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
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
}
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;
}
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

\[ (i \to x, x \to y, y \to i) \]
Bidirected graphs

- CFL-models of alias/pointer analysis
- Used to handle mutable heap data
- Quick overapproximation of CFL-reachability
Dyck reachability on bidirected graphs is an equivalence relation.

Key Observation

Dyck reachability on bidirected graphs is an equivalence relation.
Key Observation

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

![Diagram showing Dyck reachability](image)
Overview

Key Observation

Dyck reachability on bidirected graphs is an equivalence relation.

\[
\begin{align*}
(x &\rightarrow y) \\
(x &\rightarrow i) \\
i &\rightarrow (y)
\end{align*}
\]
Key Observation
Dyck reachability on bidirected graphs is an equivalence relation.

- Compute Dyck-Strongly Connected Components (DSCC)
Key Observation

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

- Compute Dyck-Strongly Connected Components (DSCC)
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)

![Diagram showing Dyck reachability and DSCC computation](image-url)
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**.

\[ \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z} \]

- 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**.

\[ x \rightarrow^i y \]

- Compute Dyck-Strongly Connected Components (DSCC)

\[ X \cup Y \cup Z \]

Unification style!
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

\[
\begin{align*}
\text{u} &\rightarrow a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow d_1 \\
&\rightarrow a_2 \rightarrow b_2 \rightarrow c_2 \rightarrow d_2
\end{align*}
\]
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
  - $\text{insert}(u, v, i)$, $\text{delete}(u, v, i)$
- **How fast?**
- Running the offline algorithm after each modification takes $O(m + n \cdot \alpha(n))$
- $o(n)$ guarantees are tricky

![Diagram of a graph showing nodes and edges with annotations for insert and delete operations.]
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
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
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
Consider given graph, DSCCs:
\{a\}, \{b\}, \{g\}, \{h\}, \{c,d,e\}, \{f\}
insert $d \xrightarrow{\bar{R}} h$

- Since edge insertion can only cause merging of components,
- Update Worklist $Q$, call fixpoint() computation
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 (Chatterjee et. al 2018)
delete $f \xrightarrow{L} d$
delete $f \xrightarrow{L} d$

- recompute from scratch?
delete $f \xrightarrow{\bar{L}} d$

- recompute from scratch?
- No of edges processed by fixpoint() function = $O(n^2)$
delete $f \xrightarrow{L} d$

- Perform forward search from DSCC(d) and find affected DSCCs
delete $f \xrightarrow{L} d$

- Perform forward search from DSCC(d) and find affected DSCCs
- Breakdown DSCCs to Primary Components (PDSCCs)
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 = $O(n)$
- $O(n \cdot \alpha(n))$ 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

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

- 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)
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.
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
Maintainance of the sets $\text{InPrimary}$

- The first edge insertion $u \xrightarrow{\alpha} x$ leads to $u \in \text{InPrimary}[x][\overline{\alpha}]$.

- $u \xrightarrow{\overline{\alpha}} y$ and $u \xrightarrow{\overline{\alpha}} z$ do not modify $\text{InPrimary}$, as $x$, $y$ and $z$ belong to the same PDSCC.

- On delete $u \xrightarrow{\overline{\alpha}} x$, we move $u$ to $\text{InPrimary}[y][\overline{\alpha}]$, thus $u$ can still be retrieved as a $\alpha$-neighbor of the PDSCC $\{y, z\}$.
Maintainance of the sets \textit{InPrimary}

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

- The first edge insertion $u \xrightarrow{\alpha} x$ leads to $u \in \text{InPrimary}[x][\alpha]$.

- $u \xrightarrow{\alpha} y$ and $u \xrightarrow{\alpha} z$ do not modify $\text{InPrimary}$, as $x$, $y$ and $z$ belong to the same PDSCC.

- On delete $u \xrightarrow{\alpha} x$, we move $u$ to $\text{InPrimary}[y][\alpha]$, thus $u$ can still be retrieved as a $\alpha$-neighbor of the PDSCC $\{y, z\}$.
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
Class Node{
    Node f;
    Node g;
};

da.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

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

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

**Figure 7:** Context-sensitive Data-Dependence graph  [Tang et al. 2015]
Compared 3 algorithms

- **Offline**
  - Invoked after each update

- **Dynamic DataLog**
  - Each update modifies 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

[Bar charts showing data and alias analysis results for various benchmarks, with bars for Offline, DDlog, and Dynamic, measured in microsecs.]
Thank You!

Questions?
Appendix
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)