

# Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure

Md. Mostofa Ali Patwary\*   Jean Blair\*\*   Fredrik Manne\*

\*Department of Informatics, University of Bergen, Norway

\*\*Department of EE and CS, United States Military Academy, USA

May 20-22, 2010  
SEA 2010, Italy.

# Overview

- ▶ Extensive experimental study comparing **55** different variations of UNION-FIND algorithm.
- ▶ The study includes:
  - ▶ All the classical algorithms.
  - ▶ Several recently suggested enhancements.
  - ▶ Different combinations and optimizations of these.
- ▶ Main Result: A somewhat **forgotten simple algorithm** developed by **Martin Rem** in 1976 is the **fastest** algorithm.

## Related Experimental Studies

Reference	Application	Computing	# of Algorithms
[Liu, 1990]	Sparse matrix	Factorization	2
[Gilbert et al., 1994]	Sparse matrix	Factorization	6
[Wassenberg et al., 2008]	Image processing	Labeling	8
[Wu et al., 2009]	Image processing	Labeling	3
[Hynes, 1998]	Graphs	Connected components	18
[Osipov et al., 2009]	Graphs	Minimum spanning tree	2

# Outline

## Introduction

Applications and Definitions  
Main Operations (Union-Find)

## Variations of Classical Algorithms

Union Techniques  
Compression Techniques  
Interleaved Algorithms

## Implementation Enhancements

1. Immediate Parent Check [Osipov et al., 2009]
2. Better Interleaved Algorithms [Manne and Patwary, 2009]
3. Memory Efficient Algorithms

## The Fastest Algorithms

# Disjoint-Set Data Structure: Definitions

- ▶  $U \Rightarrow$  set of  $n$  elements and  $S_i \Rightarrow$  a subset of  $U$ .
- ▶  $S_1$  and  $S_2$  are **disjoint** if  $S_1 \cap S_2 = \emptyset$ .
- ▶ **Maintains a dynamic collection**  $S_1, S_2, \dots, S_k$  of disjoint sets which together cover  $U$ .
- ▶ Each set is identified by a **representative**  $x$ .
- ▶ A set of algorithms that operate on this data structure is often referred to as a **UNION-FIND algorithm**.

# Main Operations

- ▶ Each set is represented by a **rooted tree**, pointer towards root.
- ▶ The element in the **root node** is the **representative** of the set.
- ▶ Parent pointer  $p(x)$  denotes the parent of node  $x$ .
- ▶ Two main operations.
  - ▶ **FIND**( $x$ ).
  - ▶ **UNION**( $x, y$ ).

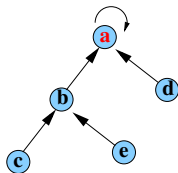


Figure:  $S_i = \{a, b, c, d, e\}$ .

# FIND( $x$ )

- ▶ To which set does a given element  $x$  belong  $\Rightarrow$  **FIND( $x$ )**.
- ▶ Returns the root (representative) of the set that contain  $x$ .

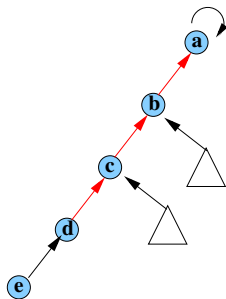


Figure: Find( $d$ ).

# UNION( $x, y$ )

- ▶ Create a new set from the union of two existing sets containing  $x$  and  $y \Rightarrow$  UNION( $x, y$ ).
- ▶ Change the parent pointer of one root to the other one.

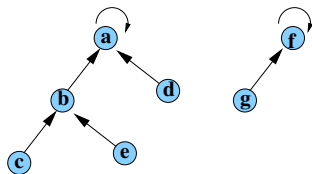


Figure: UNION( $c, g$ ).

# UNION( $x, y$ )

- ▶ Create a new set from the union of two existing sets containing  $x$  and  $y \Rightarrow$  UNION( $x, y$ ).
- ▶ Change the parent pointer of one root to the other one.

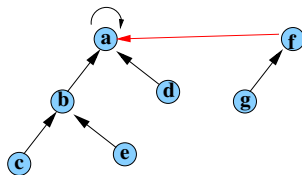


Figure: UNION( $c, g$ ).

Use of UNION-FIND for Computing Connected Components:  $G = (V, E)$ 

```
1:  $S \leftarrow \emptyset$ 
2: for each  $x \in V$  do
3:   MAKESET( $x$ )
4: for each  $(x, y) \in E$  do
5:   if FIND( $x$ )  $\neq$  FIND( $y$ ) then
6:     UNION( $x, y$ )
7:    $S \leftarrow S \cup \{(x, y)\}$ 
```

- ▶ Note that if the edges are processed by increasing weight then this algorithm is **Kruskal's algorithm**.

# UNION Techniques

- ▶ NAIVE-LINK (NL)
- ▶ LINK-BY-SIZE (LS)
- ▶ LINK-BY-RANK (LR)

# UNION Techniques : LINK-BY-RANK (LR)

- ▶ Each set maintains a rank value, initially 0.
- ▶ **Lowest ranked root**  $\Rightarrow$  **higher ranked root.**
- ▶ Equal ranked roots  $\Rightarrow$  root of the combined tree is increased by 1.

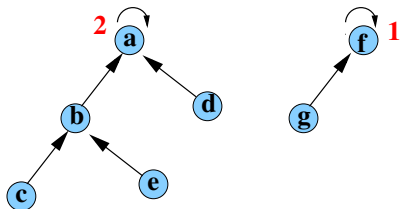


Figure: UNION.

## UNION Techniques : LINK-BY-RANK (LR)

- ▶ Each set maintains a rank value, initially 0.
- ▶ **Lowest ranked root**  
⇒ **higher ranked root.**
- ▶ Equal ranked roots ⇒  
root of the combined  
tree is increased by 1.

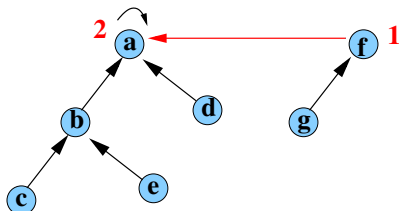


Figure: UNION.

## UNION Techniques : LINK-BY-RANK (LR)

- ▶ Each set maintains a rank value, initially 0.
- ▶ **Lowest ranked root**  
⇒ **higher ranked root.**
- ▶ Equal ranked roots ⇒  
root of the combined tree is increased by 1.

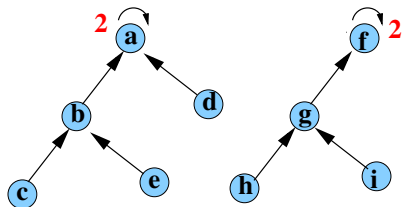


Figure: UNION.

## UNION Techniques : LINK-BY-RANK (LR)

- ▶ Each set maintains a rank value, initially 0.
- ▶ **Lowest ranked root**  
⇒ **higher ranked root.**
- ▶ Equal ranked roots ⇒  
root of the combined tree is increased by 1.

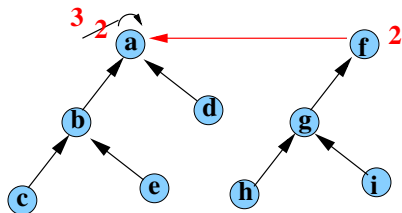


Figure: UNION.

# COMPRESSION Techniques

- ▶ NAIVE-FIND (NF)
- ▶ **PATH-COMPRESSION (PC)**
- ▶ PATH-SPLITTING (PS)
- ▶ **PATH-HALVING (PH)**
- ▶ TYPE-0-REVERSAL (R0)
- ▶ TYPE-1-REVERSAL (R1)
- ▶ **COLLAPSING (CO)**

# COMPRESSION Techniques

- ▶ Reduce the height of the tree during the FIND operation.
- ▶ Subsequent FIND operations require less time.
- ▶ *Find-path* of a node  $x$  is the path of parent pointers from  $x$  up to the root of the tree.

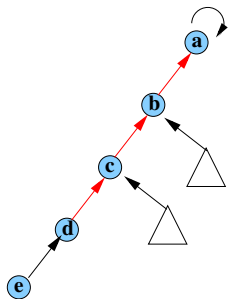


Figure: Find-path( $d$ ).

# COMPRESSION Techniques : PATH-COMPRESSION (PC)

- ▶ Set the parent pointers of all nodes in the find path to the **root**.
- ▶ Need to traverse the find-path **twice**.

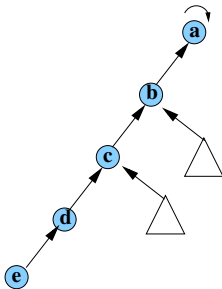


Figure: FIND(e) with PC

# COMPRESSION Techniques : PATH-COMPRESSION (PC)

- ▶ Set the parent pointers of all nodes in the find path to the **root**.
- ▶ Need to traverse the find-path **twice**.

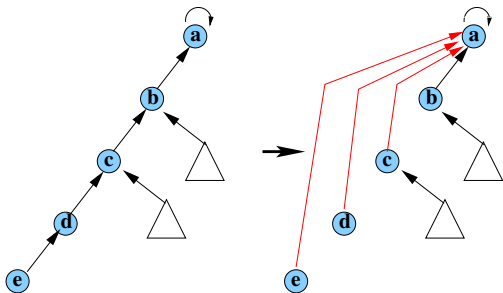


Figure: FIND(e) with PC

# COMPRESSION Techniques : PATH-HALVING (PH)

- ▶ Set the parent pointers of **every other nodes** in the find-path to its **grandparent**.
- ▶ Traverse the find-path **once**.

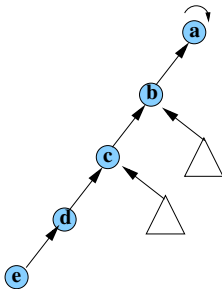


Figure: FIND(*e*) with PH

# COMPRESSION Techniques : PATH-HALVING (PH)

- ▶ Set the parent pointers of **every other nodes** in the find-path to its **grandparent**.
- ▶ Traverse the find-path **once**.

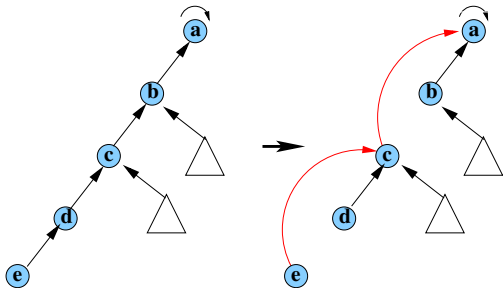


Figure: FIND(*e*) with PH

# COMPRESSION Techniques : COLLAPSING (CO)

- ▶ Every node **points directly** to the root.
- ▶ FIND operation takes **constant time**.
- ▶ In a UNION operation, **all nodes of one tree** point to the root of other tree.

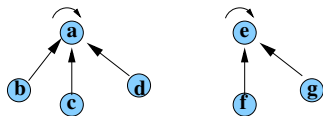


Figure: CO

## COMPRESSION Techniques : COLLAPSING (CO)

- ▶ Every node **points directly** to the root.
- ▶ FIND operation takes **constant time**.
- ▶ In a UNION operation, **all nodes of one tree** point to the root of other tree.

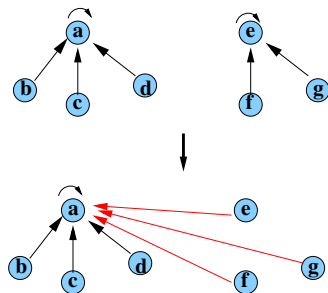


Figure: CO

# Worst Case COMPLEXITY

- ▶ For **any combination** of  $m$  MAKESET, UNION and FIND operations on  $n$  elements.

UNION	COMPRESSION	COMPLEXITY
NL	NF	$O(mn)$
NL	PC, PH, PS	$O(m \log_{(1+m/n)} n)$
NL	CO	$O(m + n^2)$
NL, LR, LS	R0, R1	$O(n + m \log n)$
LR, LS	CO	$O(m + n \log n)$
LR, LS	PC, PH, PS	$O(m \cdot \alpha(m, n))$

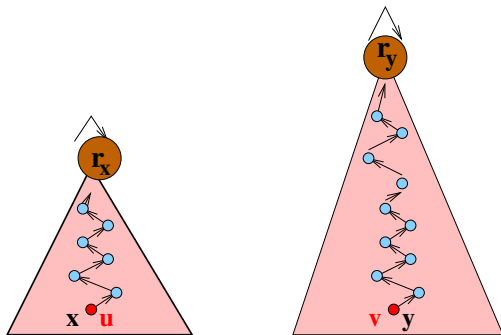
# INTERLEAVED (INT) Algorithm

- ▶ During a UNION operation, the two FIND are performed as a **single interleaved operation**.
- ▶ The first INT algorithm is **Rem's algorithm** [Dijkstra, 1976].

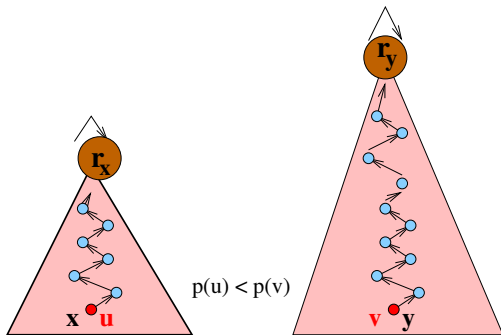
# The Rem Algorithm [Dijkstra, 1976]

- ▶ Each node has a **unique identifier**  $\Rightarrow$  index of the node.
- ▶ Lowered numbered node points to higher numbered node or to itself (if it is a root).

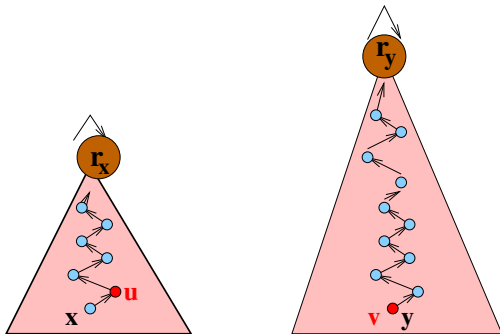
# The Rem Algorithm: Example - Edge $(x, y)$



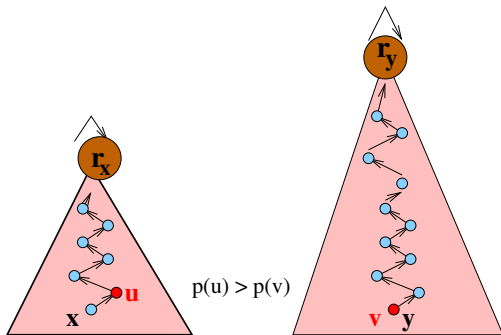
## The Rem Algorithm: Different Sets - Compare ID of Parents



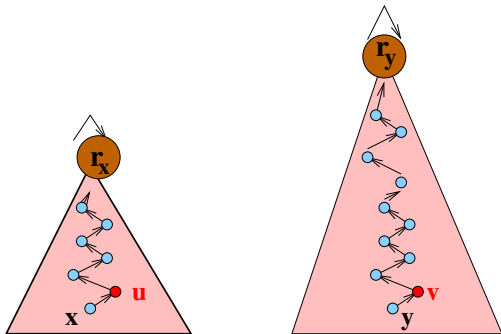
## The Rem Algorithm: Different Sets - Compare ID of Parents



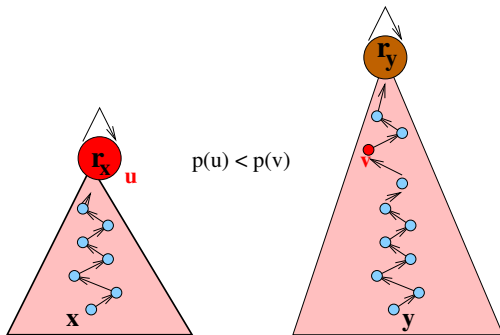
## The Rem Algorithm: Different Sets - Compare ID of Parents



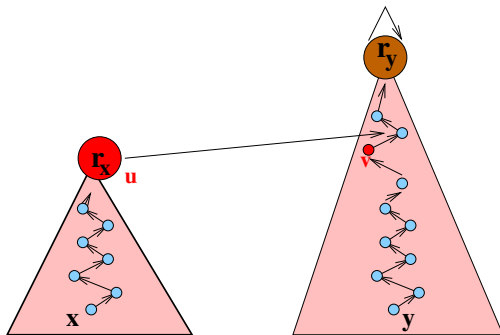
## The Rem Algorithm: Different Sets - Compare ID of Parents



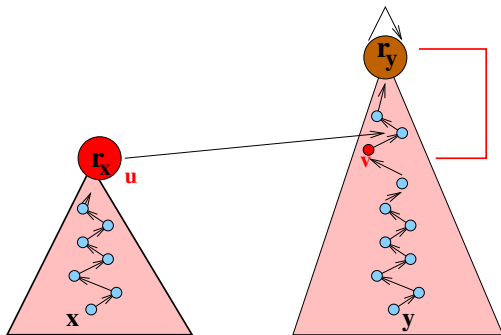
## The Rem Algorithm: Different Sets - Compare ID of Parents



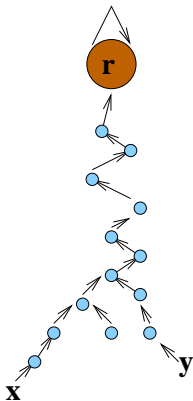
## The Rem Algorithm: Different Sets - Compare ID of Parents



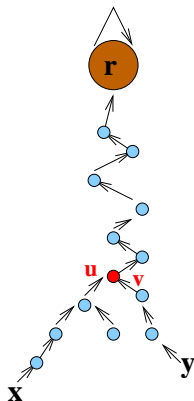
## The Rem Algorithm: Different Sets - Compare ID of Parents



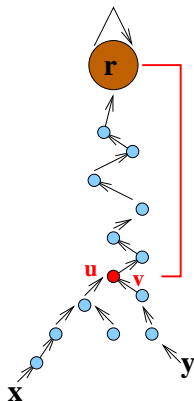
# The Rem Algorithm: Same Set - Edge $(x, y)$



# The Rem Algorithm: Same Set - Edge $(x, y)$



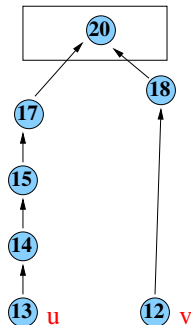
# The Rem Algorithm: Same Set - Edge $(x, y)$



# The Rem Algorithm: Compression

## SPLICING (SP)

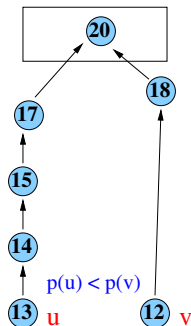
- ▶ Set parent pointer to a higher valued node  $\Rightarrow$  compressing the tree.
- ▶ Intuition: Higher valued node should be closer to the root.
- ▶ The running time of **RemSP** is  $O(m \log_{(2+m/n)} n)$ .



# The Rem Algorithm: Compression

## SPLICING (SP)

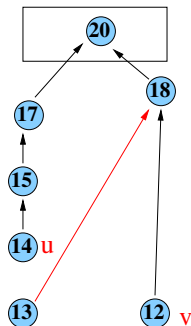
- ▶ Set parent pointer to a higher valued node  $\Rightarrow$  compressing the tree.
- ▶ Intuition: Higher valued node should be closer to the root.
- ▶ The running time of **RemSP** is  $O(m \log_{(2+m/n)} n)$ .



# The Rem Algorithm: Compression

## SPLICING (SP)

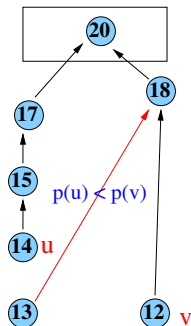
- ▶ Set parent pointer to a higher valued node  $\Rightarrow$  compressing the tree.
- ▶ Intuition: Higher valued node should be closer to the root.
- ▶ The running time of **RemSP** is  $O(m \log_{(2+m/n)} n)$ .



# The Rem Algorithm: Compression

## SPLICING (SP)

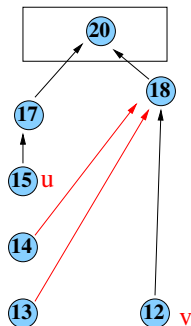
- ▶ Set parent pointer to a higher valued node  $\Rightarrow$  compressing the tree.
- ▶ Intuition: Higher valued node should be closer to the root.
- ▶ The running time of **RemSP** is  $O(m \log_{(2+m/n)} n)$ .



# The Rem Algorithm: Compression

## SPLICING (SP)

- ▶ Set parent pointer to a higher valued node  $\Rightarrow$  compressing the tree.
- ▶ Intuition: Higher valued node should be closer to the root.
- ▶ The running time of **RemSP** is  $O(m \log_{(2+m/n)} n)$ .



## A Variation of Rem: “TVL” [Tarjan and van Leeuwen, 1984]

- ▶ Uses **ranks** rather than **identifier**.
- ▶ This algorithms is slightly **more complicated** than Rem.

## Test Sets and Experimental Setup

- ▶ Dell computer, Intel Core 2 CPU (2.40 GHz), Fedora 10, C++ and GCC (-O3).
- ▶ Three test sets.
  1. rw: 9 real world graphs.
    - ▶ Linear programming, Medical science.
    - ▶ Structural engineering, Civil engineering.
    - ▶ Automotive industry.
  2. sw: 5 synthetic small world graphs .
  3. er: 6 synthetic Erdős-Rényi random graphs.
- ▶ For each graph: 5 runs with 5 different random orderings of edges.

# Structural Properties of the Input Graphs

Graph	$ V $	$ E $	Comp	Max Deg	Avg Deg	# Edges Processed
rw1 (m.t1)	97,578	4,827,996	1	236	99	692,208
rw2 (crankseg_2)	63,838	7,042,510	1	3,422	221	803,719
rw3 (inline_1)	503,712	18,156,315	1	842	72	5,526,149
rw4 (ldoor)	952,203	22,785,136	1	76	48	7,442,413
rw5 (af_shell10)	1,508,065	25,582,130	1	34	34	9,160,083
rw6 (boneS10)	914,898	27,276,762	1	80	60	11,393,426
rw7 (bone010)	986,703	35,339,811	2	80	72	35,339,811
rw8 (audikw_1)	943,695	38,354,076	1	344	81	10,816,880
rw9 (spal_004)	321,696	45,429,789	1	6,140	282	28,262,657
sw1	50,000	6,897,769	17,233	6,241	276	6,897,769
sw2	75,000	12,039,043	9,467	8,624	321	12,039,043
sw3	100,000	16,539,557	34,465	10,470	331	16,539,557
sw4	175,000	26,985,391	43,931	14,216	308	26,985,391
sw5	200,000	34,014,275	68,930	16,462	340	34,014,275
er1	100,000	453,803	24	25	9	453,803
er2	100,000	1,650,872	1	61	33	603,141
er3	500,000	2,904,660	8	30	12	2,904,660
er4	1,000,000	5,645,880	31	31	11	5,645,880
er5	500,000	9,468,353	1	70	38	3,476,740
er6	1,000,000	20,287,048	1	76	41	7,347,376

# Relative performance of the classical algorithms

CL	Compression Technique								
	UNION	NF	PC	PH	PS	CO	R0	R1	SP
NL									X
LR			1	2					X
LS									X
Rem				X		X	X	X	3
TVL				X		X	X	X	

Table: 29 variations of classical algorithms. Each cell is an algorithm.

# $X$ dominates $Y$

- ▶ An algorithm  $X$  dominates another algorithm  $Y$  if  $X$  performs at least as well as  $Y$ .
- ▶ Since **LRPC** and **LRPH** are generally accepted as best, we begin by examining these.

# Relative performance of the classical algorithms

CL	Compression Technique								
	UNION	NF	PC	PH	PS	CO	R0	R1	SP
NL									X
LR		1		2					X
LS									X
Rem				X		X	X	X	3
TVL				X		X	X	X	

Table: 29 variations of classical algorithms.

# Relative performance of the classical algorithms

CL	Compression Technique								
	UNION	NF	PC	PH	PS	CO	R0	R1	SP
NL	LRPC <sub>1</sub>					LRPC <sub>1</sub>	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
LR	LRPC <sub>1</sub>	①	②				LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
LS	LRPC <sub>1</sub>	LRPC <sub>1</sub>					LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
Rem	LRPC <sub>1</sub>			✗		✗	✗	✗	③
TVL	LRPC <sub>1</sub>	LRPC <sub>1</sub>		✗		✗	✗	✗	

Table: LRPC dominates 14 algorithms.

# Relative performance of the classical algorithms

CL	Compression Technique							
	NF	PC	PH	PS	CO	R0	R1	SP
NL	LRPC <sub>1</sub>	LRPH <sub>2</sub>			LRPC <sub>1</sub>	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
LR	LRPC <sub>1</sub>	① LRPC <sub>1</sub>	②			LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
LS	LRPC <sub>1</sub>	LRPC <sub>1</sub>				LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
Rem	LRPC <sub>1</sub>		✗		✗	✗	✗	③
TVL	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗		✗	✗	✗	LRPH <sub>2</sub>

Table: LRPC<sub>1</sub> dominates 3 additional, including LRPC<sub>1</sub> - Total 17.

# Relative performance of the classical algorithms

CL	Compression Technique							
	NF	PC	PH	PS	CO	R0	R1	SP
NL	LRPC <sub>1</sub>	LRPH <sub>2</sub>	RemSP <sub>3</sub>	RemSP <sub>3</sub>	LRPC <sub>1</sub>	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
LR	LRPC <sub>1</sub>	① LRPH <sub>2</sub>	② RemSP <sub>3</sub>	RemSP <sub>3</sub>	RemSP <sub>3</sub>	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
LS	LRPC <sub>1</sub>	LRPC <sub>1</sub>	RemSP <sub>3</sub>	RemSP <sub>3</sub>	RemSP <sub>3</sub>	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗
Rem	LRPC <sub>1</sub>	RemSP <sub>3</sub>	✗		✗	✗	✗	③
TVL	LRPC <sub>1</sub>	LRPC <sub>1</sub>	✗	RemSP <sub>3</sub>	✗	✗	✗	LRPH <sub>2</sub>

Table: RemSP dominates 10 of remaining, including LRPH - Total 27.



# Implementation Enhancements

- ▶ Ways to make the classical algorithms faster.
- ▶ Enhancements:
  1. **Immediate parent check** [Osipov et al., 2009].
  2. **Better interleaved** algorithms [Manne and Patwary, 2009].
  3. **Memory efficient** algorithms, Reduce memory usage.

## Enhancement 1: Immediate Parent Check (IPC) [Osipov et al., 2009]

- ▶ IPC applies to **any classical algorithm** (Rem already implements IPC).
- ▶  $\{\text{IPC}\} \times \{\text{LR}, \text{LS}\} \times \{\text{PC}, \text{PH}, \text{PS}\}$
- ▶  $\{\text{IPC}\} \times \{\text{TVL}\} \times \{\text{PC}, \text{PS}, \text{SP}\}$
- ▶ **9** more variations.
- ▶ **RemSP** dominates all **9** variations.

## Enhancement 2: Better Interleaved Algorithms (INT) [Manne and Patwary, 2009]

- ▶ Better interleaved algorithms than  $TvL \Rightarrow eTvL, ZZ$ .
- ▶  $\{eTvL\} \times \{PC, PS, SP\}$
- ▶  $\{ZZ\} \times \{PC, PS\}$
- ▶ **5** more variations.
- ▶ **RemSP** dominates all **5** variations.

## Enhancement 3: Memory Efficient (MS) Algorithms

- ▶ Reduce memory used by each algorithm.
- ▶  $\{\text{MS}\} \times \{\text{NL}\} \times \{\text{PC}, \text{PS}\}$ .
- ▶  $\{\text{MS}\} \times \{\text{LR}, \text{LS}\} \times \{\text{PC}, \text{PS}, \text{CO}\}$ .
- ▶  $\{\text{MS}\} \times \{\text{IPC}\} \times \{\text{LR}, \text{LS}\} \times \{\text{PC}, \text{PS}\}$ .
- ▶ 12 more variations.
- ▶ Note that Rem does not use size or rank, so it is automatically an MS algorithm.

## Enhancement 3: MS relative performance

MS-UNION Method	Compression Technique			
	PC	PS	CO	SP
NL			<del></del>	<del></del>
LR				<del></del>
LS				<del></del>
IPC-LR			<del></del>	<del></del>
IPC-LS			<del></del>	<del></del>
Rem	RemSP <sub>3</sub> ≡	undom. ≡	<del></del>	③ undom. ≡

Table: 12 more variations. Shaded row has already been considered.

## Enhancement 3: MS relative performance

MS-UNION Method	Compression Technique			
	PC	PS	CO	SP
NL	RemSP <sub>3</sub>	RemSP <sub>3</sub>	<del>X</del>	<del>X</del>
LR	RemSP <sub>3</sub> ↑			<del>X</del>
LS	RemSP <sub>3</sub> ↑			<del>X</del>
IPC-LR			<del>X</del>	<del>X</del>
IPC-LS	RemSP <sub>3</sub>	RemSP <sub>3</sub>	<del>X</del>	<del>X</del>
Rem	RemSP <sub>3</sub> ≡	undom. ≡	<del>X</del>	③ undom. ≡

Table: RemSP dominates 6 algorithms - Total 47 of 55.

## Enhancement 3: MS relative performance ▶ Figure

MS-UNION Method	Compression Technique			
	PC	PS	CO	SP
NL	RemSP <sub>3</sub>	RemSP <sub>3</sub>	<del>X</del>	<del>X</del>
LR	RemSP <sub>3</sub> ↑	undom.↑	undom.↑	<del>X</del>
LS	RemSP <sub>3</sub> ↑	undom.↑	undom.↑	<del>X</del>
IPC-LR	undom.↑	undom.	<del>X</del>	<del>X</del>
IPC-LS	RemSP <sub>3</sub>	RemSP <sub>3</sub>	<del>X</del>	<del>X</del>
Rem	RemSP <sub>3</sub> ≡	undom.≡	<del>X</del>	③ undom.≡

Table: 6 algorithms are undominated - Total 8 undominated of 55.

# The Fastest Algorithms

- ▶ **Different metric** than the dominates technique.
- ▶ **Fictious algorithm** (GLOBAL-MIN)  $\Rightarrow$  run-time equal to the best of any algorithm for **each graph**.
- ▶ For **each algorithm**
  1. Compute average relative time **for each graph**.
  2. Compute average relative time for **each type of graph** (rw, sw and er).
  3. Compute **average of the three types**  $\Rightarrow$  **final average**.
- ▶ **Rank** the order of the algorithms based on **final averages**.

# Rank order of the fastest algorithms ▸ Figure

Algorithm	Rank based on graphs of type			
	All graphs	Real-World	Small-World	Erdős-Rényi
<b>RemSP</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>
RemPS	2	5	2	4
MS-IPC-LRPC	3	6	3	7
MS-LSPS	4	2	10	2
MS-IPC-LSPC	5	8	4	8
MS-LRPS	6	4	13	3
MS-IPC-LRPS	7	3	11	5
MS-LSCO	8	9	6	9
MS-LRCO	9	10	5	10
MS-IPC-LSPS	10	7	15	6

# The Fastest Algorithms: Observations

- ▶ All the top 10 algorithms use MS enhancement.
- ▶ 8 out of top 10 are **one pass** algorithms.
- ▶ Out of the top 5 algorithms, **2** uses **PC**.
- ▶ **LRPC**, **LRPH** are not in top 10.

1. RemSP
2. RemPS
3. **MS-IPC-LRPC**
4. MS-LSPS
5. **MS-IPC-LSPC**
6. MS-LRPS
7. MS-IPC-LRPS
8. MS-LSCO
9. MS-LRCO
10. MS-IPC-LSPS

# Related Experimental Studies: Improvement

Reference	# of Algorithms	Recommended Algorithm	RemSP improves by
[Liu, 1990]	2	NLPC	56%
[Gilbert et al., 1994]	6	NLPH	45%
[Wassenberg et al., 2008]	8	LRCO	24%
[Wu et al., 2009]	3	LIPC	48%
[Hynes, 1998]	18	LICO, LSCO	28%, 24%
[Osipov et al., 2009]	2	IPC-LRPC	29%
-	-	LRPC	52%
-	-	LRPH	28%

## Future Works

- ▶ Extend to other application areas.
- ▶ Consider arbitrary sequences of intermixed MAKESET, UNION, and FIND operations.
- ▶ More formal profiling including cache misses, pointer jumps, number of comparisons etc.

Thank you.

# Best Enhanced Classical Algorithm

- ▶ Best enhanced classical algorithm is MS-IPC-LRPC.
- ▶ **RemSP** improved over MS-IPC-LRPC : **12%**  
**(-3%-18%)**

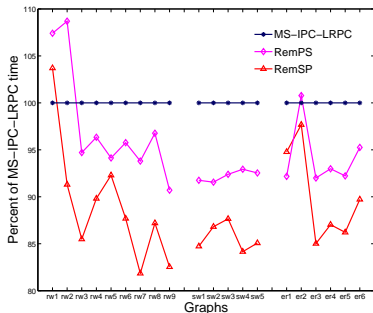


Figure: Improvement over MS-IPC-LRPC.

# UNION Techniques : NAIVE-LINK (NL)

- ▶ **Arbitrarily** choose one root to point to the other one.
- ▶ Height of the tree could be  $O(n)$ .

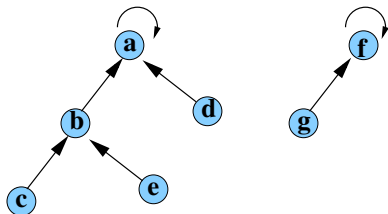


Figure: UNION.

# UNION Techniques : NAIVE-LINK (NL)

- ▶ **Arbitrarily** choose one root to point to the other one.
- ▶ Height of the tree could be  $O(n)$ .

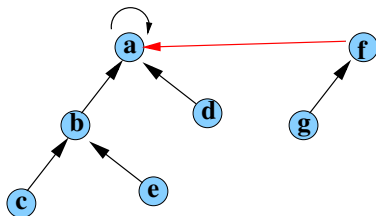


Figure: UNION.

## UNION Techniques : NAIVE-LINK (NL)

- ▶ **Arbitrarily** choose one root to point to the other one.
- ▶ Height of the tree could be  $O(n)$ .

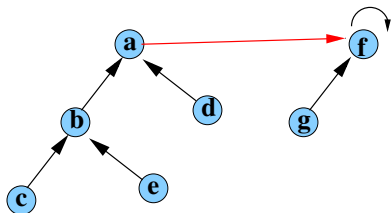


Figure: UNION.

## UNION Techniques : LINK-BY-SIZE (LS)

- ▶ Root of tree with fewer nodes points to the other root.
- ▶ To implement LS efficiently, size of each tree is maintained in the root.

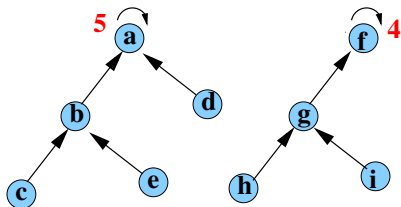


Figure: UNION.

## UNION Techniques : LINK-BY-SIZE (LS)

- ▶ Root of tree with fewer nodes points to the other root.
- ▶ To implement LS efficiently, size of each tree is maintained in the root.

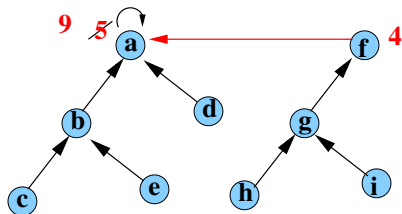


Figure: UNION.

## UNION Techniques : LINK-BY-SIZE (LS)

- ▶ Root of tree with fewer nodes points to the other root.
- ▶ To implement LS efficiently, size of each tree is maintained in the root.

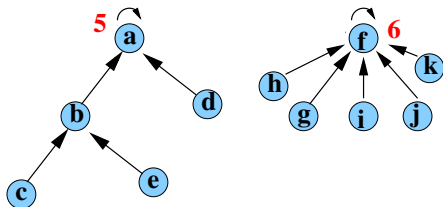


Figure: UNION.

## UNION Techniques : LINK-BY-SIZE (LS)

- ▶ Root of tree with fewer nodes points to the other root.
- ▶ To implement LS efficiently, size of each tree is maintained in the root.

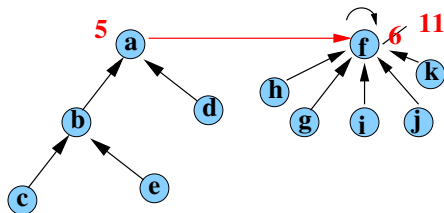


Figure: UNION.

# How much improvement

- ▶ **RemSP** substantially outperforms **LRPC** even though theoretically inferior.
- ▶ **LRPC**  $\Rightarrow$  real world, algorithm courses, libraries.
- ▶ **RemSP** improved over **LRPC**: 52% (38%-66%)
- ▶ **RemSP** improved over **LRPH**: 28% (15%-45%)

# Relative Performance of the Classical Algorithms ▸ Back

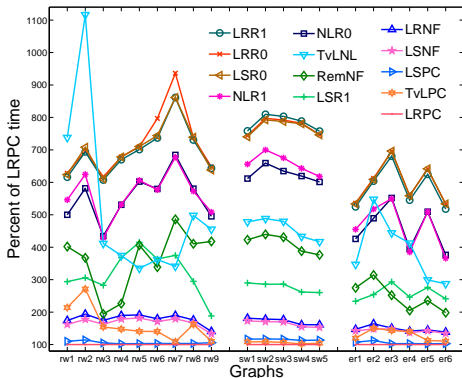


Figure: LRPC dominates 14 algorithms.

# Relative Performance of the Classical Algorithms ▸ Back

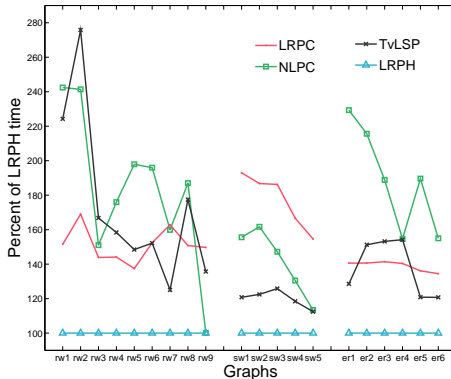


Figure: LRPH dominates 3 additional algorithms, including LRPC

# Relative Performance of the Classical Algorithms ▸ Back

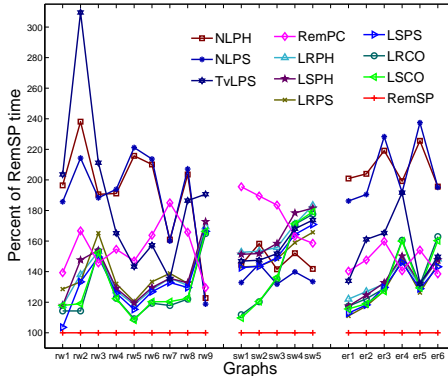


Figure: RemSP dominates 10 remaining algorithms.

# Relative Performance of the Classical Algorithms ▸ Back

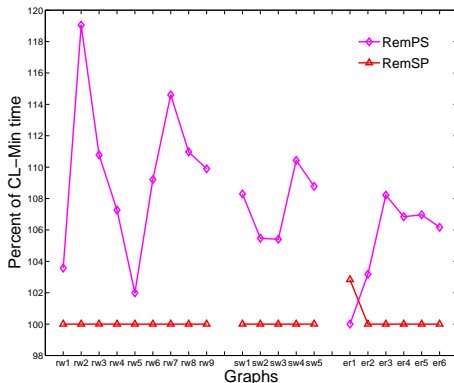


Figure: Only 2 algorithms are undominated.

# Enhancement 1: IPC Relative Performance [▶ Back](#)

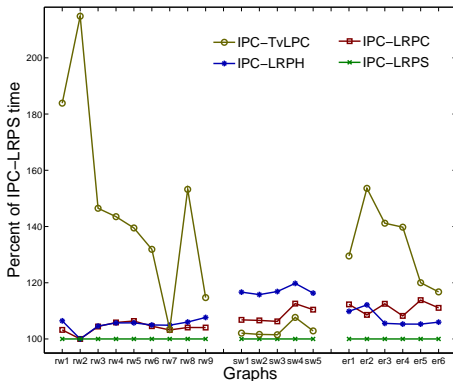


Figure: **IPC-LRPS** dominates **3** algorithms.

# Enhancement 1: IPC Relative Performance [Back](#)

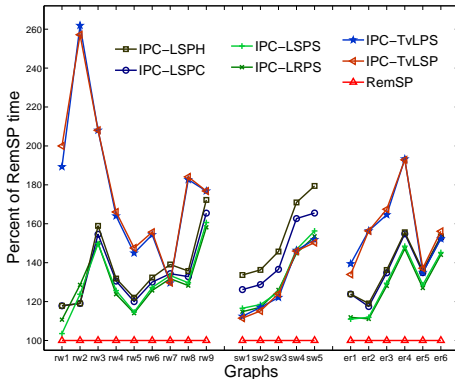


Figure: RemSP dominates 6 remaining algorithms, including IPC-LRPS

# Enhancement 2: INT Relative Performance ▶ INT

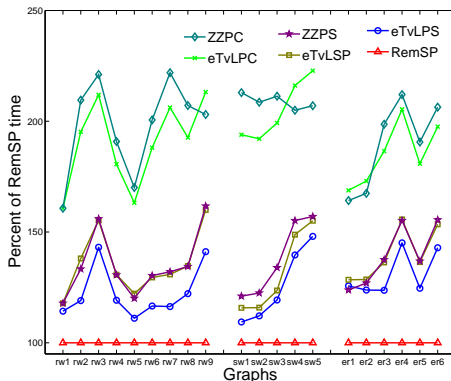


Figure: RemSP dominates all the 5 variations.

# Enhancement 3: MS Relative Performance ▶ MS

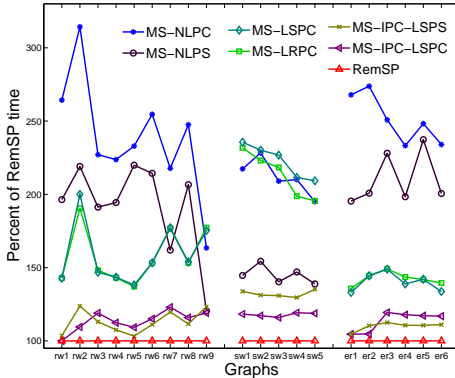


Figure: RemSP dominates 6 algorithms.

# The Fastest Algorithms ▶ [Back](#)

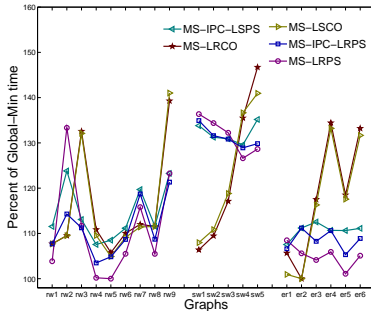
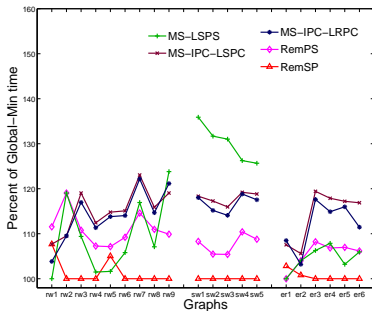
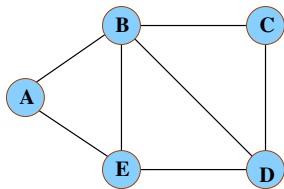
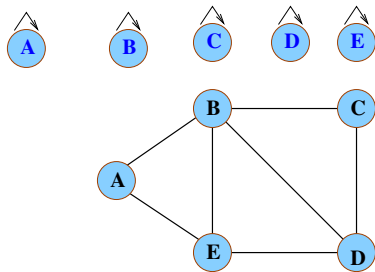


Figure: Top 10 algorithms.

## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)

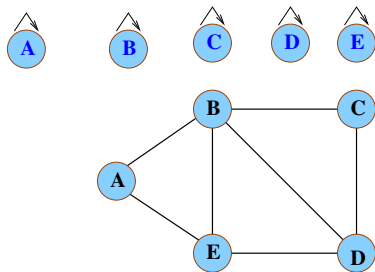


## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)



## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)

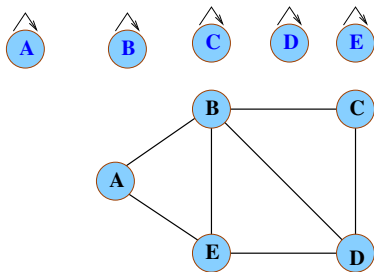
▶ (C, D)



$S = \{ \text{emptyset} \}$

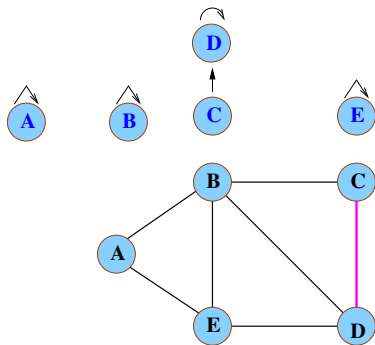
# Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)

▶  $(C, D) \Rightarrow \text{FIND}(C), \text{FIND}(D)$



$S = \{ \text{emptyset} \}$

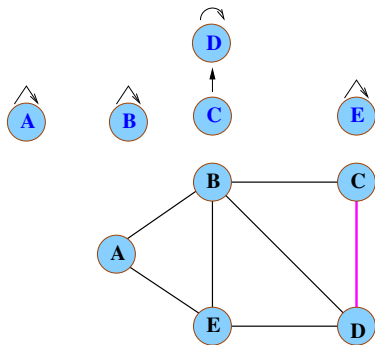
## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)



▶  $(C, D) \Rightarrow \text{UNION}(C, D)$

$S = \{(C, D)\}$

## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)

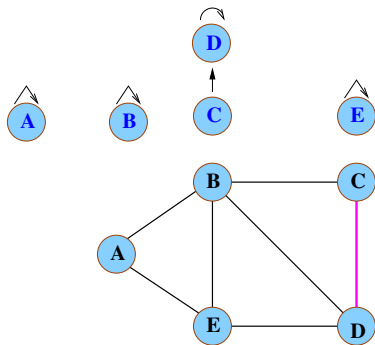


▶ (C, D)

▶ (B, C)

$S = \{(C, D)\}$

## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)

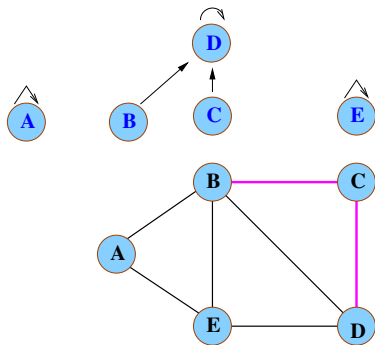


▶ (C, D)

▶ (B, C)  $\Rightarrow$  FIND(B), FIND(C)

$S = \{(C, D)\}$

## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)

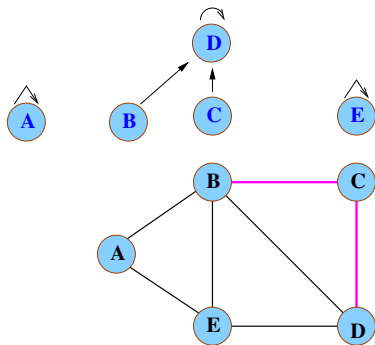


▶ (C, D)

▶ (B, C)  $\Rightarrow$  UNION(B, C)

$S = \{(C, D), (B, C)\}$

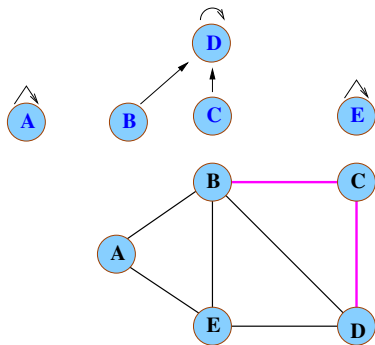
## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)



- ▶ (C, D)
- ▶ (B, C)
- ▶ (B, D)

$$S = \{(C, D), (B, C)\}$$

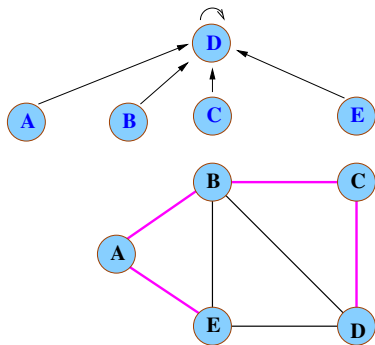
## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)



- ▶ (C, D)
- ▶ (B, C)
- ▶ (B, D)  $\Rightarrow$  FIND(B), FIND(D)

$$S = \{(C, D), (B, C)\}$$




## Use of UNION-FIND for Computing Connected Components: Example: [▶ Back](#)






- ▶ (C, D)
- ▶ (B, C)
- ▶ (B, D)
- ▶ (A, B)
- ▶ (A, E)
- ▶ (B, E)
- ▶ (D, E)

$$S = \{(C, D), (B, C), (A, B), (A, E)\}$$




# Bibliography I

-  Dijkstra, E. W. (1976).  
*A Discipline of Programming*.  
Prentice-Hall.
-  Galil, Z. and Italiano, G. F. (1991).  
Data structures and algorithms for disjoint set union problems.  
*ACM Comput. Surv.*, 23(3):319–344.
-  Gilbert, J. R., Ng, E. G., and Peyton, B. W. (1994).  
An efficient algorithm to compute row and column counts for  
sparse Cholesky factorization.  
*SIAM J. Matrix Anal. Appl.*, 15(4):1075–1091.

## Bibliography II

-  Hynes, R. (1998).  
A new class of set union algorithms.  
Master's thesis, Department of Computer Science, University of Toronto, Canada.
-  Liu, J. W. H. (1990).  
The role of elimination trees in sparse factorization.  
*SIAM J. Matrix Anal. Appl.*, 11:134–172.
-  Manne, F. and Patwary, M. M. A. (2009).  
A scalable parallel union-find algorithm for distributed memory computers.  
To appear in the proceedings of PPAM 2009, LNCS.

## Bibliography III

-  Osipov, V., Sanders, P., and Singler, J. (2009).  
The filter-Kruskal minimum spanning tree algorithm.  
In *ALLENEX*, pages 52–61.
-  Tarjan, R. E. and van Leeuwen, J. (1984).  
Worst-case analysis of set union algorithms.  
*J. ACM*, 31(2):245–281.
-  Wassenberg, J., Bulatov, D., Middelman, W., and Sanders, P. (2008).  
Determination of maximally stable extremal regions in large images.  
In *From Proceeding of Signal Processing, Pattern Recognition, and Applications (SPPRA)*. Acta Press.

## Bibliography IV



Wu, K., Otoo, E., and Suzuki, K. (2009).

Optimizing two-pass connected-component labeling algorithms.

*Pattern Anal. Appl.*, 12(2):117–135.

## Use of UNION-FIND for Computing Connected Components:

QUICK-UNION( $x, y$ )[Galil and Italiano, 1991]

```
1:  $S \leftarrow \emptyset$ 
2: for each  $x \in V$  do
3:   MAKESET( $x$ )
4: for each  $(x, y) \in E$  do
5:   if FIND( $x$ )  $\neq$  FIND( $y$ ) then
6:     UNION( $x, y$ )
7:    $S \leftarrow S \cup \{(x, y)\}$ 
```

```
1:  $r_x \leftarrow x, r_y \leftarrow y$ 
2: while  $r_x \neq p(r_x)$  do
3:    $r_x \leftarrow p(r_x)$ 
4: while  $r_y \neq p(r_y)$  do
5:    $r_y \leftarrow p(r_y)$ 
6: if  $r_x \neq r_y$  then
7:    $p(r_x) \leftarrow r_y$ 
8:    $S \leftarrow S \cup \{(x, y)\}$ 
```

## QUICK-UNION [Galil and Italiano, 1991]

```
1:  $S \leftarrow \emptyset$ 
2: for each  $x \in V$  do
3:   MAKESET( $x$ )
4: for each  $(x, y) \in E$  do
5:   if FIND( $x$ )  $\neq$  FIND( $y$ ) then
6:     UNION( $x, y$ )
7:      $S \leftarrow S \cup \{(x, y)\}$ 
```

- ▶ Store the results of the two FIND operations.
- ▶ Use them as input to the UNION operation.
- ▶ **Speed up** the algorithm  $\rightarrow$  QUICK-UNION.
- ▶ Stop when  $|S| = (|V| - 1)$ , indicates no more edges will be added to  $S$ .

# COMPRESSION Techniques : NAIVE-FIND (NF)

- ▶ FIND operation with  
no compression.
- ▶ FIND is time  
consuming.

# COMPRESSION Techniques : PATH-SPLITTING (PS)

- ▶ Set the parent pointers of all nodes in the find-path to its **grandparent**.
- ▶ Traverse the find-path **once**.

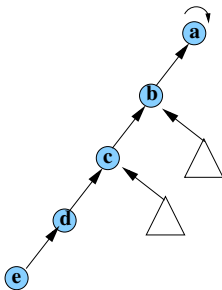


Figure: FIND(*e*) with PS

# COMPRESSION Techniques : PATH-SPLITTING (PS)

- ▶ Set the parent pointers of all nodes in the find-path to its **grandparent**.
- ▶ Traverse the find-path **once**.

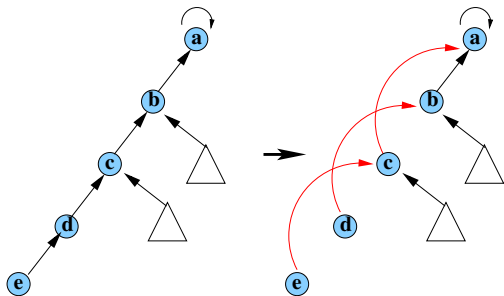


Figure: FIND(e) with PS

# COMPRESSION Techniques : R0

- ▶ All nodes except the last  $k$  nodes on the find-path point to the first node.
- ▶ Need to traverse the find-path **once**.

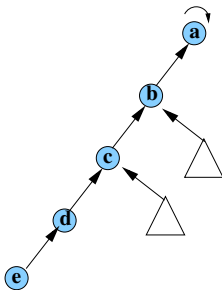


Figure: FIND( $e$ ) with R0

# COMPRESSION Techniques : R0

- ▶ All nodes except the last  $k$  nodes on the find-path point to the first node.
- ▶ Need to traverse the find-path **once**.

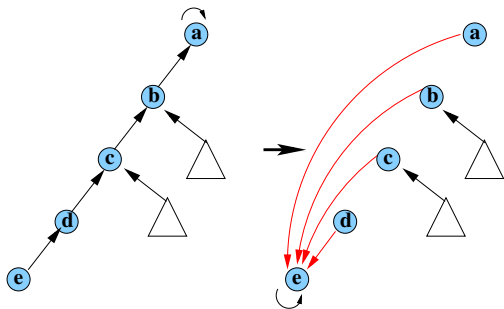


Figure: FIND( $e$ ) with R0

# COMPRESSION Techniques : R1

- ▶ All nodes except the last  $k$  nodes on the find-path point to the first node.
- ▶ Need to traverse the find-path **once**.

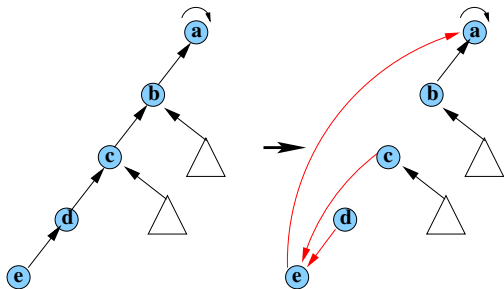


Figure: FIND( $e$ ) with R1

## Algorithm Count

- ▶ The running time of **RemSP** is  $O(m \log_{(2+m/n)} n)$ .
- ▶ The running time of **TVLSP** is  $O(m \cdot \alpha(m, n))$ .
- ▶  $\{\text{Rem}, \text{TVL}\} \times \{\text{NL}, \text{PC}, \text{PS}, \text{SP}\}$  but **not with**  $\{\text{PH}, \text{R0}, \text{R1}\}$ .
- ▶ **8** more algorithms.
- ▶ Total algorithm:  $21 + 8 = \mathbf{29}$  algorithms SO FAR.

## Test Sets and Experimental Setup...

- ▶ For each graph: **5 runs with 5 different random orderings.**
- ▶ Compute the average of these runtimes.
  - ▶ rw: **25 total runs.**
  - ▶ sw and er: **125 total runs** since each graph has 5 different instances.

## Enhancement 3: Memory Efficient ( $MS$ ) Algorithms

- ▶ Reduce memory used by each algorithm.
- ▶ Each node has:
  - ▶ a parent pointer
  - ▶ either a size or rank for some algorithms
  - ▶ a sibling pointer for CO algorithm.
- ▶ Thus have one to three fields in each record.

## Enhancement 3: Memory Efficient (MS) Algorithms

- ▶ **Save one field** by coding the size or rank of the root into the parent pointer.
- ▶  $\{\text{MS}\} \times \{\text{NL}\} \times \{\text{PC}, \text{PS}\}$ .
- ▶  $\{\text{MS}\} \times \{\text{LR}, \text{LS}\} \times \{\text{PC}, \text{PS}, \text{CO}\}$ .
- ▶  $\{\text{MS}\} \times \{\text{IPC}\} \times \{\text{LR}, \text{LS}\} \times \{\text{PC}, \text{PS}\}$ .
- ▶ **12** more variations.
- ▶ Total algorithm:  $43 + 12 = 55$  variations.
- ▶ Note that **Rem** does not use size or rank, so it is automatically an **MS** algorithm.