**CS6843 - Program Analysis** **Jan-May 2017**

# Week #6

*Lecturer: Rupesh Nasre.* *Scribes: Bhuvan Agrawal & Seshagiri*

## 1 Pointer analysis as a graph problem

Here, we will try to model pointer analysis as a graph theoretic problem. We will try to see if we can get a different formulation from the existing analyses and try to do some optimisations.
Every graph is defined by a set of vertices and edges. We need to define a particular set of vertices and edges to formulate the points to analysis as a graph problem.

### 1.1 A natural attempt

We can use the graph defined by the points-to relation. Suppose $G = (V, E)$ is a directed graph. Then,

$$V = \text{The set of variables (pointers and pointees)}$$

$$E = \{(p, q) \mid p \text{ points to } q\}$$

**Some observations**

- If our language is strongly typed, then $G$ will be a DAG. Furthermore, edges will only go from a vertex of type $(T*)$ to $(T)$ (Figure 1).
- We can merge two vertices if they have the same *points-to* and *pointed-by* sets.

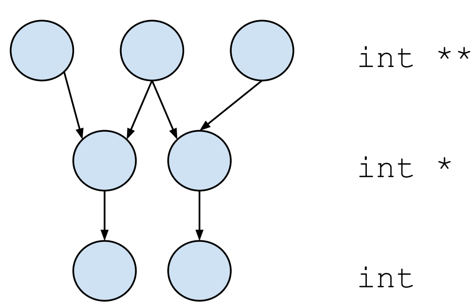



Figure 1: A sample points-to graph for a strictly typed language

### 1.2 Constraint graph

Consider the following formulation which mimics Andersen's analysis. Suppose $G = (V, E)$ is a directed graph. Then,

$$V = \text{The set of pointers}$$

$$E = \{(p, q) \mid \text{PtsTo}(p) \subseteq \text{PtsTo}(q)\}$$

where $\text{PtsTo}(p)$ denotes the points to set of $p$.

We have the following algorithm.

---

**Algorithm 1**

---

**Input:** Set $C$ of points-to constraints
**Output:** Points-to information

```
Process address-of constraints
Add edges to constraint graph G using copy statements [1]
repeat
    Propagate points-to information in G
    Add edges to G using load and store statments
until Fixed point
```

---

**Example 1.** Consider the following constraints.

$e = d; b = a;$ (Copy constriants)
$*e = c; c = *a; *a = p;$ (Load/store constraints)
$a \mapsto \{a, q, r, s, t\}, p \mapsto \{b, c, d\}$ (Initial points-to sets)

Figure 2 shows each iteration of the above algorithm on this input. The nodes $q, r, s, t$ have been combined into one in the figure for clarity because we know that they have the same points-to sets at the end of the algorithm (*after having computed it once before!*).[2]

Till now, it seems as if the graph formulation is identical to Andersen's analysis. Then what benefits does such a formulation provide?

- A naive formulation offers no benefits over the constraint-based formulation.
- We need to exploit structural properties of the constraint graph for efficient execution.
  - Online cycle detection
  - Online dominator detection
  - Propagation order: Topological sort, depth first

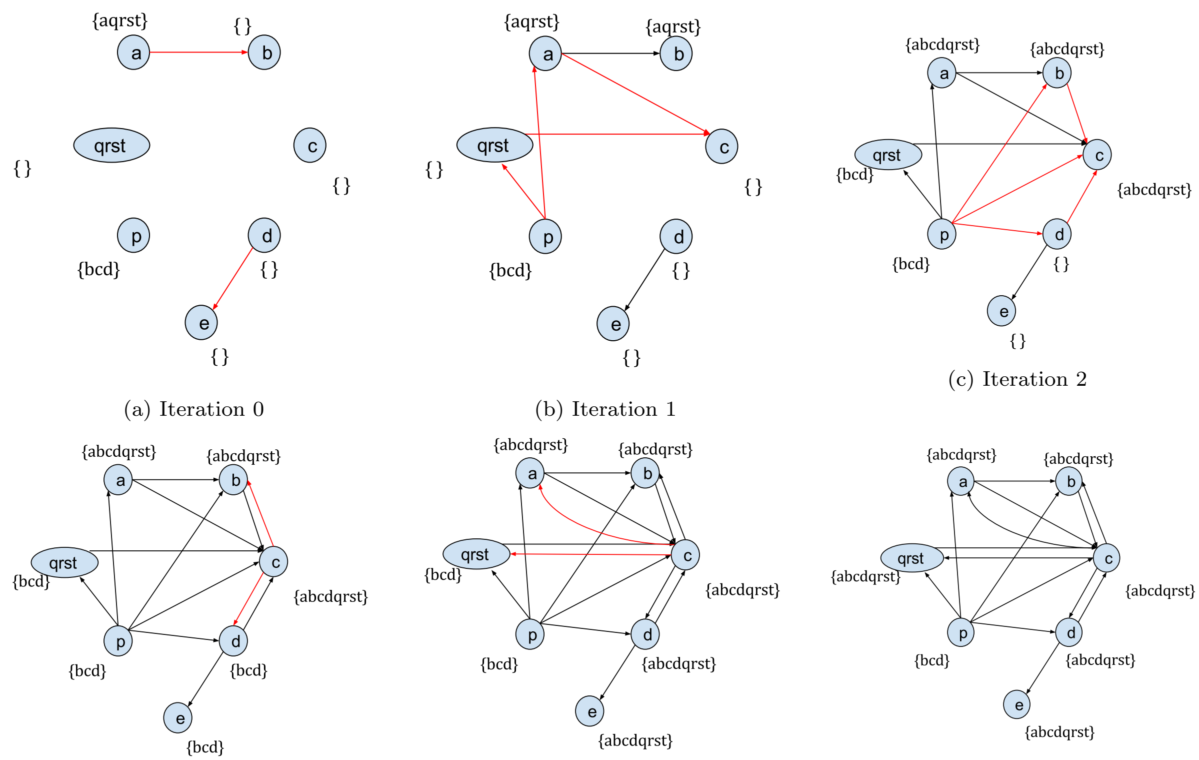
#### 1.2.1 Pointer Equivalence

**Definition 1** (Pointer Equivalence)**.** *Two pointers $p$ and $q$ are equivalent if they have the same points-to set (at the end of the analysis).*

---

[1] Note that we only need to process copy statements once, as opposed to Andersen's analysis

[2] Interestingly, in this example, we can see that the input is symmetric with respect to $q, r, s, t$. Suppose we interchange $q$ and $r$ (or any permutation of $q, r, s, t$), the input as a whole does not change. So one can argue that $q, r, s, t$ will have the same points-to sets without actually running the algorithm.




(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

(d) Iteration 3 (e) Iteration 4 (f) Iteration 5 (fixed point)

Figure 2: Constraint graph after each iteration as computed by Algorithm 1

Our aim is identify such equivalent pointers *before* computing their points-to information. This will reduce the number of nodes in our constraint graph, and consequently, the amount of information propagated during the analysis, which is usually the costliest operation.

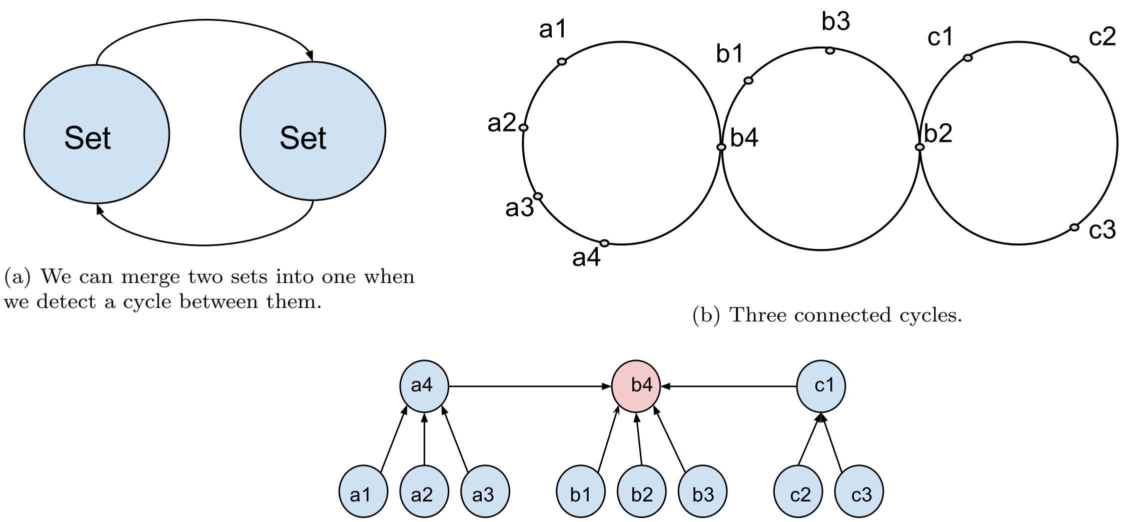
### 1.2.2 Online cycle detection

**Theorem 1.** *Suppose the pointers $v_1 \dots v_k$ form a cycle in the constraint graph. Then, these pointers are equivalent.*

*Proof.* From the definition of the constraint graph, $\text{PtsTo}(v_1) \subseteq \text{PtsTo}(v_2) \subseteq \ldots \subseteq \text{PtsTo}(v_k) \subseteq \text{PtsTo}(v_1)$. Thus, $\text{PtsTo}(v_1) = \text{PtsTo}(v_2) = \ldots = \text{PtsTo}(v_k)$. □

Since edges are added to the graph dynamically, cycle detection is performed online so that we can *collapse* cycles (due to theorem 1). When we say that we *collapse* cycles, we actually mean that we merge the PtsTo sets, the incoming edges, and the outgoing edges of the pointers in the cycle and replace the whole cycle with a representative element. Since some pointers might have some extra edges added to them, it might seem that this operation would decrease the precision of our analysis. However, this is not the case since the extra edges are not going to flow any *new* information that wouldn't have already flowed through the old edges in the cycle.

We can use the union find data structure to efficiently perform the cycle collapse. The basic idea is that all the pointers in a particular cycle would be placed in the same set and the whole set would be referred by a particular representative pointer. If we find that two sets are connected by a cycle, or if they have a common pointer, then we will merge the two sets. Figure 3 shows an example of how the union-find data structure is used in collapsing cycles.




(a) We can merge two sets into one when we detect a cycle between them.

(b) Three connected cycles.

(c) Disjoint set data structure. The representative $b4$ is coloured in pink.

Figure 3: Example showing how disjoint set data structure is used in cycle collapse.

#### 1.2.3 Online dominator detection

Suppose we have a graph structure like that in Figure 4. Let us ignore the address of statements for now. We can see that the only incoming edges to $B$ and $C$ are through the path from $A$. Then, what can we say about the points-to sets of $A, B,$ and $C$? We know that

$$\text{PtsTo}(A) \subseteq \text{PtsTo}(B) \subseteq \text{PtsTo}(C)$$

The only way any information can flow to $B$ and $C$ is through $A$. So, we can conclude that $\text{PtsTo}(A) = \text{PtsTo}(B) = \text{PtsTo}(C)$.

**Note:** We can only perform this optimisation if the graph structure at the *end* (fixed point) looks like this. If some other incoming edge gets added to $B$ or $C$ during the analysis and we keep the nodes merged, then our analysis will become imprecise.

We will now formalise this notion of 'no other incoming edge other than A'.

**Definition 2** (Domination)**.** *In a directed graph with a start node $S$, we say that a node $A$ dominates $B$ (written as $A \,\mathsf{dom}\, B$) if every path from $S$ to $B$ must go through $A$.*

*Remark:* $\mathsf{dom}$ *is a transitive relation.*

**Theorem 2.** *Suppose we define an appropriate start node for our constraint graph and ignore the address-of constraints.*[3] *Then we have the following:*

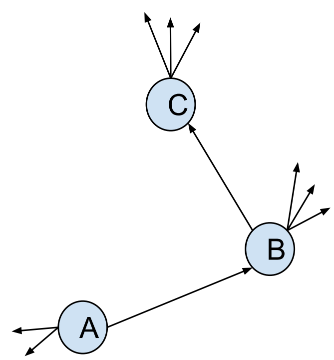



Figure 4: Note that $B$ and $C$ don't have any other incoming edges other than through $A$

1. *If two nodes in a constraint graph have the same dominator, they are pointer equivalent.*[4]
2. *A dominator and its dominees are pointer equivalent.*

*Proof.* Suppose $A$ dom $B$ and $A$ dom $C$. We know that $\text{PtsTo}(A) \subseteq \text{PtsTo}(B)$ and $\text{PtsTo}(A) \subseteq \text{PtsTo}(C)$. By definition of dom, any information that flows into $B$ or $C$, must pass through $A$, so they cannot have any 'extra' information which is not present in $A$. Therefore, we can say that $\text{PtsTo}(A) = \text{PtsTo}(B)$ and $\text{PtsTo}(A) = \text{PtsTo}(C)$. □

If we want to use this optimisation to merge some nodes in our graph, we need to be able to perform dominator detection online (during the analysis). Furthermore, we need to make sure that once we merge a set of nodes, some more edges are not added later to the set in such a way that it changes the dom relation within the set.

Can we make sure that a particular dom relation (say, $A$ dom $B$) will never change? In other words, once a dominator, always a dominator. Here is a sufficient condition: if we can ensure that $B$ has only certain fixed predecessors, then we can ensure the above condition. So, in our algorithm, after all copy statements are processed the only way any new incoming edge can be added to $B$ is through a statement of the type $*P = Q$, where $B$ is present in the points-to set of $P$. Now, if $B$'s address is never taken, i.e. there is no statement of the type $P = \&B$, then we can say that such a load statement will not add any incoming edge to $B$.

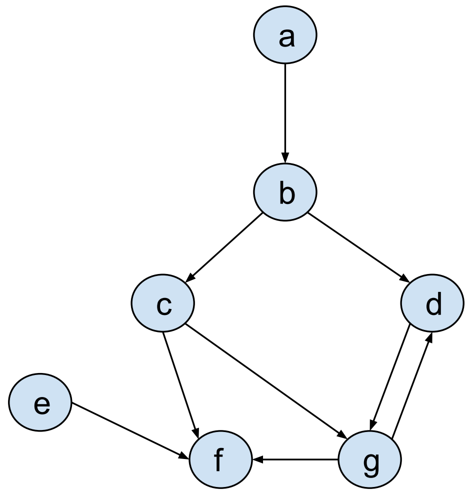
#### 1.2.4 Offline Variable Substitution

Some constraints are easy to check without running the analysis, such as

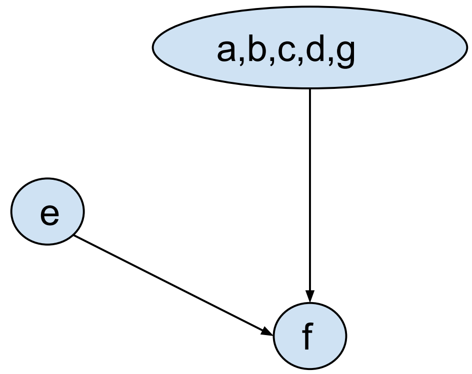
- $a = b,\ b = a$ : This forms a cycle between $a$ and $b$.
- $a = *p,\ *p = a$ : This forms a cycle between $a$ and the elements in the points-to set of $p$.

[3] We can model the address-of constraints by adding a node '$\&x$' for every variable whose address has been taken and adding the edge $\&x \to p$ for every statement of the form $p = \&x$. We can then define a start node $S$ and add edges from $S$ to every address-of node.

[4] This does not hold if we choose the start node $S$ as our dominator, or else, every variable would have the same points-to set.




(a) Original constraint graph




(b) Constraint tree after collapsing the dominators and dominees

Figure 5: An example showing the dominator-dominee collapse

- $a = b$, $c = a$, $c = b$ and no other incoming edge to $c$ : This creates the $a$ dom $c$ relation.

Offline variable substitution is performed before running the pointer analysis. We need to perform OVS in such a way that the total time and space of the analysis (including OVS) is less than the time taken in the analysis without any OVS.

#### 1.2.5 Propagation Order

1. **Wave propagation:** A topological ordering is beneficial for propagating points-to information. It may reduce the time taken per iteration. This is called wave propagation.
2. **Deep propagation:** Wave propagation may consume a lot of memory. In such cases, propagating information in a depth-first manner can reduce the memory requirement. Only the differences may also be propagated to further reduce the memory usage. It also takes advantage of the locality of reference. This is called deep propagation.

## References

[1] R. Nasre, CS6843 course slides

[2] R. Nasre, Exploiting the structure of the constraint graph for efficient points-to analysis