PROGRAMMING LANGUAGES LABORATORY
Universidade Federal de Minas Gerais - Department of Computer Science

WORKLIST ALGORITHMS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from "Principles of Program Analysis", Chapter 6, by Niesen *et al*, and from Miachel Schwartzbach's "Lecture notes in Static Analysis", Chapter 6, First Section.

Solving Constraints

- The dataflow analyses that we have seen determine constraint systems.
- A constraint system is a high-level way to see a problem.
- There are efficient algorithms to solve constraint systems.
  - Thus, a description of the problem to be solved also gives a solution to this problem.
- The objective of this class is to see how some constraint systems can be solved in practice.
  - Using efficient and provably correct algorithms.

$IN[x_1] = \{\}$

$IN[x_2] = OUT[x_1] \cup OUT[x_3]$

$IN[x_3] = OUT[x_2]$

$IN[x_4] = OUT[x_1] \cup OUT[x_5]$

$IN[x_5] = OUT[x_4]$

$IN[x_6] = OUT[x_2] \cup OUT[x_4]$

$OUT[x_1] = IN[x_1]$

$OUT[x_2] = IN[x_2]$

$OUT[x_3] = (IN[x_3] \backslash \{3,5,6\}) \cup \{3\}$

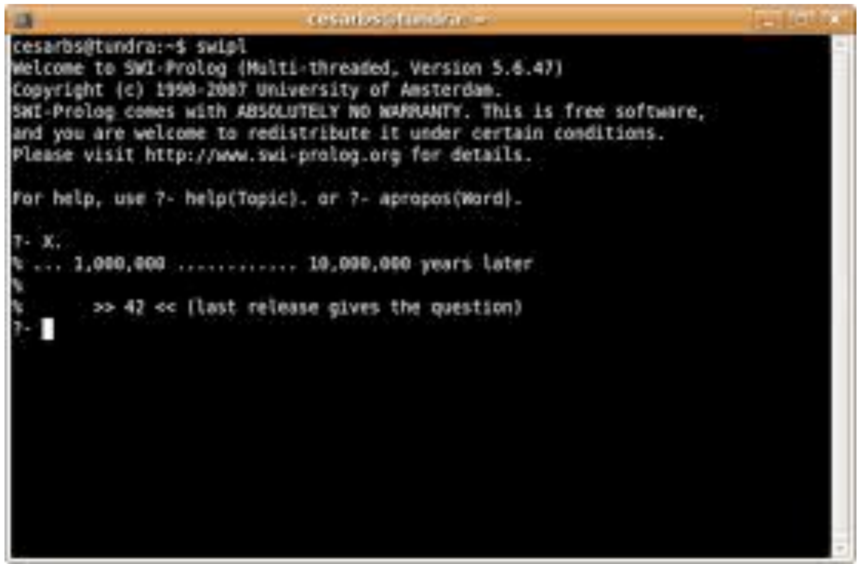
$OUT[x_4] = IN[x_4]$

$OUT[x_5] = (IN[x_5] \backslash \{3,5,6\}) \cup \{5\}$

$OUT[x_6] = (IN[x_6] \backslash \{3,5,6\}) \cup \{6\}$

Where do you think these constraints come from?

SOLVING CONSTRAINTS IN PROLOG



DCC 888

The Running Example

$$
\begin{array}{l}
\text{if } b_1 \\
\quad \text{then} \\
\qquad \text{while } b_2 \text{ do } x = a_1 \\
\quad \text{else} \\
\qquad \text{while } b_3 \text{ do } x = a_2 \\
x = a_3
\end{array}
$$

How many basic blocks do we have in this program?

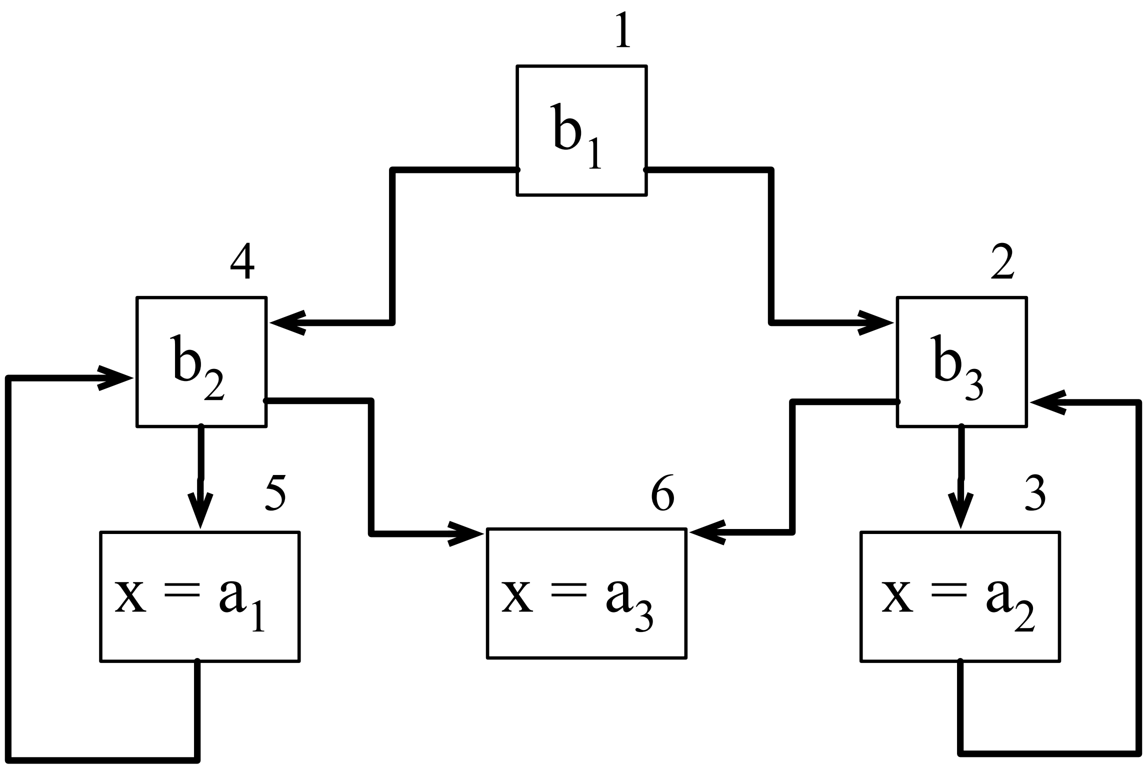
The Running Example

if $[b_1]^1$
  then
    while $[b_2]^2$ do $[x = a_1]^3$
  else
    while $[b_3]^4$ do $[x = a_2]^5$
$[x = a_3]^6$

Can you draw the CFG of this program?

We will be doing data-flow analysis; hence, we will be talking about program points all the time, as these analyses bind information to program points. We shall use these labels to represent the program points. So, we can talk about the information at the label *x*, for instance.

The Running Example

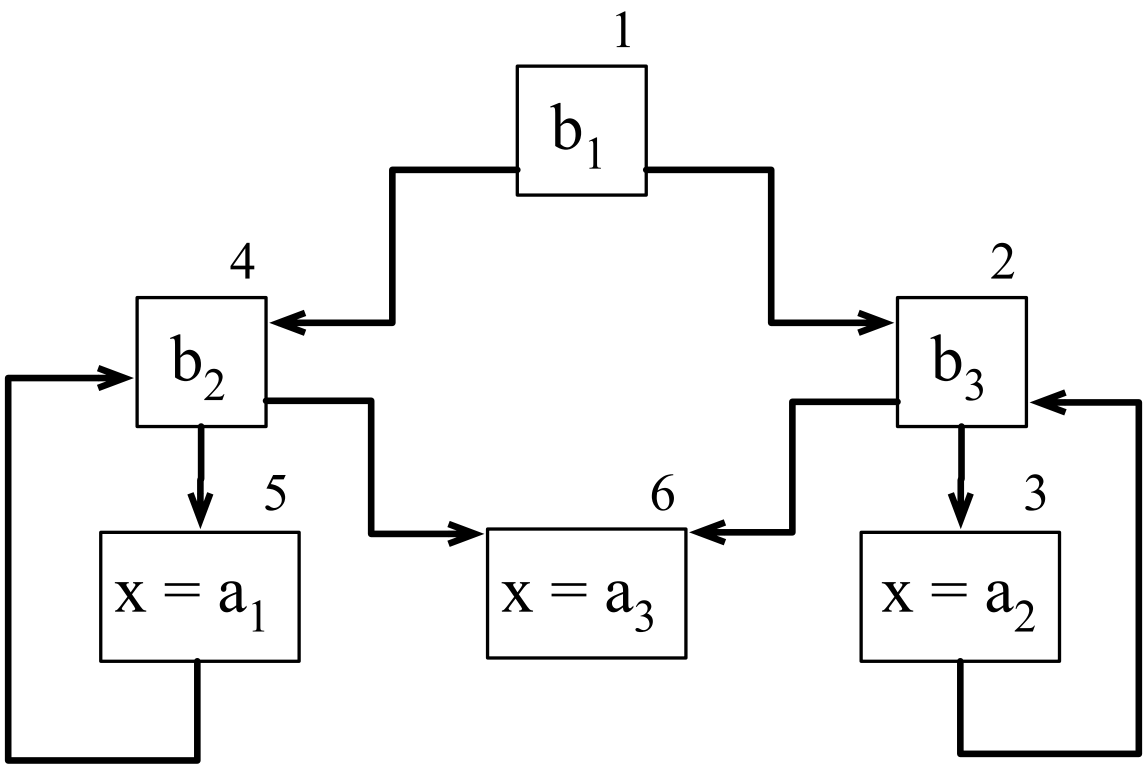


Can you produce the IN and OUT equations of reaching definitions for this example?

Remember: the definition of a variable *v*, at a program point p_v, reaches another program point *p* if:

1) There exists a path in the CFG from p_v to *p*.
2) The variable *v* is not redefined along this path.

The Running Example



$IN[x_1] = \{\}$

$IN[x_2] = OUT[x_1] \cup OUT[x_3]$

$IN[x_3] = OUT[x_2]$

$IN[x_4] = OUT[x_1] \cup OUT[x_5]$

$IN[x_5] = OUT[x_4]$

$IN[x_6] = OUT[x_2] \cup OUT[x_4]$

$OUT[x_1] = IN[x_1]$

$OUT[x_2] = IN[x_2]$

$OUT[x_3] = (IN[x_3] \backslash \{3,5,6\}) \cup \{3\}$

$OUT[x_4] = IN[x_4]$

$OUT[x_5] = (IN[x_5] \backslash \{3,5,6\}) \cup \{5\}$

$OUT[x_6] = (IN[x_6] \backslash \{3,5,6\}) \cup \{6\}$

Now, write these equations in Prolog, and find a solution for them.

Reaching Definitions in Prolog

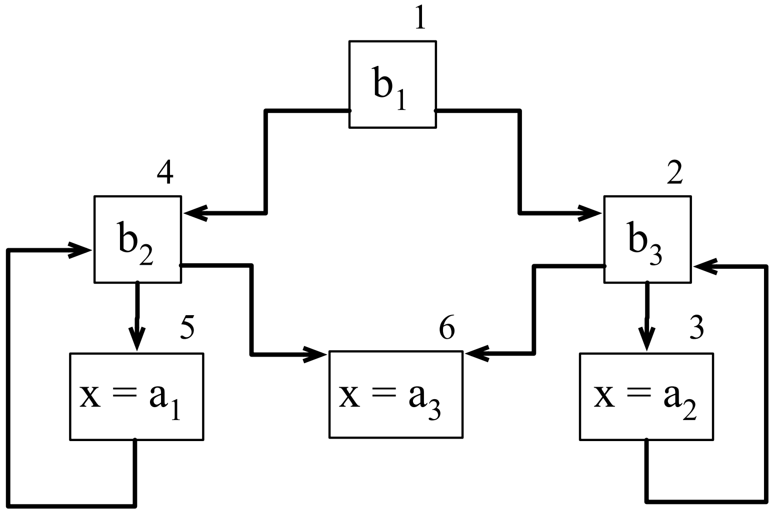
```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT), !.
```

How is the implementation of these predicates, union and diff?

Reaching Definitions in Prolog

How could we test this solution?

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT),
  !.
```




```
union([], S, S).
union([H|T], S, [H|SU]) :- union(T, S, SU).

diff([], _, []).
diff([H|T], S, SD) :- member(H, S), diff(T, S, SD).
diff([H|T], S, [H|SD]) :- \+member(H, S), diff(T, S, SD).
```

Reaching Definitions in Prolog

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT), !.
```

Can you devise a general way to build systems of equations for a given dataflow analysis?

```
?- consult(rd).
% rd compiled 0.00 sec, 4,272 bytes
true.

?- solution([[], [3], [3], [5], [5], [3, 5], [], [3], [3], [5], [5], [6]]).
true ;
```

Systems of Equations

- We can think of a given dataflow problem as a system of equations.
  - If we have *n* variables, then we have *n* equations.

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT), !.
```

Can you write this Prolog program as a set of equations, like F above?

System of Equations

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT), !.
```

$$(in_1, in_2, in_3, in_4, in_5, in_6, out_1, out_2, out_3, out_4, out_5, out_6) =$$
$$(\{\}, out_1 \cup out_3, out_2, out_1 \cup out_5, out_4, out_2 \cup out_4, in_1, in_2,$$
$$in_3 \setminus \{3, 5, 6\} \cup out_3, in_4, in_5 \setminus \{3, 5, 6\} \cup out_5, in_6 \setminus \{3, 5, 6\} \cup out_6\ )$$

Conservative Solutions

- A system of equations gives us a solution that is *conservative*.
  – It may be larger than the best, most precise solution!

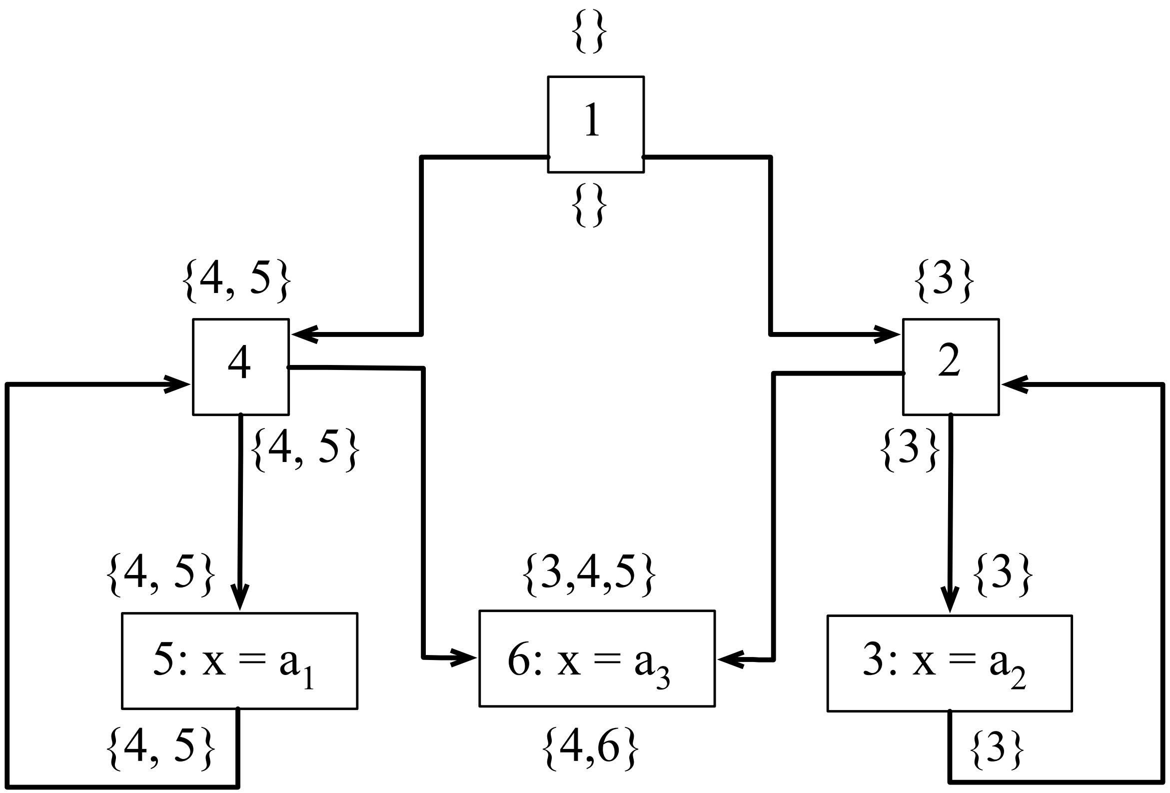
```
?- solution([[], [3], [3], [5], [5], [3, 5], [], [3], [3], [5], [5], [6]]).
true ;

?- solution([[], [3], [3], [4, 5], [4, 5], [3, 4, 5], [], [3], [3], [4, 5],
  [4, 5], [4, 6]]).
true ;
```

- In the example above, 4 is not in the most precise solution, but it is not wrong if we put it in some of the *IN* and *OUT* sets.

Why is this conservative solution still correct?

Conservative Solutions

We are assuming that there exists a definition of x at x_4. That is ok, as long as we want to know *all the definitions* that reach a basic block. For instance, in constant propagation we want to know if all the definitions that reach a block represent the same constant. If we have more definitions reaching a basic block, we might end up optimizing the program a bit less, as some of them might be different constants.



$IN[x_1] = \{\}$

$IN[x_2] = OUT[x_1] \cup OUT[x_3]$

$IN[x_3] = OUT[x_2]$

$IN[x_4] = OUT[x_1] \cup OUT[x_5]$

$IN[x_5] = OUT[x_4]$

$IN[x_6] = OUT[x_2] \cup OUT[x_4]$

$OUT[x_1] = IN[x_1]$

$OUT[x_2] = IN[x_2]$

$OUT[x_3] = (IN[x_3] \backslash \{3,5,6\}) \cup \{3\}$

$OUT[x_4] = IN[x_4]$

$OUT[x_5] = (IN[x_5] \backslash \{3,5,6\}) \cup \{5\}$

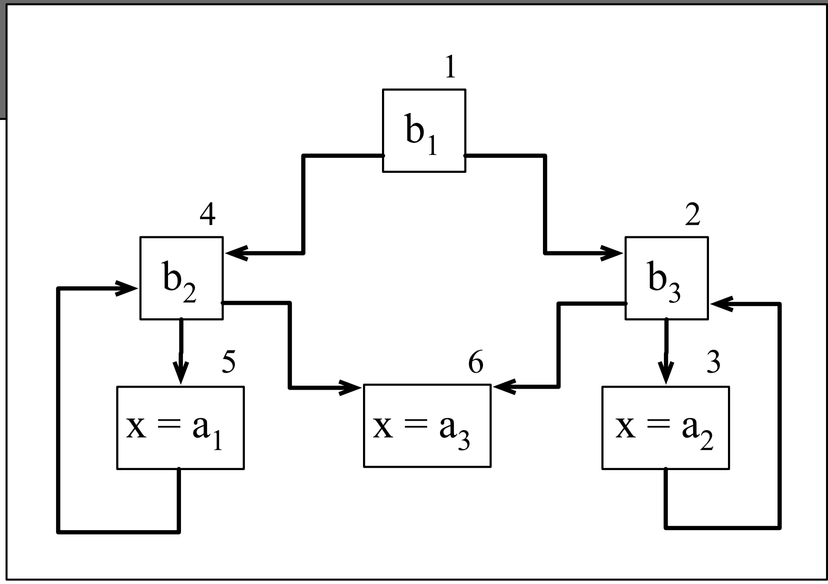
$OUT[x_6] = (IN[x_6] \backslash \{3,5,6\}) \cup \{6\}$

Conservative Solutions

- Yet, solutions must be consistent!

```
?- solution([[], [3], [3], [5], [5], [3, 5], [], [3], [3], [5], [5], [6]]).
true ;

?- solution([[], [3], [3], [4, 5], [4, 5], [3, 4, 5], [], [3], [3], [4, 5],
[4, 5], [4, 6]]).
true ;

?- solution([[], [3], [3], [4, 5], [4, 5], [3, 4, 5], [], [3], [3], [4, 5],
[4, 5], []]).
false.
```

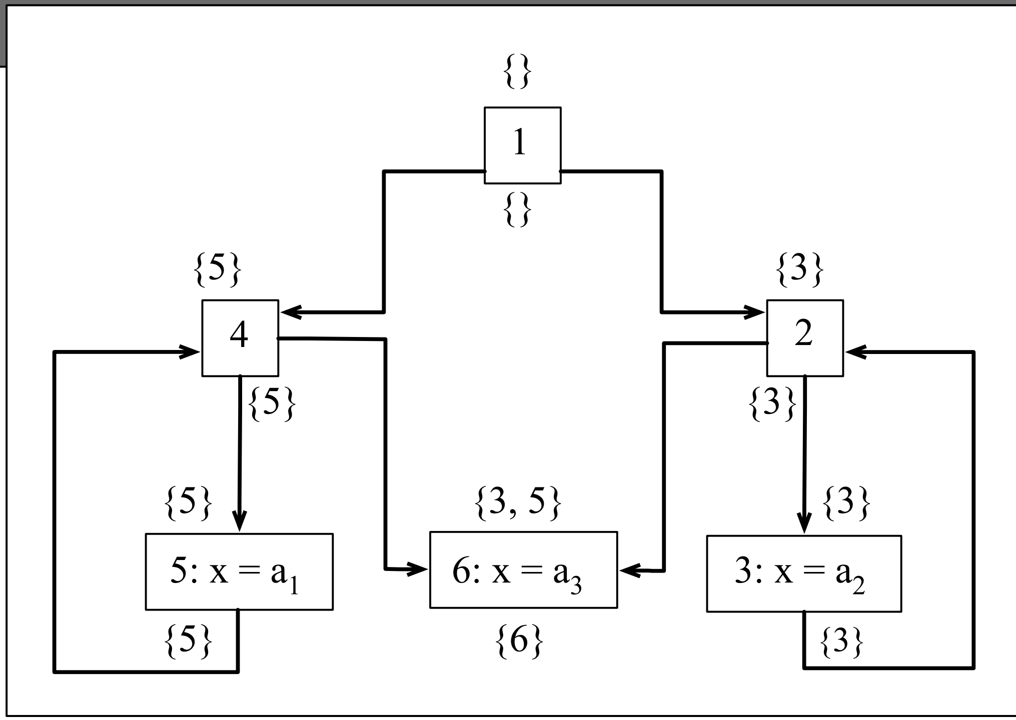
Why is the third query invalid at OUT_X6?



False Negatives

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT])
```

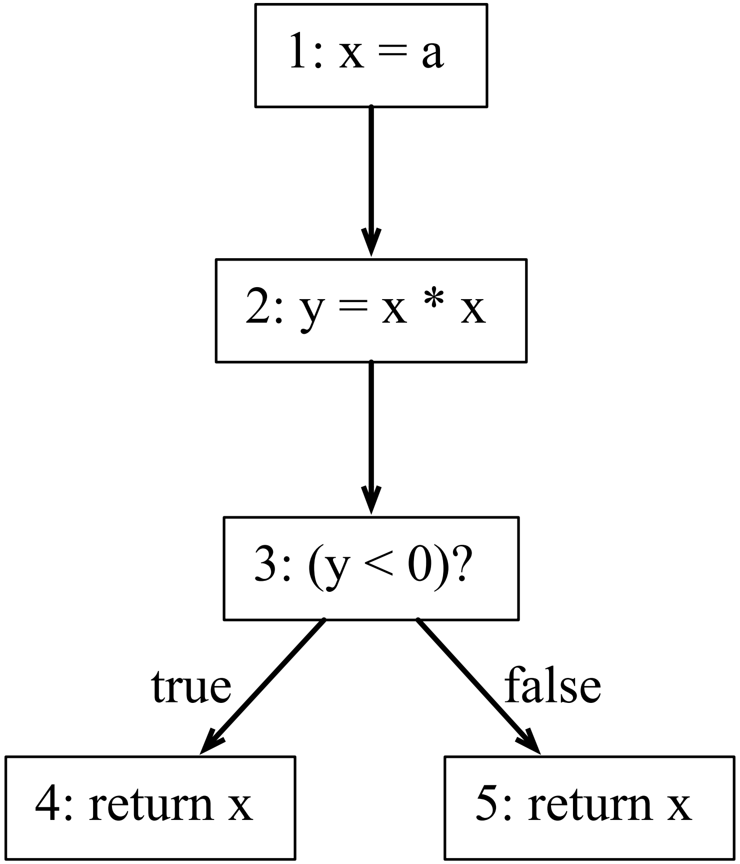
```
?- solution([[], [3], [3], [4, 5], [4, 5], [3, 4, 5], [], [3], [3], [4, 5],
  [4, 5], []]).
  false.
```



Variable x is defined at program point 6. Therefore, it must be present in the OUT set of 6, or the solution will be incorrect, given that OUT[6] = $in_6 \setminus \{3, 5, 6\} \cup out_6$.

Static and Dynamic Solutions

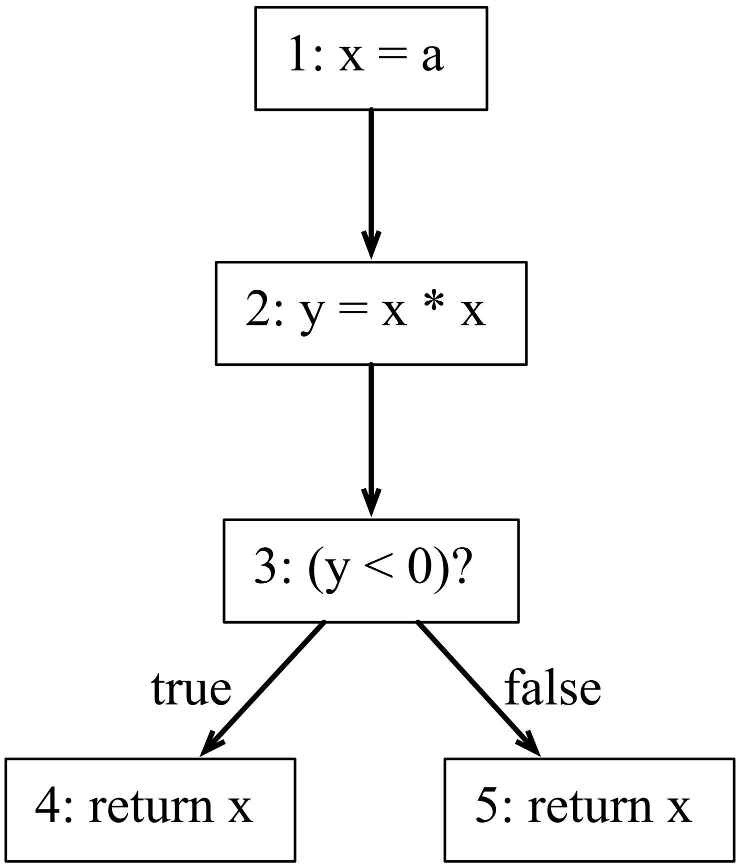
- Static Analyses can, sometimes, provide solutions that will never happen in an actual run of the program.



1) Which definitions would our static analysis tells us that reach block 4?
2) Once we run the program, is block 4 ever reached by any definitions?

Static and Dynamic Solutions

- Static Analyses can, sometimes, provide solutions that will never happen in an actual run of the program.



If, in an actual run of the program, a definition D reaches a block B, then the static analysis must say so, otherwise we will have a *false negative*. A false negative means that the analysis is **wrong**.

However, if the static analysis says that a definition D reaches a block B, and it never does it in practice, then this is is a *false positive*. False positives mean that the analysis is **imprecise**, but not wrong.

Finding Solutions with Prolog

- We can also use Prolog as a way to find solutions.
  - In the previous example, we had only used Prolog to check if a solution was valid or not.

```
?- solution([X1_IN, X2_IN, X3_IN, X4_IN, X5_IN, X6_IN, X1_OUT, X2_OUT,
X3_OUT, X4_OUT, [5], X6_OUT]).

X1_IN = [],
X2_IN = [3],
X3_IN = [3],
X4_IN = [5],
X5_IN = [5],
X6_IN = [3, 5],
X1_OUT = [],
X2_OUT = [3],
X3_OUT = [3],
X4_OUT = [5],
X6_OUT = [6].
```

Infinity Queries

- But we must be careful: some of our queries may not terminate!

```
?- solution([X1_IN, X2_IN, X3_IN, X4_IN, X5_IN, X6_IN, X1_OUT, X2_OUT,
X3_OUT, X4_OUT, X5_OUT, X6_OUT]).

Action (h for help) ? abort
% Execution Aborted
```

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT).
```

What is the problem with this query? Why does it not terminate?

Infinity Queries

- But we must be careful: some of our queries may not terminate!

```
?- solution([X1_IN, X2_IN, X3_IN, X4_IN, X5_IN, X6_IN, X1_OUT, X2_OUT,
X3_OUT, X4_OUT, X5_OUT, X6_OUT]).

Action (h for help) ? abort
% Execution Aborted
```

- Prolog does an exhaustive search system.
  - It tries lists with every possible number of elements.
    - Unless these sizes are fixed beforehand.
  - Thus, we are trying solutions, e.g., `X1_IN`, `X2_IN`, etc, with sizes 1, 2, etc.

Infinity Queries

- We could fix the size of the list `X5_OUT`, and our query would terminate:

```
?- length(X5_OUT, 1), solution([X1_IN, X2_IN, X3_IN, X4_IN, X5_IN, X6_IN,
X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]).
X5_OUT = [5],
X1_IN = [],
X2_IN = [3],
X3_IN = [3],
X4_IN = [5],
X5_IN = [5],
X6_IN = [3, 5],
X1_OUT = [],
X2_OUT = [3],
X3_OUT = [3],
X4_OUT = [5],
X6_OUT = [6].
```

How could we implement and algorithm to solve these constraint systems? Our algorithm must:

1. always terminate
2. find good solutions

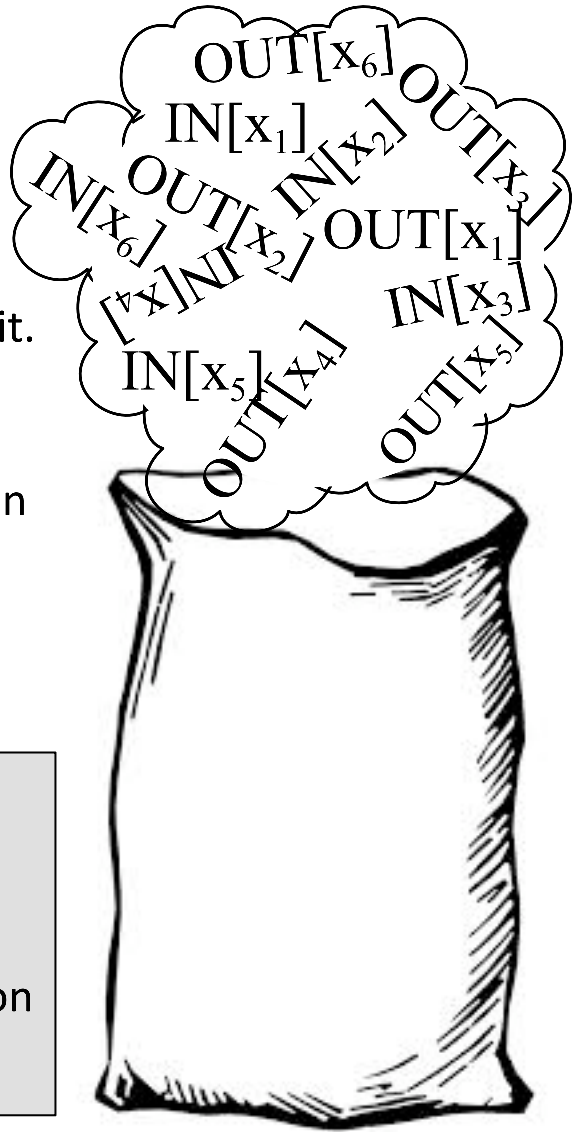
ITERATIVE WORKLIST SOLVERS

DCC 888

Chaotic Iterations: Constraints in a Bag

Imagine the following way to solve these constraints:

1) Put all of them in a bag.
2) Shake the bag breathlessly
3) Take one constraint C out of it.
4) Solve C.
5) If nothing has changed:
    1) If there are constraints in the bag, go to step (3)
    2) else you are done!
6) else go to step 1.



$IN[x_1] = \{\}$

$IN[x_2] = OUT[x_1] \cup OUT[x_3]$

$IN[x_3] = OUT[x_2]$

$IN[x_4] = OUT[x_1] \cup OUT[x_5]$

$IN[x_5] = OUT[x_4]$

$IN[x_6] = OUT[x_2] \cup OUT[x_4]$

$OUT[x_1] = IN[x_1]$

$OUT[x_2] = IN[x_2]$

$OUT[x_3] = (IN[x_3] \backslash \{3,5,6\}) \cup \{3\}$

$OUT[x_4] = IN[x_4]$

$OUT[x_5] = (IN[x_5] \backslash \{3,5,6\}) \cup \{5\}$

$OUT[x_6] = (IN[x_6] \backslash \{3,5,6\}) \cup \{6\}$

1) Does this thing terminate?

2) How many loops would have an imperative implementation of this algorithm?

Example: Reaching Definitions

for each i ∈ {1, 2, 3, 4, 5, 6}

  $IN[x_i] = \{\}$

  $OUT[x_i] = \{\}$

repeat

  for each i ∈ {1, 2, 3, 4, 5, 6}

    $IN'[x_i] = IN[x_i];$

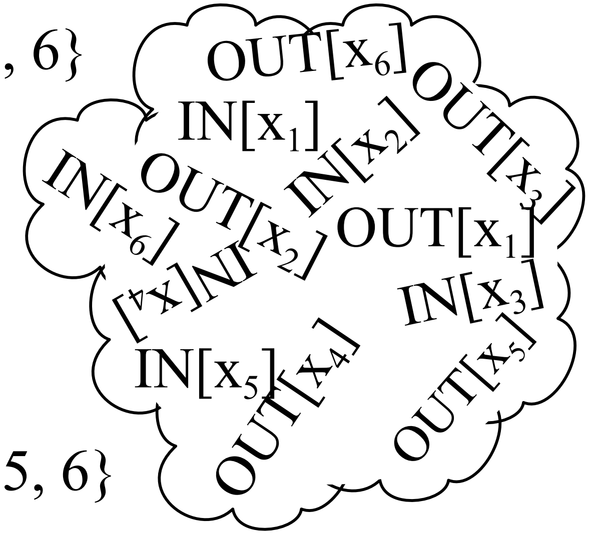
    $OUT'[x_i] = OUT[x_i];$

    $OUT[x_i] = def(x_i) \cup (IN[x_i] \setminus kill(x_i));$

    $IN[x_i] = \cup_{s \in pred[xi]} OUT[s];$

until ∀i ∈ {1, 2, 3, 4, 5, 6}

  $IN'[x_i] = in[x_i]$ **and** $OUT'[x_i] = OUT[x_i]$



$IN[x_1] = \{\}$

$IN[x_2] = OUT[x_1] \cup OUT[x_3]$

$IN[x_3] = OUT[x_2]$

$IN[x_4] = OUT[x_1] \cup OUT[x_5]$

$IN[x_5] = OUT[x_4]$

$IN[x_6] = OUT[x_2] \cup OUT[x_4]$

$OUT[x_1] = IN[x_1]$

$OUT[x_2] = IN[x_2]$

$OUT[x_3] = (IN[x_3] \setminus \{3,5,6\}) \cup \{3\}$

$OUT[x_4] = IN[x_4]$

$OUT[x_5] = (IN[x_5] \setminus \{3,5,6\}) \cup \{5\}$

$OUT[x_6] = (IN[x_6] \setminus \{3,5,6\}) \cup \{6\}$

1) What is the kill set of each variable x_i?

2) What is the complexity of this algorithm?

Chaotic Iterations

$$x_1 = \perp, x_2 = \perp, \ldots, x_n = \perp$$

do

$$t_1 = x_1 \; ; \ldots ; \; t_n = x_n$$

$$x_1 = F_1(x_1, \ldots, x_n)$$

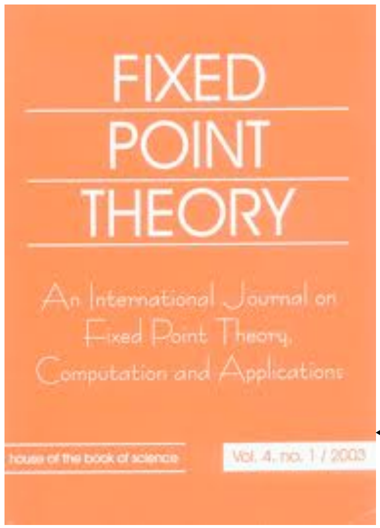
$$\ldots$$

$$x_n = F_n(x_1, \ldots, x_n)$$

while $(x_1 \neq t_1$ or $\ldots$ or $x_n \neq t_n)$

- We keep solving equations iteratively, until we reach a fixed point.
- Even though we do not enforce any particular order to solve the equations, a fixed point is guaranteed to exist, at least to the **reaching definitions** problem.
- In the next class, we shall see why that is the case. But, for this, we need a bit more of **theory**.

Could you give an intuition on why reaching defs always stop?



Complexity Analysis

for each $i \in \{1, 2, 3, 4, 5, 6\}$

$IN[x_i] = \{\}$

$OUT[x_i] = \{\}$

repeat

for each $i \in \{1, 2, 3, 4, 5, 6\}$

$IN'[x_i] = IN[x_i];$

$OUT'[x_i] = OUT[x_i];$

$OUT[x_i] = def(x_i) \cup (IN[x_i] \setminus kill(x_i));$

$IN[x_i] = \cup_{s \in pred[xi]} OUT[s];$

until $\forall i \in \{1, 2, 3, 4, 5, 6\}$

$IN'[x_i] = in[x_i]$ **and** $OUT'[x_i] = OUT[x_i]$

1) Assuming that we have N variables in the program, and O(N) instructions, how many times can the repeat loop iterate?
2) How many times can the inner for loop iterate?
3) What is the complexity of each set union?
4) How many set unions we might have, per iteration of the inner for loop?
5) So, what is the final complexity?

Complexity Analysis

1. Each set might have up to N elements. We have N sets. Thus, we might have N^2 changes that might cause an iteration in the repeat loop.

2. The inner for loop will iterate once for each pair of IN/OUT sets; hence, we will have N iterations.

3. We can implement each union operation to be O(N)

4. We might have up to O(N) predecessors per block.

5. Thus, in the end, we might end up with an algorithm that is $O(N^5)$. *Almost nothing that really matters in computer science has such a high polynomial complexity*.

1) Assuming that we have N variables in the program, and O(N) instructions, how many times can the repeat loop iterate?

2) How many times can the inner for loop iterate?

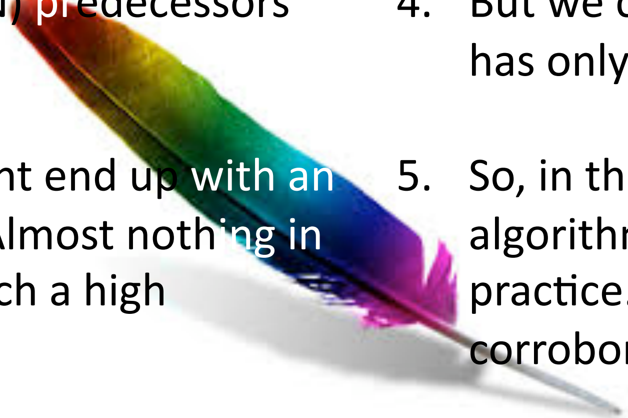
3) What is the complexity of each set union?

4) How many set unions we might have, per iteration of the inner for loop?

5) So, what is the final complexity?

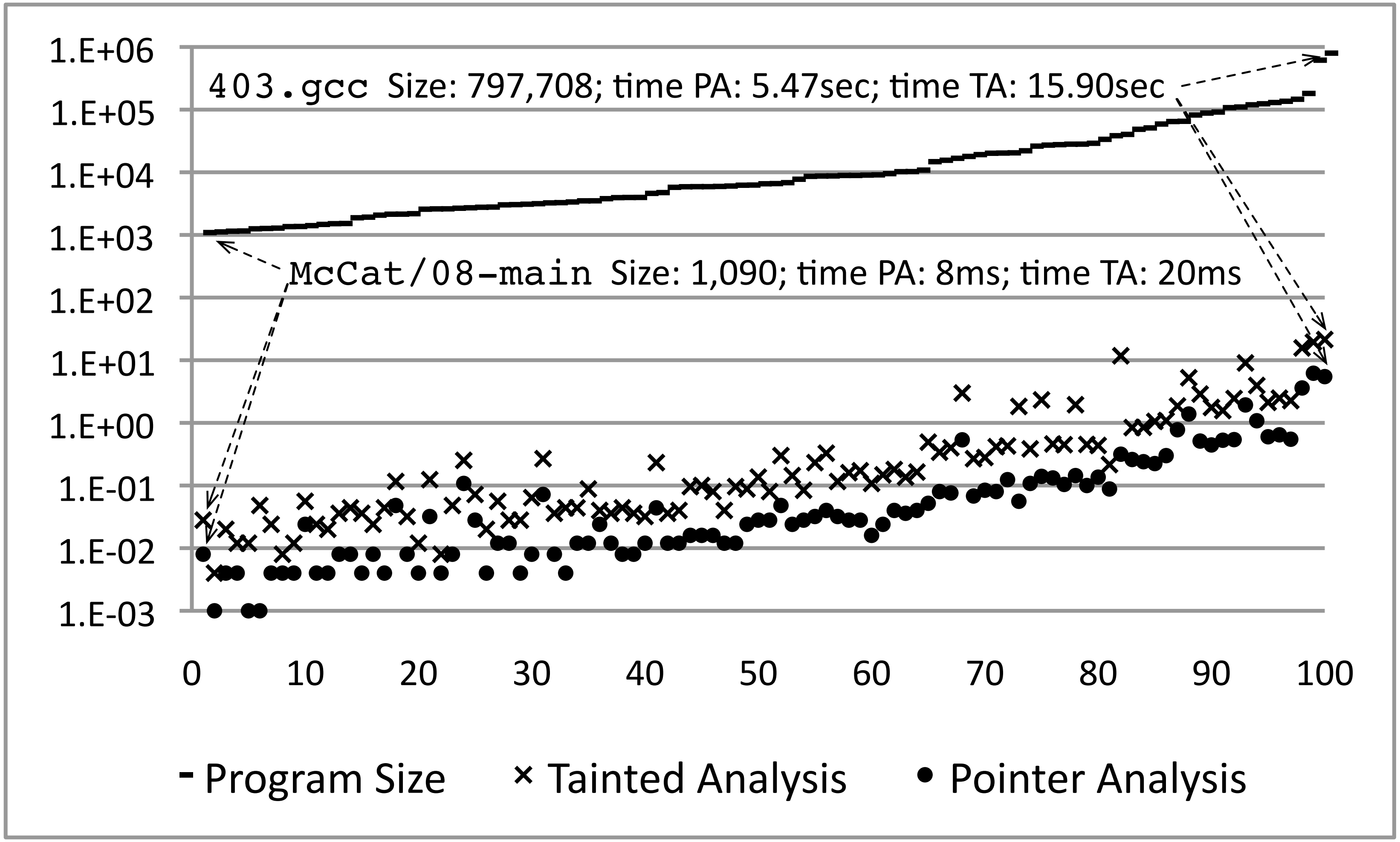
Complexity Analysis

1. Each set might have up to N elements. We have N sets. Thus, we might have N^2 changes that might cause an iteration in the repeat loop.
2. The inner for loop will iterate once for each pair of IN/OUT sets; hence, we will have N iterations.
3. We can implement each union operation to be O(N)
4. We might have up to O(N) predecessors per block.
5. Thus, in the end, we might end up with an algorithm that is $O(N^5)$. Almost nothing in computer science has such a high polynomial complexity.



1. Yet, if we order the nodes properly, the repeat loop will iterate two or three times, i.e., O(1)
2. Ok, **here** we cannot do much: we will have to go over each program point at least once.
3. Usually the sets contain a fairly small number of elements. We can assume that we can do union in O(1) time.
4. But we can assume that each branch has only two successors, e.g., O(1).
5. So, in the end, many data-flow algorithms will run in linear time in practice. Many experiments corroborate this assumption.

Empirical Evidence of Linear Complexity



Tainted analysis and pointer analysis are two well-known static analyses that can be solved via chaotic iterations on a constraint system, and they seem to run in linear time in practice. This chart shows data for some fairly large programs.

Speeding up Chaotic Iterations

$x_1 = \perp, x_2 = \perp, \ldots, x_n = \perp$

do

$t_1 = x_1 ; \ldots ; t_n = x_n$

$x_1 = F_1(x_1, \ldots, x_n)$

$\ldots$

$x_n = F_n(x_1, \ldots, x_n)$

while $(x_1 \neq t_1$ or $\ldots$ or $x_n \neq t_n)$

- Not enforcing an order may be bad
  - We may solve equations that do not add anything new to our constraint system.
- We can improve the algorithm by solving equations in some particular ordering

Which ordering could we rely upon?

Dependencies

- A constraint system is made of variables. Some variables depend on others.

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT).
```

Which dependences do we have in this system?

The Constraint Dependence Graph

- The dependence graph of constraints has one vertex for each constraint variable X.
- The graph contains an edge from Y to X if the constraint that generates variable X uses variable Y.

$IN[x_1] = \{\}$

$IN[x_2] = OUT[x_1] \cup OUT[x_3]$

$IN[x_3] = OUT[x_2]$

$IN[x_4] = OUT[x_1] \cup OUT[x_5]$

$IN[x_5] = OUT[x_4]$

$IN[x_6] = OUT[x_2] \cup OUT[x_4]$

$OUT[x_1] = IN[x_1]$

$OUT[x_2] = IN[x_2]$

$OUT[x_3] = (IN[x_3] \setminus \{3,5,6\}) \cup \{3\}$

$OUT[x_4] = IN[x_4]$

$OUT[x_5] = (IN[x_5] \setminus \{3,5,6\}) \cup \{5\}$

$OUT[x_6] = (IN[x_6] \setminus \{3,5,6\}) \cup \{6\}$

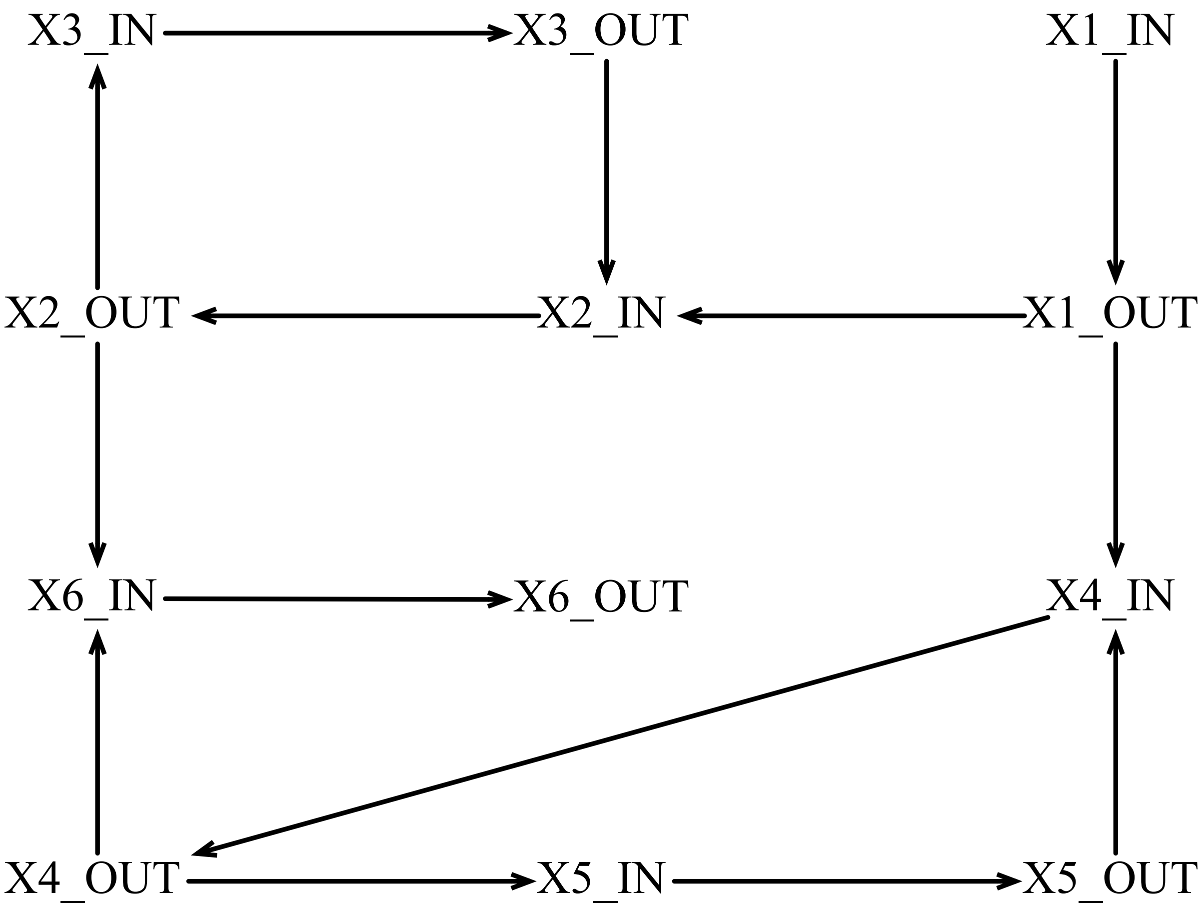
How is the dependence graph of this system?

1) How many nodes do we have?

2) How many edges?

The Dependence Graph of Constraint Variables

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA),
  union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB),
  union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC),
  union(XC, [6], X6_OUT).
```



How could we use this graph of dependencies to speed up our algorithm?

Worklists

$x_1 = \bot, x_2 = \bot, \ldots, x_n = \bot$

$w = [v_1, \ldots, v_n]$

while ($w \neq []$)

  $v_i = \text{extract}(w)$

  $y = F_i(x_1, \ldots, x_n)$

  if $y \neq x_i$

    for $v \in \text{dep}(v_i)$

      $w = \text{insert}(w, v)$

    $x_i = y$

- We can improve chaotic iterations with a worklist.
- The worklist – w in our case – contains the variables that still need to be processed.
- Once the worklist is empty, we are done.

How could we guarantee that the worklist will be always empty after some iterations?

Extract and Insert

$x_1 = \perp, x_2 = \perp, \ldots, x_n = \perp$

$w = [v_1, \ldots, v_n]$

while ($w \neq []$)

  $v_i = \text{extract}(w)$

  $y = F_i(x_1, \ldots, x_n)$

  if $y \neq x_i$

    for $v \in \text{dep}(v_i)$

      $w = \text{insert}(w, v)$

    $x_i = y$

- The functions *extract* and *insert* are abstract.
- The implementation of these functions is important.
  - We can still speed up our resolution system with good orderings of insertion and extraction.

What would be a simple implementation of insert and extract?

Simplifying the Running Example

- In the rest of this class, we shall use tables to show the abstract state of the constraint variables.
  - But 12 variables are too many.
  - We only need the IN sets for the reaching definition analysis.

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT).
```

How could we then remove the OUT sets?

Simplifying the Running Example

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN,
          X1_OUT, X2_OUT, X3_OUT, X4_OUT, X5_OUT, X6_OUT]) :-
  X1_IN = [],
  union(X1_OUT, X3_OUT, X2_IN),
  X3_IN = X2_OUT,
  union(X1_OUT, X5_OUT, X4_IN),
  X5_IN = X4_OUT,
  union(X2_OUT, X4_OUT, X6_IN),
  X1_OUT = X1_IN,
  X2_OUT = X2_IN,
  diff(X3_IN, [3, 5, 6], XA), union(XA, [3], X3_OUT),
  X4_OUT = X4_IN,
  diff(X5_IN, [3, 5, 6], XB), union(XB, [5], X5_OUT),
  diff(X6_IN, [3, 5, 6], XC), union(XC, [6], X6_OUT).
```

```
solution([X1_IN,  X2_IN,  X3_IN,  X4_IN,  X5_IN,  X6_IN]) :-
  X1_IN = [],
  diff(X3_IN, [3, 5, 6], XA), union(XA, X1_IN, XB), union(XB, [3], X2_IN),
  X3_IN = X2_IN,
  diff(X5_IN, [3, 5, 6], XC), union(XC, X1_IN, XD), union(XD, [5], X4_IN),
  X5_IN = X4_IN,
  union(X2_IN, X5_IN, X6_IN), !.
```

Well, given that everything now is Xi_IN, lets henceforth just call the constraint variables x_i

Last-in, First-out extraction/insertion

w	x1	x2	x3	x4	x5	x6
[x1, x2, x3, x4, x5, x6]	[]	[]	[]	[]	[]	[]
[x2, x4, x2, x3, x4, x5, x6]	[]	[]	[]	[]	[]	[]
[x3, x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[]	[]	[]	[]
[x2, x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[]
[x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[]
[x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[3]
[x5, x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[3]
[x4, x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[]	[3]
[x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3]
[x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[]	[]	[3]	[3]	[5]	[5]	[3,5]

$x1 = \{\}$
$x2 = x1 \cup (x3 \setminus \{3, 5, 6\}) \cup \{3\}$
$x3 = x2$
$x4 = x1 \cup (x5 \setminus \{3, 5, 6\}) \cup \{5\}$
$x5 = x4$
$x6 = x2 \cup x4$

Last-in, First-out extraction/insertion

w	x1	x2	x3	x4	x5	x6
[x1, x2, x3, x4, x5, x6]	[]	[]	[]	[]	[]	[]
[x2, x4, x2, x3, x4, x5, x6]	[]	[]	[]	[]	[]	[]
[x3, x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[]	[]	[]	[]
[x2, x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[]
[x6, x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[]
[x4, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[3]
[x5, x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[]	[]	[3]
[x4, x6, x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[]	[3]
[**x6**, x2, x3, x4, x5, **x6**]	[]	[3]	[3]	[5]	[5]	[3]
[x2, x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x3, x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x4, x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x5, x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[x6]	[]	[3]	[3]	[5]	[5]	[3,5]
[]	[]	[3]	[3]	[5]	[5]	[3,5]

Notice that we do not bother about verifying if a node is already in the worklist. Why is this approach still sensible?

In Search of a Better Ordering

$x1 = \{\}$

$x2 = x1 \cup (x3 \setminus \{3, 5, 6\}) \cup \{3\}$

$x3 = x2$

$x4 = x1 \cup (x5 \setminus \{3, 5, 6\}) \cup \{5\}$

$x5 = x4$

$x6 = x2 \cup x4$

- It is still possible to improve on the LIFO insertion/extraction.
- To find a better ordering, we can take a look into the constraint dependence graph.

How is the dependence graph of this constraint system?

In Search of a Better Ordering

x1 = {}

x2 = x1 ∪ (x3 \ {3, 5, 6}) ∪ {3}

x3 = x2

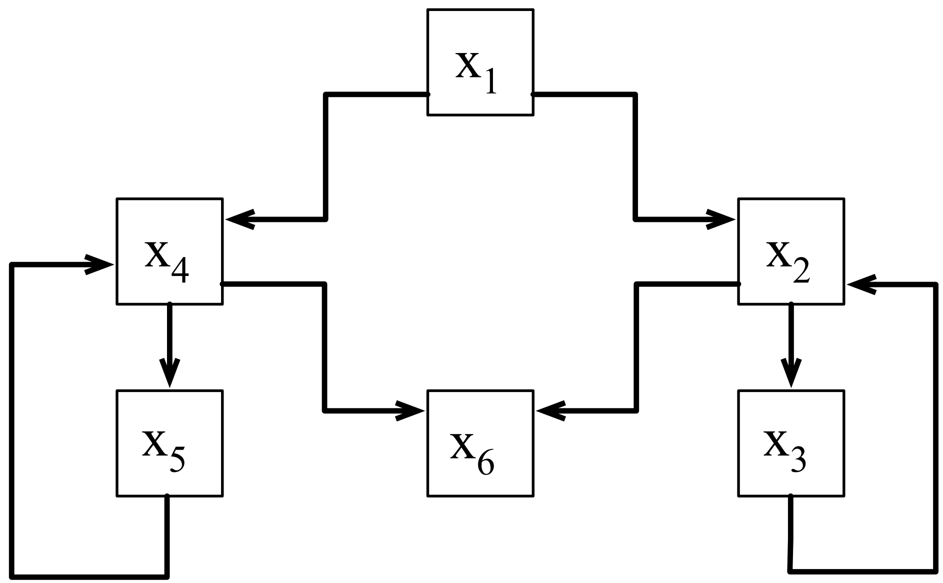
x4 = x1 ∪ (x5 \ {3, 5, 6}) ∪ {5}

x5 = x4

x6 = x2 ∪ x4

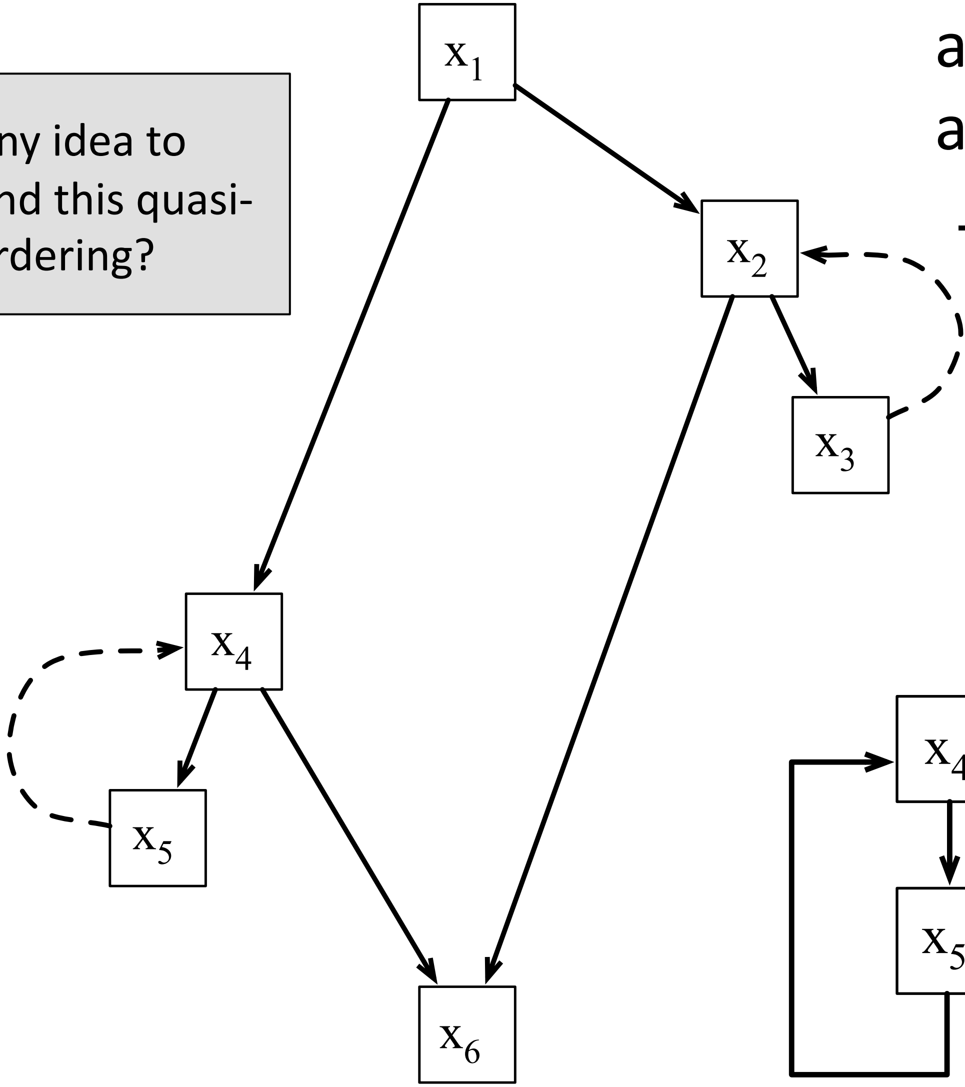
- It is still possible to improve on the LIFO insertion/extraction.
- To find a better ordering, we can take a look into the constraint dependence graph.

Which ordering would be good for this constraint graph?

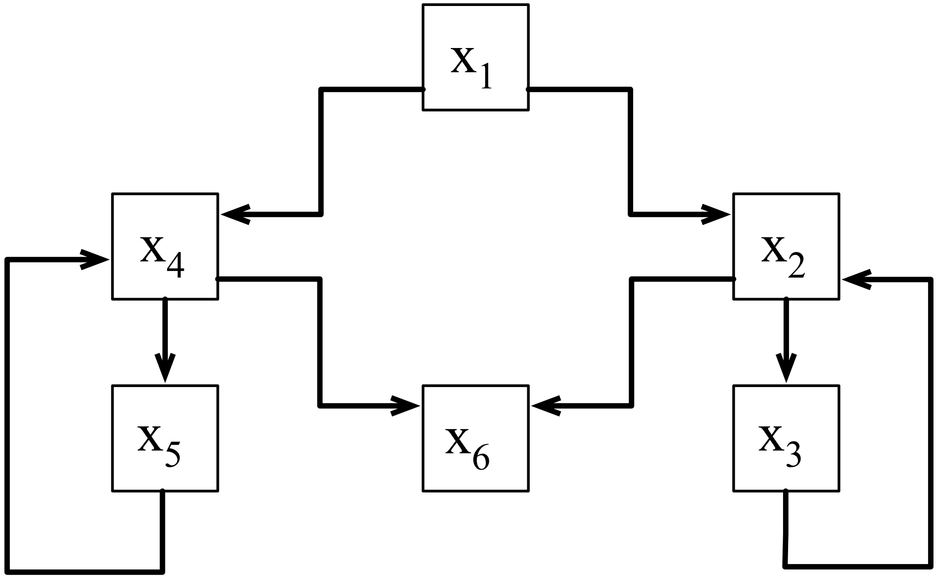


In Search of a Better Ordering

- The dependence graph is not acyclic; hence, it does not have a topological ordering.
  – But we may try to find a "quasi-ordering"



Any idea to find this quasi-ordering?



Reverse Postorder

```
i = number of nodes
mark all nodes as unvisited
while exist unvisited node h
    DFS(h)

DFS(n):
    mark n as visited
    for each edge (n, n')
        if unvisited n', DFS(n')
    rPostorder[n] = i
    i = i - 1
```

- Reverse postorder is the node ordering that we obtain after a depth-first search in the graph.
- If we visit node n' from node n, then we say that the edge (n, n') belongs into the *depth-first spanning tree* (DFST)
- The ordering in *rPostorder* topologically sorts the DFST

Reverse Postorder

i = number of nodes

mark all nodes as unvisited

while exist unvisited node h

    DFS(h)

DFS(n):

    mark n as visited

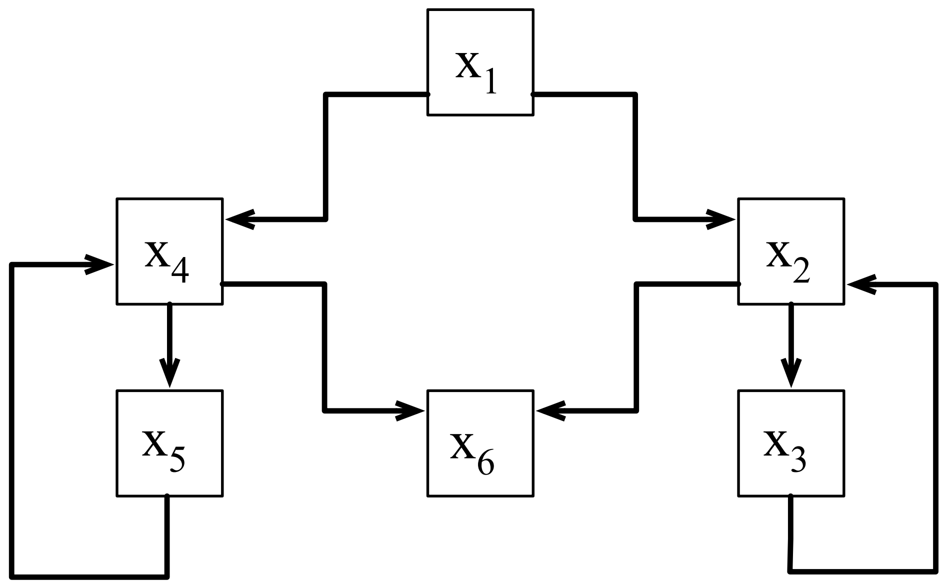
    for each edge (n, n')

        if unvisited n', DFS(n')

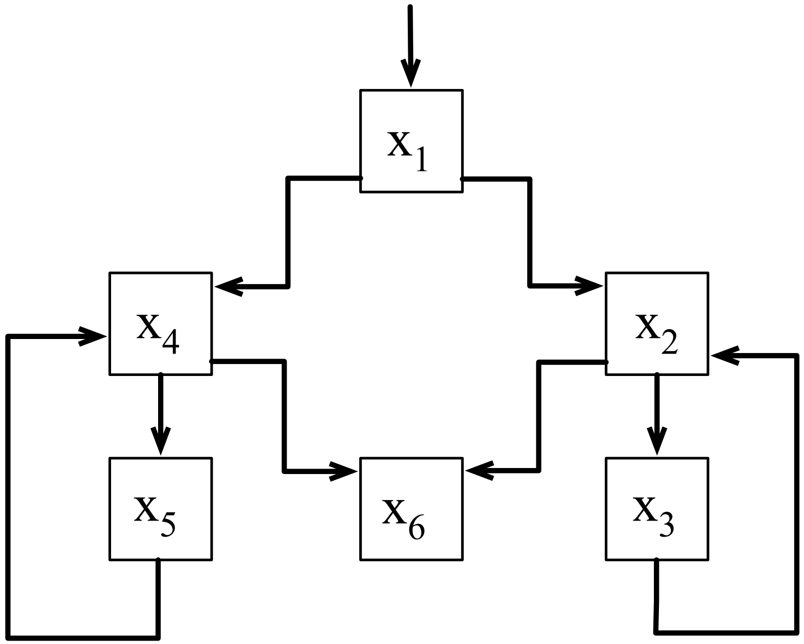
    rPostorder[n] = i

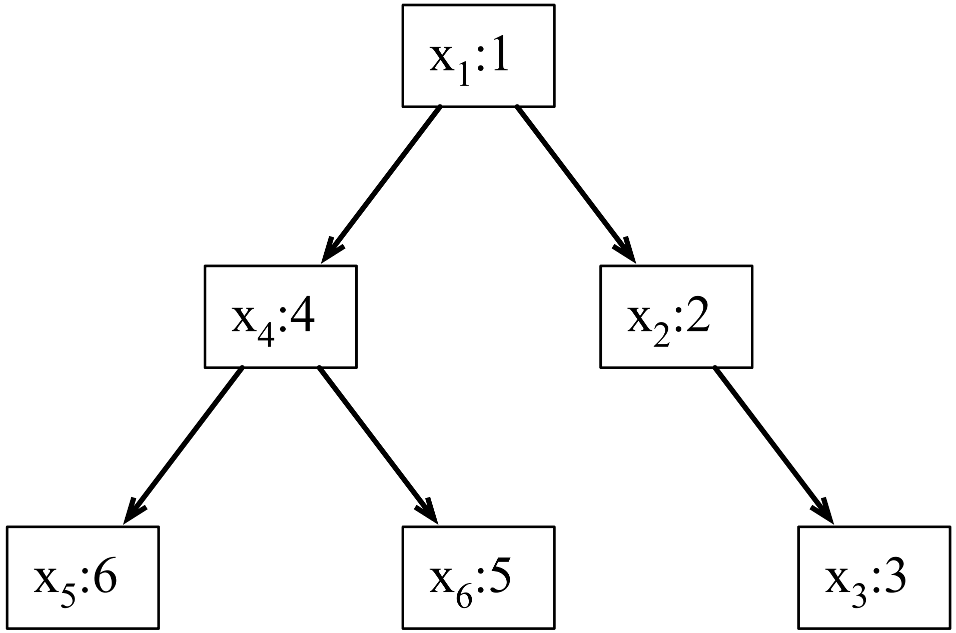
    i = i - 1

What would be a possible reverse post ordering of the graph below?



Reverse Postorder





How could we implement this ordering in our worklist?

Reverse Postorder

- We keep two data structures: C and P
- C is a <u>list</u> of current nodes to be visited
- P is a <u>set</u> of pending nodes to be visited after all the nodes in C have been visited
- Nodes are always inserted in P, and always extracted from C
- Once we are done with C, we sort the nodes in P in reverse postorder, and copy them into C

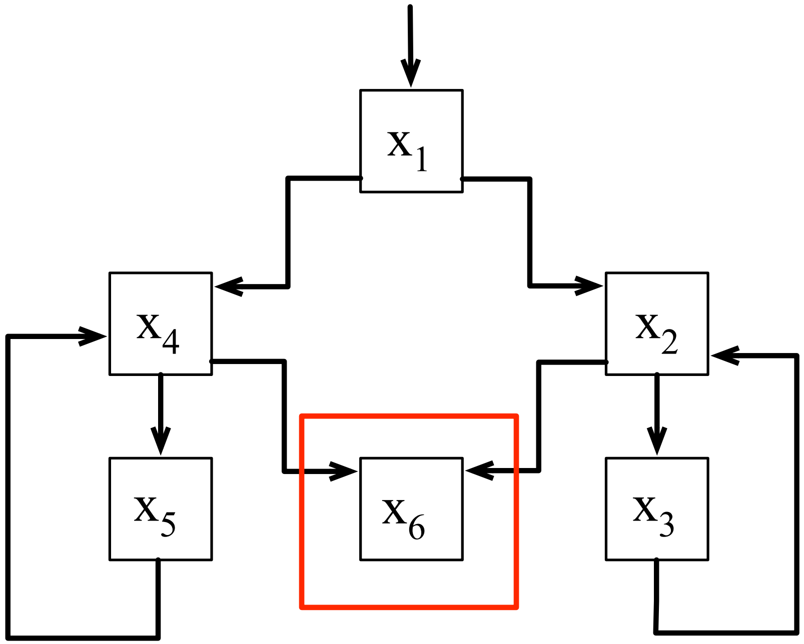
```
init = ([], {})

insert(v, (C, P)):
   return (C, (P ∪ {v}))

extract(C, P)
   if C = []
      C = sort_rPostorder(P)
      P = {}
   return (head(C), (tail(C), P))
```

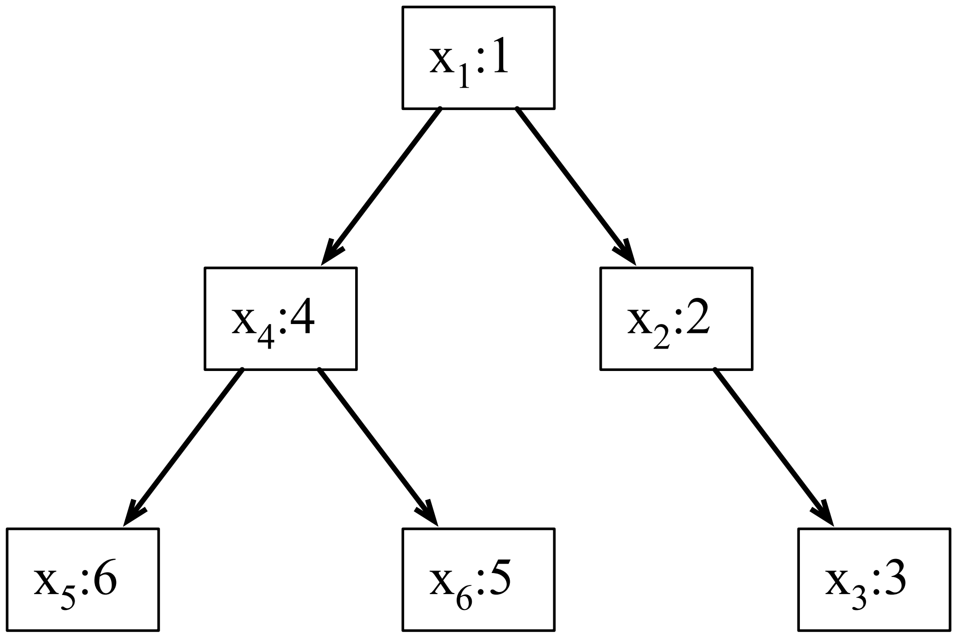
Reverse Postorder

x1 = {}
x2 = x1 ∪ (x3 \ {3, 5, 6}) ∪ {3}
x3 = x2
x4 = x1 ∪ (x5 \ {3, 5, 6}) ∪ {5}
x5 = x4
x6 = x2 ∪ x4

C	P	x1	x2	x3	x4	x5	x6
[]	{x1, ..., x6}	[]	[]	[]	[]	[]	[]
[x2, x3, x4, x6, x5]	{x2, x4}	[]	[]	[]	[]	[]	[]
[x3, x4, x6, x5]	{x2, x3, x4, x6}	[]	[3]	[]	[]	[]	[]
[x4, x6, x5]	{x2, x3, x4, x6}	[]	[3]	[3]	[]	[]	[]
[x6, x5]	{x2, ..., x6}	[]	[3]	[3]	[5]	[]	[]
[x5]	{x2, ..., x6}	[]	[3]	[3]	[5]	[]	[3,5]
[x2, x3, x4, x6, x5]	{}	[]	[3]	[3]	[5]	[5]	[3,5]
[x3, x4, x6, x5]	{}	[]	[3]	[3]	[5]	[5]	[3,5]
[x4, x6, x5]	{}	[]	[3]	[3]	[5]	[5]	[3,5]
[x6, x5]	{}	[]	[3]	[3]	[5]	[5]	[3,5]
[x5]	{}	[]	[3]	[3]	[5]	[5]	[3,5]
[]	{}	[]	[3]	[3]	[5]	[5]	[3,5]

Reverse Postorder



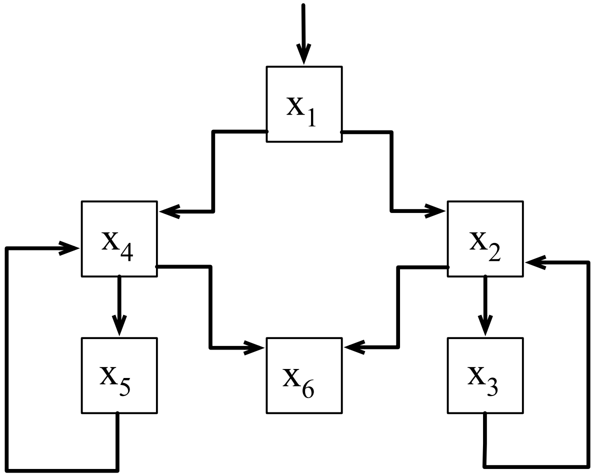


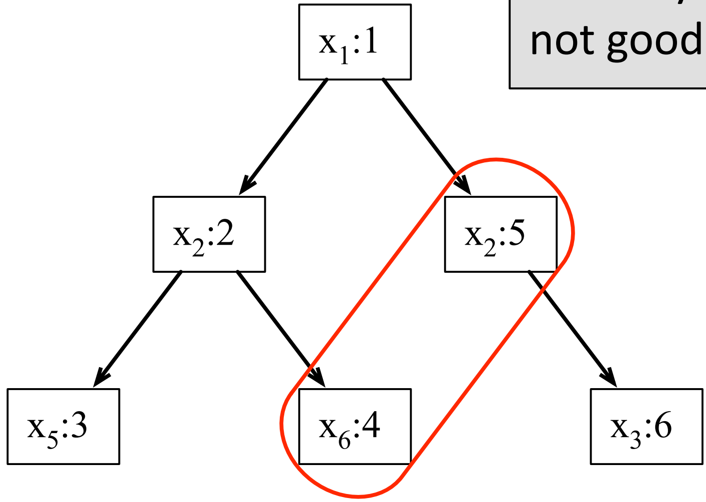
The reverse postorder ensures, for instance, that when we evaluate node x6, all its predecessors have been evaluated already.

Is there any other way to order the nodes that you could think about?

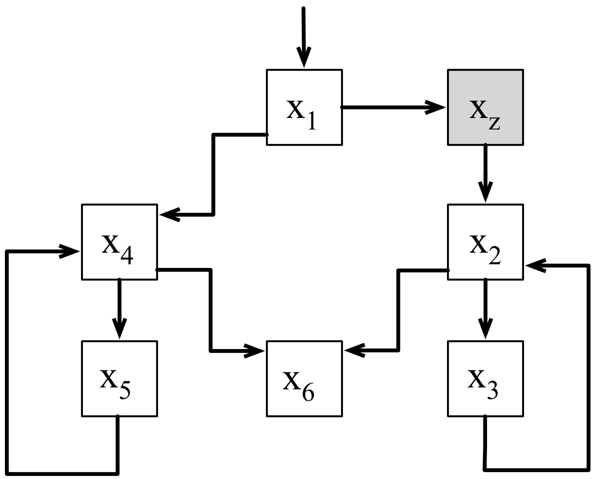
Other Common Orderings

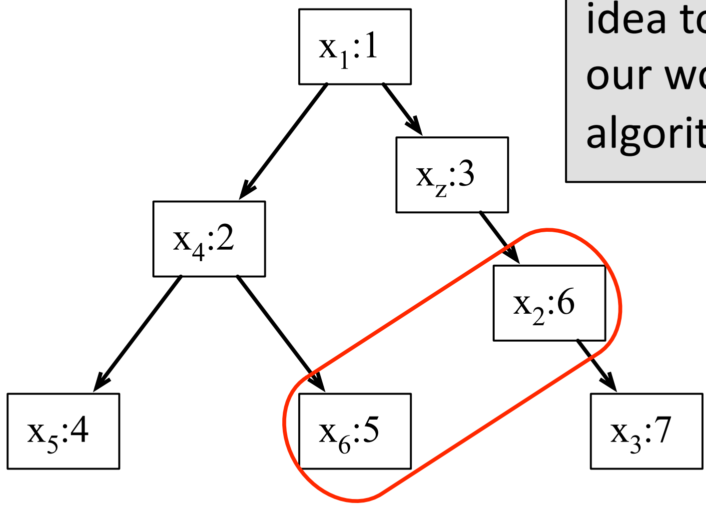
Which ordering is each one of these, and why are they not good?





Is there any other idea to speedup our worklist algorithm?



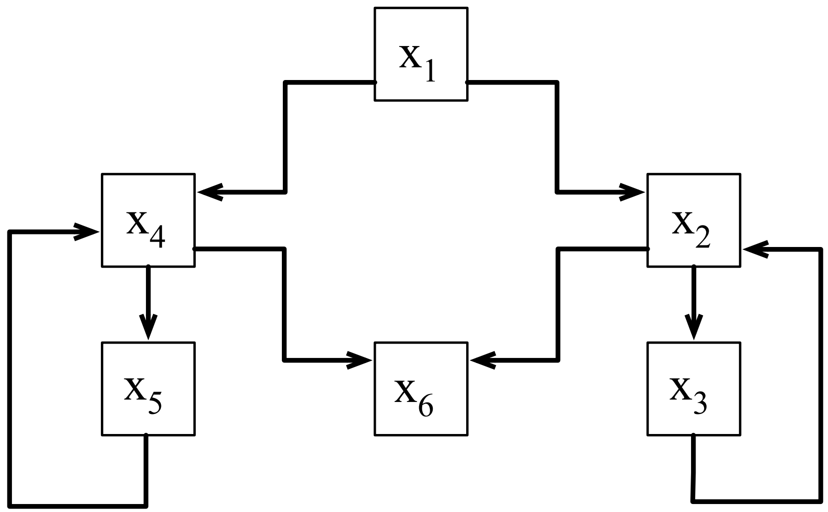


Strong Components

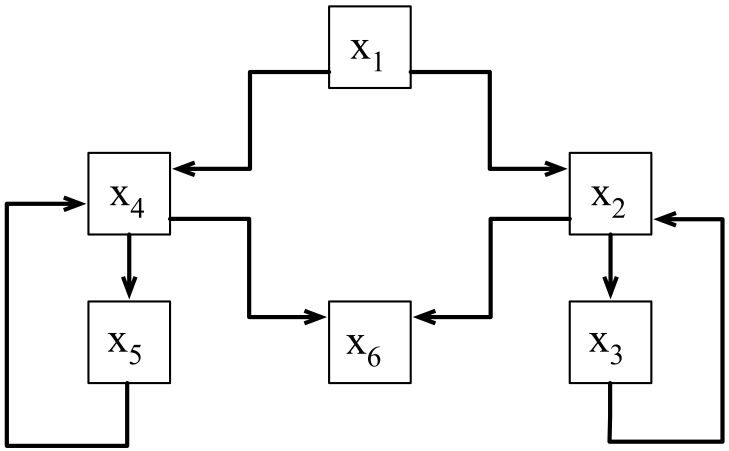
- A strong component of a graph is a maximal subgraph with the property that there is a path between any two nodes
- The reduced graph G_r of a graph G is the graph that we obtain by replacing each strong component of G by a single node.
  - We can order the reduced graph topologically.

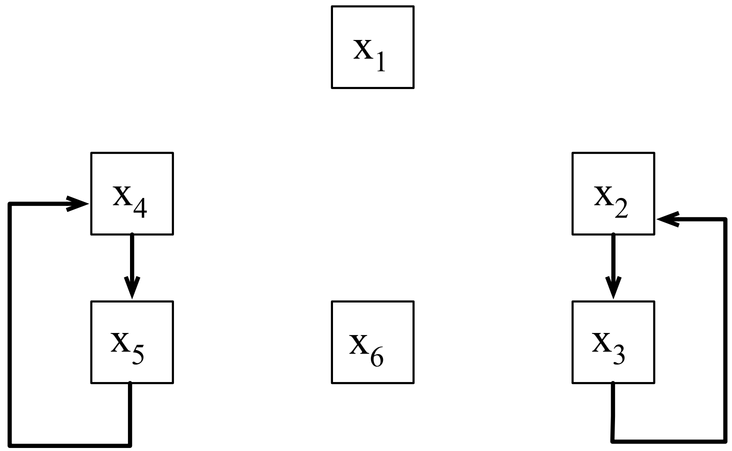
What is the reduced graph of the dependence graph below?

Topological ordering is well-defined for acyclic graphs. Why is the reduced graph always acyclic?



Strong Components

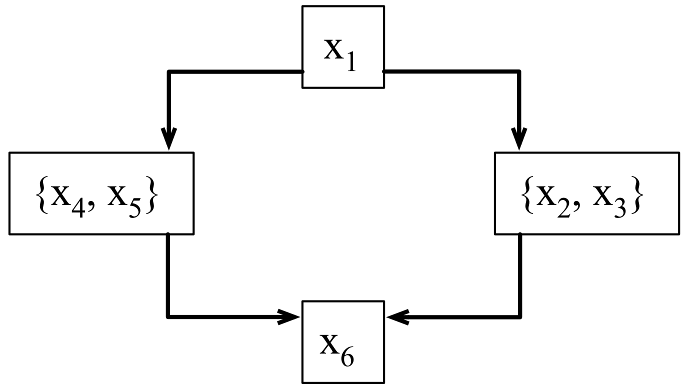




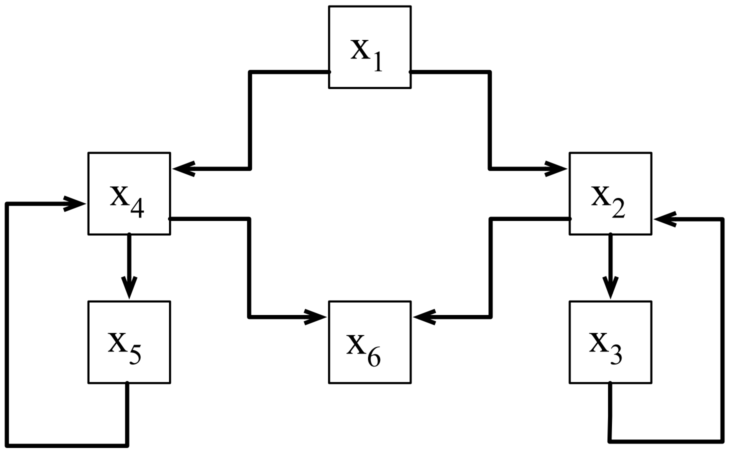
Dependence graph of constraints | Strong components

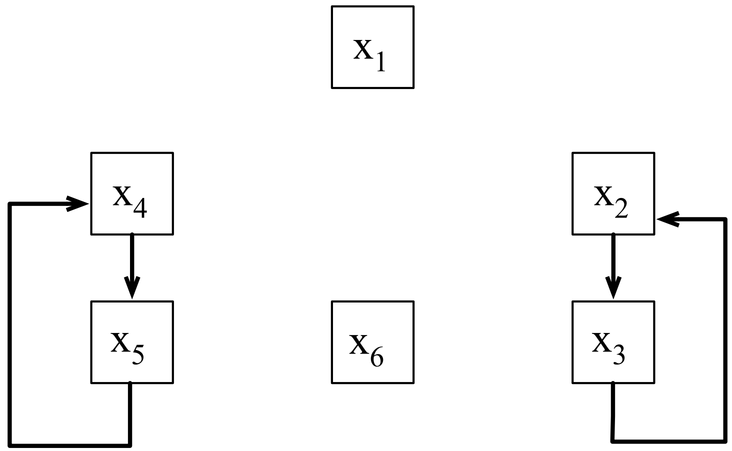
Question | Reduced Graph

How can we use this notion of strong components and reduced graphs to speed up our worklist algorithm?

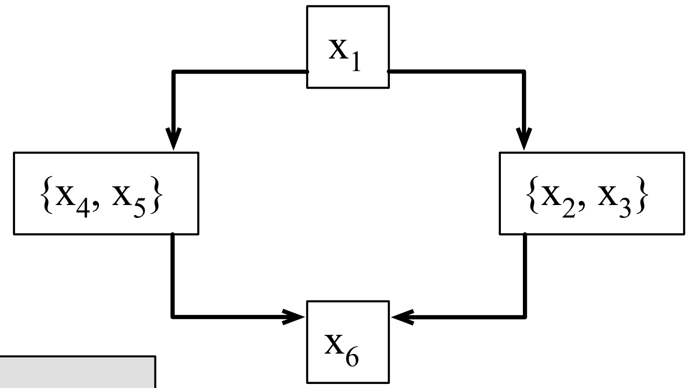


Strong Components



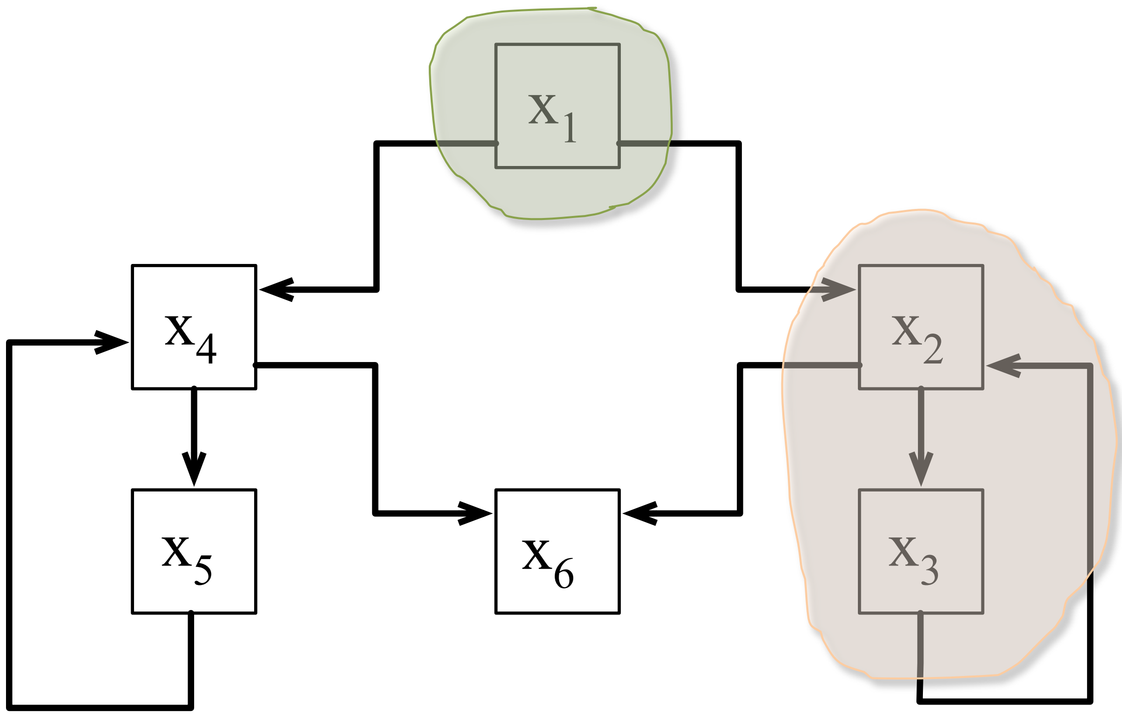


We can solve constraints for each SC, following a topological ordering of the reduced graph. *After we solve a component SC_i, the states of its variables* ***will not change*** *when we solve the components that come after SC_i in the topological ordering.*



Is ***this*** property always true?

Solve and Forget



The only way that we can change the information bound to a constraint variable is if the information of one of its predecessors in the dependence graph changes.

Is this statement obvious to you?

x1 = {}
x2 = x1 ∪ (x3 \ {3, 5, 6}) ∪ {3}
x3 = x2
x4 = x1 ∪ (x5 \ {3, 5, 6}) ∪ {5}
x5 = x4
x6 = x2 ∪ x4

For instance, in our running example, the only way that the information bound to x_2 and x_3 can change is if the information bound to x_1 changes. If we solve x_1 before, we can rest assured that x_2 and x_3, once solved, will not change anymore.

Strong Components

x1 = {}
x2 = x1 ∪ (x3 \ {3, 5, 6}) ∪ {3}
x3 = x2
x4 = x1 ∪ (x5 \ {3, 5, 6}) ∪ {5}
x5 = x4
x6 = x2 ∪ x4

SCC	P	x1	x2	x3	x4	x5	x6
{x1}	{x1, ..., x6}	[]	[]	[]	[]	[]	[]
{x2, x3}	{x2, ..., x6}	[]	[]	[]	[]	[]	[]
{x2, x3}	{x3, ..., x6}	[]	[3]	[]	[]	[]	[]
{x2, x3}	{x2, ..., x6}	[]	[3]	[3]	[]	[]	[]
{x2, x3}	{x3, ..., x6}	[]	[3]	[3]	[]	[]	[]
{x4, x5}	{x4, ..., x6}	[]	[3]	[3]	[]	[]	[]
{x4, x5}	{x5, ..., x6}	[]	[3]	[3]	[5]	[]	[]
{x4, x5}	{x4, ..., x6}	[]	[3]	[3]	[5]	[5]	[]
{x4, x5}	{x5, ..., x6}	[]	[3]	[3]	[5]	[5]	[]
{x6}	{x6}	[]	[3]	[3]	[5]	[5]	[]
{}	{}	[]	[3]	[3]	[5]	[5]	[3,5]

Simplification: Round-Robin Iterations

$x_1 = \perp, x_2 = \perp, \ldots, x_n = \perp$

change = true

while (change)

  change = false

  for i = 1 to n do

  $y = F_i(x_1, \ldots, x_n)$

  if $y \neq x_i$

    change = true

    $x_i = y$

- This implementation simplifies our worklist
  - We do not need to keep a list of pending nodes P
  - We do not need to worry about efficient implementations of extract and insert
- The drawback is that we may have to iterate more times over the constraint variables.

Why we may have to iterate more times over the constraint variables?

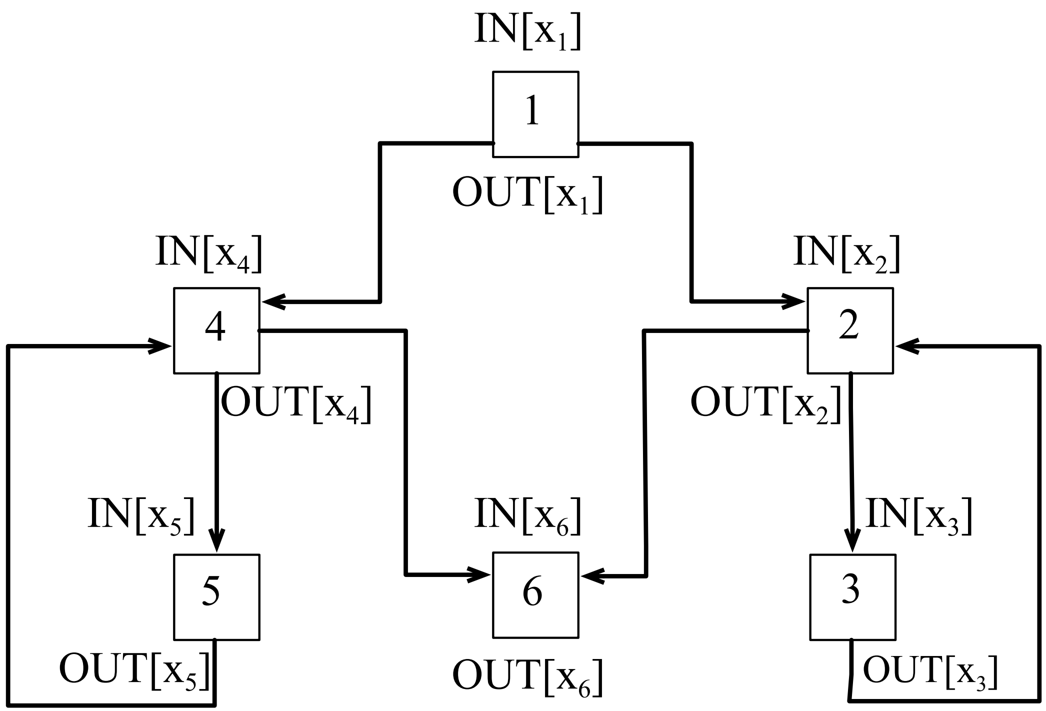
Round-Robin Iterations

x1 = {}
x2 = x1 ∪ (x3 \ {3, 5, 6}) ∪ {3}
x3 = x2
x4 = x1 ∪ (x5 \ {3, 5, 6}) ∪ {5}
x5 = x4
x6 = x2 ∪ x4

change	x1	x2	x3	x4	x5	x6
TRUE	[]	[]	[]	[]	[]	[]
FALSE	Beginning of a new iteration					
FALSE	[]	[]	[]	[]	[]	[]
TRUE	[]	[3]	[]	[]	[]	[]
TRUE	[]	[3]	[3]	[]	[]	[]
TRUE	[]	[3]	[3]	[5]	[]	[]
TRUE	[]	[3]	[3]	[5]	[5]	[]
TRUE	[]	[3]	[3]	[5]	[5]	[3,5]
FALSE	Beginning of a new iteration					
FALSE	[]	[3]	[3]	[5]	[5]	[3,5]
FALSE	[]	[3]	[3]	[5]	[5]	[3,5]
FALSE	[]	[3]	[3]	[5]	[5]	[3,5]
FALSE	[]	[3]	[3]	[5]	[5]	[3,5]
FALSE	[]	[3]	[3]	[5]	[5]	[3,5]
FALSE	[]	[3]	[3]	[5]	[5]	[3,5]

Representation of Sets

- Our data flow analyses have been storing information in sets.
- The result of each analysis is a data structure that maps program points to sets of facts.



1) How should these sets be implemented?
2) Does your implementation ensure fast union, difference and intersection operations?
3) Does your implementation perform well when you have a sparse data flow analyses, in which each set contains only a few elements?
4) What about a dense analysis?

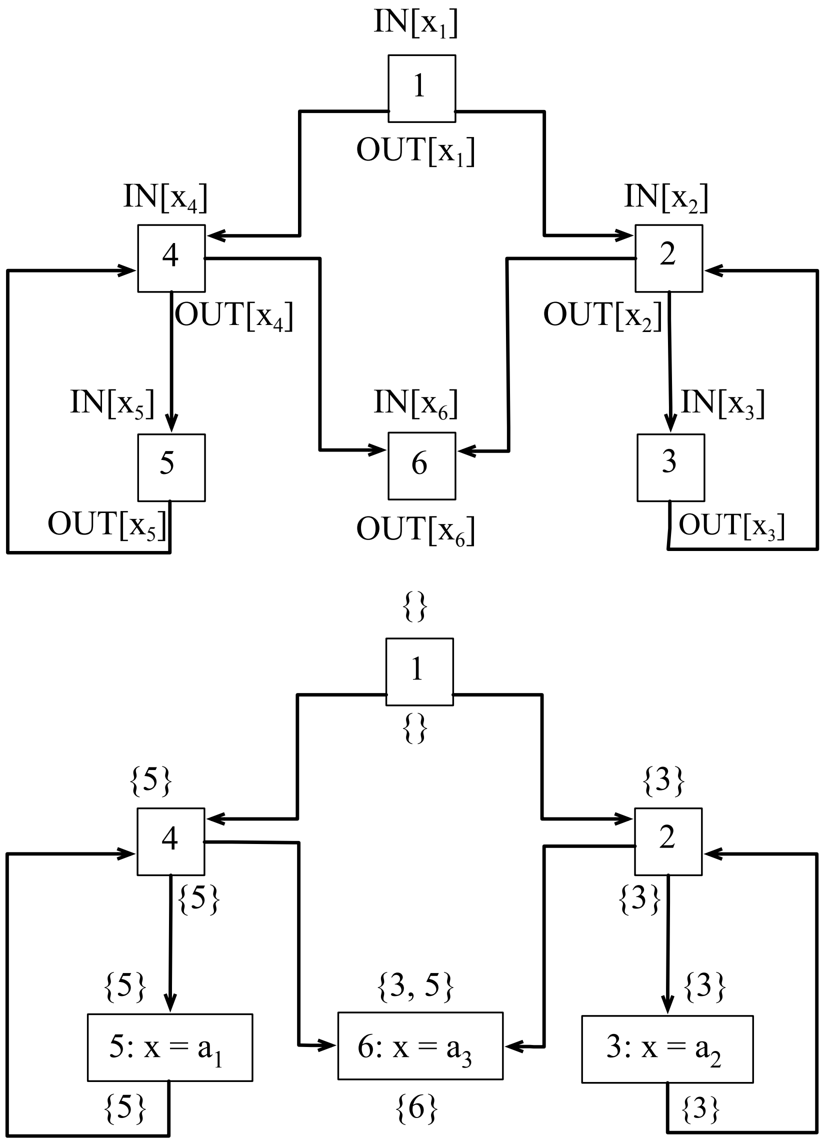
Representation of Sets

- Usually these sets can be implemented as bit-vectors, as hash-tables, or as linked lists.
- Bit-vectors do well in dense analyses, in which case each program point might be associated with many variables or expressions.
  - If the int word is K bits wide, then we need N/K words, where N is the number of data-flow facts.
  - Insertion is usually O(1).
  - Set operations are linear on the number of words in the set.

1) How could we represent, for instance, reaching definitions as bit vectors?

2) How to represent available expressions as bit vectors?

Bit-Vectors



	1	2	3	4	5	6
$IN[x_1]$	○	○	○	○	○	○
$OUT[x_1]$	○	○	○	○	○	○
$IN[x_2]$	○	○	●	○	○	○
$OUT[x_2]$	○	○	●	○	○	○
$IN[x_3]$	○	○	●	○	○	○
$OUT[x_3]$	○	○	●	○	○	○
$IN[x_4]$	○	○	○	○	●	○
$OUT[x_4]$	○	○	○	○	●	○
$IN[x_5]$	○	○	○	○	●	○
$OUT[x_5]$	○	○	○	○	●	○
$IN[x_6]$	○	○	●	○	●	○
$OUT[x_6]$	○	○	○	○	○	●

A Bit of History

- Worklist algorithms have been used in computer science for many years. The first use to solve dataflow problems seems to have appeared in a paper by Kildall.
- For a use of strong components to solve dataflow problems, see the work of Horwitz *et al*.
- There are many, really many, variations of worklist solvers in the literature, e.g., local, differential, region based, etc.

- Kildall, G. "A Unified Approach to Global Program Optimization", POPL, 194-206 (1973)
- Hecht, M. S. "Flow Analysis of Computer Programs", North Holland, (1977)
- Horwitz, S. Demers, A. and Teitelbaum, T. "An efficient general iterative algorithm for dataflow analysis", Acta Informatica, 24, 679-694 (1987)