6.110 Computer Language Engineering

Quiz 2 Review Session

May 1, 2024

Quiz 2: **Friday, May 3**

- Quiz will be in class, worth **10%** of the overall grade
- **Open-book**, any *direct* link from course website is OK except Godbolt, your own notes are OK
- No usage of compilers, and no wider internet or ChatGPT
- Covers second half of lecture content:
  - Program analysis / dataflow analysis
  - Loop optimizations
  - Register allocation
  - Parallelization
  - Foundations of dataflow analysis
- Past quizzes are now on course website

Program analysis ←

Loop optimizations

Register allocation

Parallelization

Foundations of dataflow

Outline

- Reaching Definitions
- Available Expressions
- Liveness

Reaching Definitions

- Concept of definition and use
  - a = x+y
  - is a definition of a
  - is a use of x and y
- A definition reaches a use if
  - value written by definition
  - may be read by use

Reaching Definitions and Constant Propagation

- Is a use of a variable a constant?
  - Check all reaching definitions
  - If all assign variable to same constant
  - Then use is in fact a constant
- Can replace variable with constant

Formalizing Analysis

- Each basic block has
  - IN - set of definitions that reach beginning of block
  - OUT - set of definitions that reach end of block
  - GEN - set of definitions generated in block
  - KILL - set of definitions killed in block
- GEN[s = s + a*b; i = i + 1;] = 0000011
- KILL[s = s + a*b; i = i + 1;] = 1010000
- Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

- IN[b] = OUT[b1] U ... U OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- IN[entry] = 0000000
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- Initialize with solution of OUT[b] = 0000000
- Repeatedly apply equations
  - IN[b] = OUT[b1] U ... U OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- Until reach fixed point
- Until equation application has no further effect
- Use a worklist to track which equation applications may have a further effect

Reaching Definitions Algorithm

```
for all nodes n in N
    OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Available Expressions

- An expression x+y is available at a point p if
  - every path from the initial node to p must evaluate x+y before reaching p,
  - and there are no assignments to x or y after the evaluation but before p.
- Available Expression information can be used to do global (across basic blocks) CSE
- If expression is available at use, no need to reevaluate it

Formalizing Analysis

- Each basic block has
  - IN - set of expressions available at start of block
  - OUT - set of expressions available at end of block
  - GEN - set of expressions computed in block
  - KILL - set of expressions killed in in block
- GEN[x = z; b = x+y] = 1000
- KILL[x = z; b = x+y] = 1001
- Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

- IN[b] = OUT[b1] ∩ ... ∩ OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- IN[entry] = 0000
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- IN[entry] = 0000
- Initialize OUT[b] = 1111
- Repeatedly apply equations
  - IN[b] = OUT[b1] $\cap$ ... $\cap$ OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) U GEN[b]
- Use a worklist algorithm to reach fixed point

Available Expressions Algorithm

```
for all nodes n in N
    OUT[n] = E;  // OUT[n] = E - KILL[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = E; // E is set of all expressions
    for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Liveness Analysis

- A variable v is live at point p if
  - v is used along some path starting at p, and
  - no definition of v along the path before the use.
- When is a variable v dead at point p?
  - No use of v on any path from p to exit node, or
  - If all paths from p redefine v before using v.

What Use is Liveness Information?

- Register allocation.
  - If a variable is dead, can reassign its register
- Dead code elimination.
  - Eliminate assignments to variables not read later.
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
  - Can eliminate other dead assignments.
  - Handle by making all externally visible variables live on exit from CFG

Conceptual Idea of Analysis

- Simulate execution
- But start from exit and go backwards in CFG
- Compute liveness information from end to beginning of basic blocks

Formalizing Analysis

- Each basic block has
  - IN - set of variables live at start of block
  - OUT - set of variables live at end of block
  - USE - set of variables with upwards exposed uses in block
  - DEF - set of variables defined in block
- USE[x = z; x = x+1;] = { z } (x not in USE)
- DEF[x = z; x = x+1;y = 1;] = {x, y}
- Compiler scans each basic block to derive USE and DEF sets

Algorithm

```
for all nodes n in N - { Exit }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    OUT[n] = emptyset;
    for all nodes s in successors(n)
        OUT[n] = OUT[n] U IN[p];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

Similar to Other Dataflow Algorithms

- Backwards analysis, not forwards
- Still have transfer functions
- Still have confluence operators
- Can generalize framework to work for both forwards and backwards analyses

Comparison

Reaching Definitions

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Available Expressions

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = E;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
            for all nodes s in successors(n)
            Changed = Changed U { s };
```

Liveness

```
for all nodes n in N - { Exit }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    OUT[n] = emptyset;
    for all nodes s in successors(n)
        OUT[n] = OUT[n] U IN[p];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

Comparison

Reaching Definitions

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Available Expressions

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = E;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Comparison

Reaching Definitions

```
for all nodes n in N
    OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

Liveness

```
for all nodes n in N
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    OUT[n] = emptyset;
    for all nodes s in successors(n)
        OUT[n] = OUT[n] U IN[p];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

Analysis Information Inside Basic Blocks

- One detail:
  - Given dataflow information at IN and OUT of node
  - Also need to compute information at each statement of basic block
  - Simple propagation algorithm usually works fine
  - Can be viewed as restricted case of dataflow analysis

Pessimistic vs. Optimistic Analyses

- Available expressions is optimistic (for common sub-expression elimination)
  - Assume expressions are available at start of analysis
  - Analysis eliminates all that are not available
  - Cannot stop analysis early and use current result
- Live variables is pessimistic (for dead code elimination)
  - Assume all variables are live at start of analysis
  - Analysis finds variables that are dead
  - Can stop analysis early and use current result
- Dataflow setup same for both analyses
- Optimism/pessimism depends on intended use

Summary

- Basic Blocks and Basic Block Optimizations
  - Copy and constant propagation
  - Common sub-expression elimination
  - Dead code elimination
- Dataflow Analysis
  - Control flow graph
  - IN[b], OUT[b], transfer functions, join points
- Paired analyses and transformations
  - Reaching definitions/constant propagation
  - Available expressions/common sub-expression elimination
  - Liveness analysis/Dead code elimination
- Stacked analysis and transformations work together

For the quiz, you should know how to:

- Perform reaching definitions, available expressions, and liveness analysis given a CFG (statements or basic blocks)
- Write dataflow equations for these optimizations
- Perform global optimizations.
- Explain advantages and limitations of each optimization

Program analysis

Loop optimizations ←

Register allocation

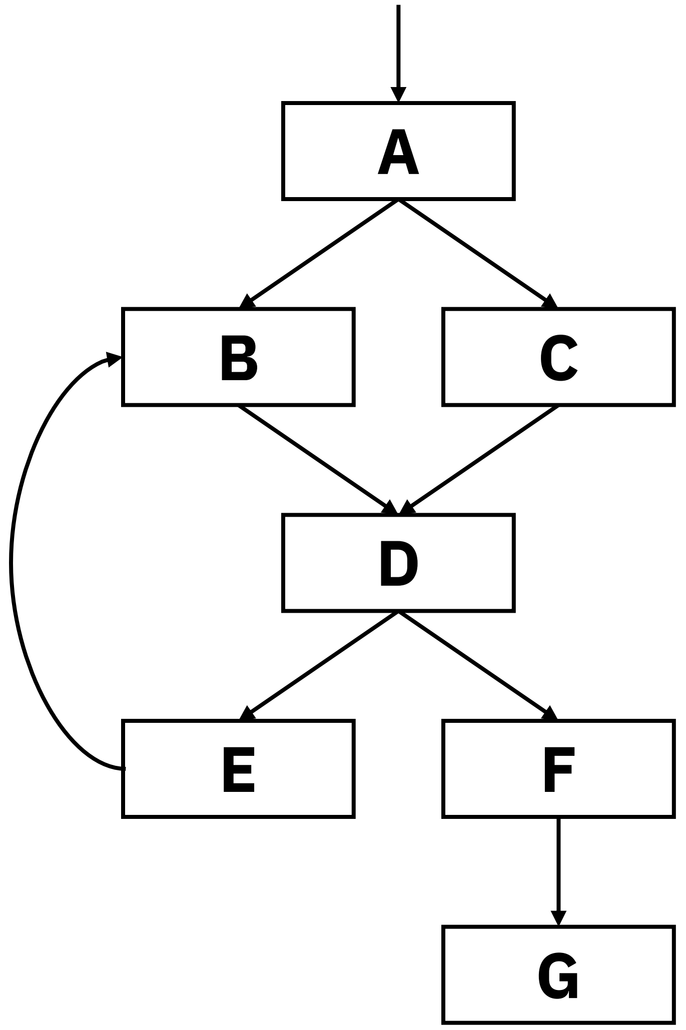
Parallelization

Foundations of dataflow

Domination

In a control-flow graph:

- A node **n** **dominates** a node **m** if every path from the entry block to **m** goes through **n.**

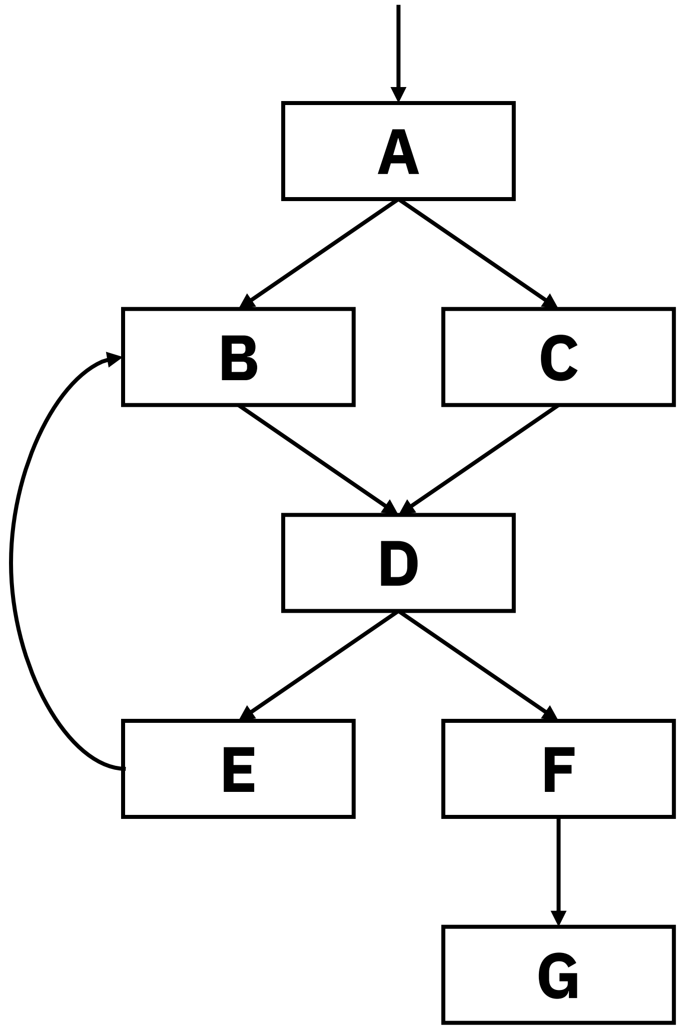


D dominates **D, E, F, G**

Domination

In a control-flow graph:

- A node **n** **dominates** a node **m** if every path from the entry block to **m** goes through **n.**
  - If **m ≠ n**, then **n** ***strictly* dominates** **m.**

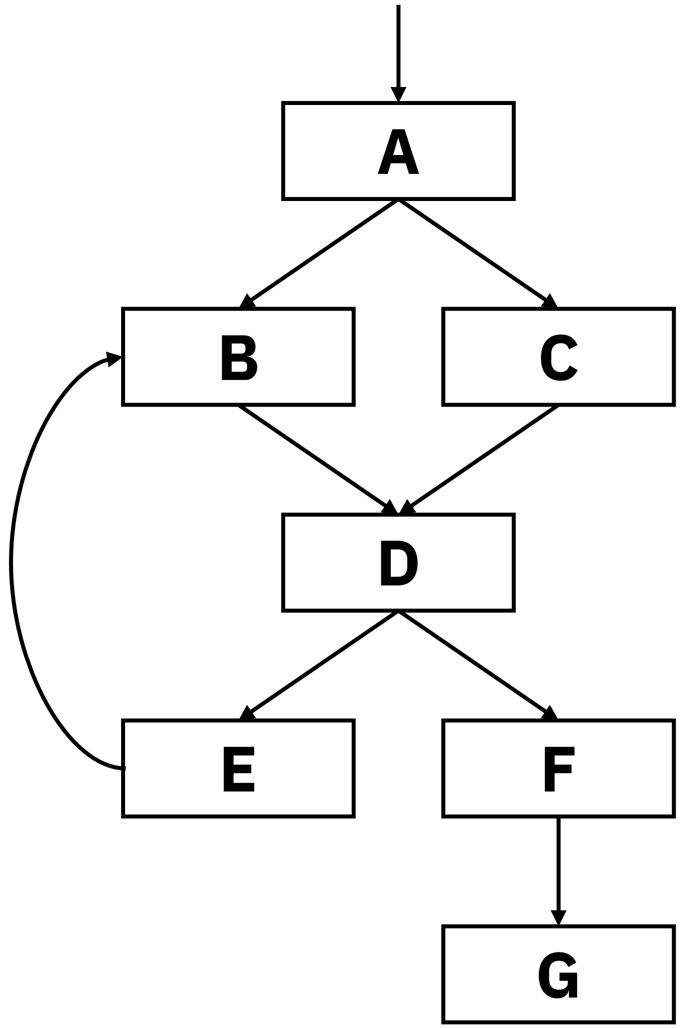


D strictly dominates **E, F, G**

Domination

In a control-flow graph:

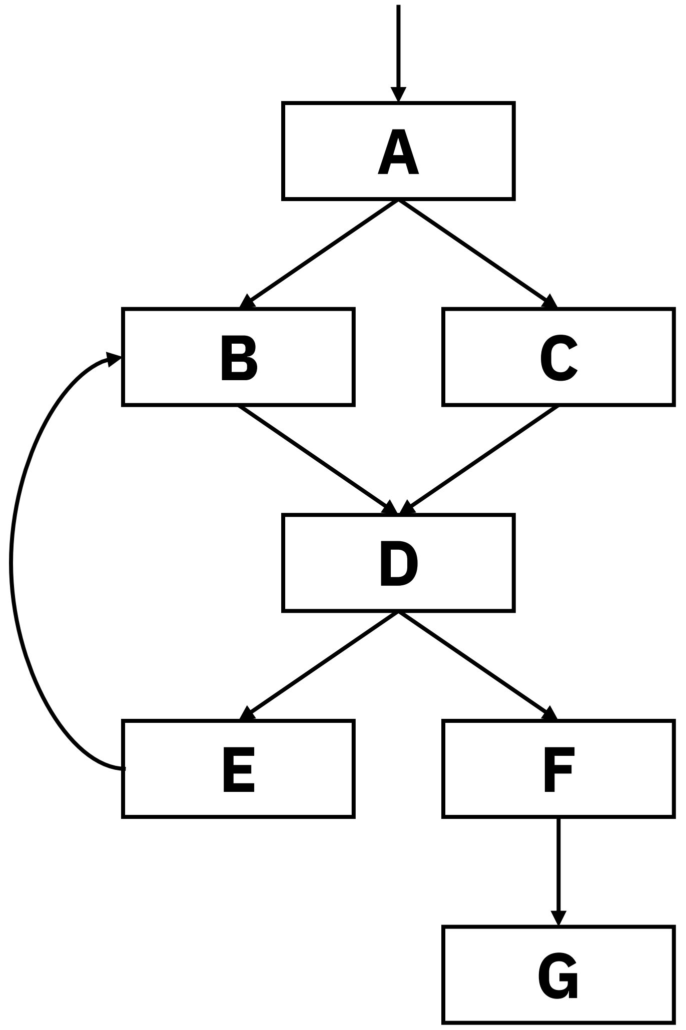
- A node **n** **dominates** a node **m** if every path from the entry block to **m** goes through **n.**
    - If **m ≠ n**, then **n** ***strictly* dominates** **m.**
    - If there are no nodes **x** such that **n** strictly dominates **x** and **x** strictly dominates **m**, then **n** ***immediately* dominates** **m.**



D immediately dominates **E, F**

Dominator tree

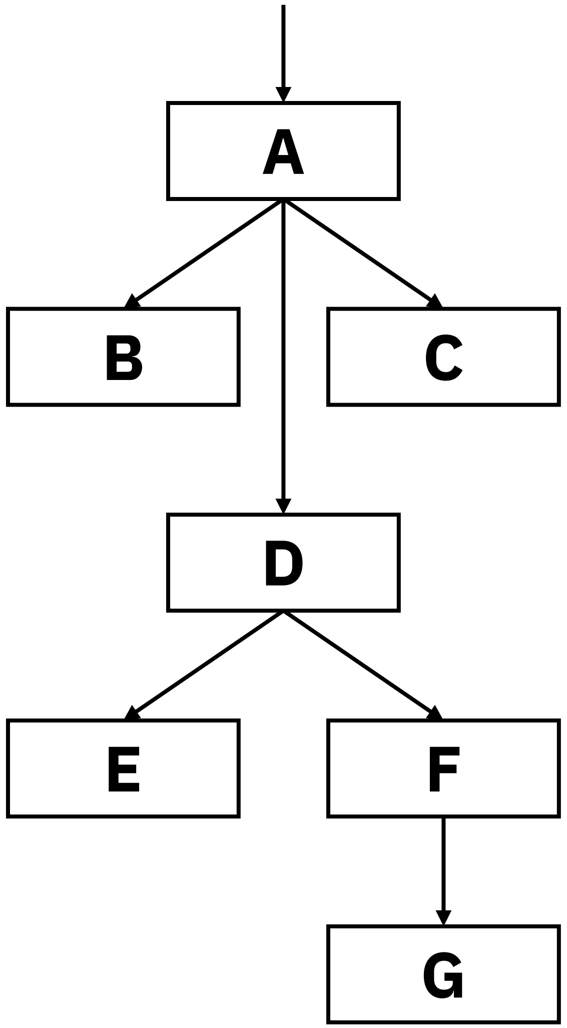
- Each node (except the entry node) has a unique **immediate dominator**



The immediate dominator of **D** is **A**

Dominator tree

- Each node (except the entry node) has a unique **immediate dominator**
- The **dominator tree** is the tree where there is an edge **n** to **m** if **n** immediately dominates **m**

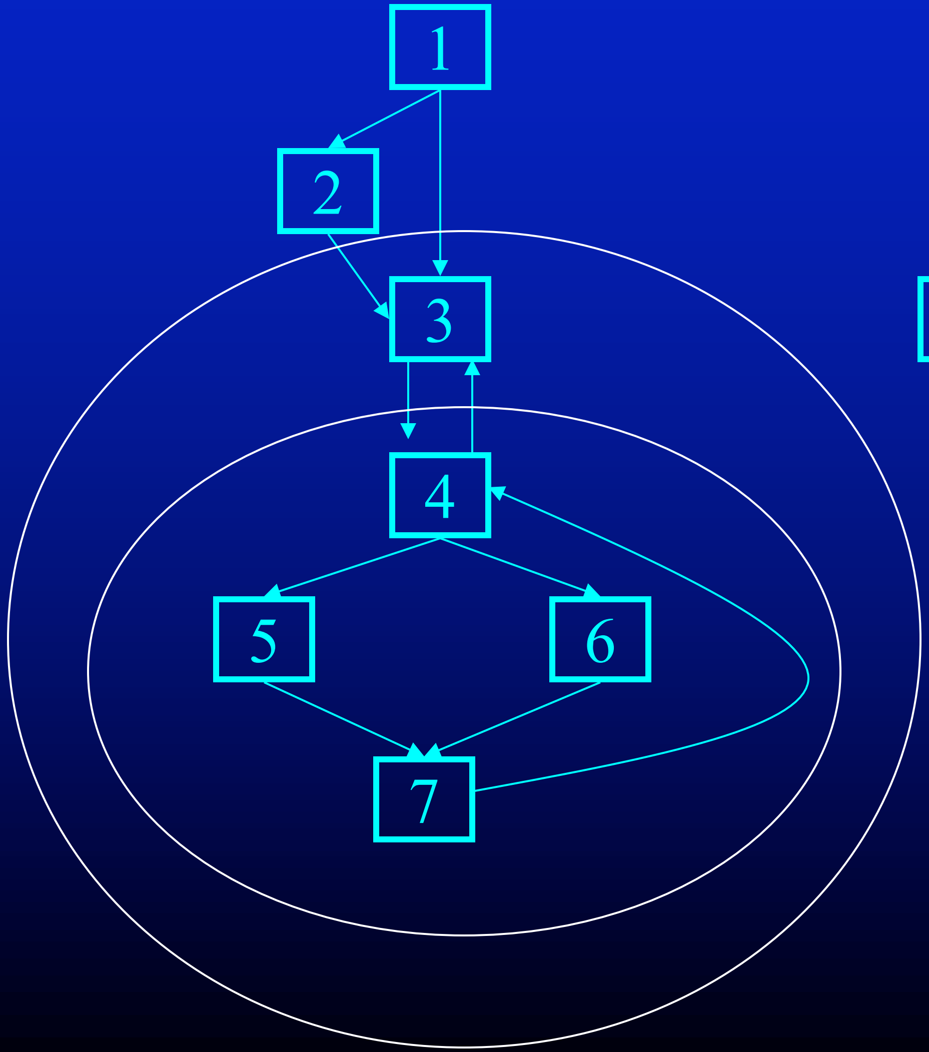


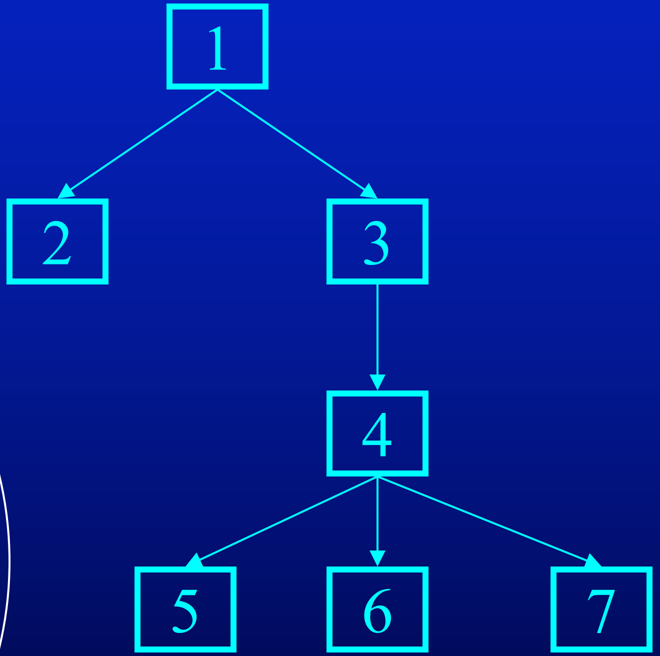
Dominator tree

Defining Loops

- Unique entry point – header
- At least one path back to header
- Find edges whose heads dominate tails
  - These edges are back edges of loops
  - Given back edge n→d
  - Loop consists of n, d plus all nodes that can reach n without going through d (all nodes "between" d and n)
  - d is loop header

Two Loops In Example





Loop Construction Algorithm

```
insert(m)
    if $m \notin$ loop then
        loop = loop $\cup \{m\}$
        push m onto stack
loop(d,n)
    loop = { d }; stack = $\emptyset$; insert(n);
    while stack not empty do
        m = pop stack;
        for all $p \in$ pred(m) do insert(p)
```

Loop Optimizations

- Now that we have the loop, can optimize it!
- Loop invariant code motion
  - Stick loop invariant code in the header

Detecting Loop Invariant Code

- A statement is invariant if operands are
  - Constant,
  - Have all reaching definitions outside loop, or
  - Have exactly one reaching definition, and that definition comes from an invariant statement
- Concept of exit node of loop
  - node with successors outside loop

Loop Invariant Code Detection Algorithm

```
for all statements in loop
    if operands are constant or have all reaching definitions
    outside loop, mark statement as invariant
do
    for all statements in loop not already marked invariant
        if operands are constant, have all reaching
        definitions outside loop, or have exactly one reaching
        definition from invariant statement then
            mark statement as invariant
until find no more invariant statements
```

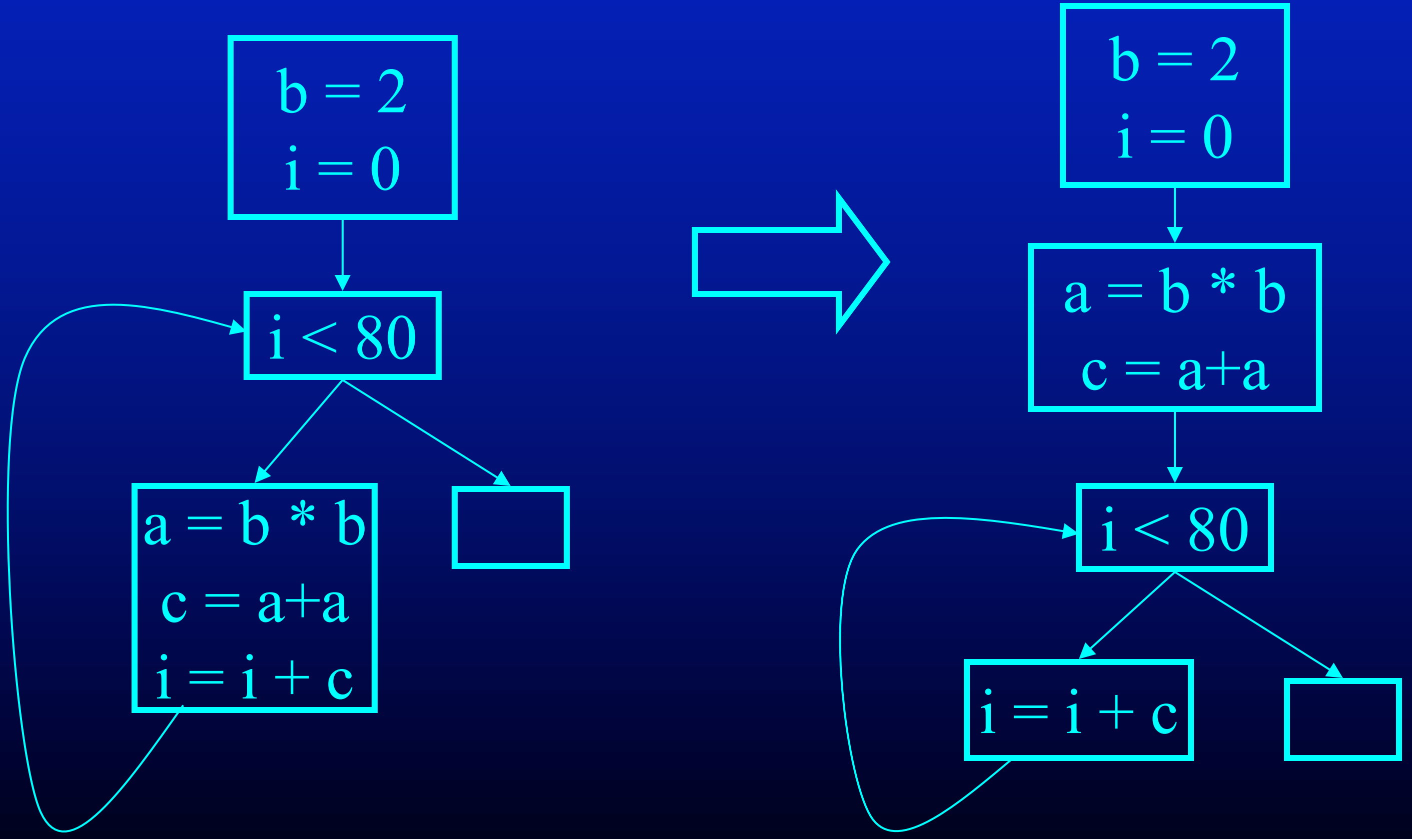
Loop Invariant Code Motion

- Conditions for moving a statement s:x:=y+z into loop header:
  - s dominates all exit nodes of loop
    - If it doesn't, some use after loop might get wrong value
    - Alternate condition: definition of x from s reaches no use outside loop (but moving s may increase run time)
  - No other statement in loop assigns to x
    - If one does, assignments might get reordered
  - No use of x in loop is reached by definition other than s
    - If one is, movement may change value read by use

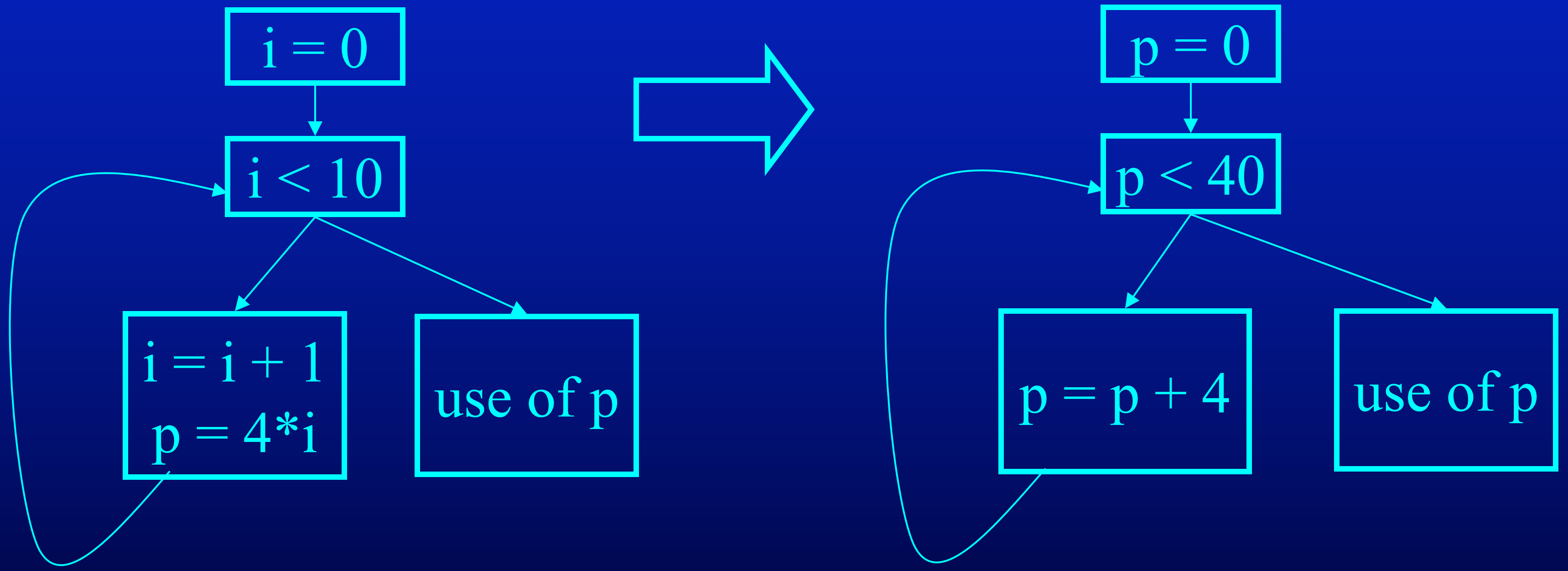
Order of Statements in Preheader

Preserve data dependences from original program

(can use order in which discovered by algorithm)



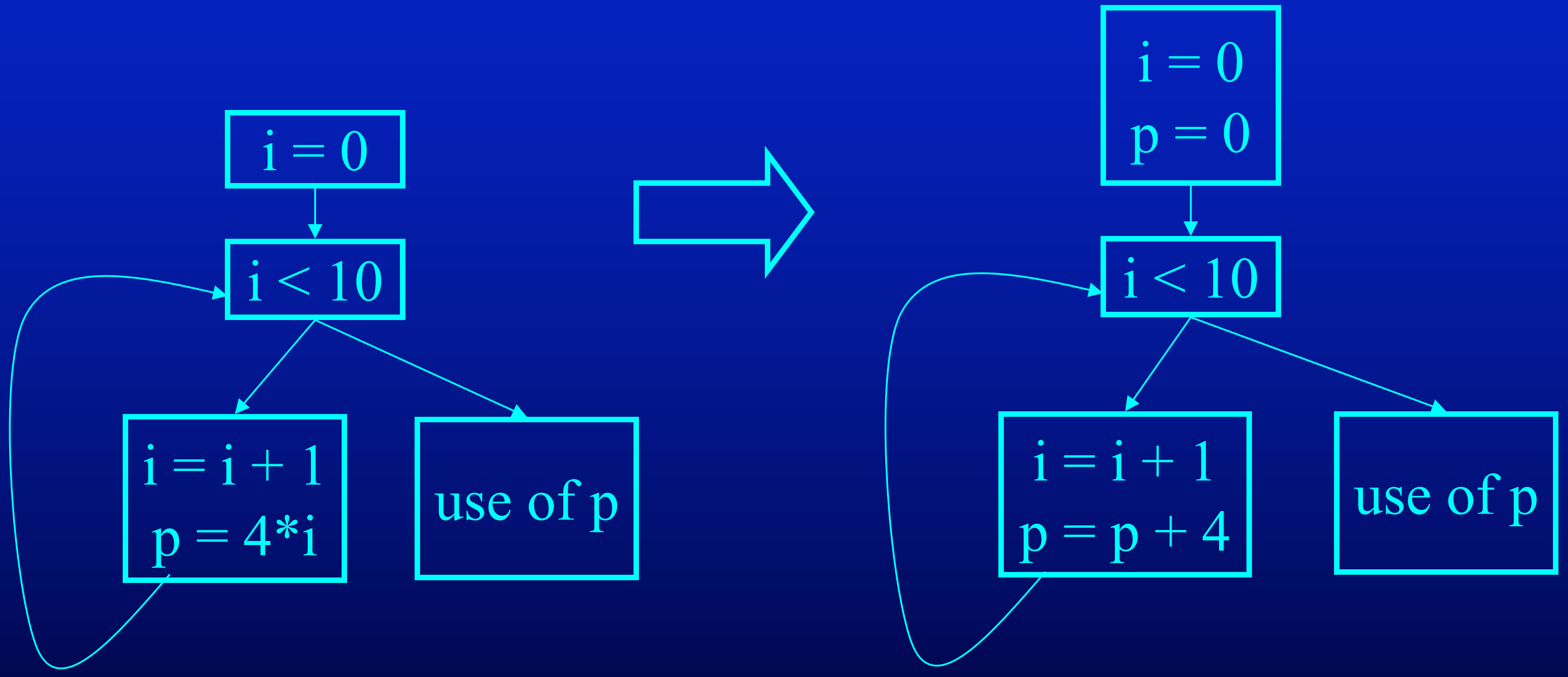
Induction Variable Elimination



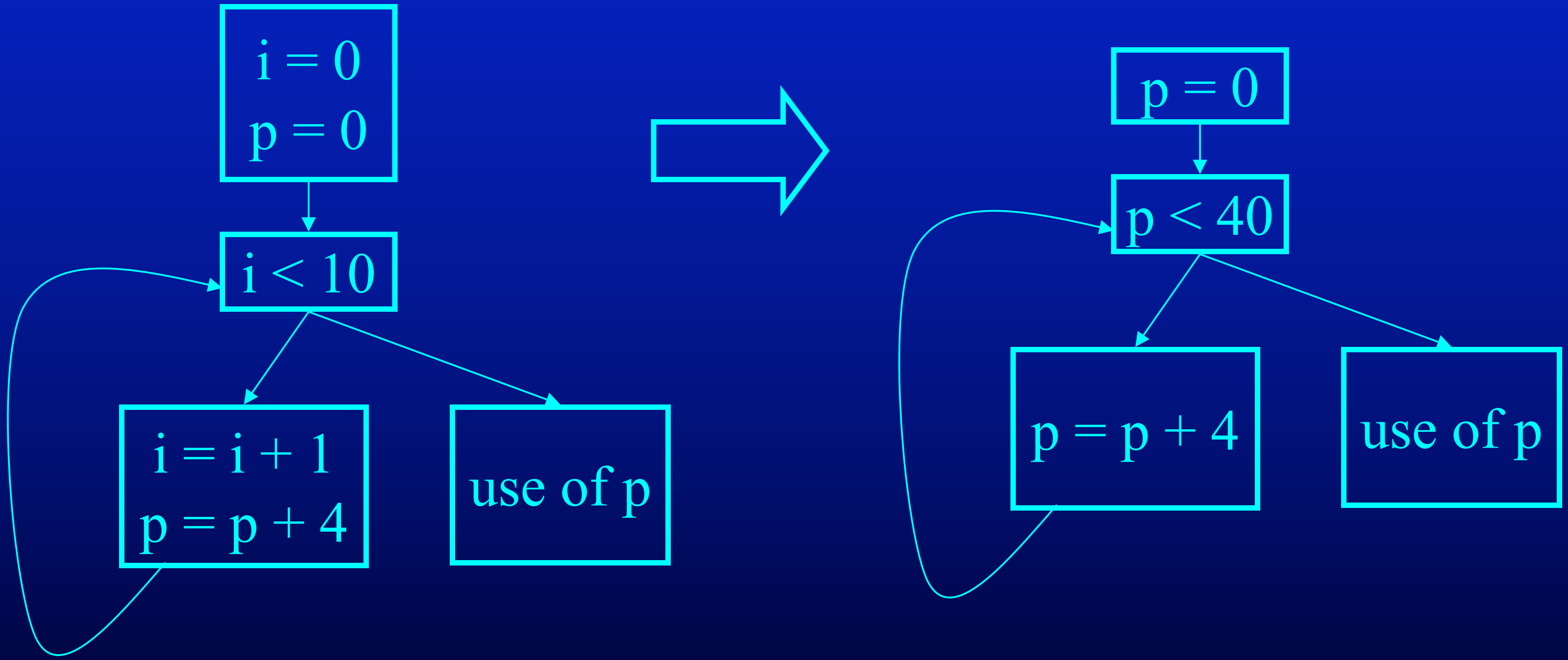
What is an Induction Variable?

- Base induction variable
  - Only assignments in loop are of form $i = i \pm c$
- Derived induction variables
  - Value is a linear function of a base induction variable
  - Within loop, $j = c*i + d$, where i is a base induction variable
  - Very common in array index expressions – an access to a[i] produces code like $p = a + 4*i$

Strength Reduction for Derived Induction Variables



Elimination of Superfluous Induction Variables



Three Algorithms

- Detection of induction variables
  - Find base induction variables
  - Each base induction variable has a family of derived induction variables, each of which is a linear function of base induction variable
- Strength reduction for derived induction variables
- Elimination of superfluous induction variables

Output of Induction Variable Detection Algorithm

- Set of induction variables
  - base induction variables
  - derived induction variables
- For each induction variable j, a triple <i,c,d>
  - i is a base induction variable
  - value of j is i*c+d
  - j belongs to family of i

What is a base induction variable?

- Variable i with one assignment in loop
- Assignment of the form
  - i = i + c; // i has triple <i, 1, 0>
  - i = i – c; // i has triple <i, 1, 0>
  - Where c is constant
  - More generally, c is loop invariant

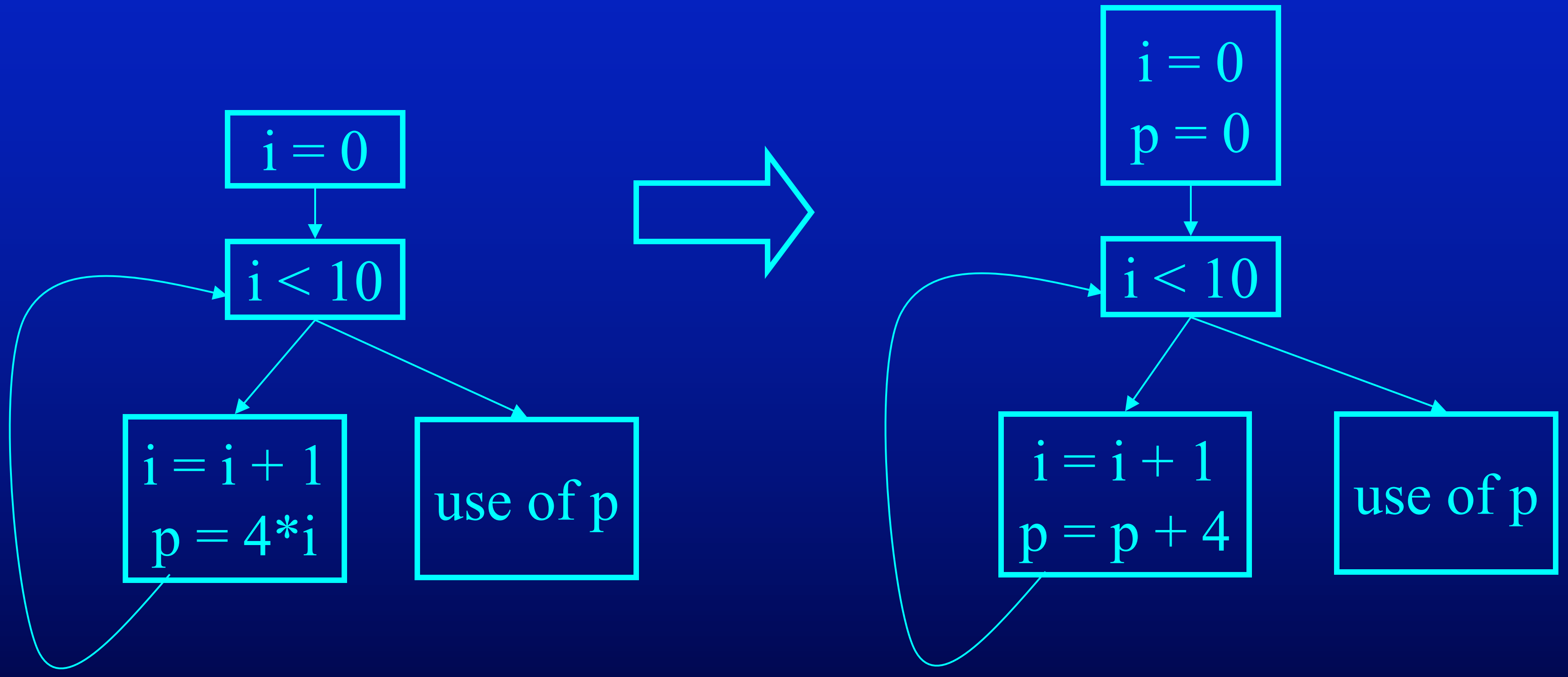
Induction Variable Detection Algorithm

```
Scan loop to find all base induction variables
do
    Scan loop to find all variables k with one assignment of
      form k = j*b where j is an induction variable with
      triple <i,c,d>
      make k an induction variable with triple <i,c*b,d*b>
    Scan loop to find all variables k with one assignment of
      form k = j±b where j is an induction variable with
      triple <i,c,d>
      make k an induction variable with triple <i,c,b±d>
until no more induction variables found
```

Strength Reduction Algorithm

```
for all derived induction variables j with triple <i,c,d>
    Create a new variable s
    Replace assignment j = i*c+d with j = s
    Immediately after each assignment i = i + e, insert statement s =
    s + c*e (c*e is constant)
    place s in family of i with triple <i,c,d>
    Insert s = c*i+d into preheader
```

Strength Reduction for Derived Induction Variables



Induction Variable Elimination

Choose a base induction variable i such that

- only uses of i are in
  - termination condition of the form $i < n$
  - assignment of the form $i = i + m$

Choose a derived induction variable k with <i,c,d>

- Replace termination condition with $k < c*n+d$

Why?

$$k = i*c+d \Rightarrow i < n \Leftrightarrow i*c < c*n \Leftrightarrow i*c+d < c*n+d$$
$$\Leftrightarrow k < c*n+d$$

For the quiz, you should know:

- How to identify loops in code
- Domination relation, dominator trees
- How to identify loop-invariant code
- Reasoning about induction variables

Program analysis

Loop optimizations

Register allocation ←

Parallelization

Foundations of dataflow

Outline

- What is register allocation
- Key ideas in register allocation
- Webs
- Interference Graphs
- Graph coloring
- Splitting
- More optimizations

Summary of Register Allocation

- You want to put each temporary in a register
  - *But*, you don't have enough registers.
- Key Ideas:
  - When a temporary goes dead, its register can be reused
  - Two live temporaries can't use the same register at the same time

Summary of Register Allocation

- When a temporary goes dead, its register can be reused
- Example:

```
a := c + d
e := a + b
f := e - 1
```

(assume that a and e die after use)

- temporaries a, e and f can go in the same register

```
r1 := c + d
r1 := r1 + b
r1:= r1 – 1
```

Summary of Register Allocation

- Two live temporaries can’t use the same register at the same time
- Example 2:

```
a := c + d
e := a + b
f := e - a
```

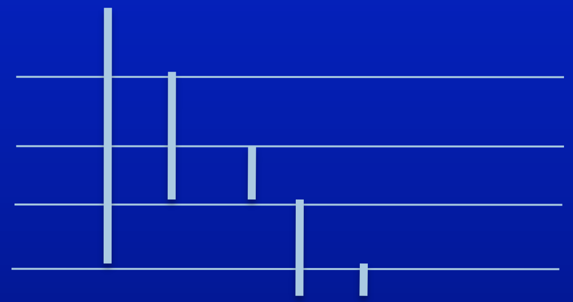
- temporaries e and a can not go in the same register

```
r1 := c + d
r2 := r1 + b
r1 := r2 – r1
```

When things don't work out

- Sometimes more live variables than registers

```
a := c + d
e := c + b
f := e – c
g := e + f
h := a + g
```

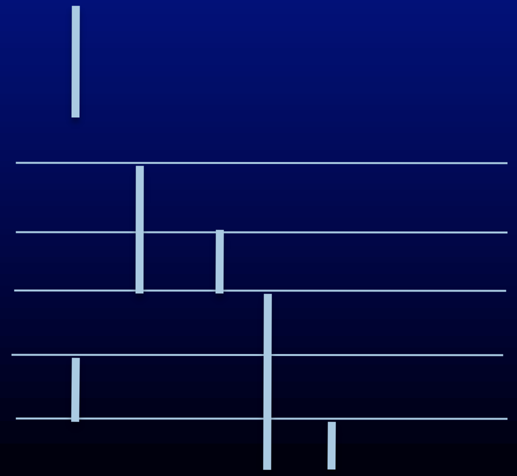


Won't work for 2 registers

(assume only g and h live at the end)

- You can split a live range by storing to memory

```
a := c + d
store a
e := c + b
f := e – c
g := e + f
load a
h := a + g
```



Web-Based Register Allocation

- Determine live ranges for each value (*web*)
- Determine overlapping ranges (interference)
- Compute the benefit of keeping each web in a register (spill cost)
- Decide which webs get a register (allocation)
- Split webs if needed (spilling and splitting)
- Assign hard registers to webs (assignment)
- Generate code including spills (code gen)

Webs

- Starting Point: def-use chains (DU chains)
  - Connects definition to all reachable uses
- Conditions for putting defs and uses into same web
  - Def and all reachable uses must be in same web
  - All defs that reach same use must be in same web
- Use a union-find algorithm

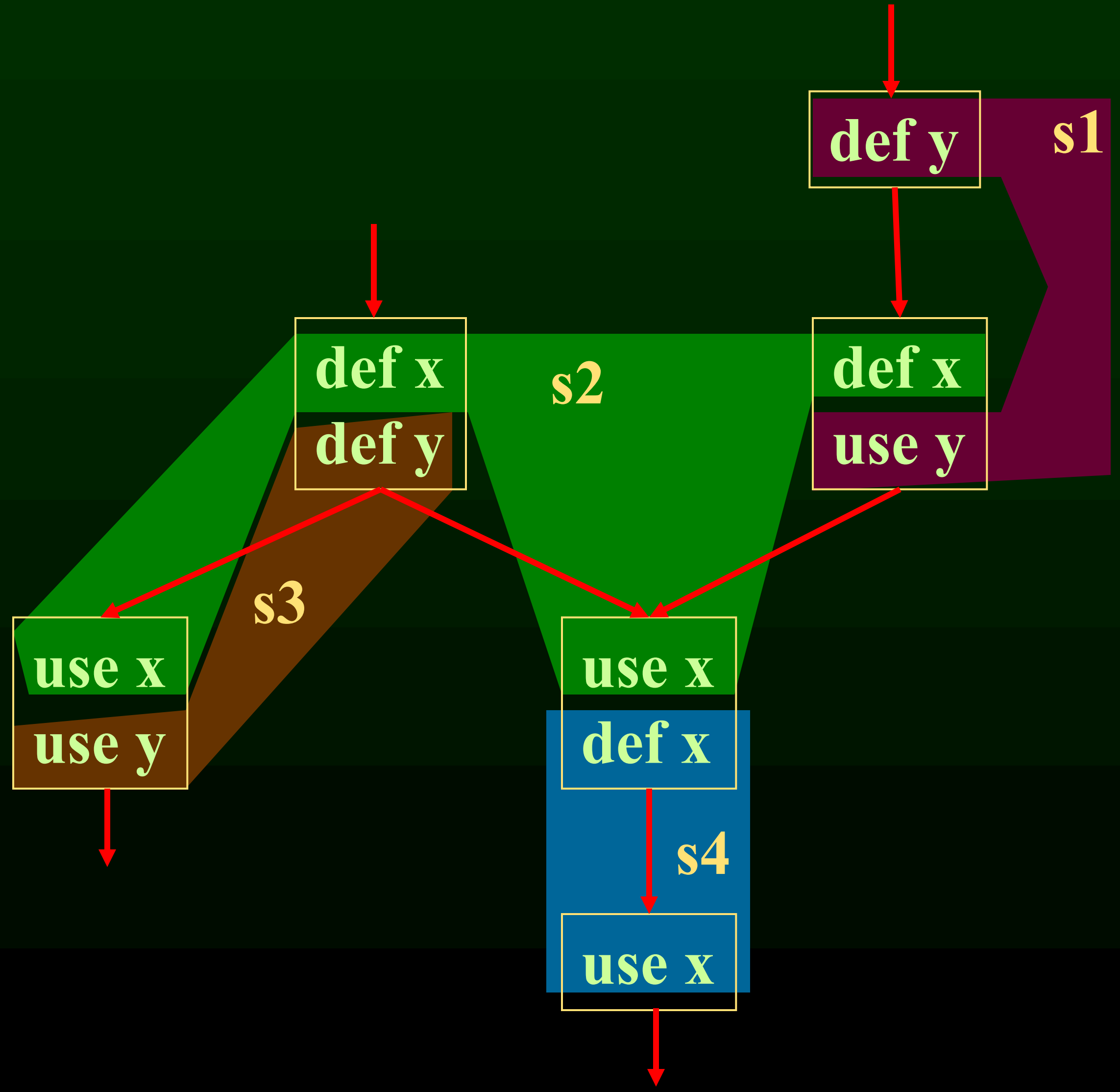
Webs

- Web is unit of register allocation

- If web allocated to a given register R
  - All definitions computed into R
  - All uses read from R

- If web allocated to a memory location M
  - All definitions computed into M
  - All uses read from M

Example

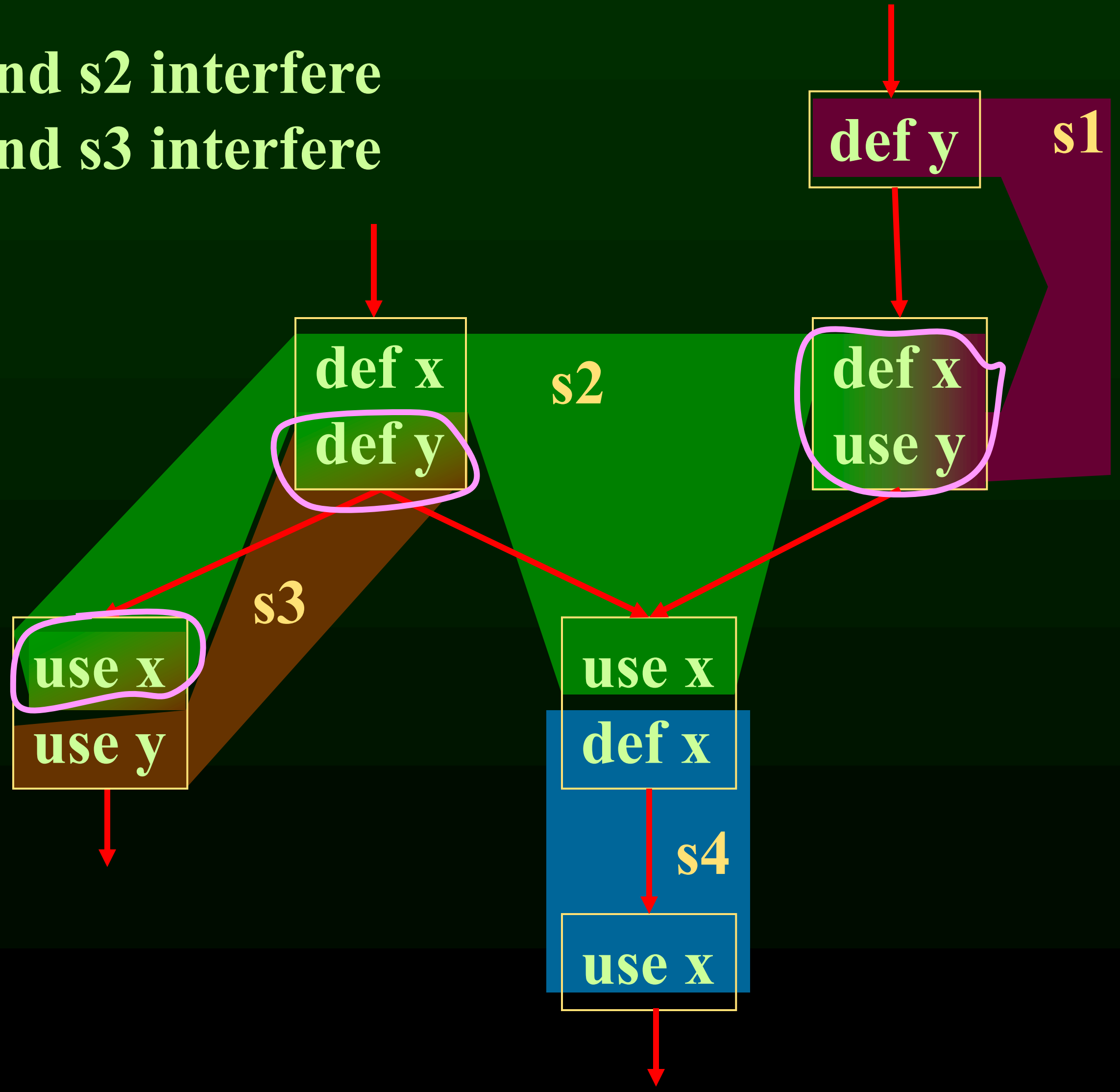


Interference

- Two webs interfere if their live ranges overlap (have a nonemtpy intersection)
- If two webs interfere, values must be stored in different registers or memory locations
- If two webs do not interfere, can store values in same register or memory location

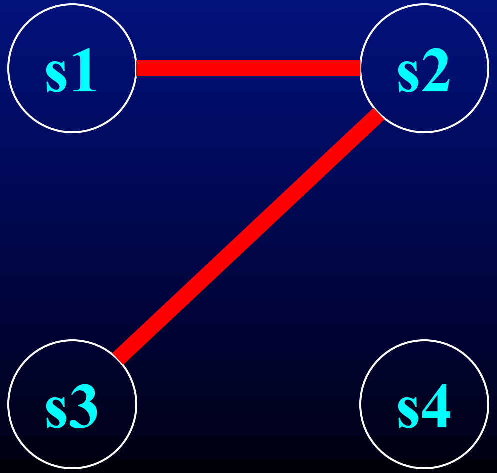
Example

Webs s1 and s2 interfere
Webs s2 and s3 interfere



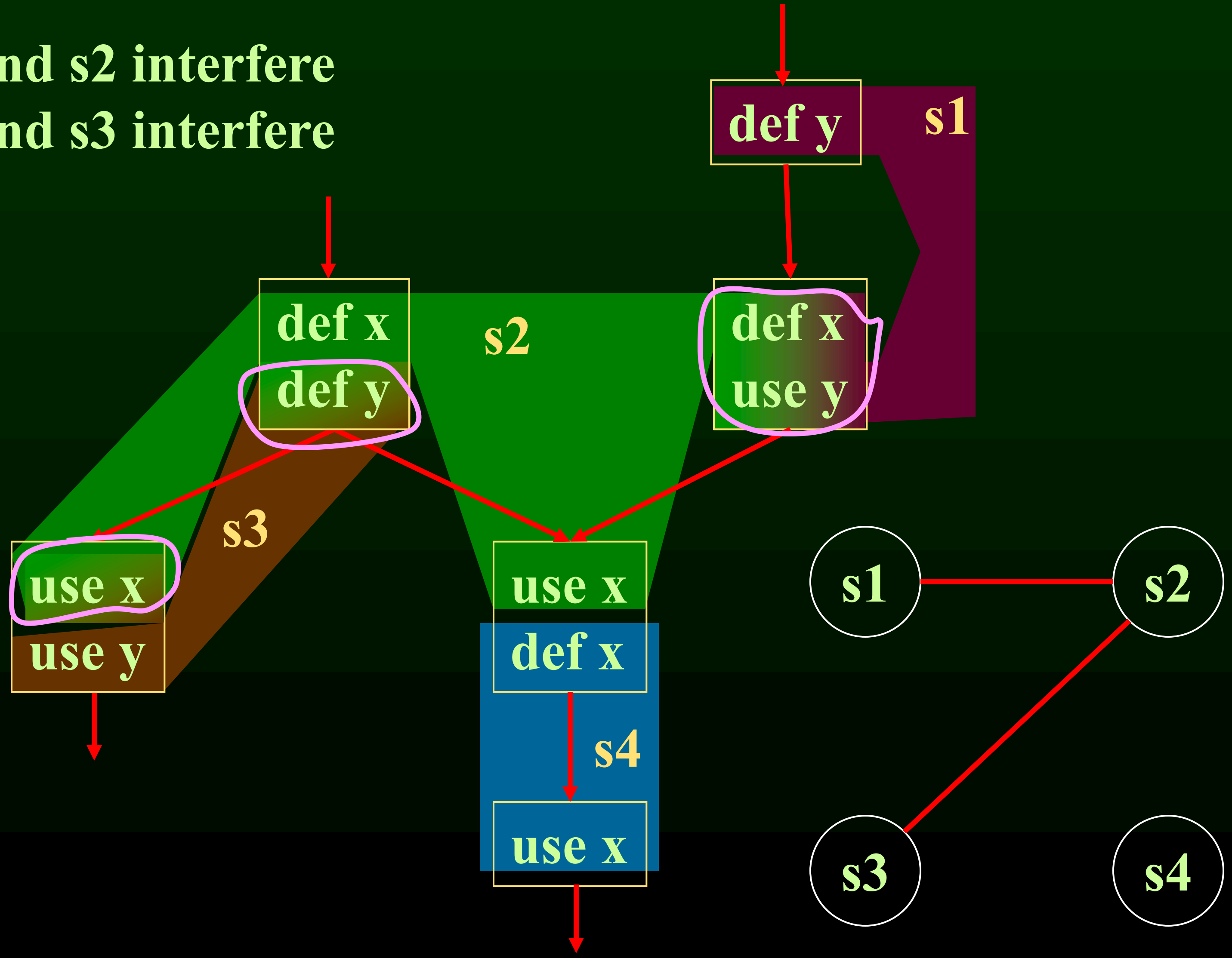
Interference Graph

- Representation of webs and their interference
  - Nodes are the webs
  - An edge exists between two nodes if they interfere



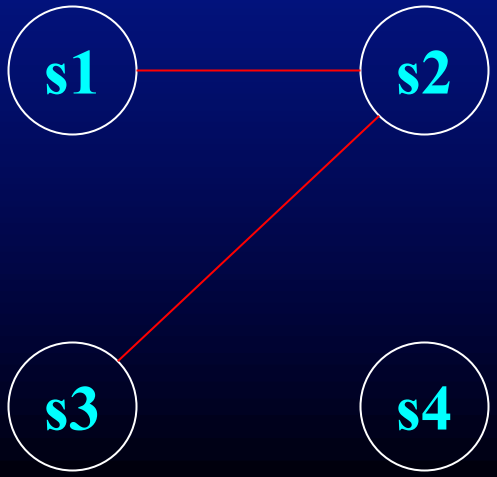
Example

Webs s1 and s2 interfere
Webs s2 and s3 interfere



Register Allocation Using Graph Coloring

- Each web is allocated a register
  - each node gets a register (color)
- If two webs interfere they cannot use the same register
  - if two nodes have an edge between them, they cannot have the same color



Graph Coloring

- Assign a color to each node in graph
- Two nodes connected to same edge must have different colors
- Classic problem in graph theory
- NP complete
  - But good heuristics exist for register allocation

Heuristics for Register Coloring

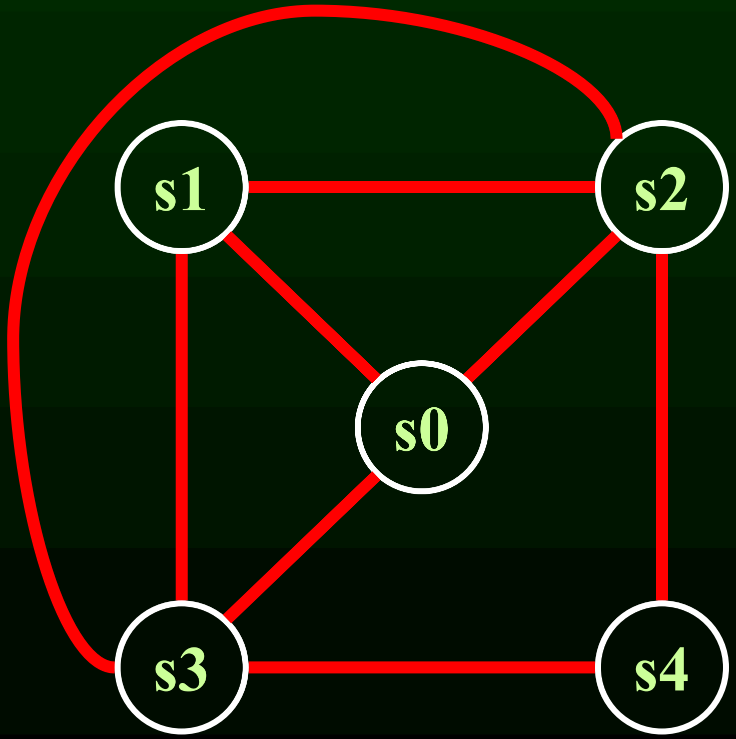
- Coloring a graph with N colors
- If degree < N (degree of a node = # of edges)
  - Node can always be colored
  - After coloring the rest of the nodes, you'll have at least one color left to color the current node
- If degree >= N
  - still may be colorable with N colors

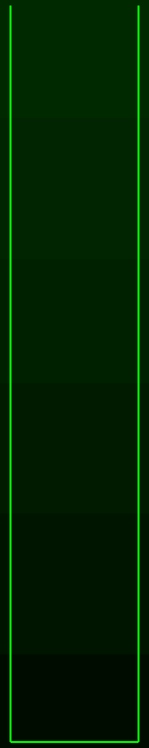
Heuristics for Register Coloring

- Remove nodes that have degree < N
  - push the removed nodes onto a stack
- When all the nodes have degree >= N
  - Find a node to spill (no color for that node)
  - Remove that node
- When empty, start to color
  - pop a node from stack back
  - Assign it a color that is different from its connected nodes (since degree < N, a color should exist)

Another Coloring Example

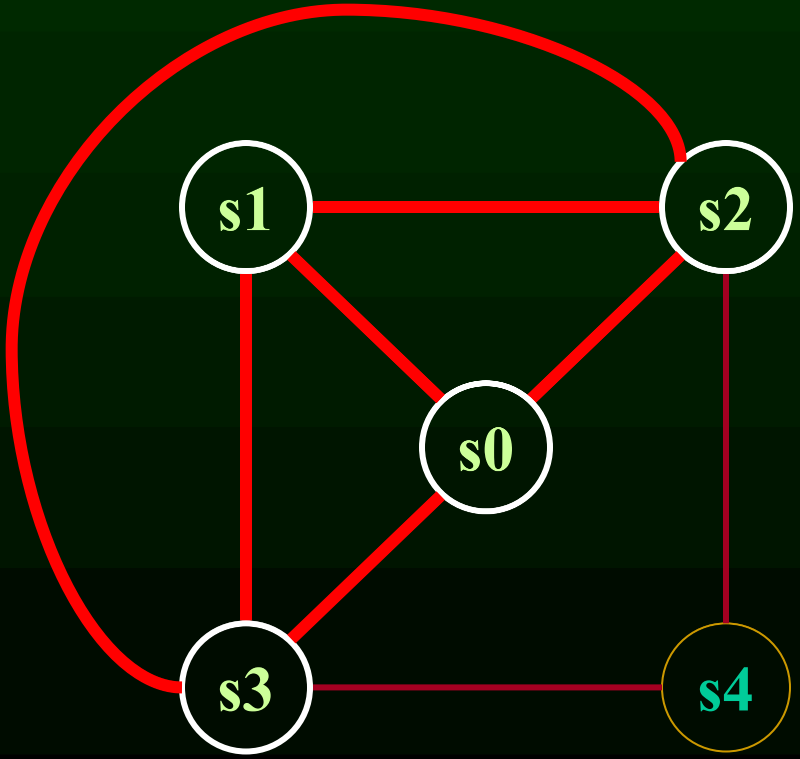
N = 3

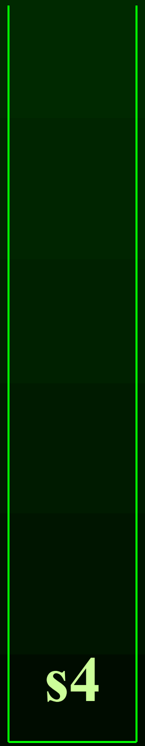




Another Coloring Example

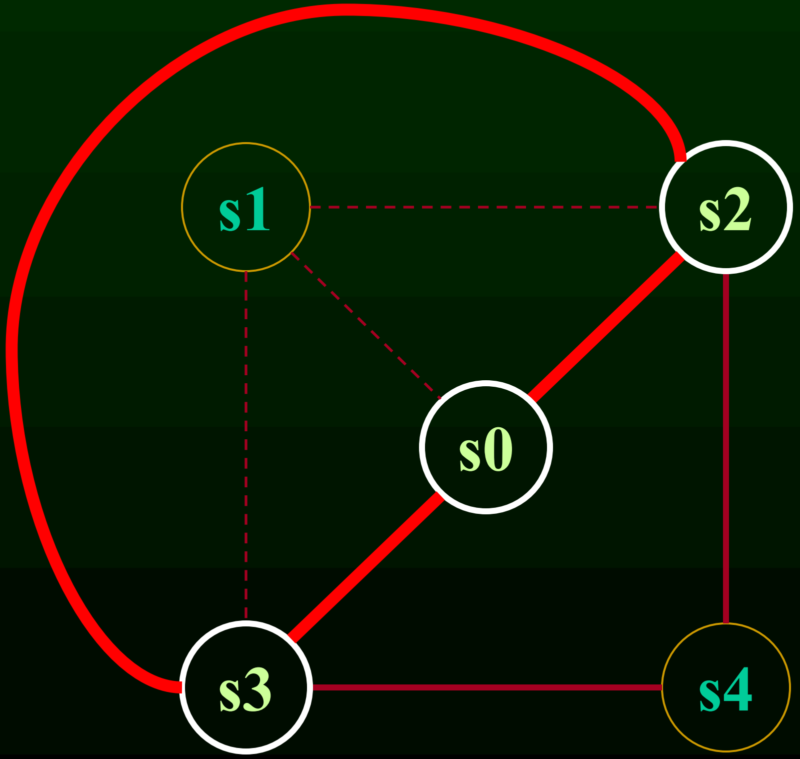
N = 3

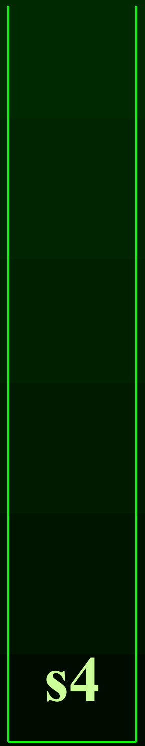




Another Coloring Example

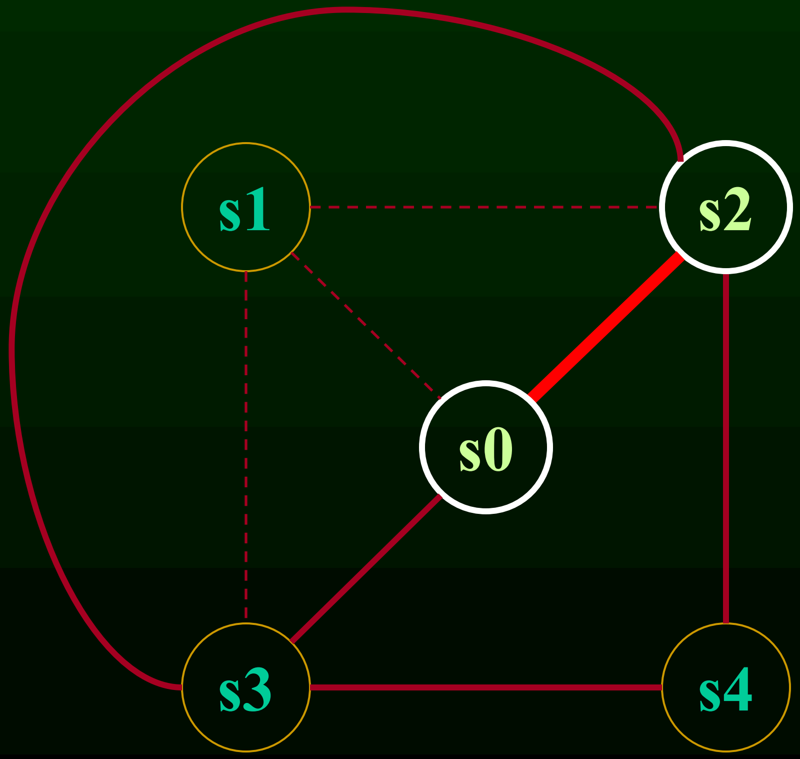
N = 3

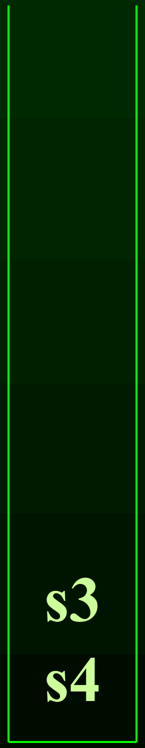




Another Coloring Example

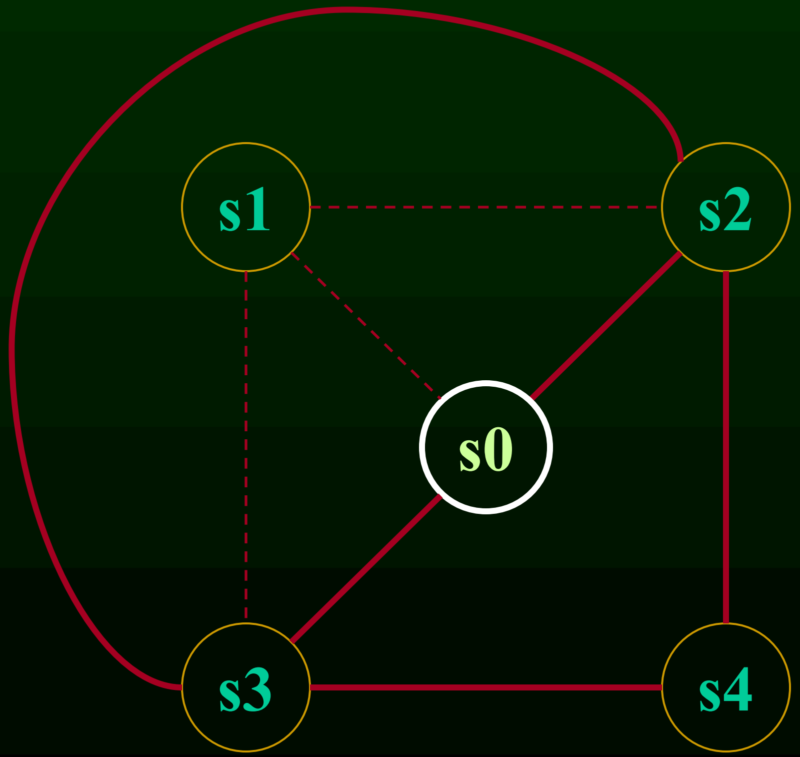
N = 3

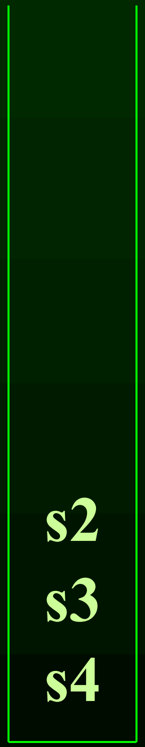




Another Coloring Example

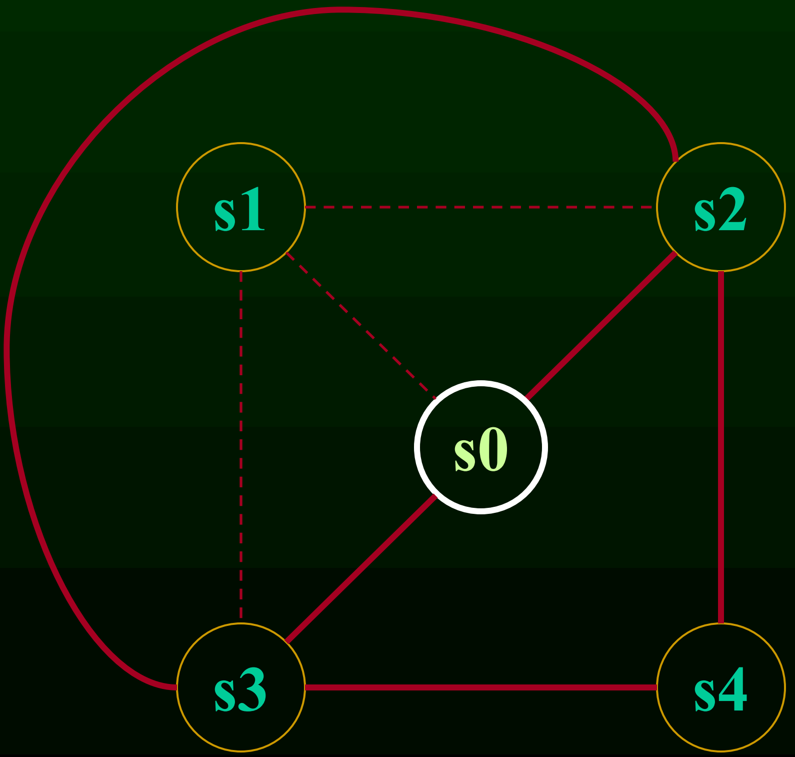
N = 3

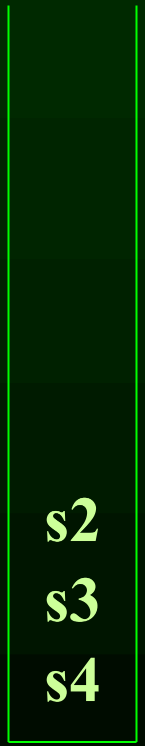




Another Coloring Example

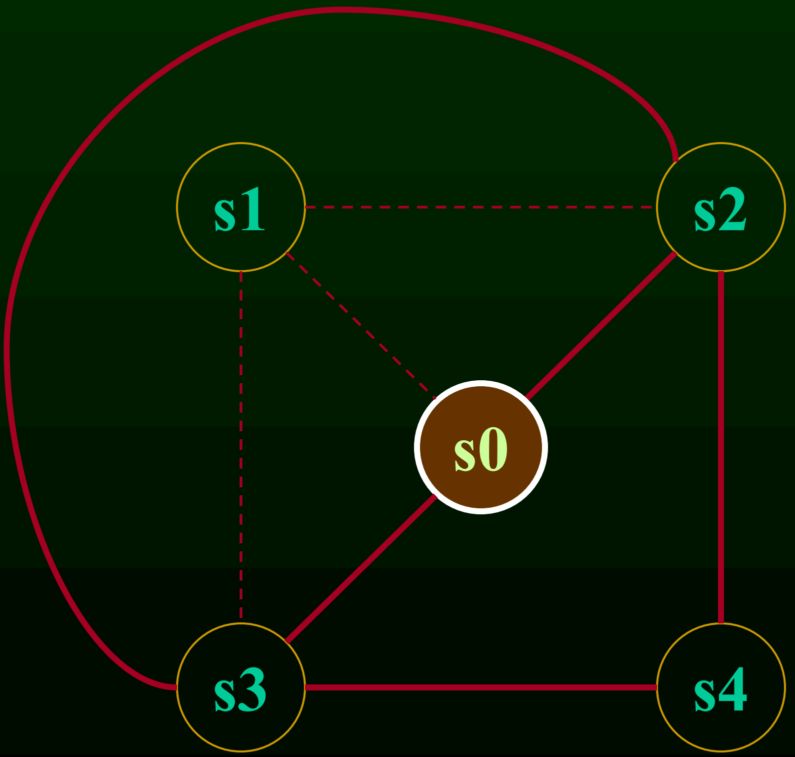
N = 3 🟦 🟩 🟫

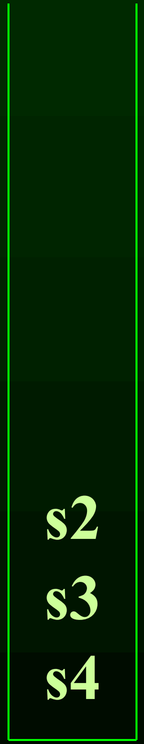




Another Coloring Example

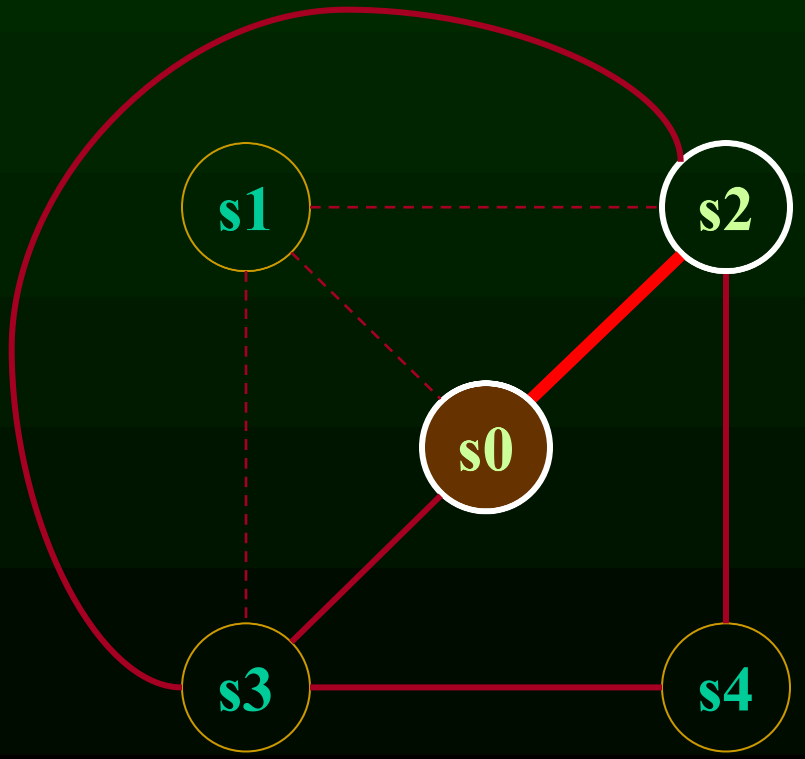
N = 3 🟦 🟩 🟫

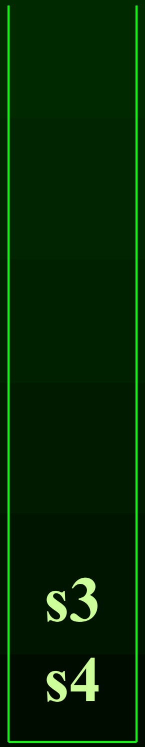




Another Coloring Example

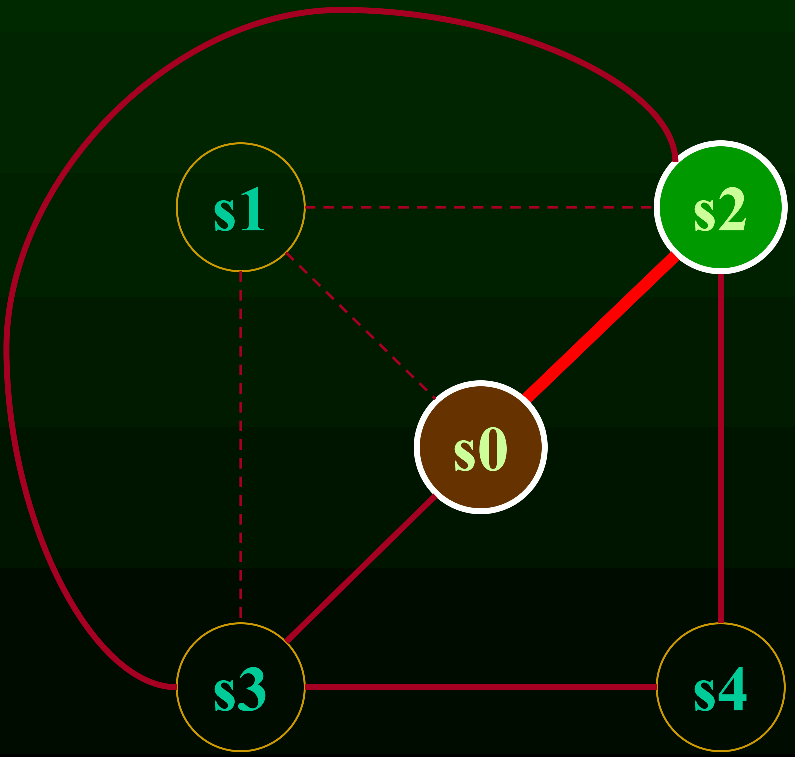
N = 3 🟦 🟩 🟫

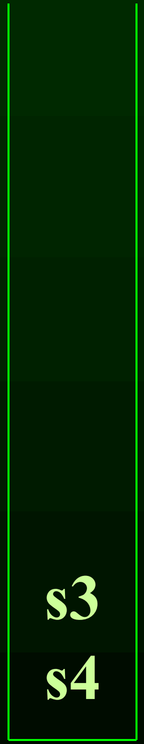




Another Coloring Example

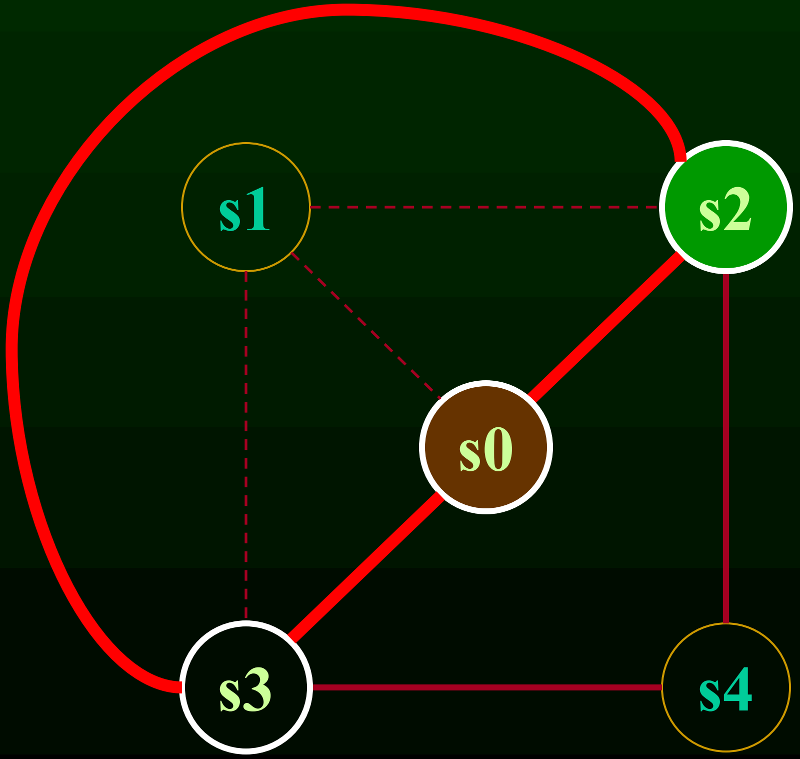
N = 3 🟦 🟩 🟫

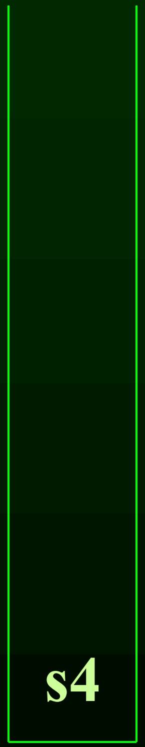




Another Coloring Example

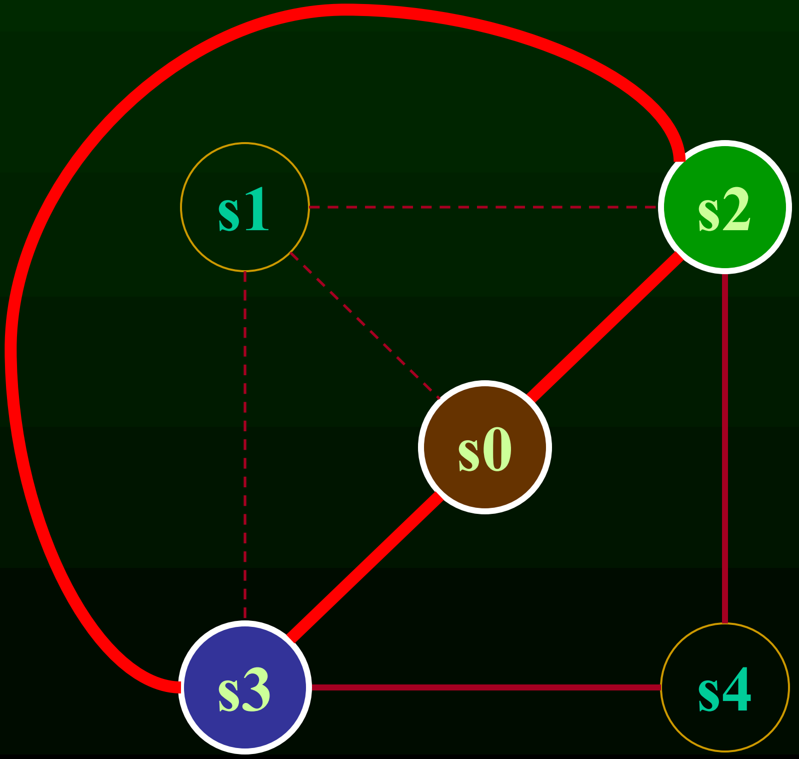
N = 3 🟦 🟩 🟫

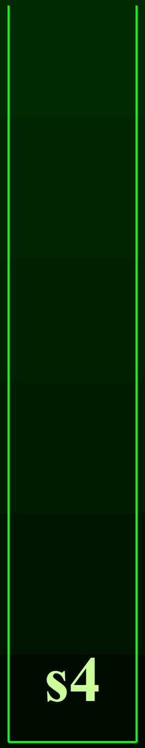




Another Coloring Example

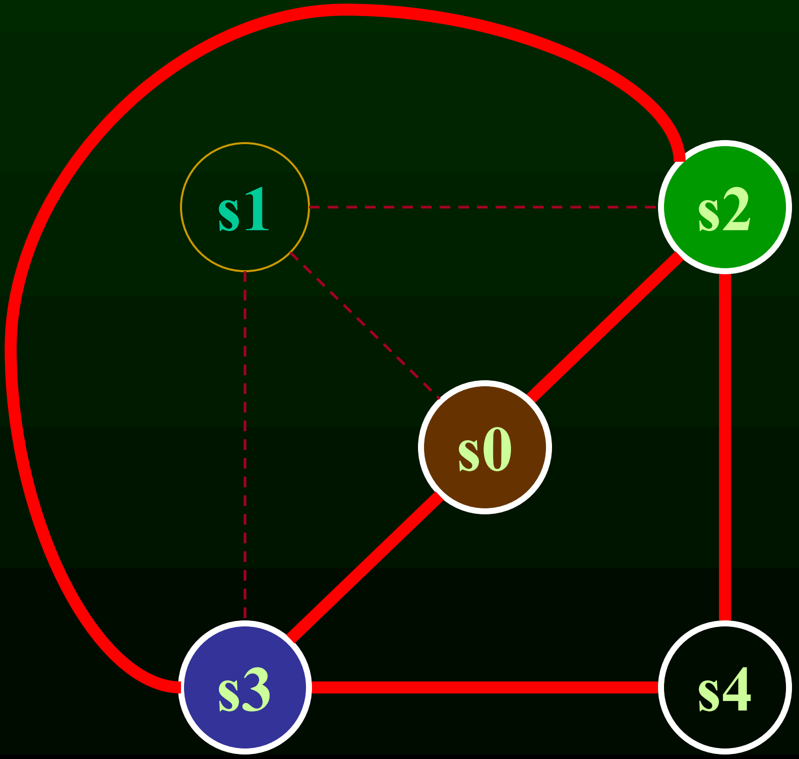
N = 3 🟦 🟩 🟫





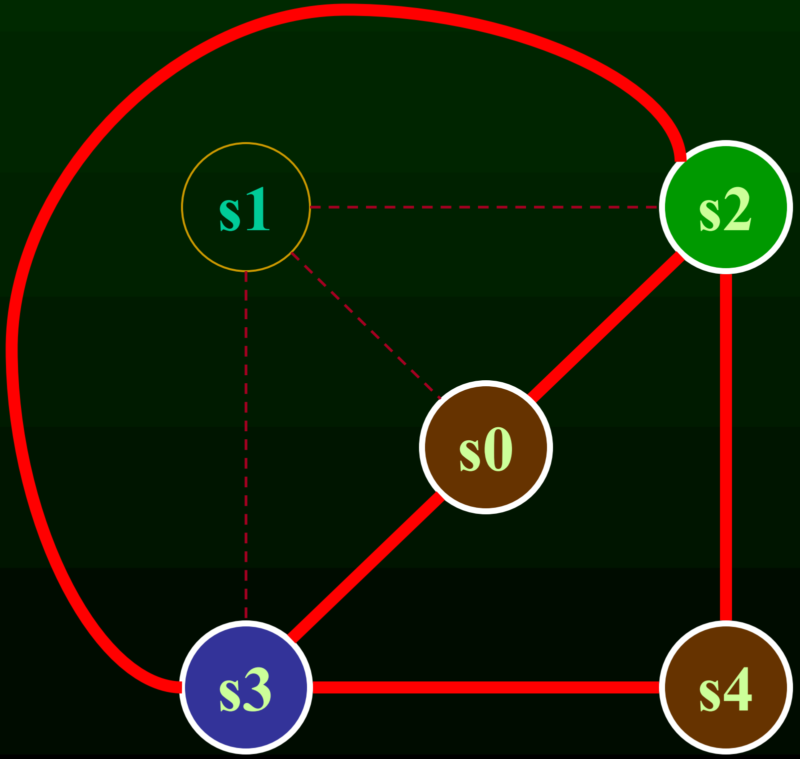
Another Coloring Example

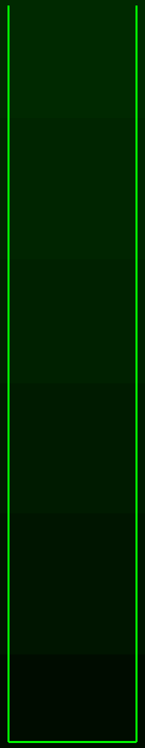
N = 3 🟦 🟩 🟫



Another Coloring Example

N = 3 🟦 🟩 🟫





What Now?

- Option 1
  - Pick a web and allocate value in memory
  - All defs go to memory, all uses come from memory
- Option 2
  - Split the web into multiple webs
- In either case, will retry the coloring

Which web to pick?

- One with interference degree >= N
- One with minimal spill cost (cost of placing value in memory rather than in register)
- What is spill cost?
  - Cost of extra load and store instructions

One Way to Compute Spill Cost

- Goal: give priority to values used in loops
- So assume loops execute 10 or 100 times
- Spill cost =
  - sum over all def sites of cost of a store instruction times 10 to the loop nesting depth power, plus
  - sum over all use sites of cost of a load instruction times 10 to the loop nesting depth power
- Choose the web with the lowest spill cost

Splitting Rather Than Spilling

- Split the web
  - Split a web into multiple webs so that there will be less interference in the interference graph making it N-colorable
  - Spill the value to memory and load it back at the points where the web is split

Splitting Heuristic

- Identify a program point where the graph is not R-colorable (point where # of webs > N)
  - Pick a web that is not used for the largest enclosing block around that point of the program
  - Split that web at the corresponding edge
  - Redo the interference graph
  - Try to re-color the graph

Cost and benefit of splitting

- Cost of splitting a node
  - Proportional to number of times splitted edge has to be crossed dynamically
  - Estimate by its loop nesting
- Benefit
  - Increase colorability of the nodes the splitted web interferes with
  - Can approximate by its degree in the interference graph
- Greedy heuristic
  - pick the live-range with the highest benefit-to-cost ration to spill

For the quiz, you should know:

- Principles of web-based register allocation
  - Webs
  - Live ranges
  - Interference graphs
  - Graph coloring
  - Heuristics for spilling and splitting

Program analysis

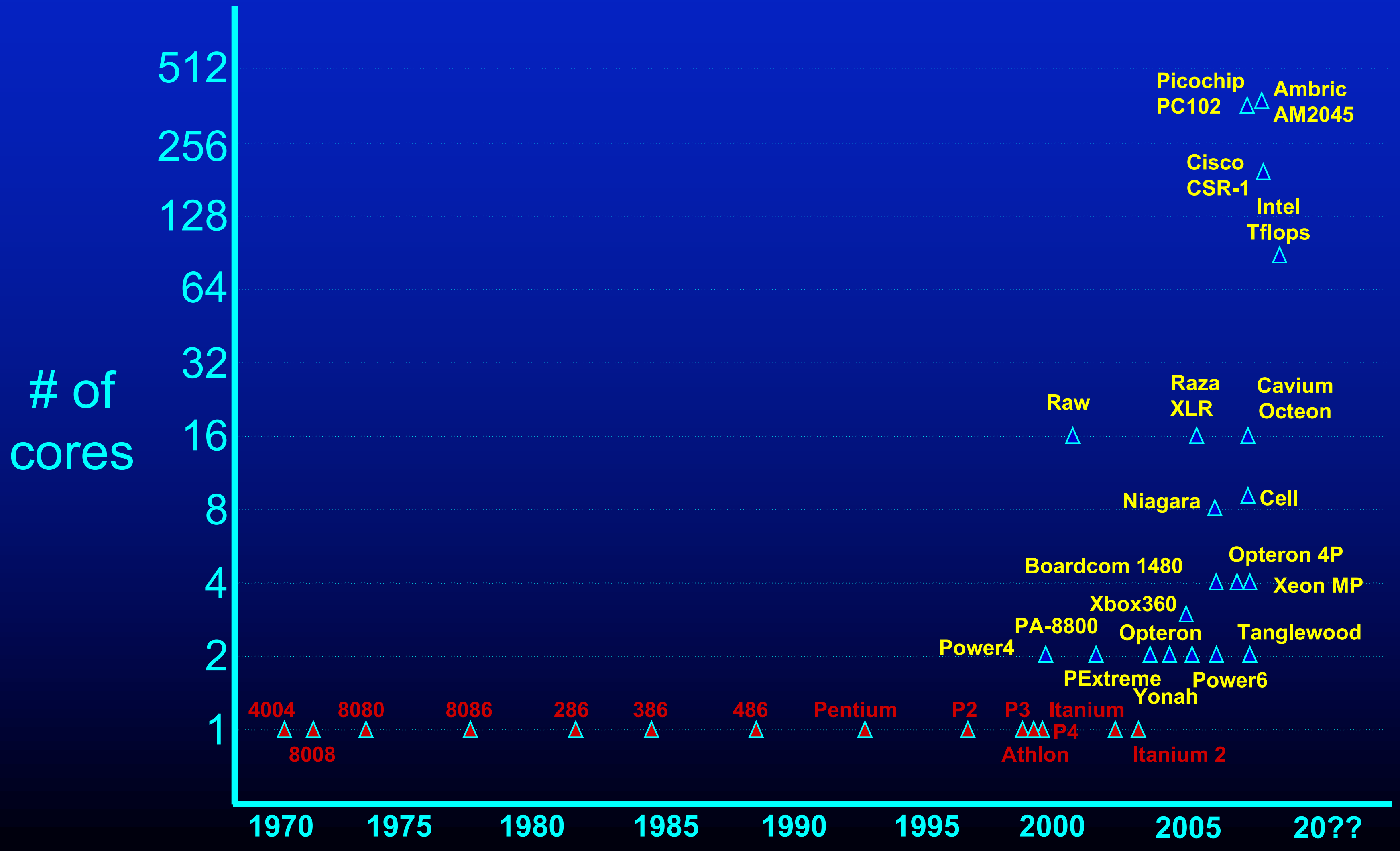
Loop optimizations

Register allocation

Parallelization ←

Foundations of dataflow

Multicores Are Here!



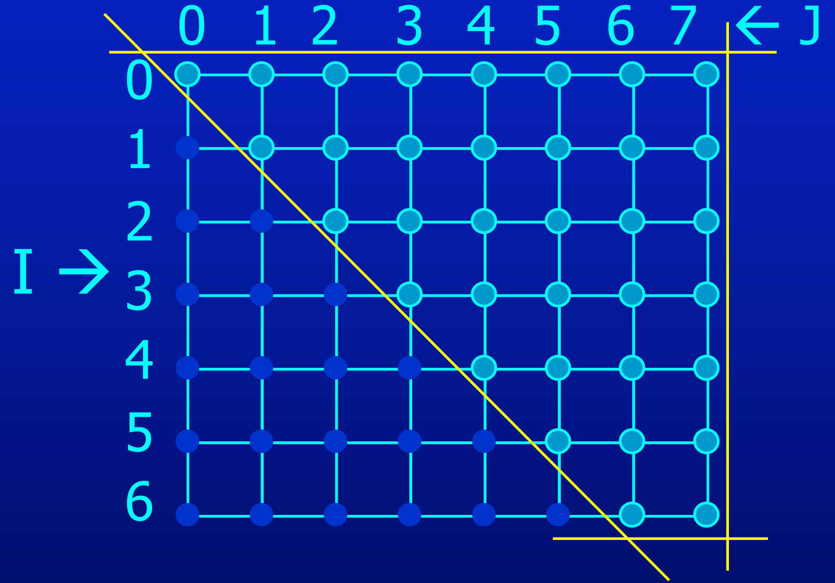
Issues with Parallelism

- Amdahl's Law
  - Any computation can be analyzed in terms of a portion that must be executed sequentially, Ts, and a portion that can be executed in parallel, Tp. Then for n processors:
  - $T(n) = Ts + Tp/n$
  - $T(\infty) = Ts$, thus maximum speedup $(Ts + Tp) / Ts$
- Load Balancing
  - The work is distributed among processors so that ***all*** processors are kept busy when parallel task is executed.
- Granularity
  - The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead).

Iteration Space

- N deep loops → N-dimensional discrete iteration space
  - Normalized loops: assume step size = 1

```
FOR I = 0 to 6
   FOR J = I to 7
```



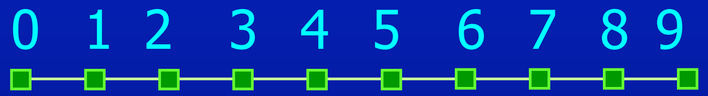
- Affine loop nest → Iteration space as a set of linear inequalities

$$0 \le I$$
$$I \le 6$$
$$I \le J$$
$$J \le 7$$

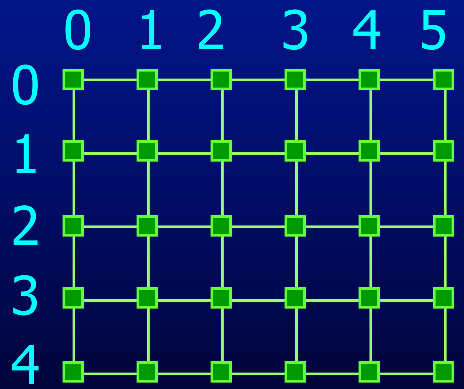
Data Space

- M dimensional arrays → M-dimensional discrete cartesian space
  - a hypercube

`Integer A(10)`



`Float B(5, 6)`



Dependences

- True dependence

```
 a  =
    =  a
```

- Anti dependence

```
    =  a
 a  =
```

- Output dependence

```
 a  =
 a  =
```

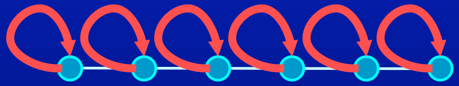
- Definition:
  Data dependence exists for a dynamic instance i and j iff
  - either i or j is a write operation
  - i and j refer to the same variable
  - i executes before j

- How about array accesses within loops?

Distance Vectors

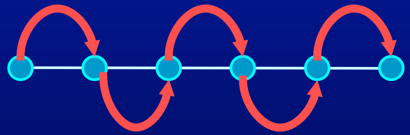
- A loop has a distance d if there exist a data dependence from iteration i to j and d = j-i

$dv = [0]$



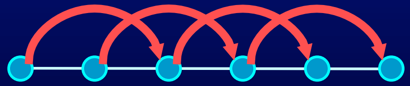
```
FOR I = 0 to 5
  A[I] = A[I] + 1
```

$dv = [1]$



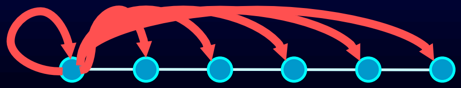
```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

$dv = [2]$



```
FOR I = 0 to 5
  A[I] = A[I+2] + 1
```

$dv = [1], \; [2] \; \ldots \; = \; [*]$

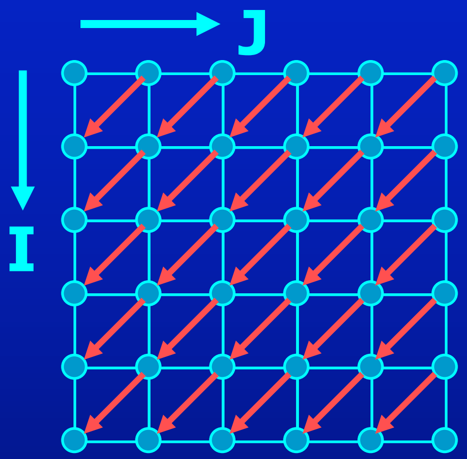


```
FOR I = 0 to 5
  A[I] = A[0] + 1
```

What is the Dependence?

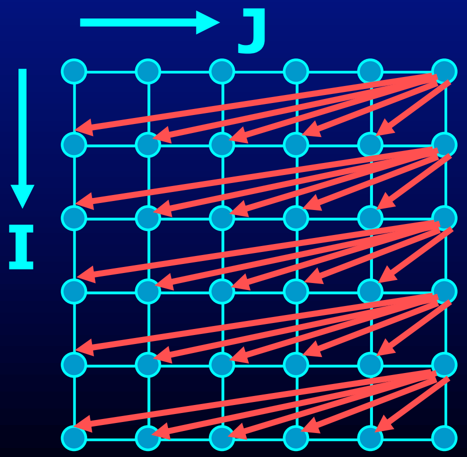
```
FOR I = 1 to n
  FOR J = 1 to n
   A[I, J] = A[I-1, J+1] + 1
```

$$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$



```
FOR I = 1 to n
  FOR J = 1 to n
   B[I] = B[I-1] + 1
```

$$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \begin{bmatrix} 1 \\ -3 \end{bmatrix}, \ldots = \begin{bmatrix} 1 \\ * \end{bmatrix}$$



Distance Vector Method

- The ith loop is parallelizable for all dependence d = $[d_1,\ldots,d_i,..d_n]$
  either

  one of $d_1,\ldots,d_{i-1}$ is > 0

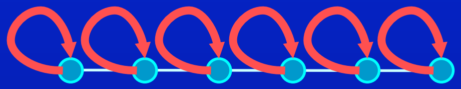
  or

  all $d_1,\ldots,d_i = 0$

Is the Loop Parallelizable?

$dv = [0]$

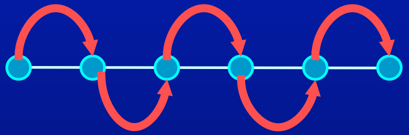
Yes



```
FOR I = 0 to 5
  A[I] = A[I] + 1
```

$dv = [1]$

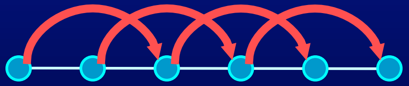
No



```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

$dv = [2]$

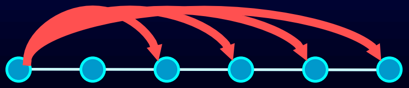
No



```
FOR I = 0 to 5
  A[I] = A[I+2] + 1
```

$dv = [*]$

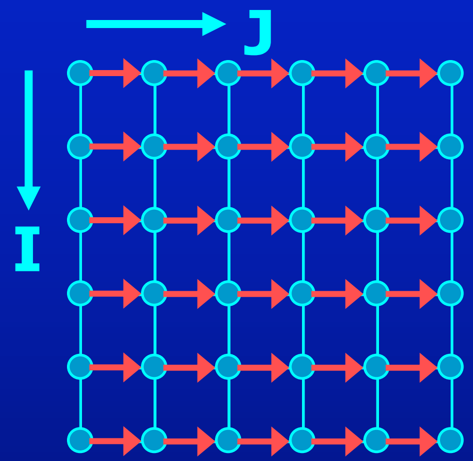
No



```
FOR I = 0 to 5
  A[I] = A[0] + 1
```

Are the Loops Parallelizable?

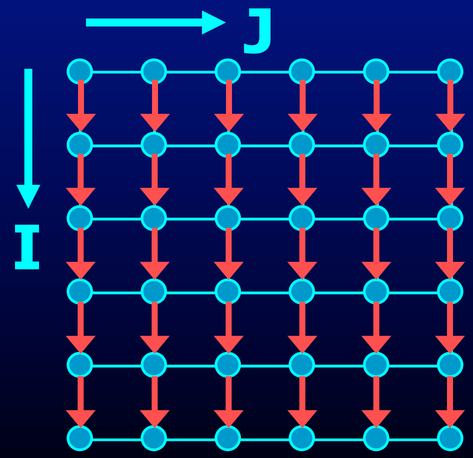
```
FOR I = 1 to n
  FOR J = 1 to n
   A[I, J] = A[I, J-1] + 1
```



$$dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Yes
No

```
FOR I = 1 to n
  FOR J = 1 to n
   A[I, J] = A[I+1, J] + 1
```



$$dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

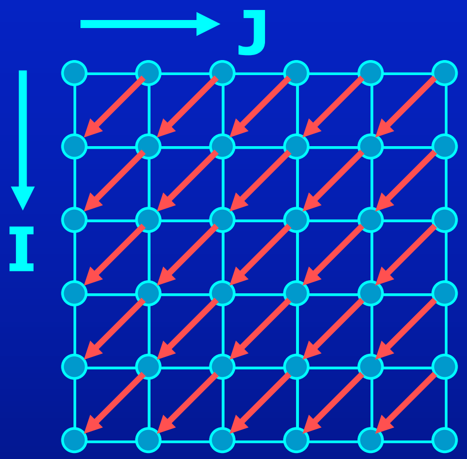
No
Yes

Are the Loops Parallelizable?

```
FOR I = 1 to n
  FOR J = 1 to n
   A[I, J] = A[I-1, J+1] + 1
```

$$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

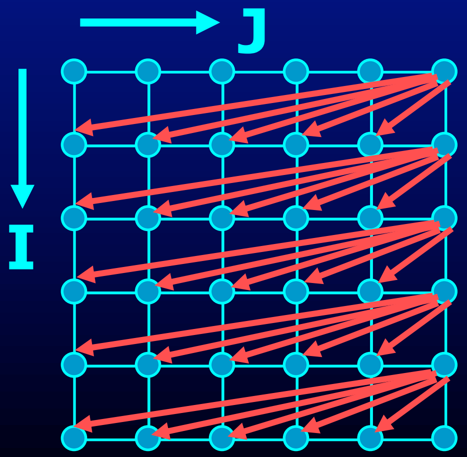
No
Yes



```
FOR I = 1 to n
  FOR J = 1 to n
   B[I] = B[I-1] + 1
```

$$dv = \begin{bmatrix} 1 \\ * \end{bmatrix}$$

No
Yes



Integer Programming Method

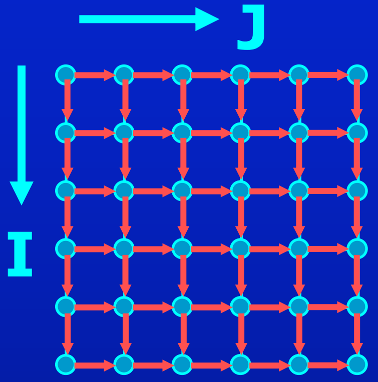
```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- Formulation
  - $\exists$ an integer vector $\bar{\imath}$ such that $\hat{A}\,\bar{\imath} \le \bar{b}$ where $\hat{A}$ is an integer matrix and $\bar{b}$ is an integer vector
- Our problem formulation for A[i] and A[i+1]
  - $\exists$ integers i_w, i_r $\quad 0 \le i_w, i_r \le 5 \quad i_w \ne i_r \quad i_w + 1 = i_r$
  - $i_w \ne i_r$ is not an affine function
    - divide into 2 problems
    - Problem 1 with $i_w < i_r$ and problem 2 with $i_r < i_w$
    - If either problem has a solution → there exists a dependence
  - How about $i_w + 1 = i_r$
    - Add two inequalities to single problem $i_w + 1 \le i_r$, and $i_r \le i_w + 1$

Loop Transformations

- A loop may not be parallel as is
- Example

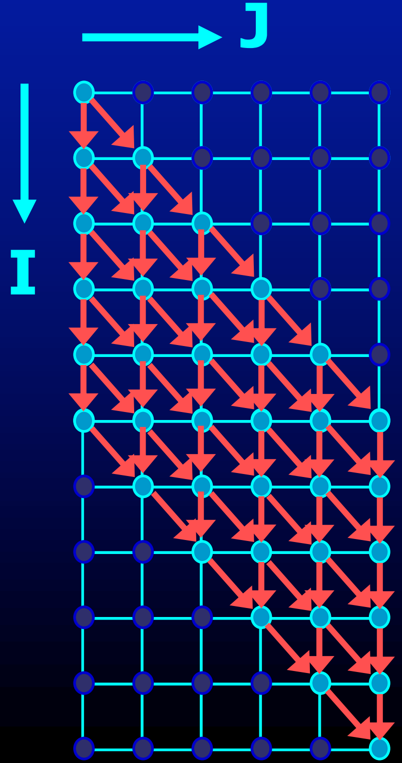
```
FOR i = 1 to N-1
  FOR j = 1 to N-1
   A[i,j] = A[i,j-1] + A[i-1,j];
```



- After loop Skewing

$$\begin{bmatrix} i_{new} \\ j_{new} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_{old} \\ j_{old} \end{bmatrix}$$

```
FOR i = 1 to 2*N-3
  FORPAR j = max(1,i-N+2) to min(i, N-1)
   A[i-j+1,j] = A[i-j+1,j-1] + A[i-j,j];
```



For the quiz, you should know:

- Why parallelize?
- Iteration spaces, data spaces
- Different types of data dependencies
- Distance vectors, and how to determine which loops can be parallelized based on distance vectors
- General concepts about integer programming

Program analysis

Loop optimizations

Register allocation

Parallelization

Foundations of dataflow ←

Dataflow analysis framework

- Dataflow analysis computes some *information* (say, of type `I`) at each statement (or basic block)
- Each statement has a transfer function $f: I \to I$
    - Given what information we have at the program point before, and what is at the statement, what information do we have atthe program point after?
- At each merge points, we combine information from the paths using a *join* function $\vee: I \times I \to I$
- Lattices are a way to formalize all this and prove that a dataflow analysis always terminates (assuming some properties of I, f and $\vee$)

Partial Orders

- Set P
- Partial order $\leq$ such that $\forall x,y,z \in P$
  - $x \leq x$ (reflexive)
  - $x \leq y$ and $y \leq x$ implies $x = y$ (asymmetric)
  - $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)
- Can use partial order to define
  - Upper and lower bounds
  - Least upper bound
  - Greatest lower bound

Upper Bounds

- If $S \subseteq P$ then
  - $x \in P$ is an upper bound of S if $\forall y \in S.\ y \leq x$
  - $x \in P$ is the least upper bound of S if
    - x is an upper bound of S, and
    - $x \leq y$ for all upper bounds y of S
  - $\vee$ - join, least upper bound, lub, supremum, sup
    - $\vee S$ is the least upper bound of S
    - $x \vee y$ is the least upper bound of $\{x,y\}$

Lower Bounds

- If $S \subseteq P$ then
  - $x \in P$ is a lower bound of S if $\forall y \in S.\ x \leq y$
  - $x \in P$ is the greatest lower bound of S if
    - x is a lower bound of S, and
    - $y \leq x$ for all lower bounds y of S
  - $\wedge$ - meet, greatest lower bound, glb, infimum, inf
    - $\wedge S$ is the greatest lower bound of S
    - $x \wedge y$ is the greatest lower bound of $\{x,y\}$

Lattices

- If $x \wedge y$ and $x \vee y$ exist for all $x,y \in P$, then P is a lattice.
- If $\wedge S$ and $\vee S$ exist for all $S \subseteq P$, then P is a complete lattice.
- All finite lattices are complete
- Example of a lattice that is not complete
  - Integers I
  - For any $x, y \in I$, $x \vee y = \max(x,y)$, $x \wedge y = \min(x,y)$
  - But $\vee I$ and $\wedge I$ do not exist
  - $I \cup \{+\infty, -\infty\}$ is a complete lattice

Top and Bottom

- Greatest element of P (if it exists) is top
- Least element of P (if it exists) is bottom ($\perp$)

For the quiz, you should know:

- Definition of posets, lattices
- Properties of lattices
  - Operations: $\leq$, $\wedge$, $\vee$
  - Lower/upper bounds, top $\top$, bottom $\bot$
  - Algebraic properties
  - Completeness

Application to Dataflow Analysis

- Dataflow information will be lattice values
  - Transfer functions operate on lattice values
  - Solution algorithm will generate increasing sequence of values at each program point
  - Ascending chain condition will ensure termination
- Will use $\vee$ to combine values at control-flow join points

Transfer Functions

- Transfer function f: P$\to$P for each node in control flow graph
- f models effect of the node on the program information

Transfer Functions

Each dataflow analysis problem has a set F of transfer functions $f: P \to P$

- Identity function $i \in F$
- F must be closed under composition: $\forall f, g \in F$. the function $h = \lambda x.f(g(x)) \in F$
- Each $f \in F$ must be monotone: $x \leq y$ implies $f(x) \leq f(y)$
- Sometimes all $f \in F$ are distributive: $f(x \vee y) = f(x) \vee f(y)$
- Distributivity implies monotonicity

Sign Analysis Example

- Sign analysis - compute sign of each variable v
- Base Lattice: P = flat lattice on {-,0,+}

$$\begin{tikzcd}
 & \text{TOP} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
- \arrow[dr, dash] & 0 \arrow[d, dash] & + \arrow[dl, dash] \\
 & \text{BOT} &
\end{tikzcd}$$

Actual Lattice

- Actual lattice records a sign for each variable
  - Example element: [$a \rightarrow +$, $b \rightarrow 0$, $c \rightarrow -$]
- Function lattice
  - Elements of lattice are functions (maps) from variables to base sign lattice
  - For function lattice elements f_1 and $_{f2}$
  - $f_1 \leq f_2$ if $\forall$ v in V. $f_1(v) \leq f_2(v)$

Interpretation of Lattice Values

- If value of v in lattice is:
  - BOT: no information about sign of v
  - -: variable v is negative
  - 0: variable v is 0
  - +: variable v is positive
  - TOP: v may be positive, negative, or zero
- What is abstraction function AF?
  - $AF([v_1,\ldots,v_n]) = [sign(v_1), \ldots, sign(v_n)]$
  - Where $sign(v) = 0$ if $v = 0$, $+$ if $v > 0$, $-$ if $v < 0$

Operation ⊗ on Lattice

⊗	BOT	-	0	+	TOP
BOT	BOT	BOT	0	BOT	BOT
-	BOT	+	0	-	TOP
0	0	0	0	0	0
+	BOT	-	0	+	TOP
TOP	BOT	TOP	0	TOP	TOP

Transfer Functions

- If n of the form $v = c$
  - $f_n(x) = x[v \to +]$ if c is positive
  - $f_n(x) = x[v \to 0]$ if c is 0
  - $f_n(x) = x[v \to -]$ if c is negative
- If n of the form $v_1 = v_2 * v_3$
  - $f_n(x) = x[v_1 \to x[v_2] \otimes x[v_3]]$
- I = TOP (if variables not initialized)
- $I = [v_1 \to 0, \ldots, v_n \to 0]$ (if variables initialized to 0)

Example

$$\begin{tikzcd}
 & a = 1 \arrow[dl, "{[a\to +]}"'] \arrow[dr, "{[a\to +]}"] \arrow[d, phantom, "{[a\to +]}"] & \\
b = -1 \arrow[dr, "{[a\to +,\ b\to -]}"'] & {} & b = 1 \arrow[dl, "{[a\to +,\ b\to +]}"] \\
 & {} \arrow[d, "{[a\to +,\ b\to \mathrm{TOP}]}"'] & \\
 & c = a*b &
\end{tikzcd}$$

$[a\to +,\ b\to \mathrm{TOP}, c \to \mathrm{TOP}]$

Imprecision In Example

Abstraction Imprecision:

[a→1] abstracted as [a→+]

$$\begin{tikzcd}
 & a = 1 \arrow[dl, "{[a\to+]}"'] \arrow[dr, "{[a\to+]}"] & \\
b = -1 \arrow[dr, "{[a\to+,\ b\to-]}"'] & & b = 1 \arrow[dl, "{[a\to+,\ b\to+]}"] \\
 & {} \arrow[d, "{[a\to+,\ b\to\mathrm{TOP}]}"'] & \\
 & c = a*b &
\end{tikzcd}$$

Control Flow Imprecision:

[b→TOP] summarizes results of all executions. In any execution state s, AF(s)[b]≠TOP

Meet Over Paths Solution

- What solution would be ideal for a forward dataflow analysis problem?
- Consider a path $p = n_0, n_1, \ldots, n_k, n$ to a node n (note that for all i $n_i \in \mathrm{pred}(n_{i+1})$)
- The solution must take this path into account:

  $f_p (\bot) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(\bot)) \ldots)) \leq in_n$
- So the solution must have the property that $\vee\{f_p (\bot)\ .\ p \text{ is a path to } n\} \leq in_n$

  and ideally

  $\vee\{f_p (\bot)\ .\ p \text{ is a path to } n\} = in_n$

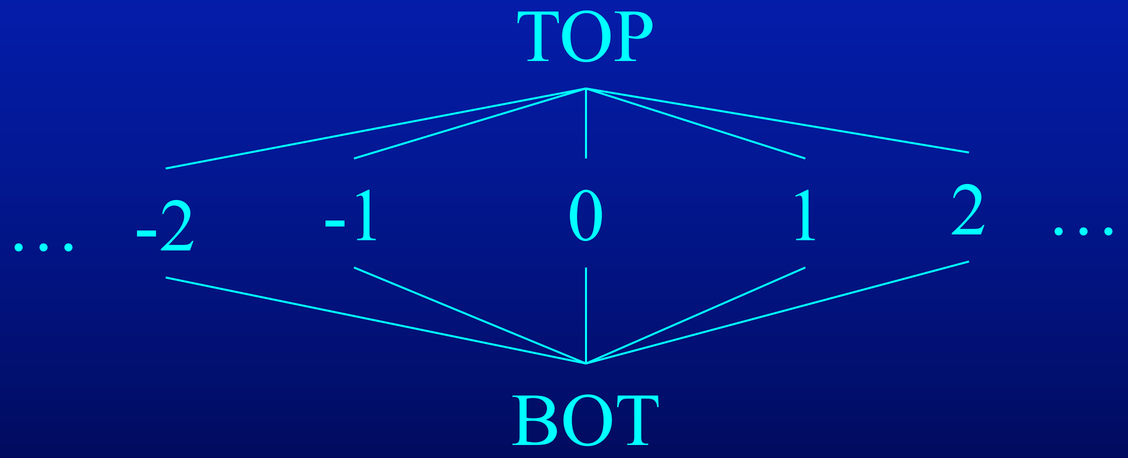
Distributivity

- Distributivity preserves precision
- If framework is distributive, then worklist algorithm produces the meet over paths solution
  - For all n:

$$\vee\{f_p(\perp) \,.\, p \text{ is a path to } n\} = in_n$$

Lack of Distributivity Example

- Constant Calculator
- Flat Lattice on Integers



- Actual lattice records a value for each variable
  - Example element: [a→3, b→2, c→5]

Transfer Functions

- If n of the form $v = c$
  - $f_n(x) = x[v \to c]$
- If n of the form $v_1 = v_2+v_3$
  - $f_n(x) = x[v_1 \to x[v_2] + x[v_3]]$
- Lack of distributivity
  - Consider transfer function f for $c = a + b$
  - $f([a\to 3, b\to 2]) \vee f([a\to 2, b\to 3]) = [a\to TOP, b\to TOP, c\to 5]$
  - $f([a\to 3, b\to 2]\vee[a\to 2, b\to 3]) = f([a\to TOP, b\to TOP]) = [a\to TOP, b\to TOP, c\to TOP]$

Lack of Distributivity Anomaly

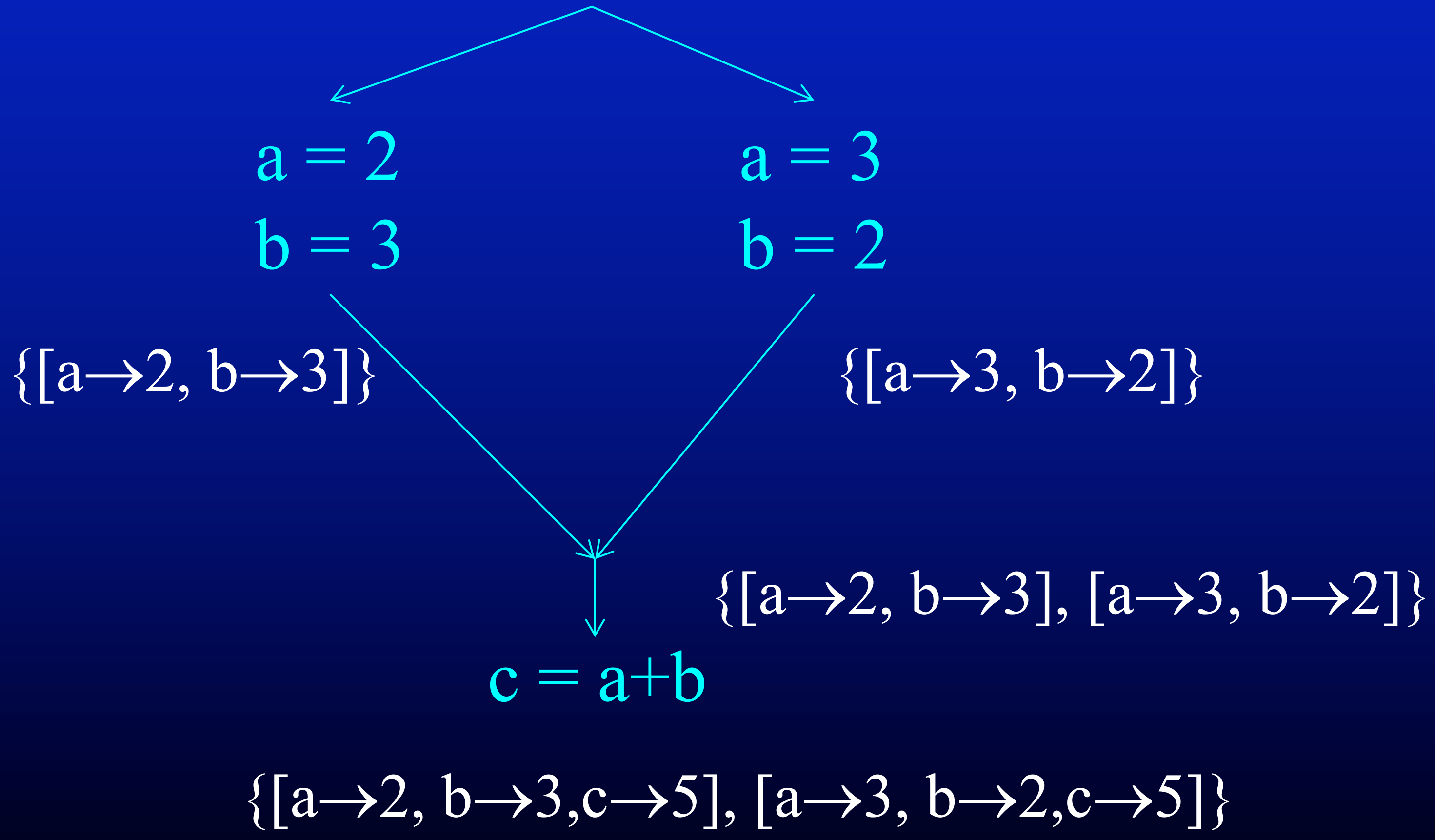
$$\begin{tikzcd}
 & {} \arrow[dl] \arrow[dr] & \\
{\begin{array}{l} a = 2 \\ b = 3 \end{array}} \arrow[dr, "{[a\to 2,\ b\to 3]}"'] & & {\begin{array}{l} a = 3 \\ b = 2 \end{array}} \arrow[dl, "{[a\to 3,\ b\to 2]}"] \\
 & {} \arrow[d, "{[a\to \mathrm{TOP},\ b\to \mathrm{TOP}]}"'] & \\
 & c = a+b & \\
 & {[a\to \mathrm{TOP},\ b\to \mathrm{TOP},\ c\to \mathrm{TOP}]} &
\end{tikzcd}$$

Lack of Distributivity Imprecision:
[a→TOP, b→TOP, c→5] more precise

What is the meet over all paths solution?

How to Make Analysis Distributive

- Keep combinations of values on different paths



Summary

- Formal dataflow analysis framework
  - Lattices, partial orders, least upper bound, greatest lower bound, ascending chains
  - Transfer functions, joins and splits
  - Dataflow equations and fixed point solutions
- Connection with program
  - Abstraction function AF: $S \rightarrow P$
  - For any state s and program point n, $AF(s) \leq in_n$
  - Meet over all paths solutions, distributivity

For the quiz, you should know:

- How to give transfer functions for simple lattices and nodes
- Abstraction functions
- Meet over paths solution
- Causes of imprecision