Fast Equivalence Relations in Datalog

Honours Project

Patrick Nappa

Supervisor: A. Prof. Bernhard Scholz

2018

THE UNIVERSITY OF SYDNEY

Soufflé & Datalog

Datalog is a declarative language - recent resurgence in popularity for large data sets

Programs are specified as *facts* and *rules* - *what*, not *how*
→ subset of predicate logic.

SOUFFLÉ is a high-performance Datalog compiler, converts to parallel C++ code

Designed for large-scale, static program analysis,
→ network analysis, security, and more!

Soufflé & Datalog

Datalog is a declarative language - recent resurgence in popularity for large data sets

Programs are specified as *facts* and *rules* - *what,* not *how*
→ subset of predicate logic.

SOUFFLÉ is a high-performance Datalog compiler, converts to parallel C++ code

Designed for large-scale, static program analysis,
→ network analysis, security, and more!

Soufflé & Datalog

Datalog is a declarative language - recent resurgence in popularity for large data sets

Programs are specified as *facts* and *rules* - *what,* not *how*
→ subset of predicate logic.

SOUFFLÉ is a high-performance Datalog compiler, converts to parallel C++ code

Designed for large-scale, static program analysis,
→ network analysis, security, and more!

Soufflé & Datalog

Datalog is a declarative language - recent resurgence in popularity for large data sets

Programs are specified as *facts* and *rules* - *what,* not *how*
→ subset of predicate logic.

SOUFFLÉ is a high-performance Datalog compiler, converts to parallel C++ code

Designed for large-scale, static program analysis,
→ network analysis, security, and more!

Soufflé & Datalog

Datalog is a declarative language - recent resurgence in popularity for large data sets

Programs are specified as *facts* and *rules* - *what,* not *how*
→ subset of predicate logic.

SOUFFLÉ is a high-performance Datalog compiler, converts to parallel C++ code

Designed for large-scale, static program analysis,
→ network analysis, security, and more!

Soufflé & Datalog

Datalog is a declarative language - recent resurgence in popularity for large data sets

Programs are specified as *facts* and *rules* - *what,* not *how*
→ subset of predicate logic.

SOUFFLÉ is a high-performance Datalog compiler, converts to parallel C++ code

Designed for large-scale, static program analysis,
→ network analysis, security, and more!

Equivalence Relations

Binary relation: *reflexivity*, *symmetry*, and *transitivity*.

Extremely common - often first Datalog program written

```
1 same_suburb(alice, bob).
2 same_suburb(bob, charlie).
3
4 same_suburb(X,X) :- same_suburb(X,_).
5 same_suburb(X,Y) :- same_suburb(Y,X).
6 same_suburb(X,Z) :- same_suburb(X,Y), same_suburb(Y,Z).
```

Single fact → a **lot** of new derived facts

Equivalence Relations

Binary relation: *reflexivity*, *symmetry*, and *transitivity*.

Extremely common - often first Datalog program written

```
1 same_suburb(alice, bob).
2 same_suburb(bob, charlie).
3
4 same_suburb(X,X) :- same_suburb(X,_).
5 same_suburb(X,Y) :- same_suburb(Y,X).
6 same_suburb(X,Z) :- same_suburb(X,Y), same_suburb(Y,Z).
```

Single fact → a **lot** of new derived facts

Equivalence Relations

Binary relation: *reflexivity*, *symmetry*, and *transitivity*.

Extremely common - often first Datalog program written

```
1 same_suburb(alice, bob).
2 same_suburb(bob, charlie).
3
4 same_suburb(X,X) :- same_suburb(X,_).
5 same_suburb(X,Y) :- same_suburb(Y,X).
6 same_suburb(X,Z) :- same_suburb(X,Y), same_suburb(Y,Z).
```

Single fact → a **lot** of new derived facts

Equivalence Relations

Binary relation: *reflexivity*, *symmetry*, and *transitivity*.

Extremely common - often first Datalog program written

```
1  same_suburb(alice, bob).
2  same_suburb(bob, charlie).
3
4  same_suburb(X,X) :- same_suburb(X,_).
5  same_suburb(X,Y) :- same_suburb(Y,X).
6  same_suburb(X,Z) :- same_suburb(X,Y), same_suburb(Y,Z).
```

Single fact → a **lot** of new derived facts

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Motivation

Memory:

Insertion of n pairs:

- Consequential insert of $\mathcal{O}(n^2)$ pairs
- Space complexity?

Time:

Insertion of a single pair:

- May trigger $\mathcal{O}(n)$ rounds of solving

Can we remove this overhead? Relatively unexplored optimisation

Disjoint Sets

Model equivalence relations efficiently with disjoint-sets

How? All elements within same disjoint-set are in the same equivalence class.

Disjoint set:

Equivalence pairs yielded:

(a, a)

$(b, b), (b, e), (b, f)$

$(e, b), (e, e), (e, f)$

$(f, b), (f, e), (f, f)$

$(c, c), (c, d), (d, c), (d, d)$

Integrating this with SOUFFLÉ ?

Disjoint Sets

Model equivalence relations efficiently with disjoint-sets

How? All elements within same disjoint-set are in the same equivalence class.

Disjoint set:

Equivalence pairs yielded:

(a, a)
$(b, b), (b, e), (b, f)$
$(e, b), (e, e), (e, f)$
$(f, b), (f, e), (f, f)$
$(c, c), (c, d), (d, c), (d, d)$

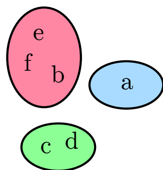
Integrating this with SOUFFLÉ ?

Disjoint Sets

Model equivalence relations efficiently with disjoint-sets

How? All elements within same disjoint-set are in the same equivalence class.

Disjoint set:



Equivalence pairs yielded:

(a, a)

$(b, b), (b, e), (b, f)$

$(e, b), (e, e), (e, f)$

$(f, b), (f, e), (f, f)$

$(c, c), (c, d), (d, c), (d, d)$

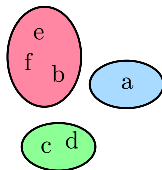
Integrating this with SOUFFLÉ ?

Disjoint Sets

Model equivalence relations efficiently with disjoint-sets

How? All elements within same disjoint-set are in the same equivalence class.

Disjoint set:



Equivalence pairs yielded:

(a, a)
$(b, b), (b, e), (b, f)$
$(e, b), (e, e), (e, f)$
$(f, b), (f, e), (f, f)$
$(c, c), (c, d), (d, c), (d, d)$

Integrating this with SOUFFLÉ ?

Data structure Design

Custom data-structure to support SOUFFLÉ 's mode of operations

Implicitly store equivalence relations with disjoint sets:

Data structure Design

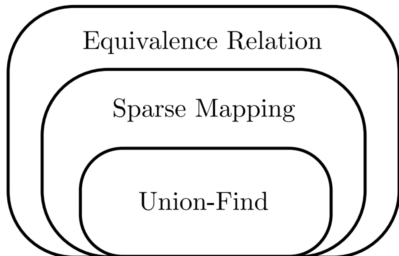
Custom data-structure to support SOUFFLÉ 's mode of operations

Implicitly store equivalence relations with disjoint sets:

Data structure Design

Custom data-structure to support SOUFFLÉ 's mode of operations

Implicitly store equivalence relations with disjoint sets:



- **Union-Find:** Store elements within disjoint sets with efficient operations
- **Sparse Mapping:** Provide value abstraction
- **Equivalence Relation:** Allow enumeration of all implicitly stored pairs

Union-Find

Union-Find; handle disjoint-set operations ($\alpha(n)$ cost)

Modified wait-free implementation of Anderson Union-Find, 1991

Extension of the algorithms for non-fixed domain
→ required a fast, concurrent list

Union-Find

Union-Find; handle disjoint-set operations ($\alpha(n)$ cost)

Modified wait-free implementation of Anderson Union-Find, 1991

Extension of the algorithms for non-fixed domain
→ required a fast, concurrent list

Union-Find

Union-Find; handle disjoint-set operations ($\alpha(n)$ cost)

Modified wait-free implementation of Anderson Union-Find, 1991

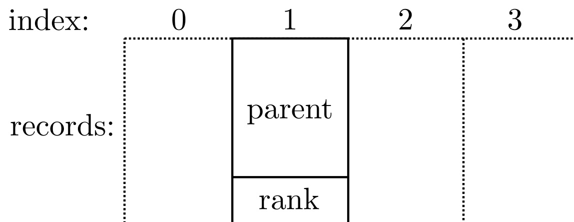
Extension of the algorithms for non-fixed domain
→ required a fast, concurrent list

Union-Find

Union-Find; handle disjoint-set operations ($\alpha(n)$ cost)

Modified wait-free implementation of Anderson Union-Find, 1991

Extension of the algorithms for non-fixed domain
→ required a fast, concurrent list



Sparse Mapping

Anderson's Union-Find: elements' value encoded by index

Sparse mapping to store *real* values in data structure

Bijection required to support internal data structure operations

Sparse Mapping

Anderson's Union-Find: elements' value encoded by index

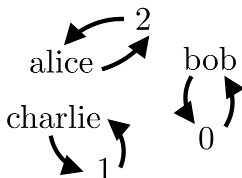
Sparse mapping to store *real* values in data structure

Bijection required to support internal data structure operations

Sparse Mapping

Anderson's Union-Find: elements' value encoded by index

Sparse mapping to store *real* values in data structure



Bijection required to support internal data structure operations

Sparse Mapping (2)

This bijection requires set-once semantics - no overwritten values

Sparse → Dense: concurrent hash-map

Dense → Sparse: concurrent list

Smart locking to ensure correctness
→ optimistic allocation
→ stratified locks

Sparse Mapping (2)

This bijection requires set-once semantics - no overwritten values

Sparse → Dense: concurrent hash-map

Dense → Sparse: concurrent list

Smart locking to ensure correctness
→ optimistic allocation
→ stratified locks

Sparse Mapping (2)

This bijection requires set-once semantics - no overwritten values

Sparse → Dense: concurrent hash-map

Dense → Sparse: concurrent list

Smart locking to ensure correctness
→ optimistic allocation
→ stratified locks

Sparse Mapping (2)

This bijection requires set-once semantics - no overwritten values

Sparse → Dense: concurrent hash-map

Dense → Sparse: concurrent list

Smart locking to ensure correctness
→ optimistic allocation
→ stratified locks

Sparse Mapping (2)

This bijection requires set-once semantics - no overwritten values

Sparse → Dense: concurrent hash-map

Dense → Sparse: concurrent list

Smart locking to ensure correctness
→ optimistic allocation
→ stratified locks

Sparse Mapping (2)

This bijection requires set-once semantics - no overwritten values

Sparse → Dense: concurrent hash-map

Dense → Sparse: concurrent list

Smart locking to ensure correctness
→ optimistic allocation
→ stratified locks

Equivalence Mapping

A cache to iterate over the disjoint-set efficiently

Partition each equivalence class into separate lists - $\mathcal{O}(n \cdot \alpha(n))$

Extracting pairs is a pair-wise closure over each list

Volatile: any inserts requires rebuilding

Equivalence Mapping

A cache to iterate over the disjoint-set efficiently

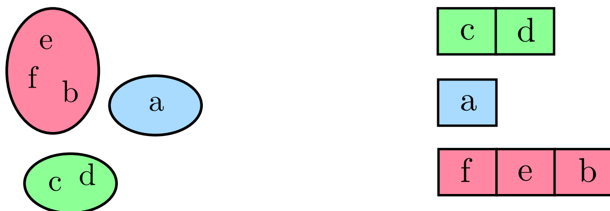
Partition each equivalence class into separate lists - $\mathcal{O}(n \cdot \alpha(n))$

Extracting pairs is a pair-wise closure over each list

Volatile: any inserts requires rebuilding

Equivalence Mapping

A cache to iterate over the disjoint-set efficiently



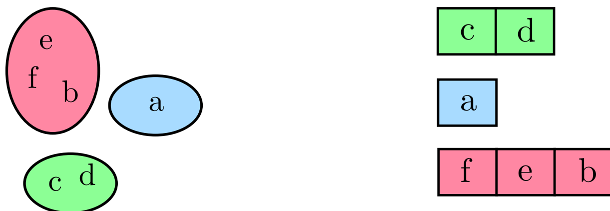
Partition each equivalence class into separate lists - $\mathcal{O}(n \cdot \alpha(n))$

Extracting pairs is a pair-wise closure over each list

Volatile: any inserts requires rebuilding

Equivalence Mapping

A cache to iterate over the disjoint-set efficiently



Partition each equivalence class into separate lists - $\mathcal{O}(n \cdot \alpha(n))$

Extracting pairs is a pair-wise closure over each list

Volatile: any inserts requires rebuilding

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

Datalog Evaluation

Bottom-up evaluation strategy
→ facts → rules → goal

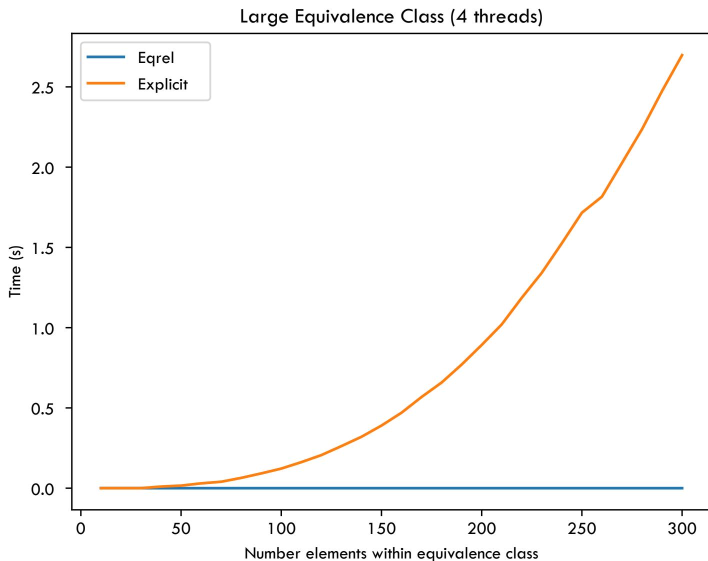
SOUFFLÉ 's semi-naïve evaluation strategy:
→ use new knowledge to derive new-new knowledge

Need to know what *new knowledge is*

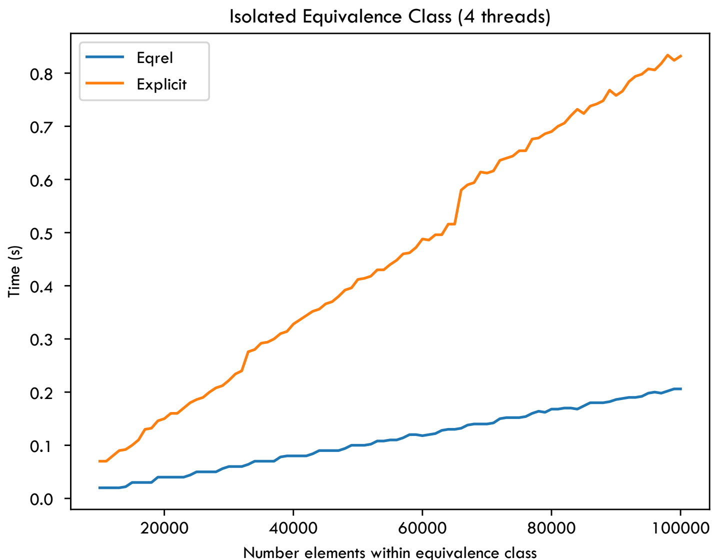
In an implicit storage system, we approximate:
→ new knowledge is stored as an equivalence relation
→ over-approximate the new knowledge

Drawback: over-approximation may cause all knowledge to be reconsidered

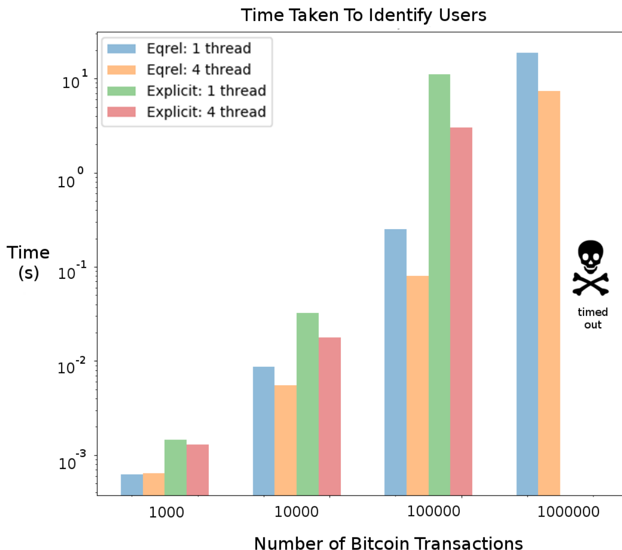
Results



Results



Results : Bitcoin Transactions



Future Work

Improve performance of Sparse Mapping
→ explore synchronisation strategies

Automatic detection of equivalence relations

Explore further data structure optimisations for relations

Future Work

Improve performance of Sparse Mapping
→ explore synchronisation strategies

Automatic detection of equivalence relations

Explore further data structure optimisations for relations

Future Work

Improve performance of Sparse Mapping
→ explore synchronisation strategies

Automatic detection of equivalence relations

Explore further data structure optimisations for relations

Future Work

Improve performance of Sparse Mapping
→ explore synchronisation strategies

Automatic detection of equivalence relations

Explore further data structure optimisations for relations

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in Soufflé
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in SOUFFLÉ
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in Soufflé
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in SOUFFLÉ
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in Soufflé
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in Soufflé
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in SOUFFLÉ
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in SOUFFLÉ
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in SOUFFLÉ
- Experiments, demonstrating efficacy

Conclusion

Using a data-structure to hold implicit information is efficient

Large equivalence classes:
→ Quadratic speed-up
→ Quadratic space improvement

Singleton equivalence classes:
→ Constant speed-up
→ Constant space improvement

Key Contribution:

- Extension of semi-naïve evaluation using equivalence relations
- Parallelised, layered data structure for equivalence relations, implemented in SOUFFLÉ
- Experiments, demonstrating efficacy

Cheers