On-The-Fly Static Analysis via Dynamic Bidirected Dyck Reachability

S. Krishna, **Aniket Lal, Andreas Pavlogiannis,** Omkar Tuppe

Dyck Reachability at a Glance

- A graph reachability problem
- Widely used model for static analyses
  - Graphs as program models

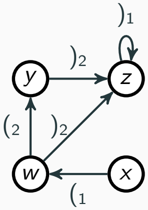
Dyck Reachability at a Glance

- A graph reachability problem
- Widely used model for static analyses
    - Graphs as program models
- A few variants
- Need to solve fast

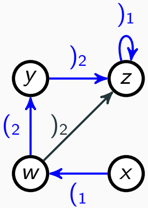
Dyck Reachability at a Glance

- A graph reachability problem
- Widely used model for static analyses
    - Graphs as program models
- A few variants
- Need to solve fast
- ... **how fast?**

Dyck Reachability Graph

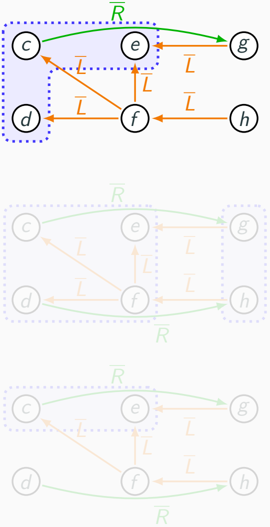


Dyck Reachability Graph



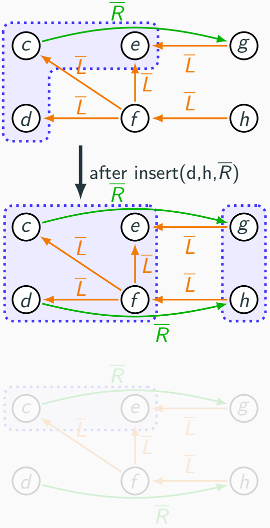
Computing Dyck Reachability for Alias Analysis

```
class ATree {
  ATree L;
  ATree R;
}
void main(){
  ATree c,d,e;
  ATree f,g,h;
  g.L=e;
  d=f.L;
  h.L=f;
  f.L=c;
  c.R=g;
  e=f.L


}
```



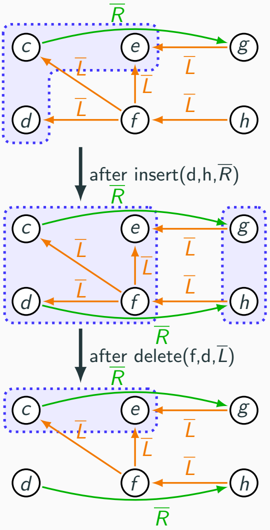
Computing Dyck Reachability for Alias Analysis

```
class ATree {
  ATree L;
  ATree R;
}
void main(){
  ATree c,d,e;
  ATree f,g,h;
  g.L=e;
  d=f.L;
  h.L=f;
  f.L=c;
  c.R=g;
  e=f.L
  h=d.R;

}
```



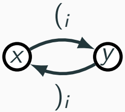
Computing Dyck Reachability for Alias Analysis

```
class ATree {
  ATree L;
  ATree R;
}
void main(){
  ATree c,d,e;
  ATree f,g,h;
  g.L=e;
  d=f.L;
  h.L=f;
  f.L=c;
  c.R=g;
  e=f.L
  h=d.R;

}
```

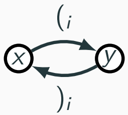


Bidirected Dyck Reachability

Bidirected graphs



Bidirected graphs



- CFL-models of alias/pointer analysis
- Used to handle mutable heap data
- Quick overapproximation of CFL-reachability

Overview

Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.

Overview

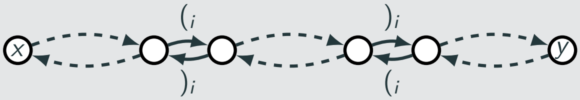
Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.

Overview

Key Observation

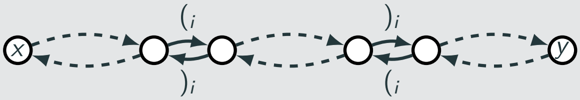
Dyck reachability on bidirected graphs is an **equivalence relation**.



Overview

Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.

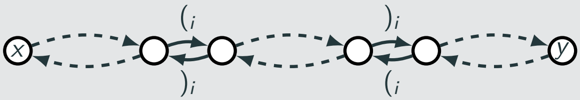


- Compute Dyck-Strongly Connected Components (DSCC)

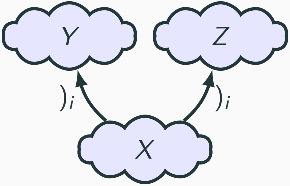
Overview

Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



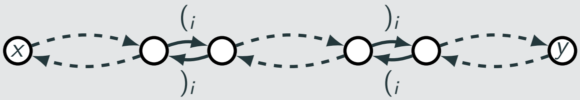
- Compute Dyck-Strongly Connected Components (DSCC)



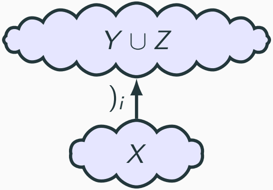
Overview

Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



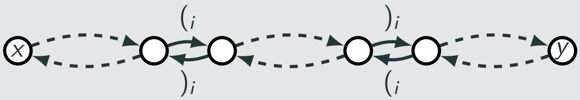
- Compute Dyck-Strongly Connected Components (DSCC)



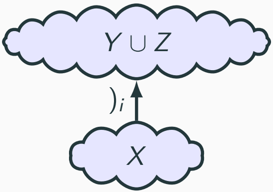
Overview

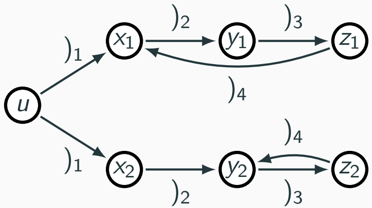
Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



- Compute Dyck-Strongly Connected Components (DSCC)

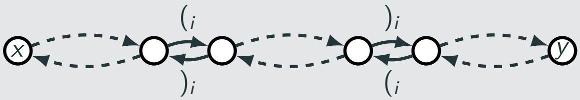




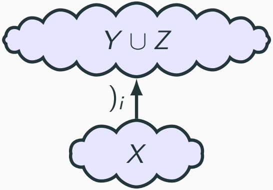
Overview

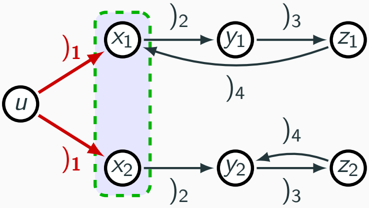
Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



- Compute Dyck-Strongly Connected Components (DSCC)

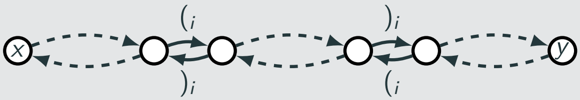




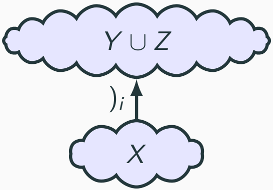
Overview

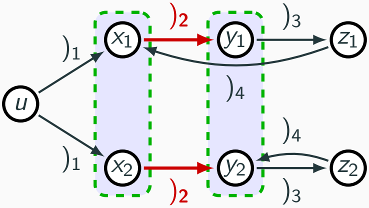
Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



- Compute Dyck-Strongly Connected Components (DSCC)

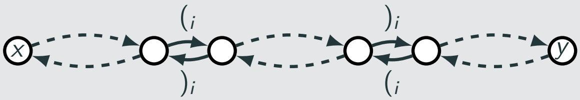




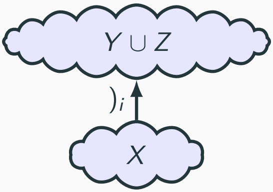
Overview

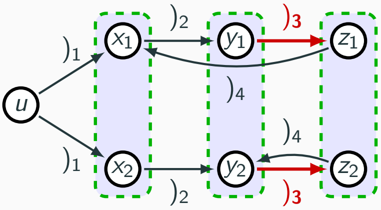
Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



- Compute Dyck-Strongly Connected Components (DSCC)

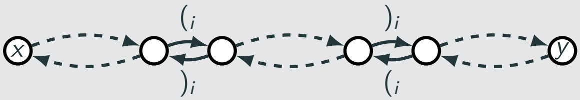




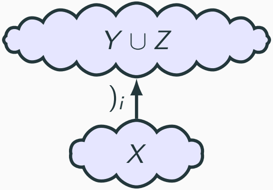
Overview

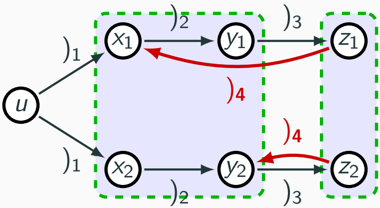
Key Observation

Dyck reachability on bidirected graphs is an **equivalence relation**.



- Compute Dyck-Strongly Connected Components (DSCC)





Unification style!

Offline Algorithm

Theorem

All DSCCs of a graph with n nodes and m edges takes $O(m + n \cdot \alpha(n)$ time.

- *$\alpha(n)$ is the inverse Ackermann function.*

On The Fly Analysis

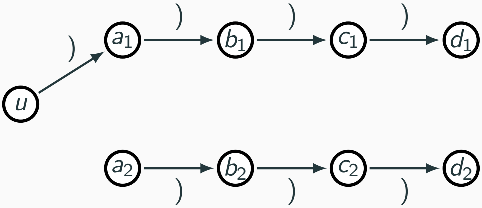
- As source code is developed, the graph changes
- Maintain analysis on the fly
- Fully-dynamic reachability
  - insert(u, v, i), delete(u, v, i)
- **How fast?**

On The Fly Analysis

- As source code is developed, the graph changes
- Maintain analysis on the fly
- Fully-dynamic reachability
  - insert(u, v, i), delete(u, v, i)
- **How fast?**
- Running the offline algorithm after each modification takes $O(m + n \cdot \alpha(n))$

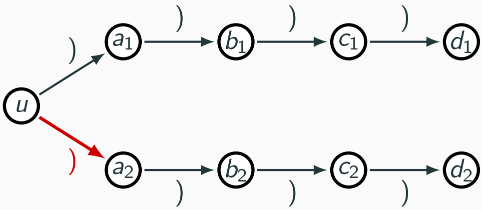
On The Fly Analysis

- As source code is developed, the graph changes
- Maintain analysis on the fly
- Fully-dynamic reachability
    - insert(u, v, i), delete(u, v, i)
- **How fast?**
- Running the offline algorithm after each modification takes $O(m + n \cdot \alpha(n))$
- $o(n)$ guarantees are tricky



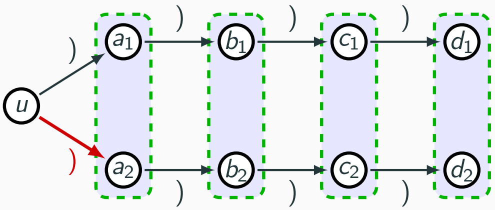
On The Fly Analysis

- As source code is developed, the graph changes
- Maintain analysis on the fly
- Fully-dynamic reachability
  - insert(u, v, i), delete(u, v, i)
- **How fast?**
- Running the offline algorithm after each modification takes $O(m + n \cdot \alpha(n))$
- $o(n)$ guarantees are tricky



On The Fly Analysis

- As source code is developed, the graph changes
- Maintain analysis on the fly
- Fully-dynamic reachability
  - insert(u, v, i), delete(u, v, i)
- **How fast?**
- Running the offline algorithm after each modification takes $O(m + n \cdot \alpha(n))$
- $o(n)$ guarantees are tricky



This Paper

Theorem

On-the-fly bidirected CFL analysis on a dynamically-changing graph of n nodes and $m \leq$ edges takes $O(n \cdot \alpha(n))$ time per update (insertion/deletion)

This Paper

Theorem

On-the-fly bidirected CFL analysis on a dynamically-changing graph of n nodes and $m \leq$ *edges takes* $O(n \cdot \alpha(n))$ *time per update (insertion/deletion)*

$+$ a practical improvement that updates (seemingly) in constant time

Inserting Edges is Easy

Deleting Edges is Tricky

Primary DSCCs

Primary DSCCs

Primary DSCCs

Our result, in two steps

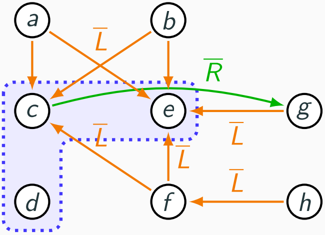
- Maintaining PDCSSs in $O(n \cdot \alpha(n))$ time
- Recomputing from the PDSCC graph in $O(n \cdot \alpha(n))$ time

Efficient Dynamic Dyck Reachability

Consider given graph,

DSCCs :

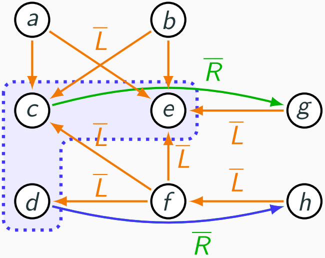
{a}, {b}, {g}, {h}, {c,d,e}, {f}



Efficient Dynamic Dyck Reachability

insert $d \xrightarrow{\bar{R}} h$

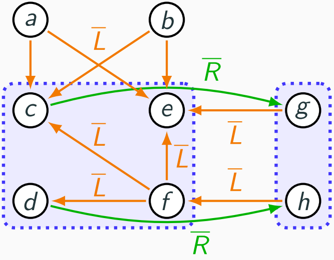
- Since edge insertion can only cause merging of components,
- Update Worklist Q, call fixpoint() computation



Efficient Dynamic Dyck Reachability

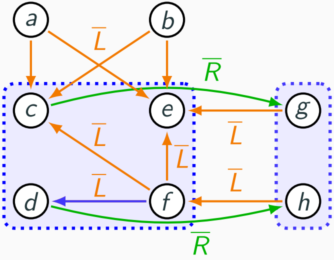
insert $d \xrightarrow{\bar{R}} h$

- Since edge insertion can only cause merging of components,
- Update Worklist Q, call fixpoint() computation
- $O(n.\alpha(n))$ for each insert update operation (Chatergee et. al 2018)



Efficient Dynamic Dyck Reachability

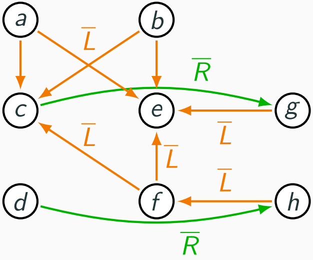
delete $f \xrightarrow{\bar{L}} d$



Efficient Dynamic Dyck Reachability

delete $f \xrightarrow{\bar{L}} d$

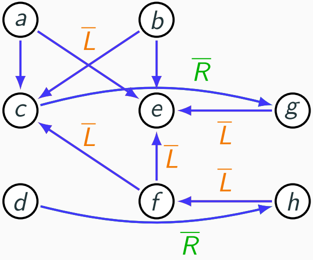
- recompute from scratch?



Efficient Dynamic Dyck Reachability

delete $f \xrightarrow{\bar{L}} d$

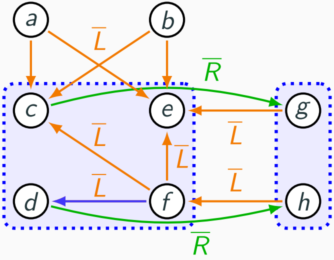
- recompute from scratch?
- No of edges processed by fixpoint() function $= O(n^2)$



Efficient Dynamic Dyck Reachability

delete $f \xrightarrow{\bar{L}} d$

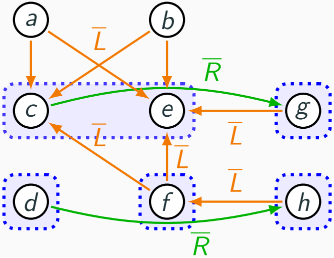
- Perform forward search from DSCC(d) and find affected DSCCs



Efficient Dynamic Dyck Reachability

delete $f \xrightarrow{\bar{L}} d$

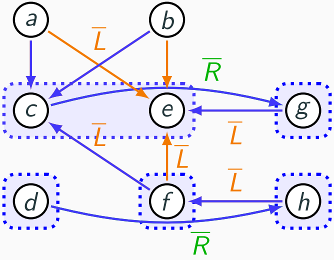
- Perform forward search from DSCC(d) and find affected DSCCs
- Breakdown DSCCs to Primary Components (PDSCCs)



Efficient Dynamic Dyck Reachability

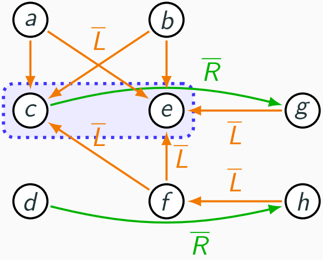
delete $f \xrightarrow{\bar{L}} d$

- Perform forward search from DSCC(d) and find affected DSCCs
- Breakdown DSCCs to Primary Components (PDSCCs)
- No of edges processed by fixpoint() function $= O(n)$



Efficient Dynamic Dyck Reachability

delete $f \xrightarrow{\bar{L}} d$



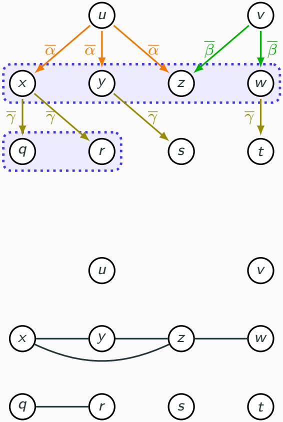
- Perform forward search from DSCC(d) and find affected DSCCs
- Breakdown DSCCs to Primary Components (PDSCCs)
- No of edges processed by fixpoint() function $= O(n)$
- $O(n.\alpha(n))$ for each delete update operation

Primary components (PDSCCs) and Primal Graphs

For Bidirected graph G = (V,E), The primal graph H = (V,L) is an unlabelled, undirected graph, such that

$$L = \{(x, y) \colon \exists u \in V. \exists \overline{\alpha} \in \Sigma^C. u \xrightarrow{\overline{\alpha}} x, u \xrightarrow{\overline{\alpha}} y \in E\}$$

- Primary DSCC (PDSCCs) of graph G is a (maximal) connected component of primal graph H
- PDSCC is a refinement of its DSCC partitioning
- We use Undirected Graph Reachability Data Structure to represent PDSCC

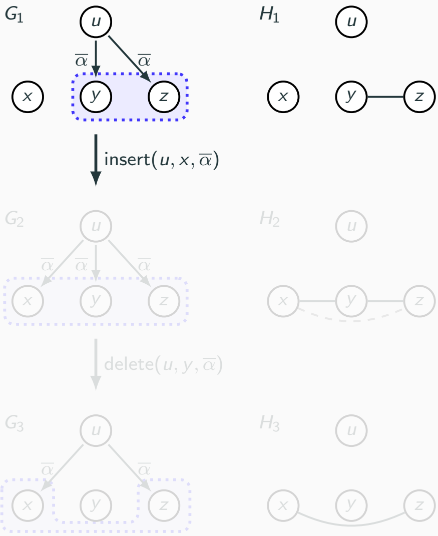


(a) A Bidirected Graph G (Top) and its corresponding primal graph H (Bottom)

Sparsification to maintain PDSCCs efficiently

PDSCCs of G_i across edge insertions and deletions, corresponding primal graphs H_i.

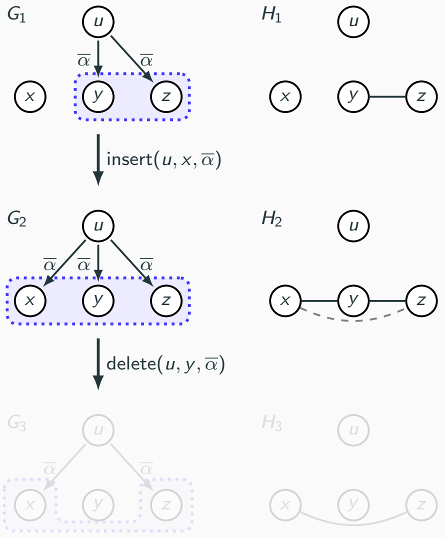
- Inserting/deleting an edge in G, may lead to addition/removal of 0 to n-1 undirected edges in primal graph
- either one of xy or xz edge is added in H_2
- xy and yz edge is deleted, xz edge is added



Sparsification to maintain PDSCCs efficiently

PDSCCs of G_i across edge insertions and deletions, corresponding primal graphs H_i.

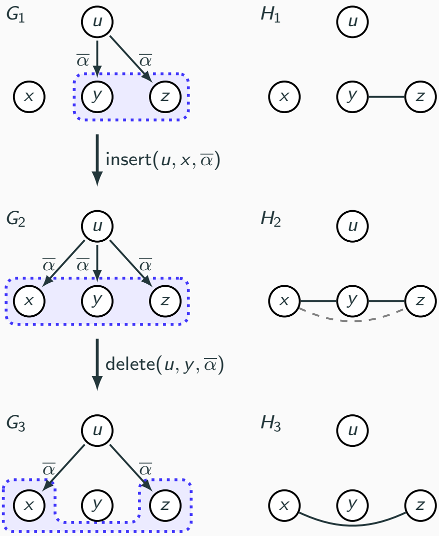
- Inserting/deleting an edge in G, may lead to addition/removal of 0 to n-1 undirected edges in primal graph
- either one of xy or xz edge is added in H_2
- xy and yz edge is deleted, xz edge is added



Sparsification to maintain PDSCCs efficiently

PDSCCs of G_i across edge insertions and deletions, corresponding primal graphs H_i.

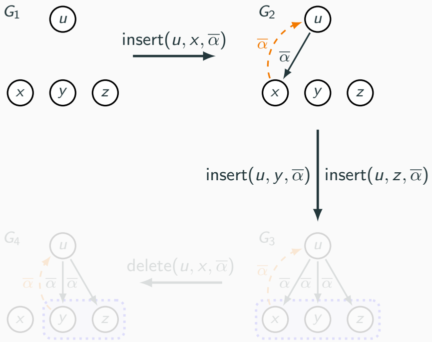
- Inserting/deleting an edge in G, may lead to addition/removal of 0 to n-1 undirected edges in primal graph
- either one of xy or xz edge is added in H_2
- xy and yz edge is deleted, xz edge is added



InPrimary

Maintainance of the sets InPrimary

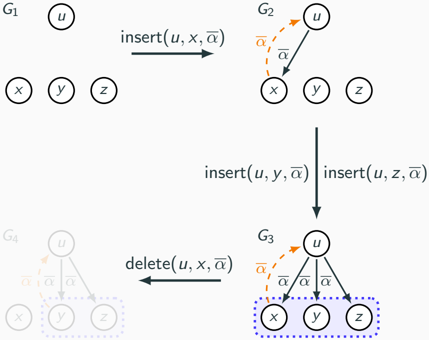
- The first edge insertion $u \xrightarrow{\overline{\alpha}} x$ leads to $u \in \mathit{InPrimary}[x][\overline{\alpha}]$.
- $u \xrightarrow{\overline{\alpha}} y$ and $u \xrightarrow{\overline{\alpha}} z$ do not modify *InPrimary*, as x, y and z belong to the same PDSCC
- On delete $u \xrightarrow{\overline{\alpha}} x$, we move u to $\mathit{InPrimary}[y][\overline{\alpha}]$, thus u can still be retrieved as a $\overline{\alpha}$-neighbor of the PDSCC $\{y, z\}$



InPrimary

Maintainance of the sets InPrimary

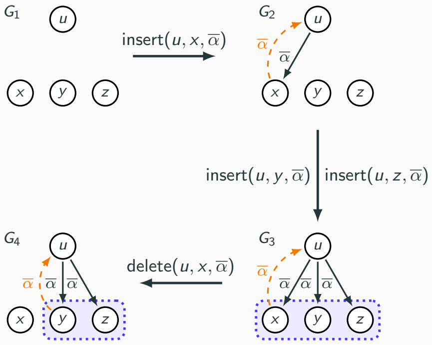
- The first edge insertion $u \xrightarrow{\overline{\alpha}} x$ leads to $u \in \mathit{InPrimary}[x][\overline{\alpha}]$.
- $u \xrightarrow{\overline{\alpha}} y$ and $u \xrightarrow{\overline{\alpha}} z$ do not modify *InPrimary*, as x, y and z belong to the same PDSCC
- On delete $u \xrightarrow{\overline{\alpha}} x$, we move u to $\mathit{InPrimary}[y][\overline{\alpha}]$, thus u can still be retrieved as a $\overline{\alpha}$-neighbor of the PDSCC $\{y, z\}$



InPrimary

Maintainance of the sets InPrimary

- The first edge insertion $u \xrightarrow{\overline{\alpha}} x$ leads to $u \in \mathit{InPrimary}[x][\overline{\alpha}]$.
- $u \xrightarrow{\overline{\alpha}} y$ and $u \xrightarrow{\overline{\alpha}} z$ do not modify *InPrimary*, as x, y and z belong to the same PDSCC
- On delete $u \xrightarrow{\overline{\alpha}} x$, we move u to $\mathit{InPrimary}[y][\overline{\alpha}]$, thus u can still be retrieved as a $\overline{\alpha}$-neighbor of the PDSCC $\{y, z\}$



Experiments – Benchmarks

Context-Sensitive Data Dependence Analysis [Tang et al. 2015]

@Aniket: Put here a very short snippet of code, and the graph it is modeled as

Field-Sensitive Alias Analysis for Java [Yan et al. 2011; Zhang et al. 2013]

@Aniket: Put here a very short snippet of code, and the graph it is modeled as

Field-Sensitive Alias Analysis for Java

```
Class Node{
  Node f;
  Node g;
};

a.g = b;
b.f = e;
c = a.g;
h = c.f;
d = a.f;
```

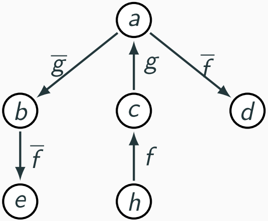


Figure 6: Field-Sensitive Inter-procedural Symbolic Points to graph [Yan et al. 2011; Zhang et al. 2013]

Context-Sensitive Data Dependence Analysis

```
f(x1) {
  y1 = x1 + 1;
  return y1;
}

x = 4;
y = f(x);
```

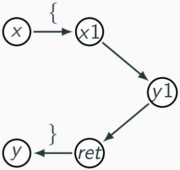


Figure 7: Context-sensitive Data-Dependence graph [Tang et al. 2015]

Experiments – Algorithms

Compared 3 algorithms

- **Offline**
  - Invoked after each update
- **Dynamic DataLog**
  - Each update modifes a DataLog program that expresses reachability
  - Dispatched to a Dynamic DataLog solver
- **Our Dynamic Algorithm**
  - As sketched so far

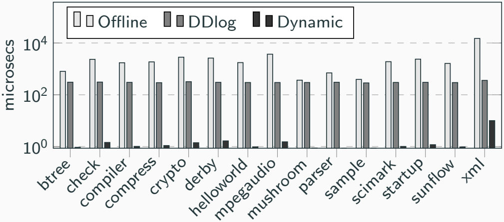
Experimentation - Formulating update sequence

For each benchmark graph G, we generate a sequence of update (edge insert/delete) operations S_G as follows:

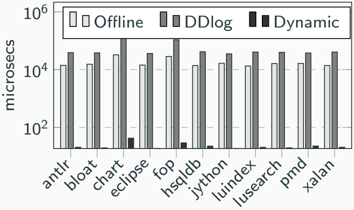
- Incremental setting - S_G^{inc} is a sequence of edge insertions from a random permutation of 90% of edges of G
- Decremental setting - S_G^{dec} is a sequence of edge deletions from a random permutation of 90% of edges of G
- Mixed setting - Randomly split G into sets E^+ and E^- with proportion 10% and 90%
    - Initial graph - E^-
    - sequence S_G^{mix} - Created by repeated stochastic sampling of E^+ and randomly selecting that edge as insert/delete operation

Experimental Results

Data Dependence Analysis



Alias Analysis



Thank You!

Questions?

Appendix

Declarative DataLog Approach

Dispatching to a DataLog solver:

- Reaches(u,u)
- Close(x,u,i) :- Edge(x,u,i)
- Close(x,u,i) :- Edge(y,u,i), Reaches(x,y)
- Reaches(u,v) :- Close(x,u,i), Close(x,v,i)
- Reaches(u,v) :- Reaches(u,v), Reaches(x,v)