POINTER ANALYSIS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides has been taken from two different papers, [Hardekopf07] and [Pereira11]. See "A bit of History" in the last slide

Pointers

- Pointers are a feature of imperative programming languages.
- They are very useful, because they avoid the need to copy entire data-structures when passing information from one program routine to another.
- Nevertheless, pointers make it very hard to reason about programs.
- The direct consequence of these difficulties is that compilers have a hard time trying to understand and modify imperative programs.
  - Compilers's main ally in this battle is the *pointer analysis*, also called alias analysis, or points-to analysis.

The Need for Pointer Analysis

```c
#include <stdio.h>
int main() {
  int i = 7;
  int* p = &i;
  *p = 13;
  printf("The value of i = %d\n", i);
}
```

1) What does the program on the left do?
2) How could this program be optimized?
3) Which information is needed to optimize this program?
4) Do you think `gcc -O1` can optimize this program?

The Need for Pointer Analysis

```
#include <stdio.h>
int main() {
  int i = 7;
  int* p = &i;
  *p = 13;
  printf("The value of i = %d\n", i);
}
```

```
$> gcc -O1 pointer.c -S
```

Which information was needed to optimize the target program?

```
_main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $20, %esp
        call    L3
"L00000000001$pb":
L3:
        popl    %ebx
        movl    $13, 4(%esp)
        leal    LC0-"L1$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    _printf
        addl    $20, %esp
        popl    %ebx
        leave
        ret
        .subsections_via_symbols
```

The Need for Pointer Analysis

```
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}
```

How could we optimize this program?

The Need for Pointer Analysis

```
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}
```

```
void sum1(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    int tmp = a[i];
    if (!b[i]) {
      tmp = b[i];
    }
    r[i] = tmp;
  }
}
```

How much faster do you think is sum1?

The Need for Pointer Analysis

```
$> time ./a.out
sum0

        0         1         0         3
        0        11         0        13

real   0m6.299s
user   0m6.285s
sys    0m0.008s


$> time ./a.out a
sum1

        0         1         0         3
        0        11         0        13
real   0m1.345s
user   0m1.340s
sys    0m0.004s
```

```c
int main(int argc, char** argv) {
  int* a = (int*) malloc(SIZE * 4);
  int* b = (int*) malloc(SIZE * 4);
  int* c = (int*) malloc(SIZE * 4);
  int i;
  for (i = 0; i < SIZE; i++) {
    a[i] = i;
    b[i] = i%2;
  }
  if (argc % 2) {
    printf("sum0\n");
    for (i = 0; i < LOOP; i++) {
      sum0(a, b, c, SIZE);
    }
  } else {
    printf("sum1\n");
    for (i = 0; i < LOOP; i++) {
      sum1(a, b, c, SIZE);
    }
  }
  print(c, 20);
}
```

If you want to test it…

```c
#include <stdio.h>
#include <stdlib.h>
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}
void sum1(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    int tmp = a[i];
    if (!b[i]) {
      tmp = b[i];
    }
    r[i] = tmp;
  }
}
void print(int* a, int N) {
  int i;
  for (i = 0; i < N; i++) {
    if (i % 10 == 0) {
      printf("\n");
    }
    printf("%8d", a[i]);
  }
}
```

```c
#define SIZE 10000
#define LOOP 100000

int main(int argc, char** argv) {
  int* a = (int*) malloc(SIZE * 4);
  int* b = (int*) malloc(SIZE * 4);
  int* c = (int*) malloc(SIZE * 4);
  int i;
  for (i = 0; i < SIZE; i++) {
    a[i] = i;
    b[i] = i%2;
  }
  if (argc % 2) {
    printf("sum0\n");
    for (i = 0; i < LOOP; i++) {
      sum0(a, b, c, SIZE);
    }
  } else {
    printf("sum1\n");
    for (i = 0; i < LOOP; i++) {
      sum1(a, b, c, SIZE);
    }
  }
  print(c, 20);
}
```

The Need for Pointer Analysis

```
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}
```

```
L4:
  movl     (%edi,%edx,4), %eax
  movl     %eax, (%ecx,%edx,4)
  cmpl     $0, (%esi,%edx,4)
  jne      L5
  movl     $0, (%ecx,%edx,4)
L5:
  incl     %edx
  cmpl     %ebx, %edx
  jne      L4
```

```
void sum1(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    int tmp = a[i];
    if (!b[i]) {
      tmp = b[i];
    }
    r[i] = tmp;
  }
}
```

This code **here** is quite smart. Do you understand it?

```
L12:
  cmpl     $0, (%esi,%edx,4)
  movl     $0, %eax
  cmovne   (%edi,%edx,4), %eax
  movl     %eax, (%ebx,%edx,4)
  incl     %edx
  cmpl     %ecx, %edx
  jne      L12
```

The Need for Pointer Analysis

```
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}
```

Why is gcc -O1 unable to optimize this program?

The Need for Pointer Analysis

```
void sum0(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    r[i] = a[i];
    if (!b[i]) {
      r[i] = b[i];
    }
  }
}
```

Would both programs produce the same result if the second and third parameters were pointers to the same array?

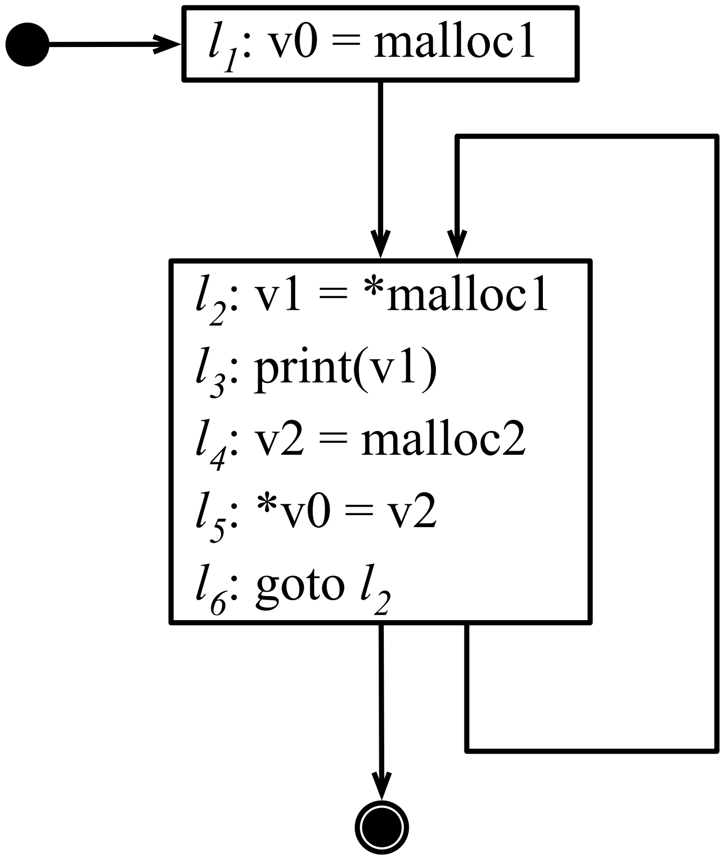
```
void sum1(int* a, int* b, int* r, int N) {
  int i;
  for (i = 0; i < N; i++) {
    int tmp = a[i];
    if (!b[i]) {
      tmp = b[i];
    }
    r[i] = tmp;
  }
}
```

Pointer Analysis

- The goal of pointer analysis is to determine which are the memory locations pointed by each pointer in the program.
- Pointer analysis is usually described and solved as a constraint based analysis.
- The most efficient algorithm that we know about is $O(n^3)$.
  - This complexity is very high, and pointer analysis takes too long in very large programs.
- After register allocation, pointer analysis is possibly the most studied topic inside the science of compiler design.
  - Research aims at speed and precision.

Can we solve Pointer Analysis with Data-flow?

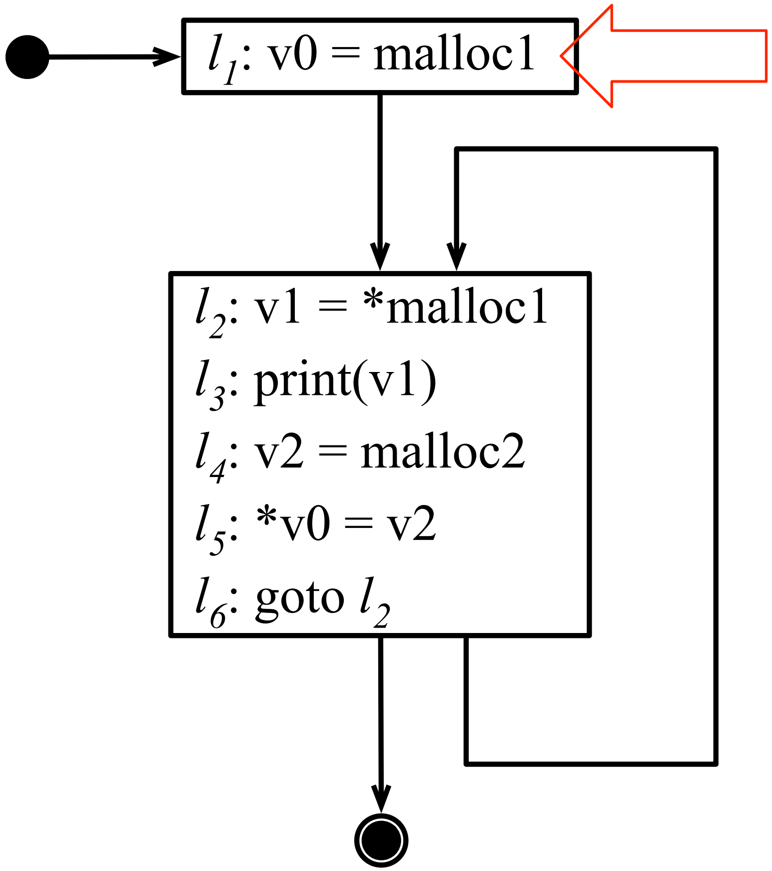
- We must determine, for each variable in the program, the set of memory locations that the variable may point to.



What are the points-to sets associated with the variables v0, v1, v2, malloc1 and malloc2 in this program?

Can we solve Pointer Analysis with Data-flow?

- We must determine, for each variable in the program, the set of memory locations that the variable may point to.

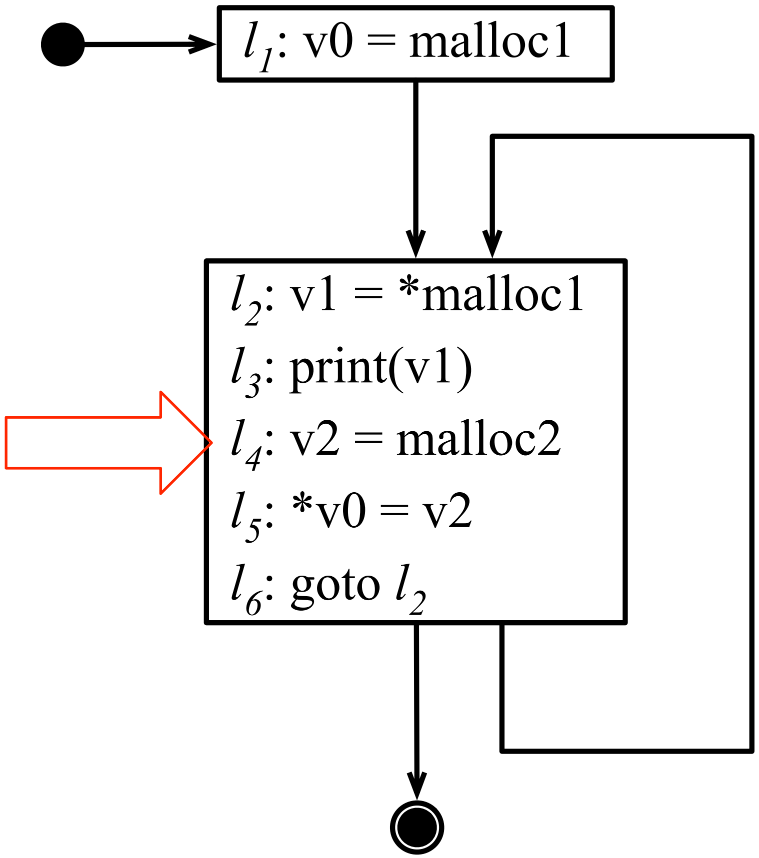


First of all, we know that malloc1 $\in$ P(v0), due to the assignment at label 1

$$P(v0) \supseteq \{malloc1\}$$

Can we solve Pointer Analysis with Data-flow?

- We must determine, for each variable in the program, the set of memory locations that the variable may point to.

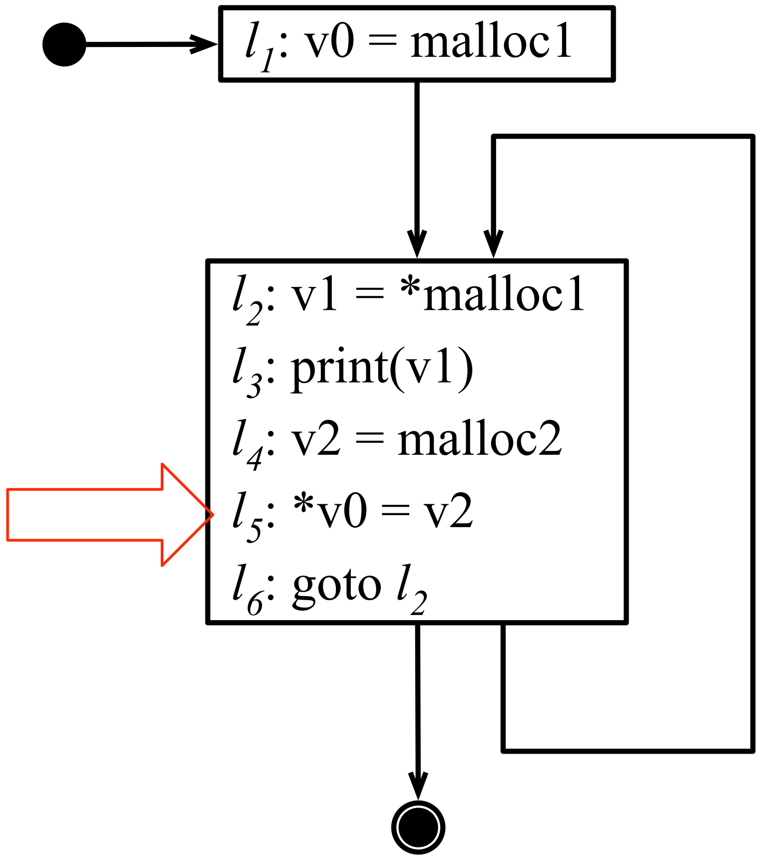


We also know that malloc2 ∈ P(v2), due to the assignment at label 4

P(v0) ⊇ {malloc1}
P(v2) ⊇ {malloc2}

Can we solve Pointer Analysis with Data-flow?

- We must determine, for each variable in the program, the set of memory locations that the variable may point to.

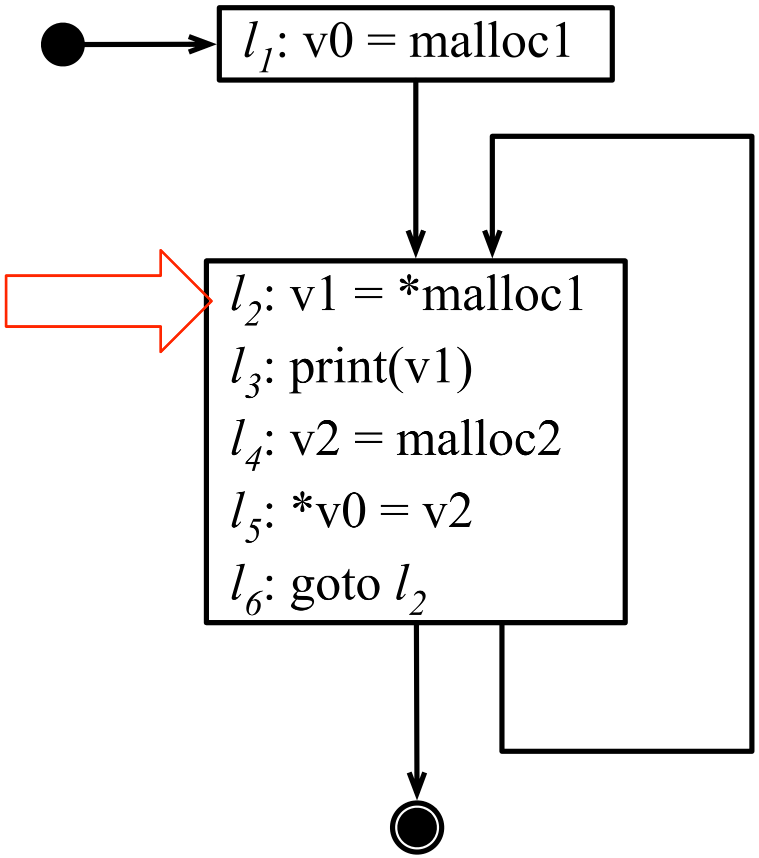


Line 5 is weird: it adds to the points-to set of the things pointed by v0 the points-to set of v2. At this points, P(v0) = {malloc1}

$P(v0) \supseteq \{malloc1\}$
$P(v2) \supseteq \{malloc2\}$
$P(malloc1) \supseteq \{malloc2\}$

Can we solve Pointer Analysis with Data-flow?

- We must determine, for each variable in the program, the set of memory locations that the variable may point to.

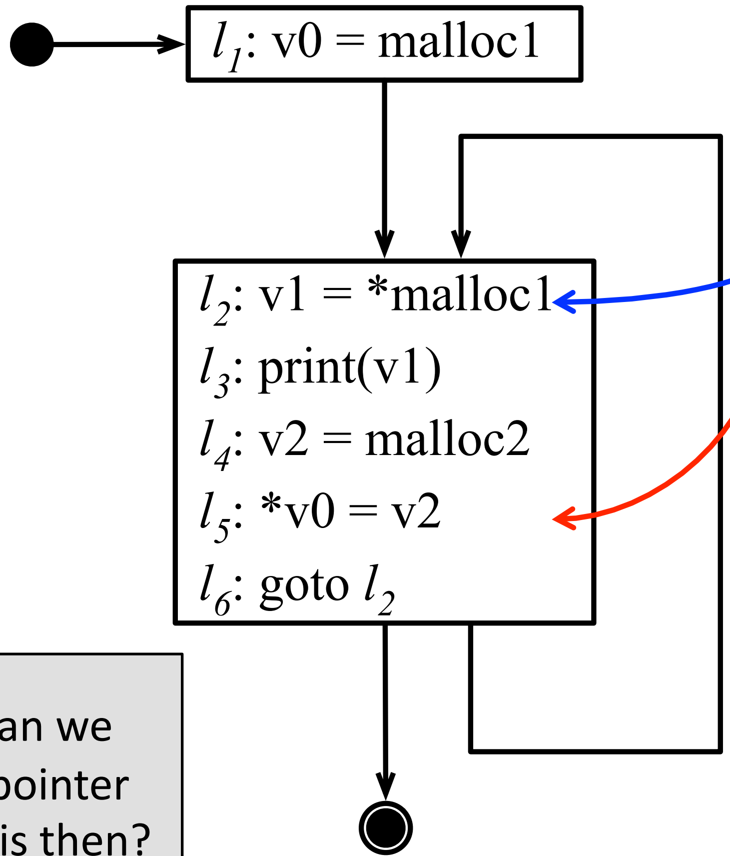


But, now P(malloc1) has been changed, and line 2 start being important: P(v1) must include the points-to sets of things pointed by malloc1

P(v0) $\supseteq$ {malloc1}
P(v2) $\supseteq$ {malloc2}
P(malloc1) $\supseteq$ {malloc2}
P(v1) $\supseteq$ {malloc2}

Can we solve Pointer Analysis with Data-flow?

- We must determine, for each variable in the program, the set of memory locations that the variable may point to.



The difficulty of pointer analysis is that an assignment at **line 5** ends up having impact in **line 2**, but there is no syntactic connection between these two lines.

Pointer analysis cannot be solved by our data-flow framework, because the flow of information is not explicit in the syntax of the program, and it changes as more information is discovered.

How can we solve pointer analysis then?

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

ANDERSEN'S POINTER ANALYSIS

DCC 888

fernando@dcc.ufmg.br

Solving Pointer Analysis with Constraints

- Four different constructs typically found in imperative programming languages give us constraints:

Statement	Constraint name	Constraint
`a = &b`	base	$P(a) \supseteq \{b\}$
`a = b`	simple	$P(a) \supseteq P(b)$
`a = *b`	load	$t \in P(b) \Rightarrow P(a) \supseteq P(t)$
`*a = b`	store	$t \in P(a) \Rightarrow P(t) \supseteq P(b)$

> Are these constraints similar to those seen in the control flow analysis?

- This type of constraints are *inclusion based*. This pointer analysis is called *Andersen style*, after Lars Andersen, who first described the inclusion based points-to analysis[♤].

Solving Points-to Analysis with Graphs

- We can use an approach similar to that seen in the constraint based analyses to solve points-to analysis; however, notice that the constraints are not exactly the same:

Constraint of Pointer Analysis

$P(a) \supseteq \{b\}$

$P(a) \supseteq P(b)$

$t \in P(b) \Rightarrow P(a) \supseteq P(t)$

$t \in P(a) \Rightarrow P(t) \supseteq P(b)$

Constraint of Control Flow Analysis

$lhs \subseteq rhs$

$\{t\} \subseteq rhs' \Rightarrow lhs \subseteq rhs$

Do you remember the graph based algorithm to solve constraint based analyses?

Solving Points-to Analysis with Graphs

- We will solve points-to analysis with the same algorithm that we have used to solve control flow analysis.
- We will show several improvements on that algorithm that we can use to make it scale to handle very large points-to sets.

Constraint of Pointer Analysis

$P(a) \supseteq \{b\}$

$P(a) \supseteq P(b)$

$t \in P(b) \Rightarrow P(a) \supseteq P(t)$

$t \in P(a) \Rightarrow P(t) \supseteq P(b)$

Constraint of Control Flow Analysis

$lhs \subseteq rhs$

$\{t\} \subseteq rhs' \Rightarrow lhs \subseteq rhs$

The Points-to Graph

- The points-to graph is a graph (V, E) used to solve the pointer analysis.
  - The graph has a node for each variable v in the constraint system.
    - Each node is associated with a points-to set P(v)
  - The graph has an edge (v_1, v_2) if $v_1 \subseteq v_2$
  - Initially, the points-to graph has an edge for each constraint $P(v_1) \supseteq P(v_2)$ in the constraint system.

```
b  =  &a
a  =  &c
d  =  a
*d  =  b
a  =  *d
```

How are the constraints for this program?

The Points-to Graph

- The points-to graph is a graph (V, E) used to solve the pointer analysis.
  - The graph has a node for each variable v in the constraint system.
    - Each node is associated with a points-to set P(v)
  - The graph has an edge (v_1, v_2) if $v_1 \subseteq v_2$
  - Initially, the points-to graph has an edge for each constraint $P(v_1) \supseteq P(v_2)$ in the constraint system.

```
b  =  &a
a  =  &c
d  =  a
*d  =  b
a  =  *d
```

How is the initial points-to graph for the program on the left?

$P(b) \supseteq \{a\}$

$P(a) \supseteq \{c\}$

$P(d) \supseteq P(a)$

$t \in P(d) \Rightarrow P(t) \supseteq P(b)$

$t \in P(d) \Rightarrow P(a) \supseteq P(t)$

The Points-to Graph

- The points-to graph is a graph (V, E) used to solve the pointer analysis.
  - The graph has a node for each variable v in the constraint system.
    - Each node is associated with a points-to set P(v)
  - The graph has an edge (v_1, v_2) if $v_1 \subseteq v_2$
  - Initially, the points-to graph has an edge for each constraint $P(v_1) \supseteq P(v_2)$ in the constraint system.

$P(b) \supseteq \{a\}$

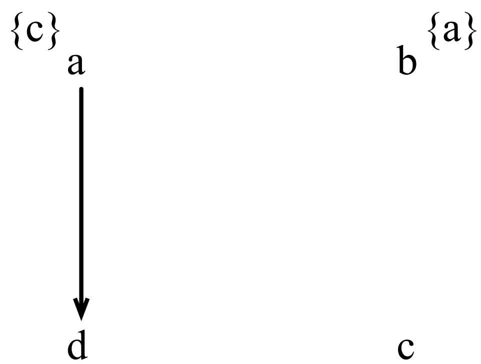
$P(a) \supseteq \{c\}$

$P(d) \supseteq P(a)$

$t \in P(d) \Rightarrow P(t) \supseteq P(b)$

$t \in P(d) \Rightarrow P(a) \supseteq P(t)$

How can we solve the rest of the points-to analysis?



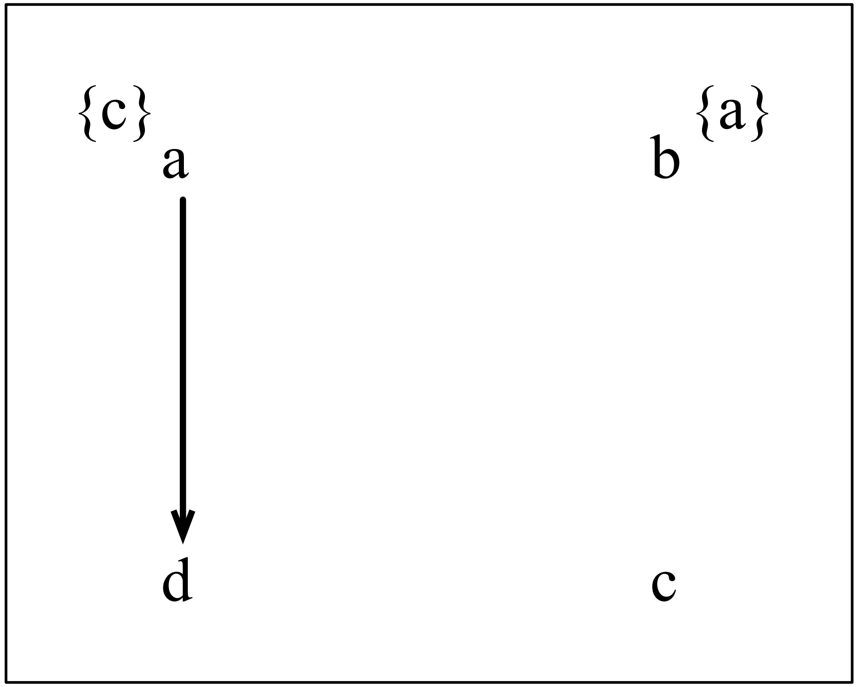
The Iterative Solver

- Once we have an initial points-to graph, we iterate over the load and store constraints (henceforth called complex constraints), alternating two steps:
  - Adding new edges to the graph
  - Propagating points-to information
- This algorithm effectively builds the transitive closure of the points-to graph

let G = (V, E)
W = V
while W ≠ [] **do**
  n = hd(W)
  for each v ∈ P(n) **do**
    for each load "a = *n" **do**
      if (v, a) ∉ E **then**
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" **do**
      if (b, v) ∉ E **then**
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E **do**
    P(z) = P(z) ∪ P(n)
    if P(z) has changed **then**
      W = z::W

The Iterative Solver

How the graph will look like after the initial propagation phase?



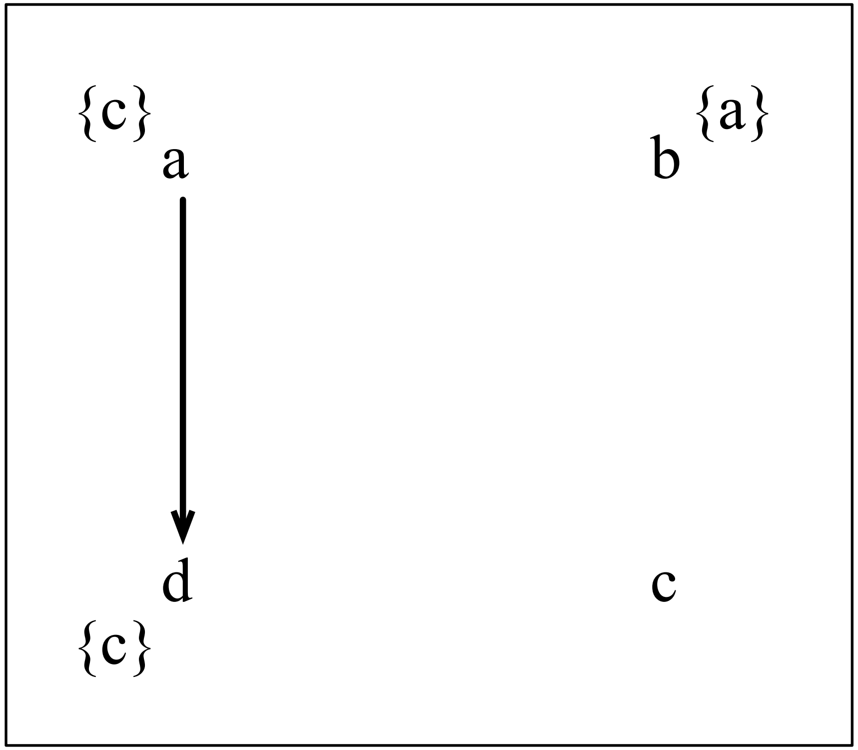

```
let G = (V, E)
W = V
while W ≠ [] do
  n = hd(W)
  for each v ∈ P(n) do
    for each load "a = *n" do
      if (v, a) ∉ E then
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" do
      if (b, v) ∉ E then
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E do
    P(z) = P(z) ∪ P(n)
    if P(z) has changed then
      W = z::W
```

The Iterative Solver

```
*d = b

a = *d
```

Now, we have two complex constraints to evaluate. How will be the first evaluation?



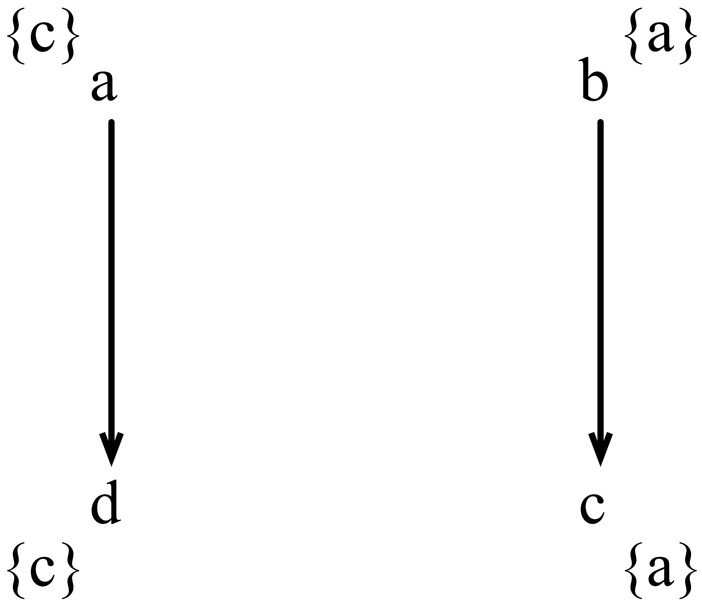
let G = (V, E)
W = V
while W ≠ [] **do**
  n = hd(W)
  for each v ∈ P(n) **do**
    for each load "a = *n" **do**
      if (v, a) ∉ E **then**
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" **do**
      if (b, v) ∉ E **then**
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E **do**
    P(z) = P(z) ∪ P(n)
    if P(z) has changed **then**
      W = z::W

The Iterative Solver

```
*d = b

a = *d
```

How will be the graph after we evaluate the load?



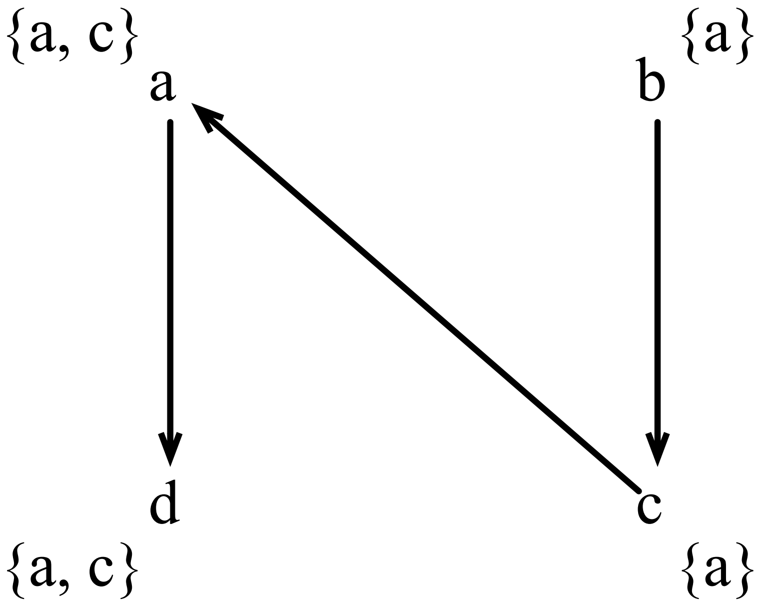

```
let G = (V, E)
W = V
while W ≠ [] do
  n = hd(W)
  for each v ∈ P(n) do
    for each load "a = *n" do
      if (v, a) ∉ E then
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" do
      if (b, v) ∉ E then
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E do
    P(z) = P(z) ∪ P(n)
    if P(z) has changed then
      W = z::W
```

The Iterative Solver

```
*d = b

a = *d
```

Are we done, or has any points-to set changed; hence, forcing a new iteration?



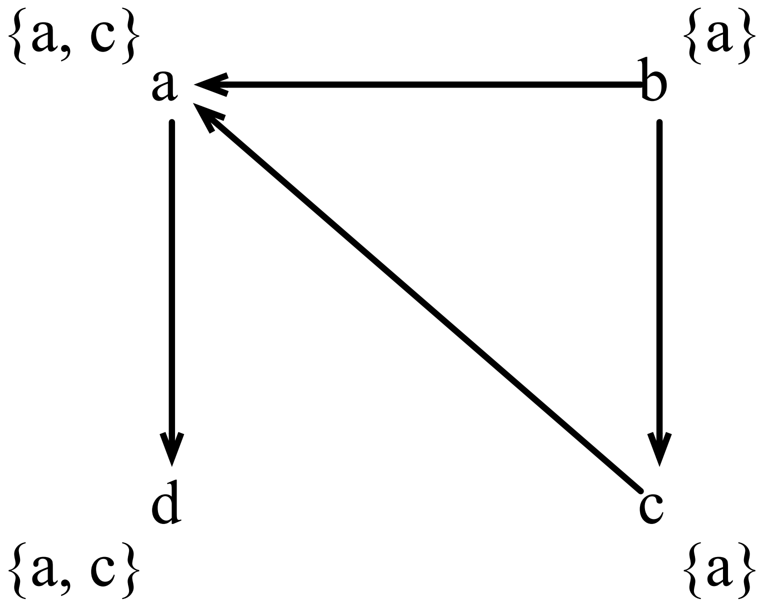

```
let G = (V, E)
W = V
while W ≠ [] do
  n = hd(W)
  for each v ∈ P(n) do
    for each load "a = *n" do
      if (v, a) ∉ E then
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" do
      if (b, v) ∉ E then
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E do
    P(z) = P(z) ∪ P(n)
    if P(z) has changed then
      W = z::W
```

The Iterative Solver

```
*d = b

a = *d
```

What is going to be the next action in this algorithm?



let G = (V, E)
W = V
while W $\neq$ [] **do**
  n = hd(W)
  for each v $\in$ P(n) **do**
    for each load "a = *n" **do**
      if (v, a) $\notin$ E **then**
        E = E $\cup$ {(v, a)}
        W = v::W
    for each store "*n = b" **do**
      if (b, v) $\notin$ E **then**
        E = E $\cup$ {(b, v)}
        W = b::W
  for each (n, z) $\in$ E **do**
    P(z) = P(z) $\cup$ P(n)
    if P(z) has changed **then**
      W = z::W

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

COLLAPSING CYCLES

DCC 888

fernando@dcc.ufmg.br

The Problems with the Iterative Solver

- Finding the transitive closure of a graph has an $O(n^3)$ algorithm.
  - This algorithm works well for small graphs, but it is very slow for very large programs.
    - Programs like the kernel of Linux.

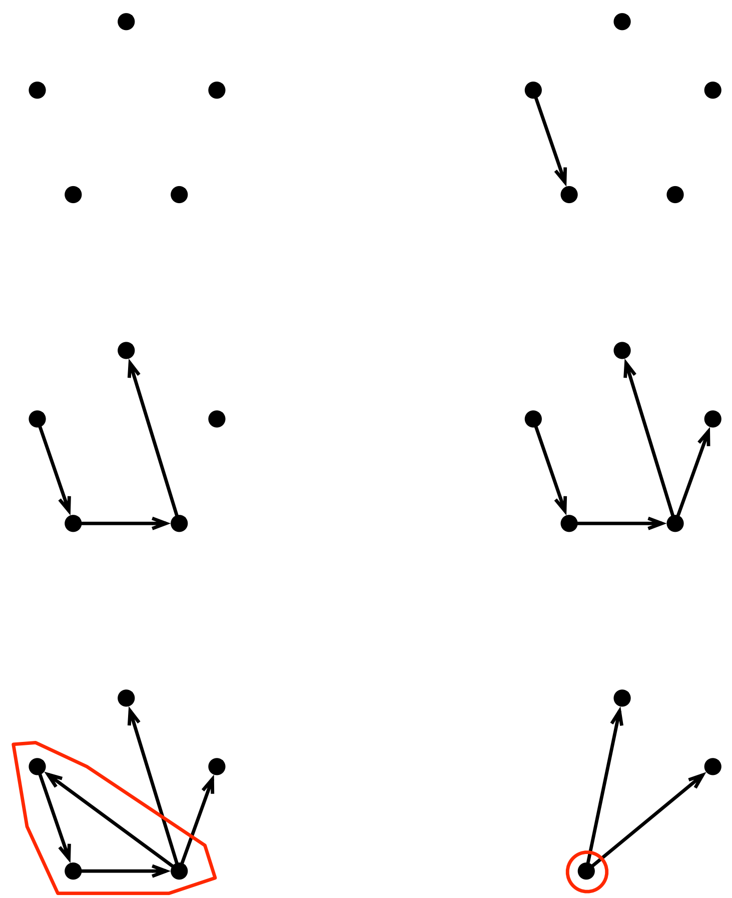
- But, in the early 2000's, researchers have made a very interesting observation:
  - All the nodes in a cycle always have the same points-to set.

1) Why is this observation true?

2) How can we use it to improve our algorithm?

Collapsing Cycles

- The key idea to capitalize on cycles is to collapse them, once we find them.



How to identify cycles in the graph?

Cycle Identification

- Collapsing cycles is very important: it is the difference between an algorithm that can be used, and one that cannot be used in practice.
- Although identifying cycles in graphs is easy, e.g., we can find them with a depth-first traversal of the graph, using this strategy in practice is not trivial.

1) What is the problem with using a DFS to find cycles?

2) But how can we still capitalize on cycles?

Lazy Cycle Detection

- Compiler researchers have identified many ways to detect cycles during the construction of the transitive closure:
  - Wave propagation
  - Deep propagation
  - Lazy cycle detection
- We shall be describing the lazy cycle detection approach♧, because it is easy to write in pseudo-code.

♧: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code, 2007

Lazy Cycle Detection

- If we connect two nodes v_1 and v_2 that have the same points-to set, chances are that we are in the middle of a cycle.
  - We can try to find a cycle starting from v_2
  - If we can indeed find a cycle, then we collapse it
  - Otherwise we mark the edge (v_1, v_2) to avoid future searches that are likely to fail.

```
let G = (V, E)
R = {}
W = V
while W ≠ [] do
  n = hd(W)
  for each v ∈ P(n) do
    for each load "a = *n" do
      if (v, a) ∉ E then
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" do
      if (b, v) ∉ E then
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E do
    if P(z) = P(n) and (n, z) ∉ R then
      DetectAndCollapseCycles(z)
      R = R ∪ {(n, z)}
    P(z) = P(z) ∪ P(n)
    if P(z) has changed then
      W = z::W
```

Lazy Cycle Detection

- We keep track of all the edges that have fired a cycle detection.
- In this way, we make sure that we do not try to find the same cycle over and over again, due to edges with the same points-to set.

```
let G = (V, E)
R = {}
W = V
while W ≠ [] do
  n = hd(W)
  for each v ∈ P(n) do
    for each load "a = *n" do
      if (v, a) ∉ E then
        E = E ∪ {(v, a)}
        W = v::W
    for each store "*n = b" do
      if (b, v) ∉ E then
        E = E ∪ {(b, v)}
        W = b::W
  for each (n, z) ∈ E do
    if P(z) = P(n) and (n, z) ∉ R then
      DETECTANDCOLLAPSECYCLES(z)
      R = R ∪ {(n, z)}
    P(z) = P(z) ∪ P(n)
    if P(z) has changed then
      W = z::W
```

Example of Lazy Cycle Detection

c = &d

e = &a

a = b

b = c

c = *e

How is the initial points-to graph for the program on the left?

Example of Lazy Cycle Detection

c = &d

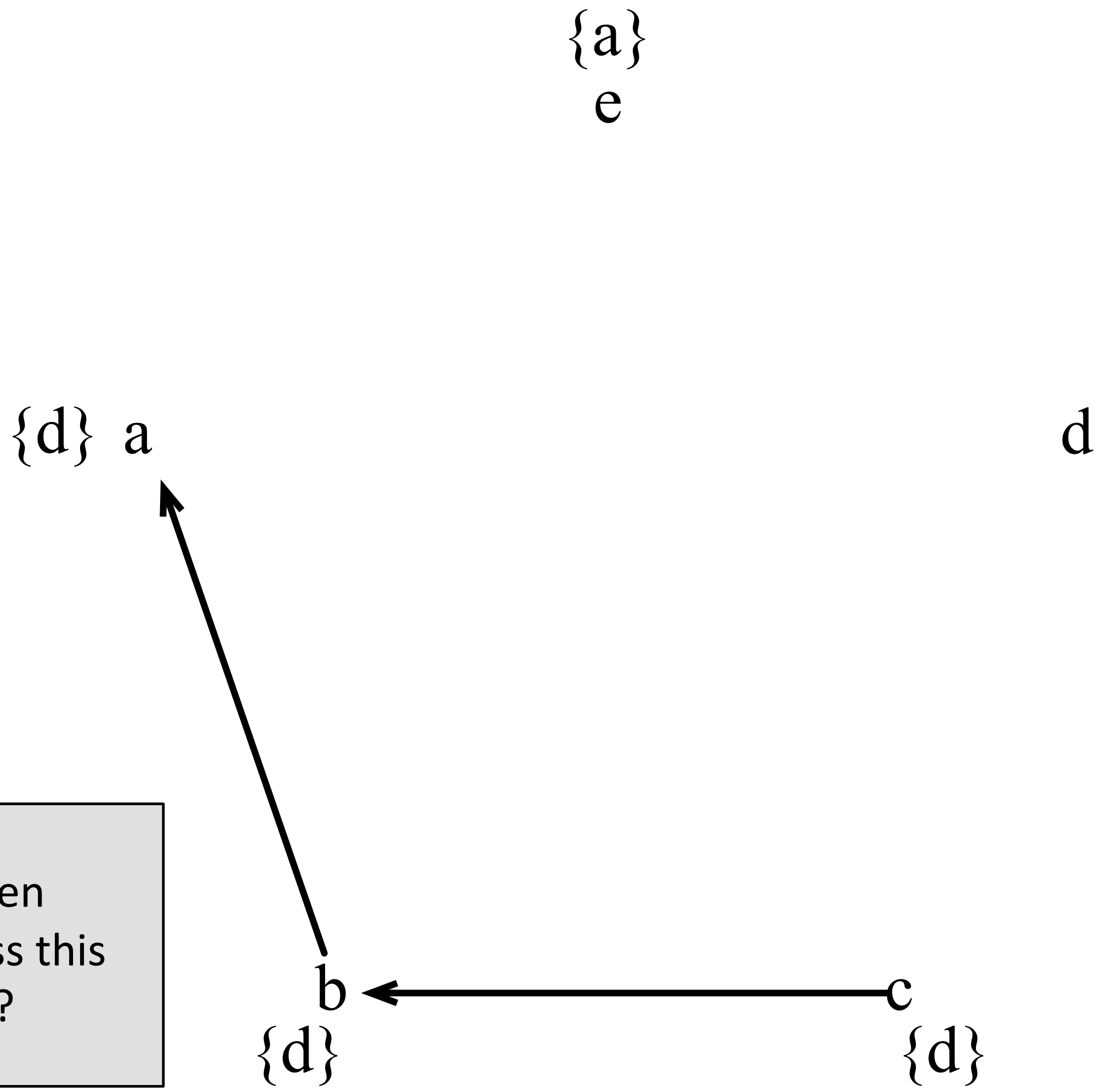
e = &a

a = b

b = c

c = *e

What will happen once we process this load constraint?



Example of Lazy Cycle Detection

c = &d

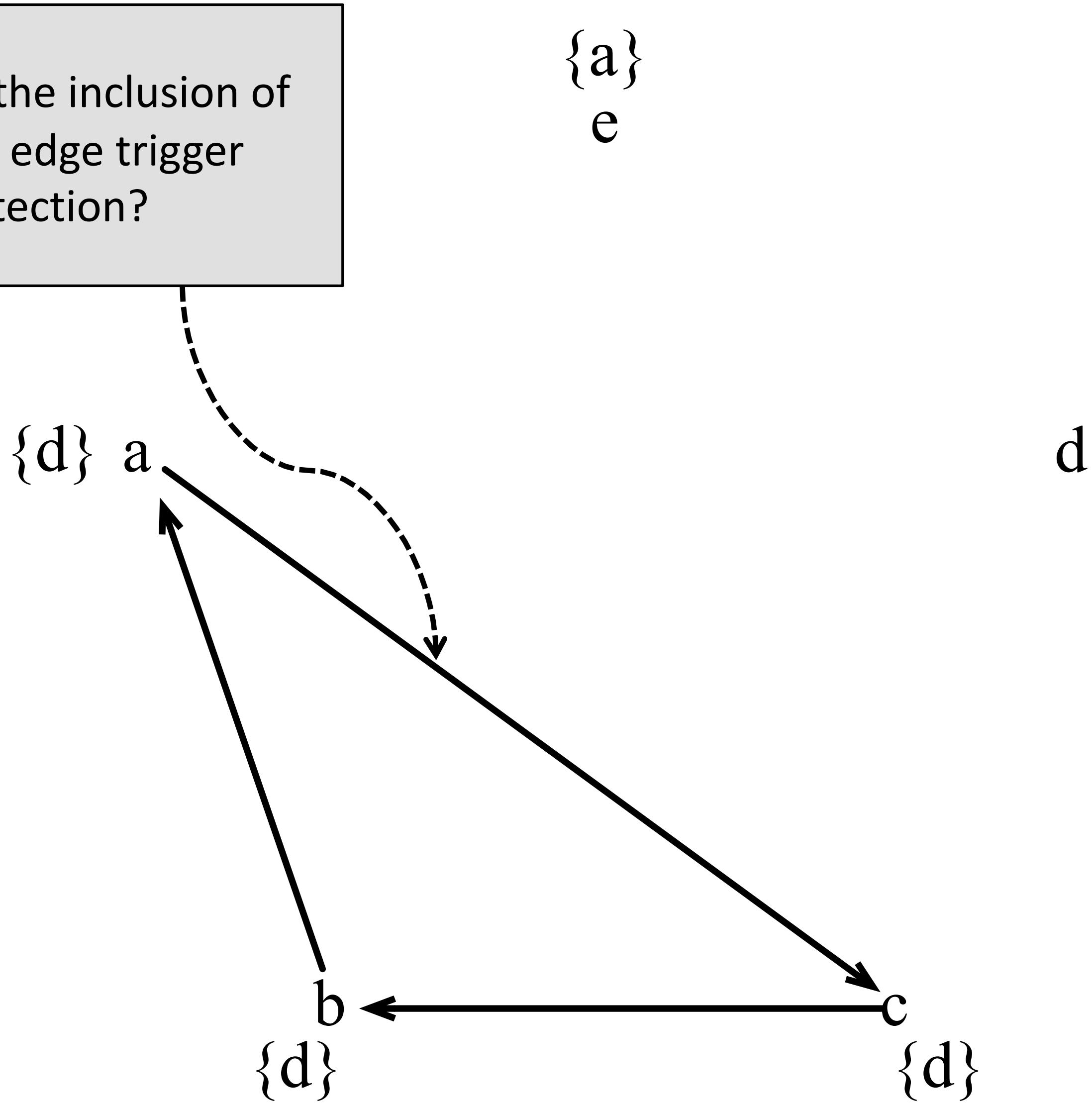
e = &a

a = b

b = c

c = *e

1) Does the inclusion of this new edge trigger cycle detection?



2) What do we do once we find a cycle in the constraint graph?

Example of Lazy Cycle Detection

{a}
e

c = &d

e = &a

a = b

{d} a/b/c

d

b = c

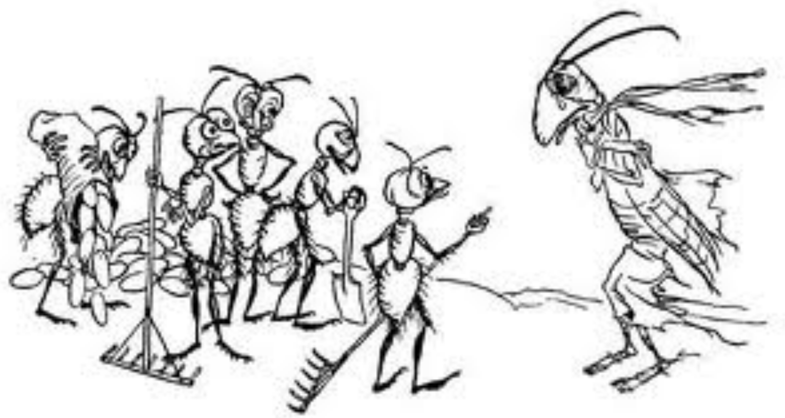
c = *e

After we find a cycle, we collapse it into a single node. In this way, we reduce the quantity of nodes we must handle in every search and update operation of our algorithm.

What are the advantages and the disadvantages of lazy cycle detection?

Tradeoffs

- The main advantage of lazy cycle detection is the fact that it only fires a search for a cycle once the likelihood to find the cycle is high
- Another advantage is that lazy cycle detection is conceptually simple, and it is easy to implement
- The main drawback is that a cycle may remain on a points-to graph for a while until we detect it
- Another problem is that many searches still fail
  - The exact proportion of searches that fail has not been reported in the literature as of today.



Wave Propagation

- Wave propagation is another algorithm used to solve points-to analysis
- It relies on a fast technique[♤] to find strongly connected components in the constraint graph
- Once these components are found and collapsed, the algorithm propagates points to facts in waves, following a topological ordering of the newly modified graph
- This pattern continues, until the points-to graph stops changing.

Could you write this descriptions as pseudo-code?

♤: On Finding the Strongly Connected Components in a Directed Graph (1994)

Wave Propagation

repeat
  changed = false
  collapse Strongly Connected Components
  WAVEPROPAGATION
  ADDNEWEDGES
  if a new edge has been added to G **then**
    changed = true
until changed = false

1) What do you think WAVEPROPAGATION does?

2) What about the ADDNEWEDGES algorithm?

Wave Propagation of Points-to Information

WAVEPROPAGATION(G, P, T)
while $T \neq []$
  $v = hd(T)$
  $P_{dif} = P_{cur}(v) - P_{old}(v)$
  $P_{old}(v) = P_{cur}(v)$
  if $P_{dif} \neq \{\}$
    for each w such that $(v, w) \in E$ **do**
      $P_{cur}(w) = P_{cur}(w) \cup P_{dif}$

Each node is associated with two sets, P_{cur} and P_{old}. The first denotes the current points-to facts that we know about the node. The second denotes the points-to fact that we knew in the last iteration of our algorithm.

The parameters of the algorithm are:

- G: the points-to graph
- P: the points-to facts
- T: the topological ordering of the nodes in G

The Creation of New Edges

$\text{ADDNEWEDGES}(G = (E, V), C)$

for each operation c such as $l = *r \in C$ **do**

$P_{new} = P_{cur}(r) - P_{cache}(c)$

$P_{cache}(c) = P_{cache}(c) \cup P_{new}$

for each $v \in P_{new}$ **do**

if $(v, l) \notin E$ **then**

$E = E \cup \{(v, l)\}$

$\boxed{P_{cur}(l) = P_{cur}(l) \cup P_{old}(v)}$

for each operation c such as $*l = r$ **do**

$P_{new} = P_{cur}(l) - P_{cache}(c)$

$P_{cache}(c) = P_{cache}(c) \cup P_{new}$

for each $v \in P_{new}$ **do**

if $(r, v) \notin E$ **then**

$E = E \cup \{(r, v)\}$

$\boxed{P_{cur}(v) = P_{cur}(v) \cup P_{old}(r)}$

We keep track of $P_{cache}(c)$, the last collection of points used in the evaluation of the complex constraint c. This optimization reduces the number of edges that must be checked for inclusion in G. $P_{cache}(c)$ is initially set to {}.

These updates will set up the ground for the next iteration of the wave propagation.

Wave Propagation: an example

- Let's illustrate the wave propagation algorithm with the following set of statements:

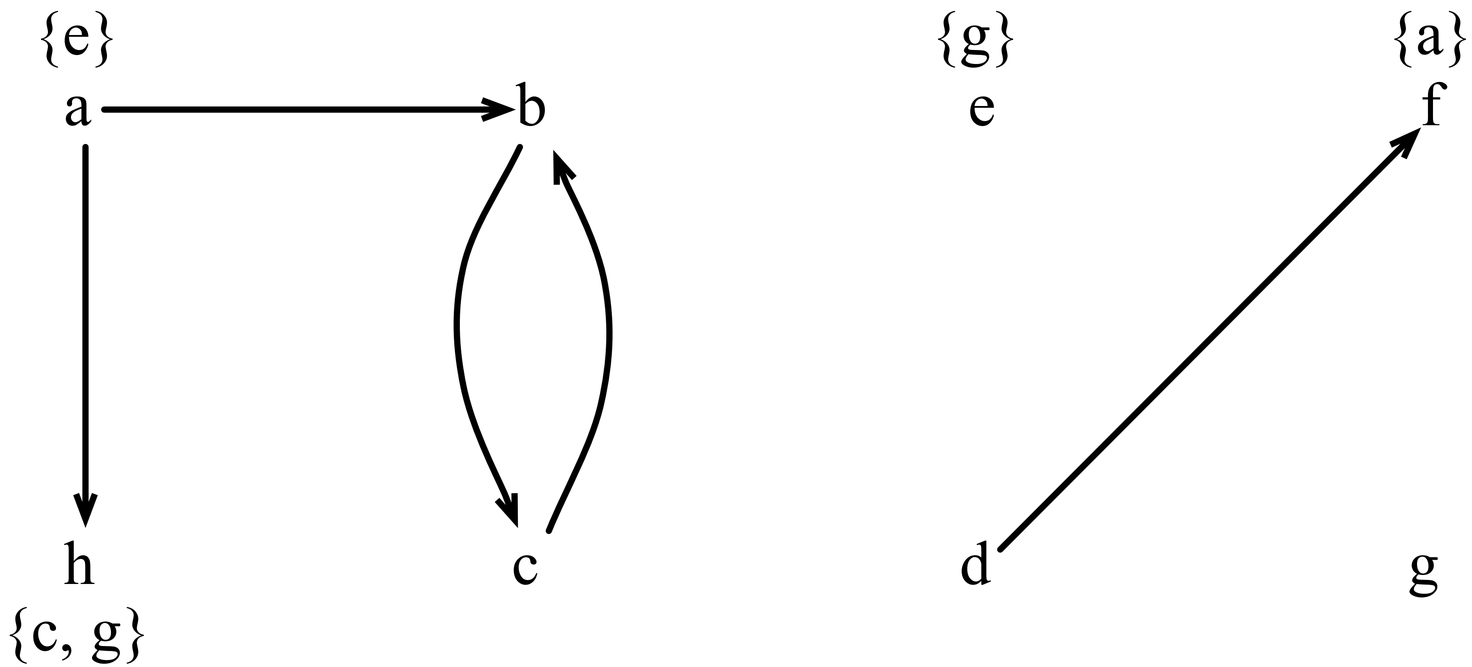
```
h = &c      e = &g      b = c
h = &g      h = a       c = b
a = &e      f = d       b = a
d = *h      *e = f      f = &a
```

What is the initial points-to graph for this program?

Answer explaining how many nodes, and how many edges this graph will have.

Wave Propagation: an example

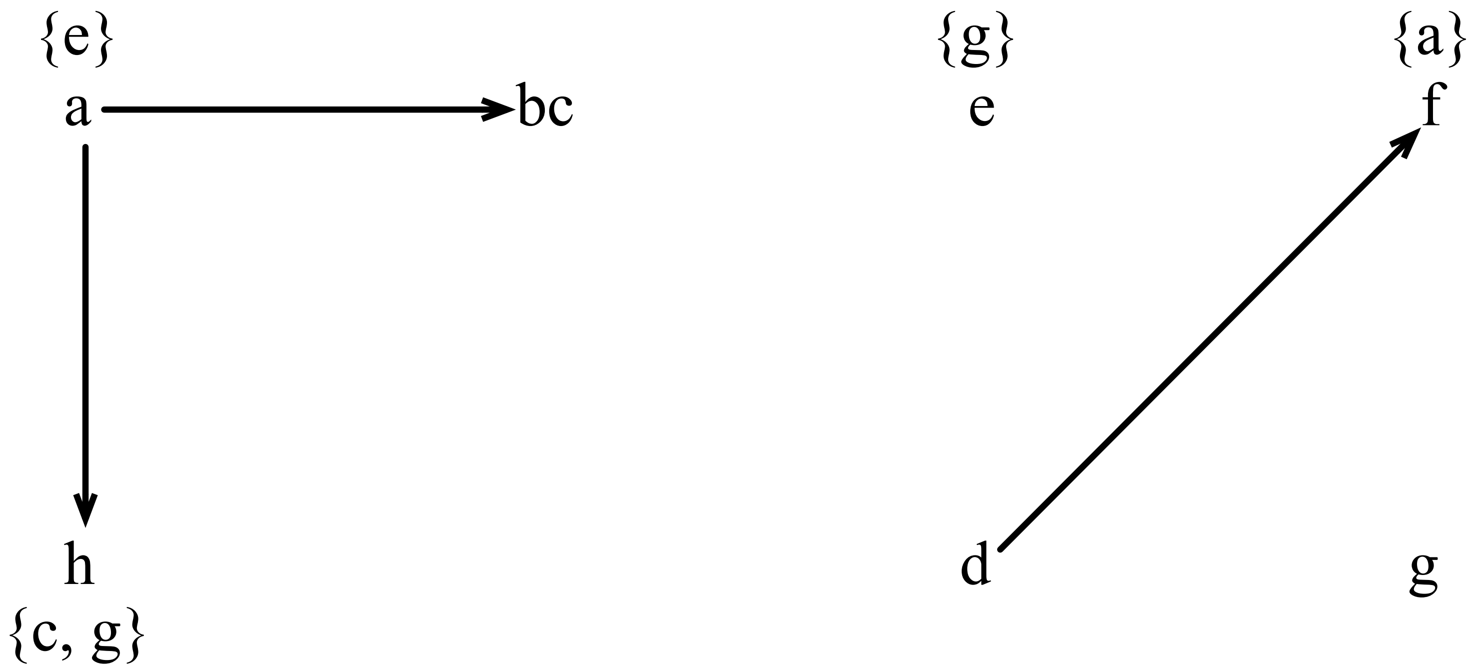
```
h = &c        e = &g        b = c        d = *h
h = &g        h = a         c = b        f = &a
a = &e        f = d         b = a        *e = f
```



How will be the graph after we merge SCCs?

Wave Propagation: an example

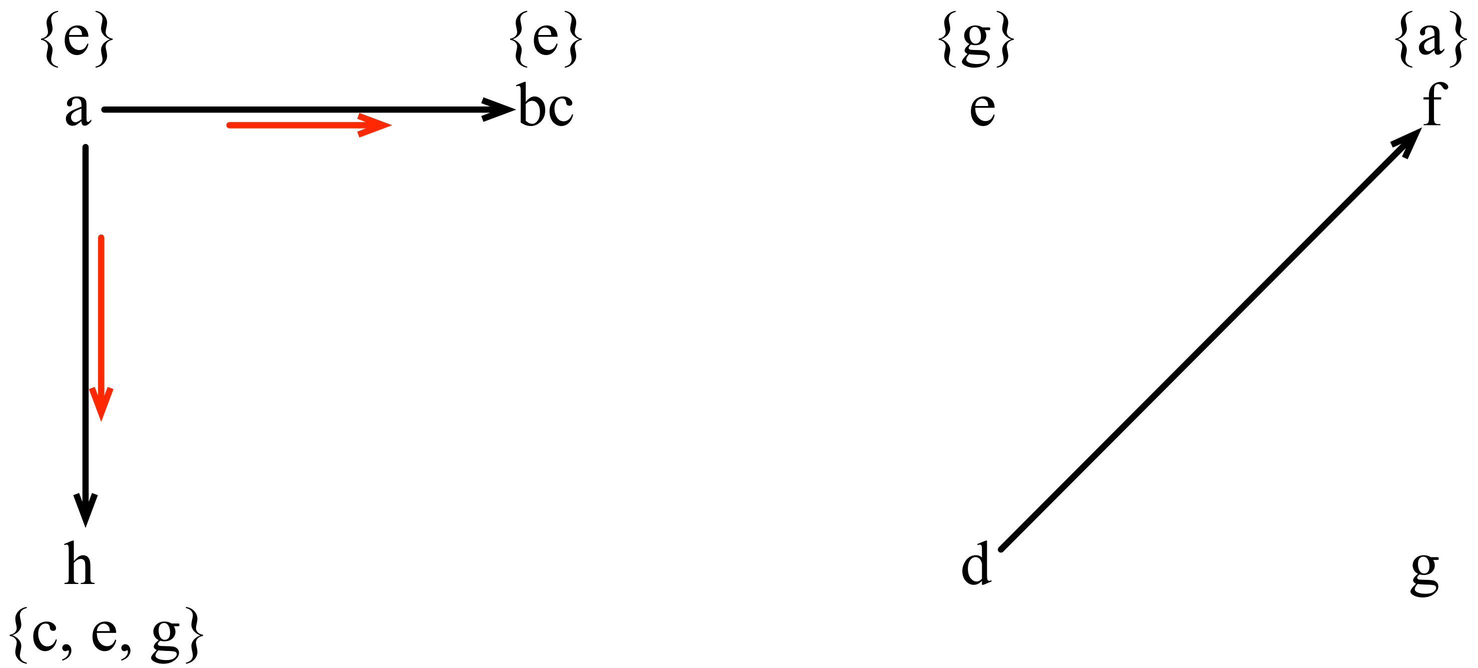
```
h = &c      e = &g      b = c      d = *h
h = &g      h = a       c = b      f = &a
a = &e      f = d       b = a      *e = f
```



How will be the graph after the first wave propagation?

Wave Propagation: an example

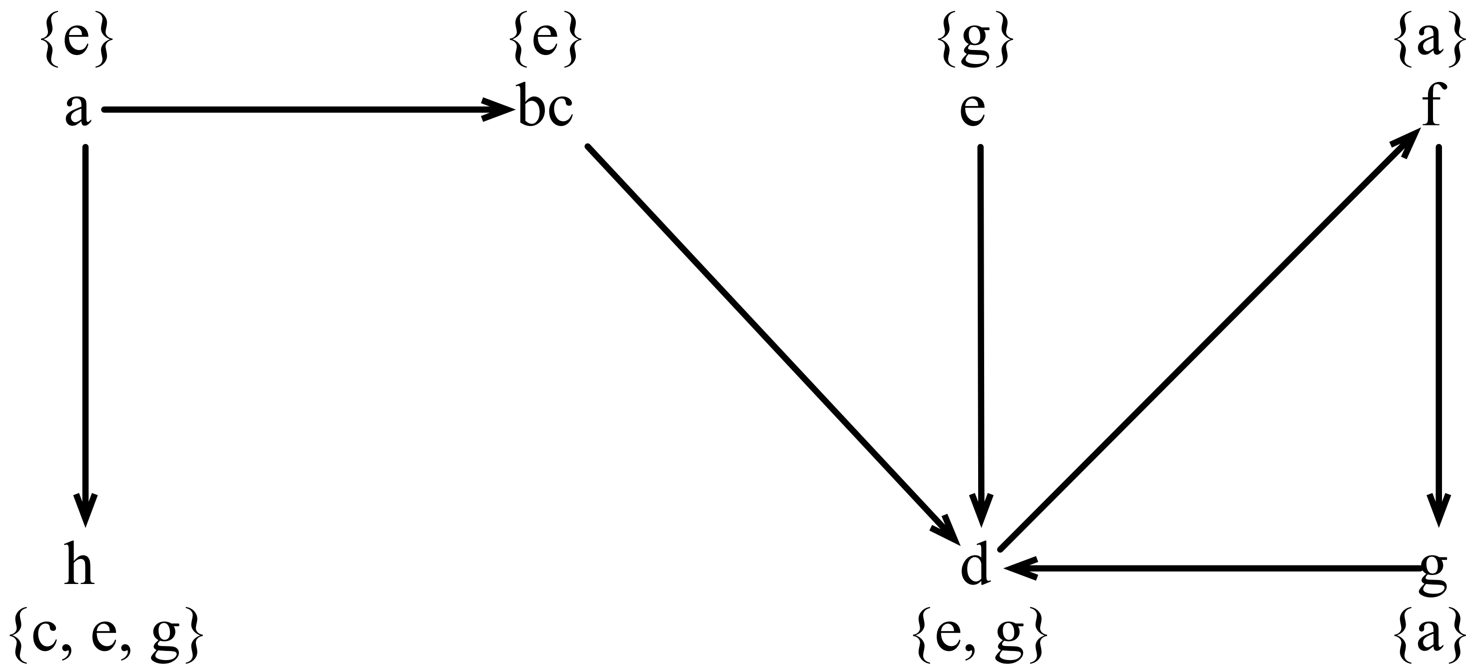
```
h = &c        e = &g        b = c        d = *h
h = &g        h = a         c = b        f = &a
a = &e        f = d         b = a        *e = f
```



How will be the graph after we add the new edges? Which constraints do we have to consider now?

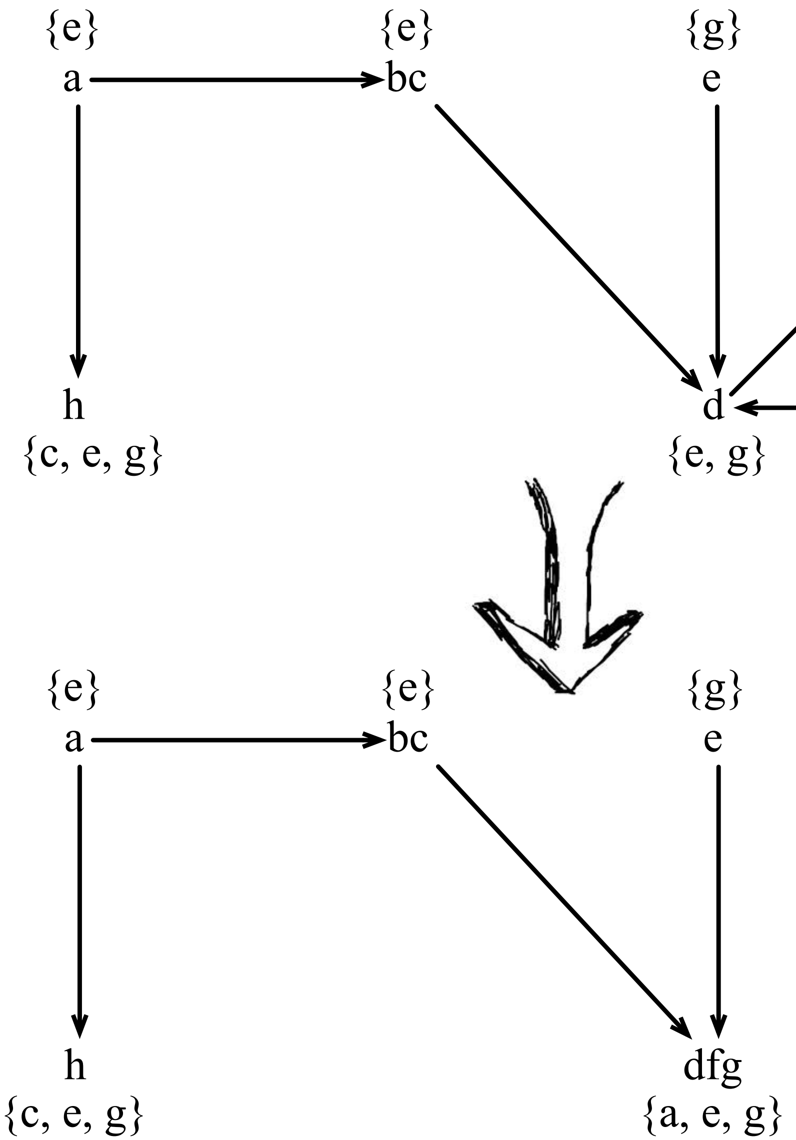
Wave Propagation: an example

```
h = &c        e = &g        b = c        d = *h
h = &g        h = a         c = b        f = &a
a = &e        f = d         b = a        *e = f
```



Can you finish the iterations of the algorithm in this graph?

Wave Propagation: an example

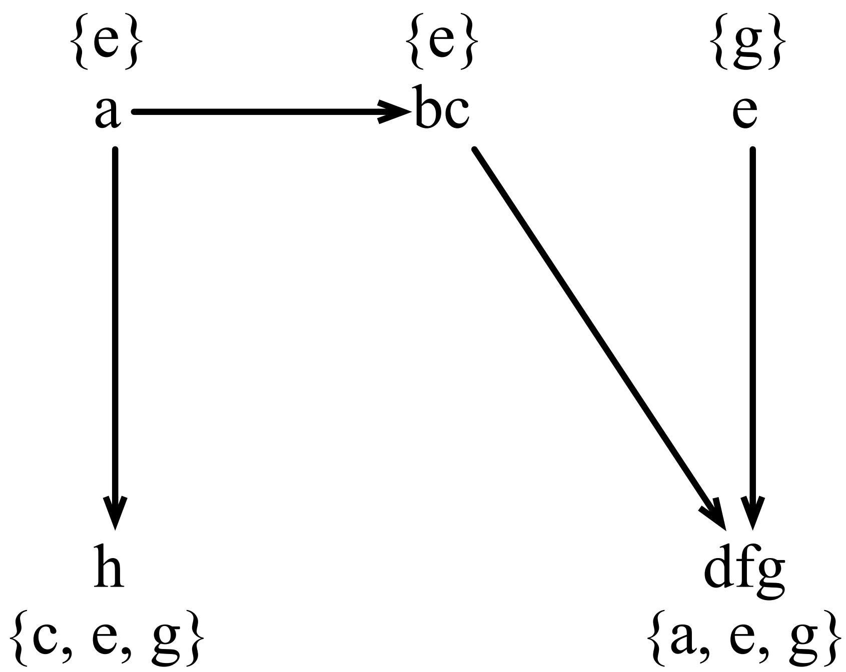


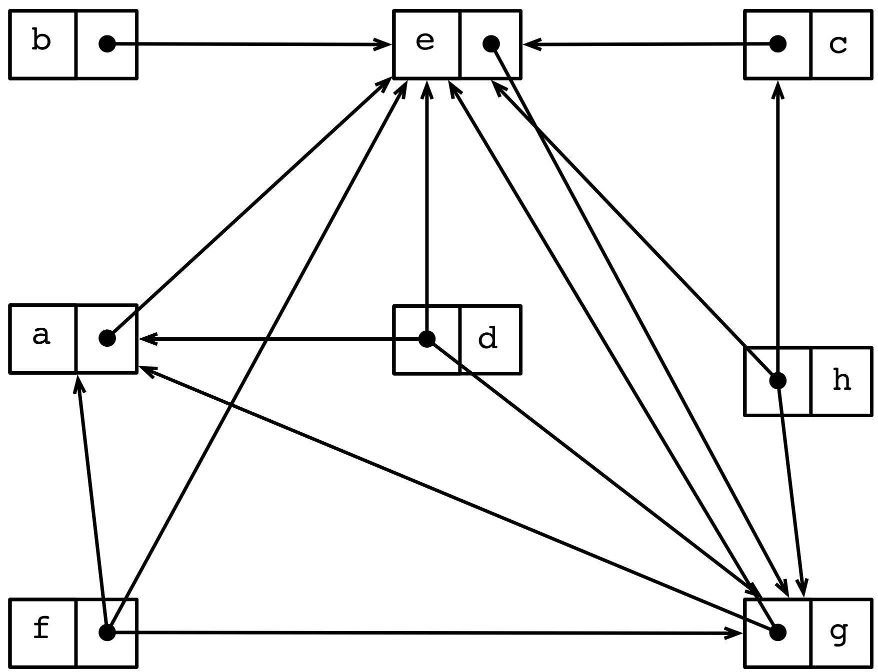
What is the complexity of the wave propagation solver? What about the lazy cycle detector?

After we collapse SCCs, the sequence of wave propagations causes no more changes, and no further edge is added to the graph. We have reached a fixed point, and we can stop the algorithm.

The Shape of the Heap

- From the result of the points-to analysis we can infer the shape that the heap may assume after the program executes.





Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

STEENSGAARD'S ANALYSIS

DCC 888

Equality vs Subset Inclusion

- In the previous approach to pointer analysis, e.g., Andersen Style, a statement like p = q means that everything that is pointed by q could be also pointed by p.

$$\boxed{P(p) \supseteq P(q)}$$

- We can speed up this analysis, by unifying both sides, instead of using the subset relation.

$$\boxed{P(p) = P(q)}$$

- This way of doing pointer analysis, e.g., based on the unification of both sides of an assignment, is called Steensgaard's analysis, after the work of *Bjarne Steensgard*♤.

♤: Points-to Analysis in Almost Linear Time (1995)

Steensgaard's Analysis at Work

- We shall demonstrate how the unification based analysis works with an example:

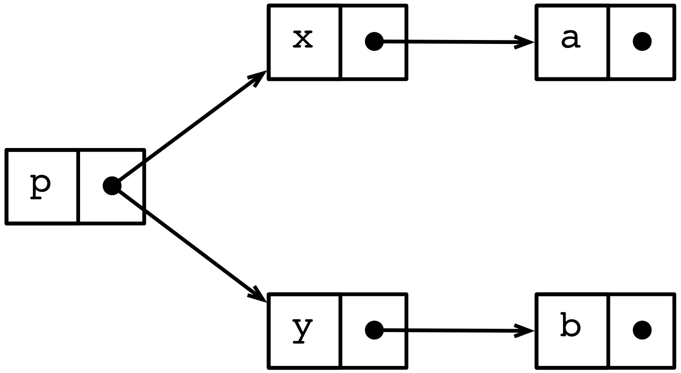
```
x = &a;
y = &b;
p = &x;
p = &y;
```

What is the shape of the heap, as reported by Andersen's analysis, after this program runs?

Steensgaard's Analysis at Work

- We shall demonstrate how the unification based analysis works with an example:

```
x = &a;
y = &b;
p = &x;
p = &y;
```



- Steensgaard's analysis proceed by joining into equivalence classes elements that appear together in assignments.
- An assignment like x = &a means that '*x' and 'a' are in the same equivalence class, which is pointed by x's class.

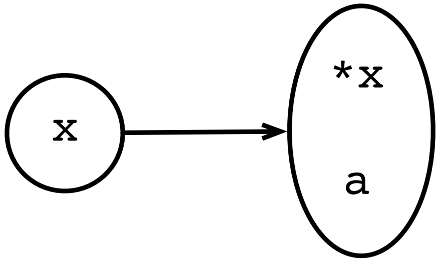
Steensgaard's Analysis at Work

```
x = &a;

y = &b;

p = &x;

p = &y;
```



What will be the effect of **this** assignment?

An assignment like `x = &a` means that `*x` and a are in the same equivalence class, which is pointed by `x`'s class.

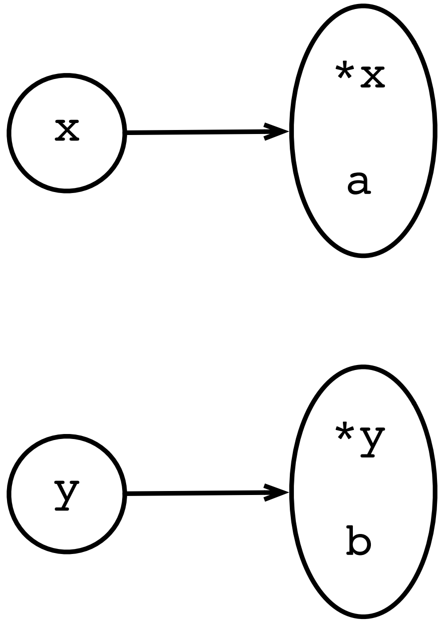
Steensgaard's Analysis at Work

```
x = &a;

y = &b;

p = &x;

p = &y;
```



What will be the effect of **the third** assignment?

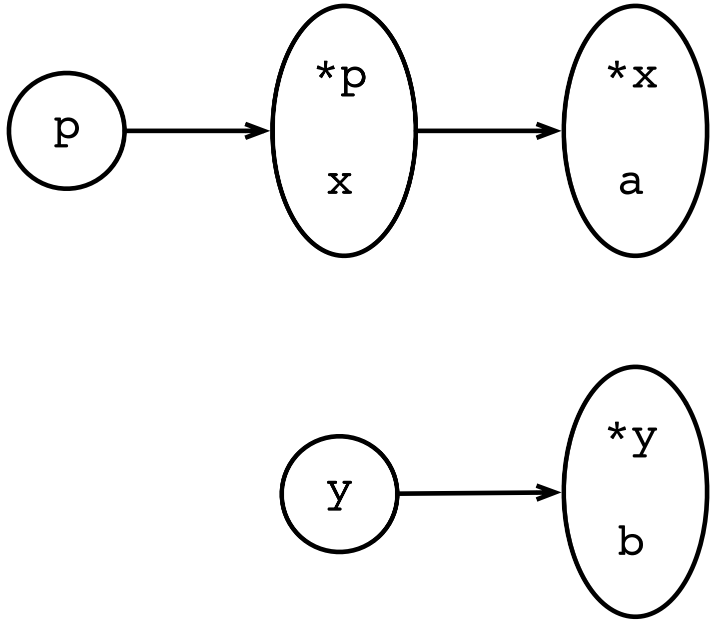
Steensgaard's Analysis at Work

```
x = &a;

y = &b;

p = &x;

p = &y;
```



And what will happen after the last statement is analyzed?

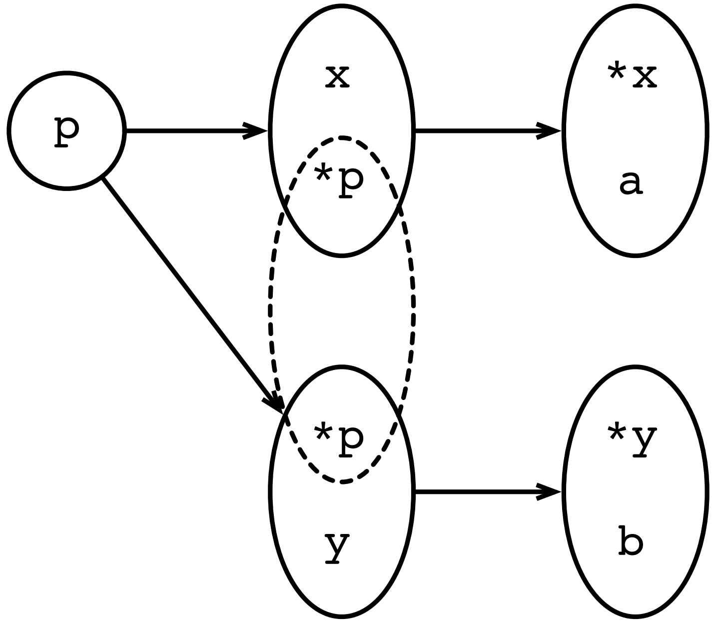
Steensgaard's Analysis at Work

```
x = &a;

y = &b;

p = &x;

p = &y;
```



How fast can we perform **this unification**?

After the assignment `p = &y`, we have that `*p` appears in two different classes. We must unify them.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

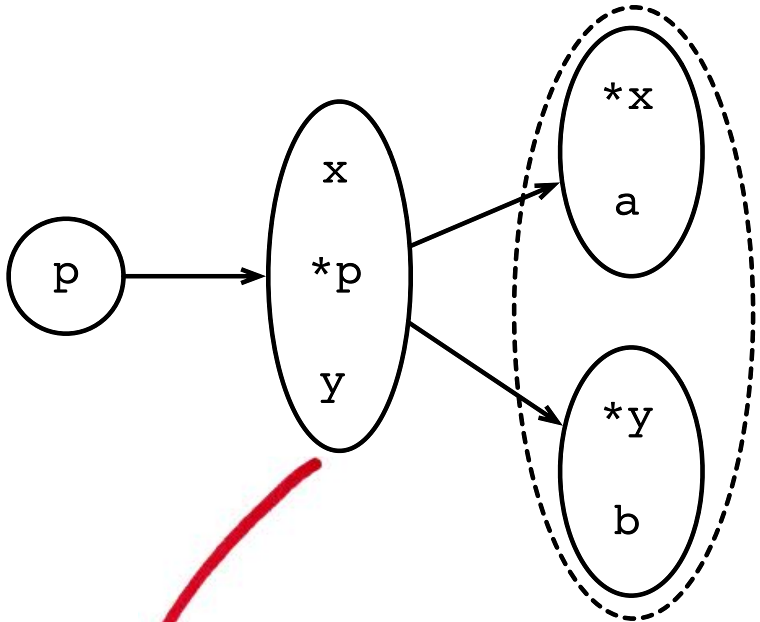
Steensgaard's Analysis at Work

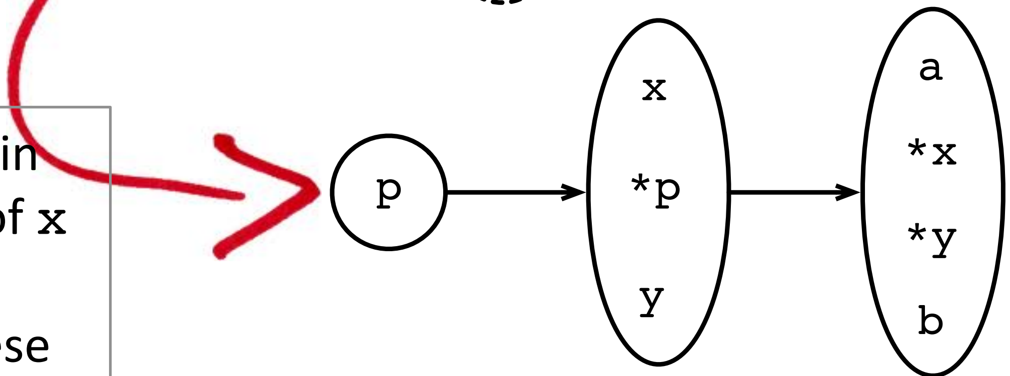
```
x = &a;

y = &b;

p = &x;

p = &y;
```



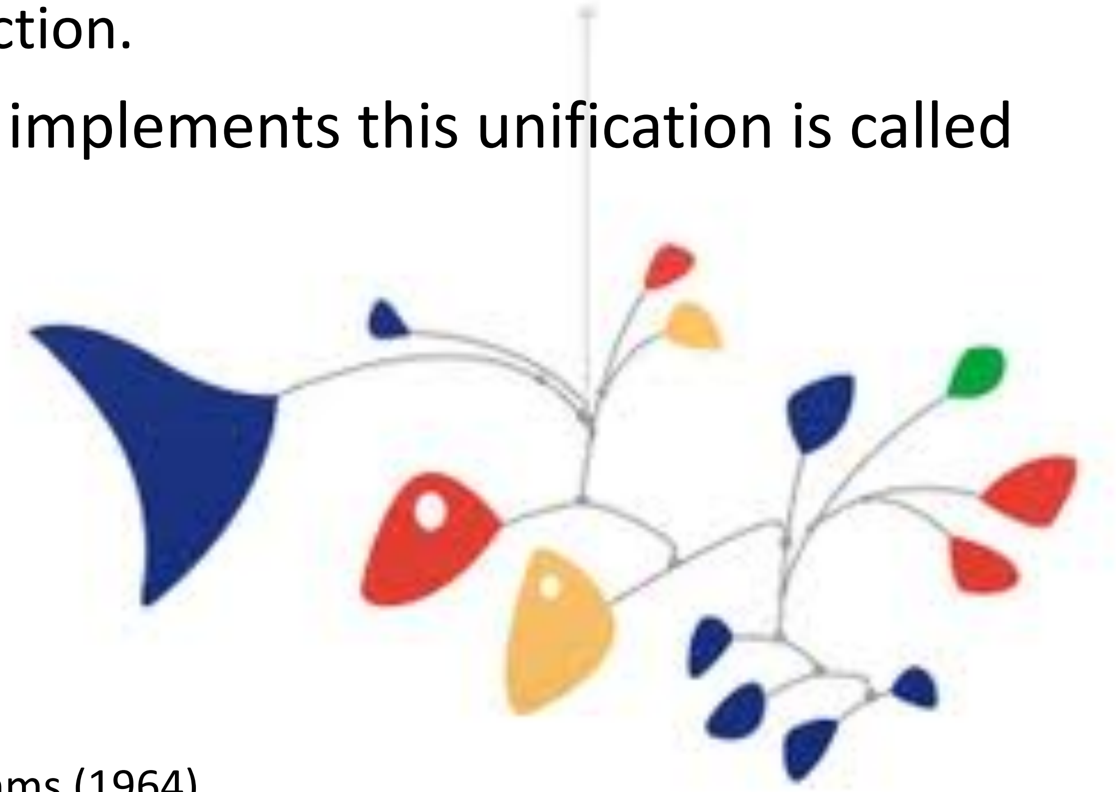


But the process does not stop in this unification of the classes of `x` and `y`. It must be propagated throughout the children of these two classes, until we have only one line of pointers.

Union-Find♡

- This process of chain unifications has a fast implementation.
  - It can be implemented to run in $\alpha(n)$, where n is the number of elements to be unified, and α is the inverse Ackermann's function.
- The algorithm that implements this unification is called **union-find**.

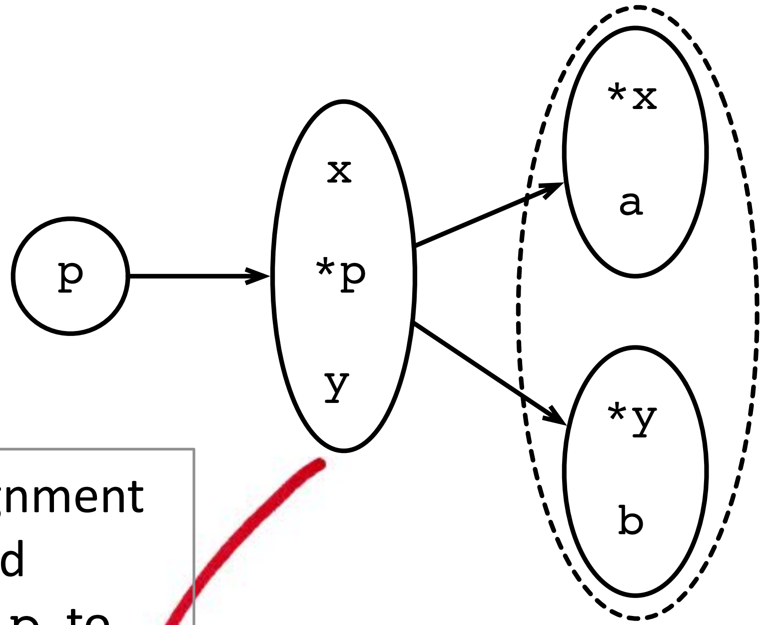
By the way, why are these chains of unifications necessary?



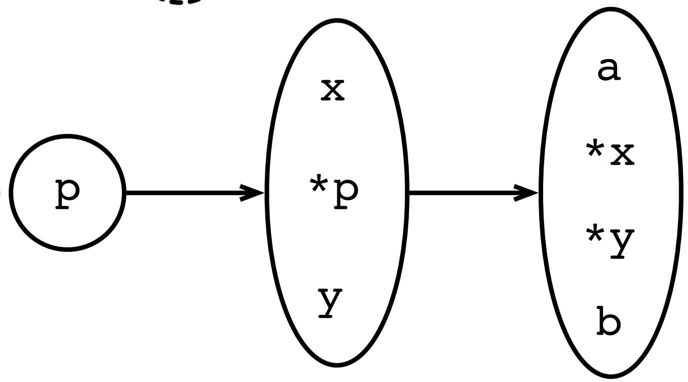
♡: An Improved Equivalence Algorithms (1964)

Why are Chains of Unification Necessary?

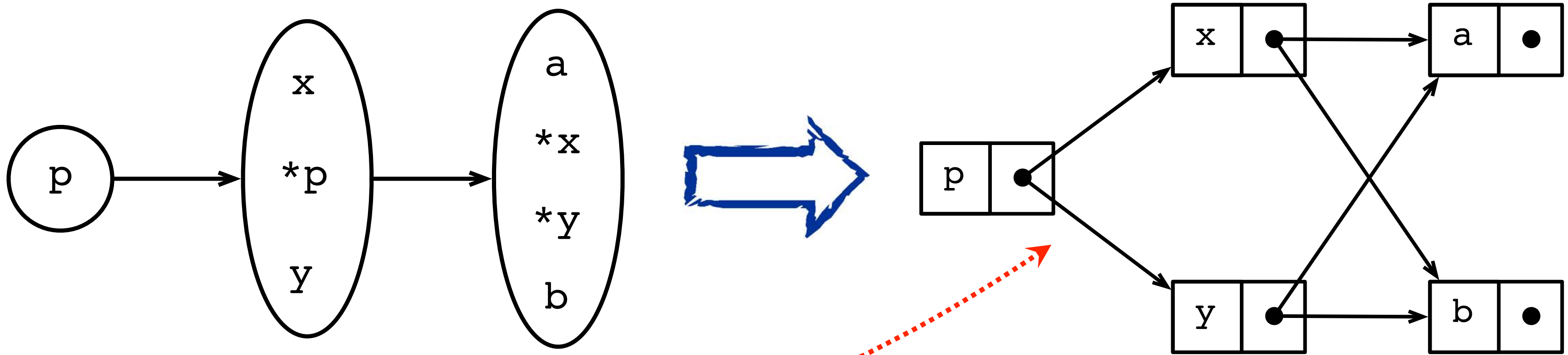
What is the shape of the heap that we have inferred with this analysis?



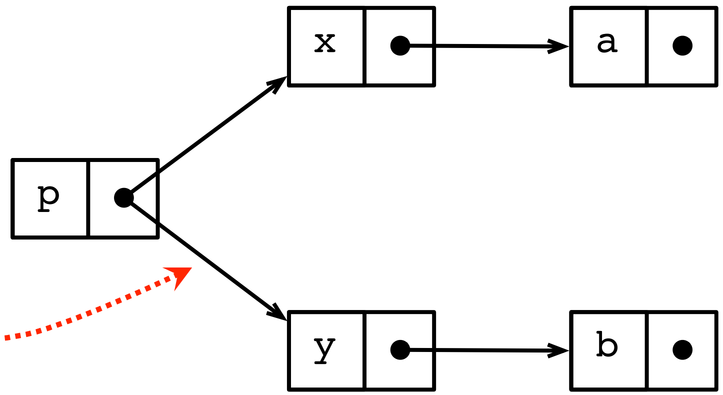
Imagine that we had an assignment like `*p = q`. We need to find everything that is pointed by `p`, to unify it with `q`. If we had kept the equivalence classes of `a` and `b` separated, we would have to update them separately as well. Because they are the same, we can update them together, in a single pass.



Steensgaard's is less precise than Andersen's



Steensgaard's analysis is very fast; however, this speed pays a price in terms of precision. In this case, we have found out that x can point to either a or b. Same thing for y. **Andersen**'s analysis would tell us that x can only point to a, and y can only point to b.



A Common Pattern

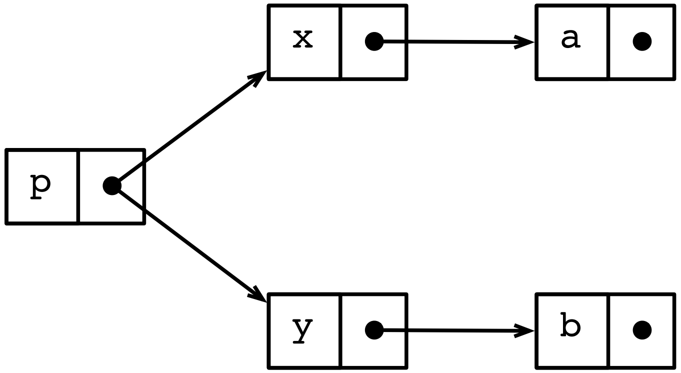
- All the algorithms that we have seen so far, follow a very common pattern: iterate until a fixed point is reached.
  - If no changes have been found in one iteration, no more changes will ever happen.
- This pattern happens in several different analyses:
  - All the data-flow analyses
  - The control flow analysis
  - The points-to analysis

Can you provide some intuition on why the different points-to solvers are guaranteed to terminate?

Flow Sensitiveness

- Even Andersen's analysis is not very precise, as it is flow insensitive.

```
1) x = &a;
2) y = &b;
3) p = &x;
4) p = &y;
```



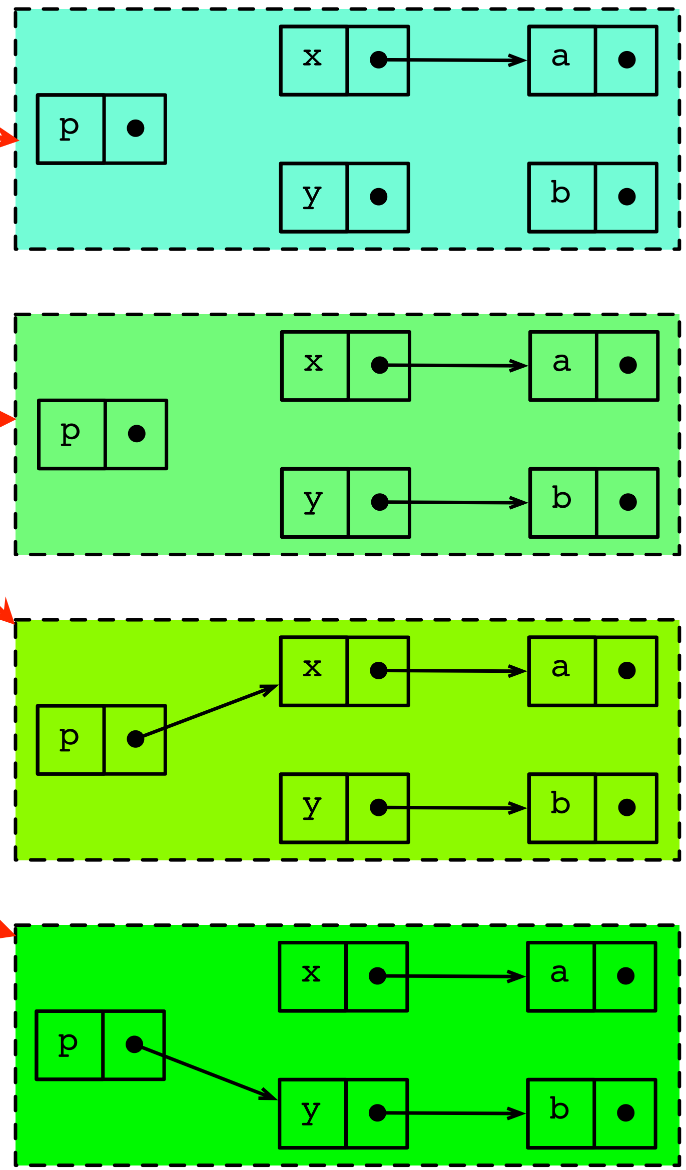
The points-to graph represents any possible edge that can exist at any moment during the execution of the program. Some of these edges, of course, cannot exist at every program point. Some of them cannot even exist at the same time.

Which edges can exist after instruction 1 executes? What about after instructions 2, 3 and 4?

Flow Sensitiveness

What is the complexity of the size of the output of shape analysis?

```
1) x = &a;
2) y = &b;
3) p = &x;
4) p = &y;
```



There are analyses that are strong enough to track the shape of the points-to graph at each program point. One such analysis is "Shape analysis"♡. You can imagine that it is pretty costly…

♡: Parametric shape analysis via 3-valued logic, 2002

A Bit of History

- Inclusion-based points-to analysis has been described by Lars Andersen in his PhD dissertation.
- Unification-based points-to analysis was an idea due to Bjarne Steensgaard, in the mid 90's.
- Lazy cycle detection was invented by Ben Hardekopf in 2007.
- Wave propagation was designed by Pereira and Berlin in 2011.

- Andersen, L. "Program Analysis and Specialization for the C Programming Language", PhD Thesis, University of Copenhagen, (1994)
- Hardekopf, B. and Lin, C. "The Ant and the Grasshopper: fast and accurate pointer analysis for millions of lines of code", PLDI, pp 290-299 (2007)
- Pereira, F. and Berlin, D. "Wave Propagation and Deep Propagation for Pointer Analysis", CGO, pp 126-135 (2009)
- Steensgaard, B., "Points-to Analysis in Almost Linear Time", POPL, pp 32-41 (1995)