Tutorial on Incremental Computation

Thomas W. Reps
University of Wisconsin

Twentieth Annual ACM Symposium on
Principles of Programming Languages

Charleston, South Carolina
January 10, 1993

Respond Well to Small Changes

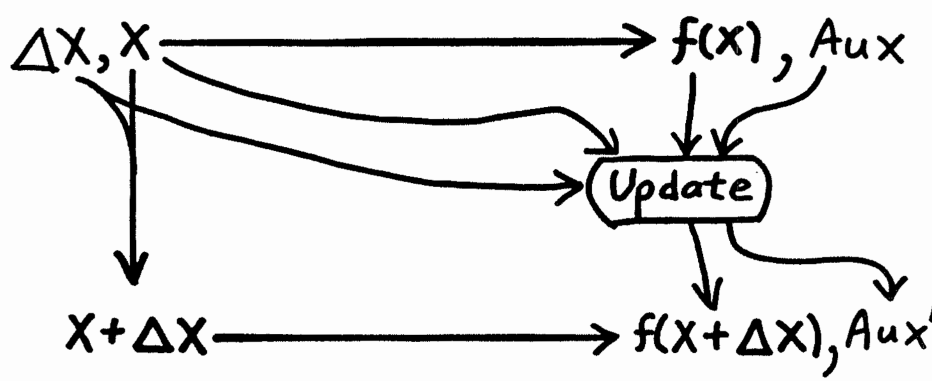
- Batch-mode systems
  - Modify document; rerun LaTeX
  - Modify source code; recompile/relink
- Reactive-mode systems
  - WYSIWYG editors
  - spreadsheets

Incremental Computation

x: input "data"

$f(x)$: result of computation on x

Problem: Given a modification $x \rightarrow x + \Delta x$, compute $f(x + \Delta x)$.



An Incremental-Computation Checklist

- How does the computation's state relate to the state of the batch computation?
- Does the computation exploit
  - independence?
  - quiescence?
  - balancing?
- What kind of auxiliary or summary information does the computation use?
- Under what circumstances is it cheaper to recompute from scratch?
- What criterion (or criteria) demonstrates the merits of the method?
- Generality of the method

Text Formatting

```
Research has----
shown that candy
is dandy but----
liquor is-------
quicker---------
```

$x = [8, 3, 5, 4, 5, 2, 5, 3, 6, 2, 7]$
$y = f(x) = [8, 12, 5, 10, 16, 2, 8, 12, 6, 9, 7]$

$$
\begin{aligned}
f:\ & y[1] = x[1] \\
& y[i] = \text{let } v = y[i-1] + 1 + x[i] \text{ in} \\
& \qquad\qquad \text{if } v > 16 \text{ then } x[i] \text{ else } v \text{ fi} \\
& \qquad\quad \text{end}
\end{aligned}
$$

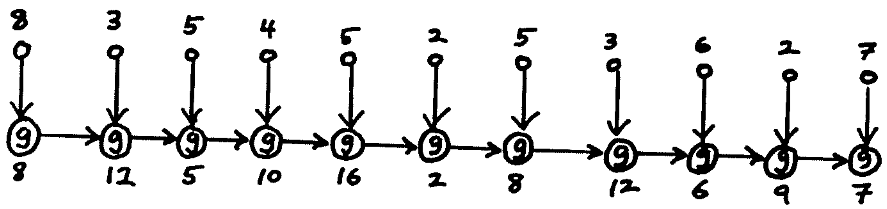
```
Research has----
shown that------
salads are dandy
but liquor is---
quicker---------
```

$x = [8, 3, 5, 4, 6, 3, 5, 3, 6, 2, 7]$
$y = [8, 12, 5, 10, 6, 10, 16, 3, 10, 13, 7]$

Independence: Research has shown that

Quiescence: quicker

Dependence graph:



$g(\alpha,\beta) =$ <u>let</u> $v = \beta + 1 + \alpha$ <u>in</u>
<u>if</u> $v > 16$ <u>then</u> α <u>else</u> v <u>fi</u>
<u>end</u>

Summing a List of Numbers

$X = [18, 20, 45, 6, 3, 81, 15, 17]$

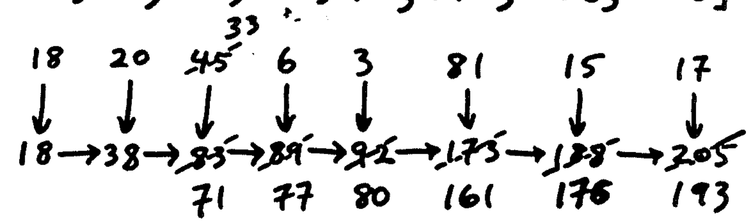
$f(X) = 205$

$X' = [18, 20, 33, 6, 3, 81, 15, 17]$

$f(X') = f(X) + (X'[3] - X[3]) = 205 - 12 = 193$

Summary information

$p = [18, 38, 83, 89, 92, 173, 188, 205]$



Summing a List of Numbers

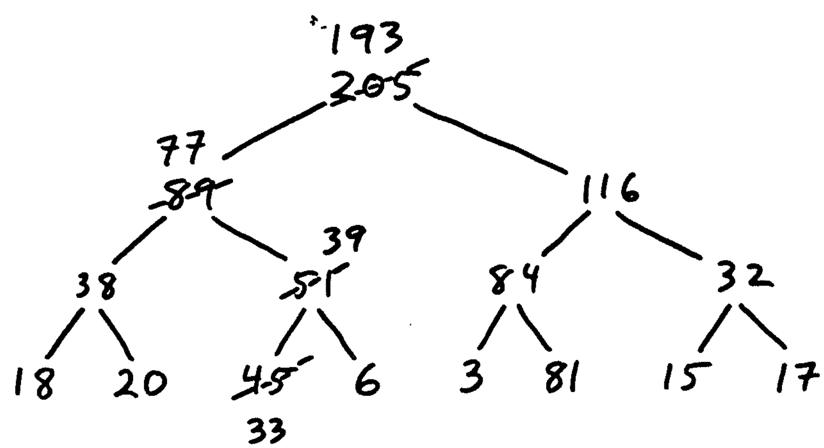
$X = [18, 20, 45, 6, 3, 81, 15, 17]$

$f(X) = 205$

$X' = [18, 20, 33, 6, 3, 81, 15, 17]$

$f(x') = f(x) + (x'[3] - X[3]) = 205 - 12 = 193$

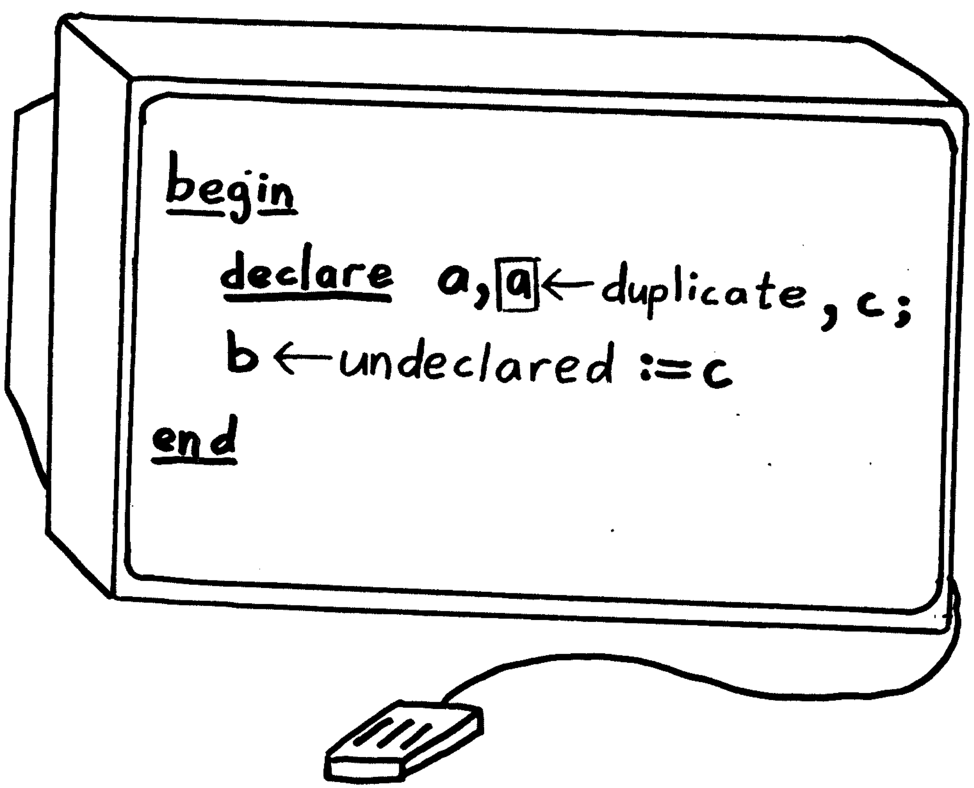
Balancing



PL Contexts for Incremental Computation

- Language support for interactive systems (“reactive-mode” systems)
- Incremental language-processing algorithms (e.g., interactive programming tools, compilation reanalysis, etc.)
- Paradigm for program optimization
- Support for tool integration via control-integration paradigm
- Implementations of compilers and tools that support the above ideas

Static Inference: Name Analysis



Optimization

Strength reduction [Cocke]

```
k:=0
i:=1
while i≤N do
   k:= k+a[2*i]
   i:= i+1
od
```

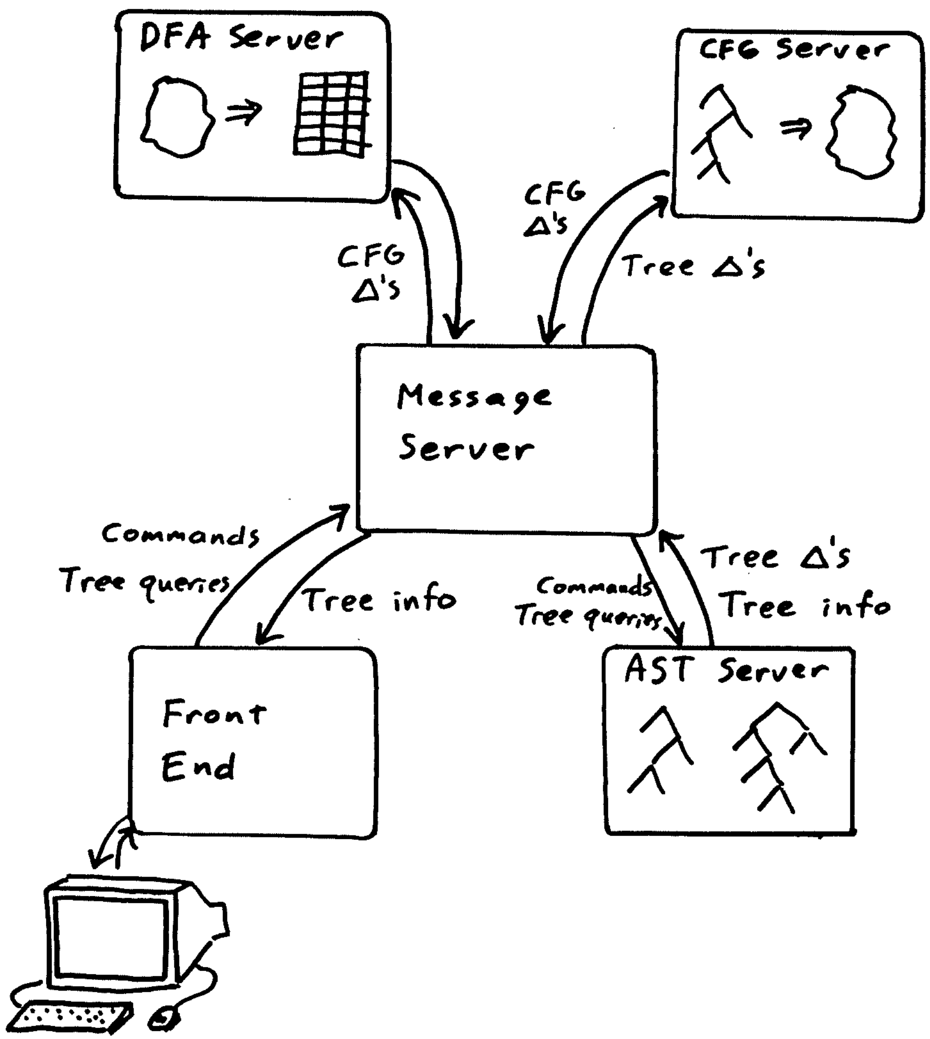
```
k:=0
i:=1
twoi := 2*i
while i≤N do
   k:= k+a[twoi]
   i:= i+1
   twoi := twoi + 2
od
```

Finite differencing [Earley, Fong & Ullman, Paige]

```
S:= ∅
forever do
   E:= {s∈S | even(s)}
   if |E|=N then break
   x:= arb T
   T:= T-{x}
   S:= S ∪ {x}
od
```

```
S:= ∅
F:= {s∈S | even(s)}
forever do
   E:= F
   if |E|=N then break
   x:= arb T
   T:= T-{x}
   S:= S ∪ {x}
   F:= F ∪ if even(x) then {x}
                    else ∅
od
```

Tool Integration via Control Integration



Goals of Talk

- Ways of assessing incremental algorithms
- General principles

  (such as they exist at present)
- Individual results

  (opportunity for new principles?)

Talk Outline

Introduction

Assessment of incremental algorithms
• Assessing the cost of a single update operation • Comparison over a sequence of update operations • Hierarchies of incremental problems • Empirical studies

Graph-annotation problems

Other update problems

"Incrementalizers"

Conclusions

Worst-Case Analysis

- Analysis of batch algorithms

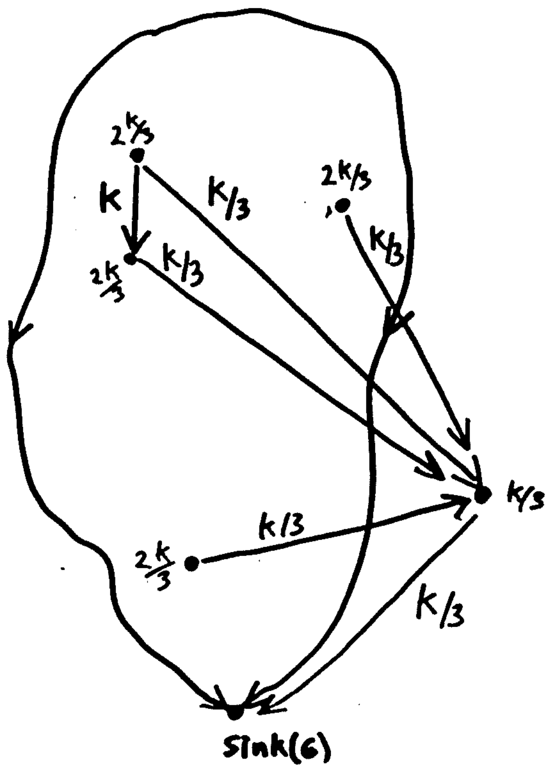
  $\forall x \quad T_{Batch}(x) = O(f(|x|))$

- Analysis of incremental algorithms

  $\forall x, \Delta^-, \Delta^+$

  $$T_{Inc}(x, \Delta^-, \Delta^+) = O(g(|(x - \Delta^-) + \Delta^+|)) \stackrel{?}{=} o(f(|(x - \Delta^-) + \Delta^+|))$$

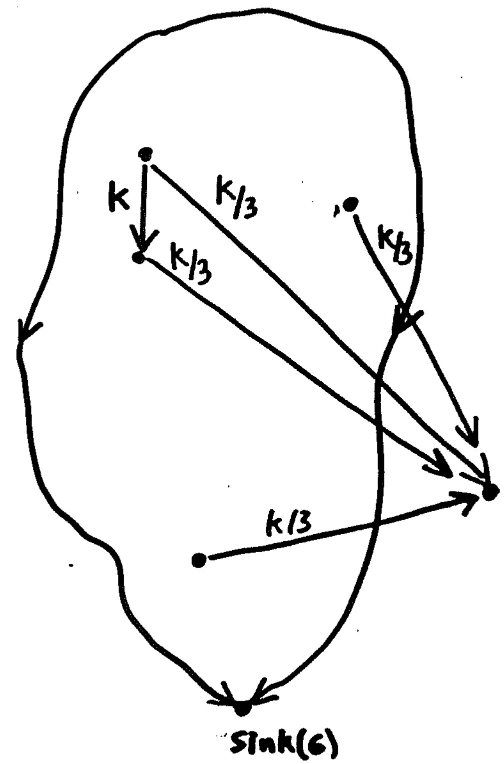
Single-Sink Shortest-Path Problem
(with positive edge weights)



[Spira + Pan 1975]

[Berman, Paull, + Ryder 1990]

Single-Sink Shortest-Path Problem
(with positive edge weights)



[Spira + Pan 1975]

[Berman, Paull, + Ryder 1990]

Worst-Case Analysis (of Incr. Algorithms)

- For many problems, no incremental algorithm can perform better than a single invocation of the best batch algorithm, in the worst case.

? ∴ The batch start-over algorithm is optimal.

? ∴ Worst-case complexity is not a good way to measure the complexity of incremental computation.

∴ Need alternative ways to characterize the performance of incremental algorithms.

Direct Comparison: Incremental vs. Batch

- Asymptotically better
  $\forall x, \Delta^-, \Delta^+$
  $$T_{Inc}(x, \Delta^-, \Delta^+) = o\,(T_{Batch}(\,|(x - \Delta^-) + \Delta^+|\,))$$
  Updating the minimum spanning forest of an n-vertex, m-edge graph [Frederickson 1986]
  $$T_{IncMSF} = O\,(\sqrt{m}) = o\,(m)$$

- Better constant factor
  $\forall x, \Delta^-, \Delta^+$
  $$T_{Inc}(x, \Delta^-, \Delta^+) \le T_{Batch}(\,|(x - \Delta^-) + \Delta^+|\,)$$
  Variety of graph problems [Cheston 1976]

- Never too much worse (and often better)
  $\forall x, \Delta^-, \Delta^+$
  $$T_{Inc}(x, \Delta^-, \Delta^+) = O\,(T_{Batch}(\,|(x - \Delta^-) + \Delta^+|\,))$$
  Bag expressions [Yellin & Strom 1991]

Direct Comparison: Incr. vs. Incr.

- Asymptotically better

  $\forall x, \Delta^-, \Delta^+$

  $$T_{Inc\,1}(x, \Delta^-, \Delta^+) = o\,(T_{Inc\,2}(x, \Delta^-, \Delta^+))$$

- Better constant factor

  $\forall x, \Delta^-, \Delta^+$

  $$T_{Inc\,1}(x, \Delta^-, \Delta^+) \le T_{Inc\,2}(x, \Delta^-, \Delta^+)$$

- Never too much worse (and often better)

  $\forall x, \Delta^-, \Delta^+$

  $$T_{Inc\,1}(x, \Delta^-, \Delta^+) = O\,(T_{Inc\,2}(x, \Delta^-, \Delta^+))$$

Change the Accounting Method

- Worst-case analysis

  $\forall x, \Delta^-, \Delta^+$

  $$T_{Inc}(x, \Delta^-, \Delta^+) = o\,(T_{Batch}(|(x - \Delta^-) + \Delta^+|))$$

- Average-case analysis

  $$T_{Inc}(x, \Delta^-, \Delta^+) = o\,(T_{Batch}(|(x - \Delta^-) + \Delta^+|))$$

- Amortized-cost analysis

  $$T_{Inc}(x, \Delta^-, \Delta^+) = o\,(T_{Batch}(|(x - \Delta^-) + \Delta^+|))$$

Comparison Over a Sequence of Operations

- Amortized-cost analysis

  $$T_{Inc}(x, \Delta^-, \Delta^+) = o\,(T_{Batch}(\,|\,(x - \Delta^-) + \Delta^+\,|\,))$$

- Competitiveness

  $\forall\, x,\ seq$

  $$T_{On-line}(x, seq) \le k * T_{Off-line}(x, seq)$$

Competitive Ratio

Requests:

Let's go skiing

Actions:

Rent : \$ 1

Buy : \$ s

Use skis already purchased: \$ 0

On-line algorithm: $\underbrace{R, R, R \ldots R}_{k}, B, U, U, \ldots, U$

For t requests: $C_{on\text{-}line} = \begin{cases} t & \text{if } t \le k \\ k+s & \text{otherwise} \end{cases}$

$C_{off\text{-}line} = \min(s, t)$

Best for adversary: L^{k+1}, "so-long sucker"

$\Rightarrow R^k$, B, "darn"

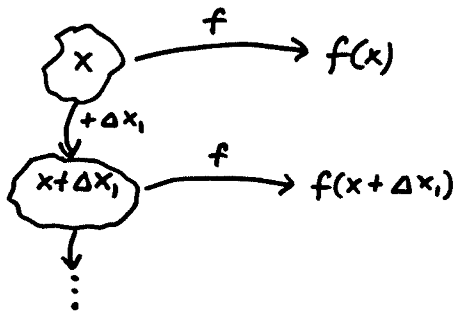
Best for C.R.: choose k to minimize

$$\frac{C_{on\text{-}line}}{C_{off\text{-}line}} = \frac{k+s}{\min(s, k+1)} \Rightarrow k = s-1$$

Competitive Ratio: $\frac{2s-1}{s}$

Incremental Computation vs. On-Line Computation

I.C. -- functional view



- I.C. as O.L.C.
  - One operation: Modify(δ)
  - One query: What are the changes in output?
  - MQ MQ MQ ...
- Complementary views
  - Functional view suggests some specific techniques (e.g. function caching)
  - OLC suggests broadening the problem
    MMMQ MQ MMMMQ ...
    [Cohen & Tamassia - SODA 91]

Boundedness

$$T_{Inc}(x, \Delta^-, \Delta^+) = O(g\,(\text{adaptive parameter}))$$

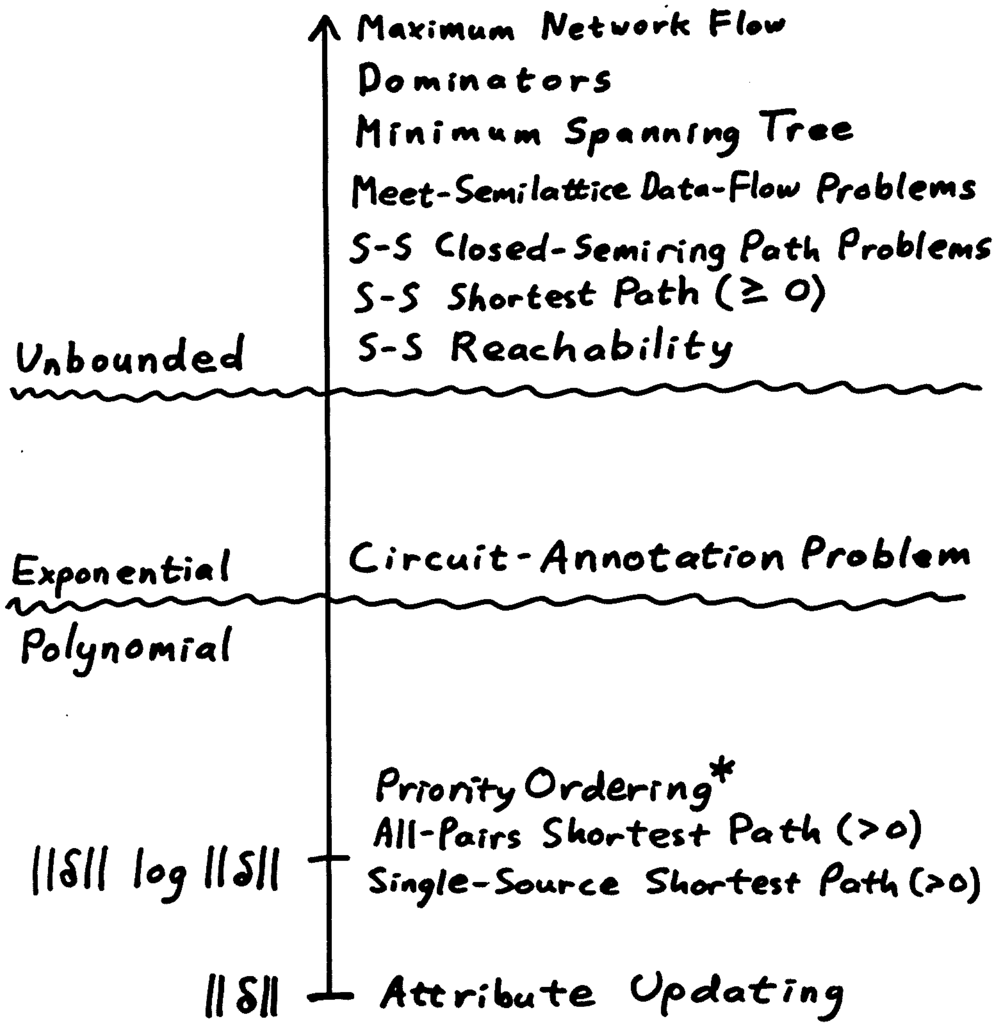
$$O(f(|\text{input}|))$$

versus

$$O(g(|\Delta\text{input}| + |\Delta\text{output}|))$$

Adaptive: $|\Delta\text{input}| \leftrightarrow |\Delta\text{input}| + |\text{entire output}|$

$\|\delta\| =_{df} |\Delta\text{input}| + |\Delta\text{output}|$



Incremental Relative Lower Bounds (IRLB)

1/(# modifications needed to "achieve" a batch evaluation)

e.g. IRLB (SSSP>0) = O(1)

IRLB * Batch lower bound = Incr. lower bound

Sorting:

$O(1/n) * \Omega(n \log n) = \Omega(\log n)$

SSSP>0:

$O(1) * \Omega(?) = \Omega(?)$

Requirement: "fast initialization"

e.g. sorting: empty sequence

⇒ limited initial auxiliary storage

[Berman, Paull, Ryder 1990]

IRLB Classification Hierarchy

O(1):

- SSSP > 0
- APSP > 0
- Transitive closure
- Planarity
- Strong connectivity
- Min-Max edge weight path
- Reaching definitions
- Available expressions
- Live uses of variables
- Dominators

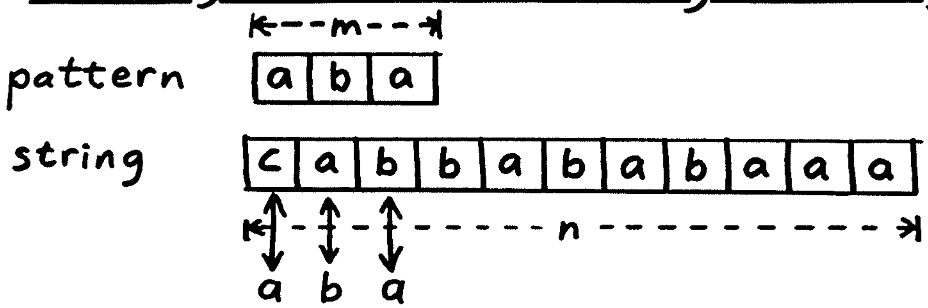
$1/\sqrt{n} \geq \;, \geq 1/n$:

- Connected components
- Biconnected components
- Minimum spanning tree
- Shortest path in undirected graph

IRLB vs. Boundedness

Problem	IRLB	$f(\|\delta\|)$
SSSP>0	O(1) ☹	$O(\|\delta\| \log \|\delta\|)$ ☺
APSP>0	O(1) ☹	$O(\|\delta\| \log \|\delta\|)$ ☺
Dominators	O(1) ☹	Unbounded ☹
Minimum spanning tree	$\frac{1}{\sqrt{n}} \geq \,, \geq \frac{1}{n}$ ☺	Unbounded ☹

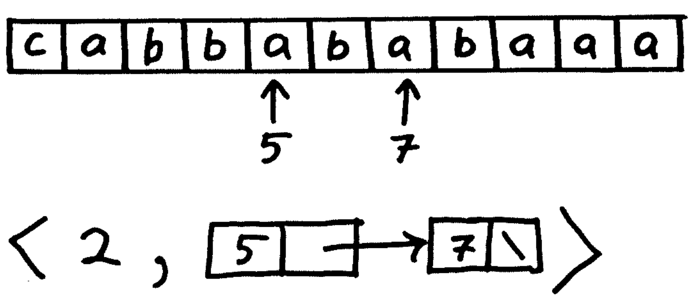
Beating an IRLB: String Matching



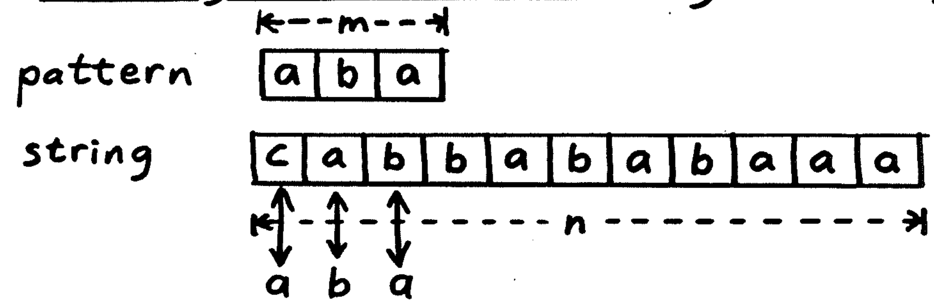
$$IRLB = 1/m$$

$$IRLB * BLB = ILB$$

$$1/m * \Omega(n) = \Omega(n/m)$$



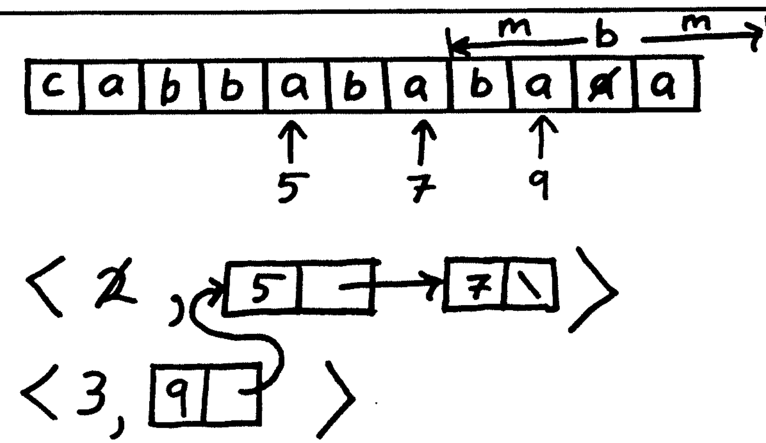
Beating an IRLB: String Matching



$$IRLB = 1/m$$

$$IRLB * BLB = ILB$$

$$1/m * \Omega(n) = \Omega(n/m)$$



$O(m)$ beats IRLB for problems with $m<\sqrt{n}$

Empirical Studies

- Very few studies
- How are modifications generated?
- What benchmarks are used?
- Compare with studies of parallel algorithms
  - standard benchmarks
  - speedup
  - efficiency
  - increase in problem size solvable

Talk Outline

Introduction

Assessment of incremental algorithms

Graph-annotation problems
• Boundedness • Problems on acyclic graphs • Problems on graphs with cycles • Unboundedness

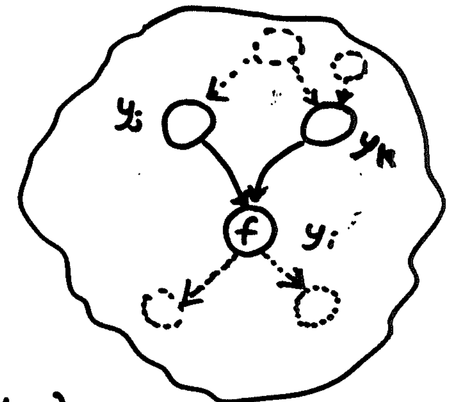
Other update problems

"Incrementalizers"

Conclusions

Graph-Annotation Problems

$y_i = f(y_j, y_k)$



- Dag problems (acyclic)
  - Attribute grammars
  - Circuit-annotation problem
- Problems on cyclic graphs
  - Reachability
  - Shortest-path problem
  - Data-flow analysis problems

Selective Recomputation

Item	Price	Quantity	Total
pen	.95	3	2.85
paper	1.50	2	3.00
Total			5.85

$$C_{pen,\ price} \times C_{pen,\ quantity} = C_{pen,\ total}$$

$$C_{paper,\ price} \times C_{paper,\ quantity} = C_{paper,\ total}$$

$$C_{pen,\ total} + C_{paper,\ total} = C_{total,\ total}$$

Selective Recomputation

Item	Price	Quantity	Total	
pen	~~.95~~ .75	3	~~2.85~~	2.25
paper	1.50	2	3.00	
Total			~~5.85~~	5.25

$$\boxed{C_{pen,\ price} \times C_{pen,\ quantity} = C_{pen,\ total}}$$

$$C_{paper,\ price} \times C_{paper,\ quantity} = C_{paper,\ total}$$

$$\boxed{C_{pen,\ total} + C_{paper,\ total} = C_{total,\ total}}$$

Differential Updating

Item	Price	Quant.	Total
pen	.75 (=.95 − .20)	3	2.25 (=2.85 − .60)
paper	1.50	2	3.00
Total			5.25 (=5.85 − .60)

$$C^{new}_{pen,\ price} - C^{old}_{pen,\ price} = \Delta C_{pen,\ price}$$

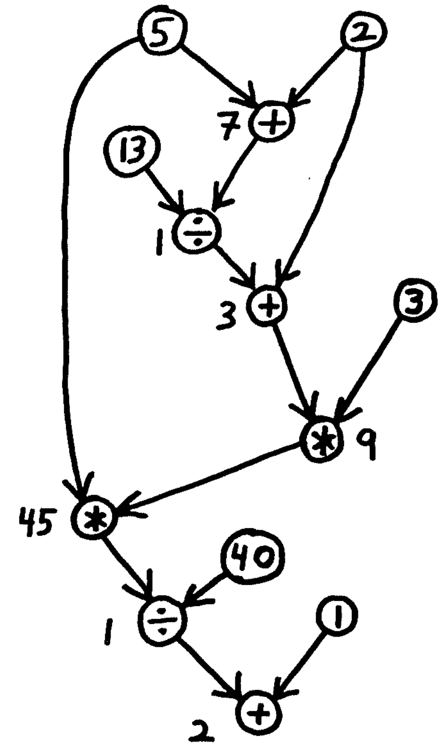
$$\Delta C_{pen,\ price} \times C_{pen,\ quantity} = \Delta C_{pen,\ total}$$

$$\Delta C_{pen,\ total} = \Delta C_{total,\ total}$$

$$C_{pen,\ total} := C_{pen,\ total} + \Delta C_{pen,\ total}$$

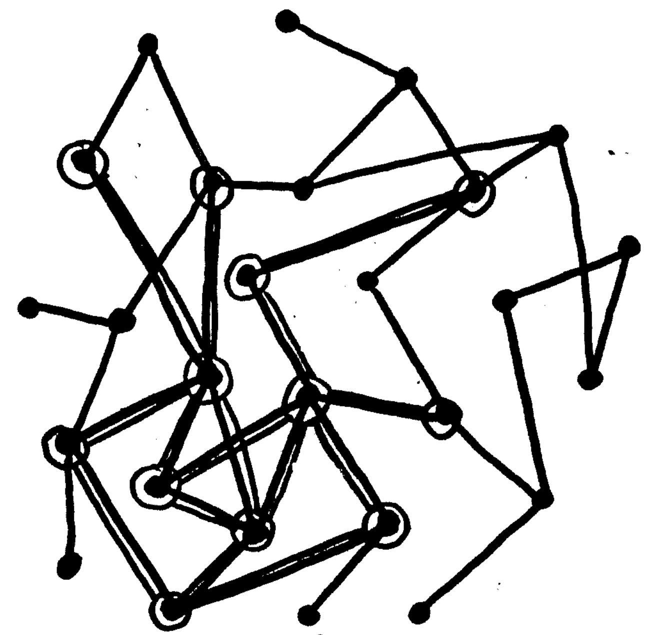
$$C_{total,\ total} := C_{total,\ total} + \Delta C_{total,\ total}$$

Circuit - Annotation Problem



"Size" of a Change

- $\text{MODIFIED}_{G,\delta}$
- $\text{AFFECTED}_{G,\delta}$
  - Not known a priori
- $\text{CHANGED}_{G,\delta} =_{df} \text{MODIFIED}_{G,\delta} \cup \text{AFFECTED}_{G,\delta}$
- Defn: $\|\delta\| = \|\text{CHANGED}_{G,\delta}\|_{G+\delta}$
  - characterizes updating costs inherent to a problem (rather than costs of a given algorithm for the problem
- Goal: $O(f(\|\delta\|))$



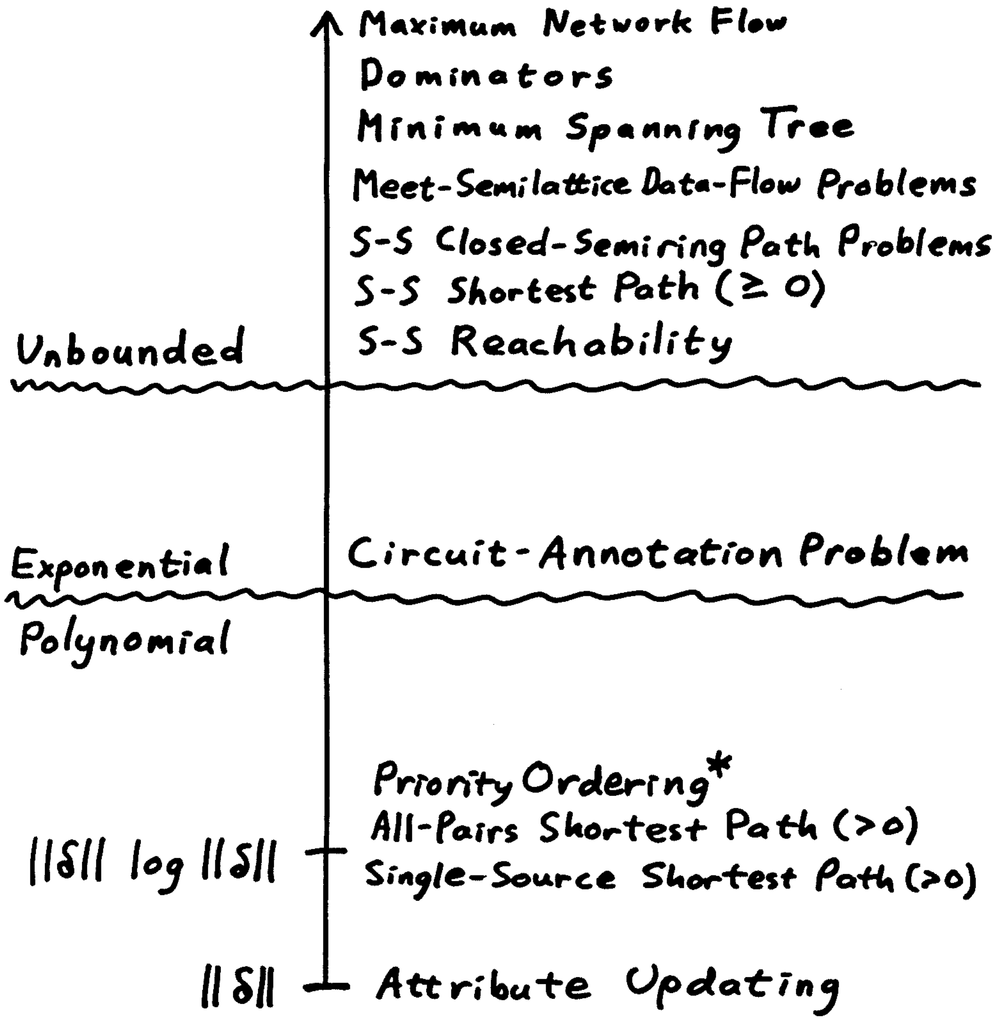
K : vertex set $|K| = 5$

$N(K)$: neighborhood of K $|N(K)| = 12$

$\langle K \rangle$: induced graph $|\langle K \rangle| = 11$

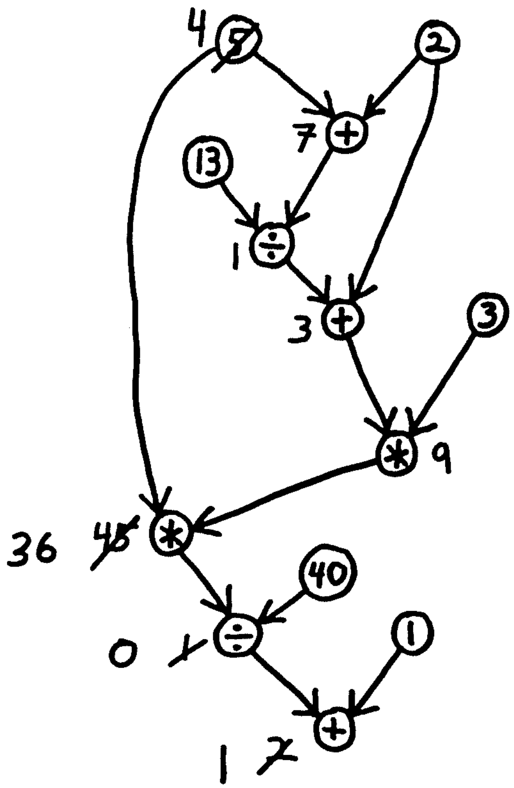
$\langle N(K) \rangle$: induced graph $|\langle N(K) \rangle| = 27$

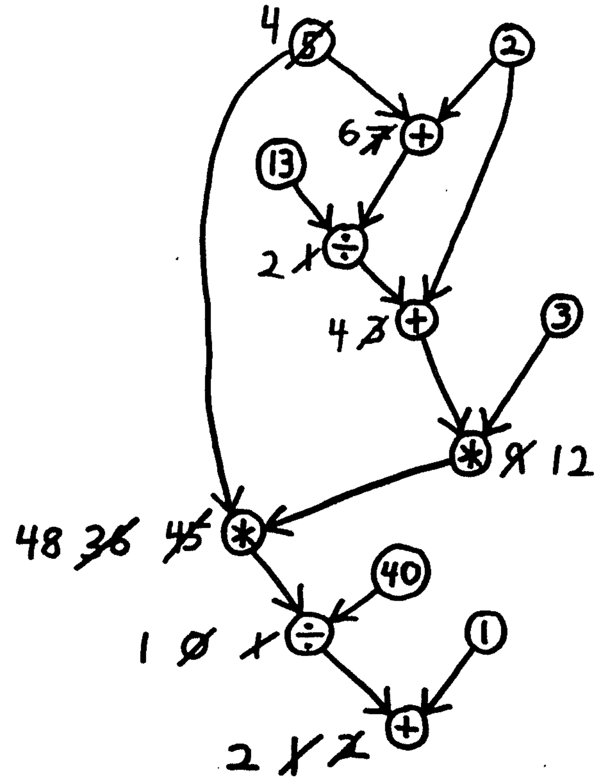
$\|K\| =_{df} |\langle N(K) \rangle|$ "extended size"

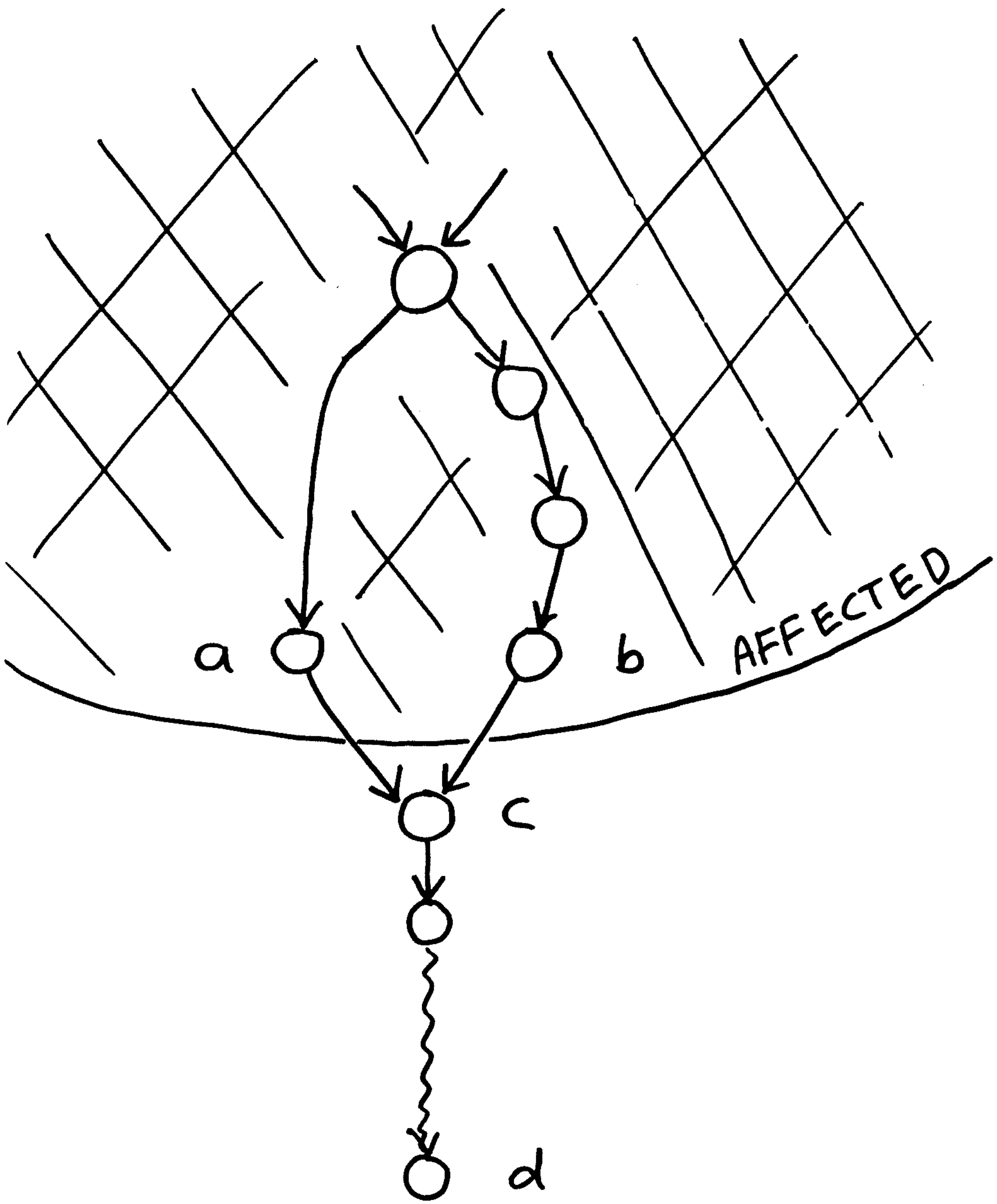


(Naive) Change Propagation

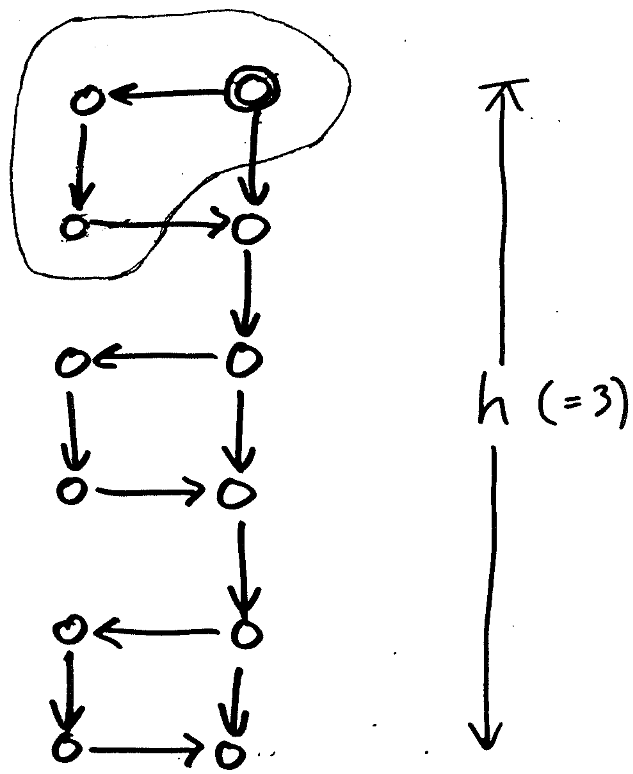
```
Propagate ( G, S )
precondition:  S = {inconsistent nodes of G}
begin
    while  S ≠ ∅ do
        Select and remove a node v from S
        oldvalue := val[V]
        Reevaluate v
        if oldvalue ≠ val[V] then
            S := S ∪ {successors of v}
        fi
    od
end
```


4 5
2
7
13
1
3
3
9
36 45
40
0 1
1
1 7


4 5
2
6 7
13
2 1
4 3
3
9 12
48 36 45
40
1 0 1
1
2 1 7

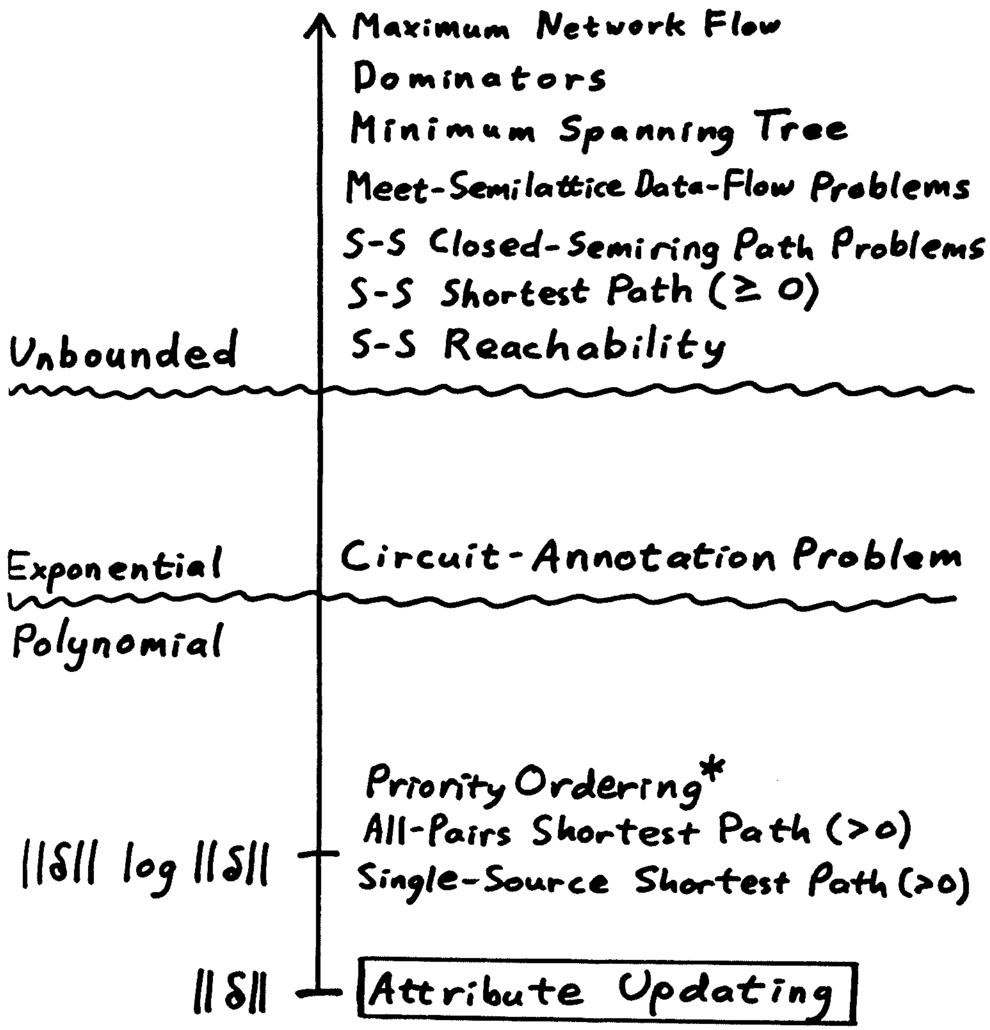


Behavior of Change Propagation



$$T(h) = 2T(h-1) + k$$

$$T(h) = O(2^h)$$



Equations for Name Analysis

begin
  declare a, b, c;
  b := c
end

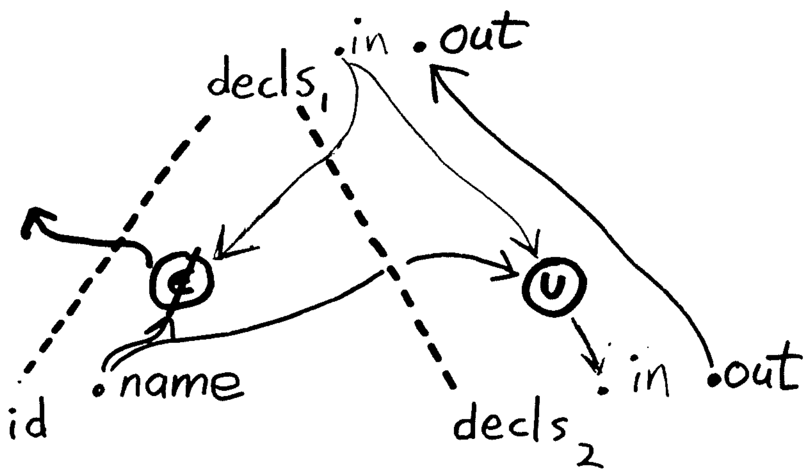
$\text{DeclaredIn}_{\text{Decls}} = \{a\} \cup \{b\} \cup \{c\}$

$\text{DeclaredFor}_{\text{Stmt}} = \text{DeclaredIn}_{\text{Decls}}$

$b \in \text{DeclaredFor}_{\text{Stmt}}$

$c \in \text{DeclaredFor}_{\text{Stmt}}$

Defining Static Inferences



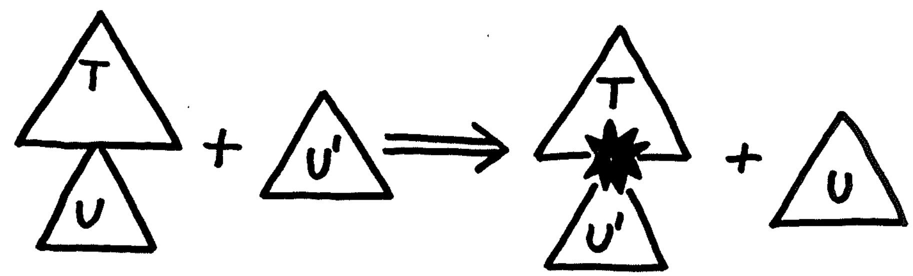
$decls_2.in = \{id.name\} \cup decls_1.in$

$decls_1.out = decls_2.out$

require $id.name \notin decls_1.in$

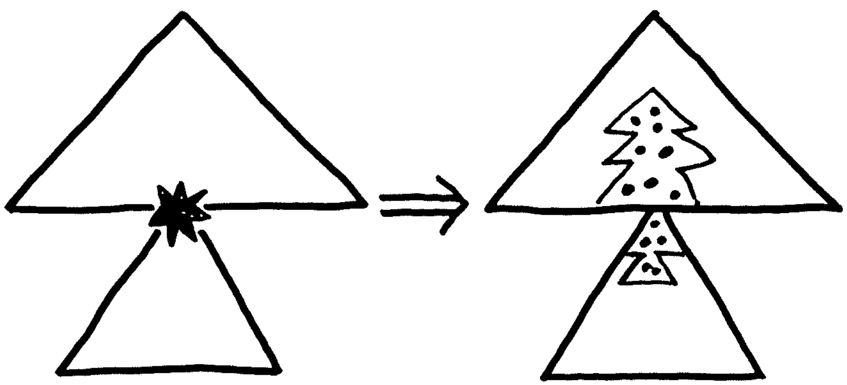
Attribute grammar [Knuth]

Subtree Replacement



Consistent Tree $\Longrightarrow$ Inconsistent Tree

Optimal Updating

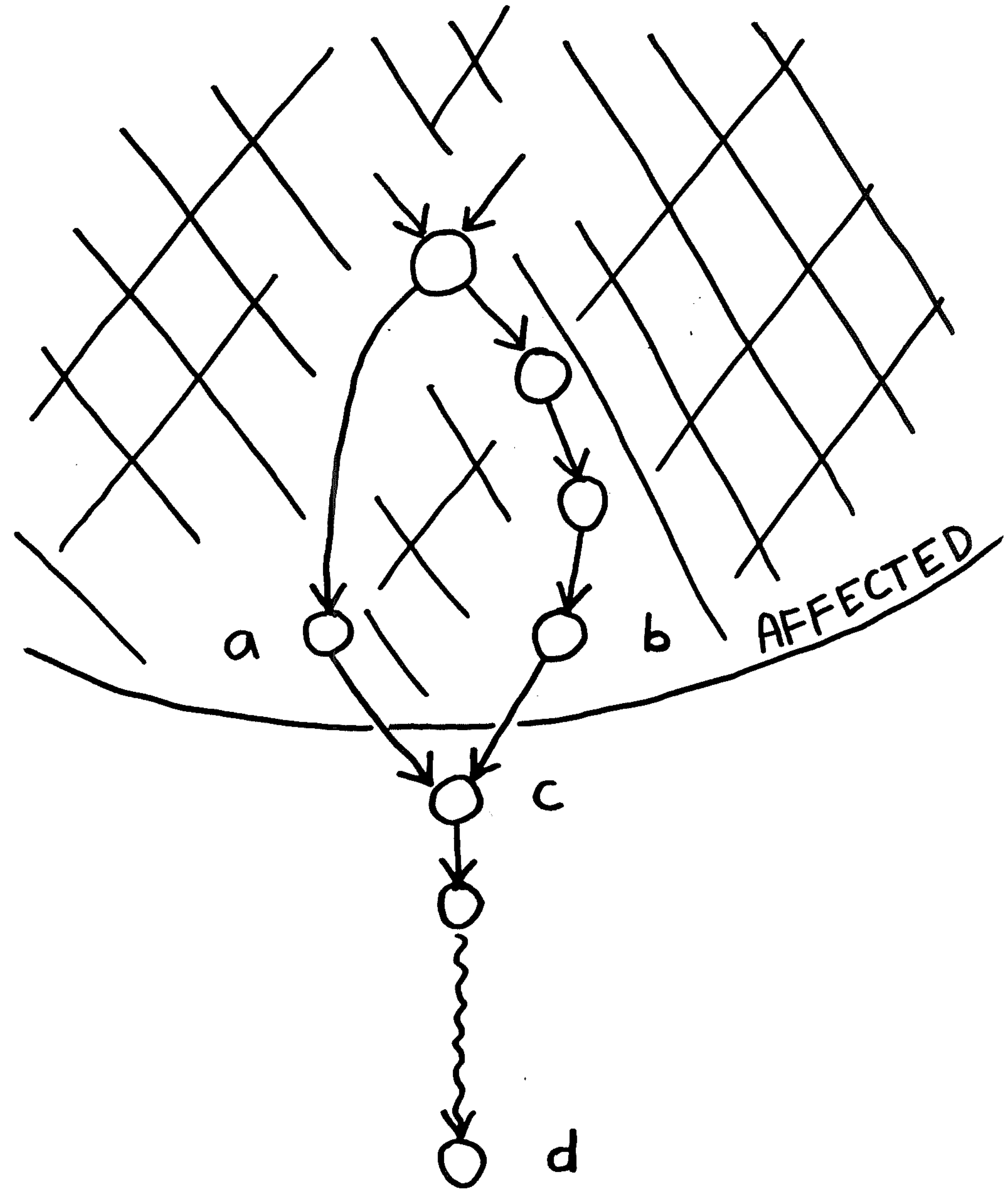


Inconsistent Tree $\Longrightarrow$ Consistent Tree

AFFECTED

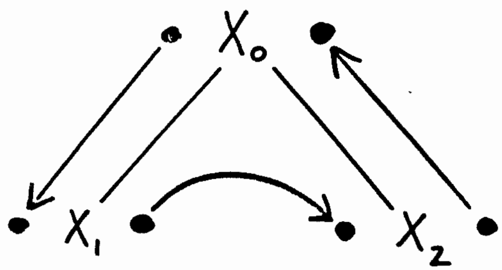
Updating cost : $O(|\text{AFFECTED}|) = O(||\delta||)$

[Reps - POPL 82]



Ordered Attribute Grammars [Kastens 1980]

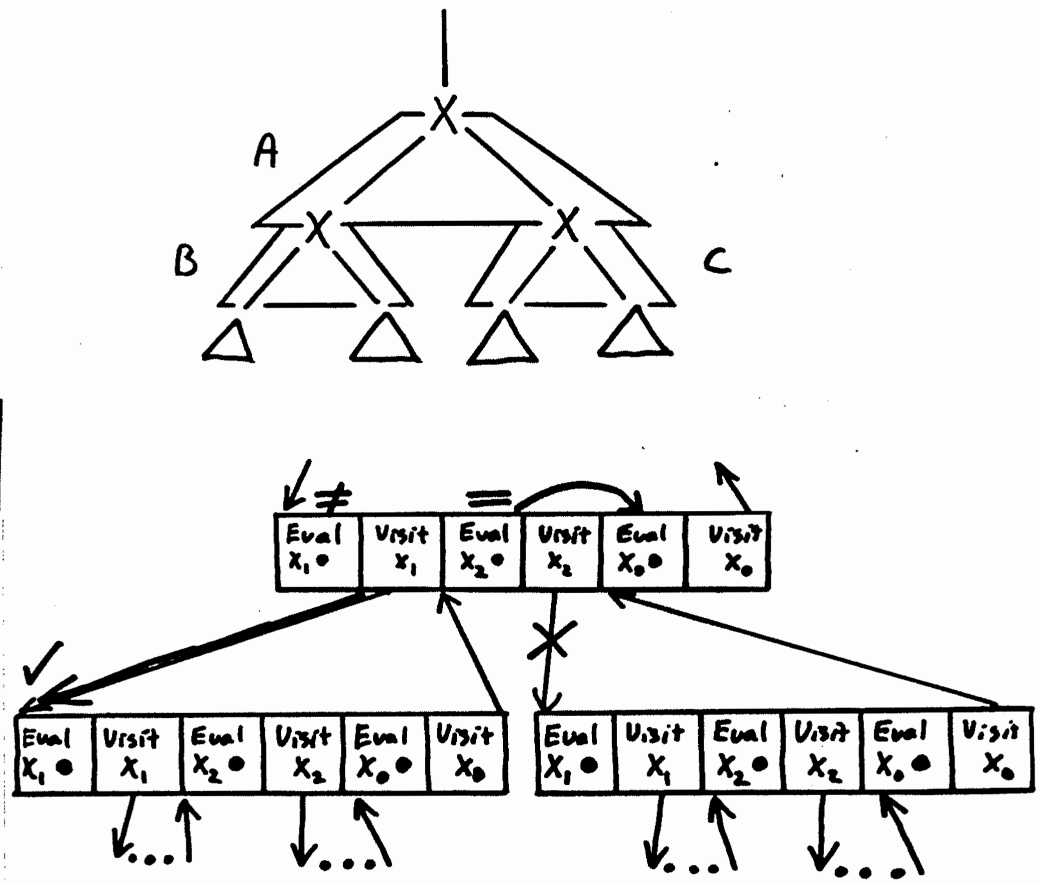
production:

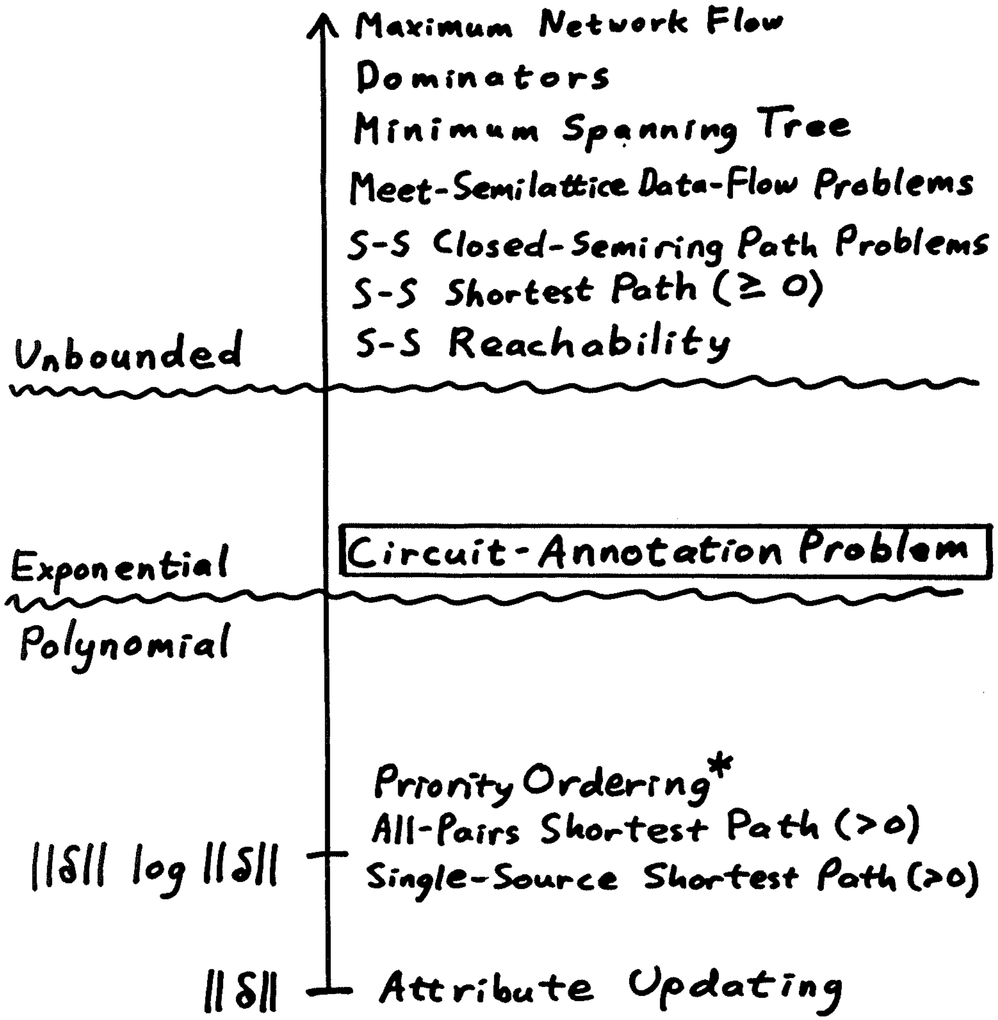


plan:

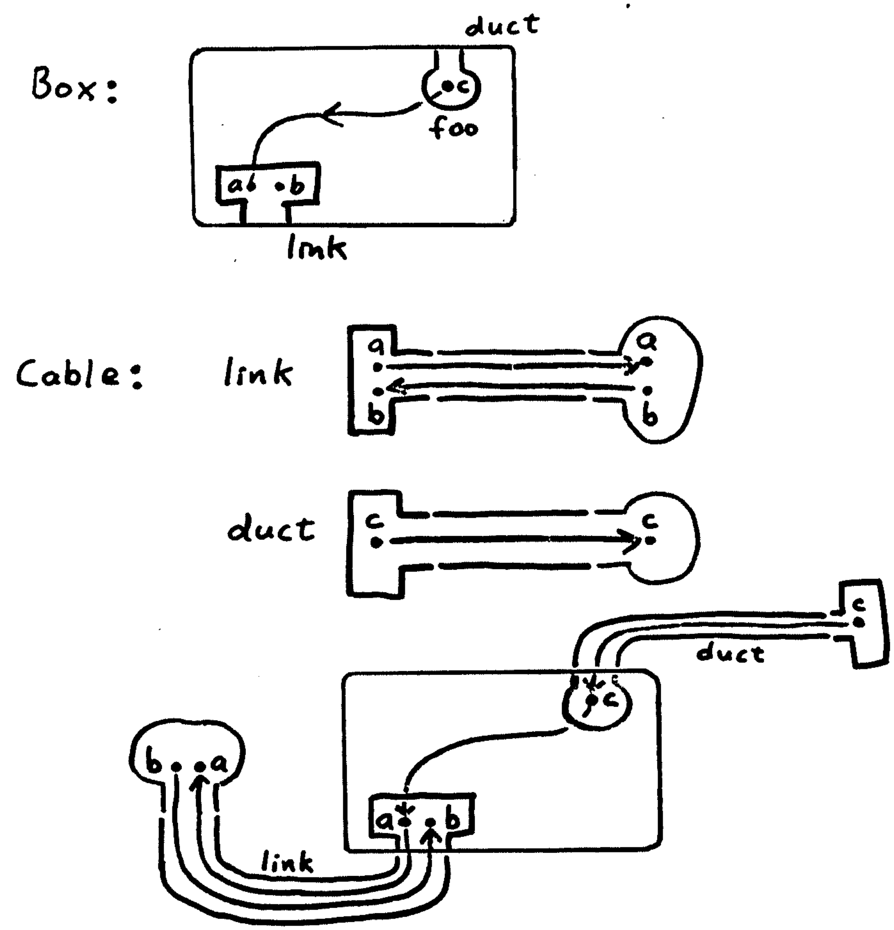
Eval X_1 •	Visit X_1	Eval X_2 •	Visit X_2	Eval X_0 •	Visit X_0

Optimal Updating for Ordered AGs [Yeh 1983]





CABLES & BOXES [Alpern et al. PE 1988]



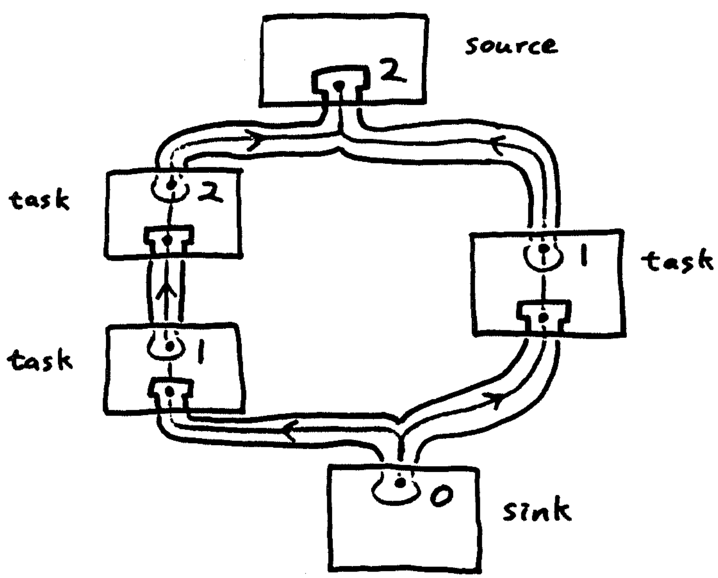
Substrate graph: Boxes + Cables

Dependence graph: Attributes + Edges

Example: PERT Chart

```
link: CABLE {  time_to_completion: BLUE→RED }

source: BOX {  critical_path_length: LOCAL REAL }
               out :  * RED link
               critical_path_length *= max(out.time_to_comp)
            }
task:   BOX {  out :  * RED link
               in  :  * BLUE link
               in.time_to_completion *= 1+max(out.time_to_comp)
            }
sink:   BOX {  in  :  * BLUE link
               in.time_to_completion = 0.0
            }
```



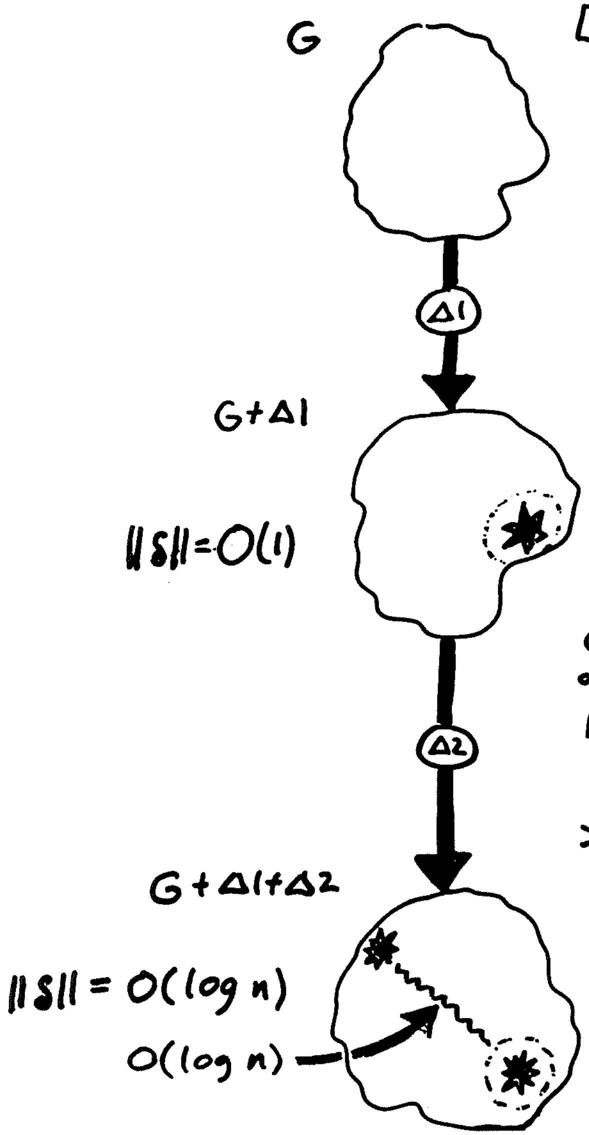
Locally Persistent Algorithms

[Alpern et al. – SODA 90]

- Vertices, edges -- blocks of storage
- Vertex block -- pointers to predecessors and successors + auxiliary information
- Edge block -- source and target + auxiliary information
- Auxiliary information cannot include any auxilsary pointers
- No global information maintained between updates
- Update algorithm follows pointers; choice can depend on vertices and edges visited so far (e.g., can use stacks or queues as worklists)

Circuit Annotation Is Not Polynomially Bounded

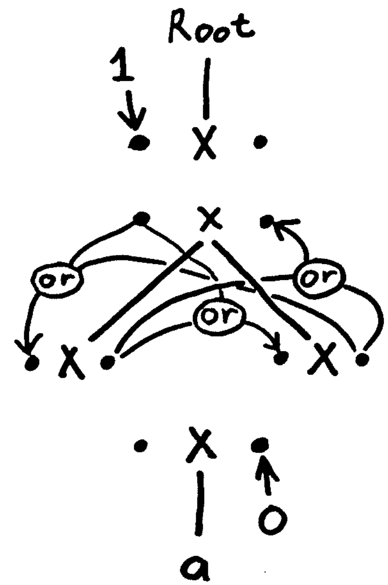
[Alpern etal. SODA 90]

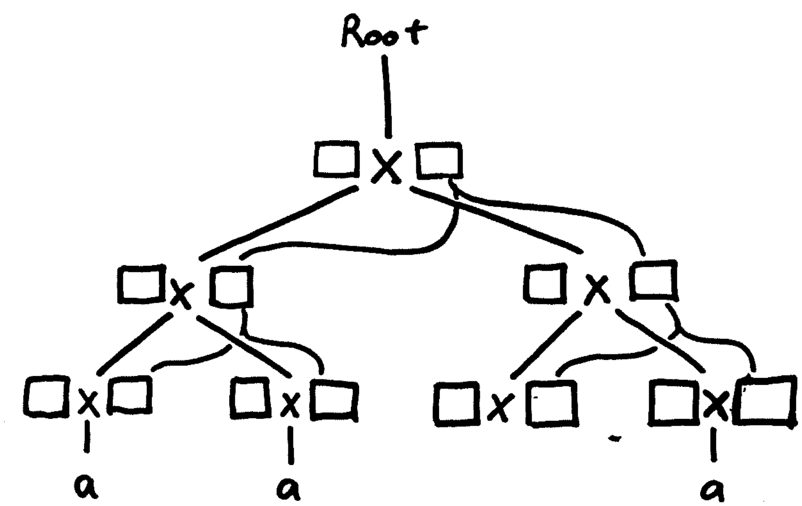


Δ1 is just one of O(n) possible modifications

Cannot distinguish among the O(n) possibilities without O(n) work

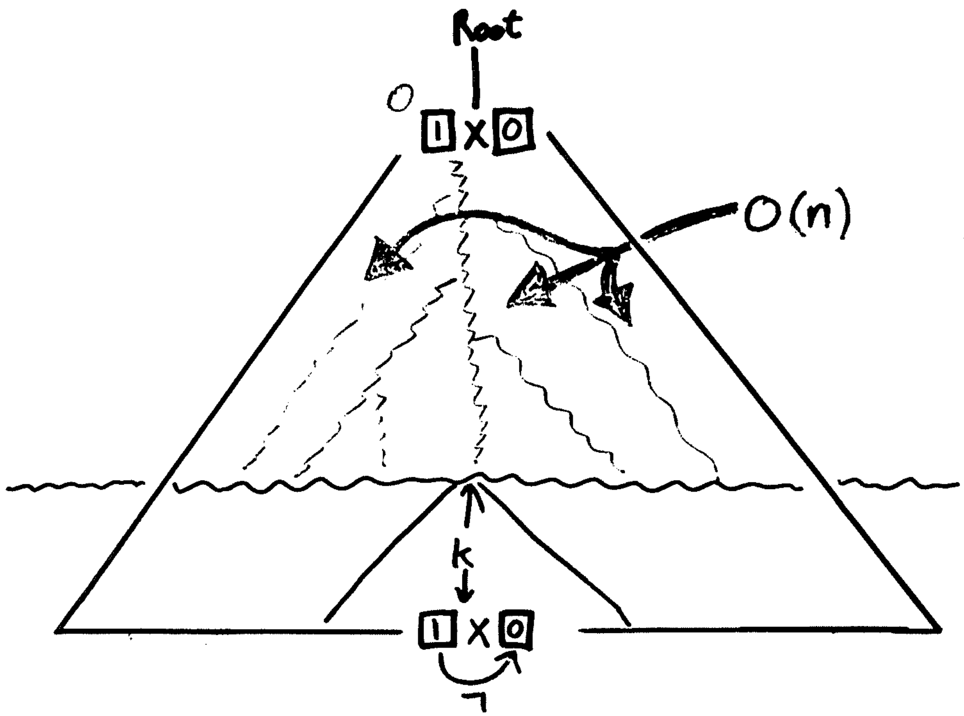
Δ1 at leaf, Δ2 at root of a complete binary tree of height O(log n)



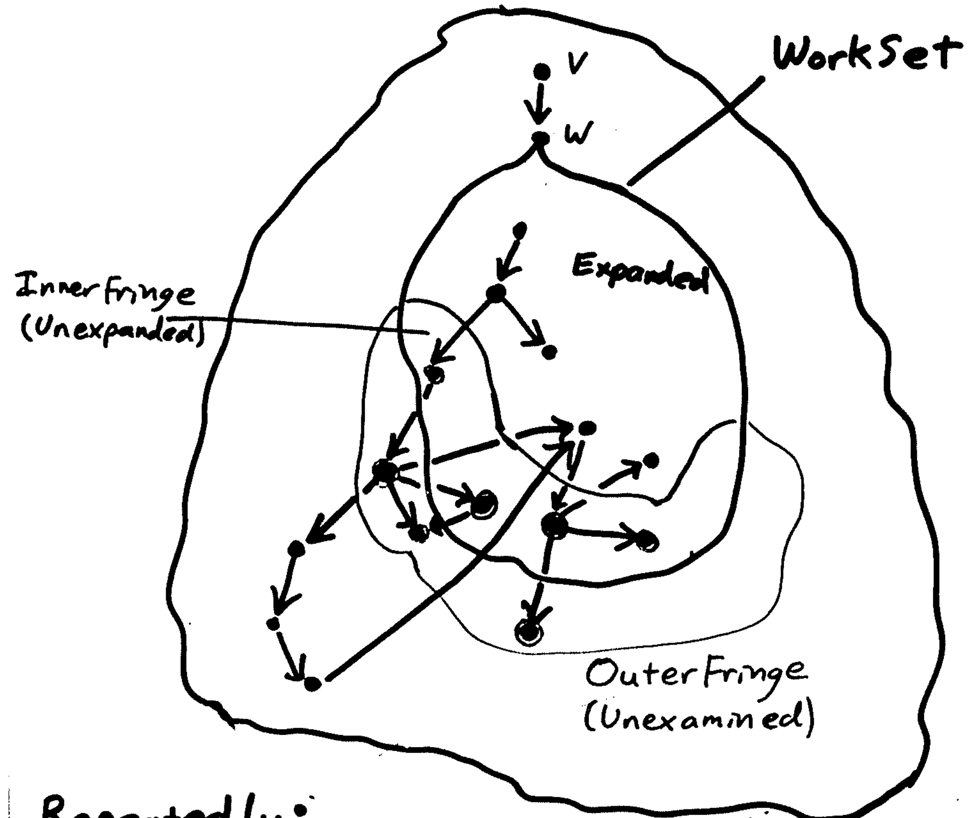


Circuit Annotation is Bounded

[Ramalingam & Reps 1991,92]



$$\left.\begin{array}{l} \|\delta\| = O(\log n) \\ \text{Work} = \Omega(n) \end{array}\right\} \Rightarrow \Omega(2^{\|\delta\|})$$

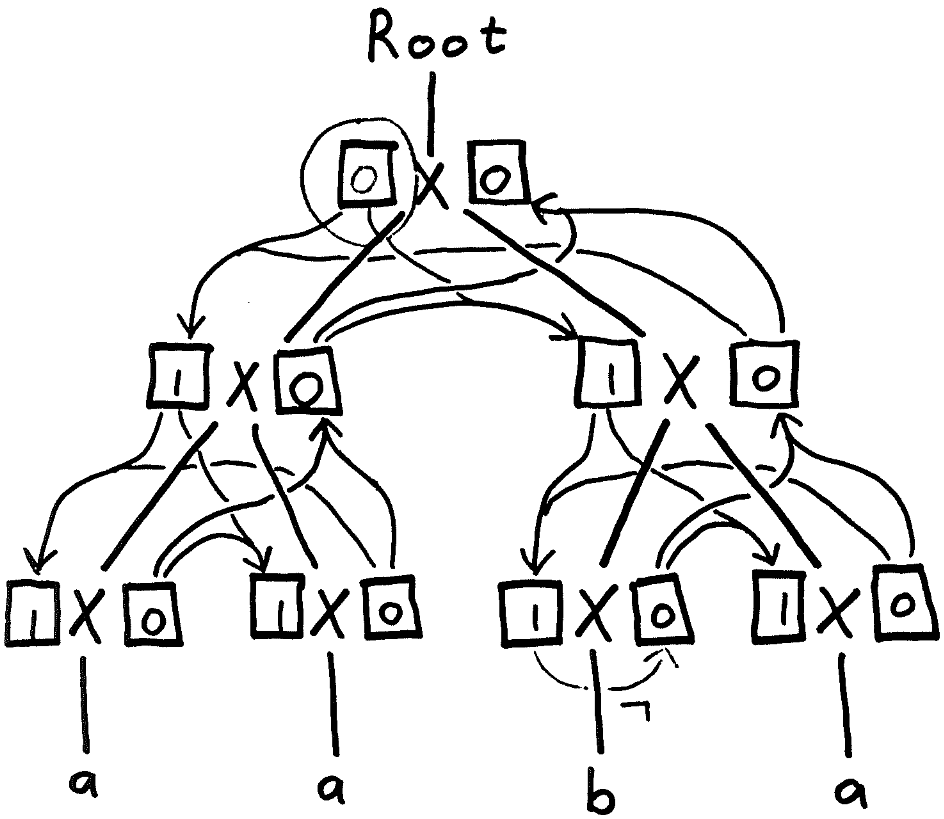


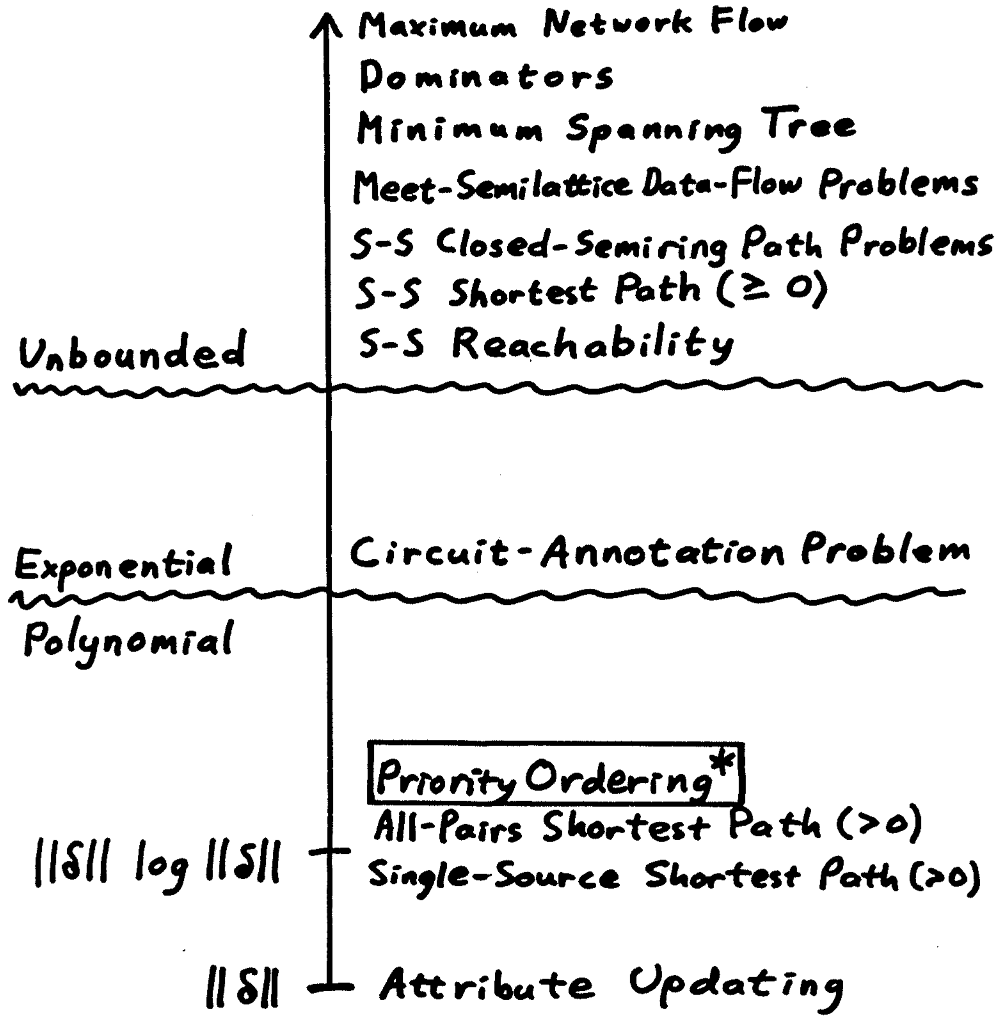
Repeatedly:

(1) Re-evaluate Workset in relative topological sort order

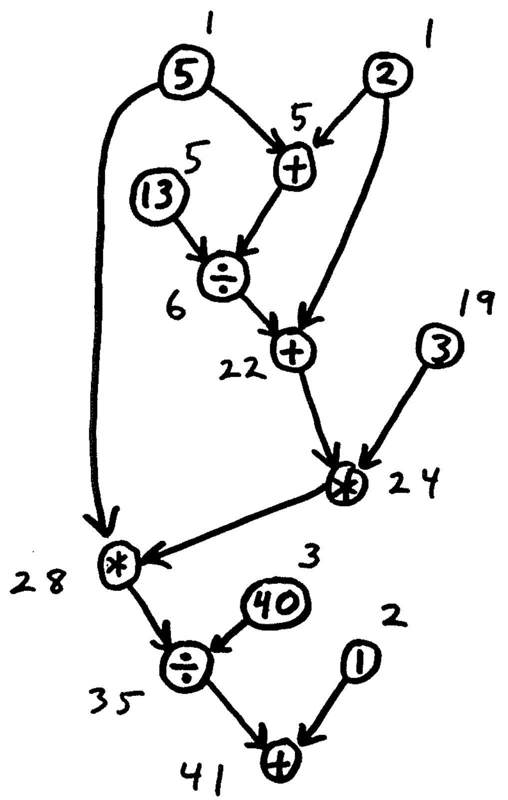
(2) Expand all vertices of Inner Fringe

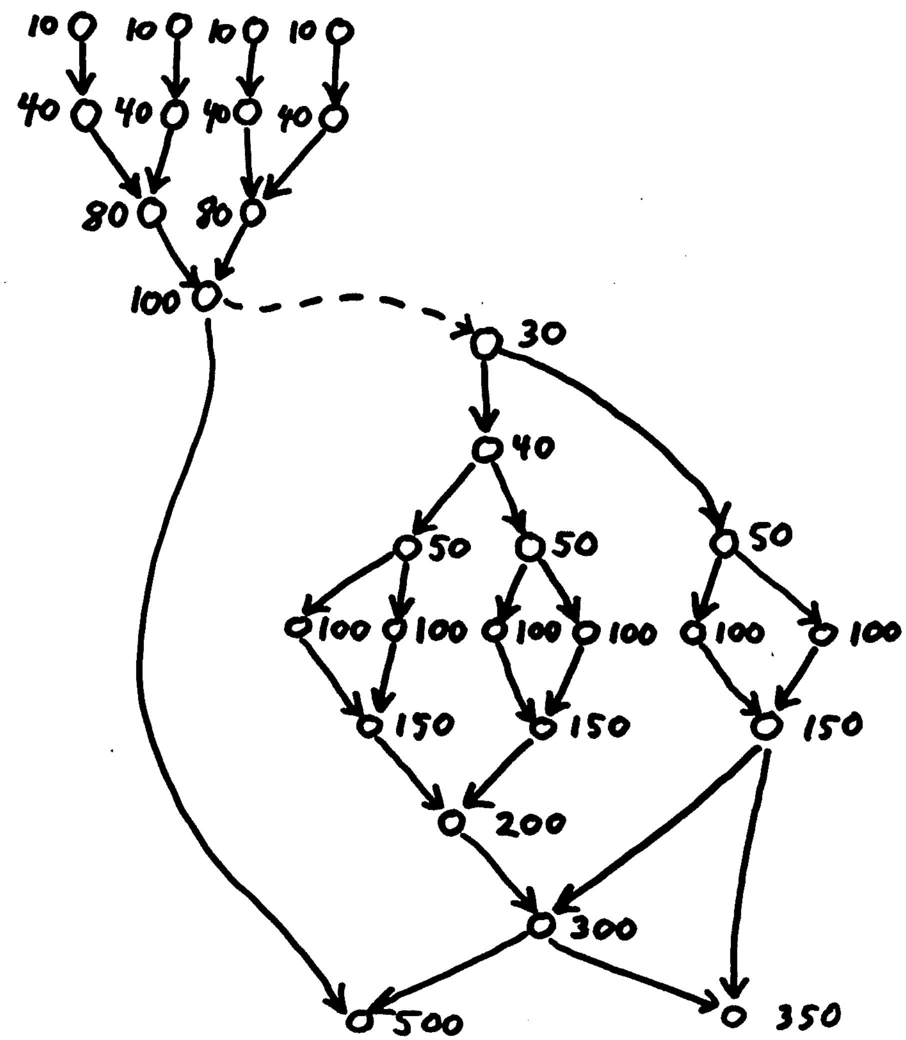
Work: $O(2^{\|\delta\|})$


Root
1
0
X
a
b
¬


Maximum Network Flow
Dominators
Minimum Spanning Tree
Meet-Semilattice Data-Flow Problems
S-S Closed-Semiring Path Problems
S-S Shortest Path (≥ 0)
S-S Reachability
Unbounded
Exponential
Circuit-Annotation Problem
Polynomial
Priority Ordering*
All-Pairs Shortest Path (>0)
||δ|| log ||δ||
Single-Source Shortest Path (>0)
||δ||
Attribute Updating

Priority Ordering

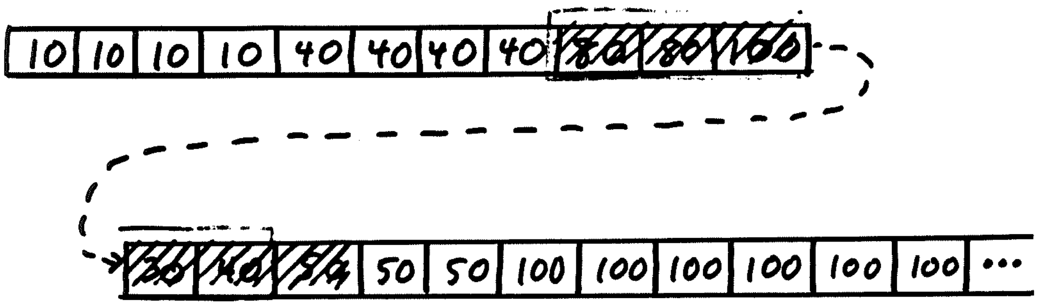




Evaluation cost: $O(n \log n)$

Updating cost: $O(\|\delta\| \log \|\delta\|)$

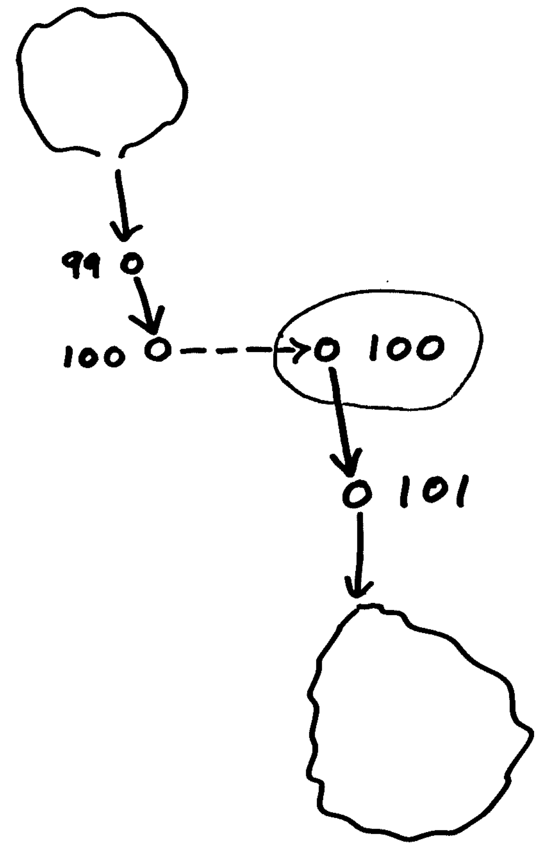
Finding a Cover



Lockstep: $\leq 2 * \|\text{min. cover}\|$

[Alpern et al. – SODA 90]

Updating Priorities



Priority space [Dietz & Sleator STOC 87]

Not locally persistent

Circuit Annotation Updating Via Priorities

1. Update priorities

   $O(\|\delta_{po}\| \log \|\delta_{po}\|)$

2. Change propagation

   $O(\|\delta_{Values}\| \log \|\delta_{Values}\|)$

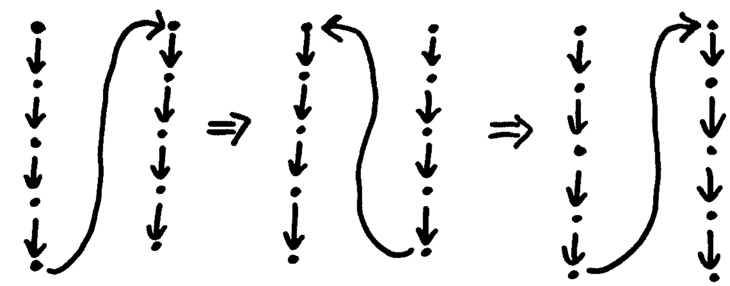
   Unbounded

   vs.

   Bounded: $O(2^{\|\delta_{values}\|})$

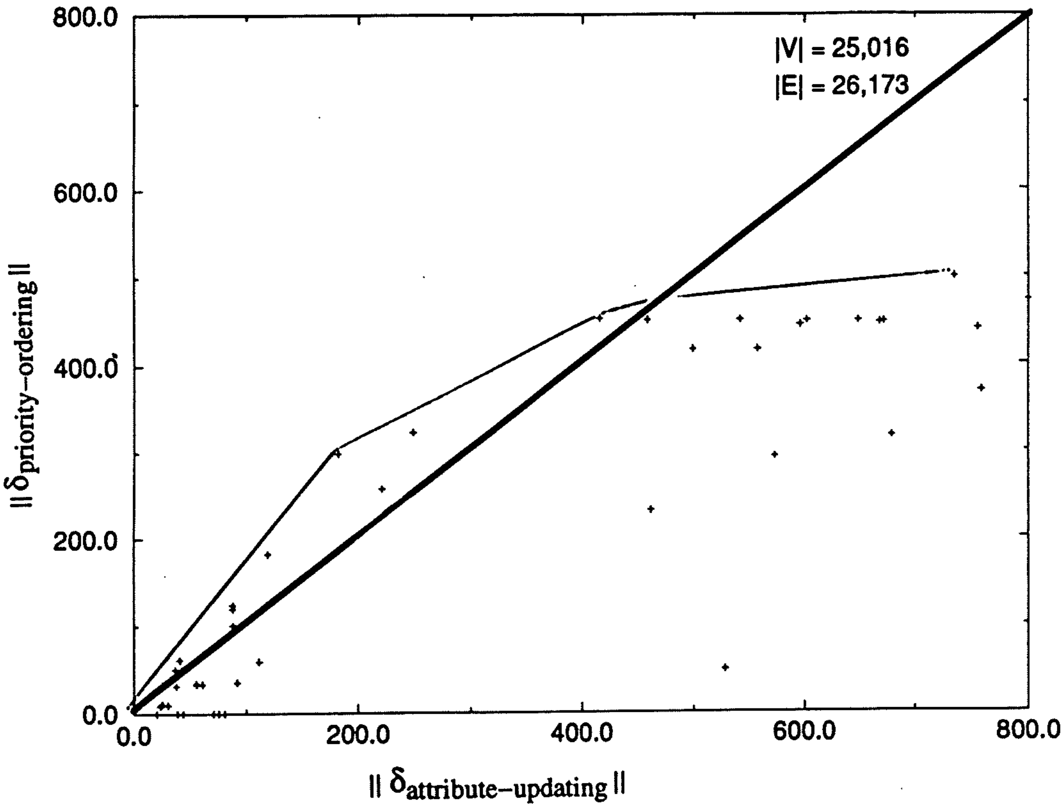
Observations

- Relation vs. function
- Algorithm uses persistent auxiliary storage
- Not candidate for amortized analysis

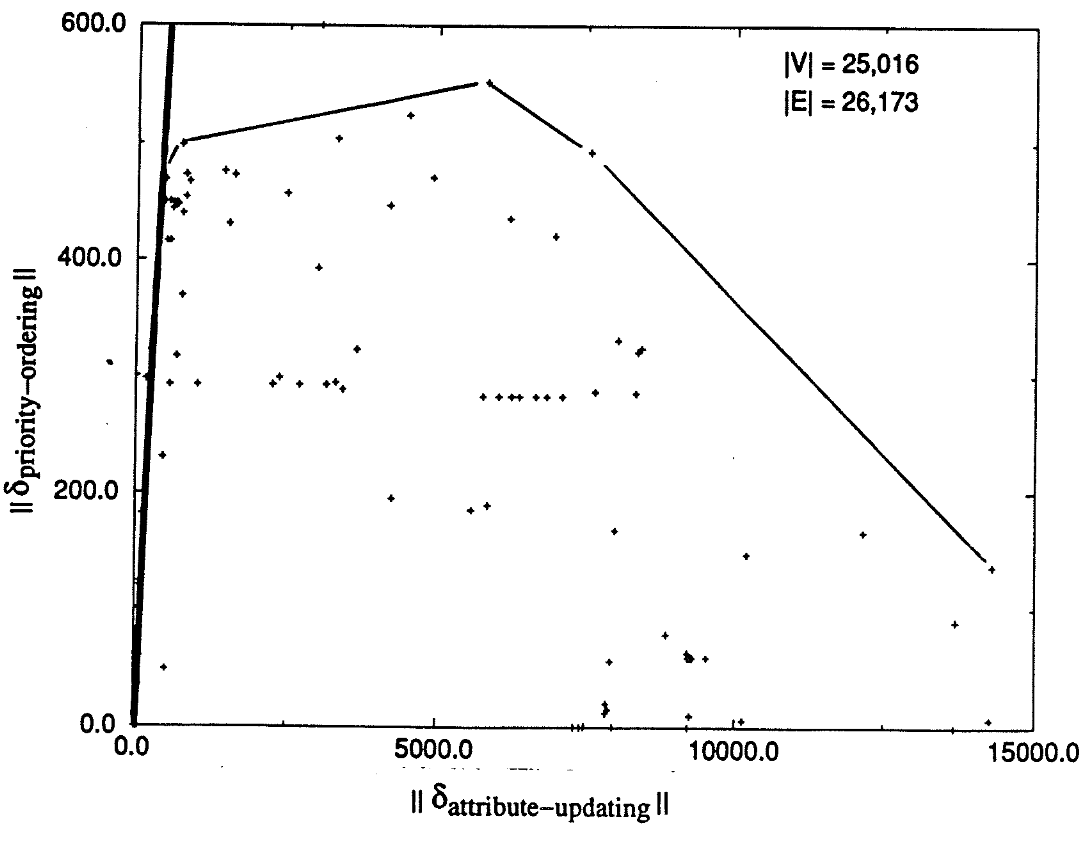


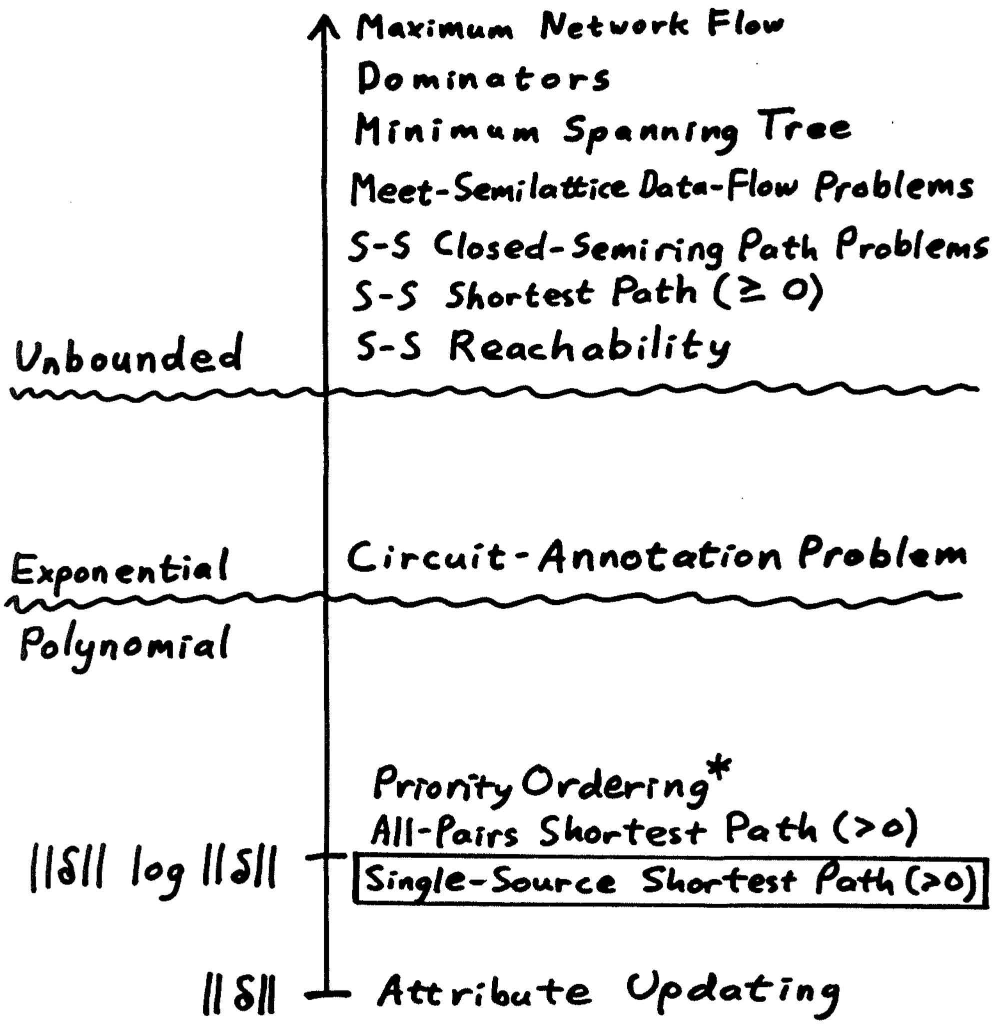
- Competitive ?

[Ramalingam 93]



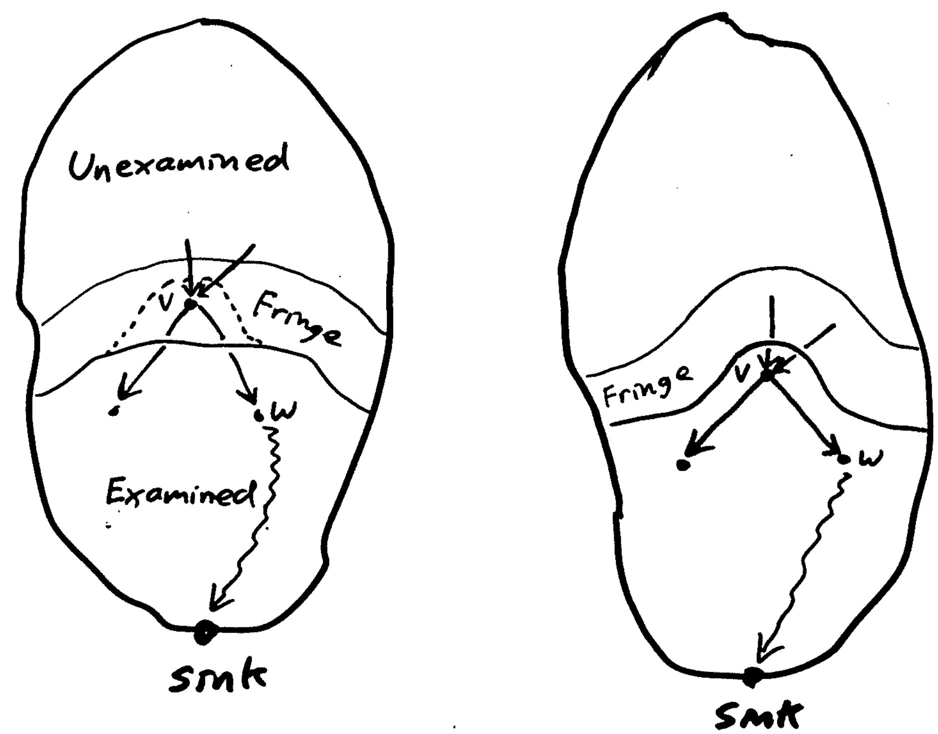
(Attribute dependence graph of a Pascal program)



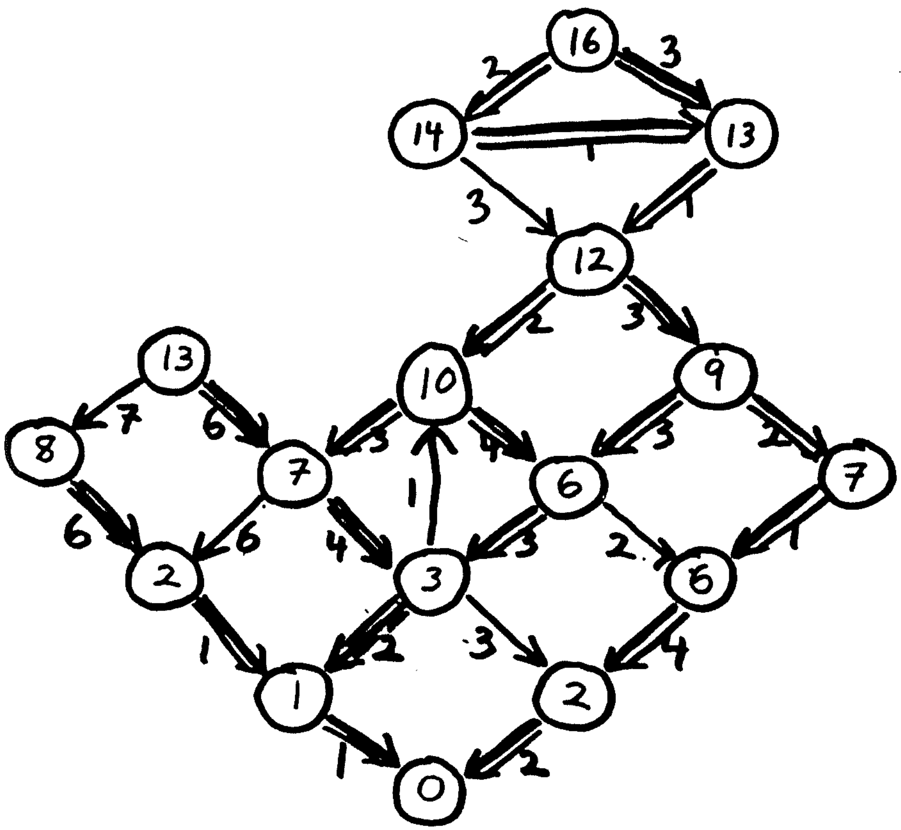


Single-Sink Shortest-Path Problem
(with positive edge weights)

Dijkstra's (batch) algorithm:



Single-Sink Shortest-Path Problem (with positive edge weights)



DeleteEdge(G, v → w)

```
Phase 1:
  WorkSet := {v}
  AffectedVertices := ∅
  while WorkSet ≠ ∅ do
    Select and remove a vertex
        u from WorkSet
    Insert u into AffectedVertices
    for each red edge x → u do
      uncolor x → u
      if ∄ a red edge x → y then
         Insert x into WorkSet
      fi
    od
  od
Phase 2:
  Determine new distances in
  induced graph <N(AffectedVertices)>
  (e.g. via Dijkstra's algorithm)
```

4 3 2 1

Complexity of Delete Edge

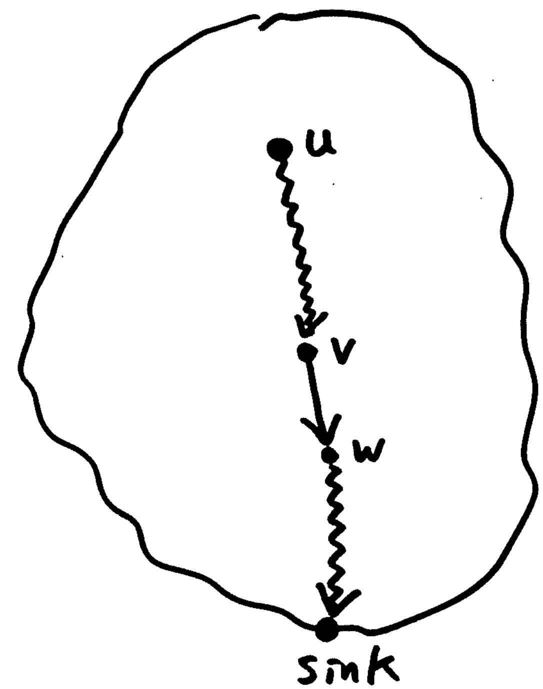
Phase 1: $O(\|\delta\|)$

Phase 2: $O(\|\delta\| \log \|\delta\|)$

Total: $O(\|\delta\| \log \|\delta\|)$

[Ramalingam & Reps 1991]

Edge Insertion: $O(\|\delta\| \log \|\delta\|)$



(*) dist(u,v) + length(v → w) + dist(w) < dist(u)?

Idea: Perform (most of) the batch single-sink problem with sink v (but only visit vertices for which (*) holds -- and their predecessors)

Maximum Network Flow

Dominators

Minimum Spanning Tree

Meet-Semilattice Data-Flow Problems

S-S Closed-Semiring Path Problems

S-S Shortest Path (≥ 0)

Unbounded — S-S Reachability

Exponential — Circuit-Annotation Problem

Polynomial

Priority Ordering*

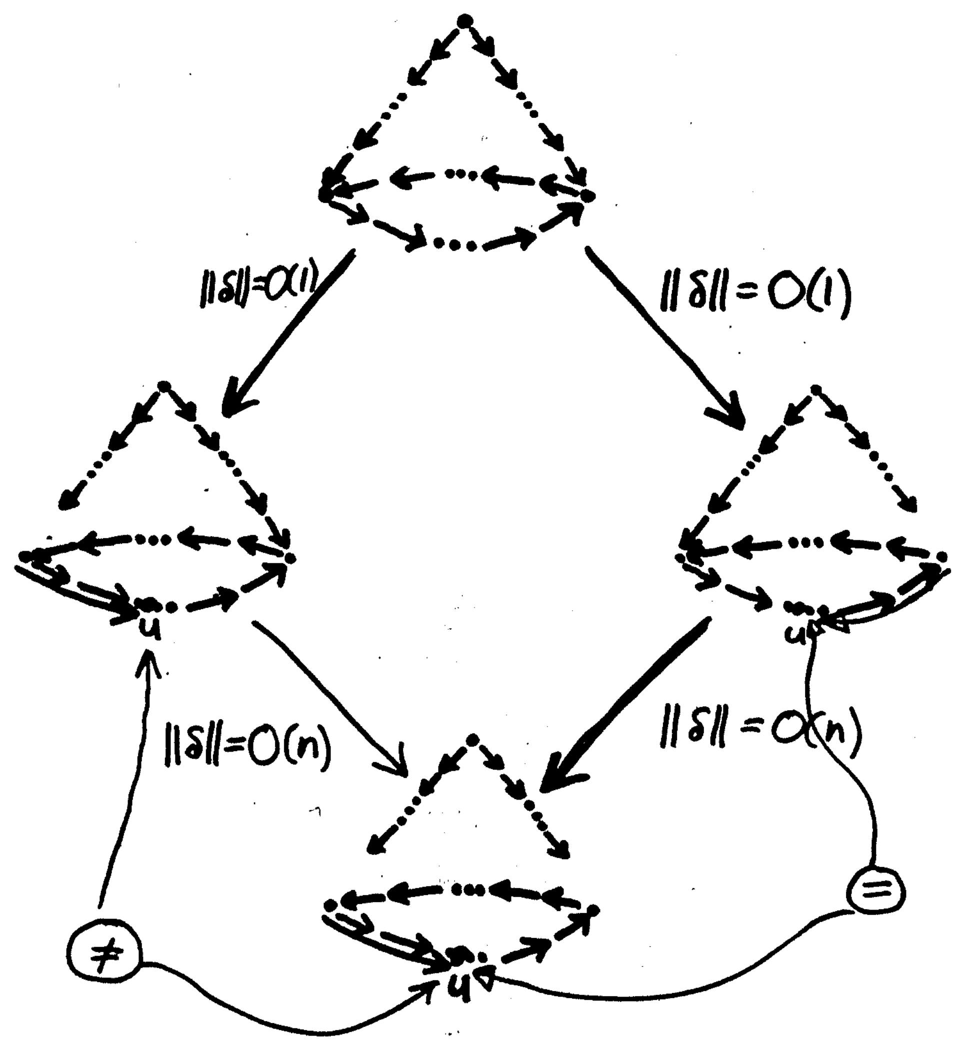
All-Pairs Shortest Path (>0)

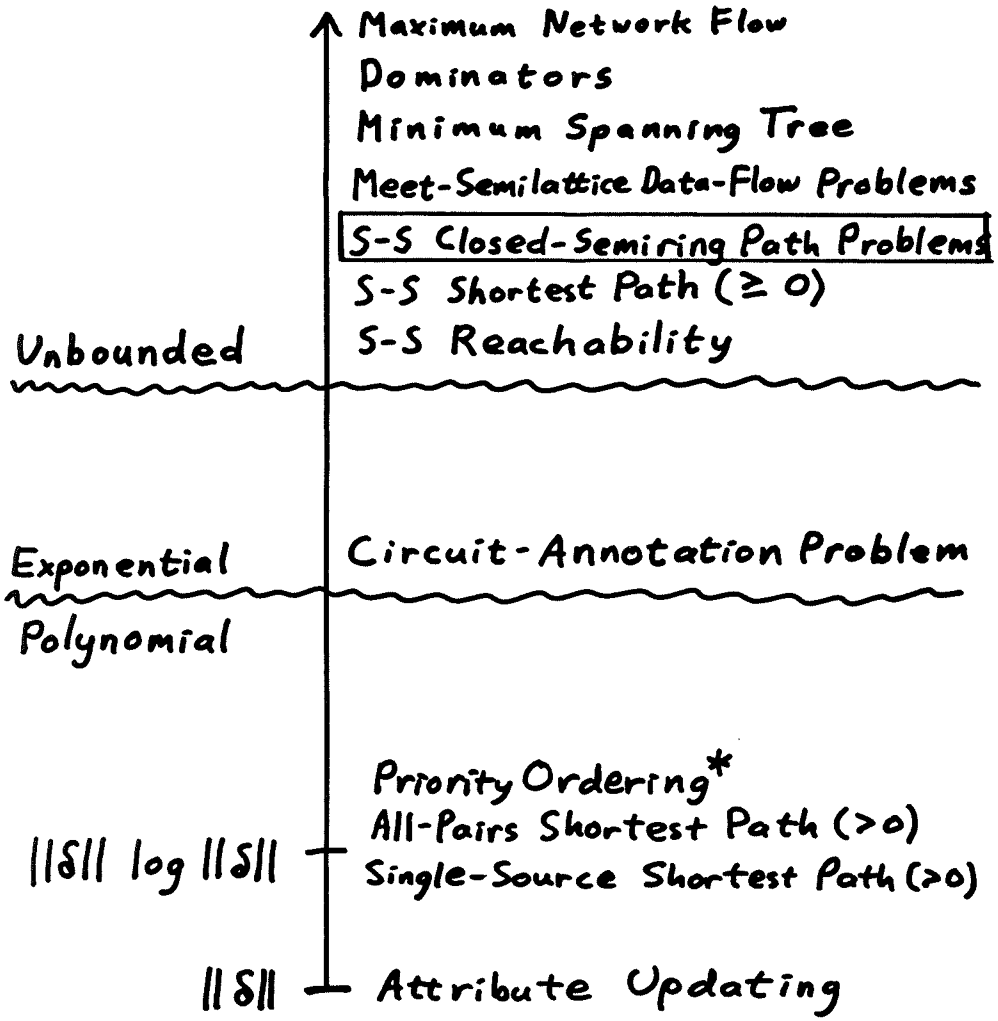
$\|\delta\| \log \|\delta\|$ — Single-Source Shortest Path (>0)

$\|\delta\|$ — Attribute Updating

Reachability is Unbounded

[Ramalingam & Reps 1991]





(Meet-Semilattice) Data-Flow Problems

G: flow graph

s: entry vertex of G

L: semilattice (with $\top$)

M: edge $\to L \to L$
(labels edges with flow functions)

$c \in L$: constant associated with s

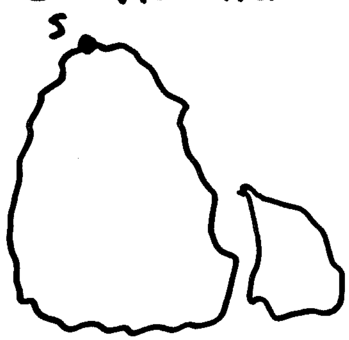
The solution is the maximal fixed point of:

$$S(s) = c$$

$$S(u) = \bigsqcap_{v \in pred(u)} M(v \to u)(S(v))$$

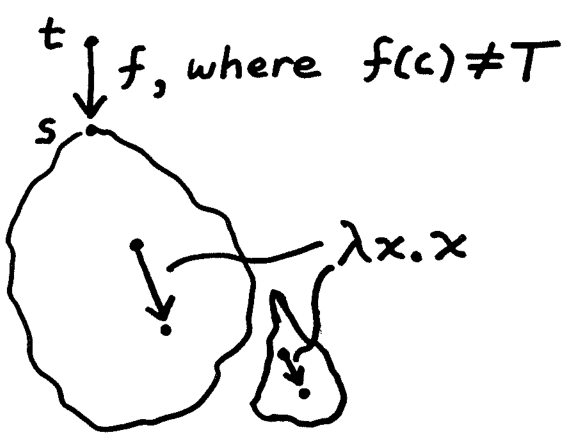
There Are Unbounded Data-Flow Problems

Instance of S-S Reachability:



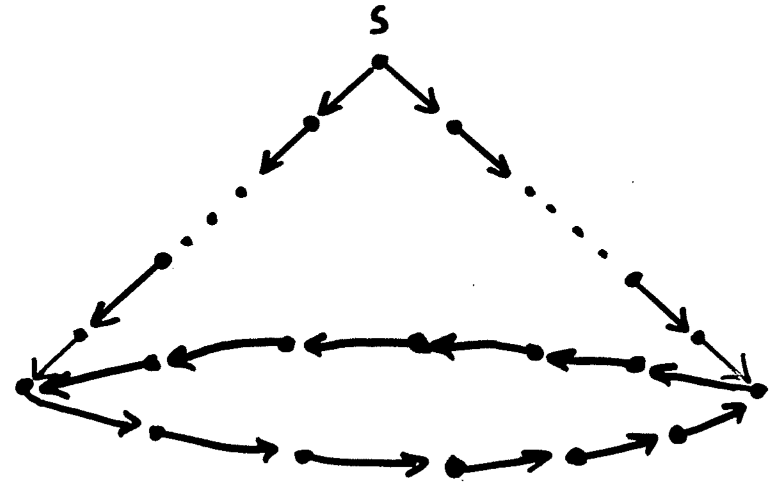
Instance of data-flow problem P:

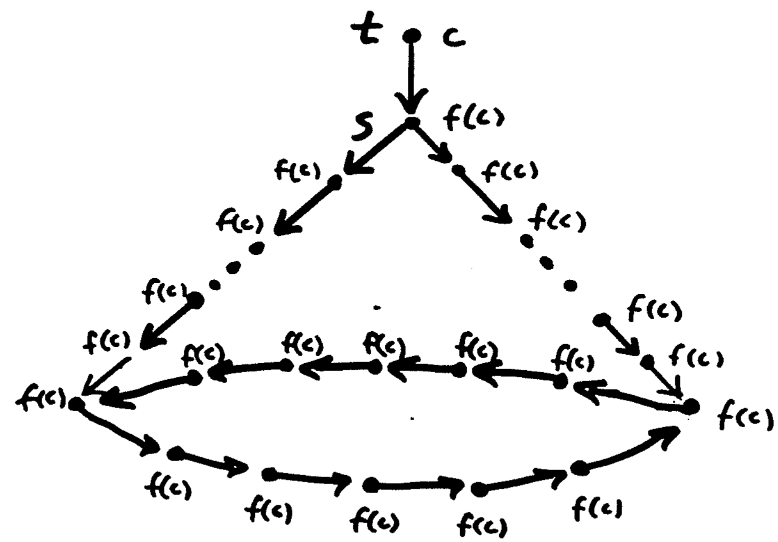
$$\boxed{\begin{array}{l} S(t) = c \neq \top \\ M(e) = \lambda x.x \end{array}}$$



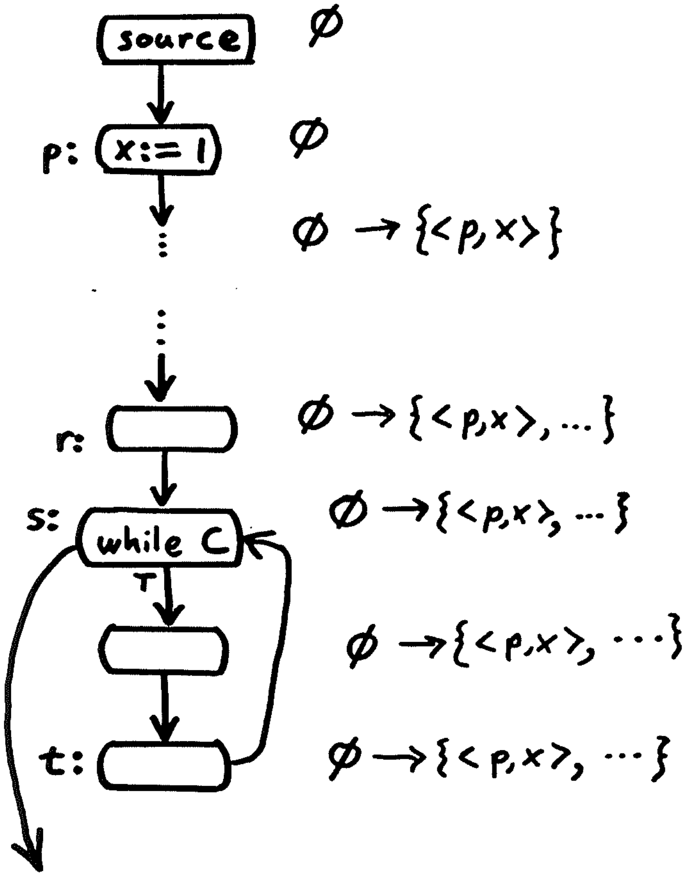
$S(u) = f(c)$ if u is reachable from s

$S(u) = \top$ if u is not reachable

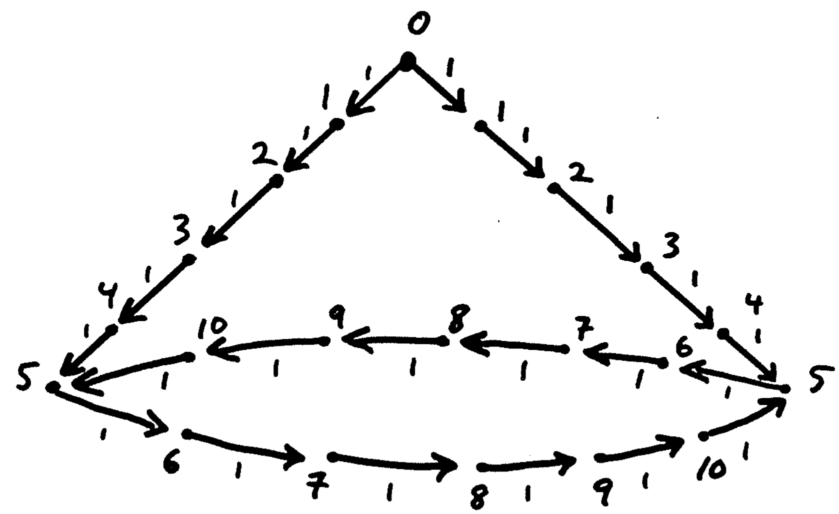


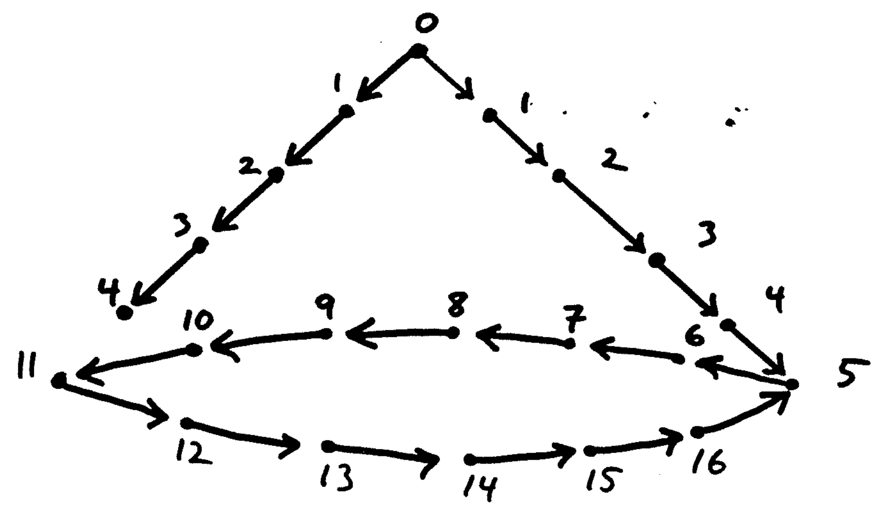


Reaching Definitions



Reachability and SSSP > 0





Reductions Between Problems

(1) Give transformations:

a. $\text{Instance}_P \longrightarrow \text{Instance}_Q$

b. $\text{Solution}_Q \longrightarrow \text{Solution}_P$

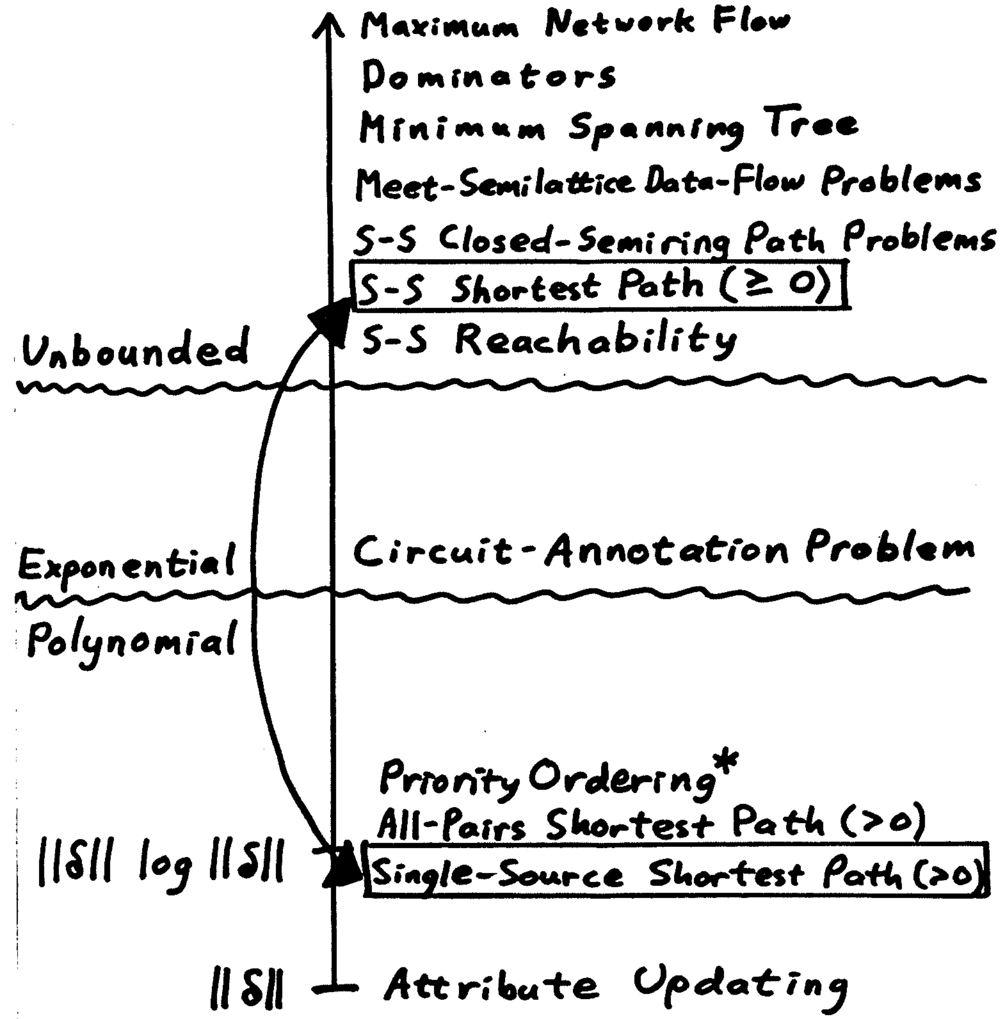
c. $\delta_P \longrightarrow \delta_Q$

d. $\Delta\,\text{Solution}_Q \longrightarrow \Delta\,\text{Solution}_P$

(2) Show that (1)c and (1)d are bounded by a function of $\|\delta_P\|$

(3) Show that $\|\delta_Q\|$ is bounded by a function of $\|\delta_P\|$

(4) Show that (1)c and (1)d are locally persistent.



Maximum Network Flow

$\boxed{\text{Dominators}}$

Minimum Spanning Tree

Meet-Semilattice Data-Flow Problems

S-S Closed-Semiring Path Problems

S-S Shortest Path (≥ 0)

S-S Reachability

Unbounded

Circuit-Annotation Problem

Exponential

Polynomial

Priority Ordering*

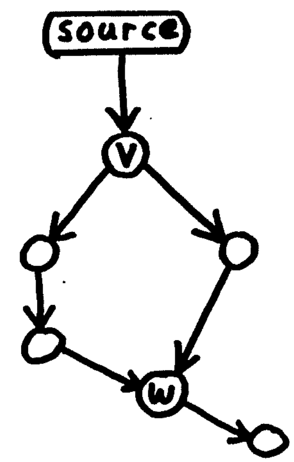
All-Pairs Shortest Path (>0)

$\|\delta\| \log \|\delta\|$ — Single-Source Shortest Path (>0)

$\|\delta\|$ — Attribute Updating

Dominators in a Directed Graph

Vertex v **dominates** w iff all paths source $\xrightarrow{*}$ w contain v.



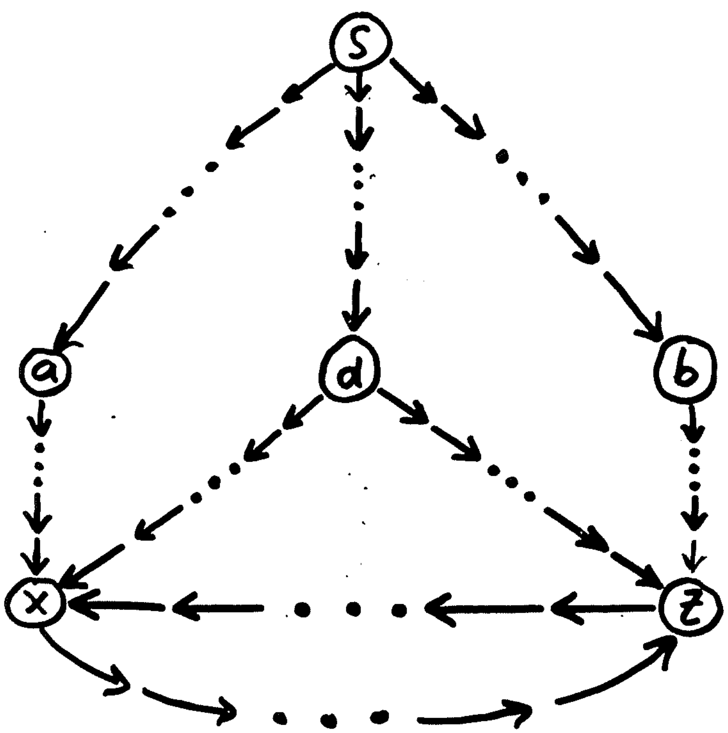
Data-flow problem: $M(v \to u) = \lambda x. \{v\} \cup x$

$S(\text{source}) = \emptyset$

$S(u) = \bigcap_{v \in pred(u)} M(v \to u)(S(v))$

Identity function not allowed on an edge.

Dominators is Unbounded



Without $\longrightarrow$ and $\longrightarrow$, d dominates the $\{x, y, z\}$- loop.

Boundedness

- Good bounded algorithms exist for certain problems
- A good heuristic exists for acyclic problems
- Many problems of interest have no bounded locally persistent algorithm
  - How can persistent auxiliary storage and non-local pointers be used?
- Hybrid analysis
  - $O(f(n, \|\delta\|))$

Talk Outline

Introduction

Assessment of incremental algorithms

Graph-annotation problems

Other update problems
• INC • Function caching • Dynamization

"Incrementalizers"

Conclusions

INC

- Language for computations on bags
- Changes to arguments $\rightarrow$ change in final result
- Differential updating used
  - selective recomputation: coarse-grained incrementality
  - differential updating: fine-grained incrementality
- "Never worse" than recomputing from scratch

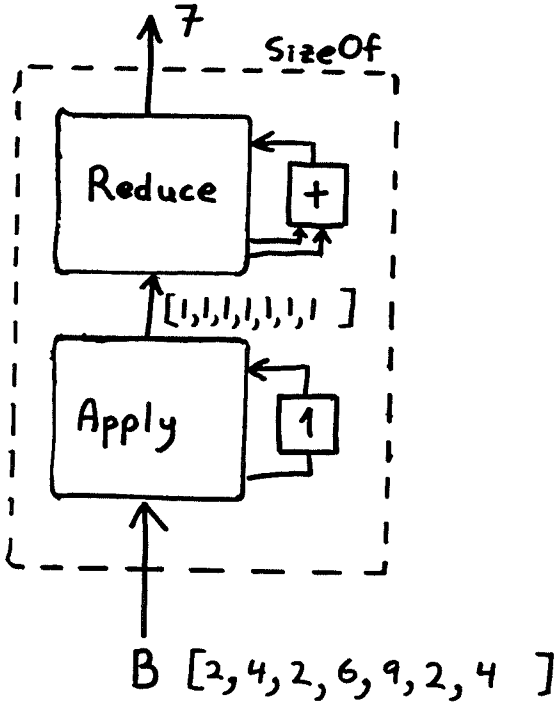
[Yellin & Strom 1991]

Computing the Size of a Bag

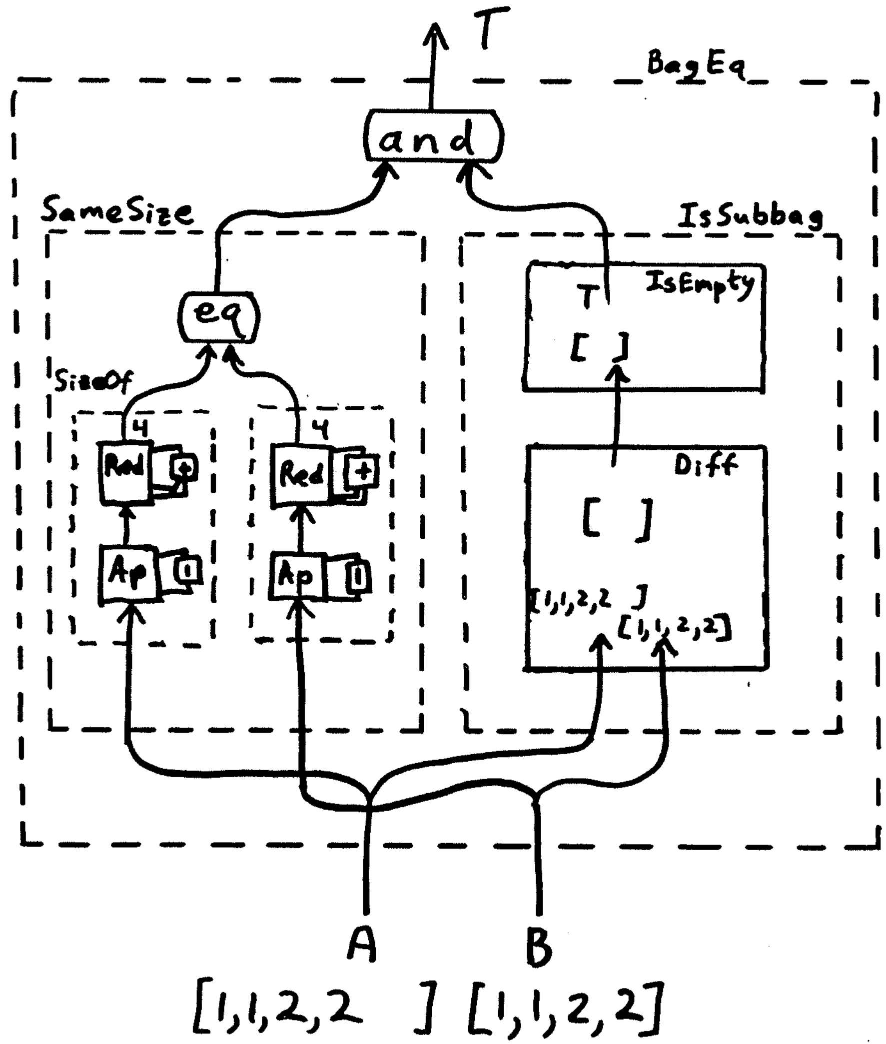
B = [2, 4, 2, 6, 9, 2, 4]

Apply[1] B = [1, 1, 1, 1, 1, 1, 1]

Reduce[+](Apply[1] B) = 7



Example: Bag Equality



Differential Updating in INC

- Produce smallest message

  $B_{old} = \{1, 2, \cdots, 50\}$

  $B_{new} = \{49, 50, \cdots, 100\}$

  $\Delta^{+} = \{51, \cdots, 100\}$

  $\Delta^{-} = \{1, 2, \cdots, 49\}$

  $Remainder = \{49, 50\}$

- $T_{Inc} = O(T_{Batch})$

  at most a constant factor to maintain structures used during updating

Incremental Computation via Function Caching

[Pugh & Teitelbaum – POPL 1991]

Applicability: Decomposable list problems

$L = A \| B$

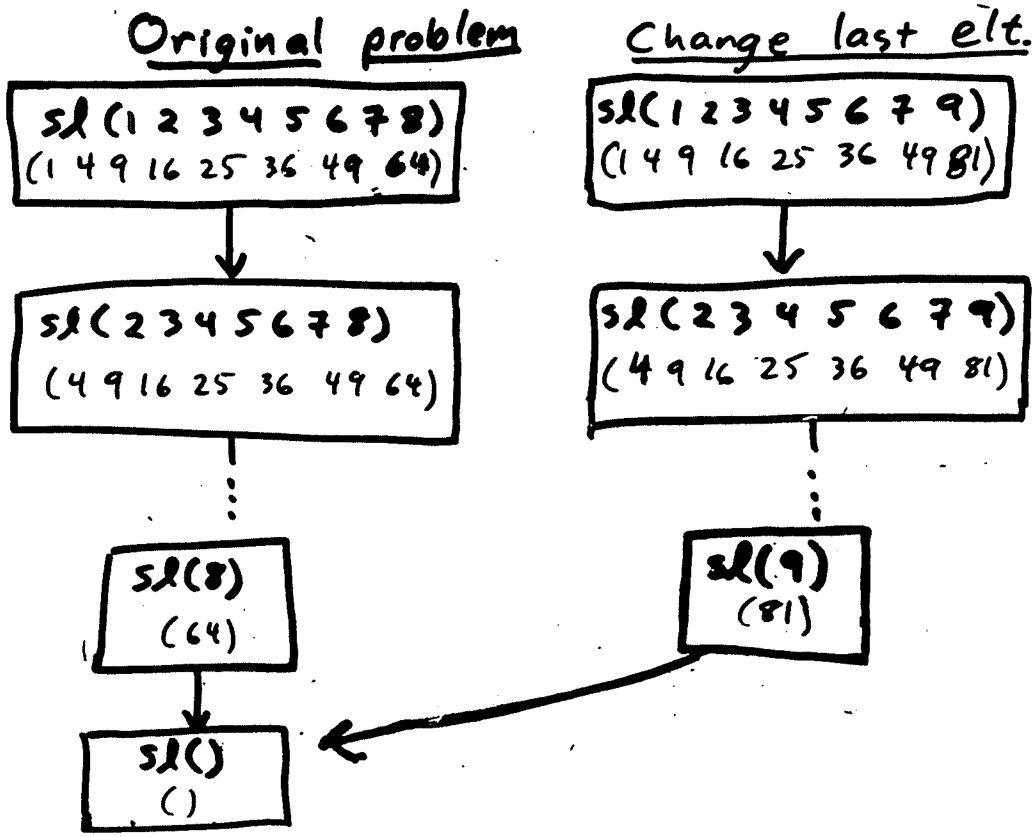
$f(L) = \square(f(A), f(B))$

Ex: Square the elements of a list of integers.

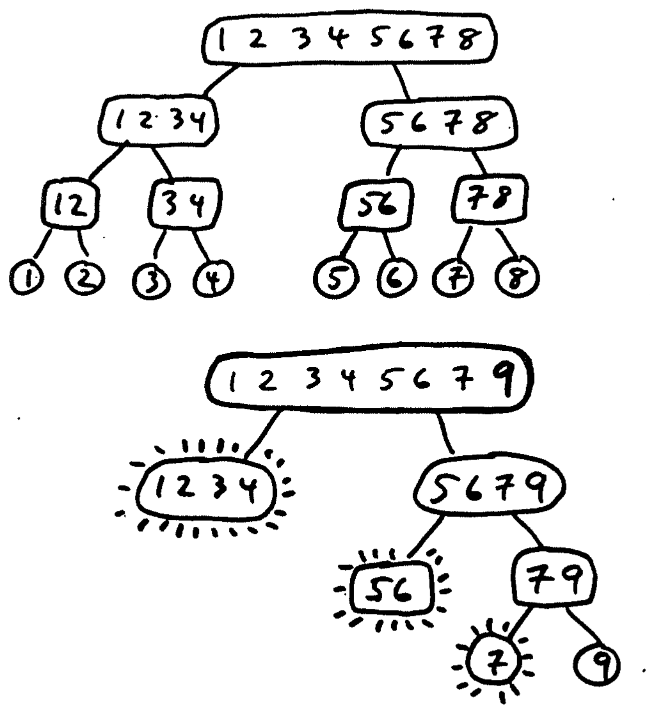
Input: (1 2 3 4 5 6 7 8)
Output: (1 4 9 16 25 36 49 64)

```
sl(L) = if null(L) then L
        else cons(hd(L)**2, sl(tl(L))) fi
```

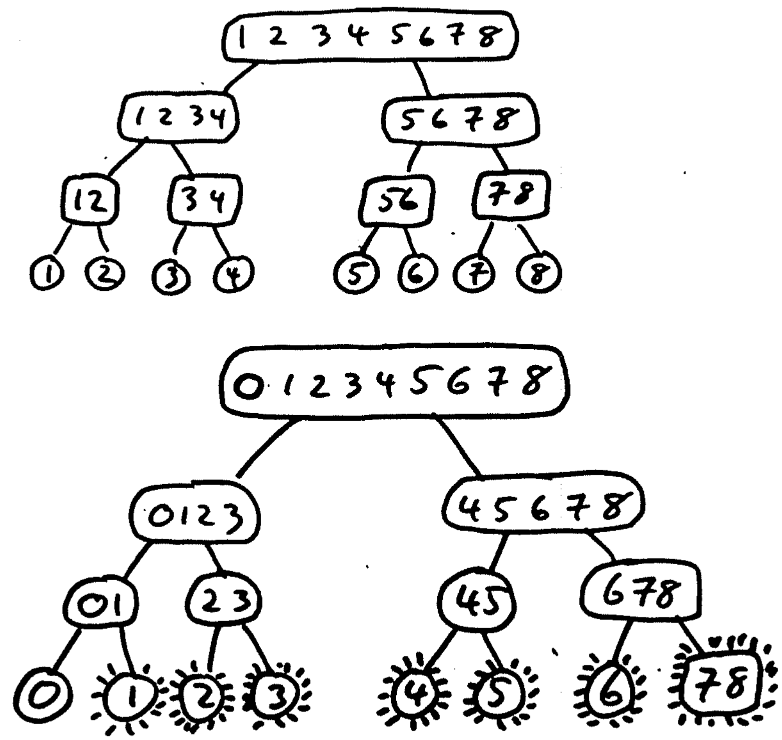
Idea: Cache previous values



ss(S) = if length(S) = 1 then hd(S)**2 :: nil
else append(ss(first-half(S)), ss(second-half(S)))



ss(S) = if length(S) = 1 then hd(S)**2 :: nil
else append(ss(first-half(S)), ss(second-half(S)))



Goal: Stable decomposition

- Given x, $hash(x)$ is a positive int
- $level(x)$ is the largest i s.t. $hash(x)$ is a multiple of 2^i

e.g., $hash(x) = 01101\underbrace{000}$

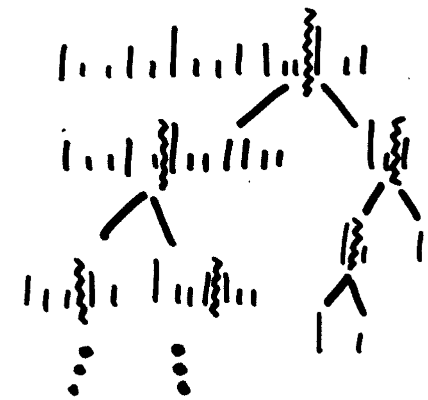
$level(x) = 3$

- Require: $\frac{1}{2}$ of all elts. are level 0 (avg.)

  $\frac{1}{4}$ of all elts. are level 1 (avg.)

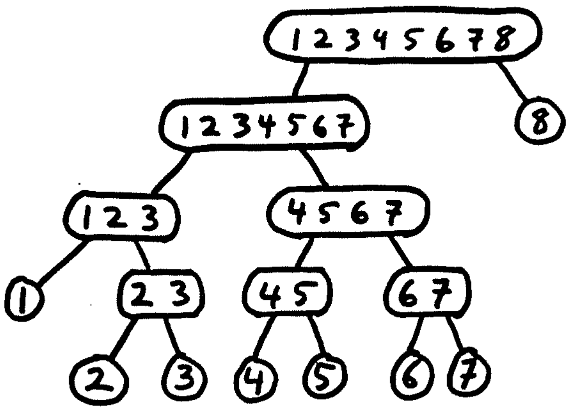
  $\vdots$

Stable decomposition

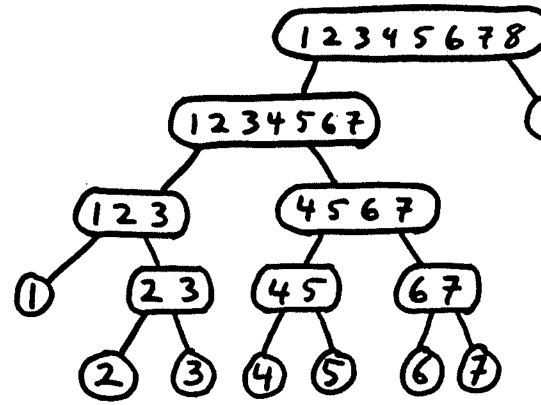


$\Rightarrow$ with high probability there will be only $O(\log n)$ cache misses.

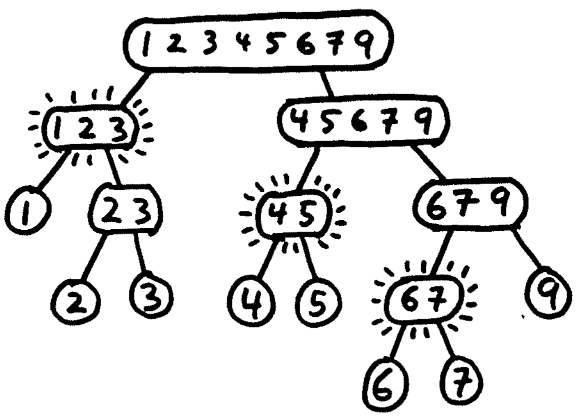
1	2	3	4	5	6	7	8
001	010	011	100	101	110	111	1000
0	1	0	2	0	1	0	3



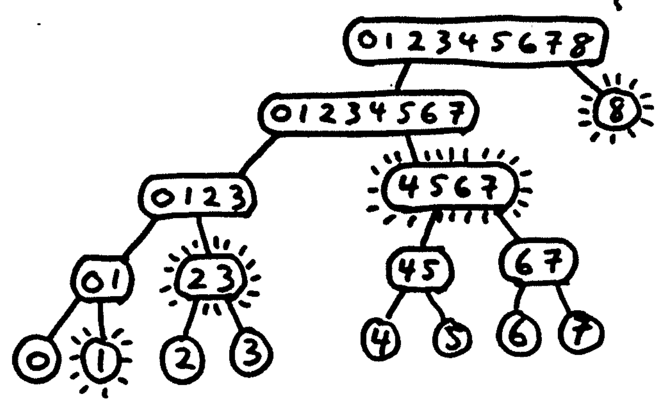
1	2	3	4	5	6	7	8
001	010	011	100	101	110	111	1000
0	1	0	2	0	1	0	3



1	2	3	4	5	6	7	9
0	1	0	2	0	1	0	0



0	1	2	3	4	5	6	7	8
32	0	1	0	2	0	1	0	3



Dynamization

Solution for static problem
[queries only]

↓

Solution for dynamic problem
[queries + insert + delete]

(semi-dynamic = queries + insert)

Search problems

Query: point × set → answer

Ex. 1: Membership in a set

Member: element × set → Boolean

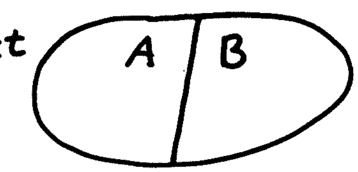
Ex. 2: Nearest neighbor

Nearest: point × point-set → distance

Decomposable Search Problem

Query: point × set → answer

$S \in$ set



$\square$: answer × answer → answer
computable in constant time

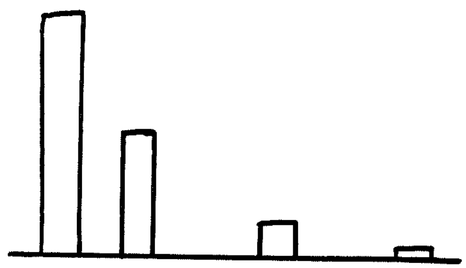
$$\text{Query}(x, S) = \square(\text{Query}(x, A), \text{Query}(x, B))$$

Block Partitioning

Let $|S_{current}| = n$

Expand n in binary

e.g. 1 1 0 1 0 1



$\leq \log_2 n$ static structures

Space: $S_{Dyn}(n) = O(S_{Static}(n))$

Query time: at most $\log n$ lookups

$$Q_{Dyn}(n) = O(Q_{Static}(n) \log n)$$

Insertion time (amortized):

$$\left(\frac{\text{Time to build static structure of size } n}{n}\right) \log n$$

On each insertion, restructure according to expansion of $|S_{current}|$ in binary.

Tie-Ins I

- Priority-ordering + change-propagation could be used for scheduling in INC
- Caching and dynamization could be used for "reduce" operations in INC
- Passing of Δ's in INC similar to passing of Δ's between tools (when integrating tools using the control-integration paradigm)

Tie-Ins II: Caching versus Blocking

Caching: $f(L) = \square(f(A), f(B))$

Blocking: $Q(x, C) = \square(Q(x, A), Q(x, B))$

$$g(C) = \square(g(A), Q(B))$$

Blocking: $\square$ associative and commutative

Caching: $\square$ associative but not necessarily commutative

Talk Outline

Introduction

Assessment of incremental algorithms

Graph-annotation problems

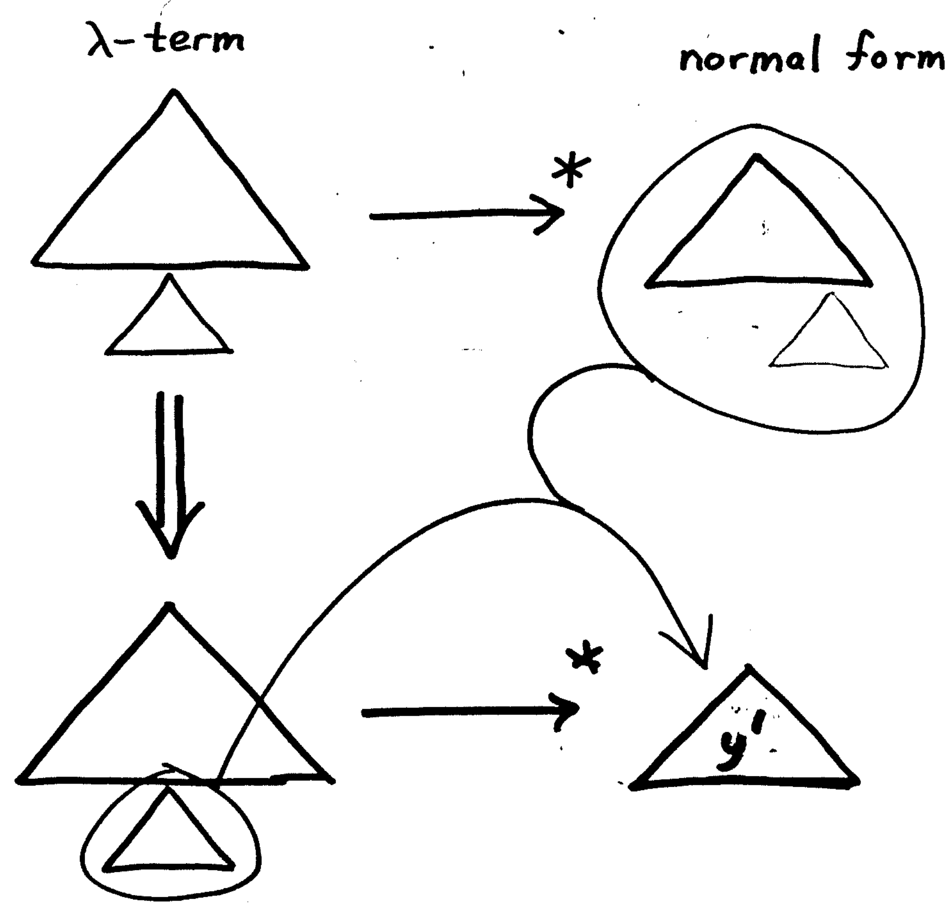
Other update problems

"Incrementalizers"
• Incremental rewriting • Alphonse • Incremental computation via partial evaluation

Conclusions

Incremental Rewriting

[Field & Teitelbaum – LFP 1990]

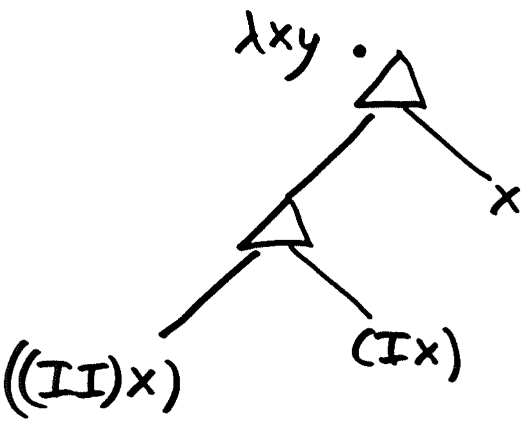


Fork Nodes

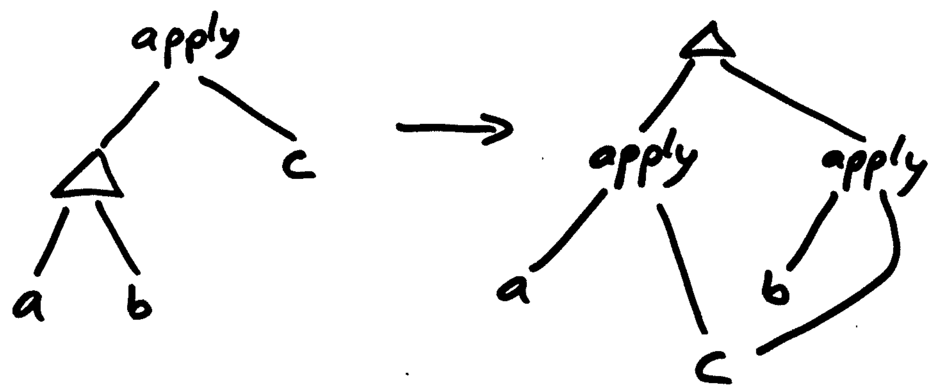
$\lambda x\, y.\ ((I\ I)x)$

$\lambda x\, y.\ (I\ x)$

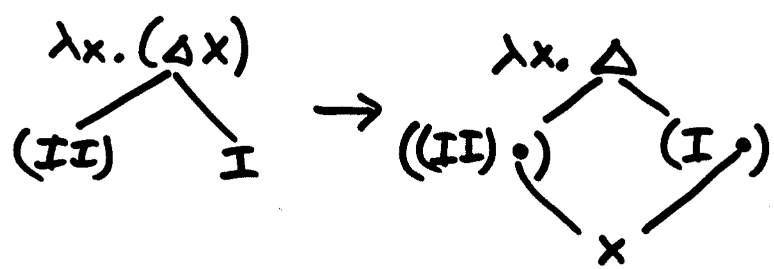
$\lambda x y.\ x$



Distribution Rule for Δ



Ex:



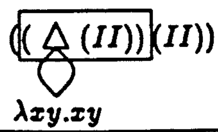
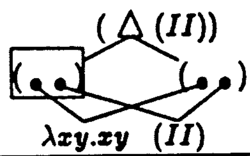
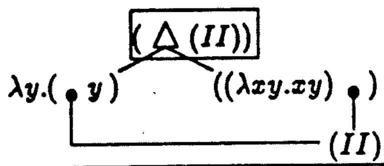
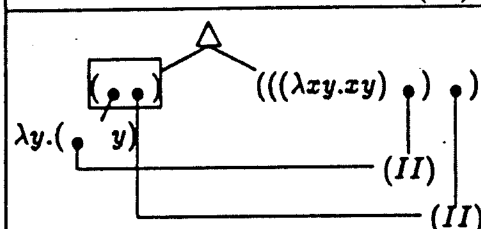
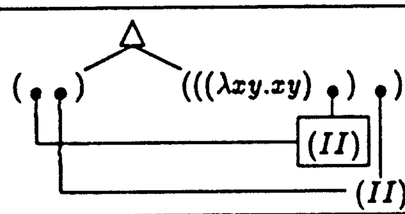
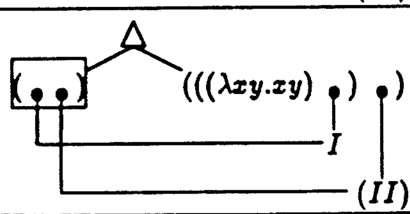
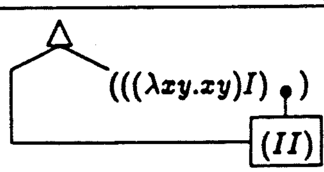
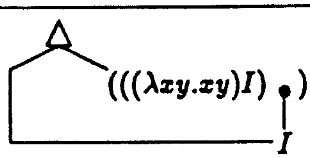
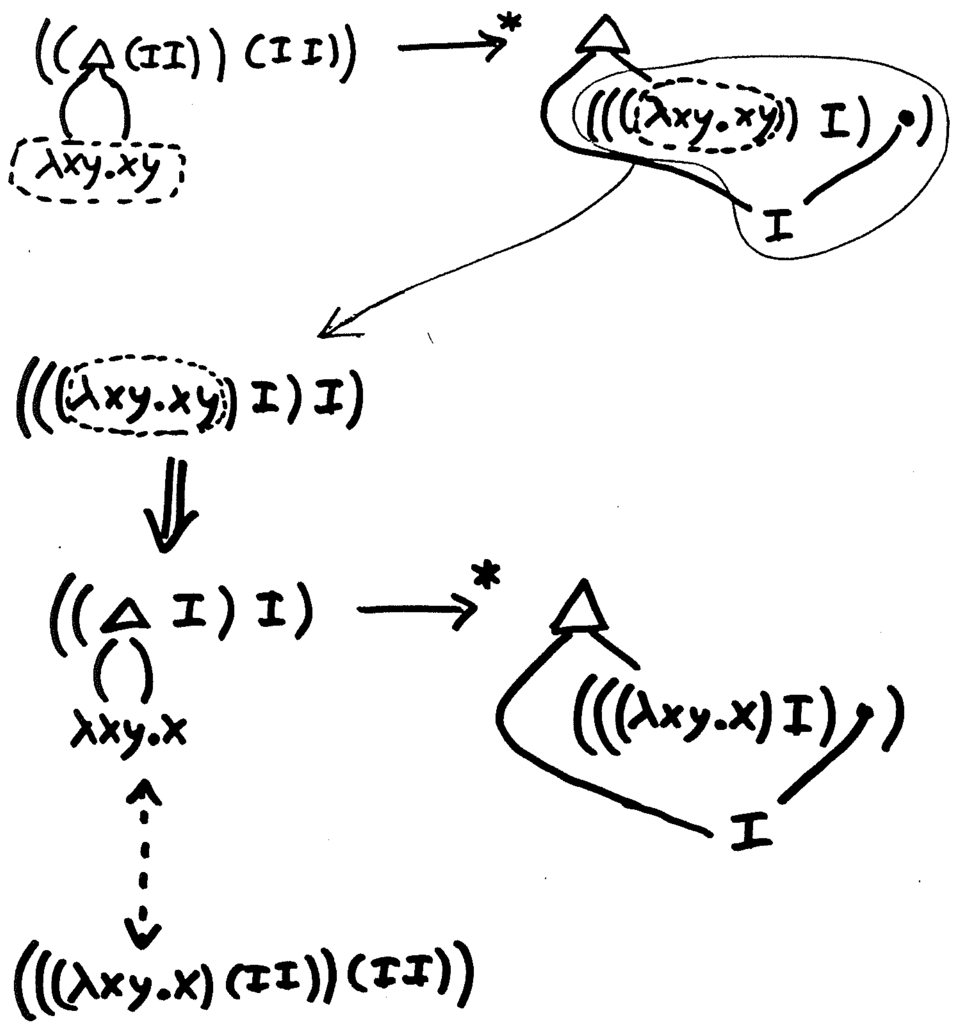
Graph	Op.	Corresponding Terms	
		Left Δ-Subterms	Right Δ-Subterms
	dist.	$((\lambda xy.xy)(II)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((\lambda xy.xy)(II)(II))$	$((\lambda xy.xy)(II)(II))$
	dist.	$((\lambda y.(II)\,y)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((\lambda y.(II)\,y)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((II)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$(I\,(II))$	$((\lambda xy.xy)\,I\,(II))$
	β	(II)	$((\lambda xy.xy)\,I\,(II))$
	β	I	$((\lambda xy.xy)\,I\,I)$

Figure 8.1: Reduction of M_1

Incremental Rewriting: Example



Binary Search Tree

```
TYPE Tree = OBJECT
   left, right: Tree;
   val: INTEGER;
METHODS    insert := Insert;
           delete := Delete;
           lookup := Lookup;
END;

TYPE TreeNil = Tree OBJECT
OVERRIDES insert := InsertNil;
          delete := DeleteNil;
          lookup := LookupNil;
END;

PROCEDURE Insert(t: Tree, v: INTEGER): Tree =
   BEGIN
      IF v < t.val THEN t.left := Insert(t.left, v)
      ELSIF v > t.val THEN t.right := Insert(t.right, v)
      END;
      RETURN t;
   END;

PROCEDURE InsertNil(t: TreeNil, v: INTEGER): Tree =
   BEGIN
      n := NEW(Tree);
      n.left := NEW(TreeNil); n.right := NEW(TreeNil);
      n.val := v;
      RETURN n;
   END;
```

AVL Tree

```
TYPE Avl = Tree OBJECT
METHODS    <* MAINTAINED EAGER *> height := Height;
           <* MAINTAINED DEMAND *> balance := Balance;
END;

TYPE AvlNil = Avl OBJECT
OVERRIDES <* MAINTAINED EAGER *> height := HeightNil;
          <* MAINTAINED DEMAND *> balance := BalanceNil;
END;

PROCEDURE Height(t: Avl): INTEGER =
  BEGIN RETURN MAX(t.left.height(), t.right.height()) + 1 END;

PROCEDURE HeightNil(t: Avl): INTEGER = BEGIN RETURN 0 END

PROCEDURE Balance(t: Avl): Avl =
  BEGIN
    t.left := t.left.balance();
    t.right := t.right.balance();
    IF Diff(t) > 1 THEN
      IF Diff(t) = 2 AND Diff(t.left) = -1 THEN
        t.left := RotateLeft(t.left)
      END;
      t := RotateRight(t).balance()
    ELSIF Diff(t) < -1 THEN
      IF Diff(t) = -2 AND Diff(t.right) = 1 THEN
        t.right := RotateRight(t.right)
      END;
      t := RotateLeft(t).balance()
    END;
    RETURN t;
  END;

PROCEDURE BalanceNil(t: Avl): Avl = BEGIN RETURN t END;
```


Alphonse

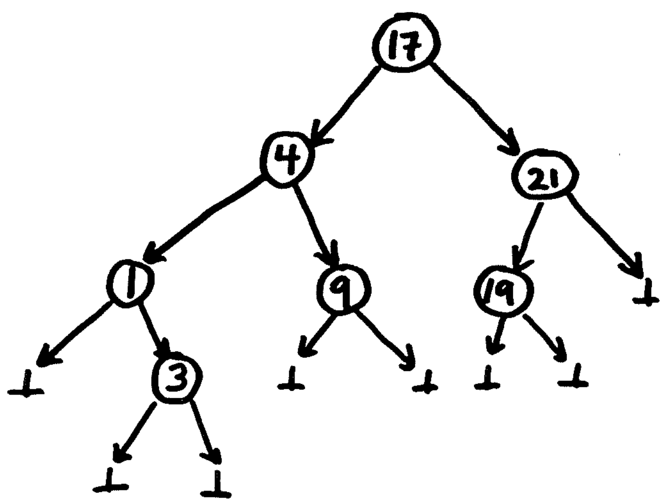
- Declarative
  - Invariants specified by imperative code—conceptually invoked after each modification
  - Must argue that side effects in eagerly maintained methods cannot affect observed output
- Introduces code to build and maintain an (acyclic) dependence graph
  - code templates for

    access(loc)

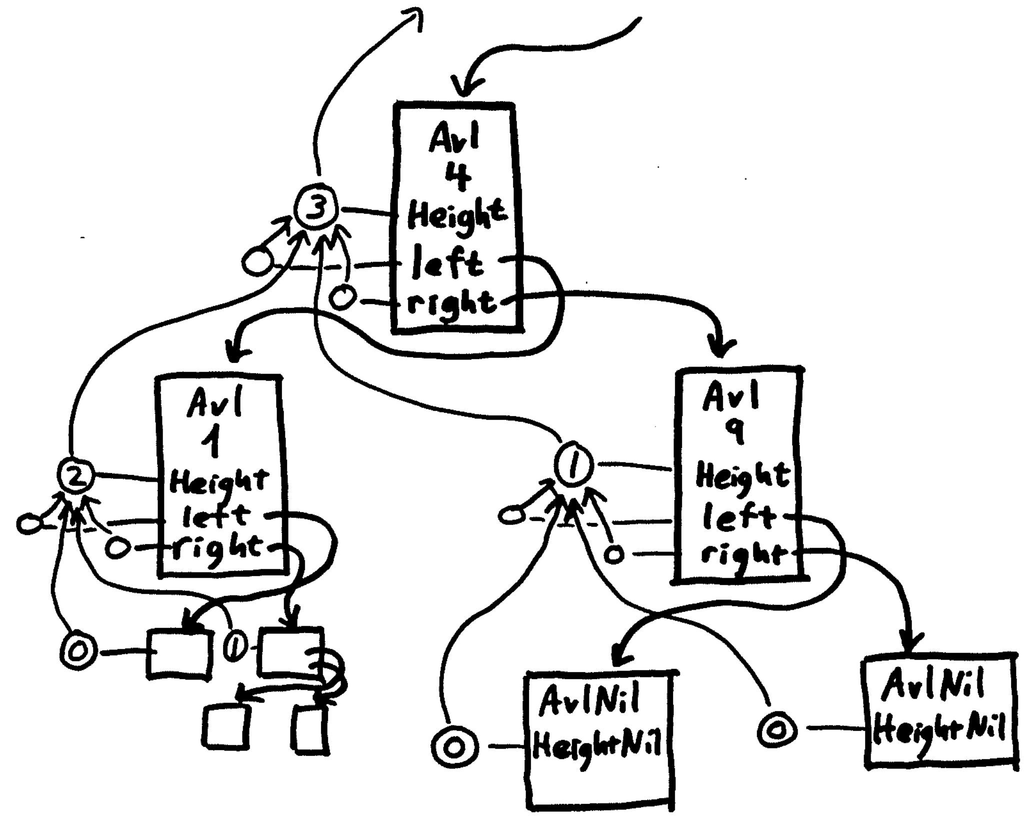
    modify(loc, val)

    call(p, a_1, $\cdots$, a_k)
- Maintains values that user requests via change propagation combined with function caching

[Hoover – PLDI 1992]



Dependence Graph

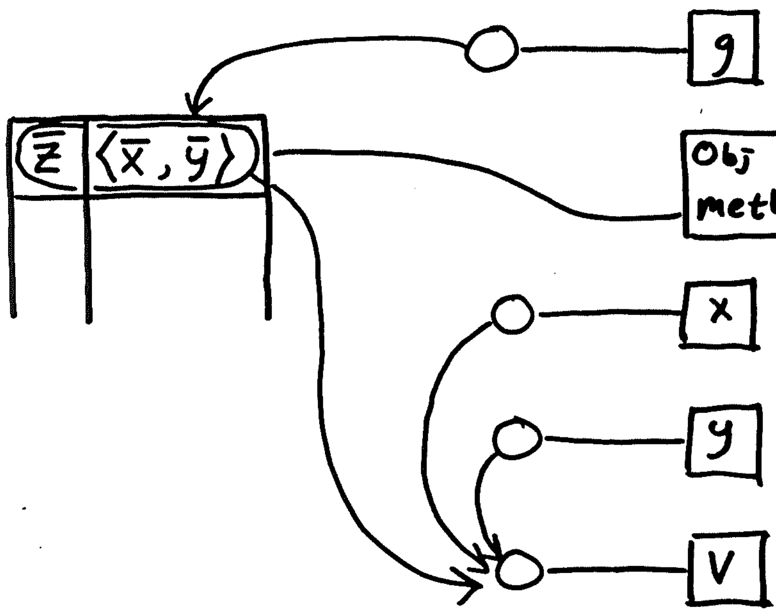


Caching in Alphonse

```
proc V (...)
   ⋮
   Obj.meth(x,y)
```

```
type ObjType =
   meth := p
```

```
proc p(o,a,b)
   ⋮
 ...g...
```

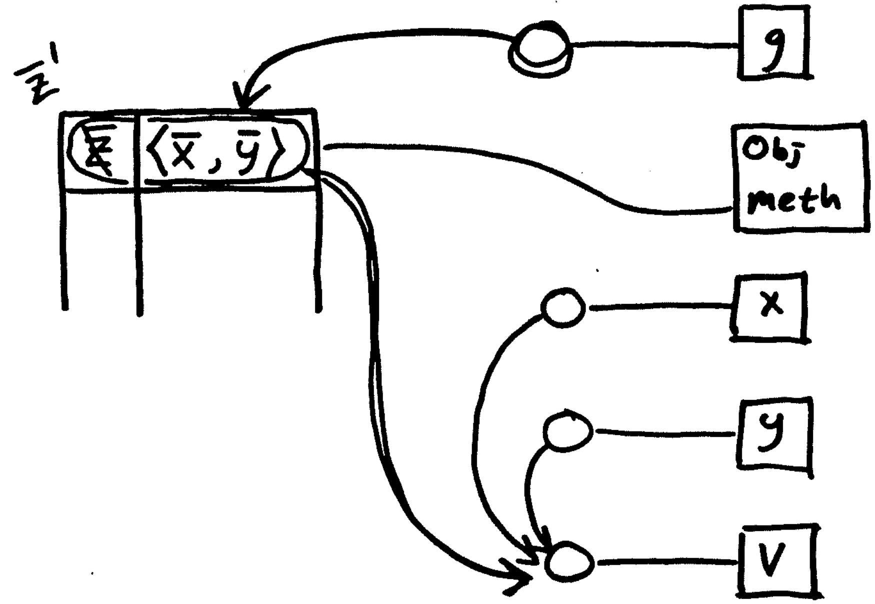


Caching in Alphonse


```
proc V (...)
   ⋮
   Obj.meth(x,y)
```

```
type ObjType =
   meth := p
```

```
proc p(o,a,b)
   ⋮
 ...g...
```

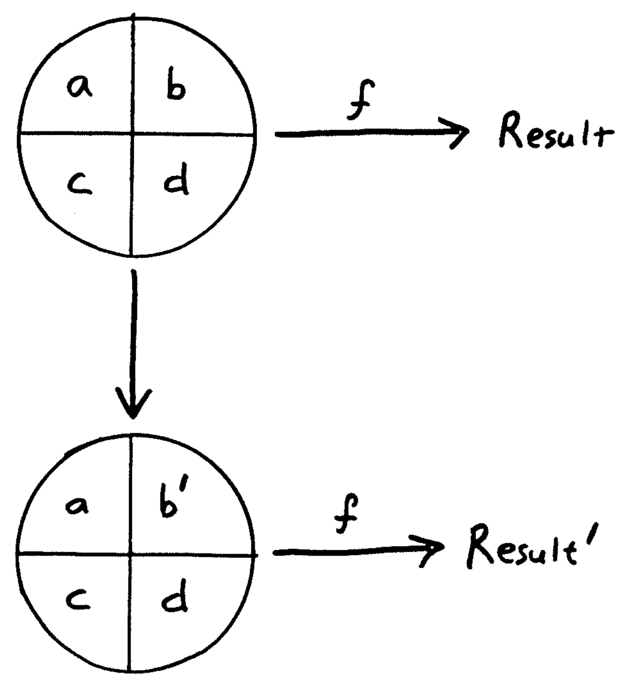



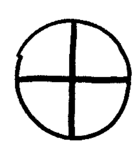
Alphonse Checklist

- Batch/incr. states: —
- Independence: via dependence graph
- Quiescence: via change propagation
- Balancing: not built in
  - Programmer may use explicit balancing
- Auxiliary information: dependence graph
- When better to recompute from scratch?
  - Depends on overhead
- Assessment criteria
  - Internal structures $O(\|\delta_{po}\| \log \|\delta_{po}\|) + O(\|\delta_{vals}\| \log \|\delta_{vals}\|)$
  - Pragmatic—Alphonse extends what is easily expressible
  - In practice?
- Generality: ++

Incremental Computation via Partial Evaluation

[Sundaresh & Hudak - POPL 1991]





defined with projections

Projections

Allow forming residuals w.r.t. data other than the first component.

$proj : D \rightarrow D$

$proj \sqsubseteq ID$ — no information addition

$proj \circ proj = proj$ — idempotence

Without projections:

$$\ell_{d_1} = P\ p\ \langle \ell, d_1 \rangle$$

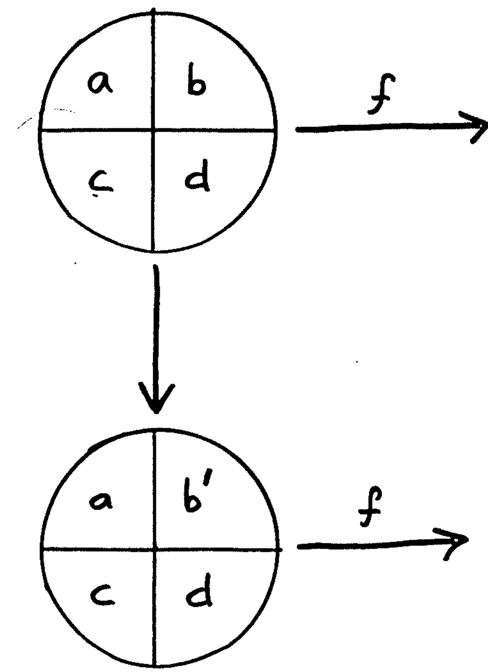
$$L\ \ell \langle d_1, d_2 \rangle = L\ \ell_{d_1}\ d_2$$

Using projections:

$$\boxed{\begin{aligned} \ell_{proj,a} &= P\ p\ \ell\ proj\ a \\ L\ \ell\ a &= L\ \ell_{proj,a}\ (apply\ \overline{proj}\ a) \end{aligned}}$$

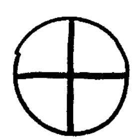
Incremental Computation via Partial Evaluation

[Sundaresh & Hudak – POPL 1991]



Result $= \bigsqcup_{i \in \{a,b,c,d\}} \left(L\ f_i\ proj_i\ (a \otimes b \otimes c \otimes d) \right)$

Result' $= \bigsqcup_{i \in \{a,b',c,d\}} \left(L\ f_i\ proj_i\ (a \otimes b' \otimes c \otimes d) \right)$



defined with projections

Talk Outline

Introduction

Assessment of incremental algorithms

Graph-annotation problems

Other update problems

"Incrementalizers"

Conclusions

Other Work

- Incremental data-flow analysis
- Dynamization of graph problems
- Finite differencing
- Incremental parsing
- Functional algebra
- Truth maintenance
- Incremental deduction
- Incremental constraint solving
- Document preparation

Implementation Frameworks

- Lotus 1-2-3
- TK!Solver
- The Synthesizer Generator
- Pan
- Centaur
- RAPTS
- ThingLab
- INC (?)
- Alphonse (?)

Unresolved Issues

- Use (and maintenance) of summary information
  - use of pointers and auxiliary storage
  - [Sairam, Vitter, & Tamassia – STACS 93]
- Models for lower bounds other than IRLBs and local persistence
  - persistent auxiliary storage
  - non-local pointers
- "Sparseness"
  - storage: only maintain certain values
  - time: only consistent at certain times
- Empirical studies of incremental algorithms
- Dissemination
  - library package
  - full-blown language (I/O, window systems, interface to existing languages)