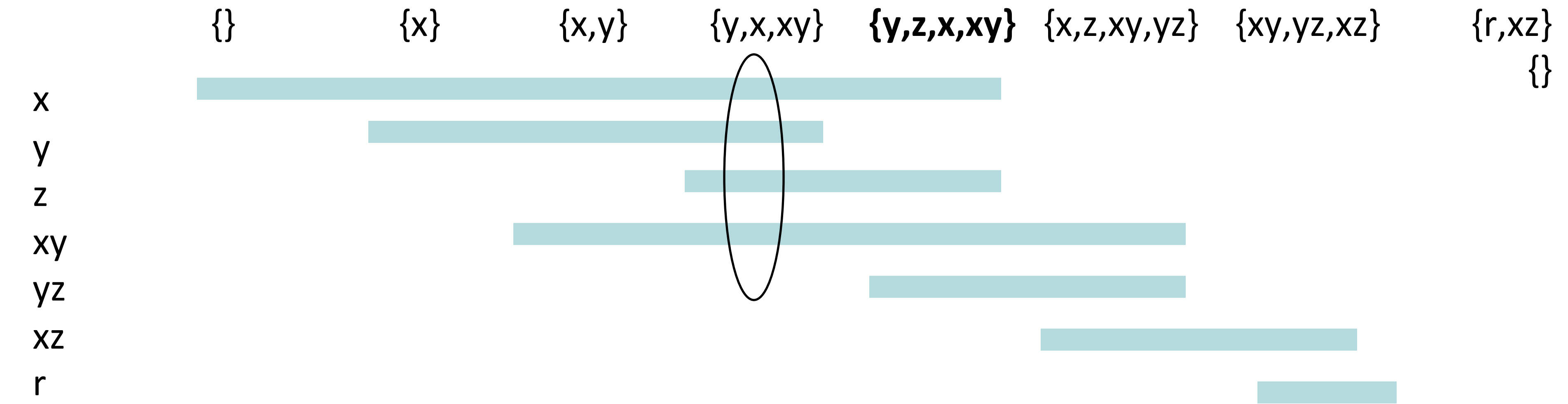
Each color denotes a register
4 registers are enough for this 7-variable program

How to assign colors to variables?

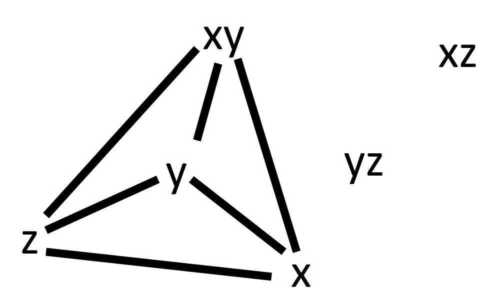
program:

x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

live variable analysis result:



For each pair of variables determine if their lifetime overlaps = there is a point at which they are both alive. Construct **interference graph**



Edges between members of each set

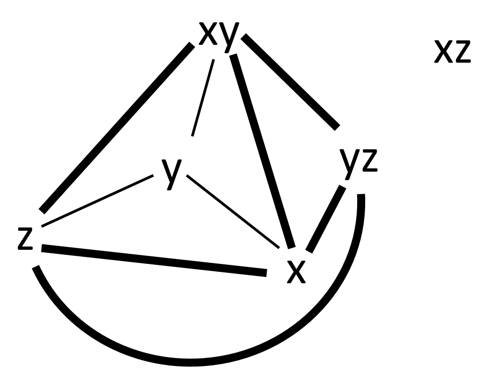
program:

x = m[0];   y = m[1];   xy = x*y;   z = m[2];   yz = y*z;   xz = x*z;   r = xy + yz;   m[3] = r + xz

live variable analysis result:

{}   {x}   {x,y}   {y,x,xy}   {y,z,x,xy}   {x,z,xy,yz}   {xy,yz,xz}   {r,xz}   {}

For each pair of variables determine if their lifetime overlaps = there is a point at which they are both alive.
Construct **interference graph**



Final interference graph

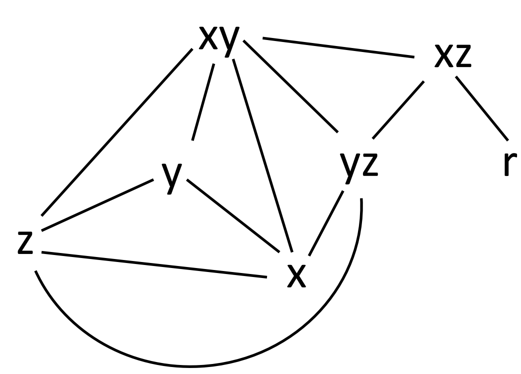
program:

x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

live variable analysis result:

{} {x} {x,y} {y,x,xy} {y,z,x,xy} {x,z,xy,yz} {xy,yz,xz} {r,xz} {}

For each pair of variables determine if their lifetime overlaps = there is a point at which they are both alive. Construct **interference graph**

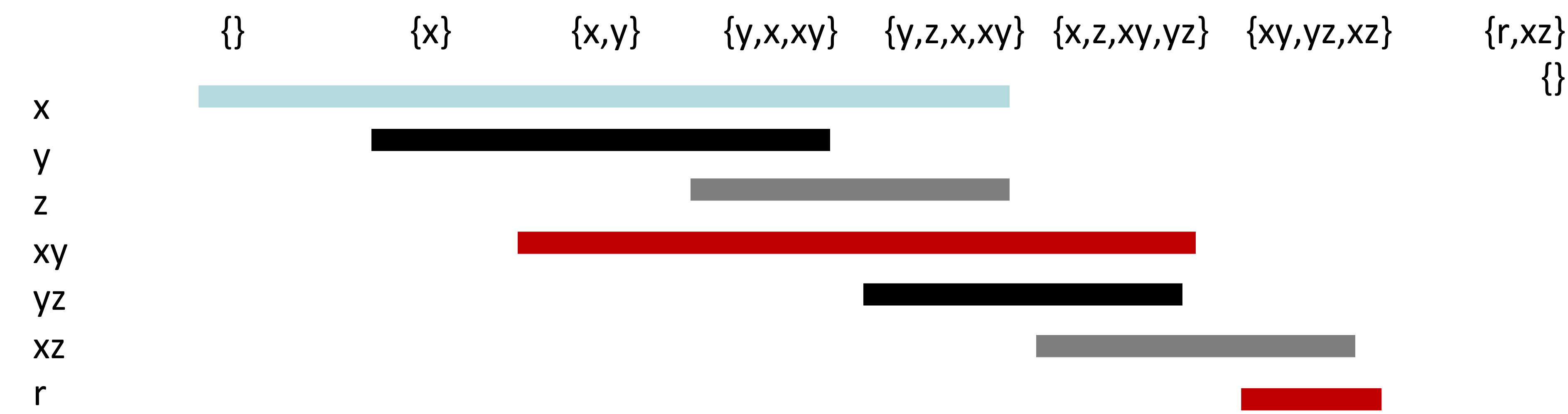


Coloring interference graph

program:

x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

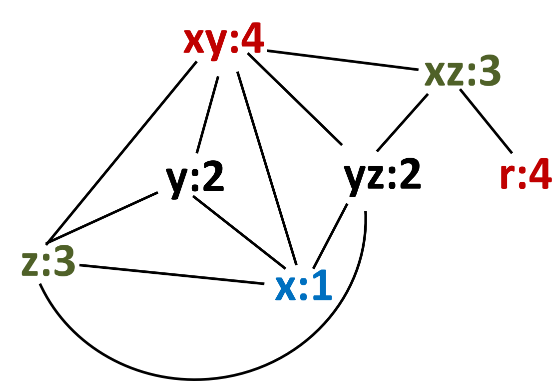
live variable analysis result:



Need to assign colors (register numbers) to nodes such that:

if there is an edge between nodes, then those nodes have different colors.

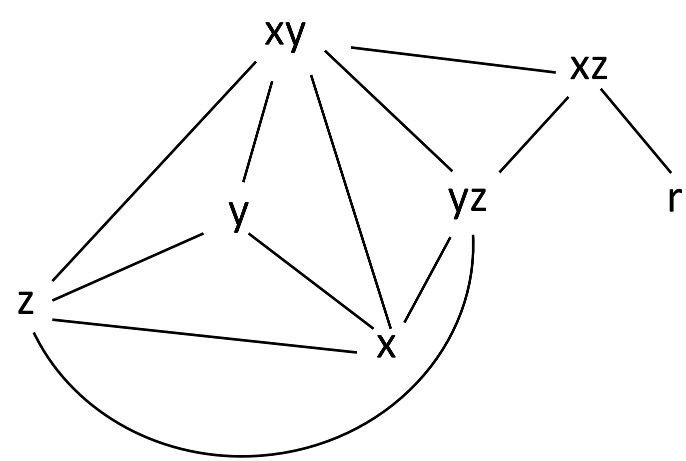
→ standard graph vertex coloring problem



Idea of Graph Coloring

- Register Interference Graph (RIG):
  - indicates whether there exists a point of time where both variables are live
  - look at the sets of live variables at all program points after running live-variable analysis
  - if two variables occur together, draw an edge
  - assign different registers to such variables
  - finding assignment of variables to K registers: corresponds to coloring graph using K colors

All we need to do is solve the graph coloring problem

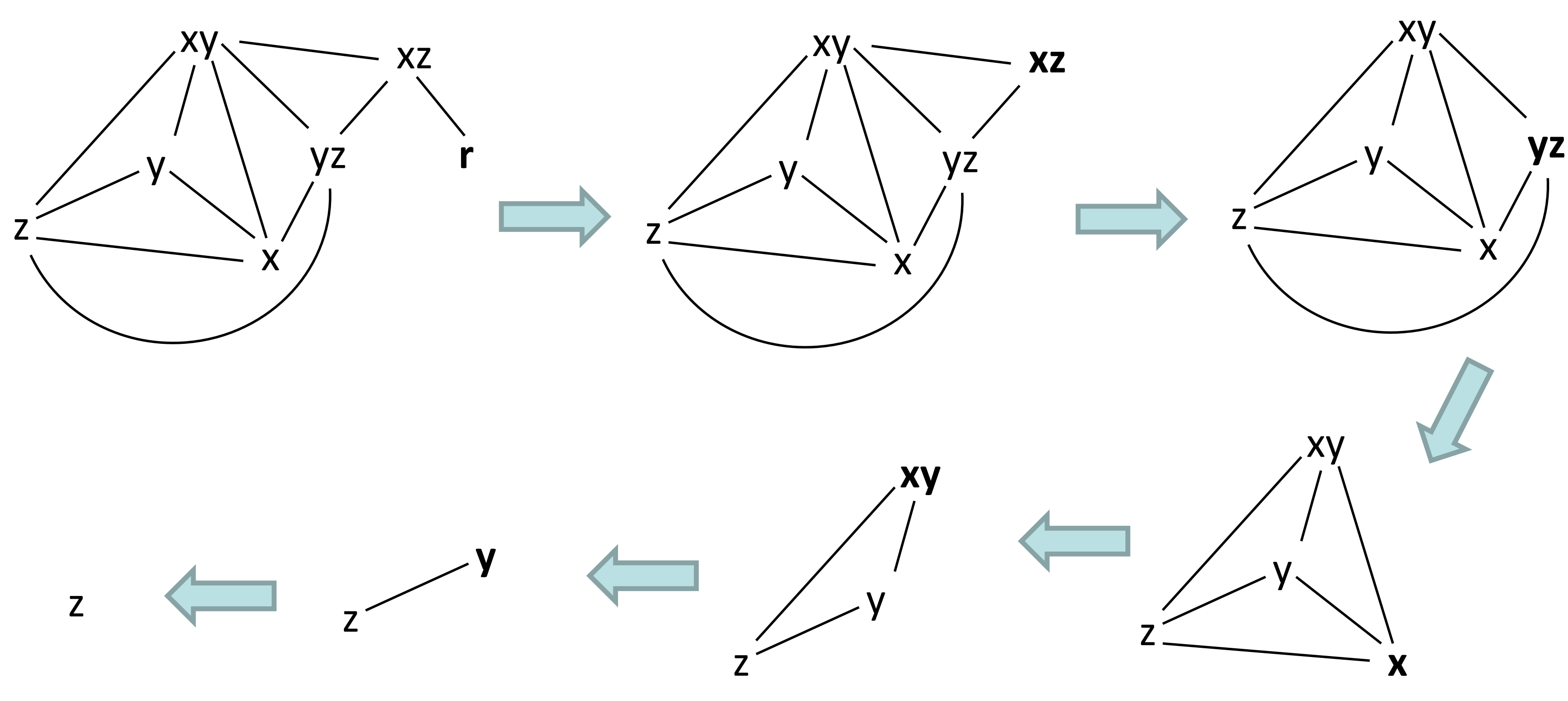


- NP hard
- In practice, we have heuristics that work for typical graphs
- If we cannot fit it all variables into registers,
  perform a **spill:**

  store variable into memory and load again before using

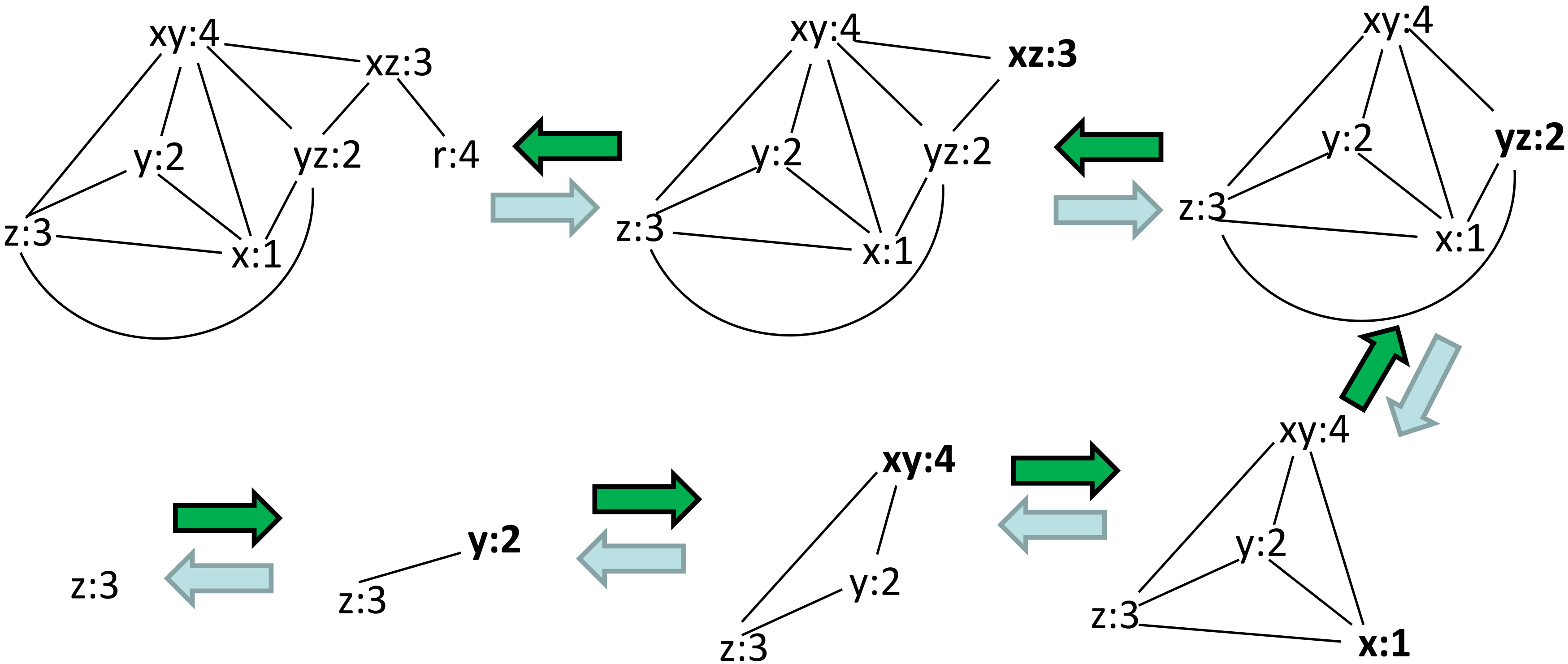
Heuristic for Coloring with K Colors

Simplify:
If there is a node with less than K neighbors, we will always be able to color it!
So we can remove such node from the graph (if it exists, otherwise remove other node)
This reduces graph size. It is useful, even though incomplete
(e.g. planar can be colored by at most 4 colors, yet can have nodes with many neighbors)



Heuristic for Coloring with K Colors

Select
Assign colors backwards, adding nodes that were removed
If the node was removed because it had <K neighbors, we will always find a color
if there are multiple possibilities, we can choose any color



Use Computed Registers

x = m[0]

y = m[1]

xy = x * y

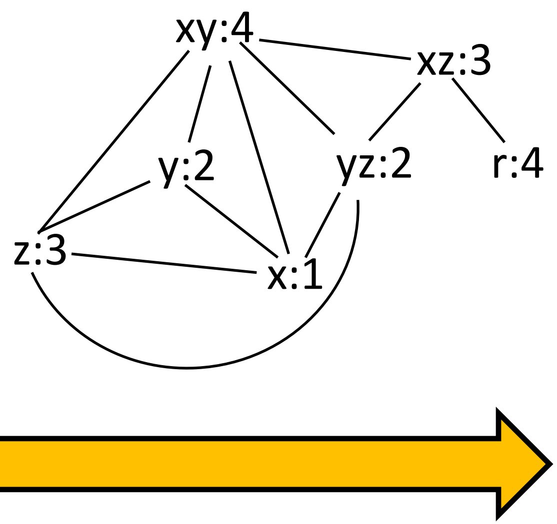
z = m[2]

yz = y*z

xz = x*z

r = xy + yz

m[3] = res1 + xz



R1 = m[0]

R2 = m[1]

R4 = R1*R2

R3 = m[2]

R2 = R2*R3

R3 = R1*R3

R4 = R4 + R2

m[3] = R4 + R3

Summary of Heuristic for Coloring

Simplify (forward, safe):
If there is a node with less than K neighbors, we will always be able to color it!
so we can remove it from the graph

Potential Spill (forward, speculative):
If every node has K or more neighbors, we still remove one of them
we mark it as node for **potential** spilling. Then remove it and continue

Select (backward):
Assign colors backwards, adding nodes that were removed

If we find a node that was spilled, we check if we are lucky, that we can color it.
if yes, continue

if not, insert instructions to save and load values from memory (**actual spill**).
Restart with new graph (a graph is now easier to color as we killed a variable)

Conservative Coalescing

Suppose variables tmp1 and tmp2 are both assigned to the same register R and the program has an instruction:

$$tmp2 = tmp1$$

which moves the value of tmp1 into tmp2. This instruction then becomes

$$R = R$$

which can be simply omitted!

How to force a register allocator to assign tmp1 and tmp2 to same register?

- merge the nodes for tmp1 and tmp2 in the interference graph!
- this is called **coalescing**

But: if we coalesce non-interfering nodes when there are assignments, then our graph may become more difficult to color, and we may in fact need more registers!

Conservative coalescing: coalesce only if merged node of tmp1 and tmp2 will have a small degree so that we are sure that we will be able to color it (e.g. resulting node has degree $< K$)

Run Register Allocation
use 3 registers, coalesce j=i

i = 0

s = s + i

i = i + b

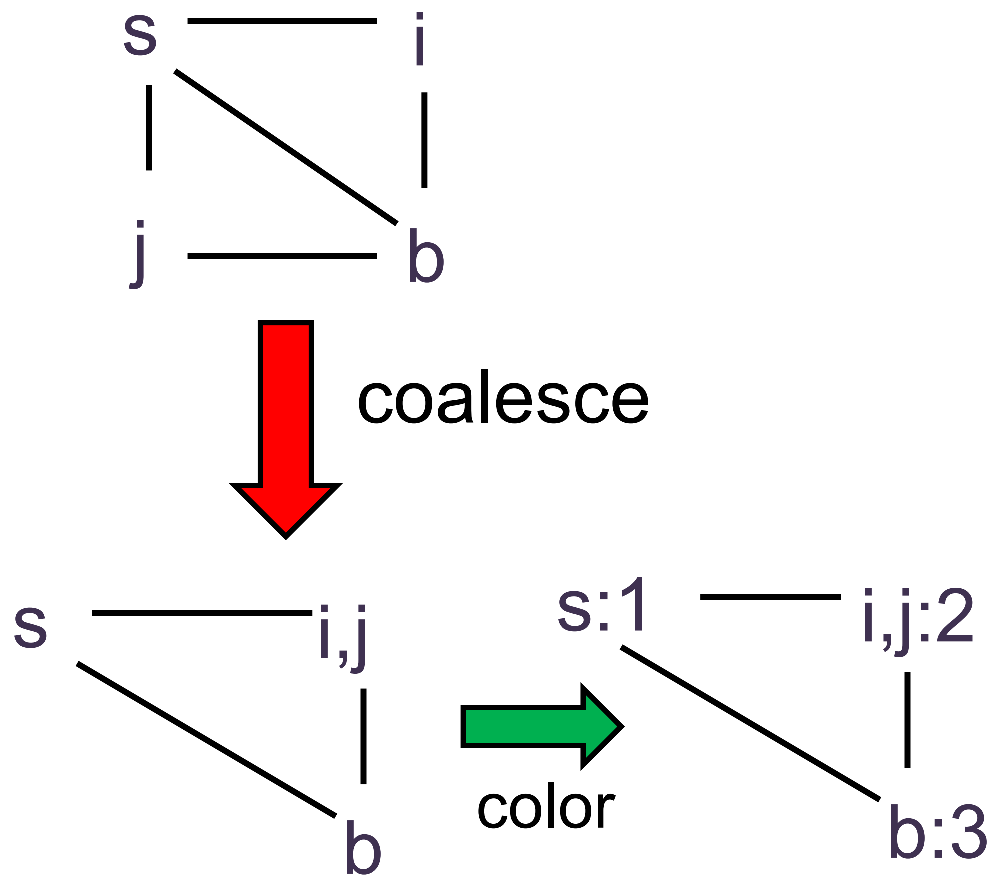
j = i

s = s + j + b

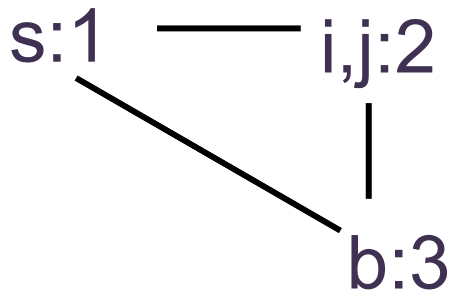
j = j + 1

Run Register Allocation
use 3 registers, coalesce j=i

{s,b}
i = 0
{s,i,b}
s = s + i
{s,i,b}
i = i + b
{s,i,b}
j = i
{s,j,b}
s = s + j + b
{j}
j = j + 1
{}



Run Register Allocation use 3 registers, coalesce j=i




```
i = 0
s = s + i
i = i + b
j = i  // puf!
s = s + j + b
j = j + 1
```

```
R2 = 0
R1 = R1 + R2
R2 = R2 + R3

R1 = R1 + R2 + R3
R2 = R2 + 1
```

Reordering Expressions

can help with register allocation

	{t}
x = t + 1	{t,x}
y = t + 2	**{t,x,y}**
a = x + t	**{y,a}**
b = y + 2	{a,b}
res = a + b	{res}

	{t}
x = t + 1	{t,x}
a = x + t	**{t,a}**
y = t + 2	**{a,y}**
b = y + 2	{a,b}
res = a + b	{res}

Going further: function calls

Problem: each function uses same CPU registers

To compile a call:

- save currently used registers
- place call args on stack or special registers
- emit call instruction
- restore registers; retrieve result from register

See: function-call overhead