EXERCISE 20

CALL TARGET ANALYSIS REVIEW

Write your name and answer the following on a piece of paper

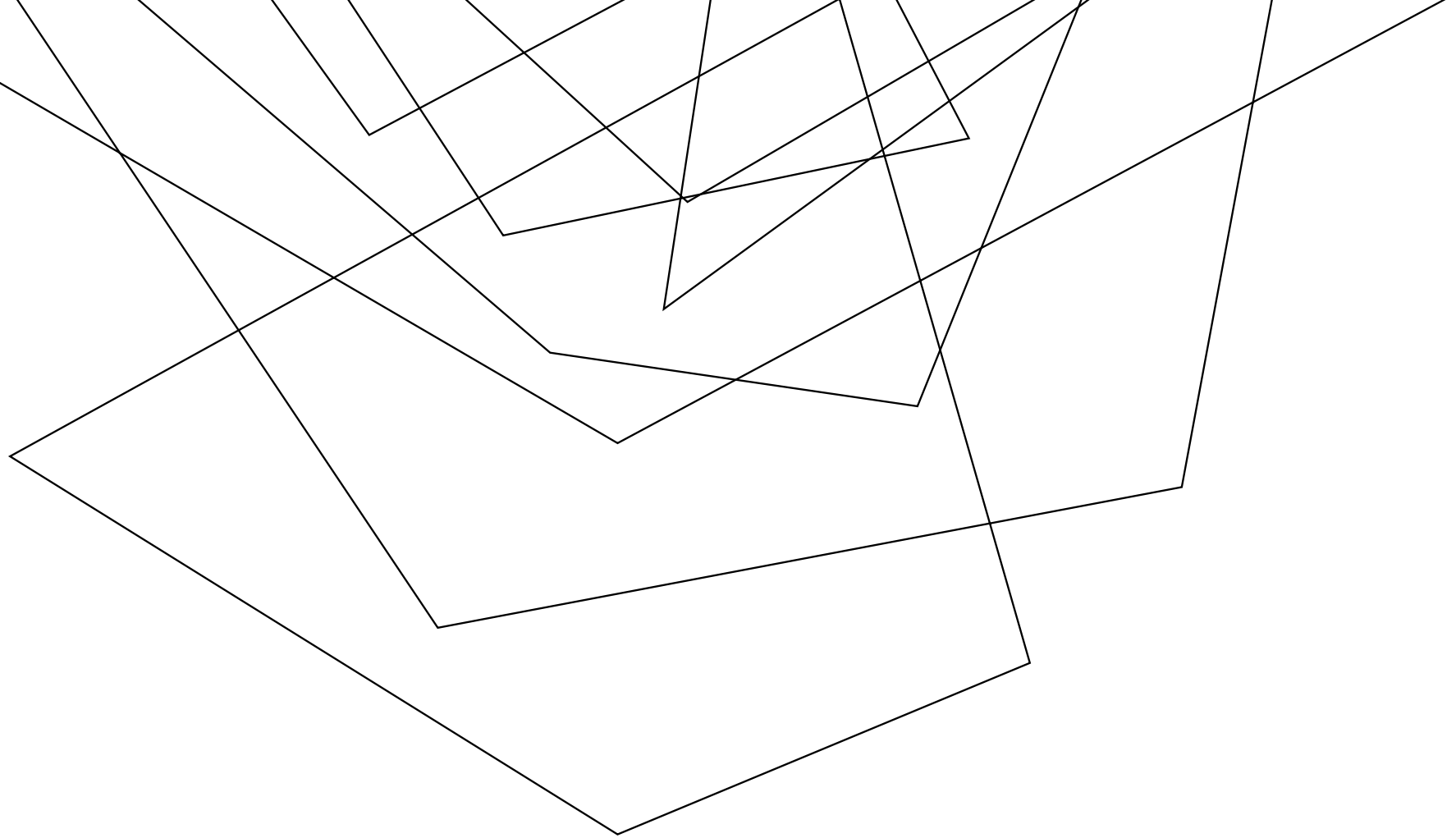
Draw the call graph of the following program according to CHA

```
 1: class SupClass{
 2: public:
 3:     virtual int fun(SupClass * in){
 4:         in->fun();
 5:     }
 6: };
 7:
 8: class SubA : public SupClass {
 9:     virtual int fun(SupClass * in){
10:         in->fun();
11:     }
12: };
13: class SubB : public SupClass {
14:     virtual int fun(SupClass * in){
15:         in->fun();
16:     }
17: };
18: int main(){
19:     SupClass * s = new SubA();
20:     s->fun();
21: }
```

EXERCISE 20 SOLUTION

CALL TARGET ANALYSIS REVIEW

ADMINISTRIVIA AND ANNOUNCEMENTS



POINTS-TO ANALYSIS

EECS 677: Software Security Evaluation

Drew Davidson

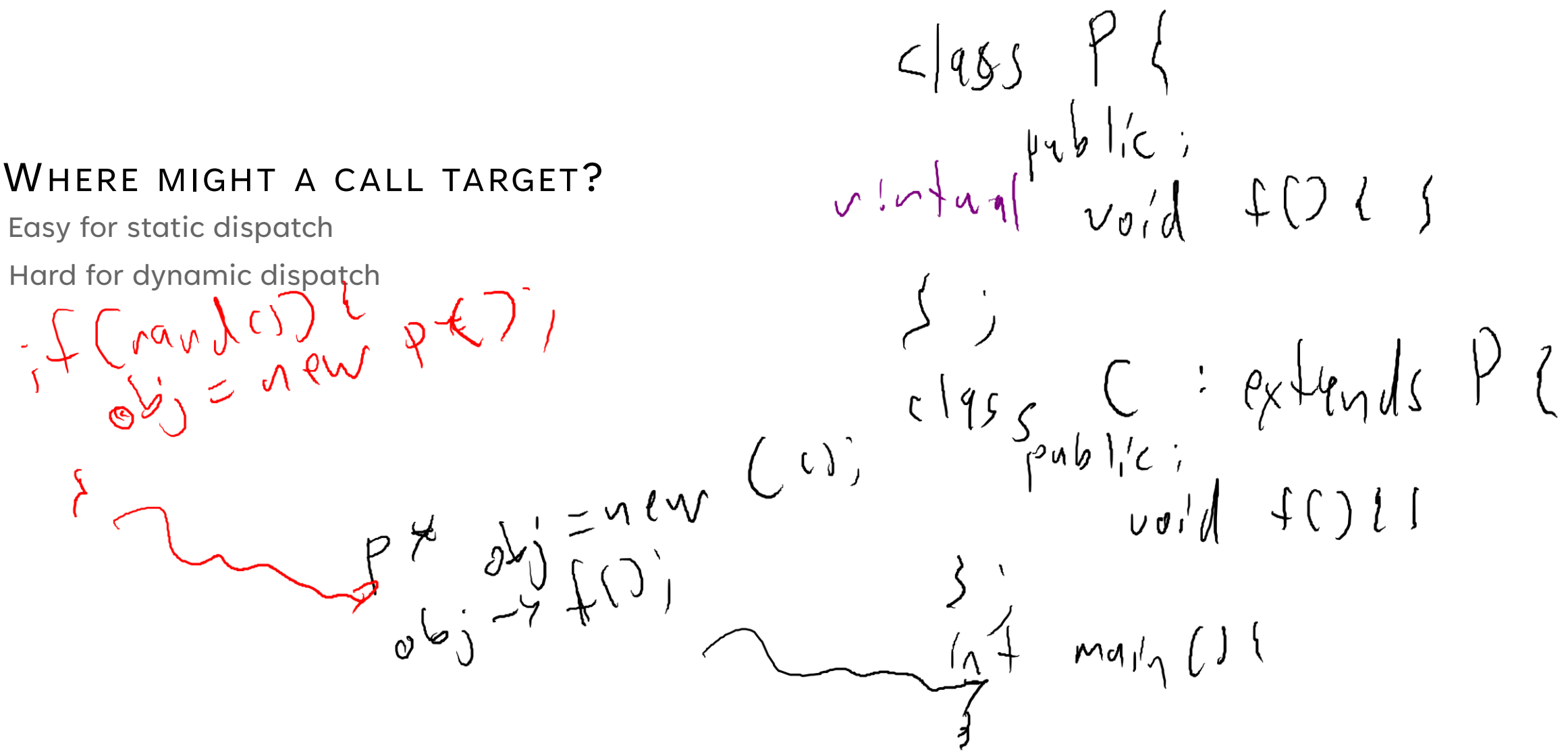
LAST TIME: CALL RESOLUTION ANALYSIS

REVIEW: CALL TARGETS

WHERE MIGHT A CALL TARGET?

Easy for static dispatch

Hard for dynamic dispatch

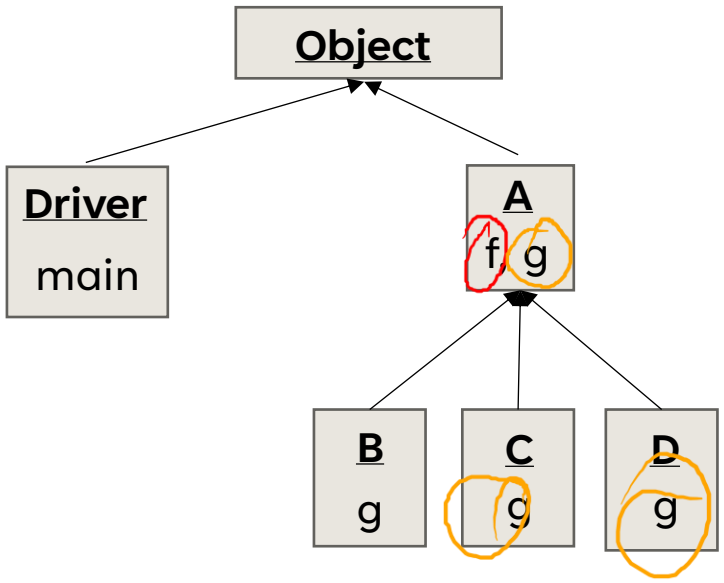


LAST TIME: CALL RESOLUTION ANALYSIS

REVIEW: CALL TARGETS

Class Hierarchy Analysis

Inheritance implies a constraint over call targets

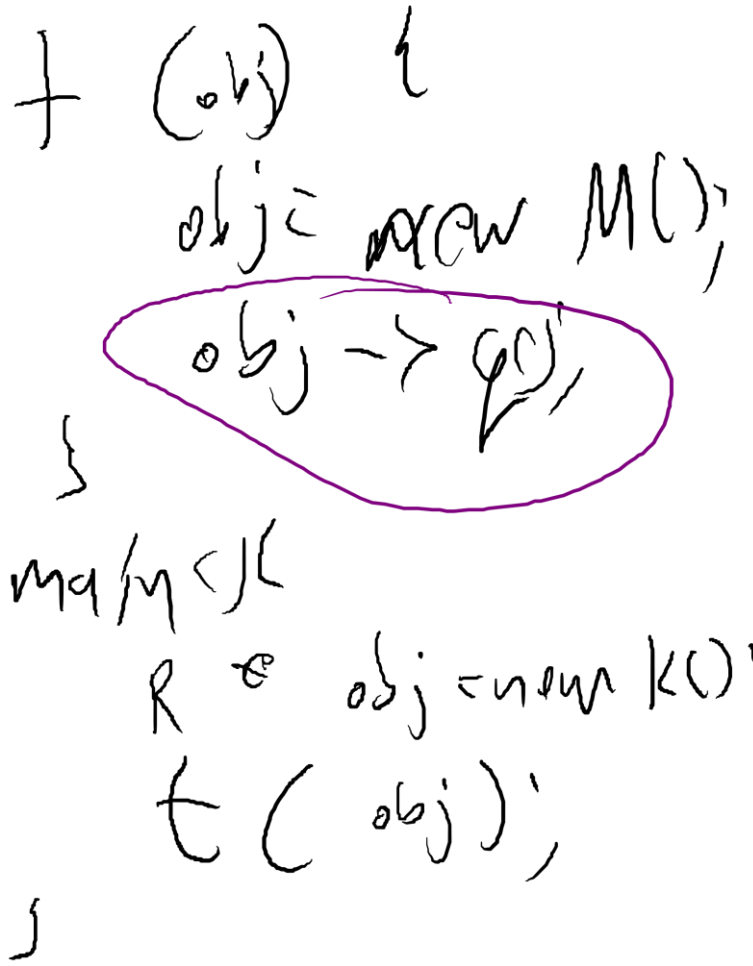



```
 1: class A{
 2:     public A f(){ return new C(); }
 3:     public String g(){ return "A"; }
 4: };
 5: class B : public A{
 6:     public String g(){ return "B"; }
 7: };
 8: class C : public A{
 9:     public String g(){ return "C"; }
10: };
11: class D : public A{
12:     public String g(){ return "D"; }
13: };
14: class Driver {
15:   public void main(String[] args){
16:     A[] aArr = {new A(), new B()};
17:     for (A a : aArr){
18:       A res = a.f();
19:       print(a.g());
20:       print(res.g());
21:     }
22:   }
23: };
```

RAPID TYPE ANALYSIS (RTA)

A HISTORY OF COMPUTING

```
RTA = call graph of functions (initially edgeless)
CHA = call graph via class hierarchy analysis
W = worklist
W.push(main)
while not W.empty:
  M = pop W
  T = allocated types in M
  T = T ∪ allocated types in RTA callers of M
  foreach callsite(C) in M:
    if C is statically-dispatched:
      add edge C to C's static target
    else:
      M' = methods called from M in CHA
      M' = M' ∩ functions declared in T or T-supertypes
      add edge from M to each M'
      W.pushAll(M')
```



RAPID TYPE ANALYSIS (RTA)

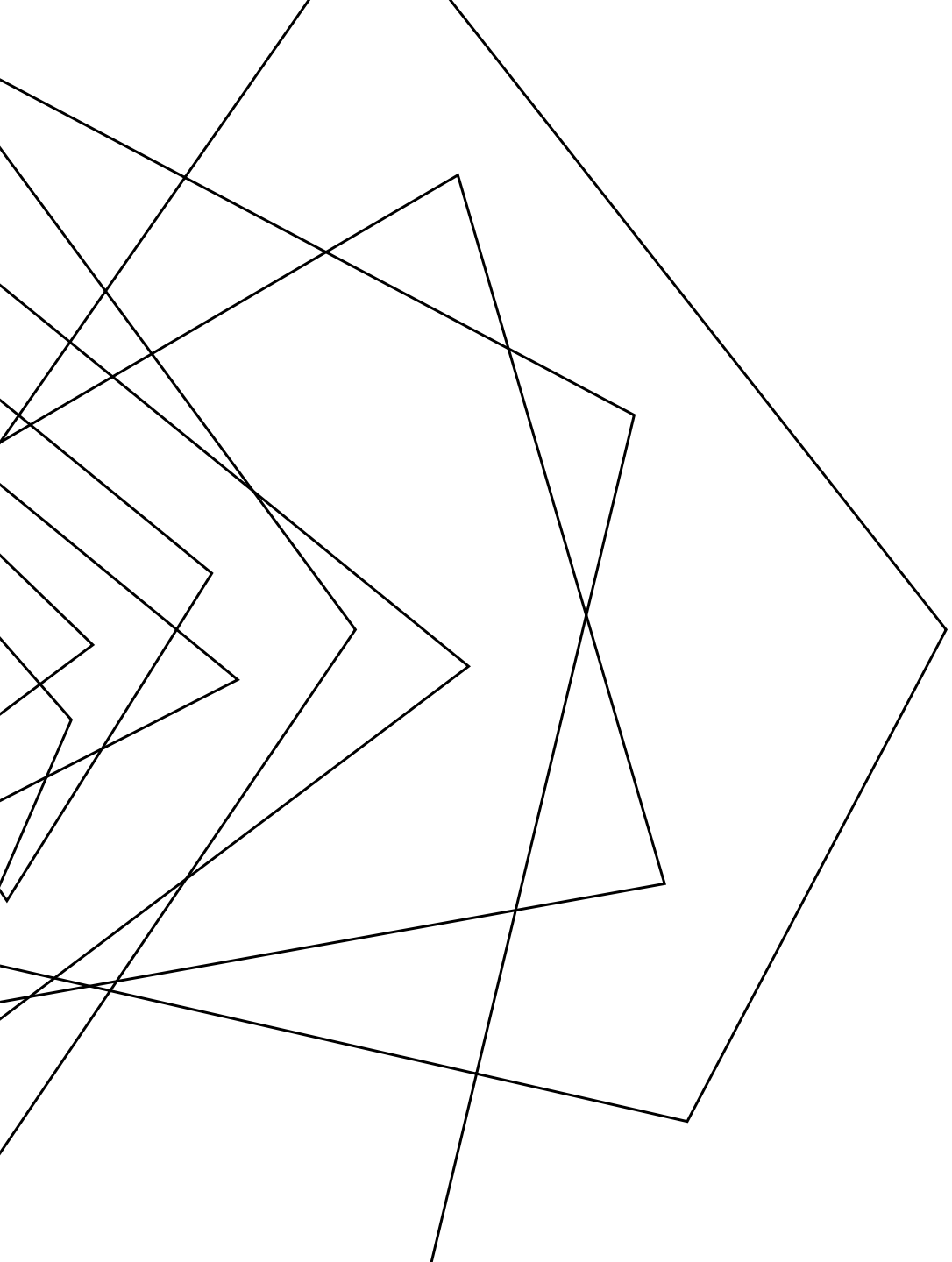
A HISTORY OF COMPUTING

AN INCOMPLETE ANALYSIS!

```
 1: public static Object gbl;
 2:
 3: public static void main(String[] args){
 4:   foo();
 5:   bar();
 6: }
 7:
 8: public static void foo(){
 9:   Object o = new A();
10:   gbl = o;
11: }
12:
13: public static void bar(){
14:   gbl.toString();
15: }
```

Call edge to A's toString missing!
Neither bar or its callers (main) allocated a type of A

RTA will not include an edge from bar to toString because neither bar or its callers (main) allocated any instance that toString could be called on



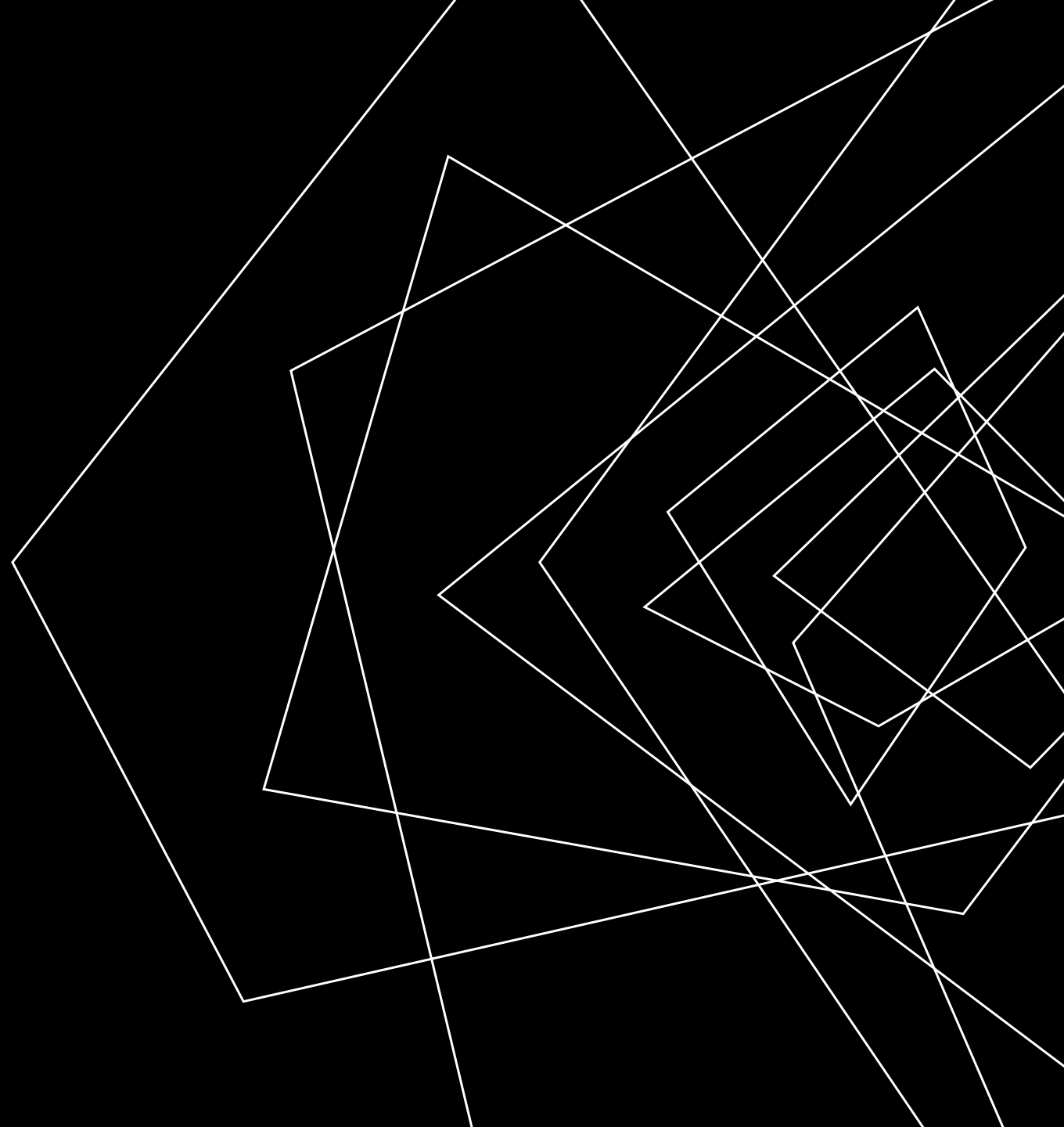
OVERVIEW

WE'VE SEEN THE NECESSITY OF MULTI-FUNCTION ANALYSIS IN REAL-WORLD PROGRAMS

TIME TO CONSIDER HOW IT IS DONE

LECTURE OUTLINE

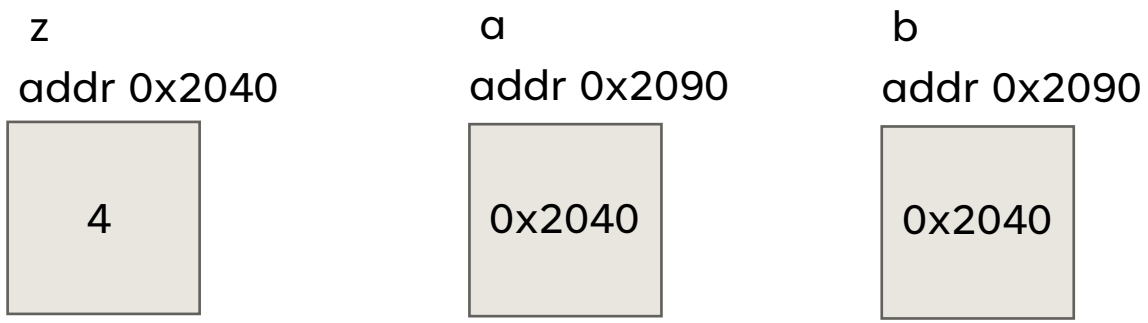
- Aliases & Points-to
- Andersen’s Analysis
- Steensgard’s Analysis



POINTERS: LOVE TO HATE ‘EM

ALIASES AND POINTS-TO SETS

```
int z = 4;
int * a = &z;
int * b = a;
int * c = &z;
*b = 2;
```

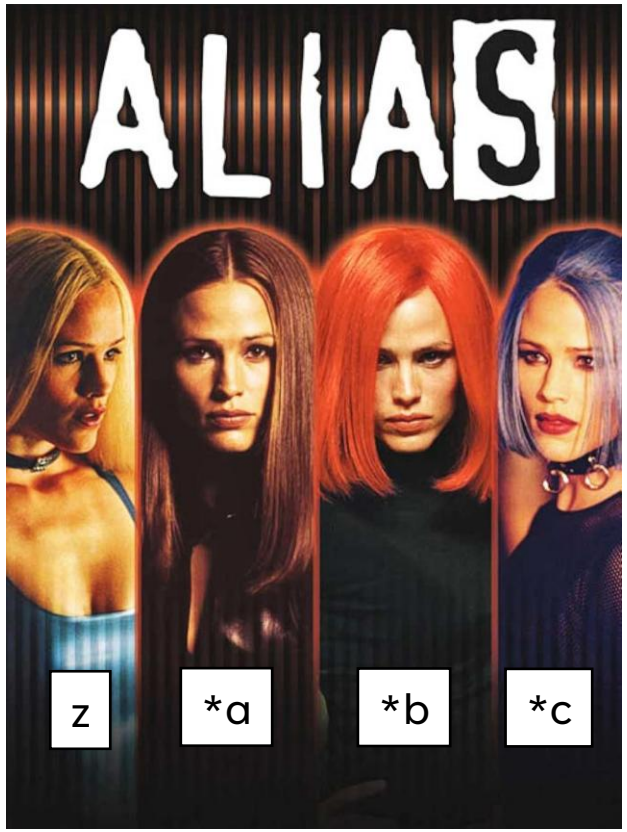


ALIAS RELATIONSHIPS

ALIASES AND POINTS-TO SETS

Create **aliasing** relationships

```
int z = 4;
int * a = &z;
int * b = a;
int * c = &z;
*b = 2;
```



ALIASES AND DATAFLOW

ALIASES AND POINTS-TO SETS

These relationships can really mess with the soundness of program verification!

```
int z = 4;
int * a = &z;
int * b = a;
int * c = &z;
*b = 2;
```

z : ?

z : 2

SAFETY IN THE PRESENCE OF ALIASES

ALIASES AND POINTS-TO SETS

may-point(p): the set of locations to which p might point

must-point(p): the set of locations to which p must point

Which of these is the “safe” set to track depends on the analysis

For us, we’ll usually over-approximate bad behavior, hence track may-point sets

SCALABLE MAY-POINT COMPUTATION

ALIASES AND POINTS-TO SETS

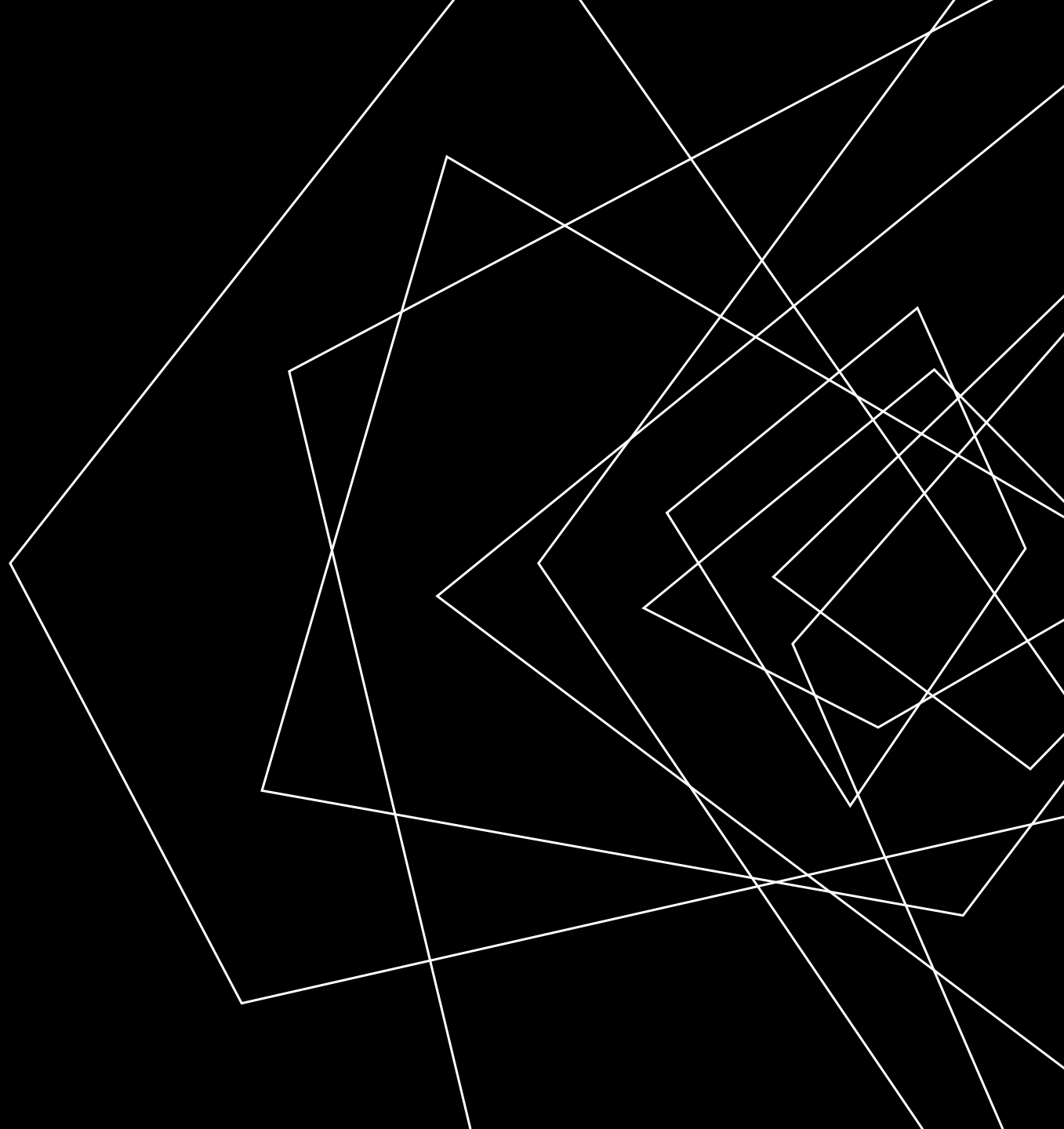
Determining points-to sets is expensive

- Interprocedural analysis somewhat out-of-scope
- Flow-sensitive analysis somewhat out-of-scope

We'll talk about 2 flow-insensitive algorithms

LECTURE OUTLINE

- May-point v Must-point
- Andersen’s Analysis
- Steensgard’s Analysis



SUBSET CONSTRAINTS

ANDERSEN'S ANALYSIS

A Flow-insensitive Algorithm

Each statement adds a constraint over the points-to sets

End up with a (solvable) system of constraints

Program

```
p = &a;
q = p;
p = &b;
r = p;
```

SUBSET CONSTRAINTS

ANDERSEN'S ANALYSIS

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$
Complex	*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$

SUBSET CONSTRAINTS

ANDERSEN'S ANALYSIS

A Flow-insensitive Algorithm

Each statement adds a constraint over the points-to sets

End up with a (solvable) system of constraints

Program	Constraints	Initial	Final
p = &a;	p ⊇ {a}	pts(p) = ~~∅~~ a, b	pts(p) = {a,b}
q = p;	q ⊇ p	pts(q) = ~~∅~~ a, b	pts(q) = {a,b}
p = &b;	p ⊇ {b}	pts(r) = ~~∅~~ a, b	pts(r) = {a,b}
r = p;	r ⊇ p	pts(a) = ∅	pts(a) = ∅
		pts(b) = ∅	pts(b) = ∅

ANOTHER EXAMPLE

ANDERSEN'S ANALYSIS

A Flow-insensitive Algorithm

Each statement adds a constraint over the points-to sets

End up with a (solvable) system of constraints

Program	**Constraints**	**Initial**	**Final**
p = &a	p ⊇ {a}	pts(p) = { a }	pts(p) = { a }
q = &b	q ⊇ {b}	pts(q) = { b }	pts(q) = { b }
*p = q;	*p ⊇ q	pts(r) = { c }	pts(r) = { c }
r = &c;	r ⊇ {c}	pts(s) = ∅	pts(s) = { a }
s = p;	s ⊇ p	pts(t) = ∅	pts(t) = { b, c }
t = *p;	t ⊇ *p	pts(a) = ∅	pts(a) = { b, c }
*s = r;	*s ⊇ r	pts(b) = ∅	pts(b) = ∅
		pts(c) = ∅	pts(c) = ∅

SOLVING CONSTRAINTS

ANDERSEN'S ANALYSIS

Graph closure on the subset relation

Assgmt.	Constraint	Meaning	Edge
a = &b	$a \supseteq \{b\}$	$b \in pts(a)$	no edge
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \rightarrow a$
a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$	no edge
*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$	no edge

OVERHEAD
ANDERSEN'S ANALYSIS

Worst Case: Cubic Time

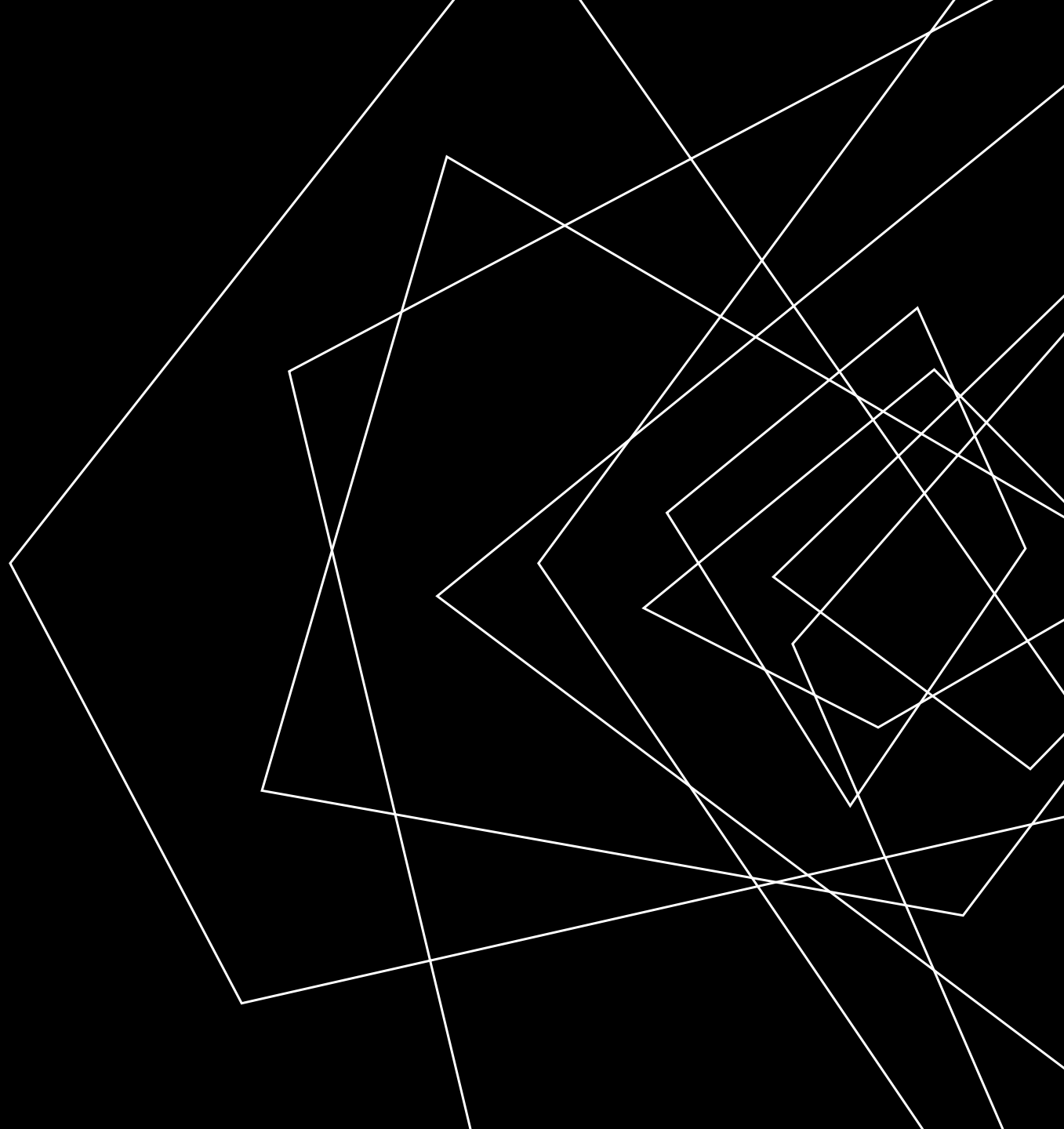
That's not great

Optimization: Cycle elimination

Detect and collapse SCCs in the points-to relation

LECTURE OUTLINE

- May-point v Must-point
- Andersen’s Analysis
- Steensgard’s Analysis



AN ALTERNATIVE APPROACH

STEENSGARD'S ANALYSIS

Aim for Near-Linear-time points-to Analysis

Going to require us to reduce our search-space somewhat

Intuition: Equality constraints

Do away with the notion of subsets

EQUALITY CONSTRAINTS

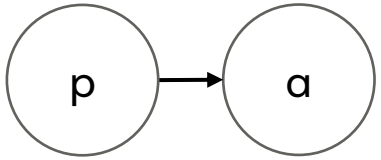
STEENSGARD'S ANALYSIS

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	a = b	pts(a) = pts(b)
Complex	a = *b	a = *b	$\forall v \in pts(b).\ pts(a) = pts(v)$
Complex	*a = b	*a = b	$\forall v \in pts(a).\ pts(v) = pts(b)$

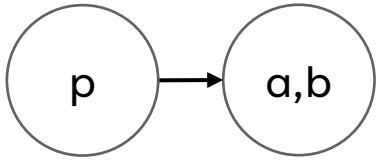
EQUALITY CONSTRAINTS

STEENSGARD'S ANALYSIS

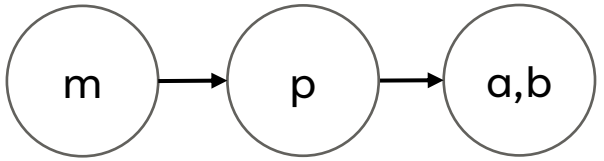
p = &a



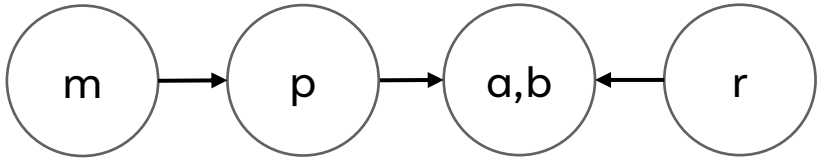
p = &b



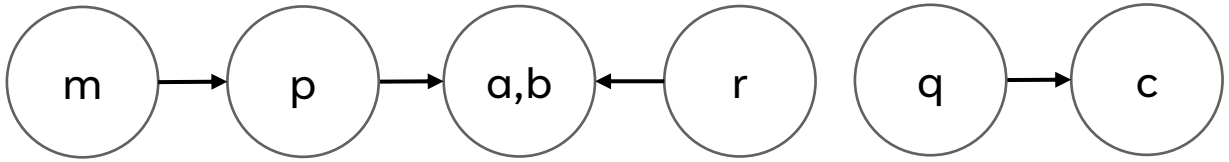
m = &p



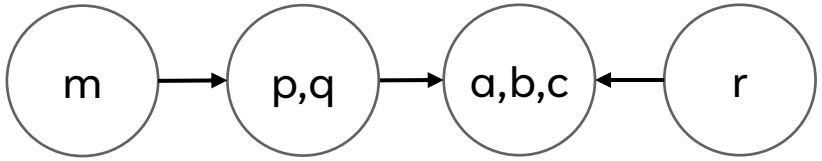
r = *m



q = &c



m = &q

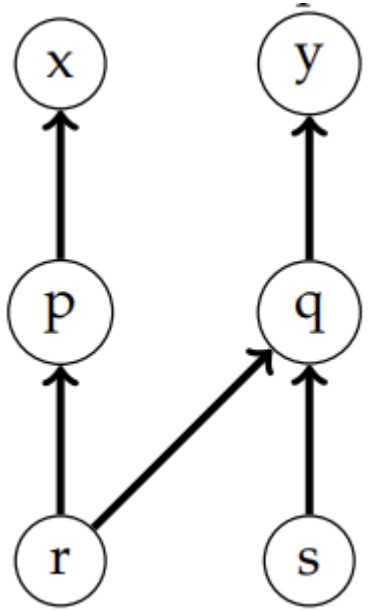


EQUALITY CONSTRAINTS

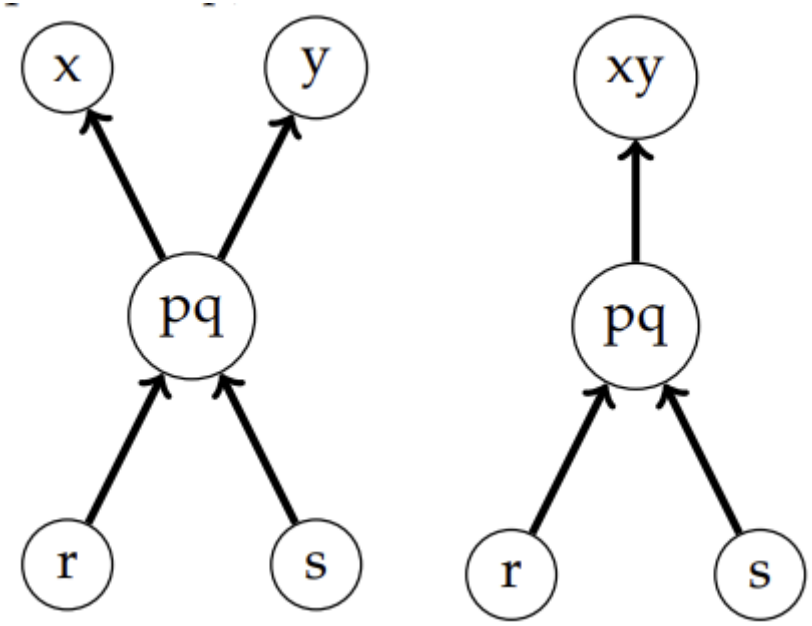
STEENSGARD'S ANALYSIS

$$
\begin{aligned}
1: &\quad p := \&x \\
2: &\quad r := \&p \\
3: &\quad q := \&y \\
4: &\quad s := \&q \\
5: &\quad r := s
\end{aligned}
$$

Andersen's



Steensgard's



WRAP-UP