PowerLog

Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing

Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng,
Rubao Lee, Xiaodong Zhang, Ge Yu

Northeastern University, China
The Ohio State University, USA


SIGMOD2020

What is a Recursive Aggregate Program(RAP)?

For a given algorithm defined by a function, it is a RAP if it satisfies:

(1) The function consists of a basic rule for an **initiation** and a small set of other rules for **iteratively self-calling** (recursion).

(2) The recursive part of the function includes **aggregate** operators, such as MIN, MAX, AVG, etc.

What is a Recursive Aggregate Program(RAP)?

For a given algorithm defined by a function, it is a RAP if it satisfies:

(1) The function consists of a basic rule for an **initiation** and a small set of other rules for **iteratively self-calling** (recursion).

(2) The recursive part of the function includes **aggregate** operators, such as MIN, MAX, AVG, etc.

Datalog perfectly expresses RAPs:

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy])     :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```

What is a Recursive Aggregate Program(RAP)?

For a given algorithm defined by a function, it is a RAP if it satisfies:

(1) The function consists of a basic rule for an **initiation** and a small set of other rules for **iteratively self-calling** (recursion).

(2) The recursive part of the function includes **aggregate** operators, such as MIN, MAX, AVG, etc.

Datalog perfectly expresses RAPs:

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy])     :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```

→ Initialization Rule

What is a Recursive Aggregate Program(RAP)?

For a given algorithm defined by a function, it is a RAP if it satisfies:

(1) The function consists of a basic rule for an **initiation** and a small set of other rules for **iteratively self-calling** (recursion).

(2) The recursive part of the function includes **aggregate** operators, such as MIN, MAX, AVG, etc.

Datalog perfectly expresses RAPs:

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy])  :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```

→ Initialization Rule

→ Recursive Rule

What is a Recursive Aggregate Program(RAP)?

For a given algorithm defined by a function, it is a RAP if it satisfies:

(1) The function consists of a basic rule for an **initiation** and a small set of other rules for **iteratively self-calling** (recursion).

(2) The recursive part of the function includes **aggregate** operators, such as MIN, MAX, AVG, etc.

Datalog perfectly expresses RAPs:

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy])     :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```

→ Initialization Rule

→ Recursive Rule

Aggregate Function

What is a Recursive Aggregate Program(RAP)?

For a given algorithm defined by a function, it is a RAP if it satisfies:

(1) The function consists of a basic rule for an **initiation** and a small set of other rules for **iteratively self-calling** (recursion).

(2) The recursive part of the function includes **aggregate** operators, such as MIN, MAX, AVG, etc.

Datalog perfectly expresses RAPs:

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy])     :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```

→ Initialization Rule

→ Recursive Rule

Aggregate Function

Non-aggregate Functions

Recursive Aggregate Programs are Everywhere

- Connected Components

```
cc(X,X)              :-  edge (X,_).
cc(Y,min [v])    : - cc(X,v), edge (X,Y).
```

Recursive Aggregate Programs are Everywhere

- Connected Components

```
cc(X,X)            :-  edge (X,_).
cc(Y,min [v])    : - cc(X,v), edge (X,Y).
```

- PageRank

```
rank(i+1, Y, sum[ry])    :-   node(Y), ry=0.2;
                          :-   rank(i, X, rx), edge(X,Y),
                          :-  degree(X, d), ry=0.8 rx/d.
```

Recursive Aggregate Programs are Everywhere

- Connected Components

```
cc(X,X)            :-  edge (X,_).
cc(Y,min [v])    : - cc(X,v), edge (X,Y).
```

- PageRank

```
rank(i+1, Y, sum[ry])     :-   node(Y), ry=0.2;
                          :-   rank(i, X, rx), edge(X,Y),
                          :-  degree(X, d), ry=0.8 rx/d.
```

- Graph Convolution Neural Network

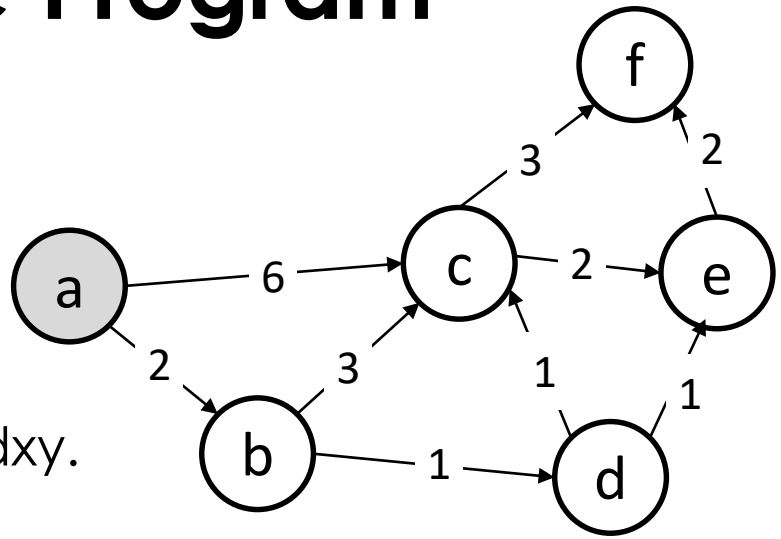
```
GCN (j+1,Y, sum [g1]) :- GCN (j,X,g), A(X,Y,w), Para (p),
                         p1 = relu(g*p) *w
```

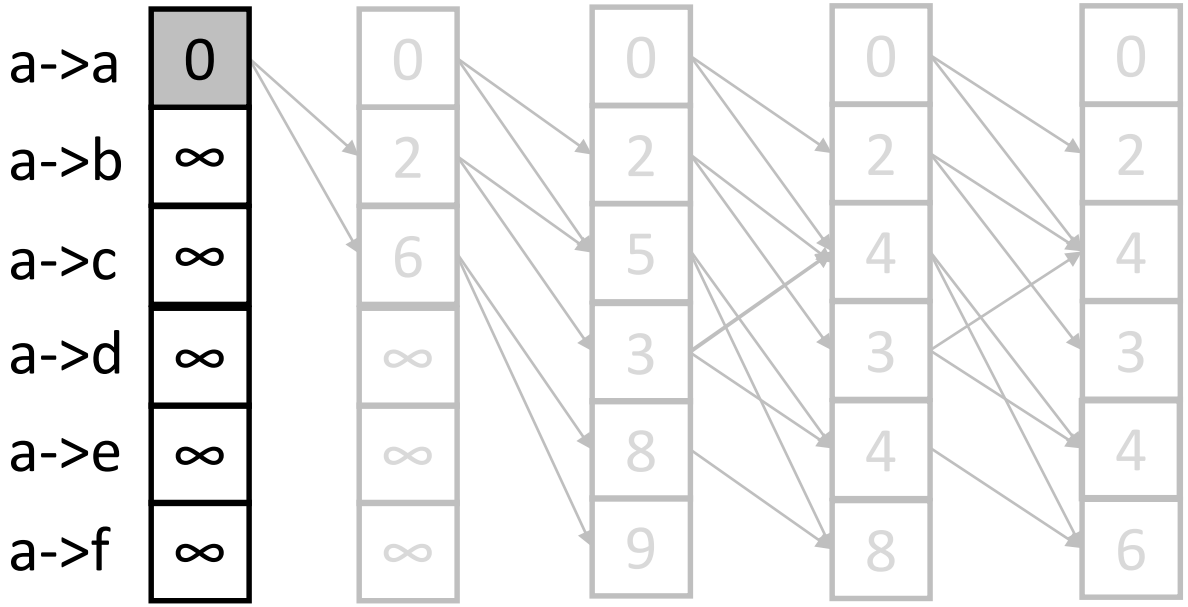
- And many others

Evaluating a Recursive Aggregate Program

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy] )  :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```

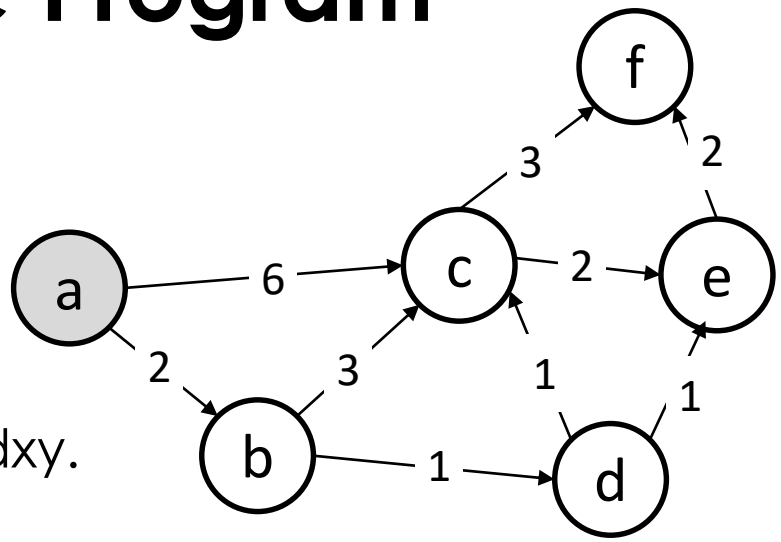


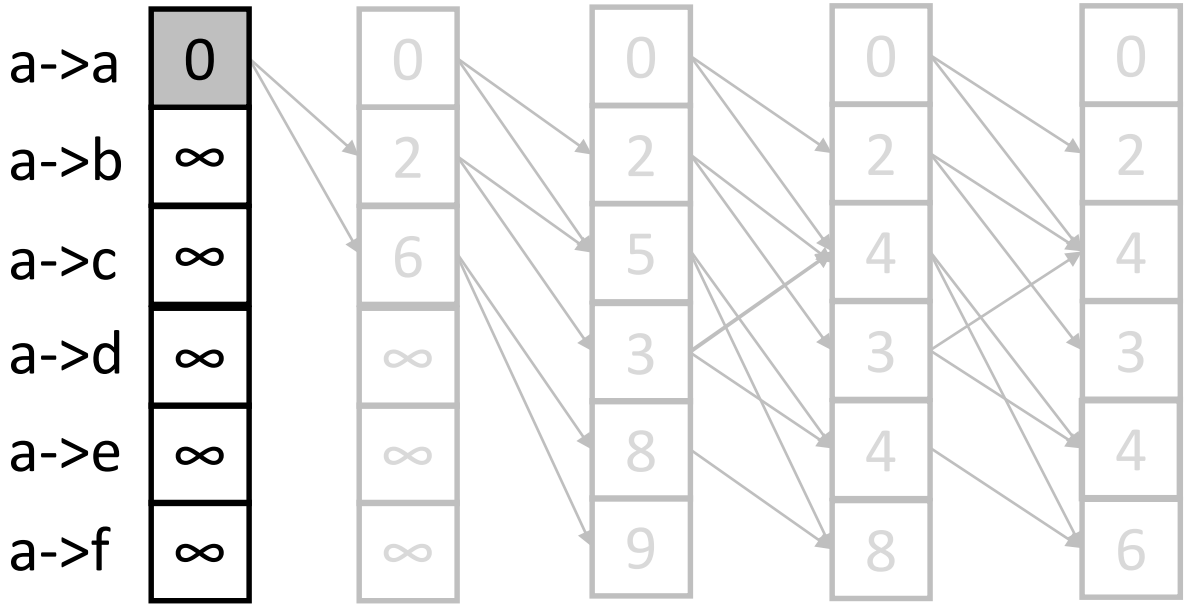


Evaluating a Recursive Aggregate Program

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy] )  :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```



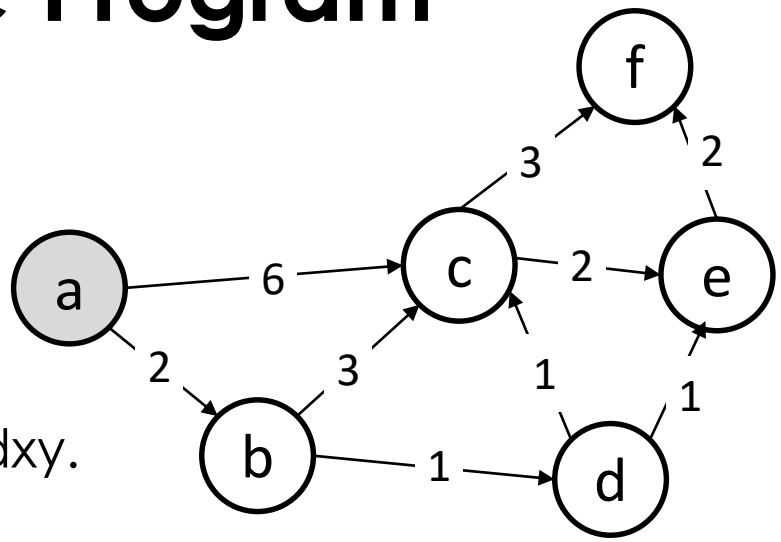


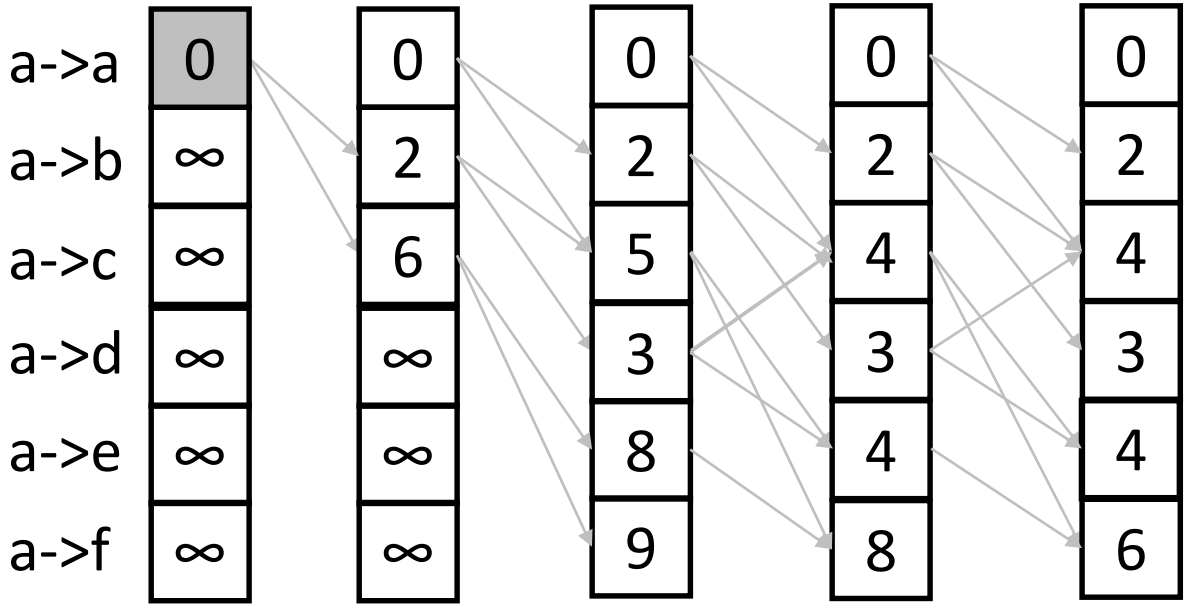
Iteratively performing aggregate and non aggregate function to derive result until the program reach the convergence.

Evaluating a Recursive Aggregate Program

Single Source Shortest Path(SSSP)

```
sssp(X, dx)          :-  X='a', dx=0.
sssp(Y, min[dy] )  :-  sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.
```





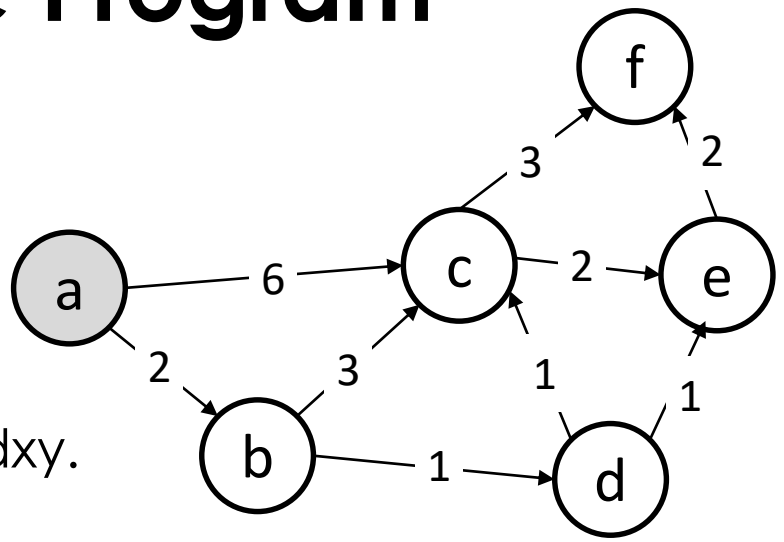
Iteratively performing aggregate and non aggregate function to derive result until the program reach the convergence.

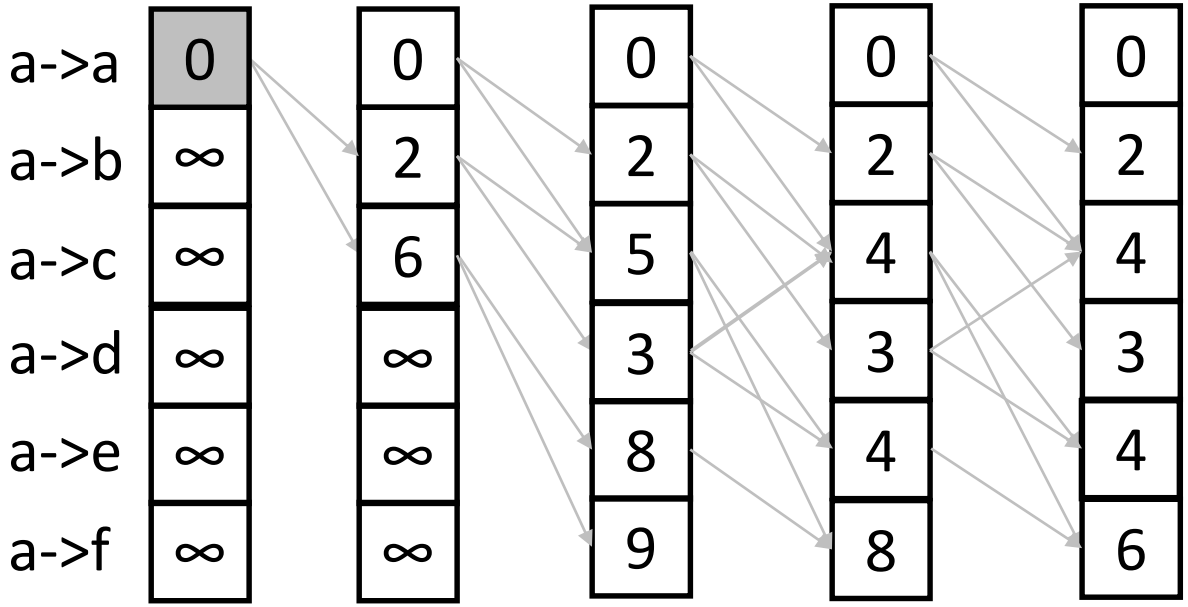
Evaluating a Recursive Aggregate Program

Single Source Shortest Path(SSSP)

sssp(X, dx) :- X='a', dx=0.

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy), dy=dx+dxy.





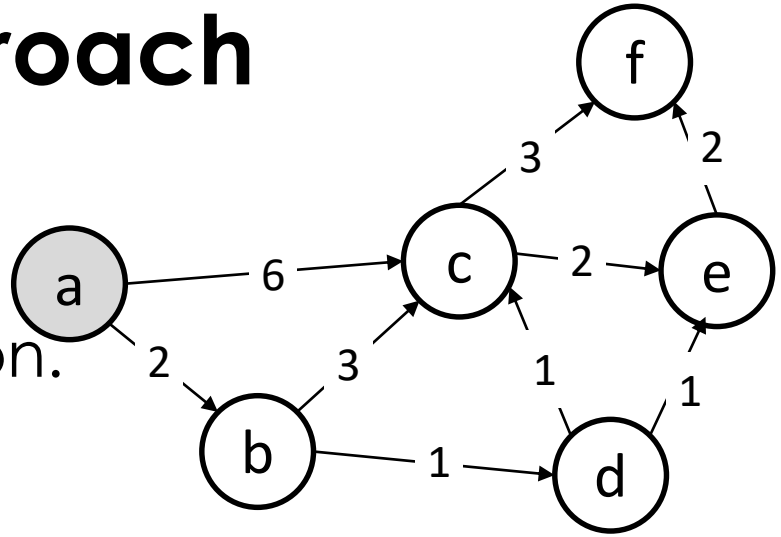
Iteratively performing aggregate and non aggregate function to derive result until the program reach the convergence.

Naïve Evaluation

Semi-naïve Evaluation

Naïve Evaluation: A Direct Approach

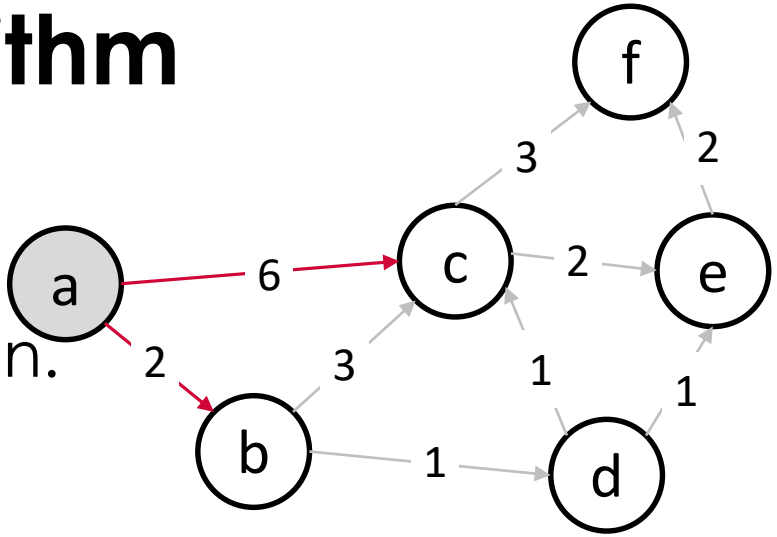
Naïve Evaluation is a direct and intuitive solution.

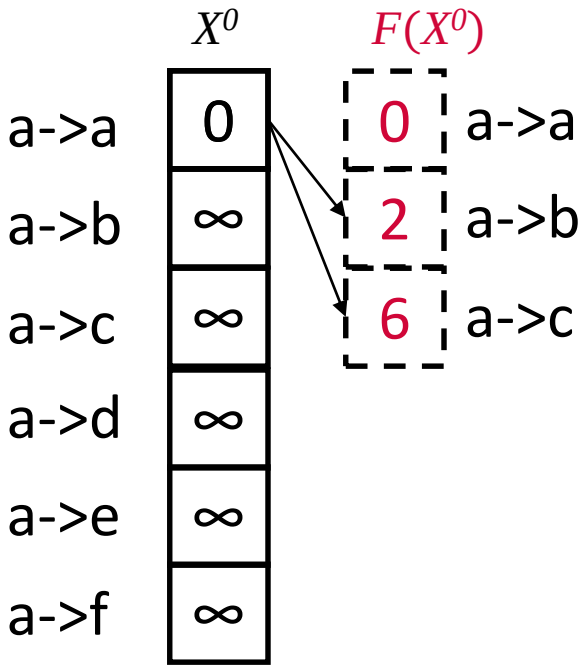


	X^0
a->a	0
a->b	∞
a->c	∞
a->d	∞
a->e	∞
a->f	∞

Naïve Evaluation on SSSP Algorithm

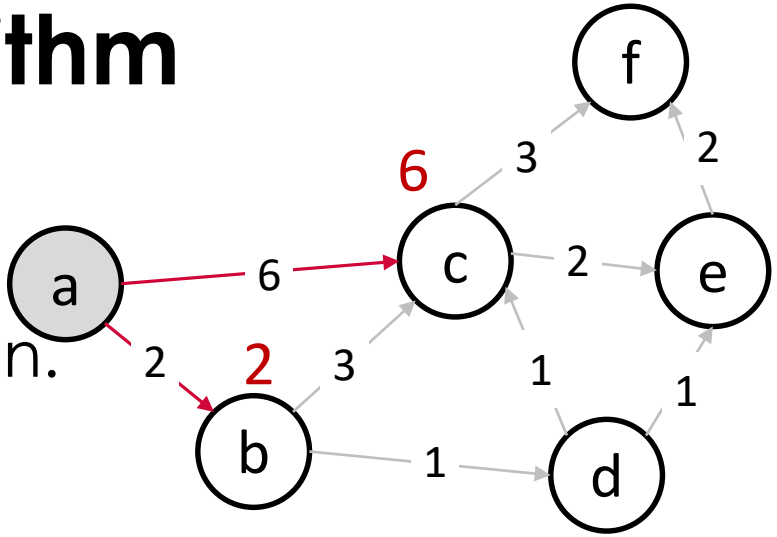
Naïve Evaluation is a direct and intuitive solution.

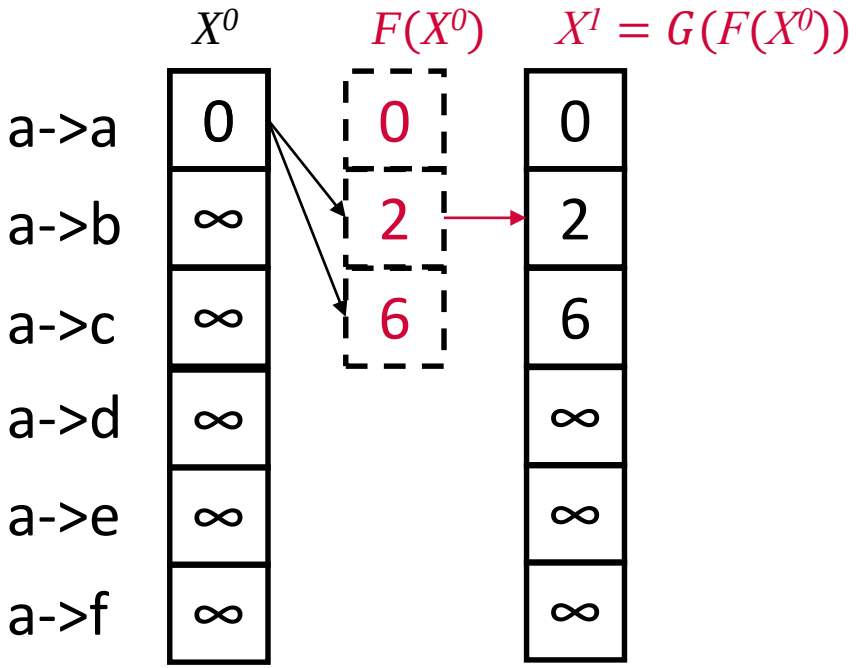




Naïve Evaluation on SSSP Algorithm

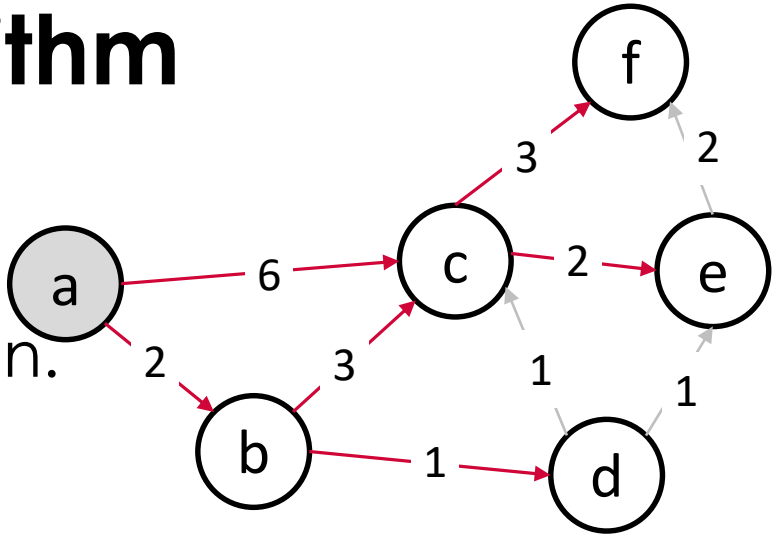
Naïve Evaluation is a direct and intuitive solution.

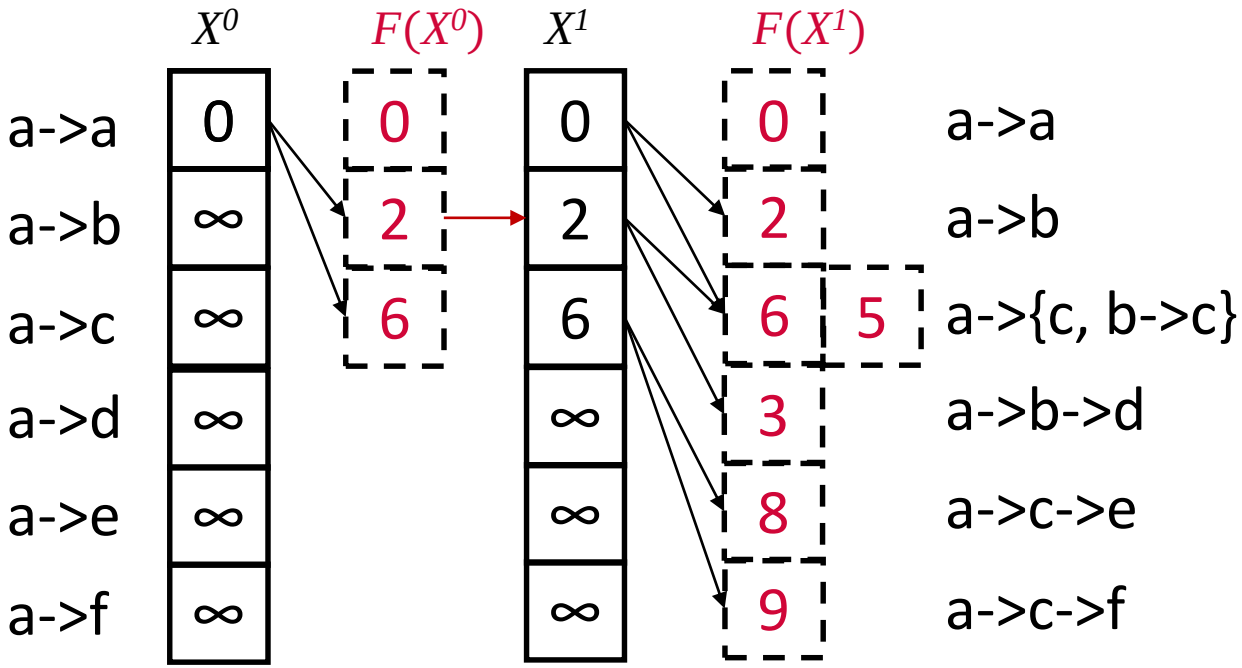




Naïve Evaluation on SSSP Algorithm

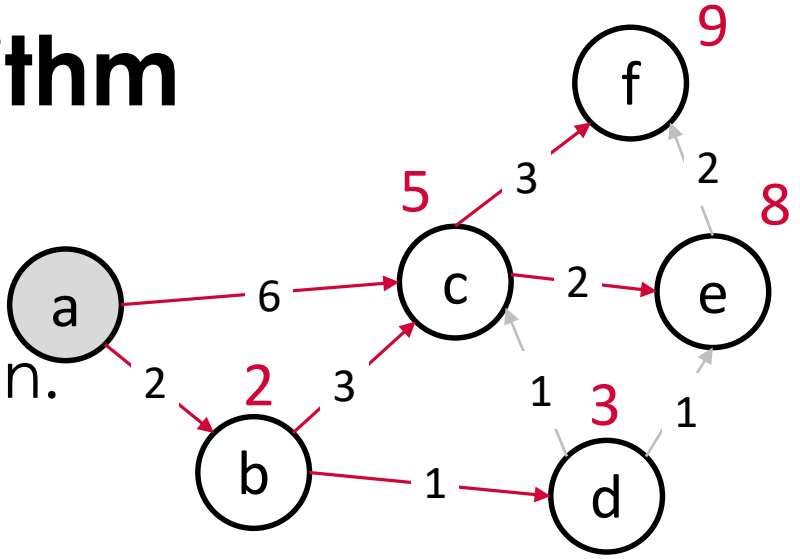
Naïve Evaluation is a direct and intuitive solution.

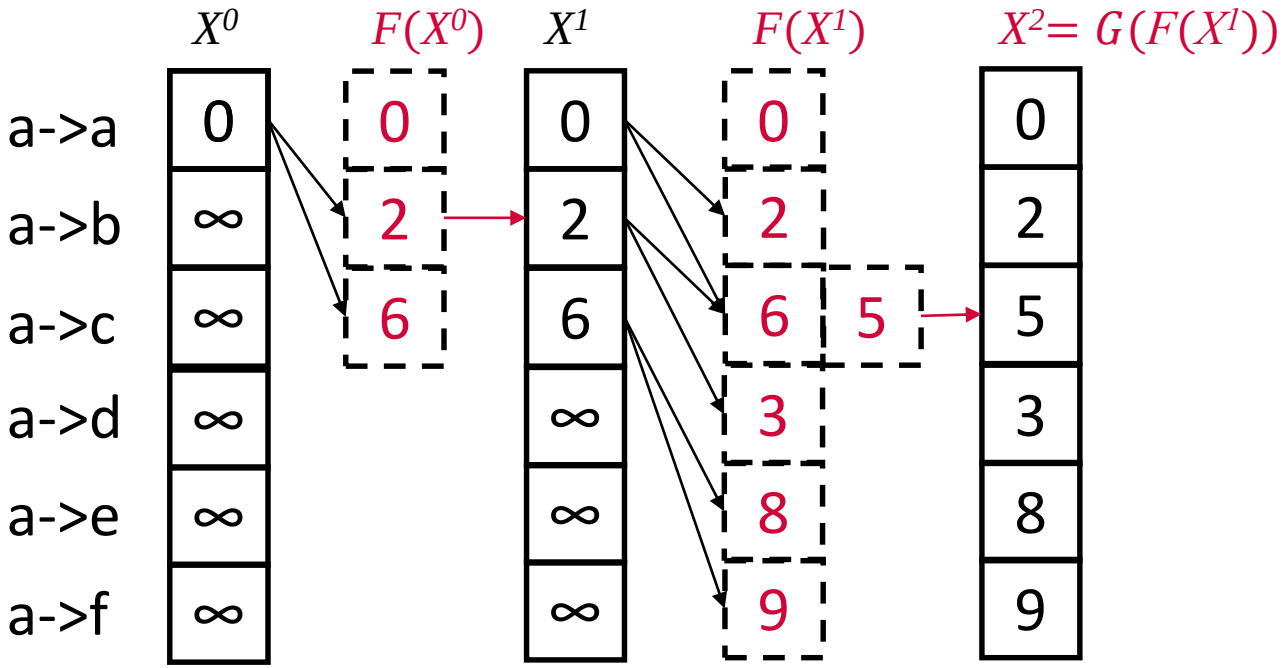




Naïve Evaluation on SSSP Algorithm

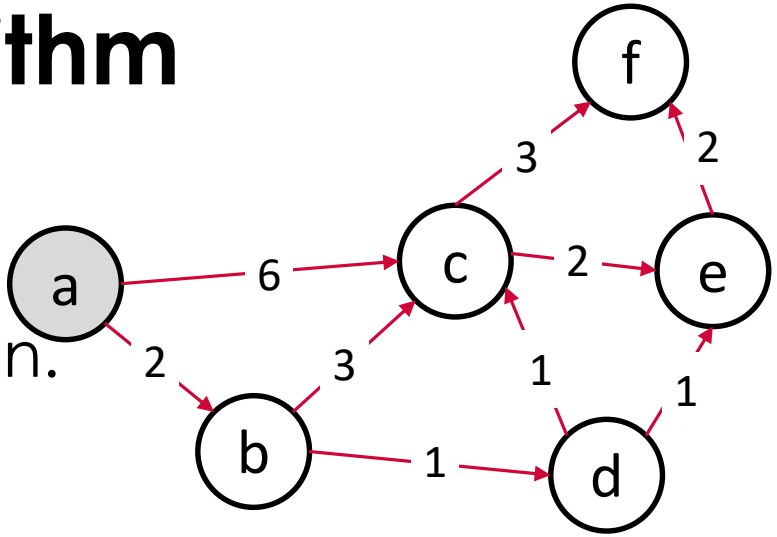
Naïve Evaluation is a direct and intuitive solution.

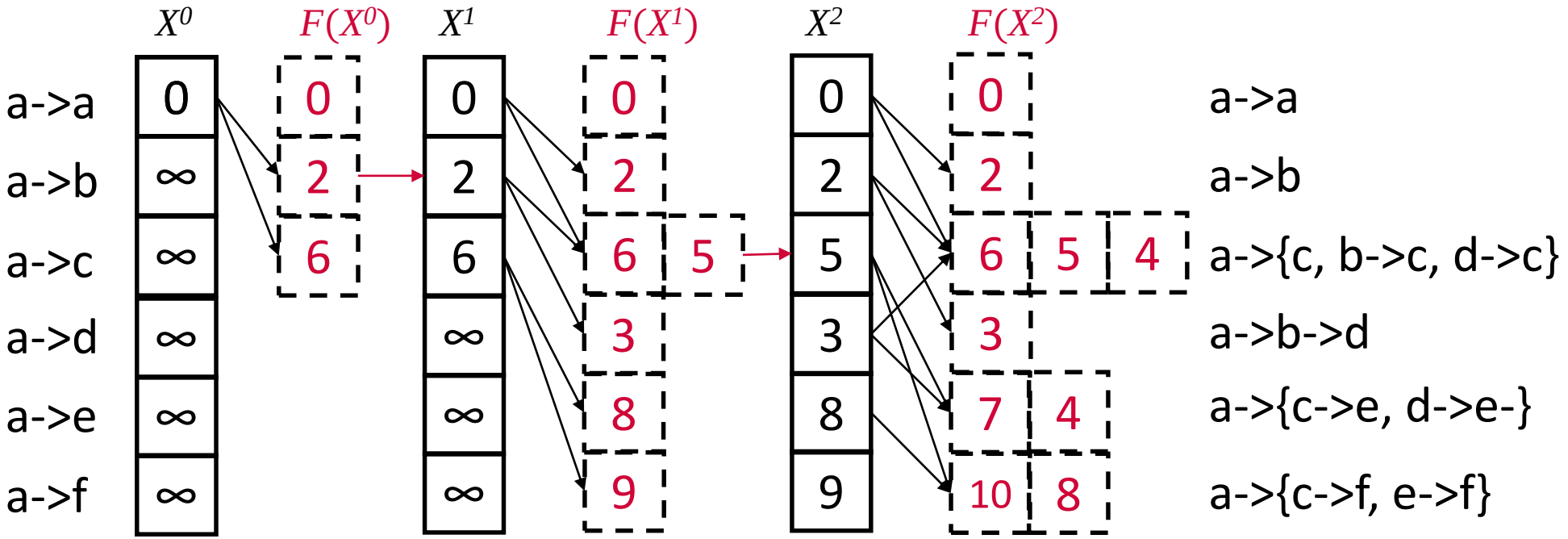




Naïve Evaluation on SSSP Algorithm

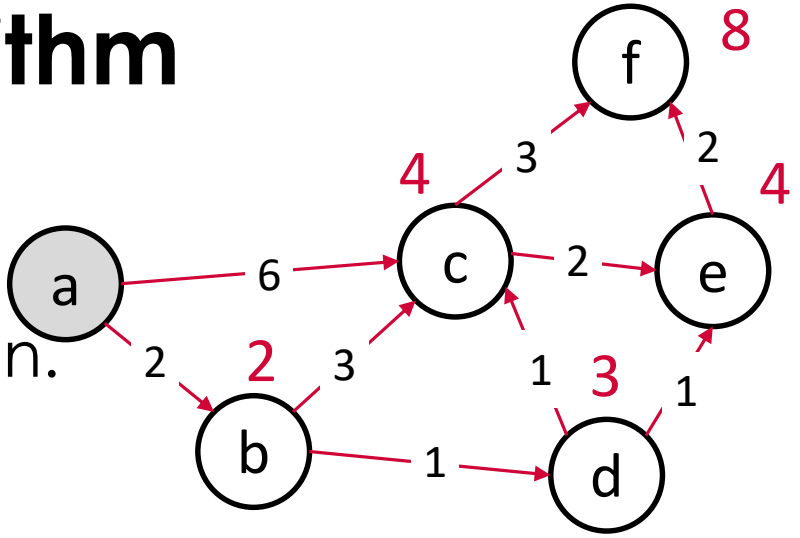
Naïve Evaluation is a direct and intuitive solution.

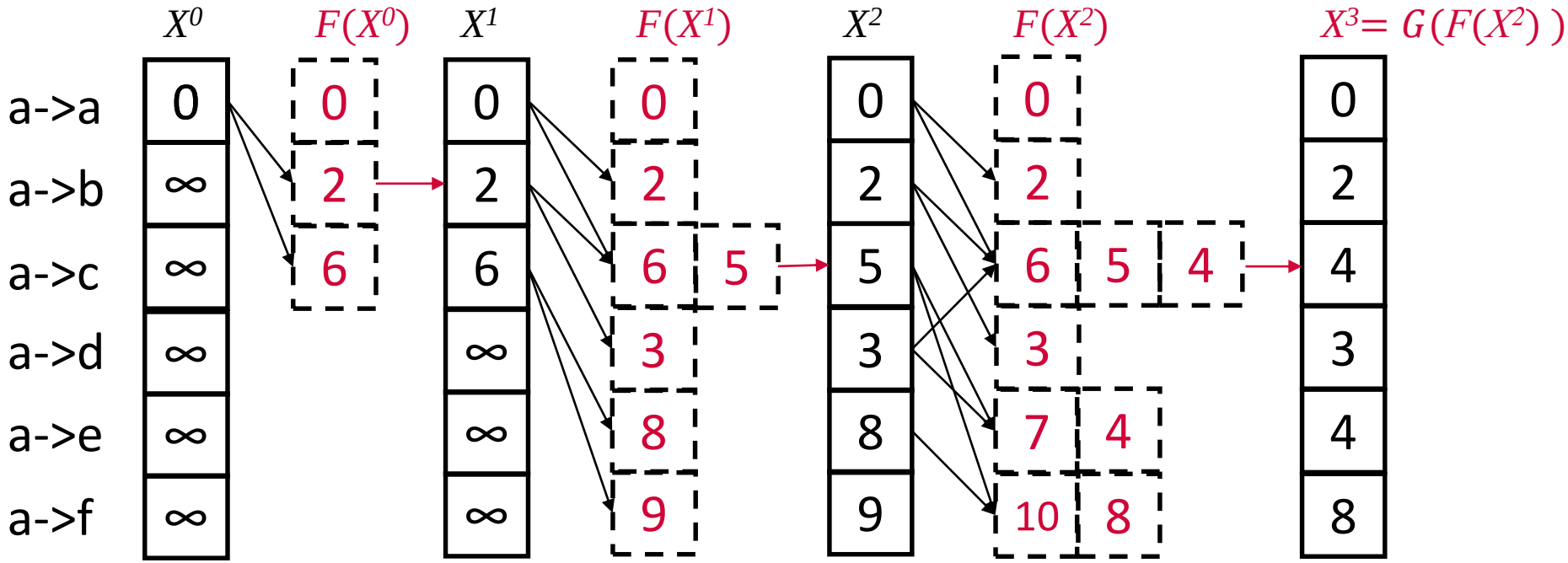




Naïve Evaluation on SSSP Algorithm

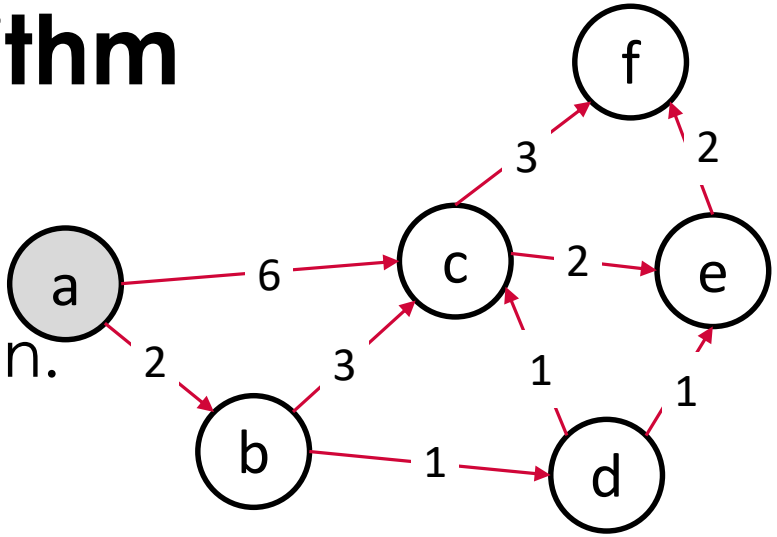
Naïve Evaluation is a direct and intuitive solution.

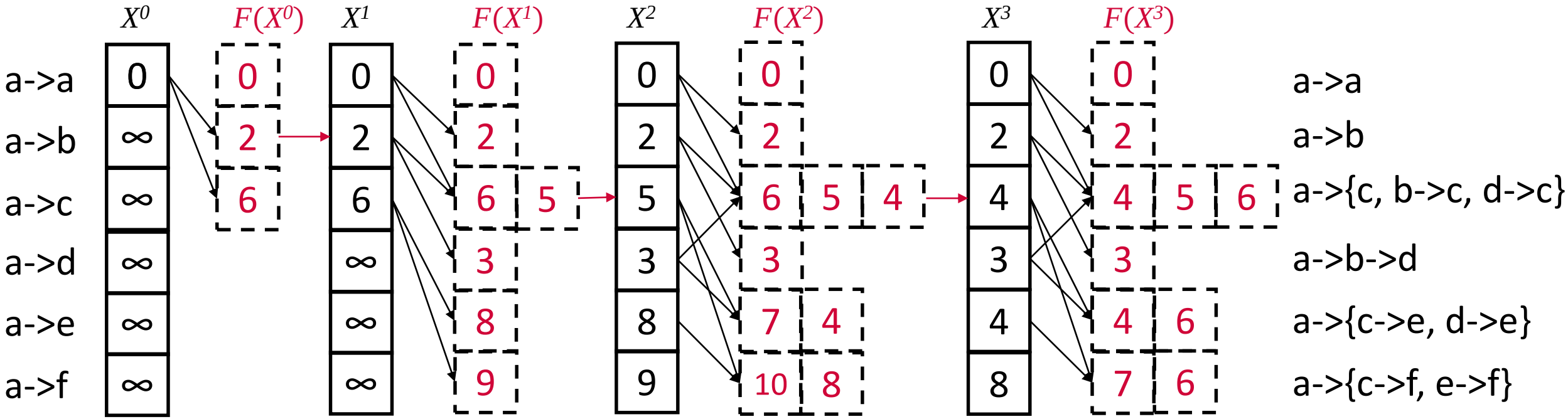




Naïve Evaluation on SSSP Algorithm

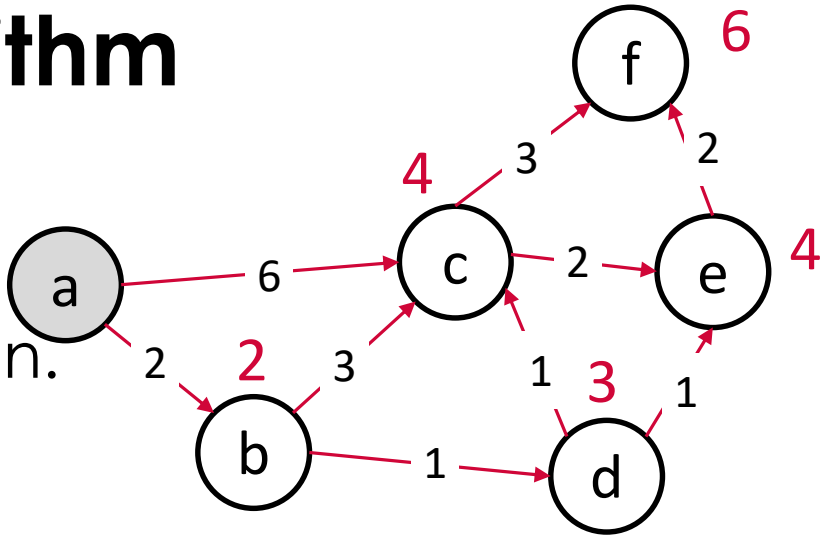
Naïve Evaluation is a direct and intuitive solution.

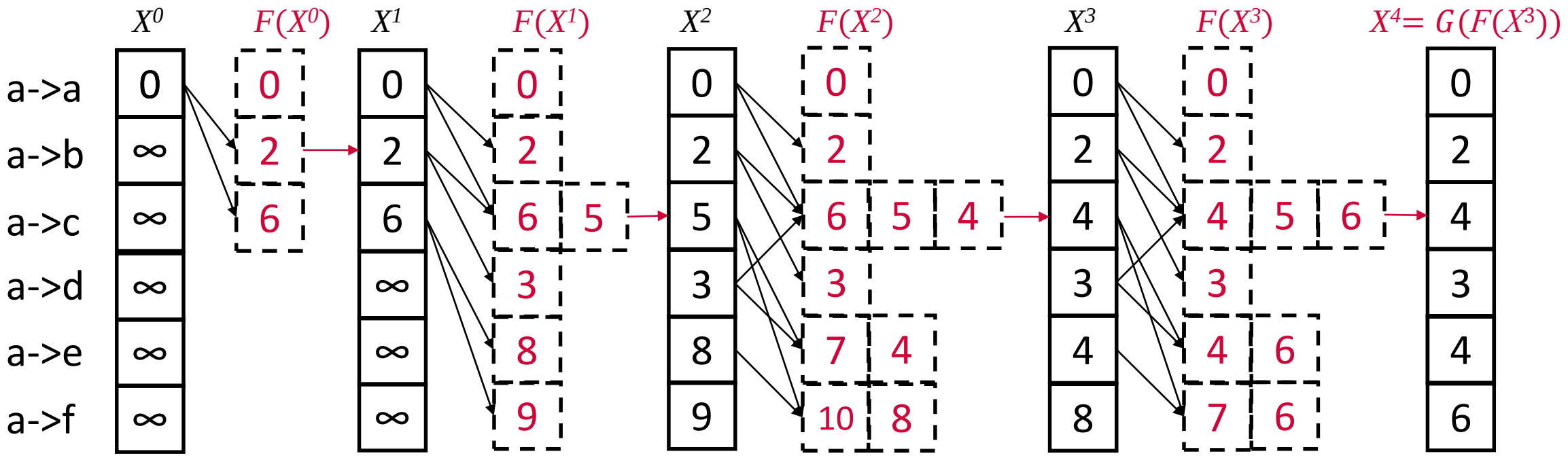




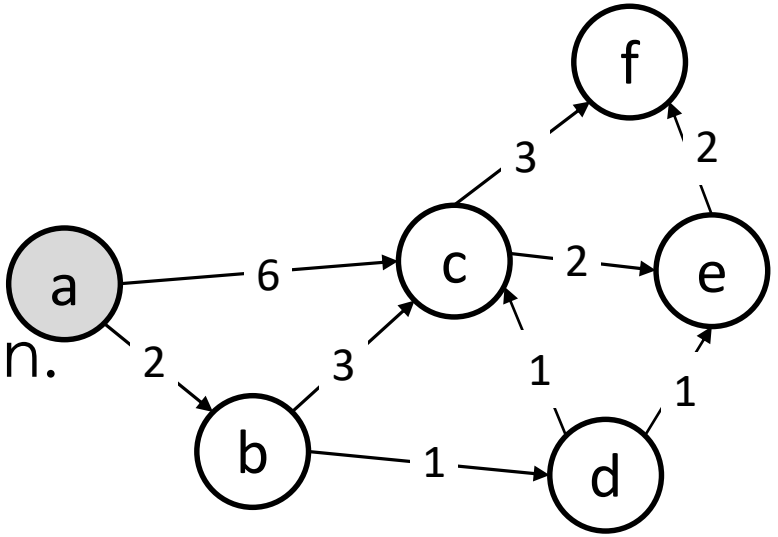
Naïve Evaluation on SSSP Algorithm

Naïve Evaluation is a direct and intuitive solution.





Problems on Naïve Evaluation



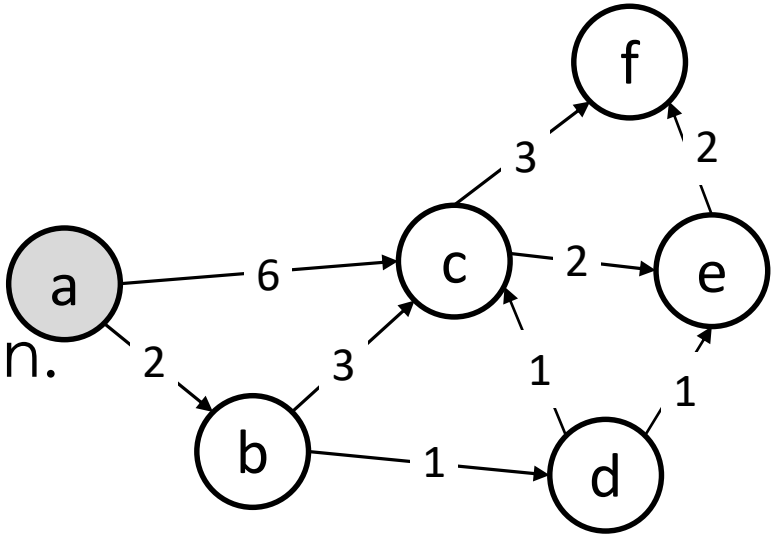
Naïve Evaluation is a direct and intuitive solution.

The converged nodes still need to involve.

The path from node a to node b has converged after the second iteration

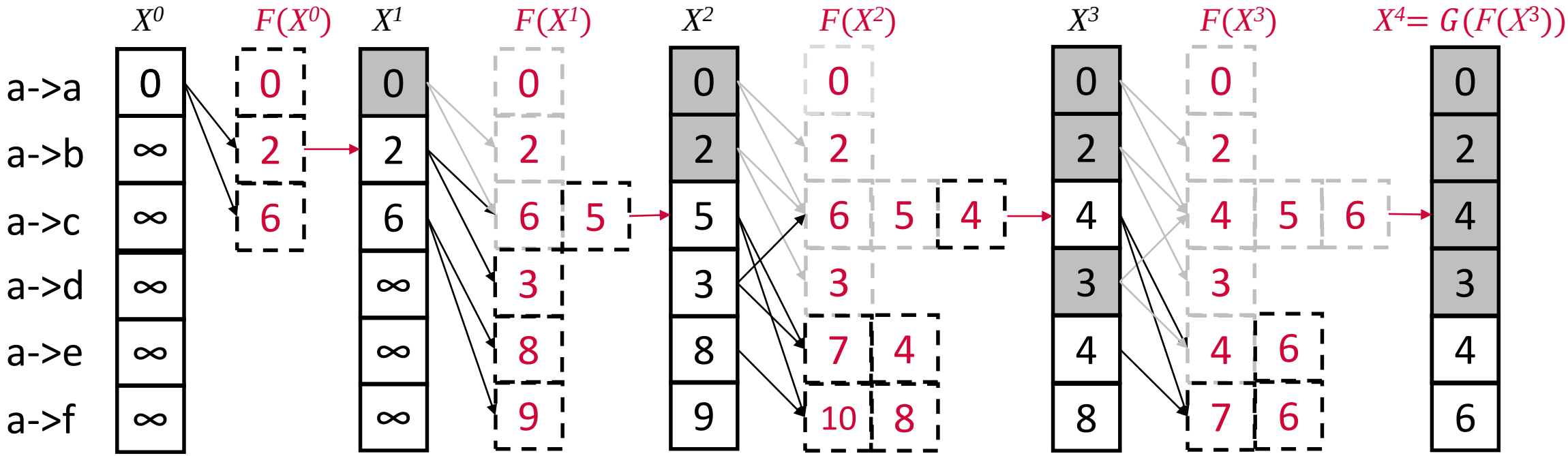
	X^0	$F(X^0)$	X^1	$F(X^1)$	X^2	$F(X^2)$		$X^4 = G(F(X^0))$
a->a	0	0	0	0	0	0		0
a->b	∞	2	2	2	2	2		2
a->c	∞	6	6	6	5	6	5 4	4
a->d	∞		∞	3	3	3		3
a->e	∞		∞	8	8	7	4	4
a->f	∞		∞	9	9	10	8	6

Problems on Naïve Evaluation



Naïve Evaluation is a direct and intuitive solution.

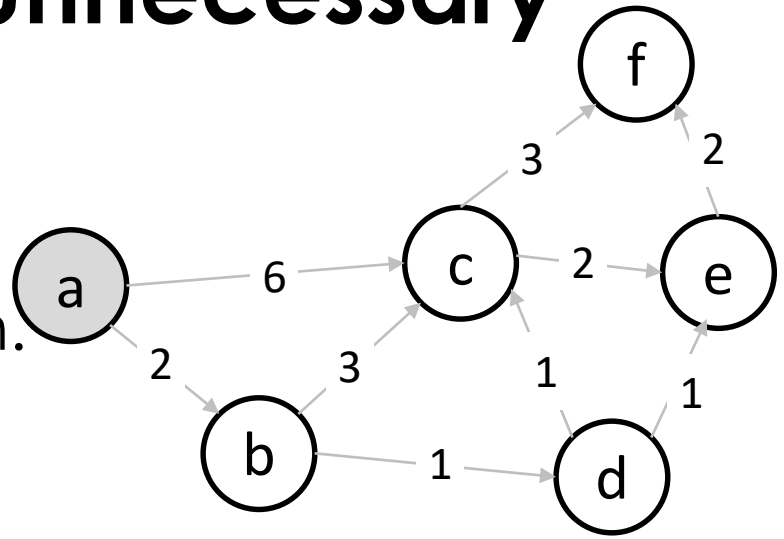
The converged nodes still need to involve.



Naïve evaluation brings unnecessary computation.

Semi-naïve Evaluation: Avoiding Unnecessary computation

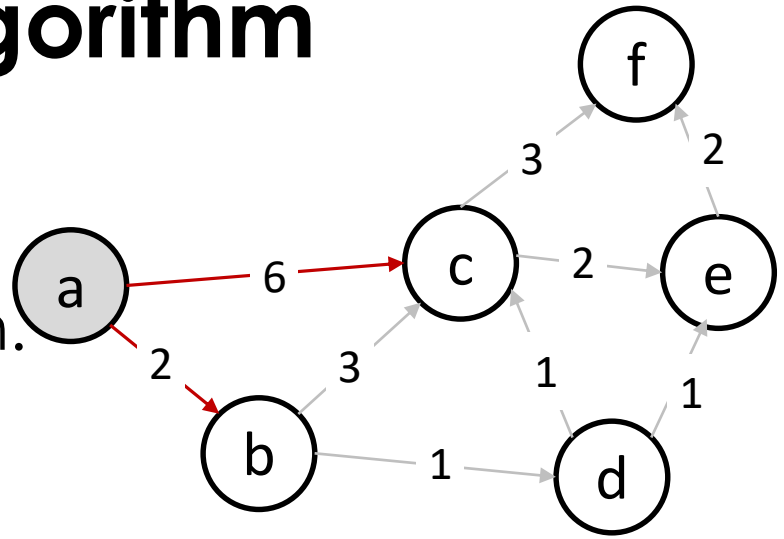
Semi-naïve Evaluation is an incremental approach.

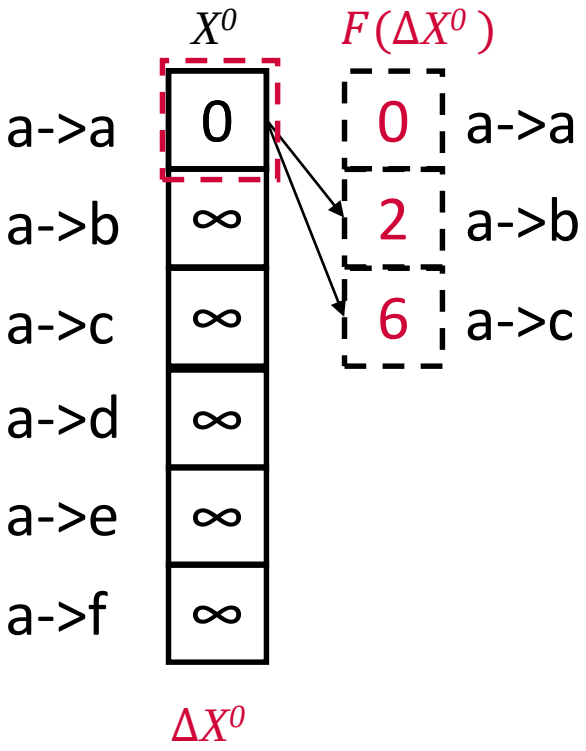


	X^0
a->a	0
a->b	∞
a->c	∞
a->d	∞
a->e	∞
a->f	∞

Semi-naïve Evaluation on SSSP Algorithm

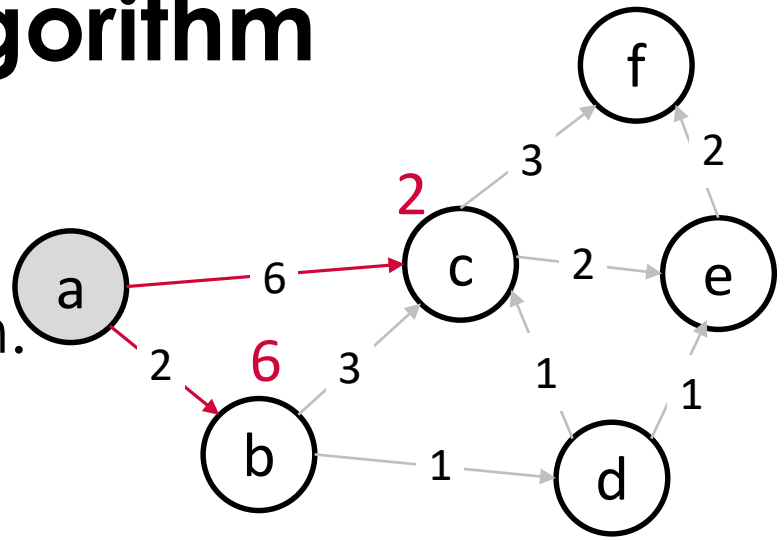
Semi-naïve Evaluation is an incremental approach.

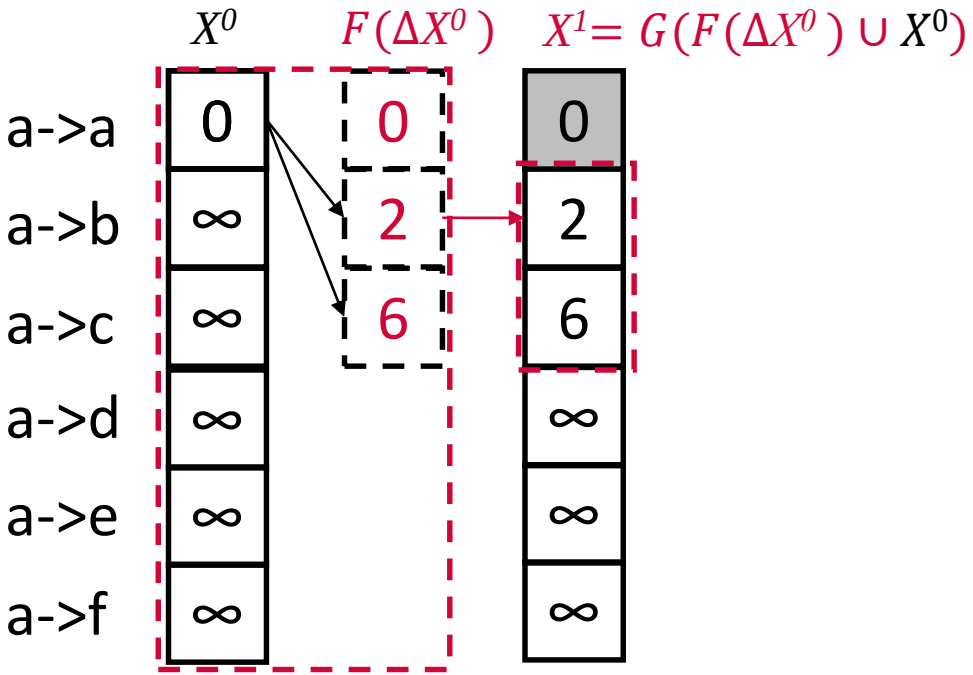




Semi-naïve Evaluation on SSSP Algorithm

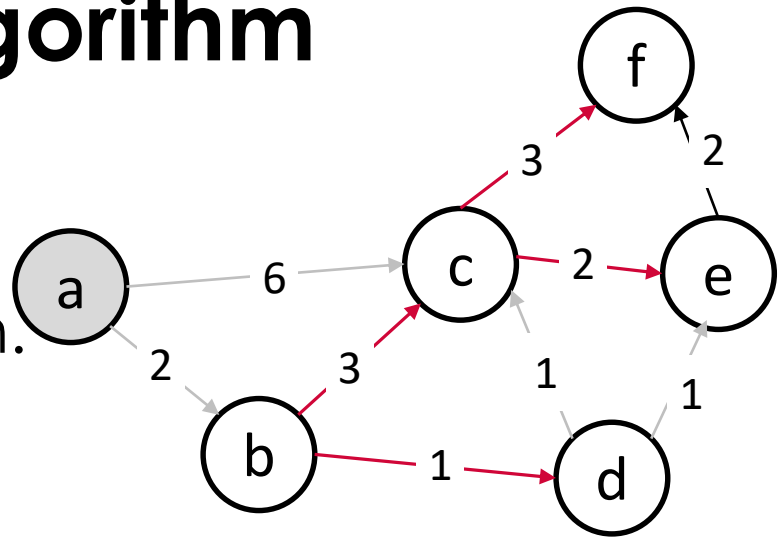
Semi-naïve Evaluation is an incremental approach.

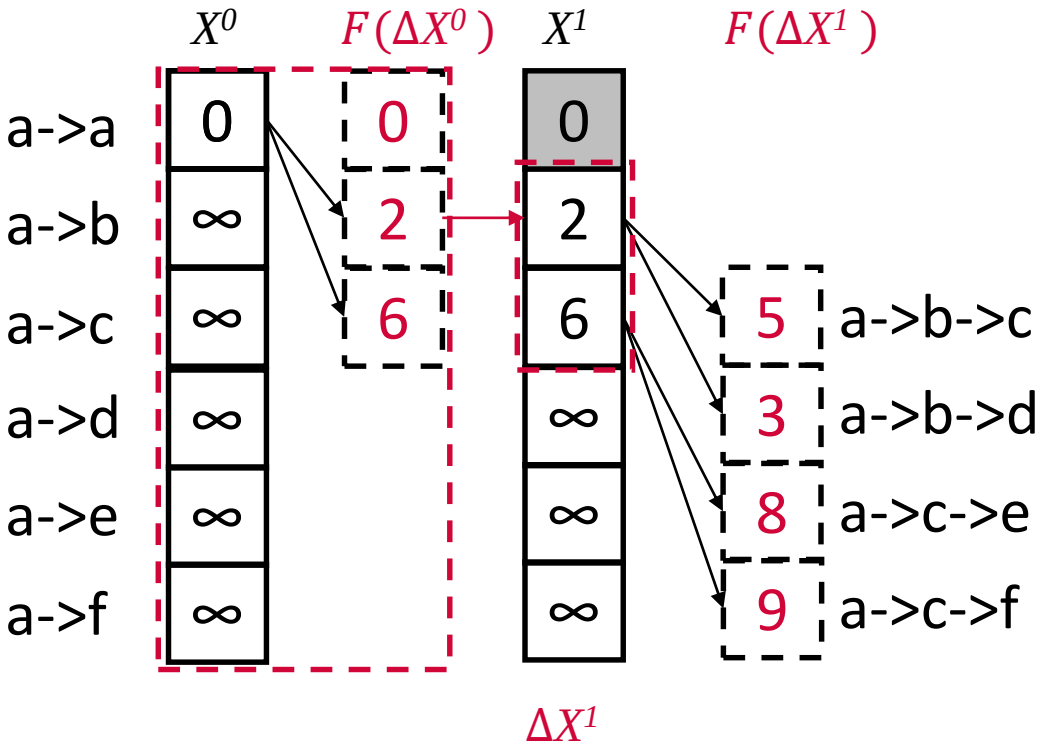




Semi-naïve Evaluation on SSSP Algorithm

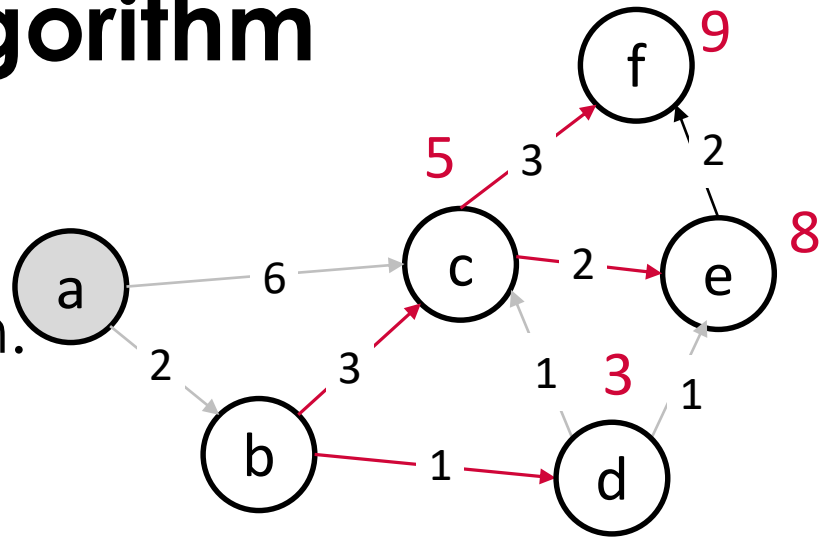
Semi-naïve Evaluation is an incremental approach.

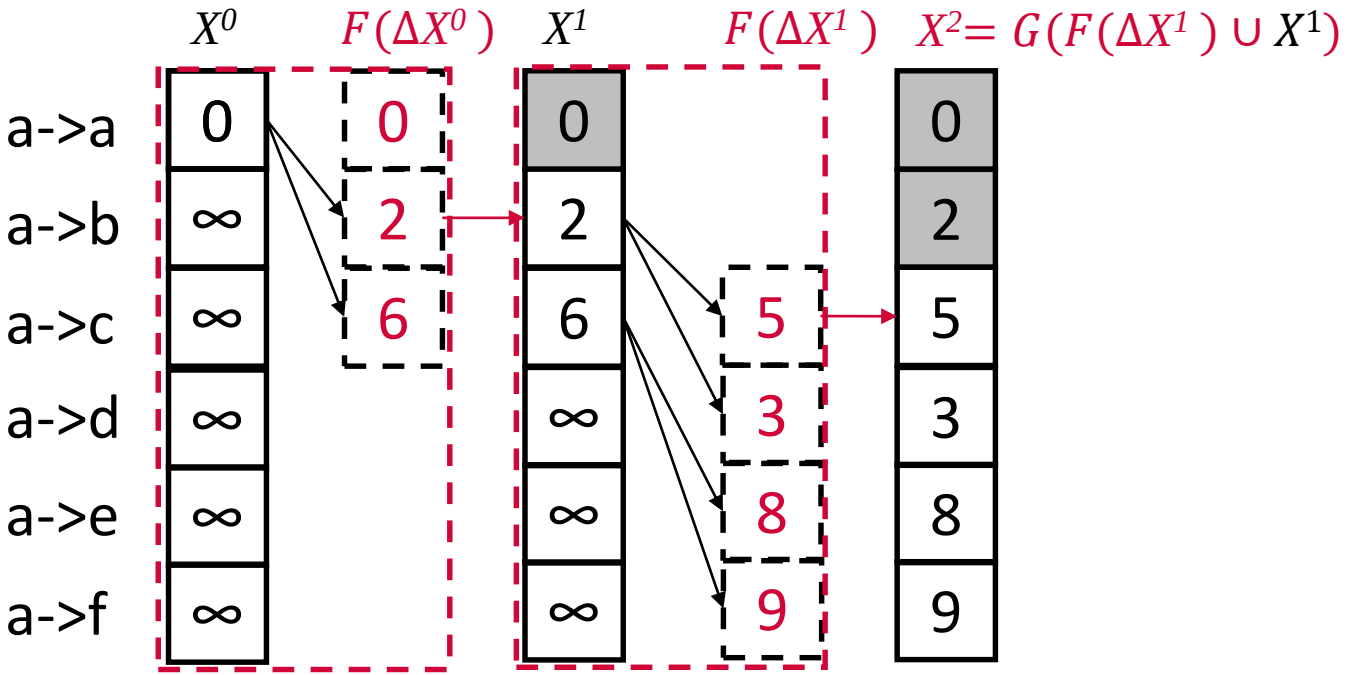




Semi-naïve Evaluation on SSSP Algorithm

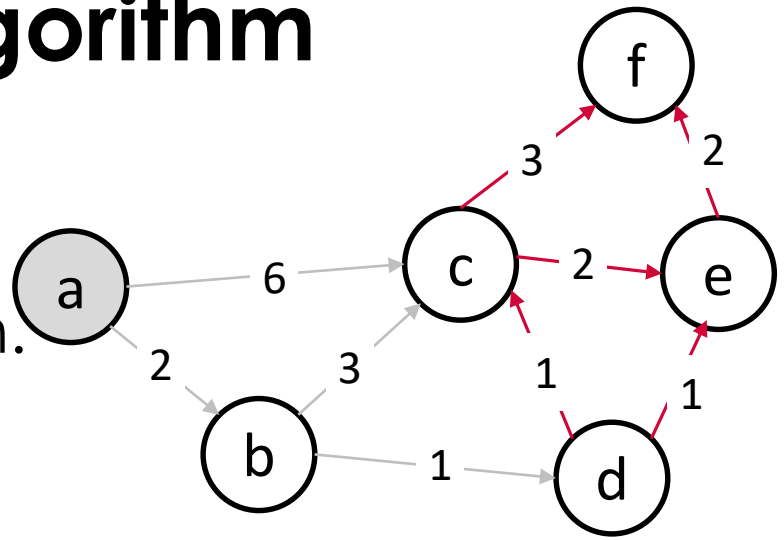
Semi-naïve Evaluation is an incremental approach.

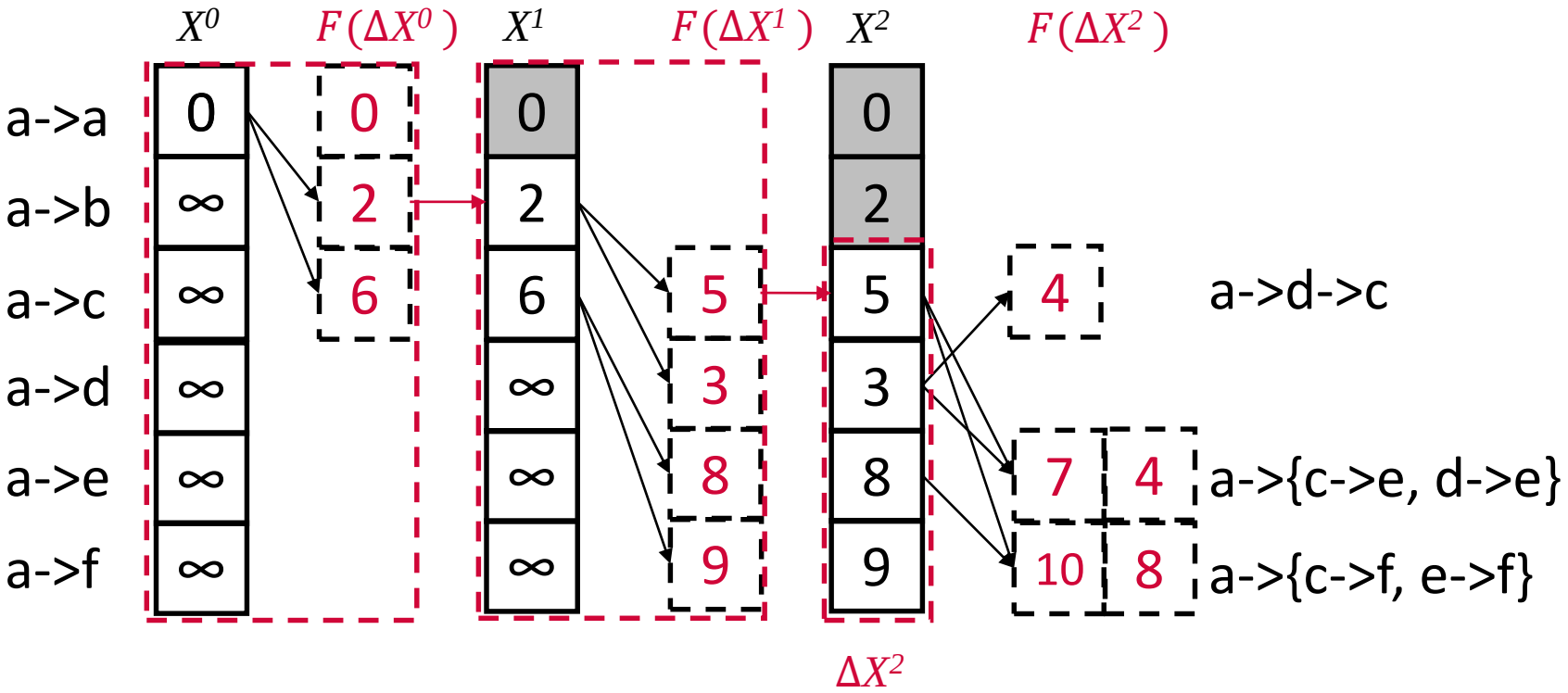




Semi-naïve Evaluation on SSSP Algorithm

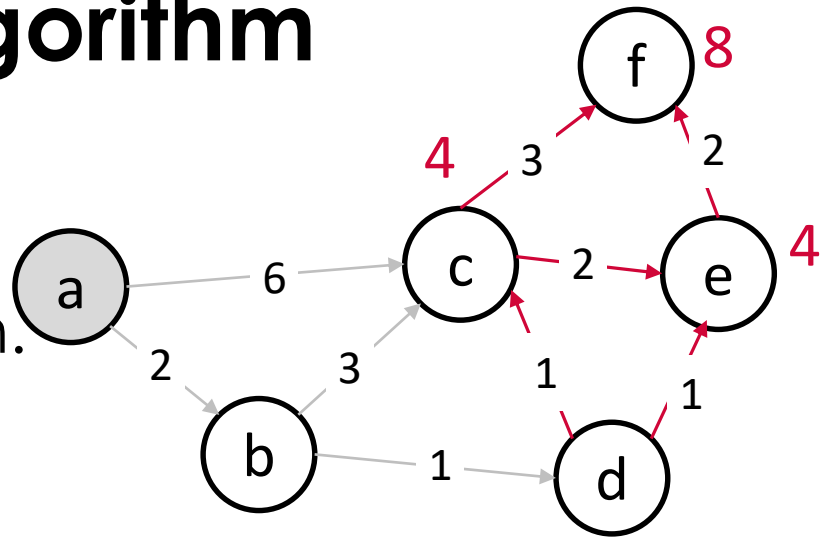
Semi-naïve Evaluation is an incremental approach.

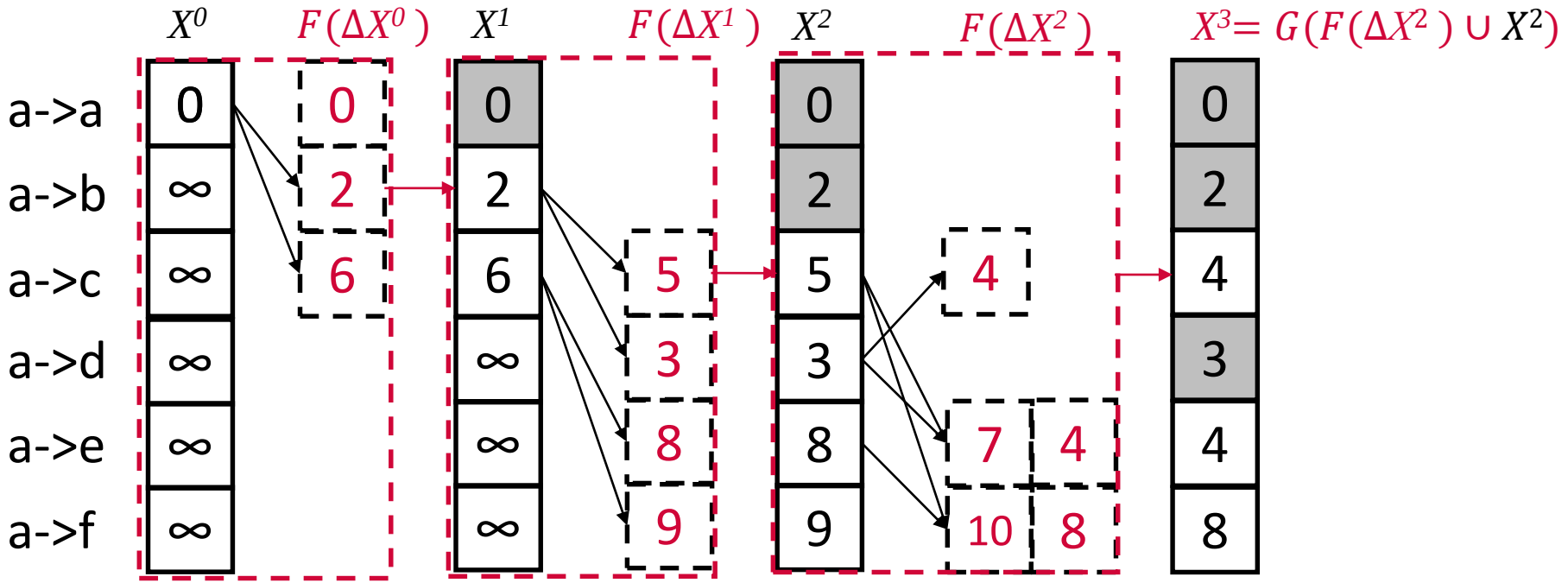




Semi-naïve Evaluation on SSSP Algorithm

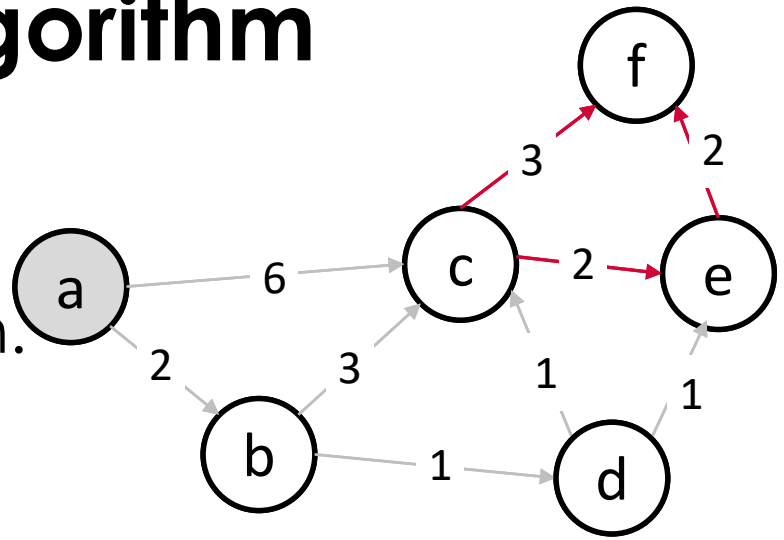
Semi-naïve Evaluation is an incremental approach.

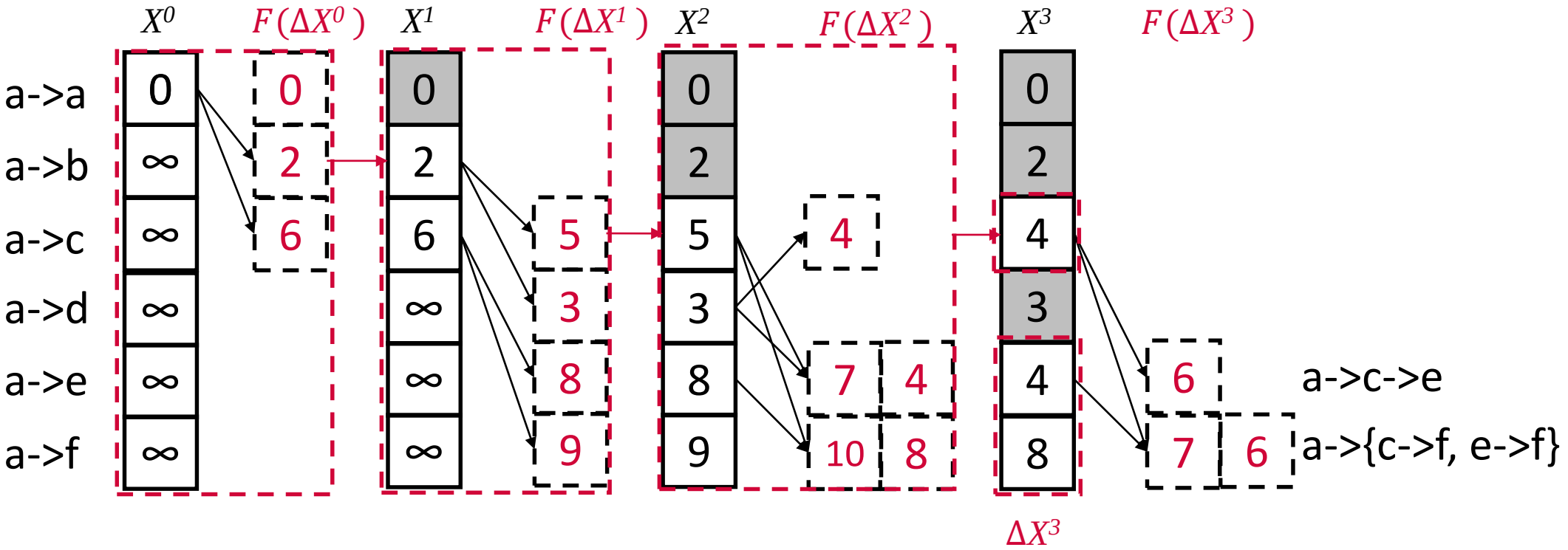




Semi-naïve Evaluation on SSSP Algorithm

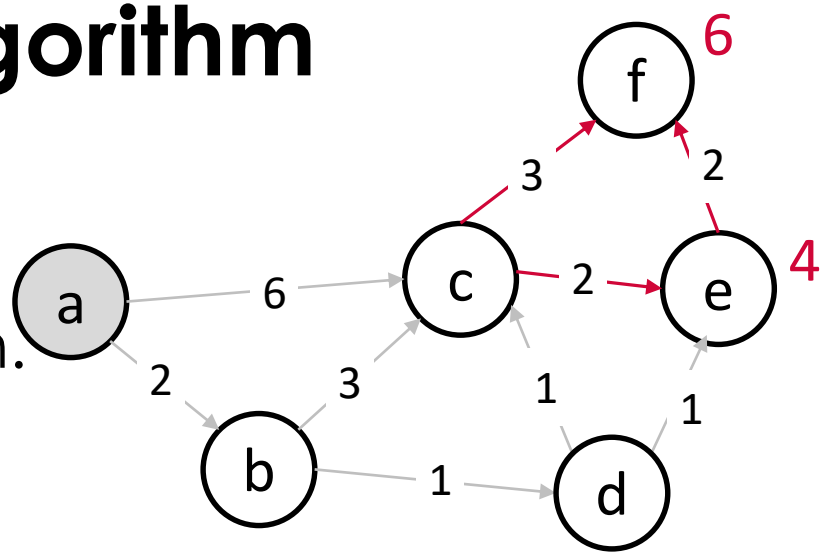
Semi-naïve Evaluation is an incremental approach.

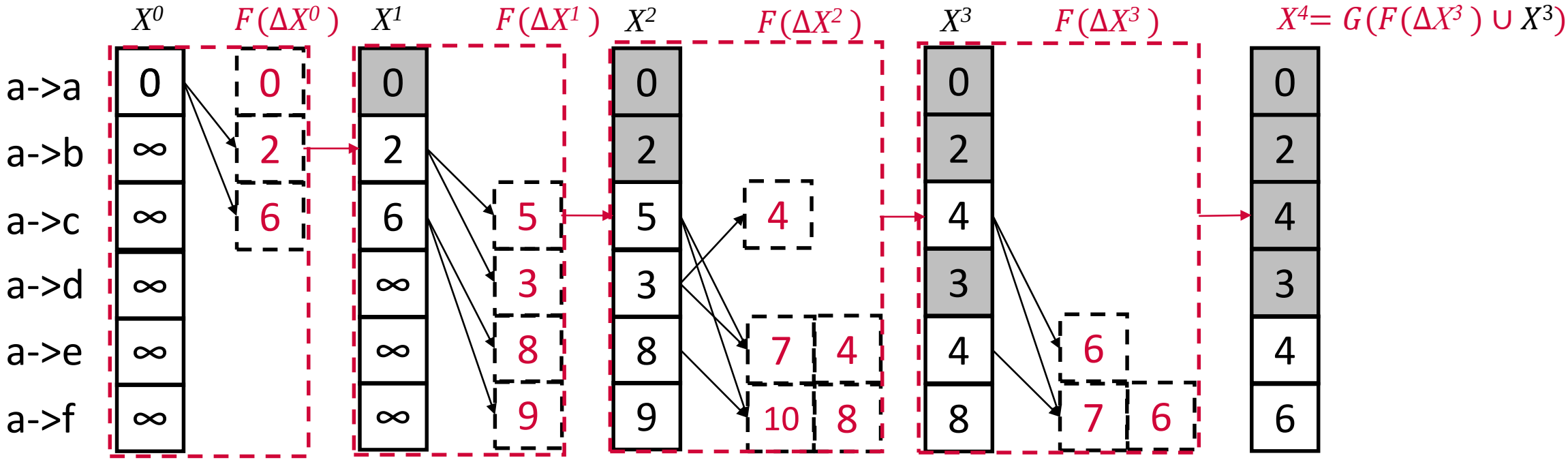




Semi-naïve Evaluation on SSSP Algorithm

Semi-naïve Evaluation is an incremental approach.

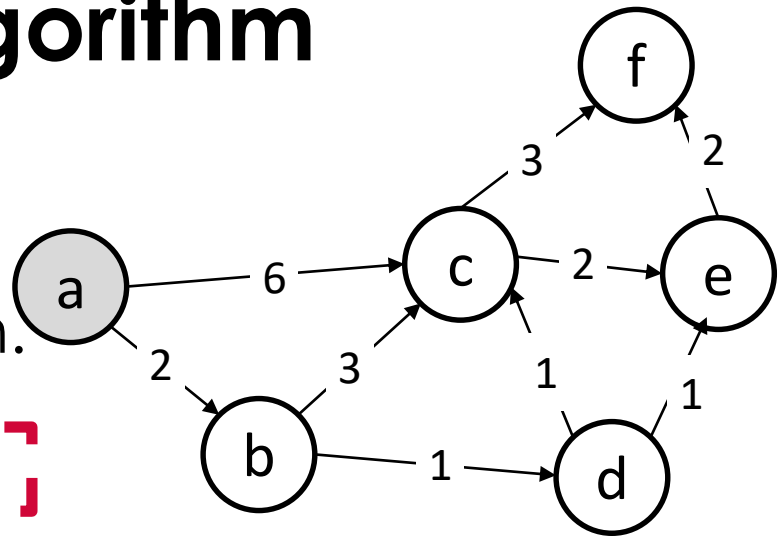


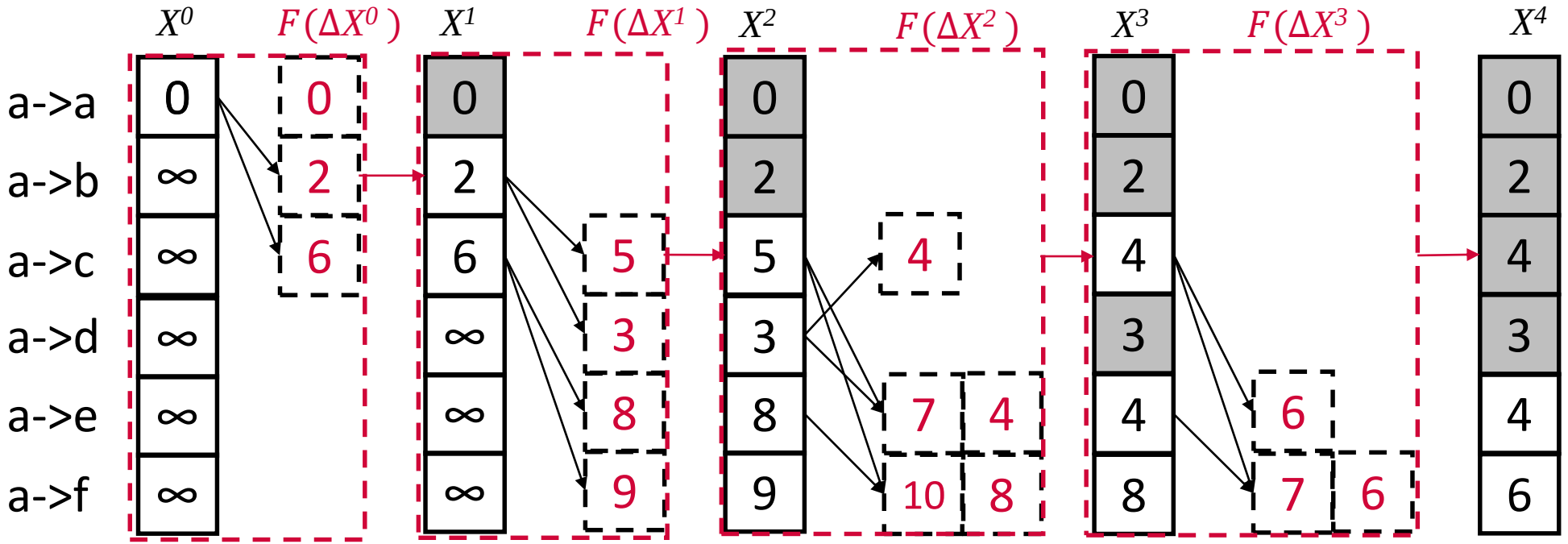


Semi-naïve Evaluation on SSSP Algorithm

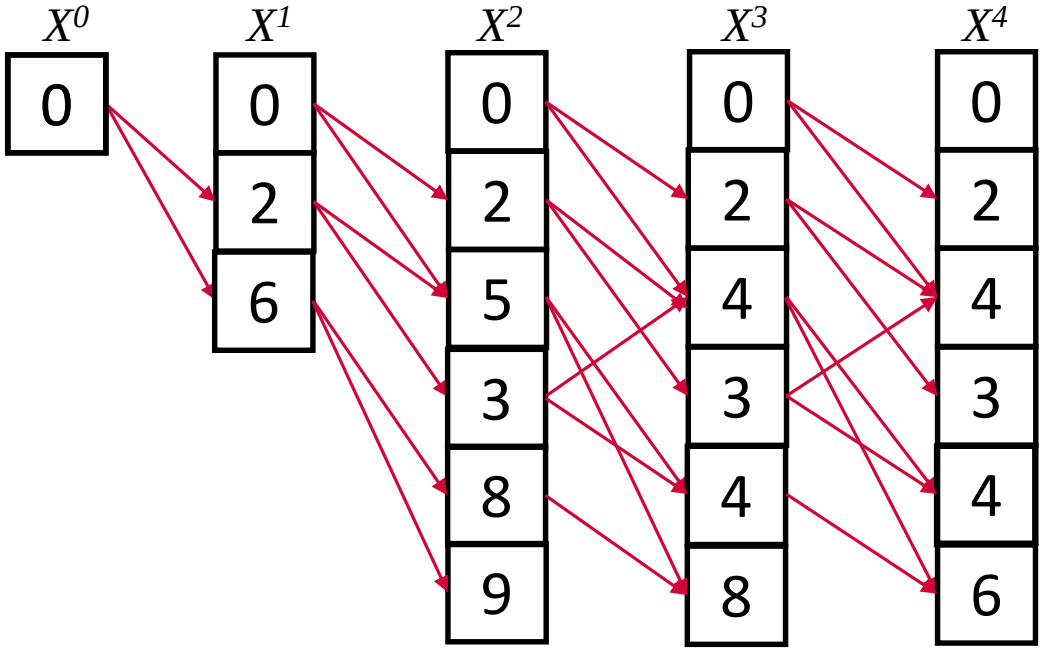
Semi-naïve Evaluation is an incremental approach.

The converged nodes do not involved in the computation.



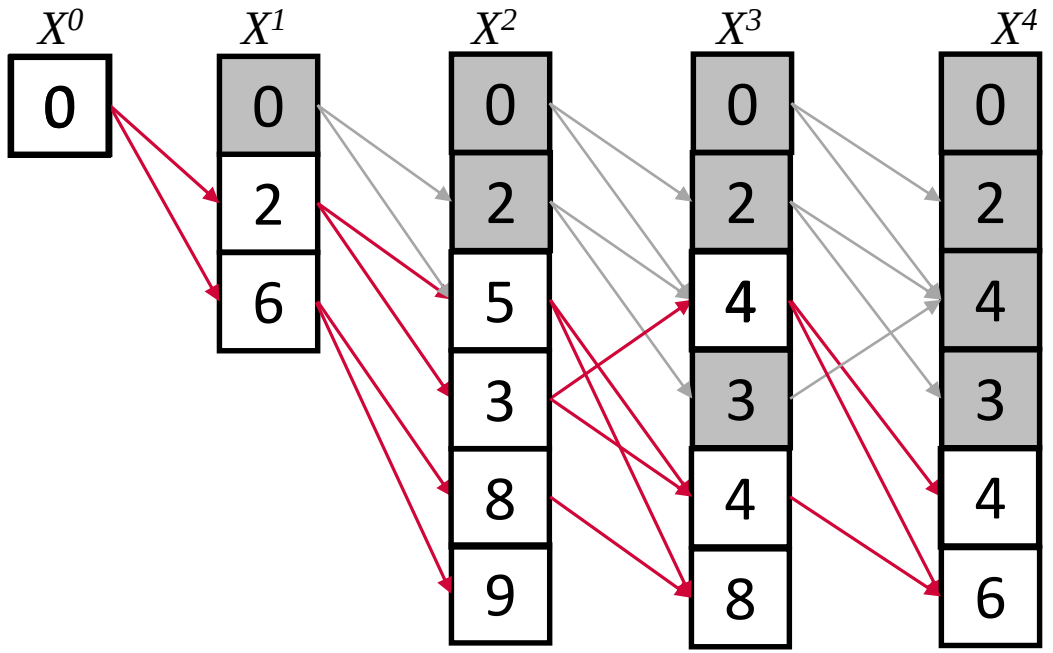


Naïve Evaluation vs. Semi-naïve Evaluation



A fully computation is performed.

The converged nodes still need to involve.

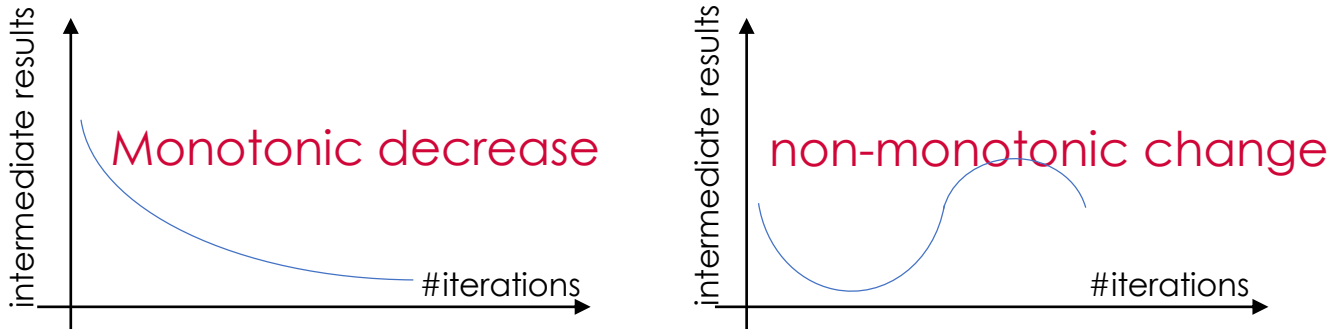


Only the necessary computation is performed.

The converged nodes do not need to involve.

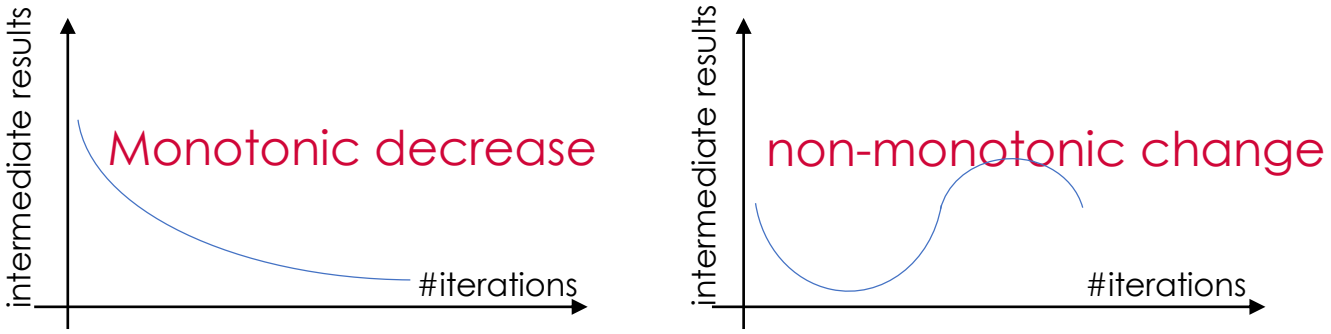
Monotonicity Requirement for Semi-naïve Evaluation

The monotonicity property



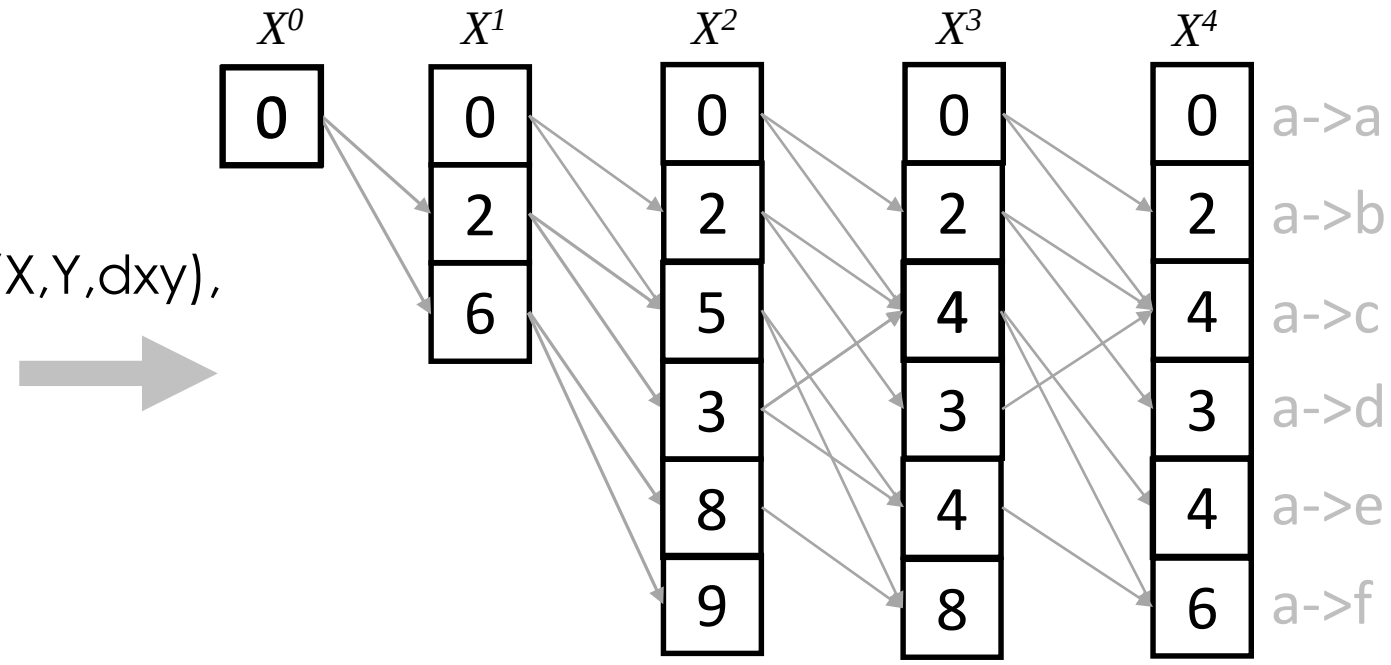
SSSP: A Monotonic Example

The monotonicity property



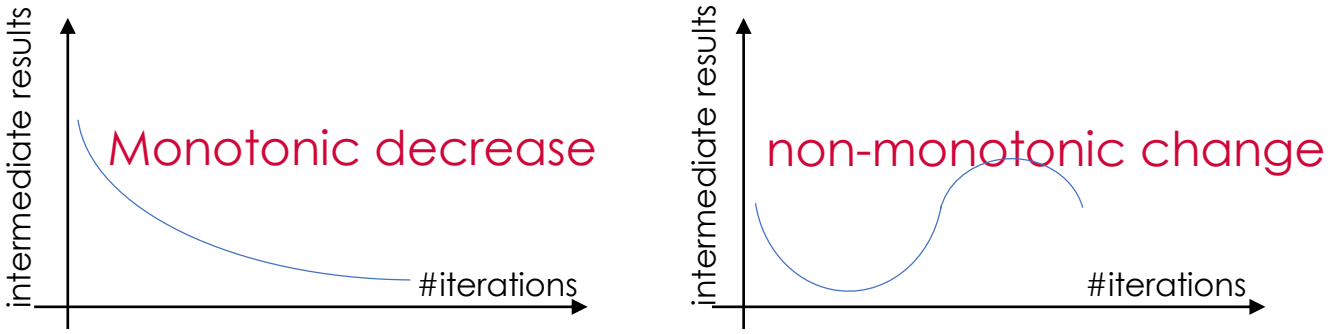
Single Source Shortest Path

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.



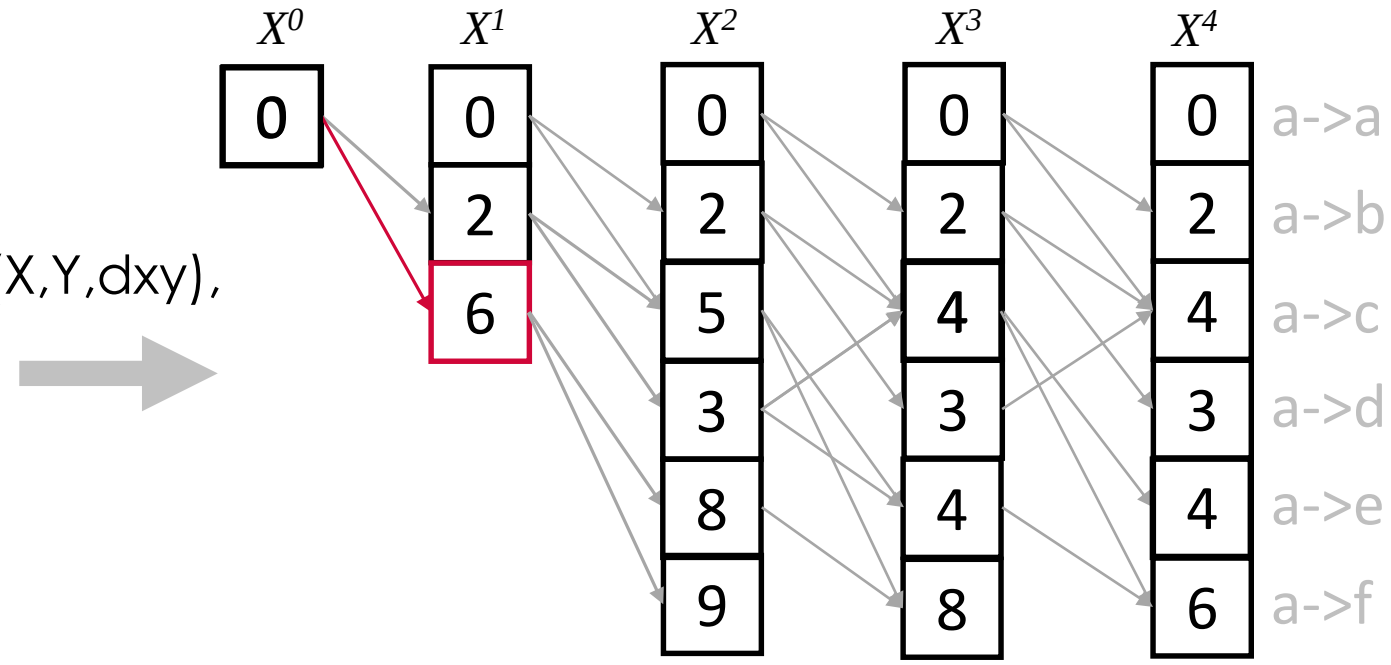
SSSP: A Monotonic Example

The monotonicity property



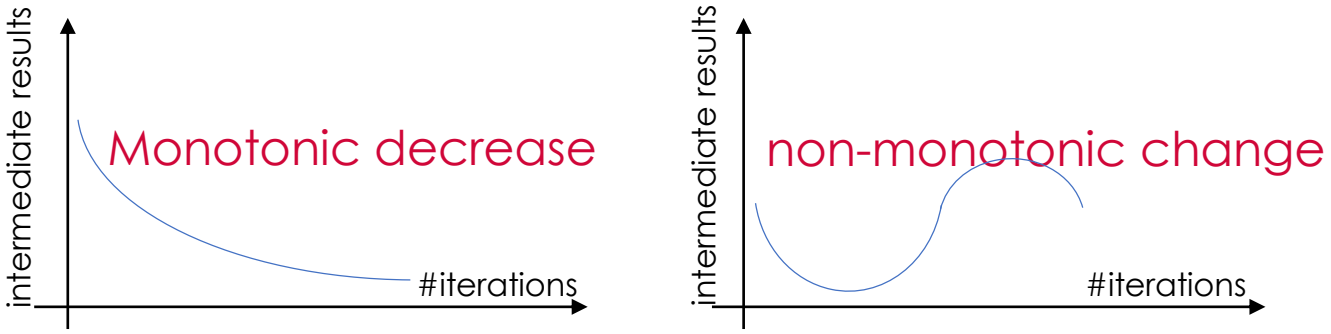
Single Source Shortest Path

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.



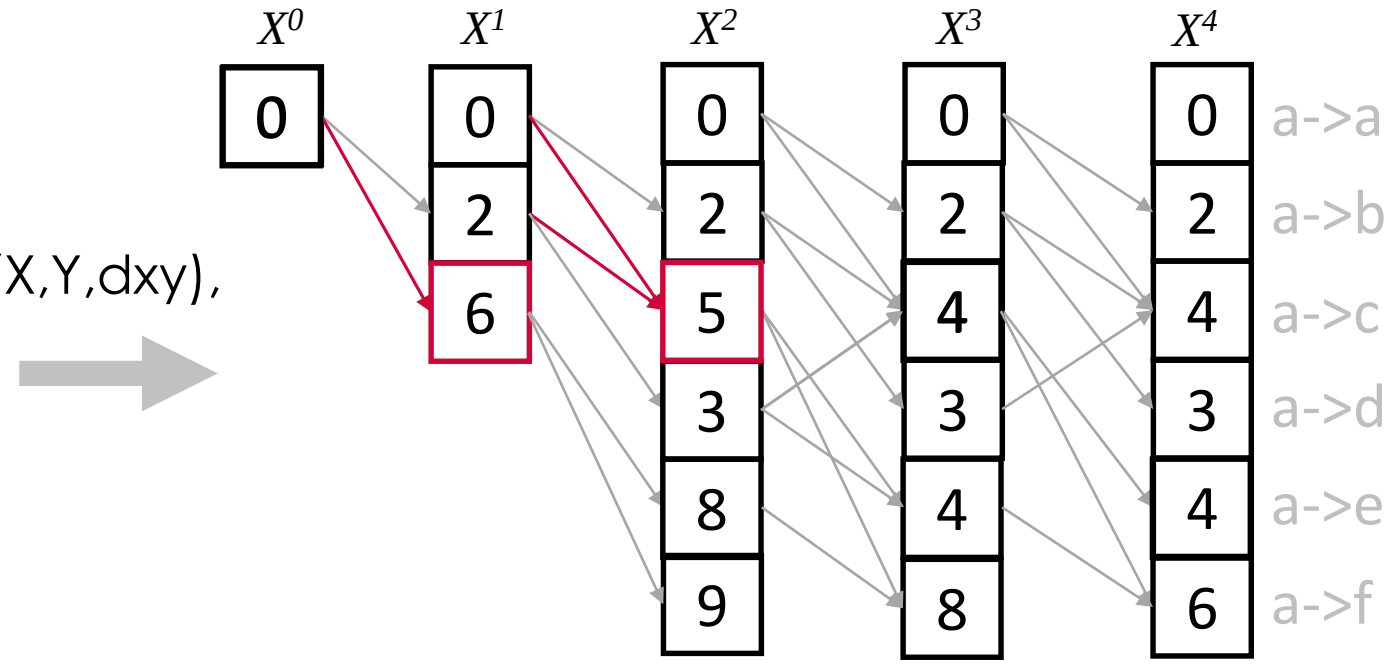
SSSP: A Monotonic Example

The monotonicity property



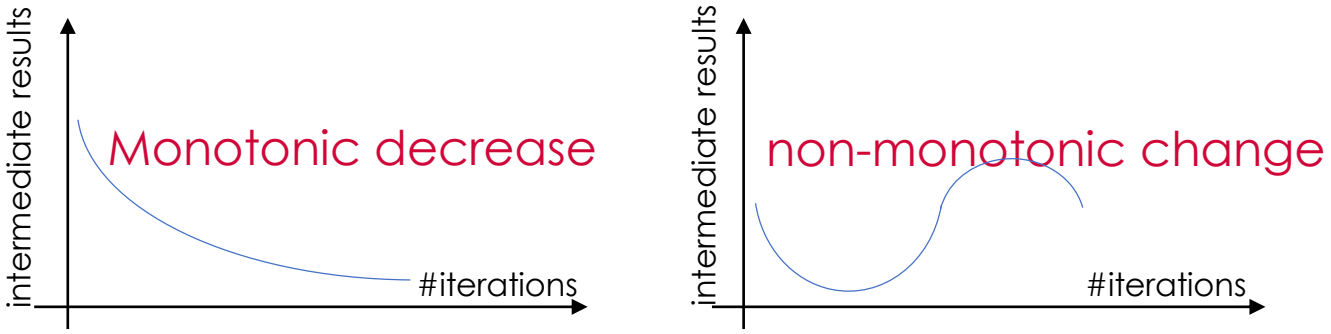
Single Source Shortest Path

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.



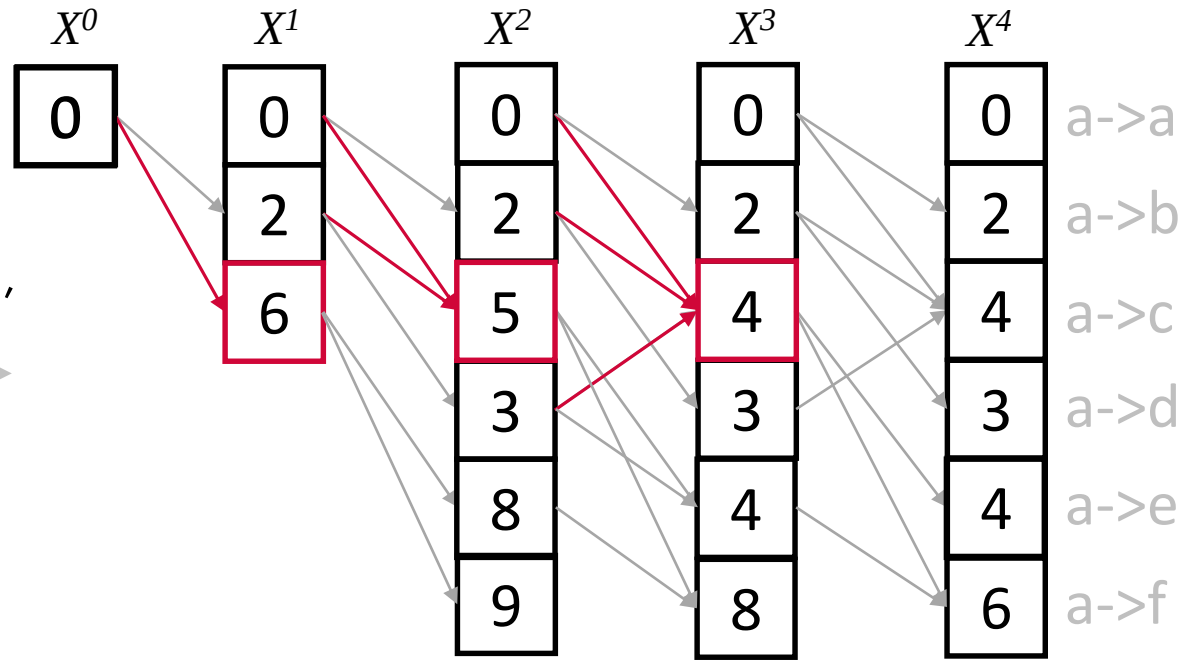
SSSP: A Monotonic Example

The monotonicity property



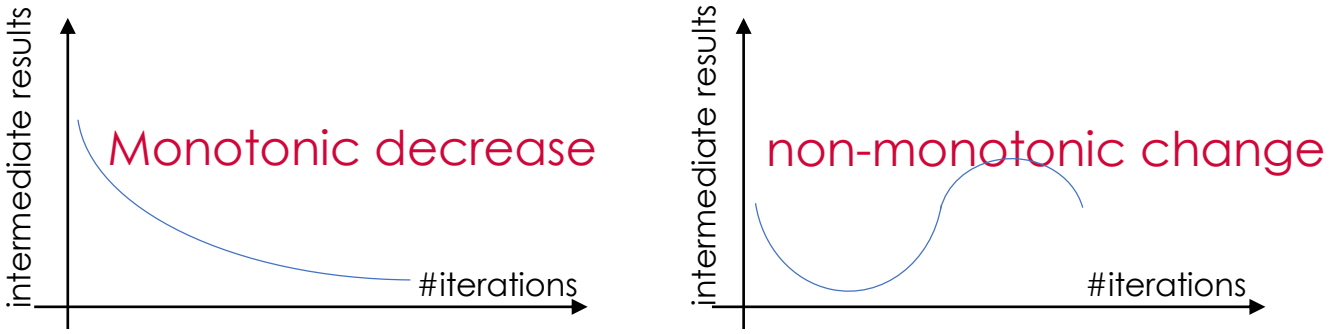
Single Source Shortest Path

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.



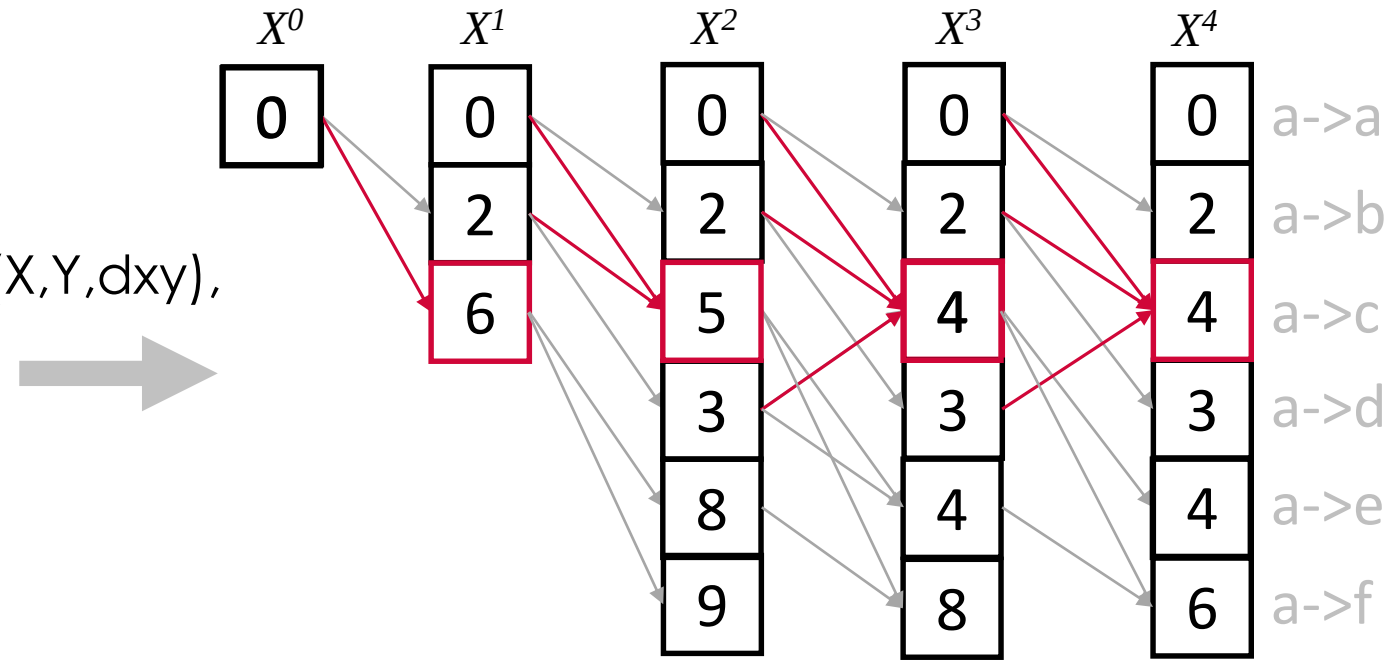
SSSP: A Monotonic Example

The monotonicity property



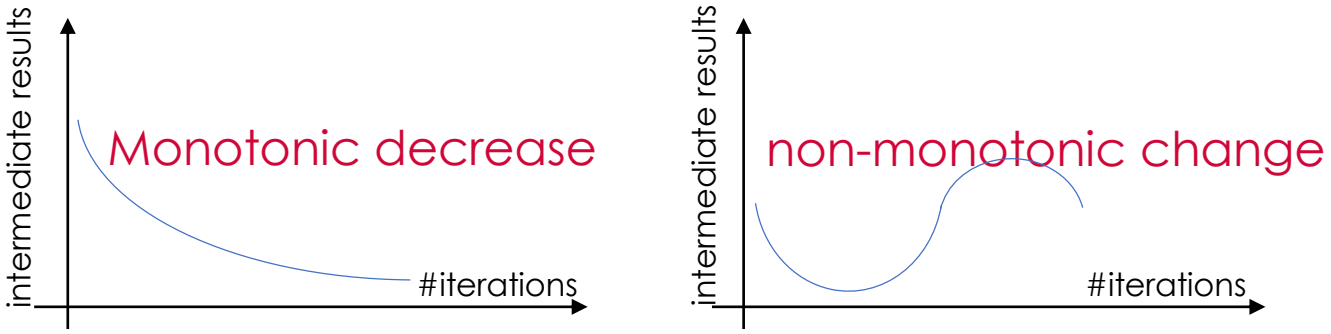
Single Source Shortest Path

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.



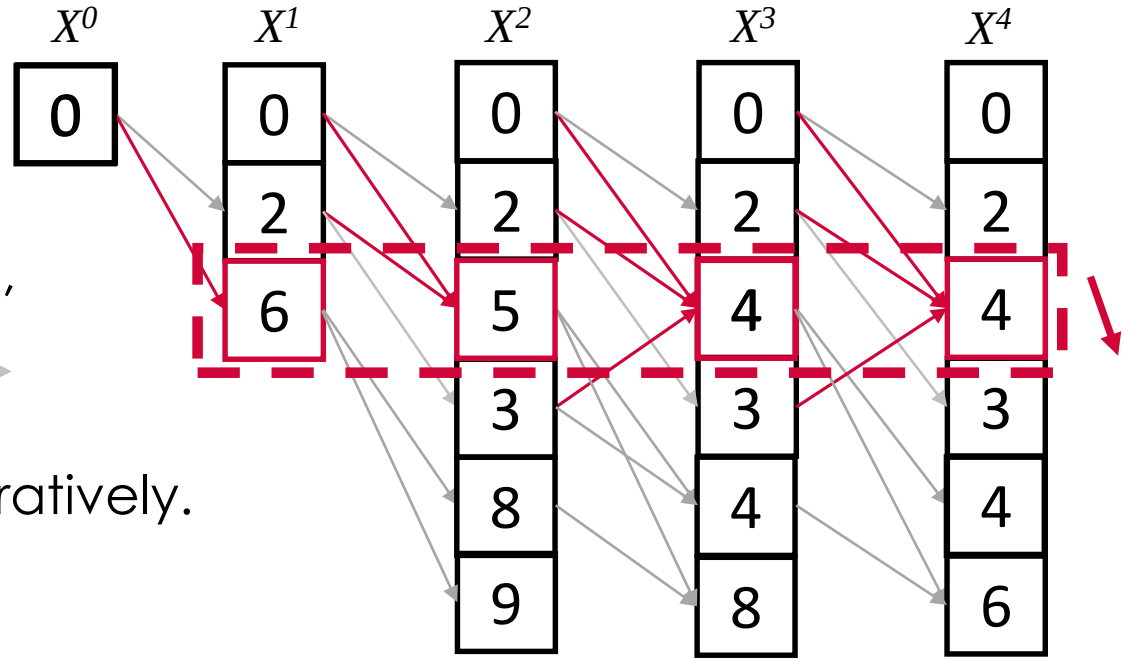
SSSP: A Monotonic Example

The monotonicity property



Single Source Shortest Path

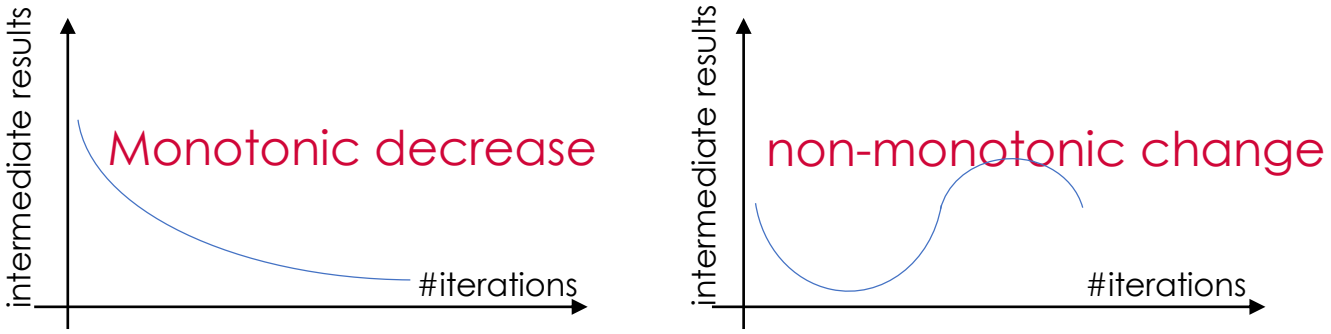
sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.



Distance a->c monotonically decreases iteratively.

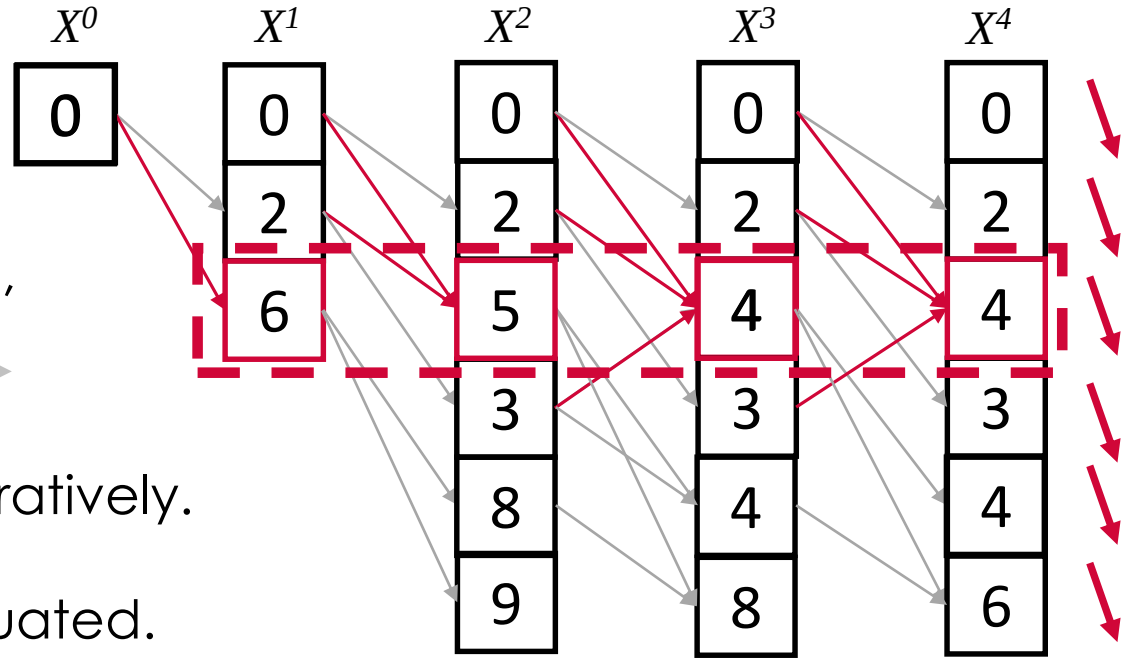
SSSP: A Monotonic Example

The monotonicity property



Single Source Shortest Path

sssp(Y, min[dy]) :- sssp(X,dx), edge(X,Y,dxy),
dy=dx+dxy.

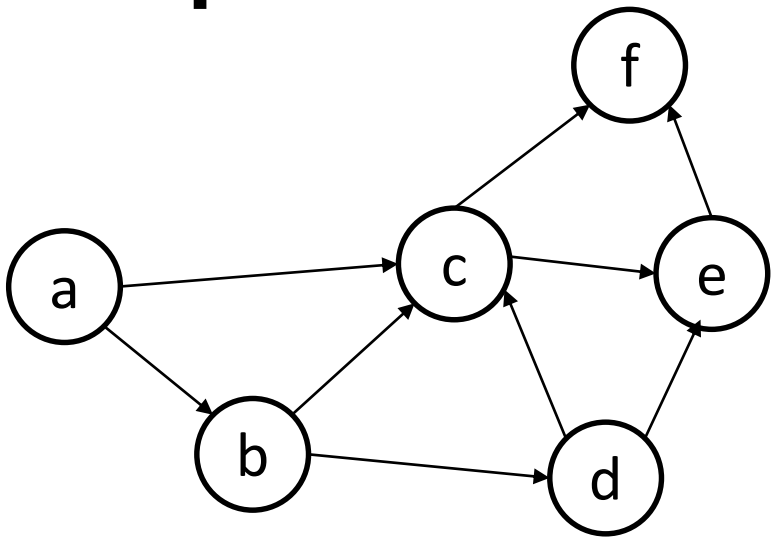


Distance a->c monotonically decreases iteratively.

The SSSP algorithm can be semi-naïve evaluated.

PageRank: A Non-monotonic Example

The monotonicity property



PageRank

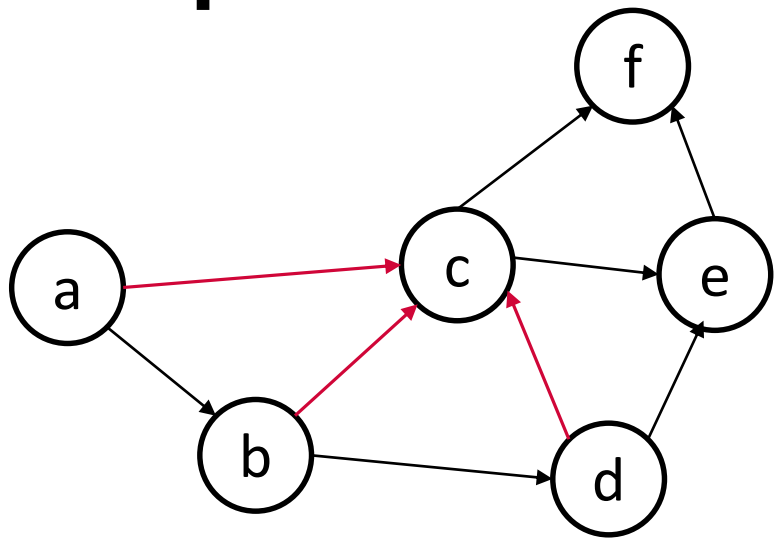
```
rank(i+1, Y, sum[ry])  :-  node(Y), ry=0.2;
                       :-  rank(i, X, rx), edge(X,Y),
                       :-  degree(X, d), ry=0.8 rx/d.
```

X^0

1
1
1
1
1
1

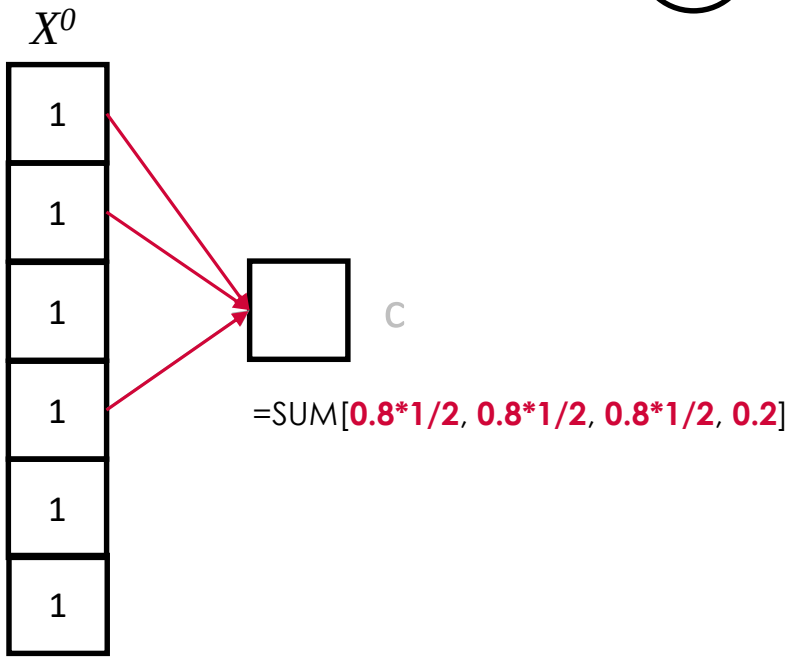
PageRank: A Non-monotonic Example

The monotonicity property



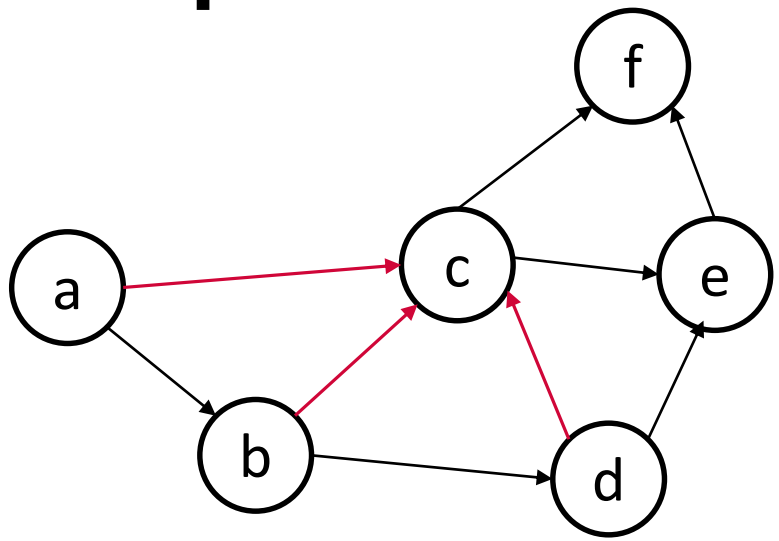
PageRank

```
rank(i+1, Y, sum[ry])  :-  node(Y), ry=0.2;
                       :-  rank(i, X, rx), edge(X,Y),
                       :-  degree(X, d), ry=0.8 rx/d.
```



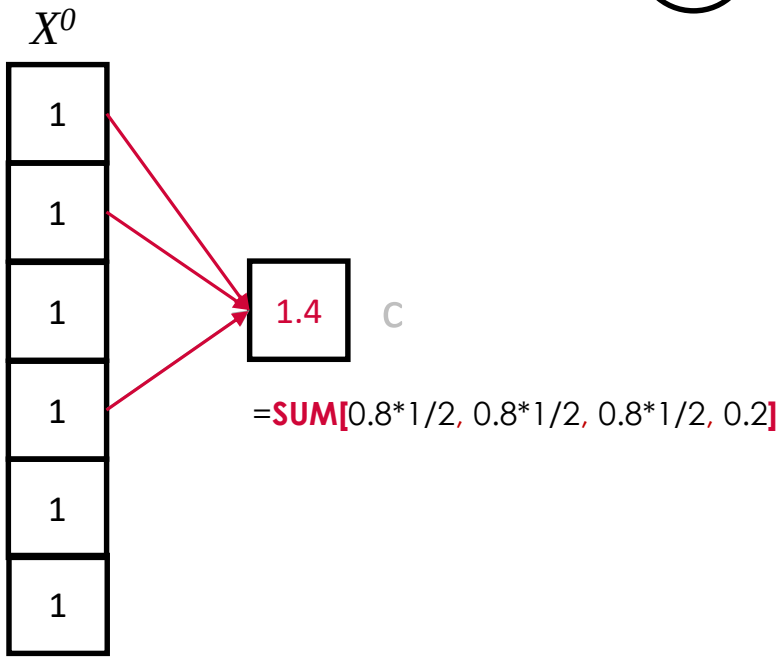
PageRank: A Non-monotonic Example

The monotonicity property



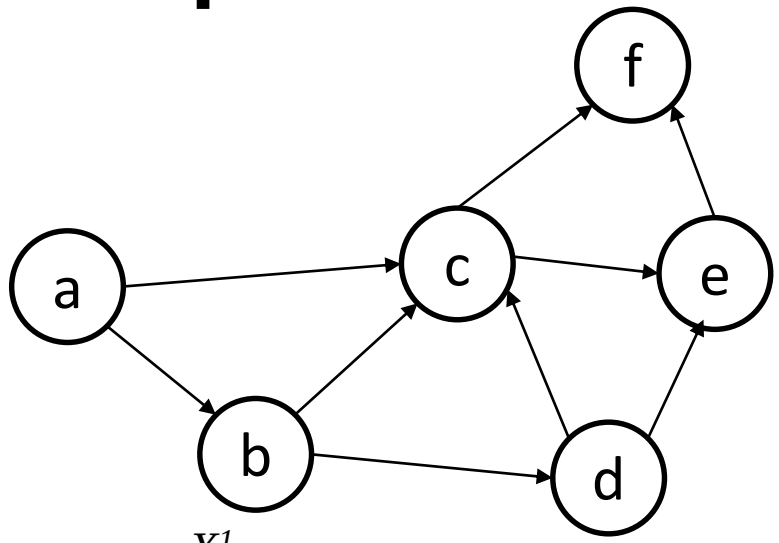
PageRank

rank(i+1, Y, **sum**[ry]) :- node(Y), ry=0.2;
:- rank(i, X, rx), edge(X,Y),
:- degree(X, d), ry=0.8 rx/d.



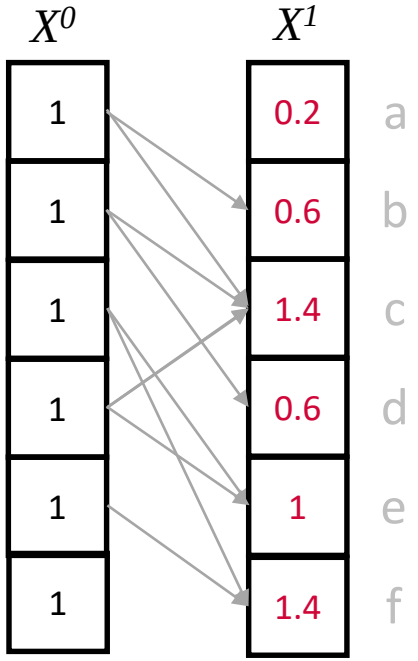
PageRank: A Non-monotonic Example

The monotonicity property



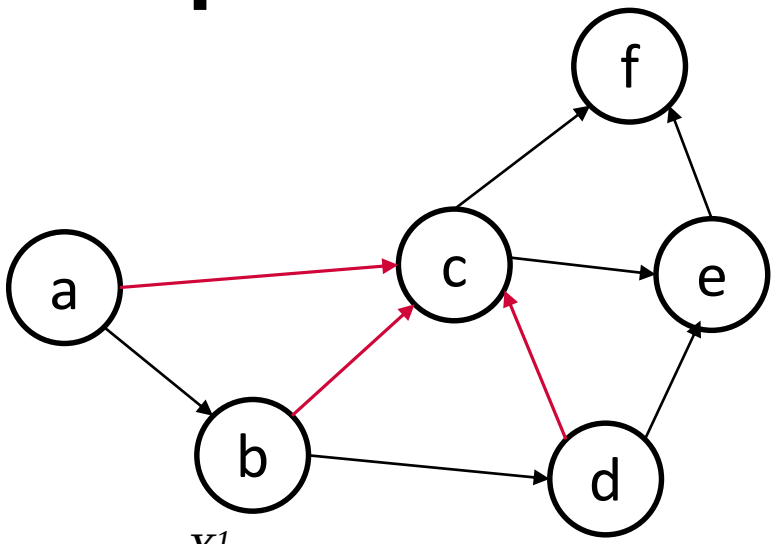
PageRank

```
rank(i+1, Y, sum[ry])  :-  node(Y), ry=0.2;
                       :-  rank(i, X, rx), edge(X,Y),
                       :-  degree(X, d), ry=0.8 rx/d.
```



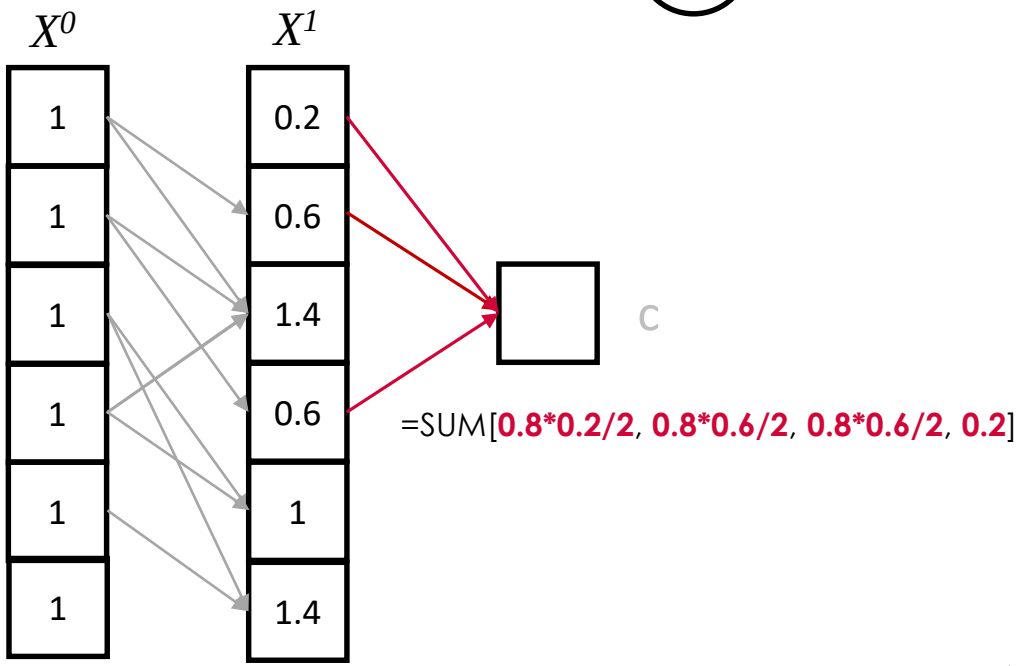
PageRank: A Non-monotonic Example

The monotonicity property



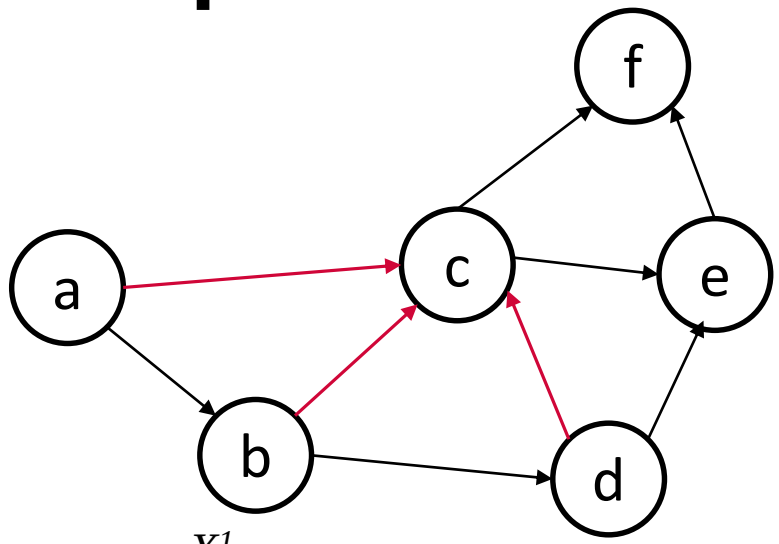
PageRank

rank(i+1, Y, sum[ry]) :- node(Y), ry=**0.2**;
:- rank(i, X, rx), edge(X,Y),
:- degree(X, d), ry=**0.8 rx/d**.



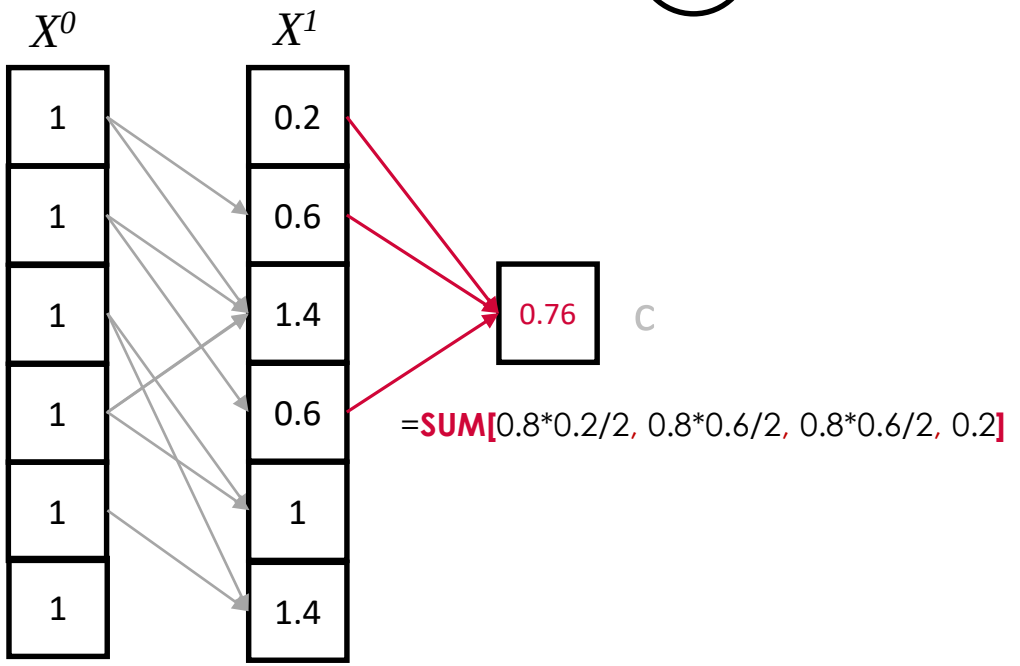
PageRank: A Non-monotonic Example

The monotonicity property



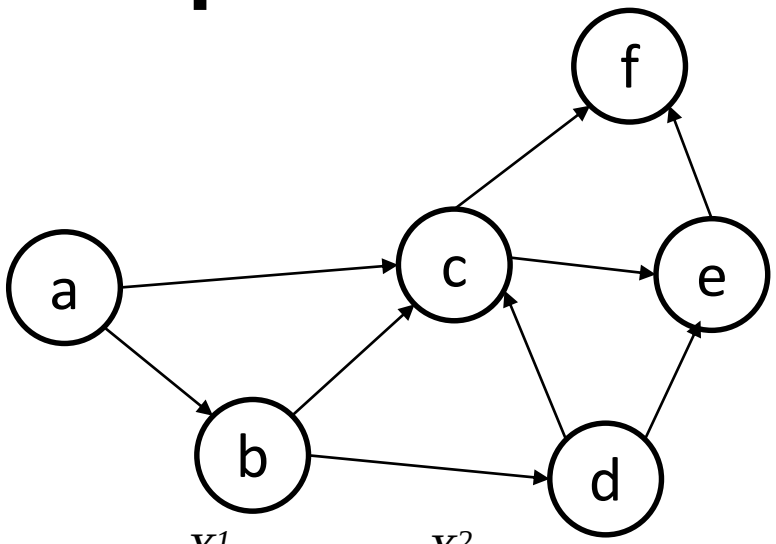
PageRank

rank(i+1, Y, **sum**[ry]) :- node(Y), ry=0.2;
:- rank(i, X, rx), edge(X,Y),
:- degree(X, d), ry=0.8 rx/d.



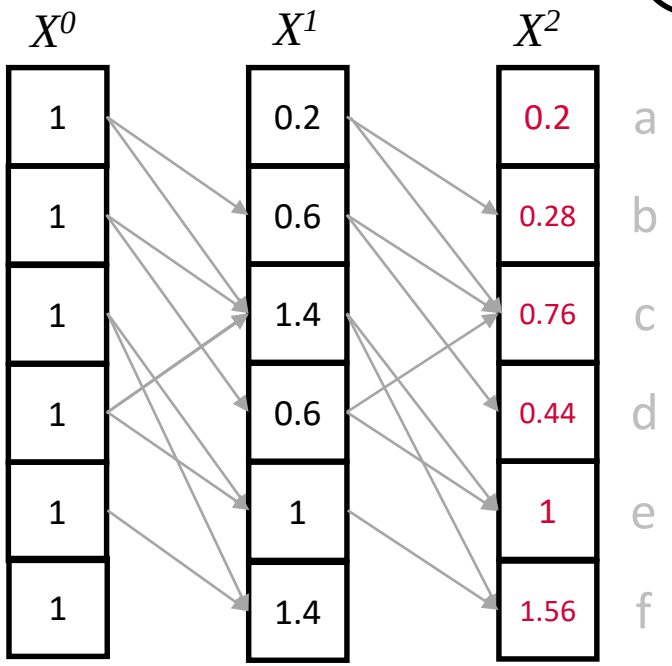
PageRank: A Non-monotonic Example

The monotonicity property



PageRank

```
rank(i+1, Y, sum[ry])  :-  node(Y), ry=0.2;
                       :-  rank(i, X, rx), edge(X,Y),
                       :-  degree(X, d), ry=0.8 rx/d.
```

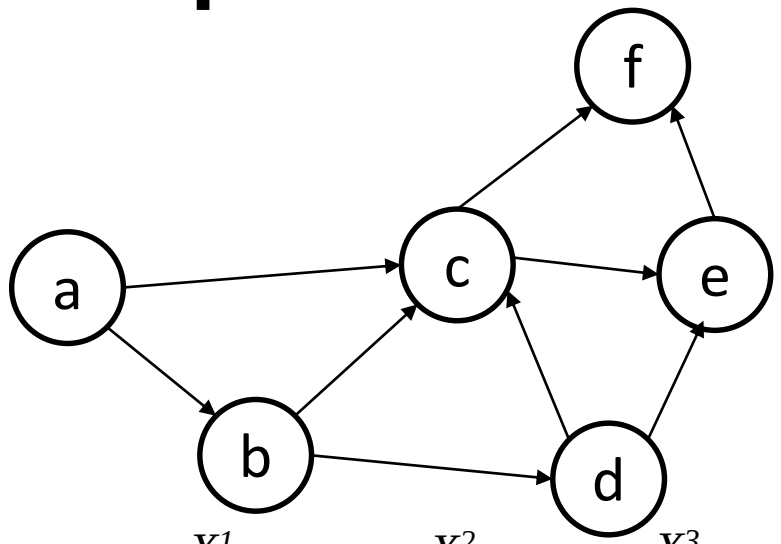


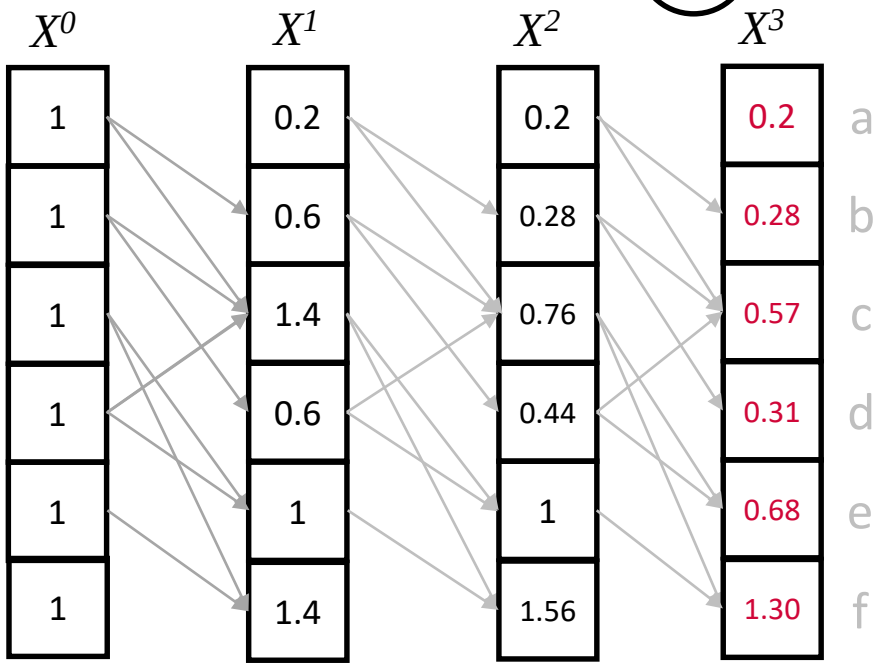
PageRank: A Non-monotonic Example

The monotonicity property

PageRank

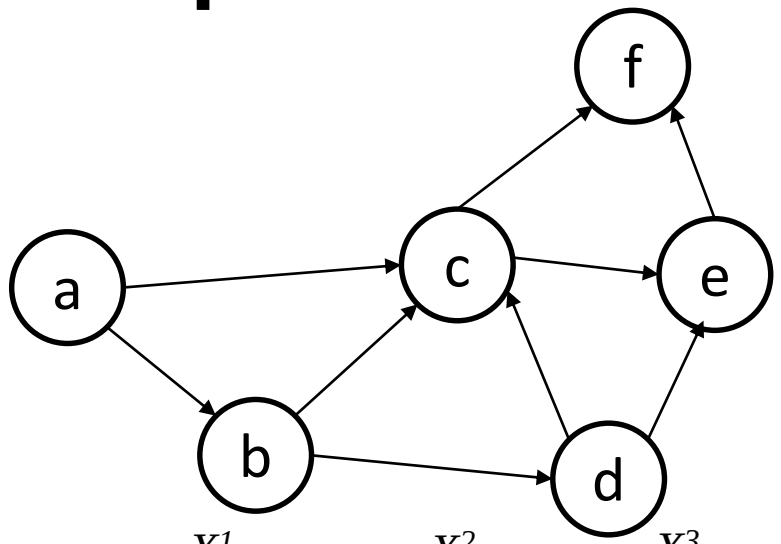
```
rank(i+1, Y, sum[ry])  :-  node(Y), ry=0.2;
                       :-  rank(i, X, rx), edge(X,Y),
                       :-  degree(X, d), ry=0.8 rx/d.
```





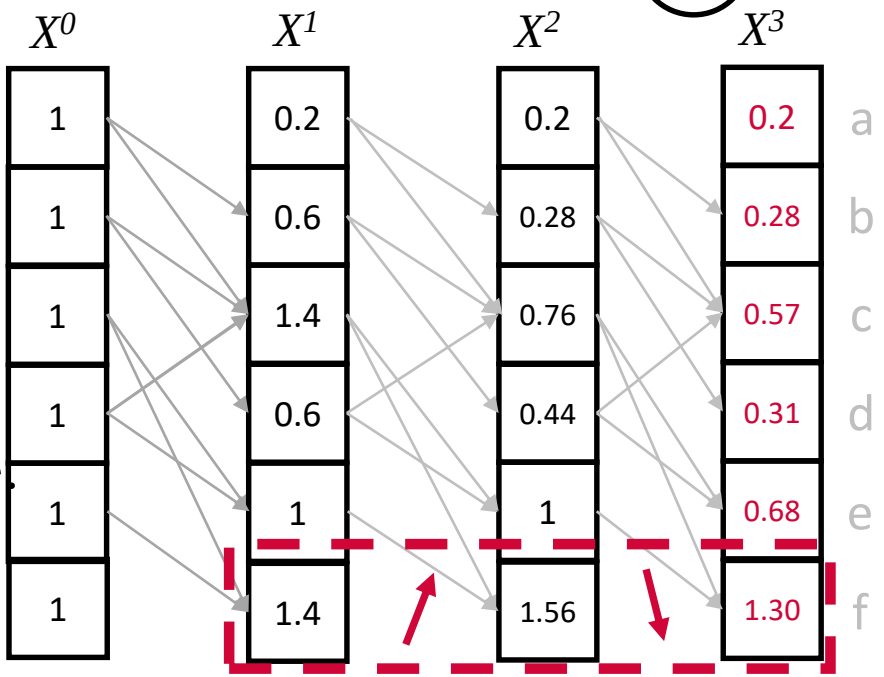
PageRank: A Non-monotonic Example

The monotonicity property



PageRank

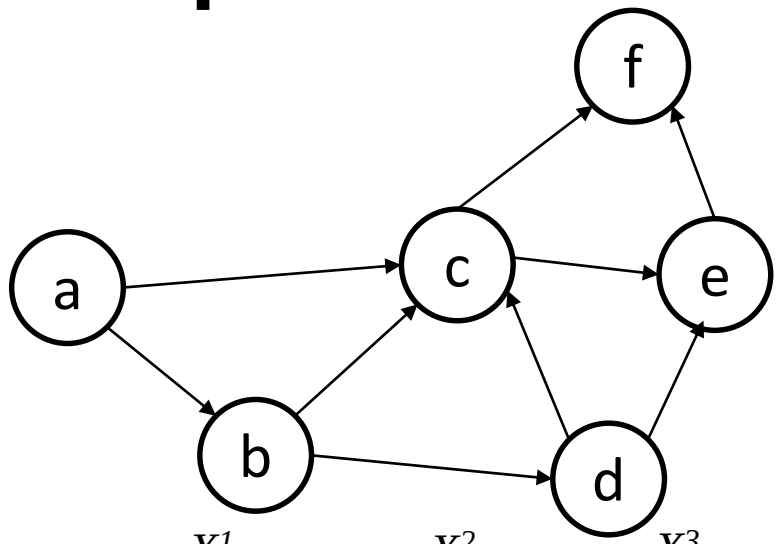
rank(i+1, Y, sum[ry]) :- node(Y), ry=0.2;
:- rank(i, X, rx), edge(X,Y),
:- degree(X, d), ry=0.8 rx/d.



Rank values are not monotonically increase/decrease.

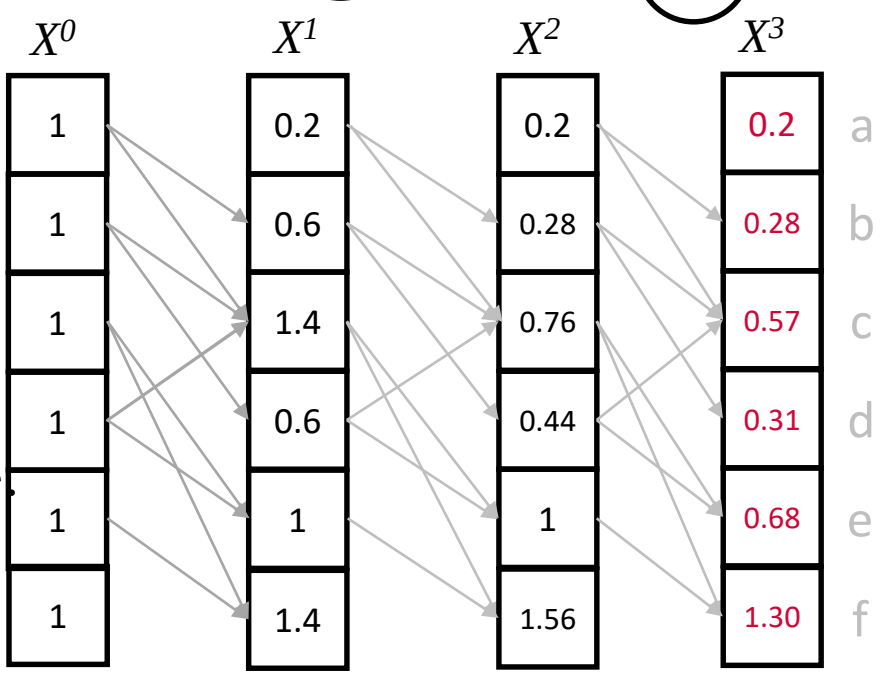
PageRank: A Non-monotonic Example

The monotonicity property



PageRank

rank(i+1, Y, sum[ry]) :- node(Y), ry=0.2;
:- rank(i, X, rx), edge(X,Y),
:- degree(X, d), ry=0.8 rx/d.



Rank values are not monotonically increase/decrease.

PageRank is a non-monotonic algorithm.

Existing Works on Monotonic Conditions

- **Ross and Sagiv** [PODS'92],**SociaLite** [ICDE'13] and **Myria** [VLDB'15] formalize the conditions for using monotonic aggregate in recursive query.

Existing Works on Monotonic Conditions

- **Ross and Sagiv** [PODS'92],**SociaLite** [ICDE'13] and **Myria** [VLDB'15] formalize the conditions for using monotonic aggregate in recursive query.

> As presented in **Myria** [VLDB'15], a recursive aggregate program can be semi-naïve evaluated if
>
> 1. It require the aggregate function to be bag-monotonic.
> 2. The non-aggregate function is monotonic w.r.t. aggregate function.

- Checking on the monotonicity for arbitrary recursive aggregate programs is still sophisticated.

Existing Works on Monotonic Conditions

- **Ross and Sagiv** [PODS'92],**SociaLite** [ICDE'13] and **Myria** [VLDB'15] formalize the conditions for using monotonic aggregate in recursive query.

  Existing works **Maiter** [TPDS'14], **GRAPE**[SIGMOD'17] demonstrate that some non-monotonic algorithm can also be incrementally executed by using a similar approach to semi-naïve evaluation.

  1. PageRank
  2. SimRank
  3. Belief Propagation

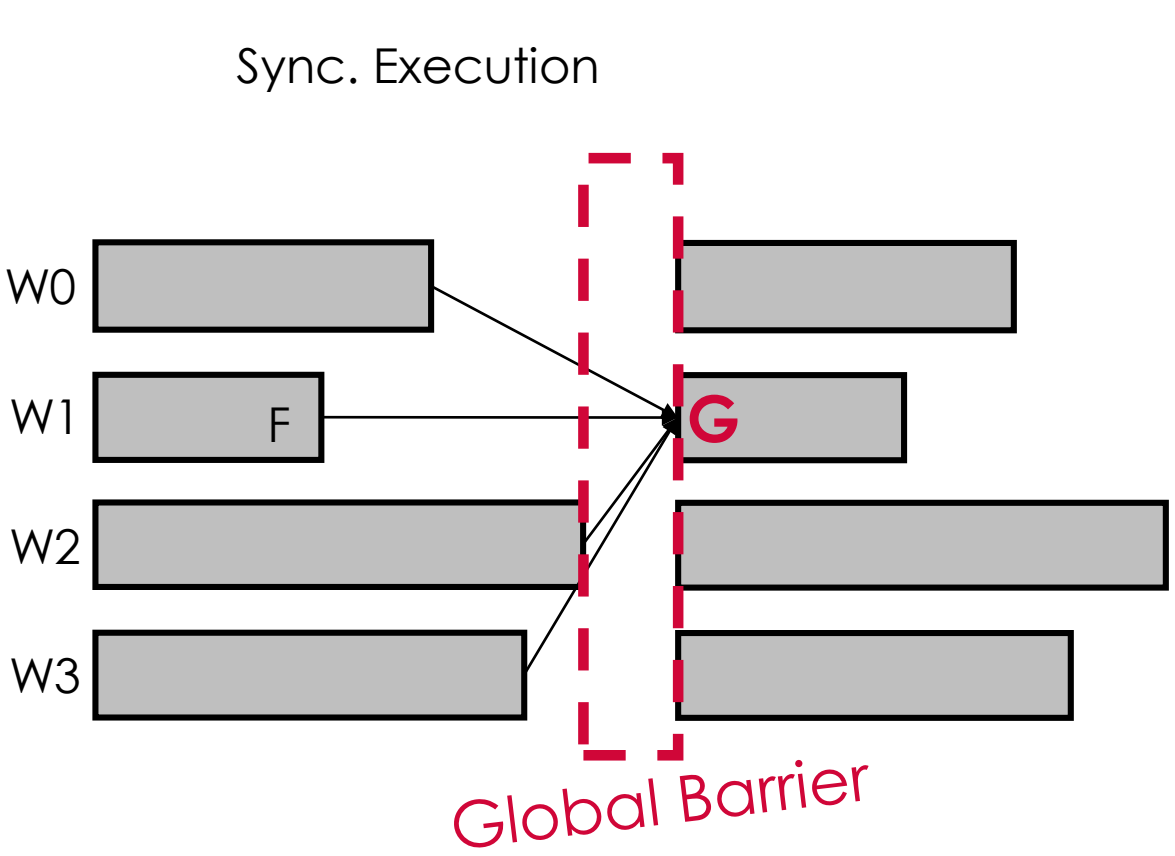
- The application scope of existing conditions is narrow.

Two Research Questions:

- Under what conditions can a Recursive Aggregate Program be evaluated with semi-naïve evaluation?
- How to automatically verify the conditions for evaluating a recursive aggregate program with semi-naïve evaluation?

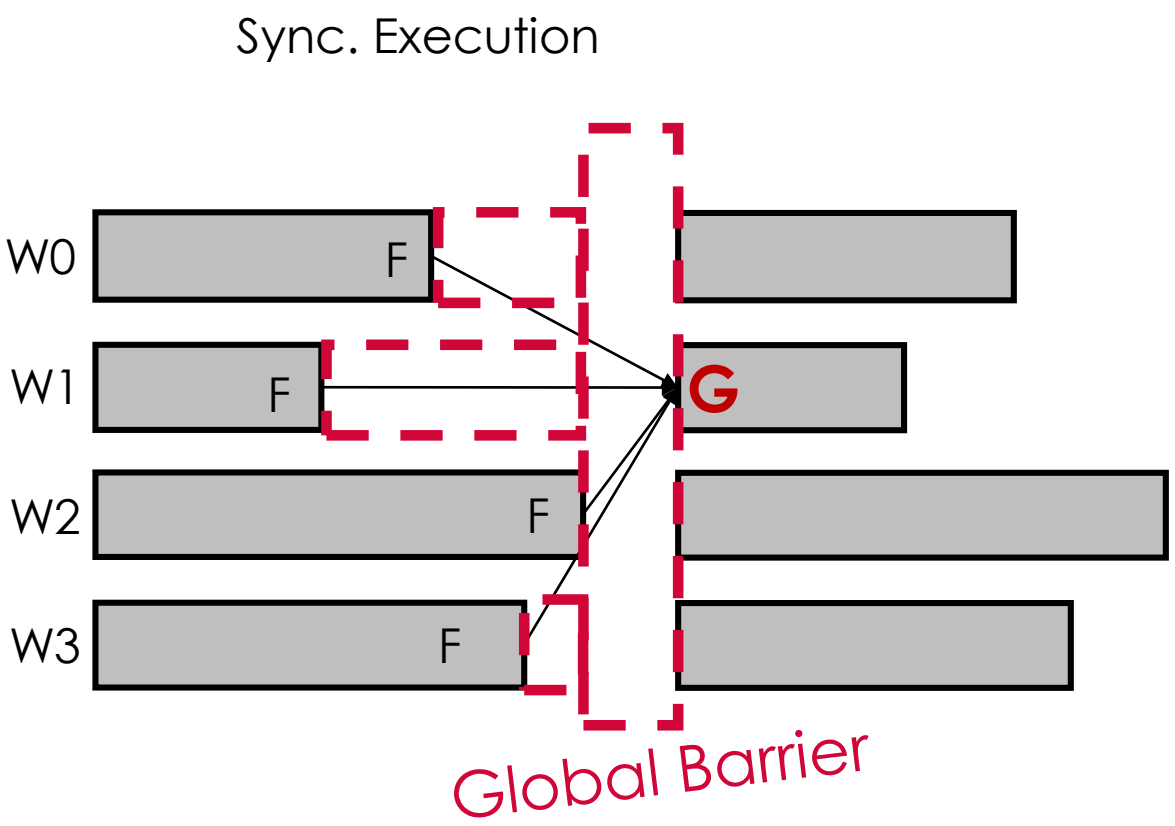
Async. Execution

The aggregate operations require communications among workers.
The global barrier of sync execution can guarantee the correctness.



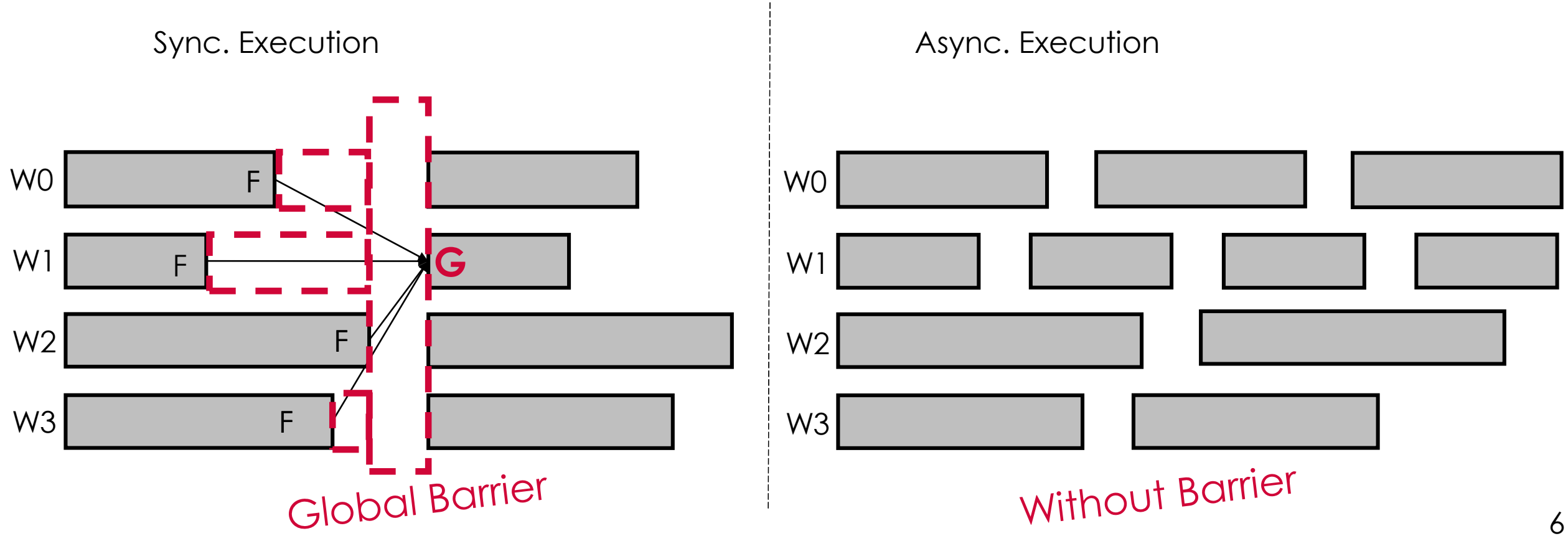
Async. Execution

The aggregate operations require communications among workers.
The global barrier of sync execution can guarantee the correctness.



Async. Execution

Recently, The asynchronous parallel processing in distributed environment have emerged in the past few years, e.g., **Maiter**[TPDS'14], **Giraph++**[VLDB'15] and **Myria** [VLDB'15].



Conditions for Async. Execution

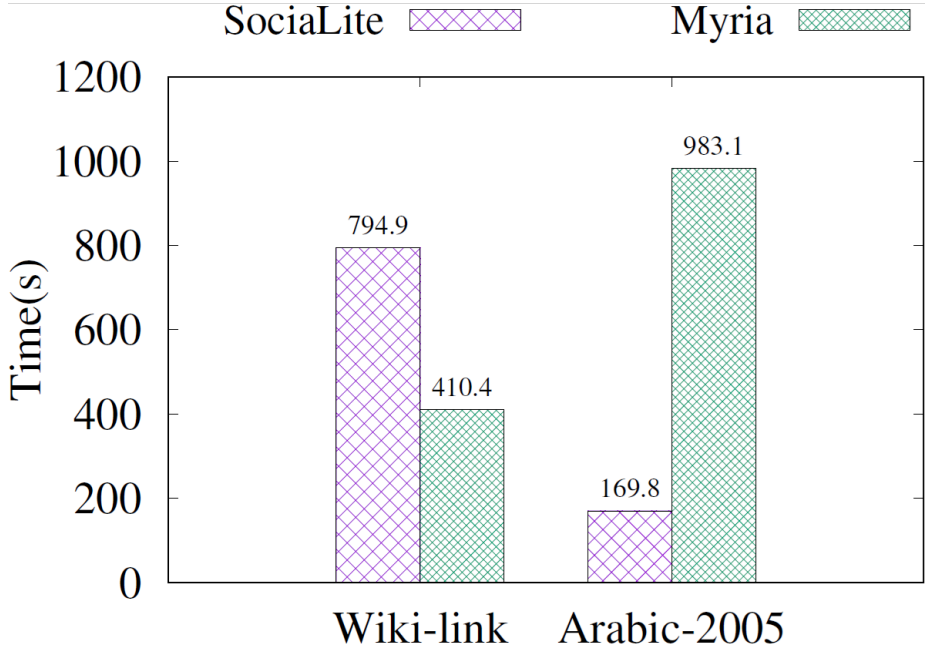
- **Maiter** [TPDS'14] and **Grape+** [SIGMOD'18] propose the correctness conditions for graph algorithms with asynchronous execution.
- The conditions are not general enough for recursive aggregate programs.

Two more Research Questions:

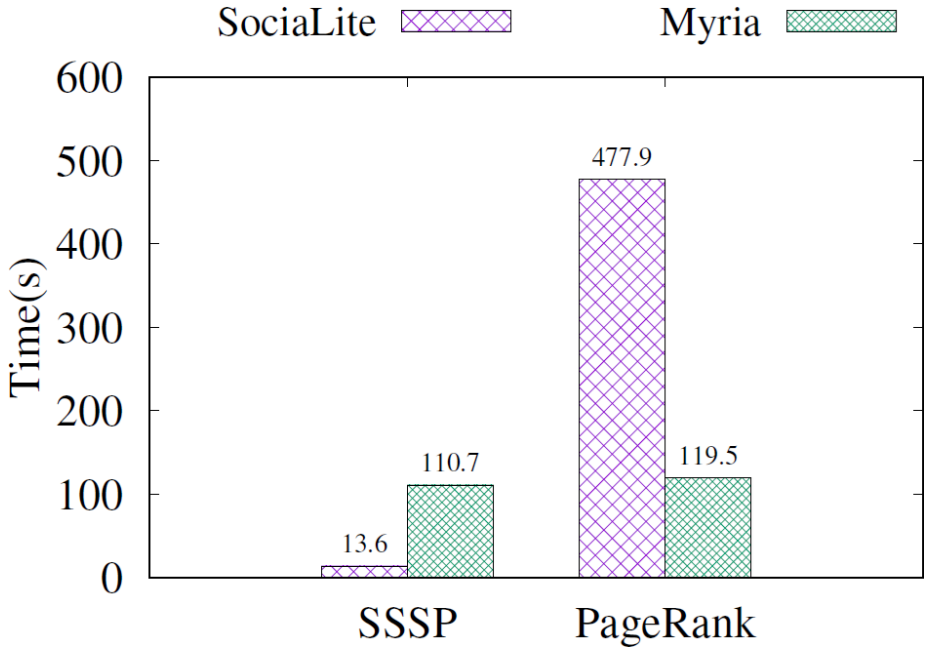
- Under what conditions can a recursive aggregate programs be asynchronously executed ?
- How to automatically verify the conditions for executing a recursive aggregate program with async model?

Neither Sync nor Async Execution is Perfect

Neither Sync nor async execution can outperform each other because
(1) sync processing may be over-controlled (too much idle time)
(2) async processing may be under-controlled (stale computation)



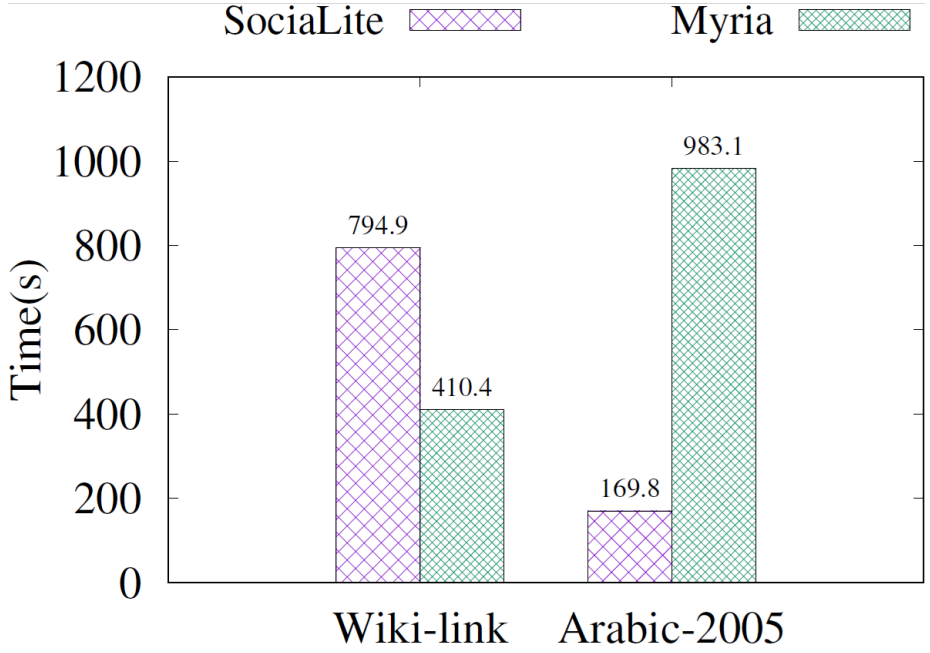
The Same algorithm on different Datasets



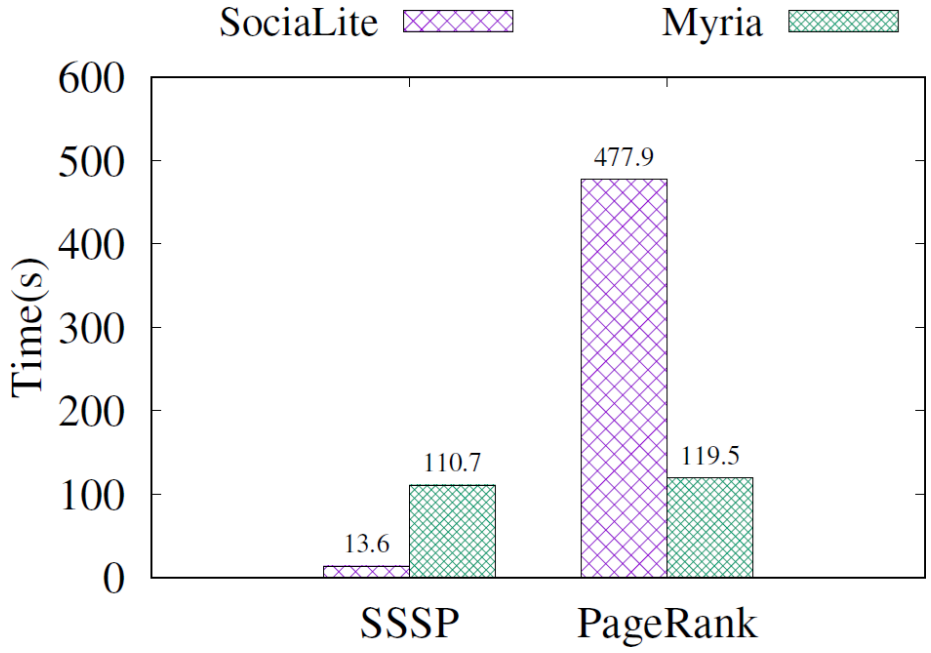
The Same dataset on different Algorithms

Neither Sync nor Async Execution is Perfect

Neither Sync nor async execution can outperform each other because
(1) sync processing may be over-controlled (too much idle time)
(2) async processing may be under-controlled (stale computation)



The Same algorithm on different Datasets



The Same dataset on different Algorithms

We develop a unified sync-async engine to realize properly controlled processing.

Our Work

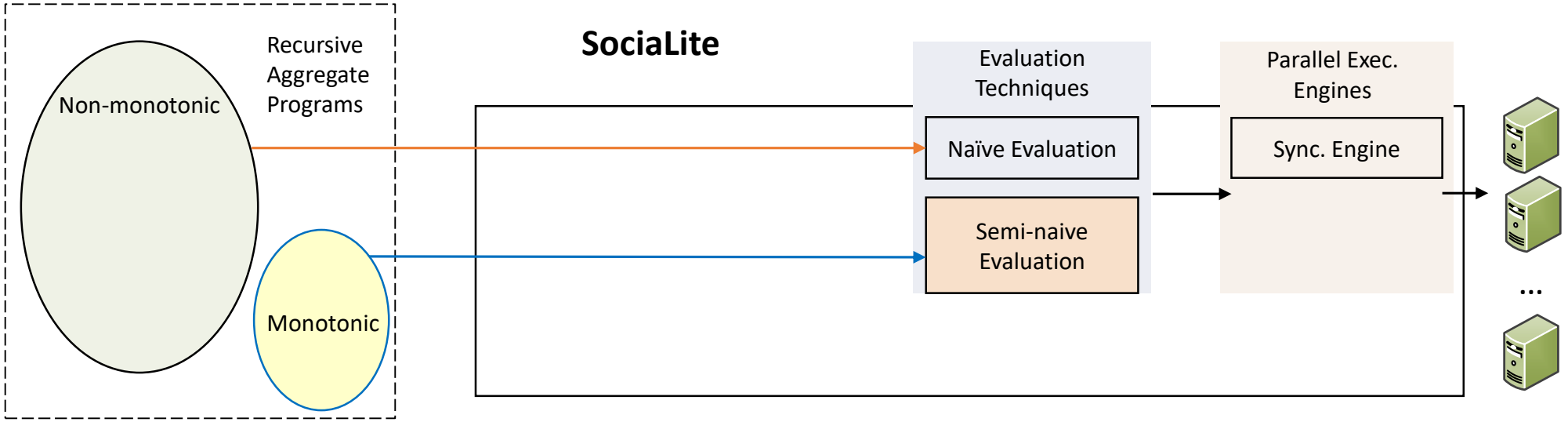
We develop and implement **PowerLog**, a high-performance distributed Datalog system based on **SociaLite** [ICDE'13].

Our Work

We develop and implement **PowerLog**, a high-performance distributed Datalog system based on **SociaLite** [ICDE'13].

SociaLite

(1) processes non-monotonic recursive aggregate programs with Naïve Evaluation
(2) processes monotonic recursive aggregate programs with Semi-naïve Evaluation
(3) executes both evaluation methods with a synchronous engine

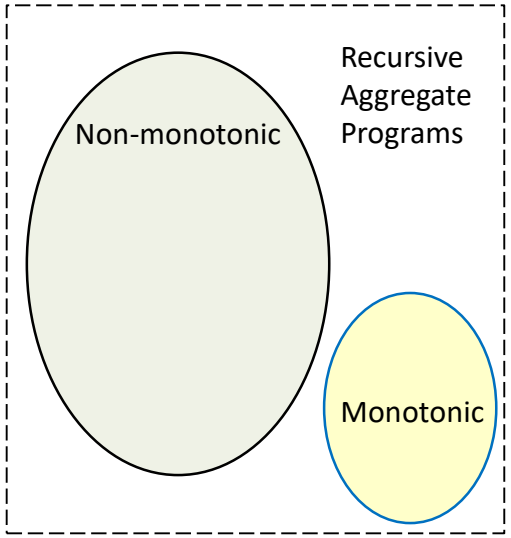


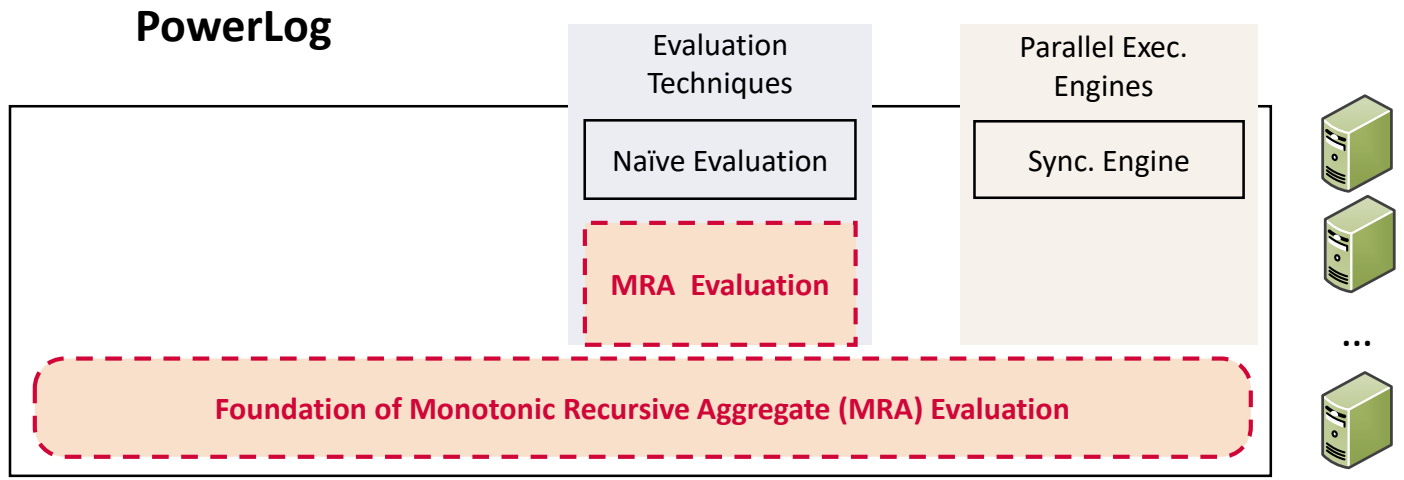
Contributions of PowerLog

We propose **M**onotonic **R**ecursive **A**ggregate evaluation **(MRA Evaluation)**, a variant of Semi-naïve Evaluation to answer.

(1) under what conditions a recursive aggregate program can be executed with Semi-naïve Evaluation **(incrementally)**

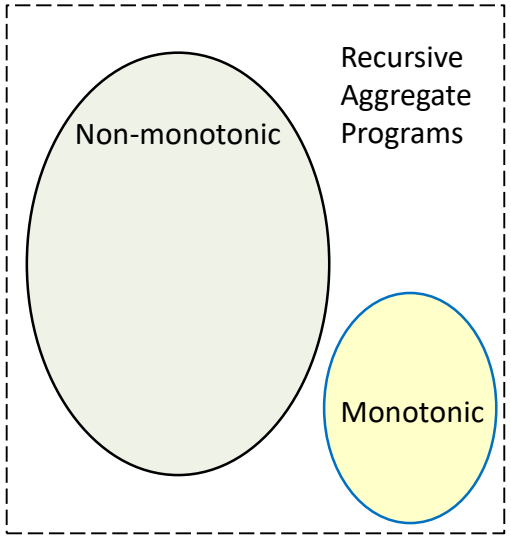
(2) under what conditions a recursive aggregate program can be executed **asynchronously**

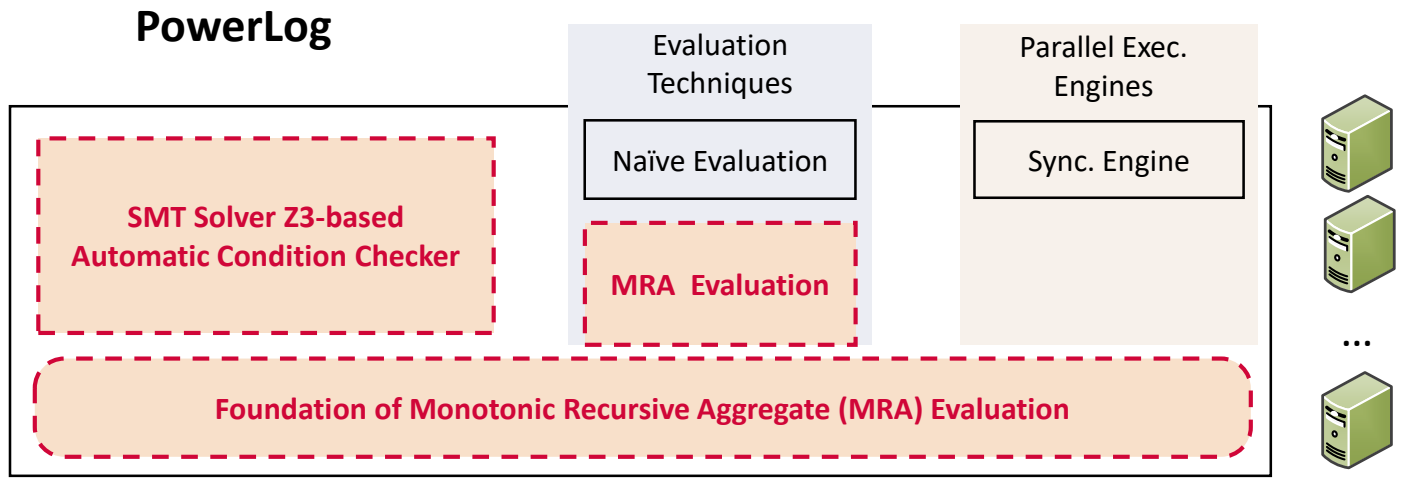




Contributions of PowerLog

We design and implement **a Z3-based condition checker** to **automatically** check if a recursive aggregate program can satisfy the MRA conditions.





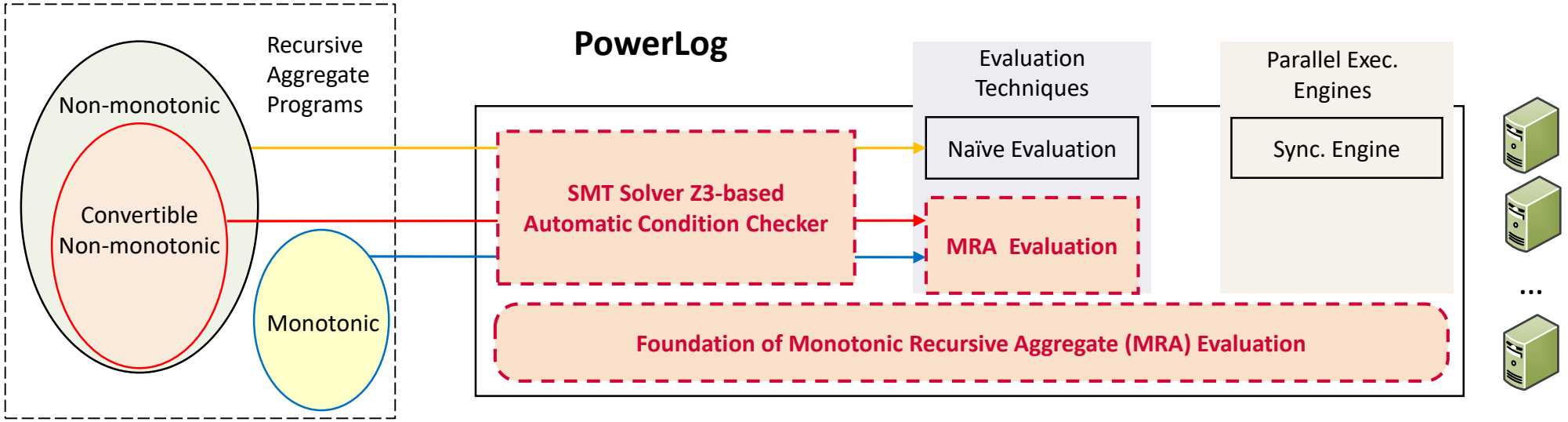
Contributions of PowerLog

We design and implement **a Z3-based condition checker** to **automatically** check if a recursive aggregate program can satisfy the MRA conditions.

(1) monotonic programs will be executed with MRA Evaluation

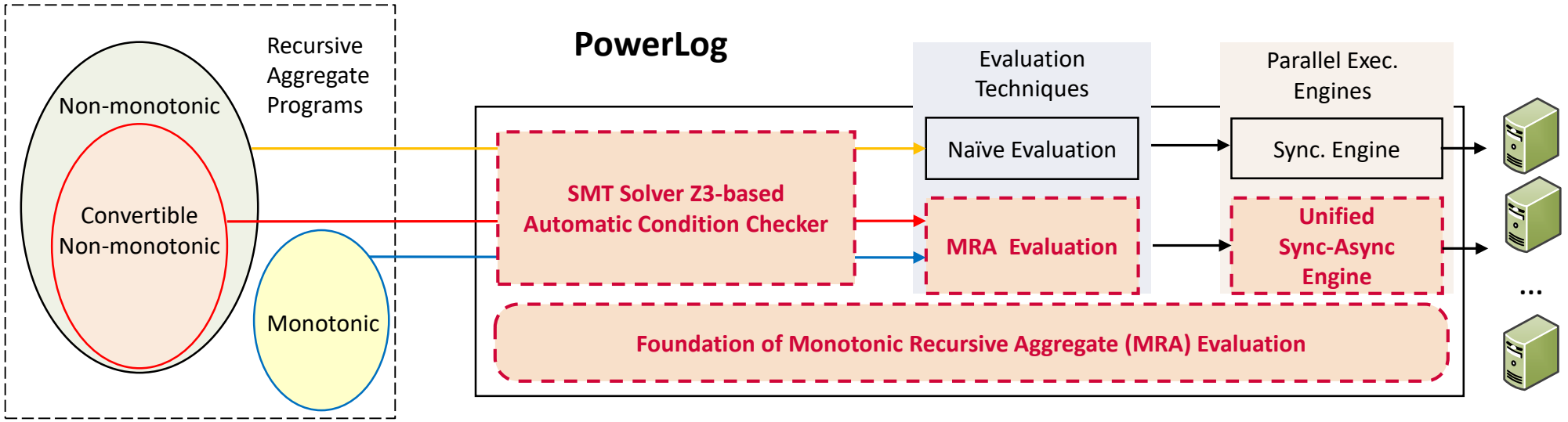
(2) convertible non-monotonic programs that pass the condition check, e.g., PageRank, will be executed with MRA Evaluation

(3) others that cannot pass the condition check will be executed with Naïve Evaluation



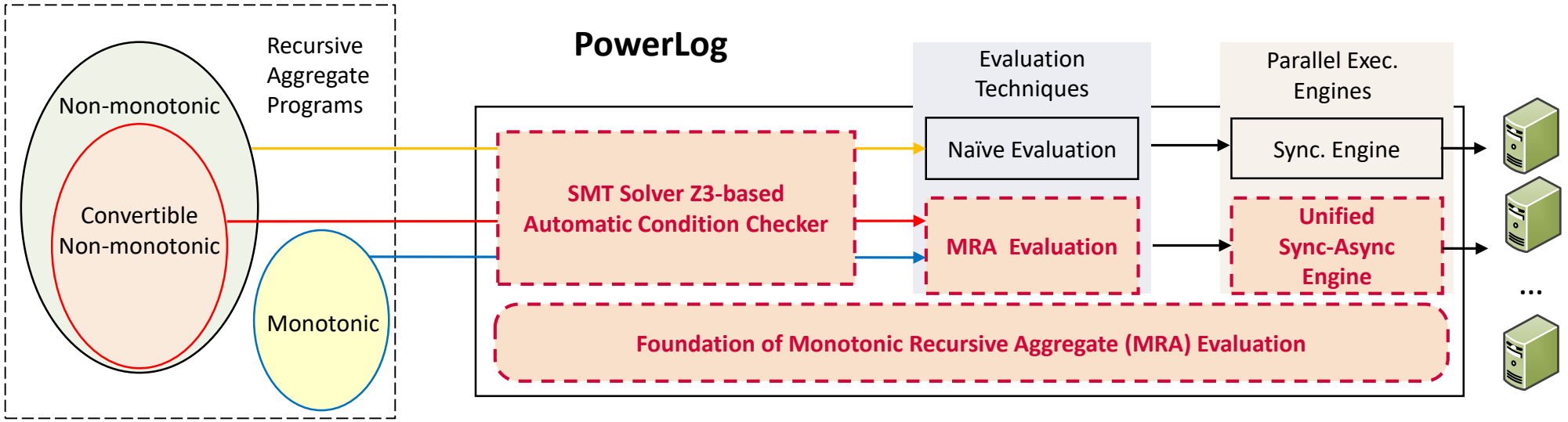
Contributions of PowerLog

We design and implement **a unified sync-async engine** to execute the programs with MRA Evaluation for high performance.



Contributions of PowerLog

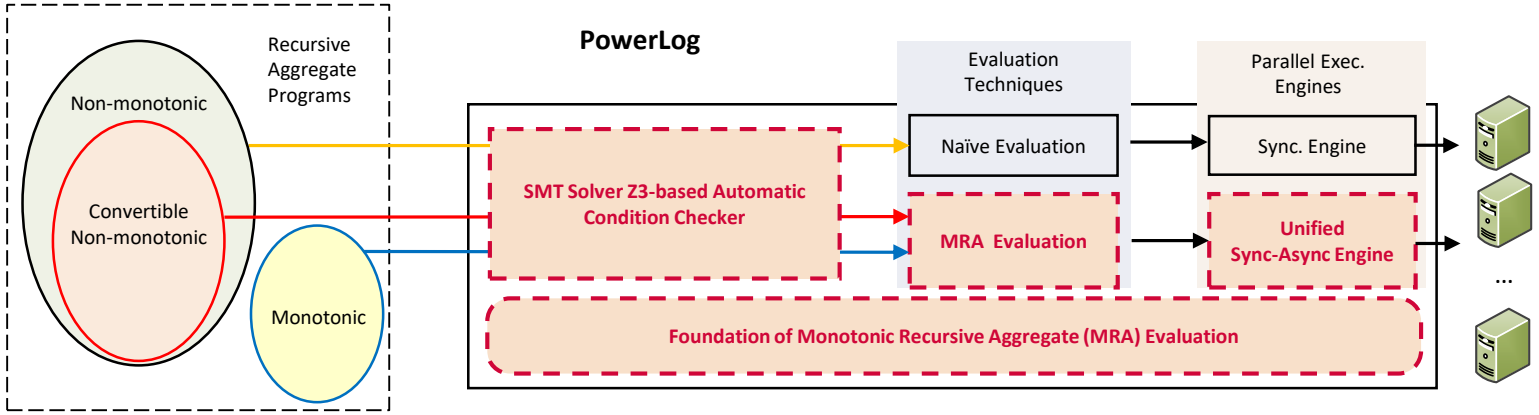
Putting all these together, we develop **PowerLog** **(MRA Evaluation + a Z3-based condition checker + a unified sync-async engine).**



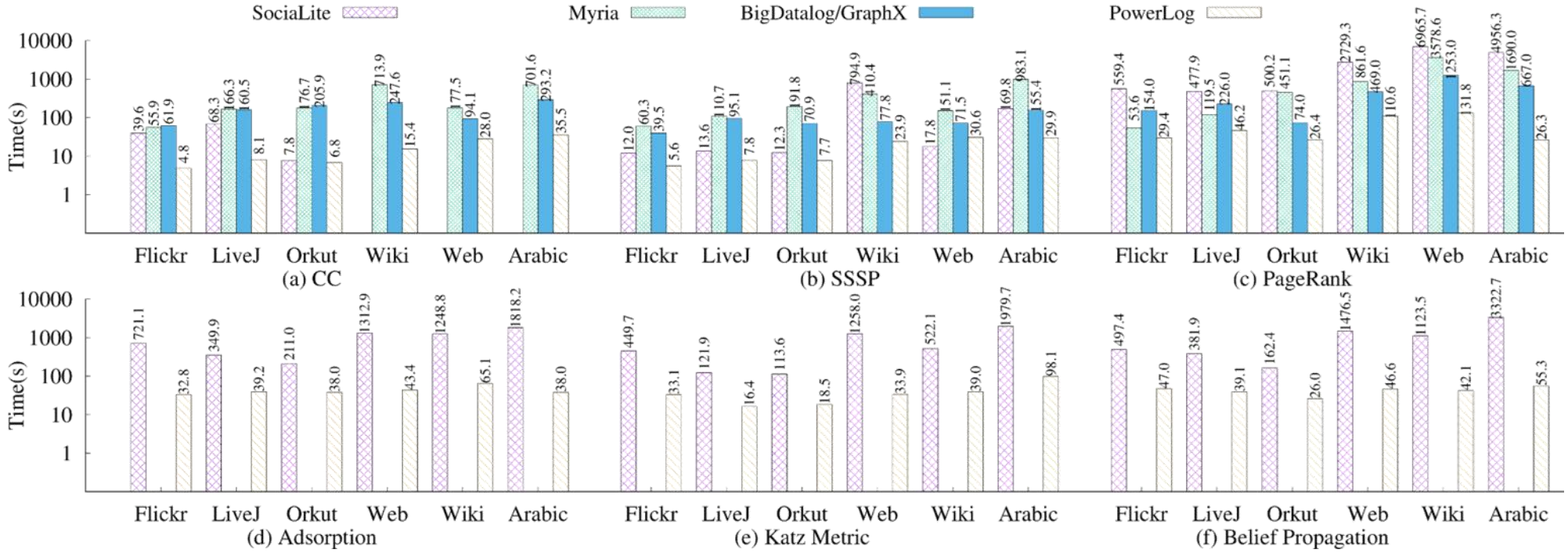
Workload Summary

By using PowerLog, we have checked **14** recursive aggregate programs. **12** of them can be executed incrementally and asynchronously but **2** cannot.

Program	MRA sat.	Aggregator	Program	MRA sat.	Aggregator
SSSP [24]	yes	min	PageRank [39]	yes	sum
CC [24]	yes	min	Adsorption [7]	yes	sum
Katz metric [21]	yes	sum	Belief Propagation [40]	yes	sum
Computing Paths in DAG [50]	yes	count	Cost [50]	yes	sum
Viterbi Algorithm [50]	yes	max	SimRank [20]	yes	sum
Lowest Common Ancestor [44]	yes	min	APSP [50]	yes	min
CommNet [52]	no	sum	GCN-Forward [22]	no	sum



Performance Summary



Compared with three representative Datalog systems **(SociaLite, Myria, BigDatalog)**, PowerLog can achieve **1.1x – 188.3x** speedups on 6 programs and several real datasets.

Summary

The development of PowerLog involves both theory and system development.

- **Scope enhancement**
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.
- **Automatic condition verification**
  We develop a machine tool to eliminate tedious and error-prone efforts.
- **A fast execution engine**
  We implement an highly optimized unified sync-async system.

Summary

The development of PowerLog involves both theory and system development.

- **Scope enhancement**
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.
- **Automatic condition verification**
  We develop a machine tool to eliminate tedious and error-prone efforts.
- **A fast execution engine**
  We implement an highly optimized unified sync-async system.

Questions

Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

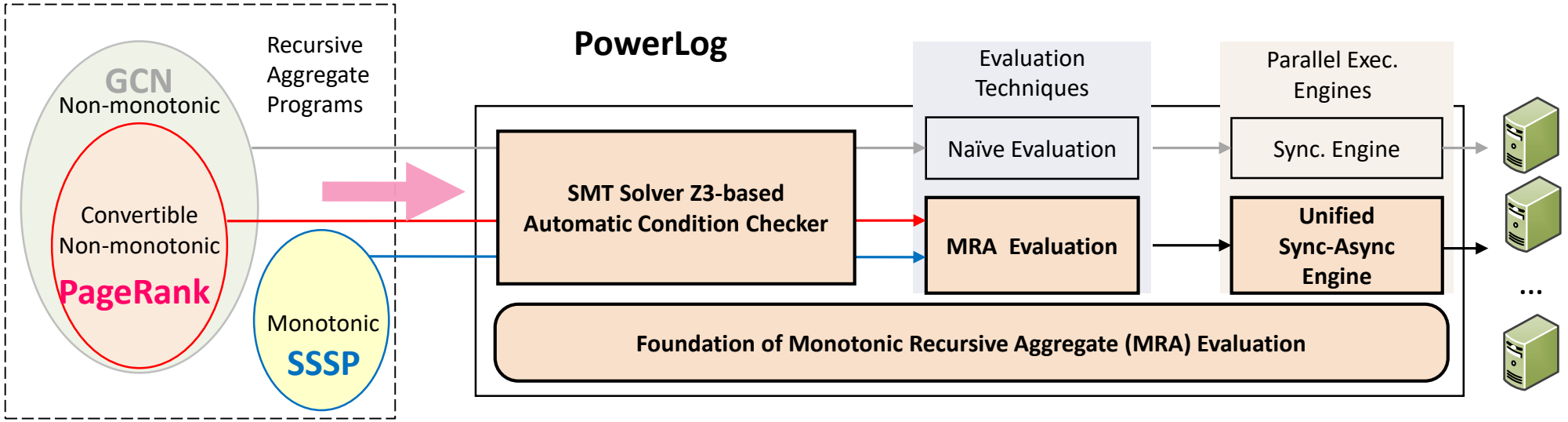
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

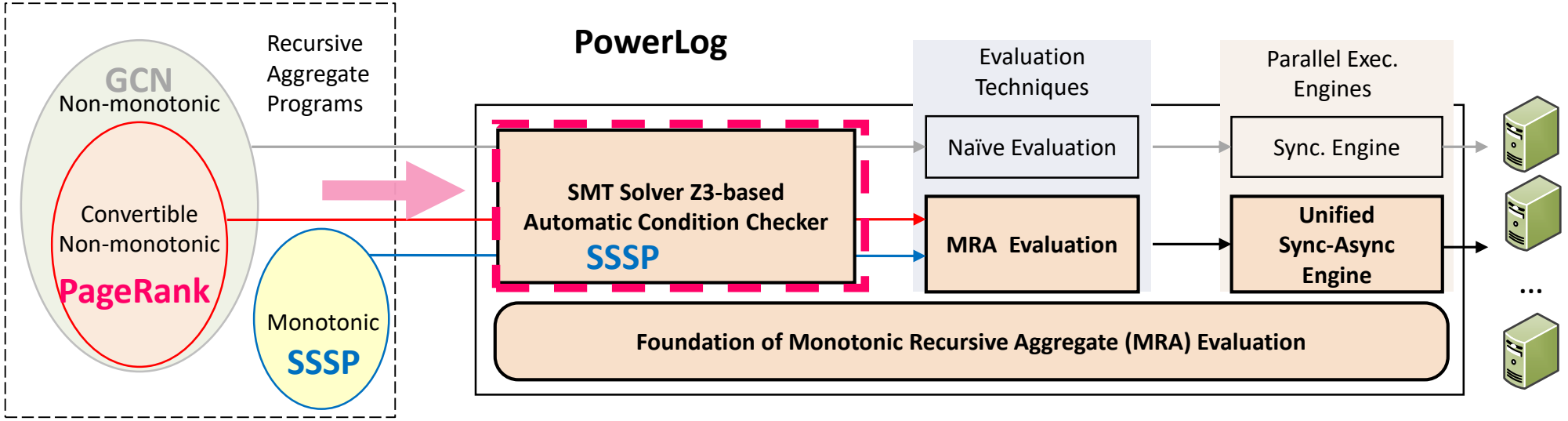
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

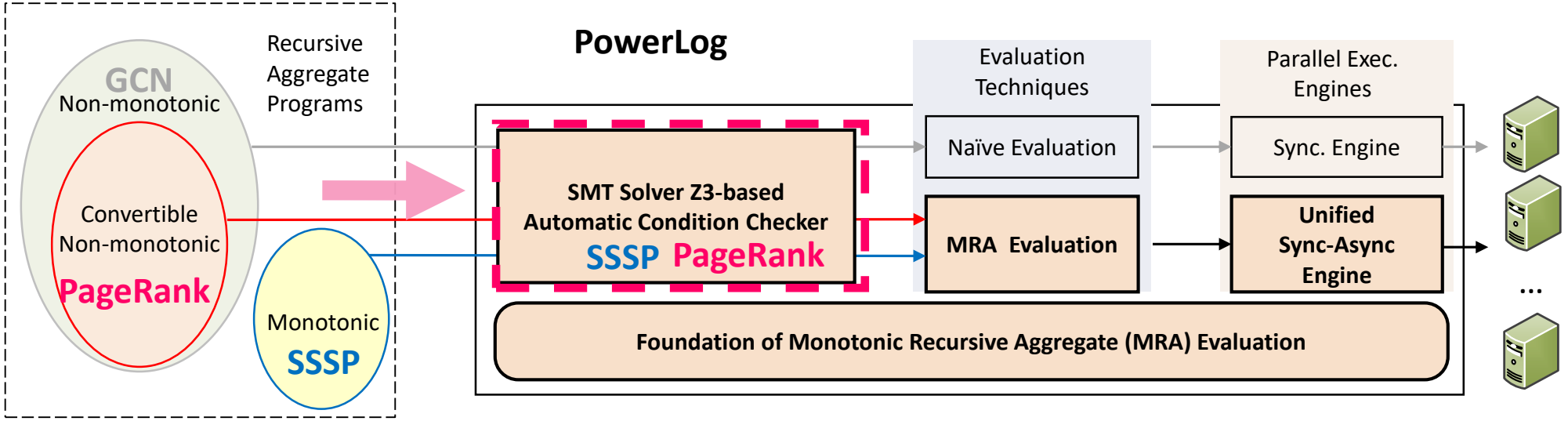
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

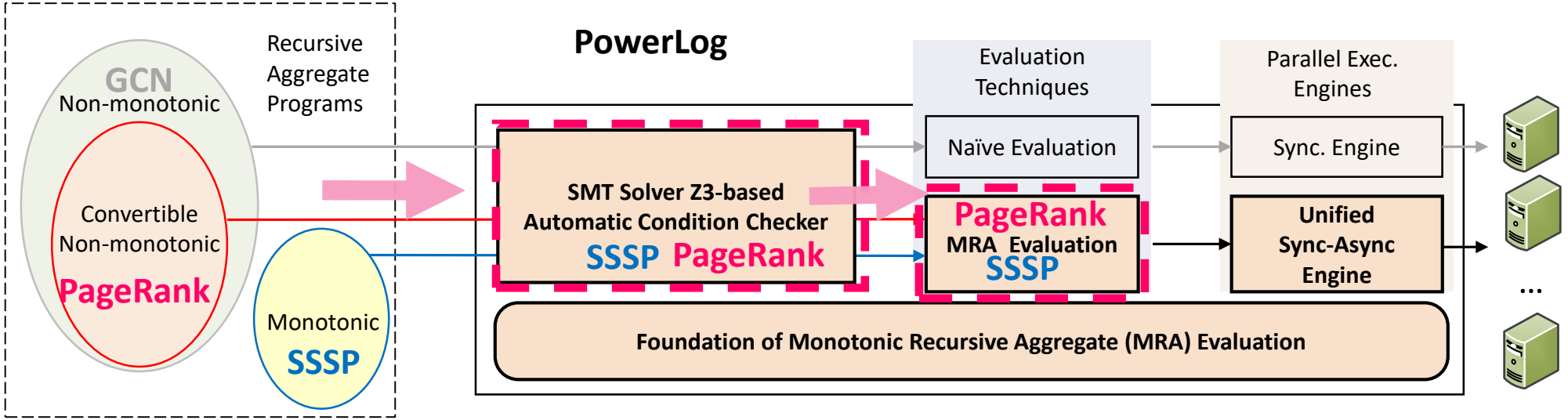
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

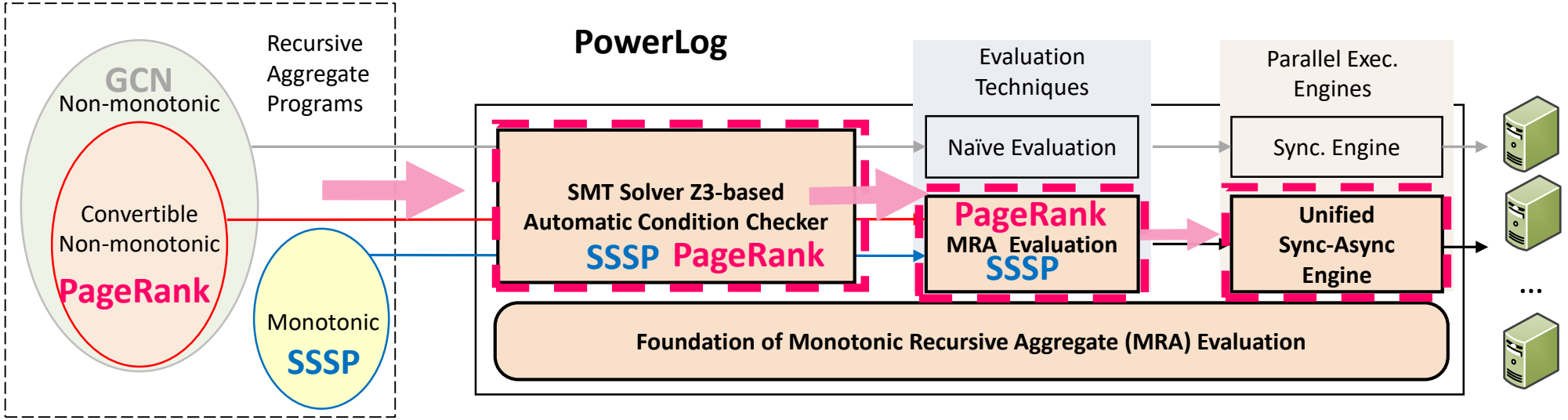
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

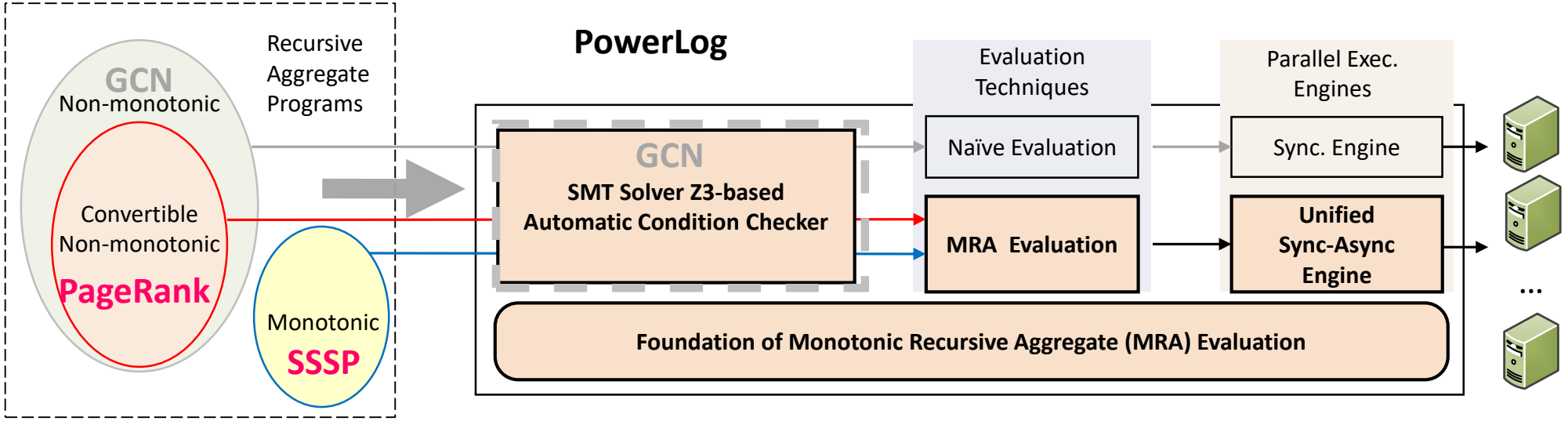
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

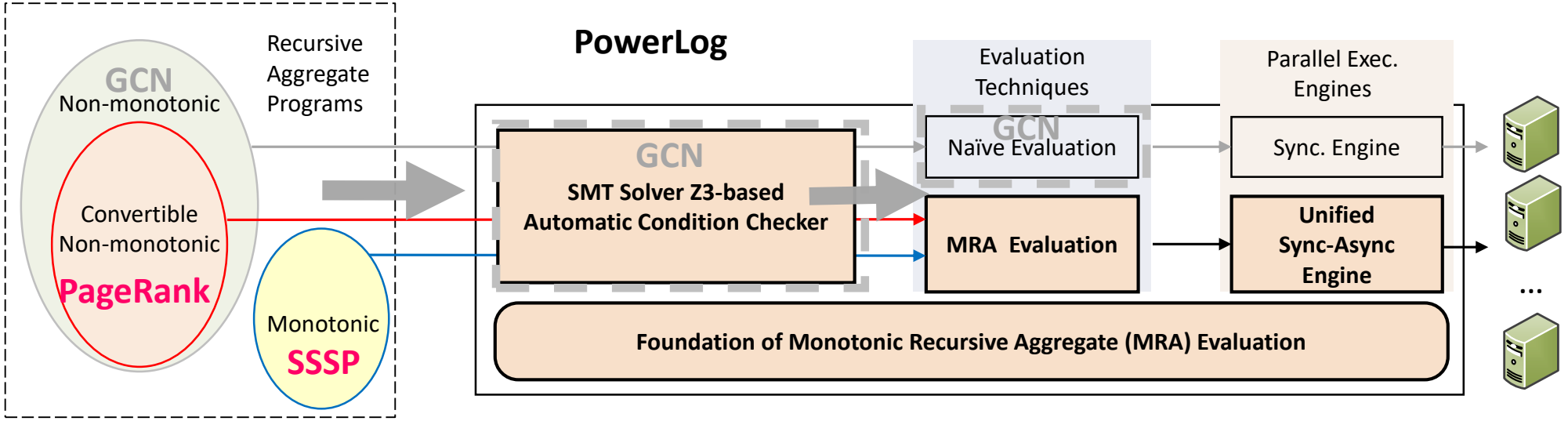
  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.



Contributions:

The development of PowerLog involves both theory and system development.

- Scope enhancement

  We lay an analytical foundation to determine the conditions for monotonic and non-monotonic programs for correct execution.

- Automatic condition verification

  We develop a machine tool to eliminate tedious and error-prone efforts.

- A fast execution engine

  We implement an highly optimized unified sync-async system.