Paper Review: On-The-Fly Static Analysis via Dynamic Bidirected Dyck Reachability

Krishna et al., POPL '24.

Jung Hyun Kim

SoftSec Lab., KAIST

IS661 Spring, 2024

Dyck Reachability

Dyck Reachability

- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

Dyck Reachability

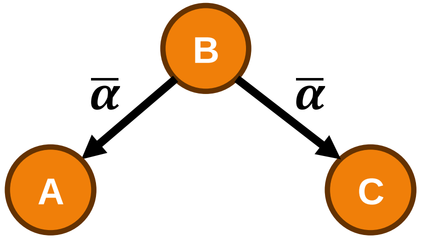
- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

$$\mathcal{I} \to \mathcal{I}\ \mathcal{I} \mid \alpha_1 \mathcal{I} \overline{\alpha}_1 \mid \cdots \mid \alpha_k \mathcal{I} \overline{\alpha}_k \mid \epsilon$$

Dyck Reachability

- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

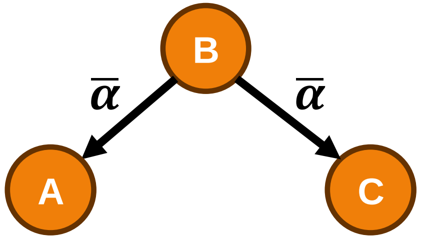
$$I \rightarrow I\ I \mid \alpha_1 I \overline{\alpha}_1 \mid \cdots \mid \alpha_k I \overline{\alpha}_k \mid \epsilon$$



Dyck Reachability

- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

$$I \rightarrow I\ I \mid \alpha_1 I \overline{\alpha}_1 \mid \cdots \mid \alpha_k I \overline{\alpha}_k \mid \epsilon$$

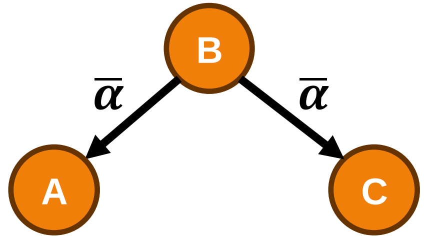


Bidirected graph

Dyck Reachability

- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

$$I \rightarrow I\ I \mid \alpha_1 I \overline{\alpha}_1 \mid \cdots \mid \alpha_k I \overline{\alpha}_k \mid \epsilon$$



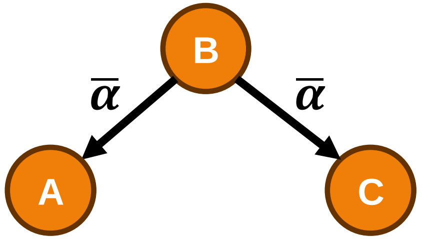
Bidirected graph

A → (α) → B → ($\overline{\alpha}$) → C

Dyck Reachability

- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

$$\mathcal{I} \rightarrow \mathcal{I}\ \mathcal{I} \mid \alpha_1 \mathcal{I} \overline{\alpha}_1 \mid \cdots \mid \alpha_k \mathcal{I} \overline{\alpha}_k \mid \epsilon$$



A → (α) → B → ($\overline{\alpha}$) → C

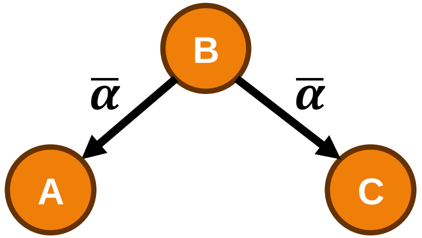
C → (α) → B → ($\overline{\alpha}$) → A

Bidirected graph

Dyck Reachability

- Dyck Reachability: two nodes have a path with labels following the grammar *I*:

$$I \rightarrow I\ I \mid \alpha_1 I \overline{\alpha}_1 \mid \cdots \mid \alpha_k I \overline{\alpha}_k \mid \epsilon$$



Bidirected graph

A → (α) → B → ($\overline{\alpha}$) → C

C → (α) → B → ($\overline{\alpha}$) → A

A, C: *Dyck reachable*

Dyck Reachability Being Widely Used

Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

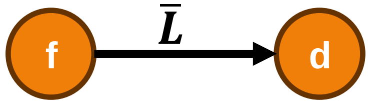
```
d = f.L
```

Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

```
d = f.L
```

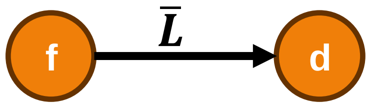


Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

```
d = f.L
f.L = c
```

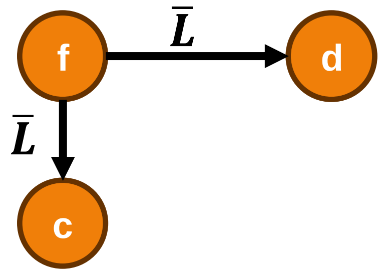


Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

```
d = f.L
f.L = c
```

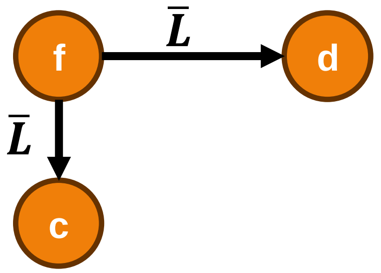


Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

```
d = f.L
f.L = c
e = f.L
```

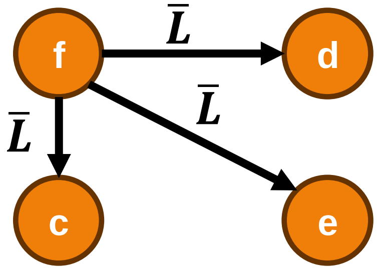


Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

```
d = f.L
f.L = c
e = f.L
```



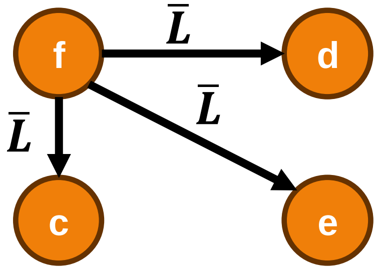
Dyck Reachability Being Widely Used

- Many analyses can be transformed into Dyck reachability problem: alias analysis, points-to analysis, etc.

- ex) field-sensitive alias analysis:

```
d = f.L
f.L = c
e = f.L
```

c, **d**, **e**: Dyck reachable



Dyck Reachability with Continuous Analysis

Dyck Reachability with Continuous Analysis

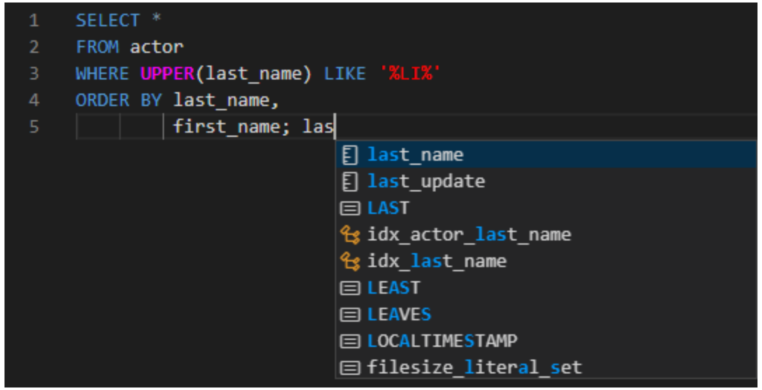
- Continuous (on-the-fly) analysis: an analysis to be run in *real-time* for *constant changes.*

Dyck Reachability with Continuous Analysis

- Continuous (on-the-fly) analysis: an analysis to be run in *real-time* for *constant changes.*
  - We may run *offline* algorithms every time a change occurs.

Dyck Reachability with Continuous Analysis

- Continuous (on-the-fly) analysis: an analysis to be run in *real-time* for *constant changes.*
  - We may run *offline* algorithms every time a change occurs.

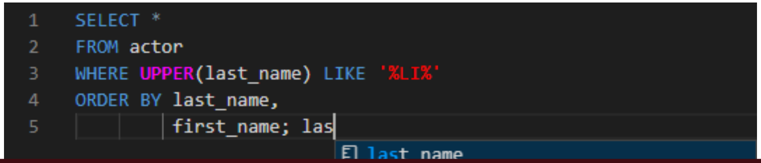


```
SELECT *
FROM actor
WHERE UPPER(last_name) LIKE '%LI%'
ORDER BY last_name,
         first_name; las
```

Type hints from an IDE

Dyck Reachability with Continuous Analysis

- Continuous (on-the-fly) analysis: an analysis to be run in *real-time* for *constant changes.*
  - We may run *offline* algorithms every time a change occurs.



```
SELECT *
FROM actor
WHERE UPPER(last_name) LIKE '%LI%'
ORDER BY last_name,
         first_name; las
                    last_name
```

Is *offline* algorithm efficient for constant changes?

Type hints from an IDE

Time Complexity of Dyck Reachability

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

Time Complexity of Dyck Reachability

	Chatterjee[a]	Krishna
Offline time	$O(m + n \cdot \alpha(n))$	
Online time	$O(m^2 + m \cdot n \cdot \alpha(n))$	

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

Time Complexity of Dyck Reachability

Offline Algorithm

	Chatterjee[a]	Krishna
Offline time	$O(m + n \cdot \alpha(n))$	
Online time	$O(m^2 + m \cdot n \cdot \alpha(n))$	

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

Time Complexity of Dyck Reachability

Offline Algorithm

	Chatterjee[a]	Krishna
Offline time	$O(m + n \cdot \alpha(n))$	$\mathbf{O(m \cdot n \cdot \alpha(n))}$
Online time	$O(m^2 + m \cdot n \cdot \alpha(n))$	

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

Time Complexity of Dyck Reachability

Offline Algorithm

	Chatterjee[a]	Krishna
Offline time	$O(m + n \cdot \alpha(n))$	$\mathbf{O(m \cdot n \cdot \alpha(n))}$
Online time	$O(m^2 + m \cdot n \cdot \alpha(n))$	

Online time in worst cases ($m = n^2$)

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

Time Complexity of Dyck Reachability

Offline Algorithm

	Chatterjee[a]	**Krishna**
Offline time	$O(m + n \cdot \alpha(n))$	$\mathbf{O(m \cdot n \cdot \alpha(n))}$
Online time	$O(m^2 + m \cdot n \cdot \alpha(n))$	

Online time in worst cases ($m = n^2$)

→ $O(n^4)$ vs $O(n^3)$

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

Time Complexity of Dyck Reachability

Offline Algorithm

	Chatterjee[a]	**Krishna**
Offline time	$O(m + n \cdot \alpha(n))$	$\mathbf{O(m \cdot n \cdot \alpha(n))}$
Online time	$O(m^2 + m \cdot n \cdot \alpha(n))$	

Online time in worst cases ($m = n^2$)

→ $O(n^4)$ vs $O(n^3)$

→ *n times* faster than before!

[a] Chatterjee, Krishnendu, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-30.

The Core Idea of Krishna Algorithm

The Core Idea of Krishna Algorithm

- We cannot avoid *inevitable recalculation* of reachability.

The Core Idea of Krishna Algorithm

- We cannot avoid *inevitable recalculation* of reachability.
    - Then, *reduce* the recalculation cost as much as possible.

The Core Idea of Krishna Algorithm

- We cannot avoid *inevitable recalculation* of reachability.
  - Then, *reduce* the recalculation cost as much as possible.
- Reduce the cost by maintaining *primitive* information.

The Core Idea of Krishna Algorithm

- We cannot avoid *inevitable recalculation* of reachability.
    - Then, *reduce* the recalculation cost as much as possible.

- Reduce the cost by maintaining *primitive* information.
    - *DSCC*: SCC w/ Dyck reachability.

The Core Idea of Krishna Algorithm

- We cannot avoid *inevitable recalculation* of reachability.
  - Then, *reduce* the recalculation cost as much as possible.
- Reduce the cost by maintaining *primitive* information.
  - *DSCC*: SCC w/ Dyck reachability.
  - Maintain and reuse *PDSCC*s (Primary DSCC).

The Core Idea of Krishna Algorithm

- We cannot avoid *inevitable recalculation* of reachability.
  - Then, *reduce* the recalculation cost as much as possible.

- Reduce the cost by maintaining *primitive* information.
  - *DSCC*: SCC w/ Dyck reachability.
  - Maintain and reuse *PDSCC*s (Primary DSCC).

    → Try not to recalculate DSCCs from scratch.

Problem: Recalculation on Deletion

Problem: Recalculation on Deletion

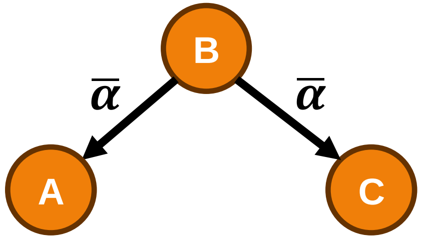
- We focus on *deletion* case.

Problem: Recalculation on Deletion

- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.

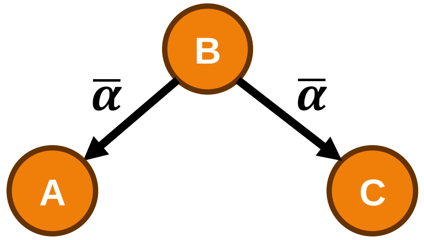
Problem: Recalculation on Deletion

- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.



Problem: Recalculation on Deletion

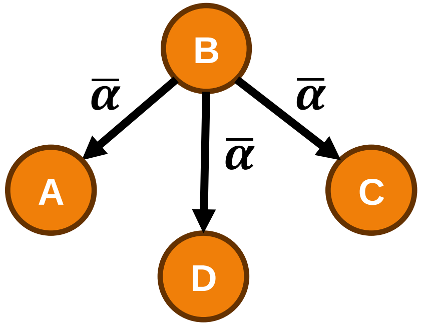
- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.



$DSCC_1$ = { A, C }

Problem: Recalculation on Deletion

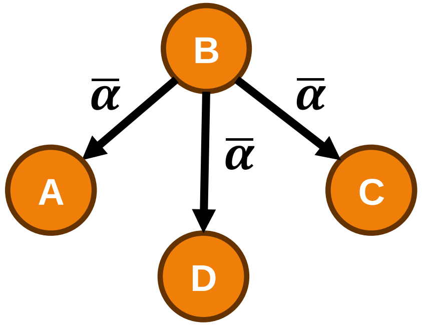
- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.



$DSCC_1$ = { A, C }

Problem: Recalculation on Deletion

- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.



$DSCC_1$ = { A, C, D }

Problem: Recalculation on Deletion

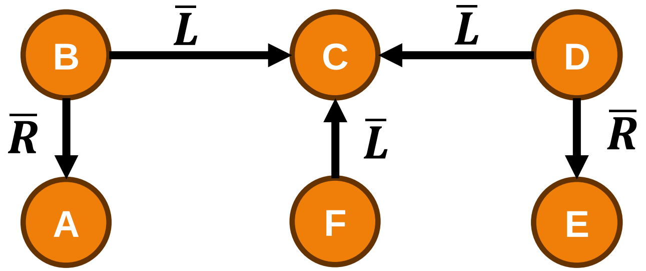
- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.

Problem: Recalculation on Deletion

- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.
- Deletion case necessitates (non-trivial) *recalculation* of DSCCs.

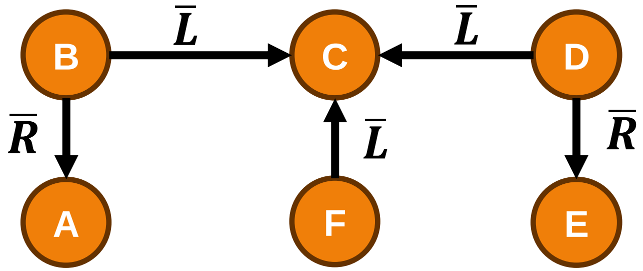
Problem: Recalculation on Deletion

- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.
- Deletion case necessitates (non-trivial) *recalculation* of DSCCs.



Problem: Recalculation on Deletion

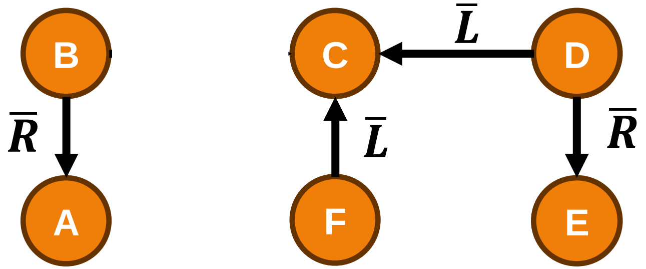
- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.
- Deletion case necessitates (non-trivial) *recalculation* of DSCCs.



$DSCC_1$ = { B, D, F }
$DSCC_2$ = { A, E }

Problem: Recalculation on Deletion

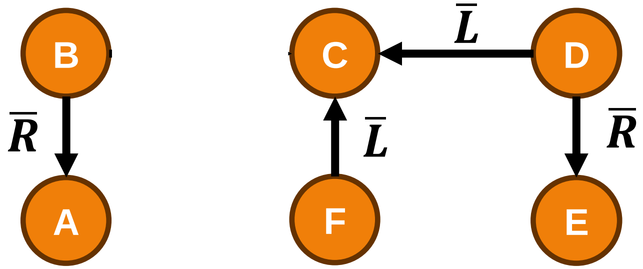
- We focus on *deletion* case.
    - *Insertion* case is trivial since we can just expand the DSCCs.
- Deletion case necessitates (non-trivial) *recalculation* of DSCCs.



DSCC$_1$ = { B, D, F }
DSCC$_2$ = { A, E }

Problem: Recalculation on Deletion

- We focus on *deletion* case.
  - *Insertion* case is trivial since we can just expand the DSCCs.
- Deletion case necessitates (non-trivial) *recalculation* of DSCCs.



DSCC_1 = { ~~B~~, D, F }
~~DSCC_2 = { A, E }~~

Use PDSCC to Split DSCC

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

$$E = \{ (p, q) \mid p, q \in V \wedge |path_{dyck}(p, q)| = 2 \}.$$

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where
$$E = \{ (p, q) \mid p, q \in V \land |path_{dyck}(p, q)| = 2 \}.$$

- Observation:

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

$$E = \{ (p, q) \mid p, q \in V \land |path_{dyck}(p, q)| = 2 \}.$$

- Observation:
  - *"DSCC consists of PDSCCs."*

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

$$E = \{ (p, q) \mid p, q \in V \wedge |path_{dyck}(p, q)| = 2 \}.$$

- Observation:
  - *"DSCC consists of PDSCCs."*
    - We can split the affected DSCCs into PDSCCs!

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

$$E = \{ (p, q) \mid p, q \in V \wedge |path_{dyck}(p, q)| = 2 \}.$$

- Observation:
  - *"DSCC consists of PDSCCs."*
    - We can split the affected DSCCs into PDSCCs!
  - *"We can over-approximate the candidates for DSCC splits."*

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

$$E = \{\ (p, q) \mid p, q \in V \land |path_{dyck}(p, q)| = 2\ \}.$$

- Observation:
  - *"DSCC consists of PDSCCs."*
    - We can split the affected DSCCs into PDSCCs!
  - *"We can over-approximate the candidates for DSCC splits."*
    - We can estimate and split the candidates!

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

$$E = \{ (p, q) \mid p, q \in V \wedge |path_{dyck}(p, q)| = 2 \}.$$

- Observation:
  - *"DSCC consists of PDSCCs."*
    - We can split the affected DSCCs into PDSCCs!
  - *"We can over-approximate the candidates for DSCC splits."*
    - We can estimate and split the candidates!
    - Will be discussed with an example.

Use PDSCC to Split DSCC

- *PDSCC*: a maximal CC in an *undirected* graph (V, E) where

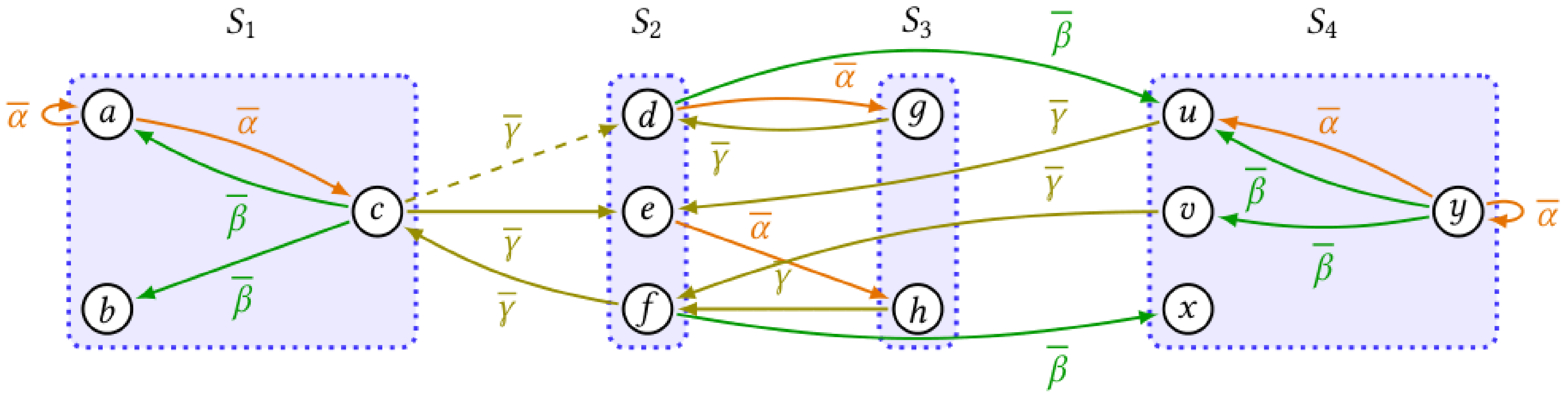
$$E = \{ (p, q) \mid p, q \in V \land |path_{dyck}(p, q)| = 2 \}.$$

- Observation:
  - *"DSCC consists of PDSCCs."*
    - We can split the affected DSCCs into PDSCCs!
  - *"We can over-approximate the candidates for DSCC splits."*
    - We can estimate and split the candidates!
    - Will be discussed with an example.

- After splitting DSCCs into PDSCCs, recalculate the fixpoint.

Example on Deletion

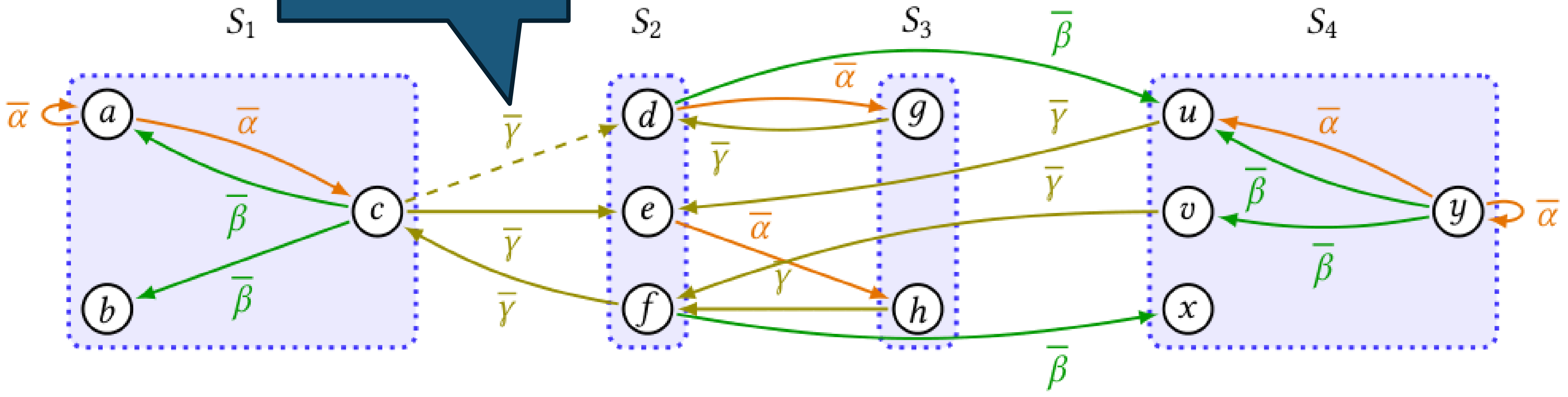
Example on Deletion



⬚ : DSCC

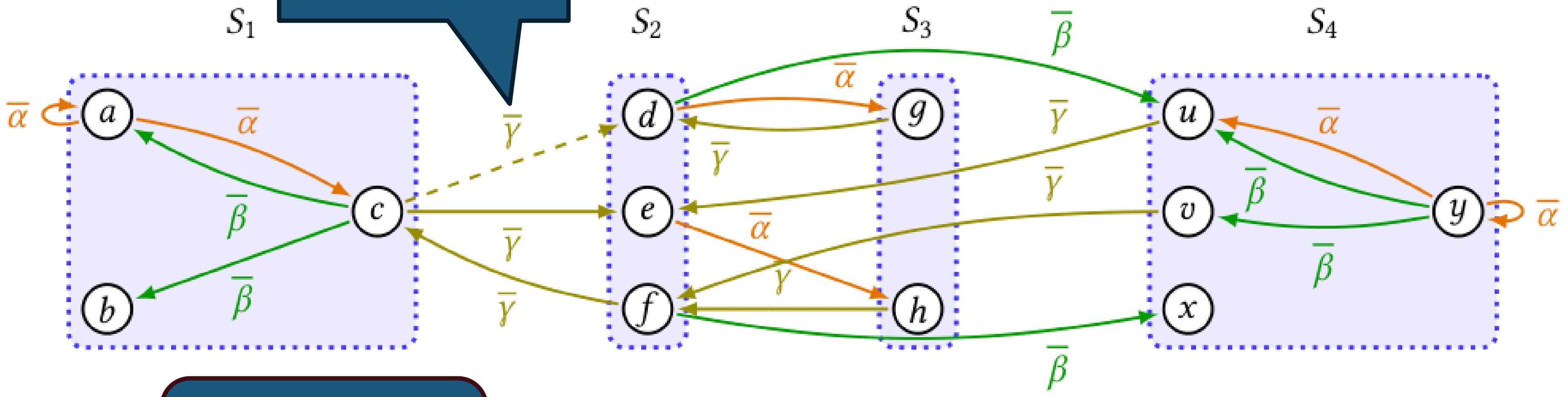
Example on Deletion

Deletion of *c* → *d*



Example on Deletion

Deletion of *c* → *d*

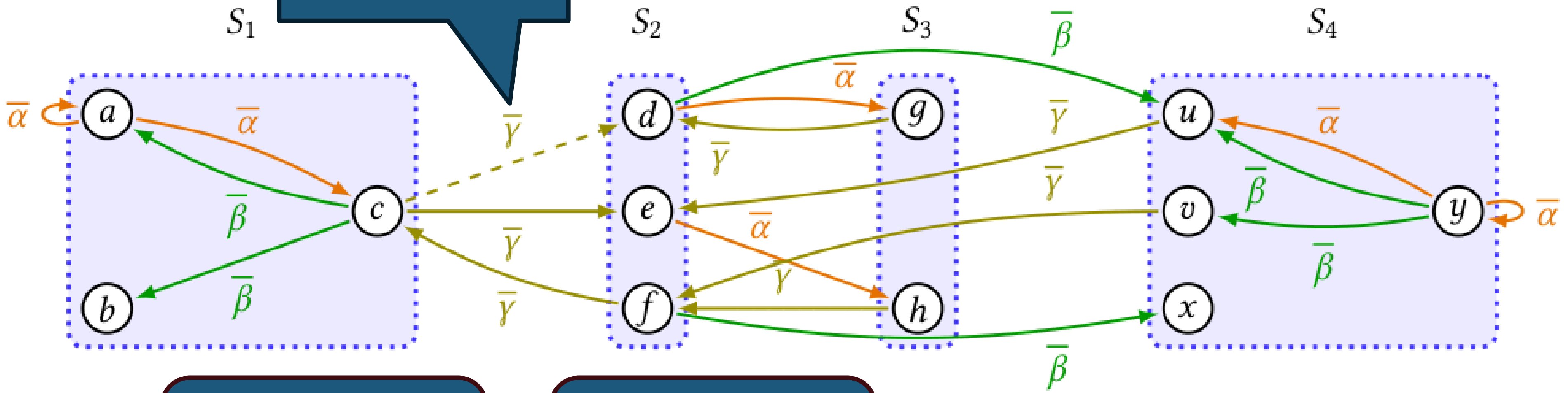


[Candidates]
S_2: DSCC(d) = S_2
S_3: d → g and e → h
S_4: d → u and f → x

⬚ : DSCC

Example on Deletion

Deletion of *c* → *d*



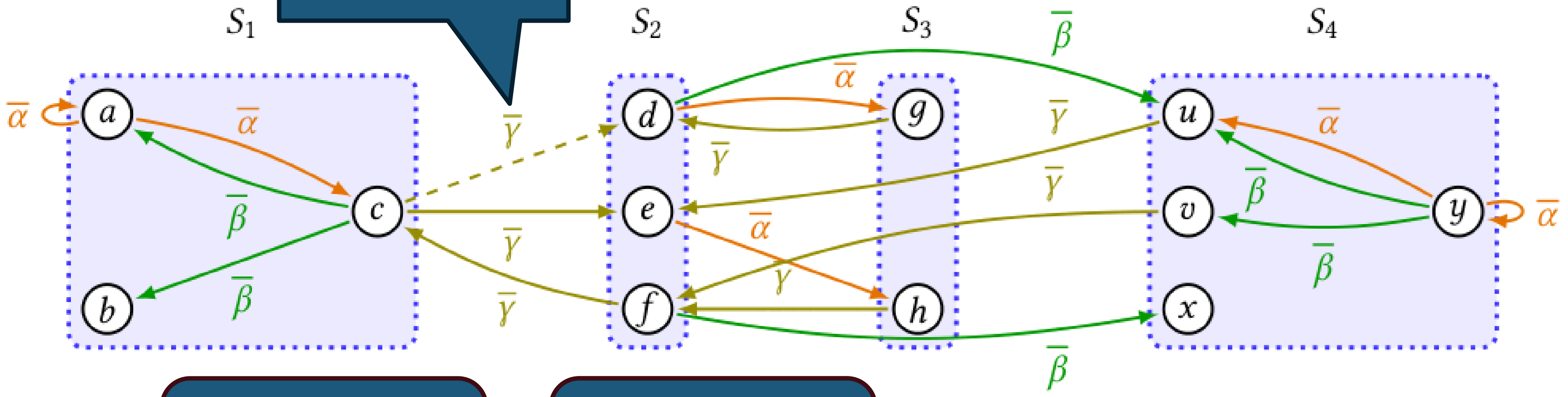
[Candidates]
S_2: DSCC(d) = S_2
S_3: d → g and e → h
S_4: d → u and f → x

[PDSCC]
P_1: { d, e }
P_2: { u, v, y }

▨ : DSCC

Example on Deletion

Deletion of *c* → *d*



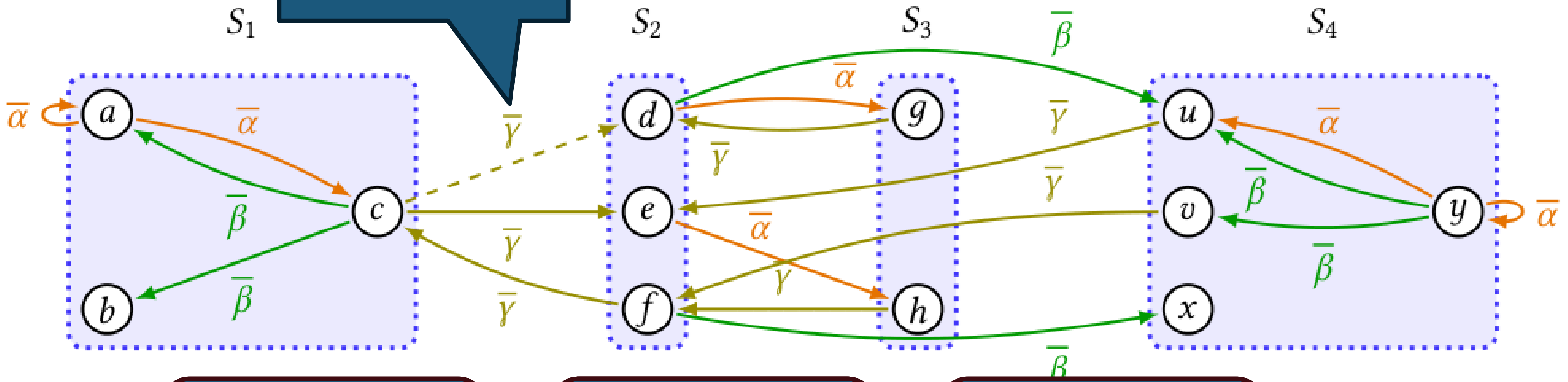
[Candidates]
S_2: DSCC(d) = S_2
S_3: d → g and e → h
S_4: d → u and f → x

[PDSCC]
~~P_1: { d, e }~~
P_2: { u, v, y }

▨ : DSCC

Example on Deletion

Deletion of $c \rightarrow d$



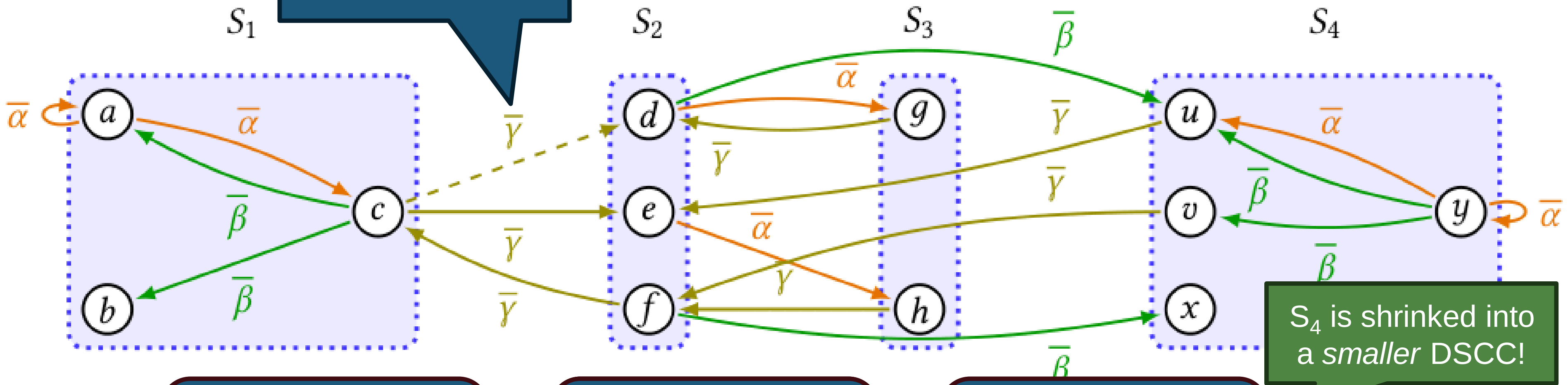
[Candidates]
S_2: DSCC(d) = S_2
S_3: d → g and e → h
S_4: d → u and f → x

[PDSCC]
~~P_1: { d, e }~~
P_2: { u, v, y }

[Prepared DSCC]
S_2 → (removed)
S_3 → (removed)
S_4 → { u, v, y }

: DSCC

Example on Deletion



[Candidates]
S_2: DSCC(d) = S_2
S_3: d → g and e → h
S_4: d → u and f → x

[PDSCC]
~~P_1: { d, e }~~
P_2: { u, v, y }

[Prepared DSCC]
S_2 → (removed)
S_3 → (removed)
S_4 → { u, v, y }

S_4 is shrinked into a *smaller* DSCC!

⬚ : DSCC

Evaluation Setup

Evaluation Setup

- Three configurations for evaluation:

Evaluation Setup

- Three configurations for evaluation:
  - Incremental updates only.

Evaluation Setup

- Three configurations for evaluation:
  - Incremental updates only.
  - Decremental updates only.

Evaluation Setup

- Three configurations for evaluation:
  - Incremental updates only.
  - Decremental updates only.
  - Both incremental and decremental updates.

Evaluation Setup

- Three configurations for evaluation:
  - Incremental updates only.
  - Decremental updates only.
  - Both incremental and decremental updates.

- Three evaluation targets:

Evaluation Setup

- Three configurations for evaluation:
  - Incremental updates only.
  - Decremental updates only.
  - Both incremental and decremental updates.

- Three evaluation targets:
  - Offline algorithm: running an offline algorithm on every edit.

Evaluation Setup

- Three configurations for evaluation:
  - Incremental updates only.
  - Decremental updates only.
  - Both incremental and decremental updates.

- Three evaluation targets:
  - Offline algorithm: running an offline algorithm on every edit.
  - DDlog: the state-of-the-art datalog engine.

Evaluation Setup

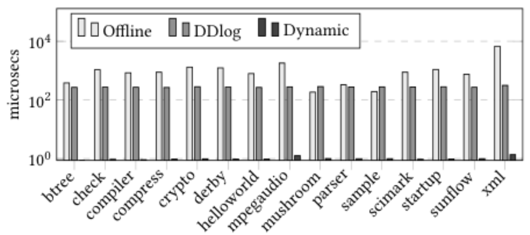
- Three configurations for evaluation:
  - Incremental updates only.
  - Decremental updates only.
  - Both incremental and decremental updates.

- Three evaluation targets:
  - Offline algorithm: running an offline algorithm on every edit.
  - DDlog: the state-of-the-art datalog engine.
  - Dynamic: the proposed method.

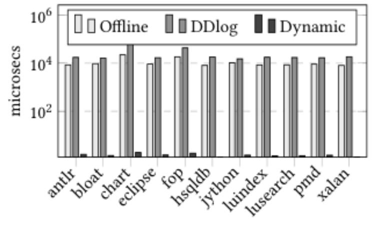
* The authors have not specified the details on the figures.

Data Dependence Analysis

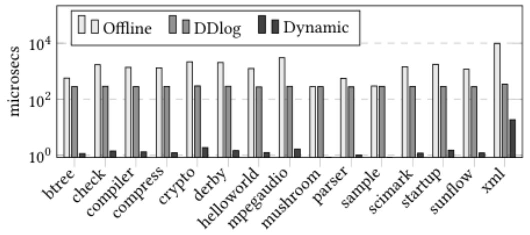
Alias Analysis



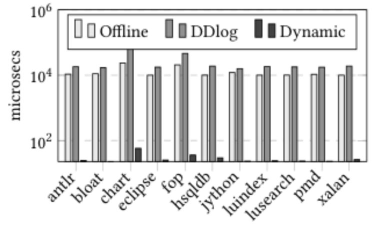
(a) Incremental updates.



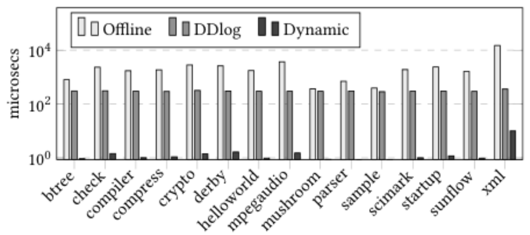
(b) Incremental updates.



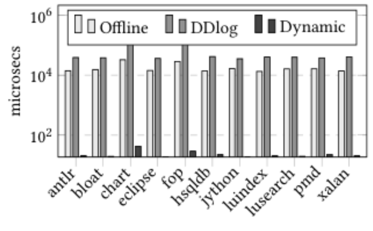
(c) Decremental updates.



(d) Decremental updates.

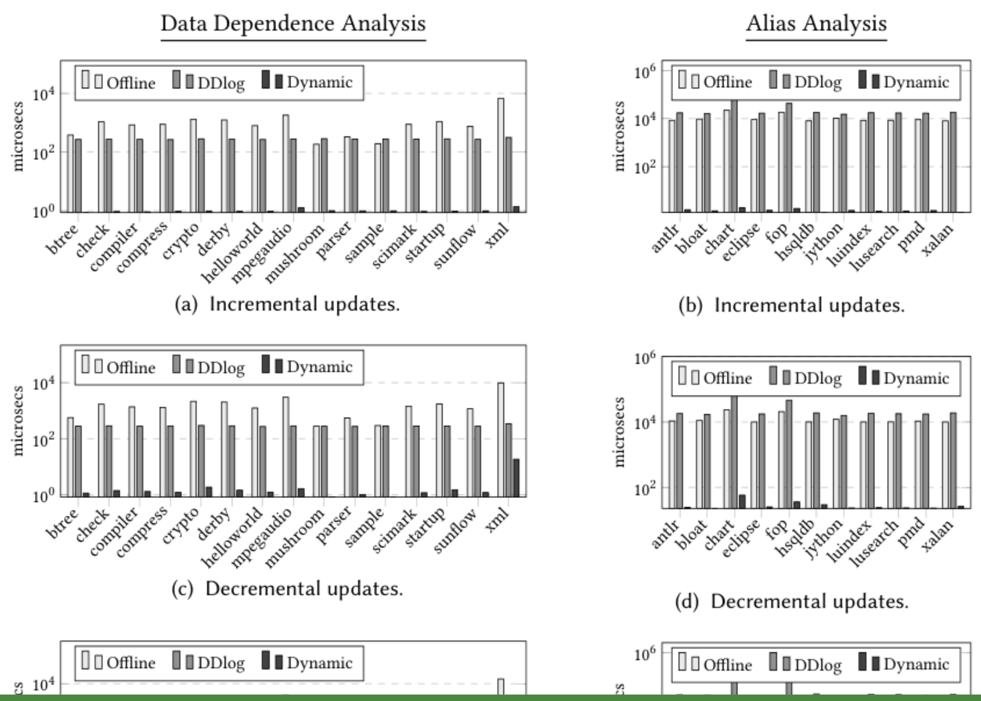


(e) Mixed updates.



(f) Mixed updates.

* The authors have not specified the details on the figures.



(a) Incremental updates.

(b) Incremental updates.

(c) Decremental updates.

(d) Decremental updates.

The online algorithm is much faster than the offline algorithm!

The authors have not specified the details on the figures.

Questions

Questions

- What about its memory usage?

Questions

- What about its memory usage?
- Why not compare against the *so-called wrong* algorithm?

[d] Ryzhyk, Leonid, and Mihai Budiu. "Differential Datalog." Datalog 2 (2019): 4-5.
[e] Fan, Zhiwei, Sunil Mallireddy, and Paraschos Koutris. "Towards Better Understanding of the Performance and Design of Datalog Systems." Datalog 2 (2022): 166-180.

Questions

- What about its memory usage?
- Why not compare against the *so-called wrong* algorithm?
- Is the datalog solver[d] used in this paper the SOTA[e]?

[d] Ryzhyk, Leonid, and Mihai Budiu. "Differential Datalog." Datalog 2 (2019): 4-5.
[e] Fan, Zhiwei, Sunil Mallireddy, and Paraschos Koutris. "Towards Better Understanding of the Performance and Design of Datalog Systems." Datalog 2 (2022): 166-180.

Summary

Summary

- Krishna’s algorithm is efficient in dynamically-changing CFGs.

Summary

- Krishna’s algorithm is efficient in dynamically-changing CFGs.
- To build an optimal online algorithm, we should observe some properties and introduce a *proper data structure* in order to optimize the *necessary recalculation* until reaching its fixpoint.

Summary

- Krishna's algorithm is efficient in dynamically-changing CFGs.
- To build an optimal online algorithm, we should observe some properties and introduce a *proper data structure* in order to optimize the *necessary recalculation* until reaching its fixpoint.
- Going forward from the previous step is sometimes better than going backward incrementally (c.f. datalog engines).

Question?