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--------- |

\[ x = [8, 3, 5, 4, 5, 2, 5, 3, 6, 2, 7] \]
\[ y = f(x) = [8, 12, 5, 10, 16, 2, 8, 12, 6, 9, 7] \]
\[ f: \quad y[1] = x[1] \]
\[ y[i] = \text{let } v = y[i-1] + 1 + x[i] \text{ in} \]
\[ \text{if } v > 16 \text{ then } x[i] \text{ else } v \text{ fi} \]

\[ x = [8, 3, 5, 4, 6, 3, 5, 3, 6, 2, 7] \]
\[ y = [8, 12, 5, 10, 6, 10, 16, 3, 10, 13, 7] \]
Independence: Research has shown that
Quiescence: quicker

Dependence graph:

\[
g(x, \theta) = \begin{cases} 
\text{let } v = \theta + 1 + x \text{ in} \\
\text{if } v > 16 \text{ then } x \text{ else } v \end{cases} 
\]

\[
\text{Summing a List of Numbers}
\]

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

\[
f(x) = 205
\]

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

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

\[
\text{Summary information}
\]

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

\[
\begin{align*}
18 & \downarrow \quad 20 \quad 33 \\
18 \quad 38 & \downarrow \quad 205 \quad 89 \\
& \downarrow \quad 92 \quad 173 \quad 188 \\
& \downarrow \quad 193
\end{align*}
\]
**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**

begin
  declare a, a <- duplicate, c;
  b <- undeclared := c
end

**Optimization**

**Strength reduction** [Cocke]

\[
\begin{align*}
k &:= 0 \\
i &:= 1 \\
twoi &:= 2 \times i \\
\text{while } i \leq N \text{ do} \\
  k &:= k + a[2 \times i] \\
i &:= i + 1 \\
\text{od} \\
k &:= k + a[twoi] \\
i &:= i + 1 \\
twoi &:= twoi + 2 \\
\text{od}
\end{align*}
\]

**Finite differencing** [Earley, Feng, Bullman, Paige]

\[
\begin{align*}
S &:= \emptyset \\
F &:= \{s \in S \mid \text{even}(s)\} \\
\text{forever do} \\
  E &:= \{s \in S \mid \text{even}(s)\} \\
  \text{if } |E| = N \text{ then break} \\
x &:= \text{arb } T \\
T &:= T - \{x\} \\
S &:= S \cup \{x\} \\
\text{od} \\
E &:= F \\
\text{if } |E| = N \text{ then break} \\
x &:= \text{arb } T \\
T &:= T - \{x\} \\
S &:= S \cup \{x\} \\
F &:= F \cup \{x\} \text{ if even}(x) \text{ then } \{x\} \\
\text{else } \emptyset \\
\text{od}
\end{align*}
\]
Tool Integration via Control Integration

- 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
  \[ \forall x, \Delta^-, \Delta^+ \]
  \[ T_{Inc}(x, \Delta^-, \Delta^+) = O(g(|(x - \Delta^-) + \Delta^+|)) \]
  \[ = o(f(|(x - \Delta^-) + \Delta^+|)) \]
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
  \[ \forall x, \Delta^-, \Delta^+ \]
  \[ T_{Inc}(x, \Delta^-, \Delta^+) \leq T_{Batch}(|(x-\Delta^-)+\Delta^+|) \]
  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
  \[ \forall x, \Delta^-, \Delta^+ \]
  \[ T_{Inc_1}(x, \Delta^-, \Delta^+) \leq T_{Inc_2}(x, \Delta^-, \Delta^+) \]

- 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
  \[ \forall x, \text{seq} \rightarrow T_{On-line}(x, \text{seq}) \leq k \times T_{Off-line}(x, \text{seq}) \]

Competitive Ratio

Requests:
  Let's go skiing

Actions:
  Rent: $1
  Buy: $S
  Use skis already purchased: $0

On-line algorithm: \( R, R, R, \ldots, R, B, U, U, \ldots, U \)

For \( t \) requests:
  \[ \text{Con-line} = \begin{cases} t & \text{if } t \leq k \\ k+1 & \text{otherwise} \end{cases} \]

Best for adversary: \( L^{k+1}, \text{"so-long sucker"} \)
  \[ \Rightarrow R^k, B, \text{"darn"} \]

Best for C.R.: choose \( k \) to minimize
  \[ \frac{\text{Con-line}}{\text{Off-line}} = \frac{k+S}{\min(S, k+1)} \Rightarrow k = S-1 \]

Competitive Ratio: \( \frac{2S-1}{S} \)
Incremental Computation vs. On-Line Computation

I.C. -- functional view

\[ x \xrightarrow{f} f(x) \]
\[ x + \Delta x_i \xrightarrow{f} f(x + \Delta x_i) \]

* I.C. as O.L.C.
  One operation: Modify($S$)
  One query: What are the changes in output?
  MQ MQ MQ MQ ...

* Complementary views
  - Functional view suggests some specific techniques (e.g. function caching)
  - OLC suggests broadening the problem
    MMM MQ MQ MMM MQ MQ ...
    [Cohen & Tamassia - SODA 91]

Boundedness

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

\[ O(f(|input|)) \]

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)

\[ \text{IRLB} \]

\[ \text{(\# modifications needed to "achieve" a batch evaluation)} \]

\[ \text{e.g. } \text{IRLB (SSSP > 0)} = O(1) \]

\[ \text{IRLB} \times \text{Batch lower bound} = \text{Incr. lower bound} \]

\[ \text{Sorting: } \]

\[ O(\frac{1}{n}) \times \Omega(n \log n) = \Omega(\log n) \]

\[ \text{SSSP > 0: } \]

\[ O(1) \times \Omega(?) = \Omega(?) \]

\[ \text{Requirement: "fast initialization" e.g. sorting: empty sequence} \]

\[ \Rightarrow \text{limited initial auxiliary storage} \]

[\text{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

\[ \frac{1}{\sqrt{n}} \geq \frac{1}{n} : \]

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

**IRLB vs. Boundedness**

<table>
<thead>
<tr>
<th>Problem</th>
<th>IRLB</th>
<th>( f(|S|) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP &gt; 0</td>
<td>( O(1) )</td>
<td>( O(|S| \log |S|) )</td>
</tr>
<tr>
<td>APSP &gt; 0</td>
<td>( O(1) )</td>
<td>( O(|S| \log |S|) )</td>
</tr>
<tr>
<td>Dominators</td>
<td>( O(1) )</td>
<td>Unbounded</td>
</tr>
<tr>
<td>Minimum spanning tree</td>
<td>( \frac{1}{\sqrt{n}} \geq \frac{1}{n} )</td>
<td>Unbounded</td>
</tr>
</tbody>
</table>
Beating an IRLB: String Matching

Pattern: $aba$

String: $cabbbababaaa$

IRLB = $\frac{1}{m}$

IRLB $\times$ BLB = ILB

$\gamma_m \times \Omega(n) = \Omega(n/m)$

$cabbbabababaaq$

$\langle 2, 5 \rightarrow 7 \rangle$

$O(m)$ 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

<table>
<thead>
<tr>
<th>Graph-annotation problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Boundedness</td>
</tr>
<tr>
<td>• Problems on acyclic graphs</td>
</tr>
<tr>
<td>• Problems on graphs with cycles</td>
</tr>
<tr>
<td>• Unboundedness</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
<th>Quantity</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>pen</td>
<td>.95</td>
<td>3</td>
<td>2.85</td>
</tr>
<tr>
<td>paper</td>
<td>1.50</td>
<td>2</td>
<td>3.00</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>5.85</td>
</tr>
</tbody>
</table>

\[ C_{pen, \text{price}} \times C_{pen, \text{quantity}} = C_{pen, \text{total}} \]

\[ C_{paper, \text{price}} \times C_{paper, \text{quantity}} = C_{paper, \text{total}} \]

\[ C_{pen, \text{total}} + C_{paper, \text{total}} = C_{total, \text{total}} \]
### Differential Updating

<table>
<thead>
<tr>
<th>Item</th>
<th>Price ((=.95 -.20))</th>
<th>Quant.</th>
<th>Total ((= 2.85 -.60))</th>
</tr>
</thead>
<tbody>
<tr>
<td>pen</td>
<td>.75</td>
<td>3</td>
<td>2.25</td>
</tr>
<tr>
<td>paper</td>
<td>1.50</td>
<td>2</td>
<td>3.00</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>5.25 ((= 5.85 -.60))</td>
</tr>
</tbody>
</table>

\[
C_{\text{new}} \text{ pen, price} - C_{\text{old}} \text{ pen, price} = \Delta C_{\text{pen, price}}
\]

\[
\Delta C_{\text{pen, price}} \times C_{\text{pen, quantity}} = \Delta C_{\text{pen, total}}
\]

\[
\Delta C_{\text{pen, total}} = \Delta C_{\text{total, total}}
\]

\[
C_{\text{pen, total}} := C_{\text{pen, total}} + \Delta C_{\text{pen, total}}
\]

\[
C_{\text{total, total}} := C_{\text{total, total}} + \Delta C_{\text{total, total}}
\]
"Size" of a Change

- MODIFIED$G,s$
- AFFECTED$G,s$
  - Not known a priori
- CHANGED$G,s = \text{def }$ MODIFIED$G,s$
  u AFFECTED$G,s$

- Defn: $\|S\| = \|\text{CHANGED}_{G,s}\|_{G+s}$
  - characterizes updating costs inherent to a problem (rather than costs of a given algorithm for the problem)

- Goal: $O(f(\|S\|))$

- $K$ : vertex set
  - $|K| = 5$
- $N(K)$ : neighborhood of $K$
  - $|N(K)| = 12$
- $<K>$ : induced graph
  - $|<K>| = 11$
- $<N(K)>$ : induced graph
  - $|<N(K)>| = 27$
- $\|K\| = \text{def } |<N(K)>|$ "extended size"
Propagate \((G, S)\)

**precondition:** \(S = \{\text{inconsistent nodes of } G\}\)

```
begin
while \(S \neq \emptyset\) do
    Select and remove a node \(v\) from \(S\)
    oldvalue := val[v]
    Reevaluate \(v\)
    if oldvalue \neq val[v] then
        \(S := S \cup \{\text{successors of } v\}\)
    fi
end
```
Behavior of Change Propagation

\[ T(h) = 2T(h-1) + k \]

\[ T(h) = \Theta(2^h) \]
Equations for Name Analysis

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

\[ \text{DeclaredIn}_{\text{Decl}} = \{a, b, c\} \]

\[ \text{DeclaredFor}_{\text{Stmt}} = \text{DeclaredIn}_{\text{Decl}} \]

\[ b \in \text{DeclaredFor}_{\text{Stmt}} \]

\[ c \in \text{DeclaredFor}_{\text{Stmt}} \]
Defining Static Inferences

\[ \text{decls}_2 \cdot \text{in} = \{ \text{id.name} \} \cup \text{decls}_1 \cdot \text{in} \]
\[ \text{decls}_1 \cdot \text{out} = \text{decls}_2 \cdot \text{out} \]

require id name \notin \text{decls}_1 \cdot \text{in}

Attribute grammar [Knuth]

Subtree Replacement

Consistent Tree \[\Rightarrow\] Inconsistent Tree
Optimal Updating

Inconsistent Tree $\Rightarrow$ Consistent Tree

::: AFFECTED

Updating cost: $O(|AFFECTED|) = O(11811)$

[Reps - POPL 82]

production:

plan:

<table>
<thead>
<tr>
<th>Eval</th>
<th>Visit</th>
<th>Eval</th>
<th>Visit</th>
<th>Eval</th>
<th>Visit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$X_2$</td>
<td>$X_2$</td>
<td>$X_0$</td>
<td>$X_0$</td>
</tr>
</tbody>
</table>
CABLES & BOXES [Alpern et al. PE 1988]

Box:

Cable:

duct

Substrate graph: Boxes + Cables
Dependence graph: Attributes + Edges
Example: PERT Chart

\[
\text{link: CABLE \{ time\_to\_completions: BLUE\rightarrow RED \}}
\]
\[
\text{source: BOX \{ critical\_path\_length: LOCAL REAL \}}
\]
\[
\text{\hspace{1cm} out: \ast RED link}
\]
\[
\text{\hspace{1cm} critical\_path\_length += max(out.time\_to\_comp)}
\]
\[
\text{\}}
\]
\[
\text{task: BOX \{ out: \ast RED link}
\]
\[
\text{\hspace{1cm} in: \ast BLUE link}
\]
\[
\text{\hspace{1cm} in.time\_to\_completion += 1 + max(out.time\_to\_comp)}
\]
\[
\text{\}}
\]
\[
\text{sink: BOX \{ in: \ast BLUE link}
\]
\[
\text{\hspace{1cm} in.time\_to\_completion = 0.0}
\]
\[
\text{\}}
\]

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 auxiliary 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

$|S| = O(1)$

$|S| = O(\log n)$

$O(\log n)$

$\Delta_1$ is just one of $O(n)$ possible modifications

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

$\Delta_1$ at leaf,
$\Delta_2$ at root of a complete binary tree of height $O(\log n)$
Circuit Annotation is Bounded
[Ramalingam & Reps 1991,92]

Repeatedly:
1. Re-evaluate Workset in relative topological sort order
2. Expand all vertices of Inner Fringe

Work: $O(2^{\|S\|})$
Priority Ordering

Evaluation cost: \( O(n \log n) \)
Updating cost: \( O(|S| \log |S|) \)
Finding a Cover

Lockstep: \( \leq 2 \times \| \text{min. cover} \| \)

[Alpern et al. – SODA 90]

Updating Priorities

Priority space [Dietz & Sleator STOC 87]
Not locally persistent
Circuit Annotation: Updating Via Priorities

1. Update priorities
   \( O(\|\Delta p_{\text{poll}}\| \log \|\Delta p_{\text{poll}}\|) \)

2. Change propagation
   \( O(\|\Delta \delta_{\text{Values}}\| \log \|\Delta \delta_{\text{Values}}\|) \)

   Unbounded
   vs.

   Bounded: \( O(2^\|\Delta \delta_{\text{Values}}\|) \)

Observations

- Relation vs. function
- Algorithm uses persistent auxiliary storage
- Not candidate for amortized analysis
- Competitive?
(Attribute dependence graph of a Pascal program)
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

Polynomial

Circuit-Annotation Problem

Priority Ordering*
  - All-Pairs Shortest Path (≥ 0)
  - Single-Source Shortest Path (≥ 0)

Attribute Updating

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

Dijkstra's (batch) algorithm:

[Diagram of Dijkstra's algorithm with priority ordering and examination of the graph]
Single-Sink Shortest-Path Problem (with positive edge weights)

DeleteEdge(G, v → w)

Phase 1:

\[ \text{WorkSet} := \{v\} \]
\[ \text{AffectedVertices} := \emptyset \]

while WorkSet ≠ \emptyset do

Select and remove a vertex u from WorkSet

Insert u into AffectedVertices

for each red edge x → u do

uncolor x → u

if \( \text{\!(if | a red edge x → y then)} \]

Insert x into WorkSet

\( \text{\!)} \]

Phase 2:

Determine new distances in induced graph < N(AffectedVertices) >
(e.g., via Dijkstra's algorithm)
**Complexity of Delete Edge**

Phase 1: $O(|S|)$

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

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

[Ramalingam & Reps 1991]

![Diagram of a network with nodes u, v, w, and sink, illustrating the edge insertion complexity.](image)

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

(*) $\text{dist}(u,v) + \text{length}(v \rightarrow w) + \text{dist}(w) < \text{dist}(w)$?

Idea: Perform (most of) the batch single-sink problem with sink $V$ (but only visit vertices for which (*) holds -- and their predecessors)
Reachability is Unbounded
[Ramalingam & Reps 1991]

Unbounded

- 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

Exponential

- Circuit-Annotation Problem

Polynomial

- Priority Ordering*
- All-Pairs Shortest Path (> 0)
- Single-Source Shortest Path (> 0)

- Attribute Updating

\[ |S| = O(1) \]
\[ |S| = O(n) \]
(Meet-Semilattice) Data-Flow Problems

G: flow graph
s: entry vertex of G
L: semilattice (with T)
M: edge \( \rightarrow \) L \( \rightarrow \) L  
(labels edges with flow functions)
c \( \in \) L: constant associated with s

The solution is the maximal fixed point of:

\[ S(s) = c \]
\[ S(u) = \prod_{v \in \text{pred}(u)} M(v \rightarrow u) (S(v)) \]
There Are Unbounded Data-Flow Problems

Instance of S-S Reachability:

$S(t) = c \neq T$
$M(e) = \lambda x.x$

Instance of data-flow problem $P$:

$S(u) = f(c)$ if $u$ is reachable from $s$
$S(u) = T$ if $u$ is not reachable
Reaching Definitions

source

∅

p: x := 1

∅

∅ → \{<p,x>\}

∅

∅ → \{<p,x>, ...\}

r:

∅ → \{<p,x>, ...\}

s: while C

∅ → \{<p,x>, ...\}

t:

∅ → \{<p,x>, ...\}

Reachability and SSSP > 0

Diagram of a graph with labeled nodes and edges.
Reductions Between Problems

1. Give transformations:
   a. Instance_p → Instance_q
   b. Solution_q → Solution_p
   c. δ_p → δ_q
   d. Δ Solution_q → Δ Solution_p

2. Show that (1)c and (1)d are bounded by a function of ||δ_p||

3. Show that ||δ_q|| is bounded by a function of ||δ_p||

4. Show that (1)c and (1)d are locally persistent.
Dominators in a Directed Graph

Vertex \( v \) dominates \( w \) iff all paths \( \text{source} \rightarrow^* w \) contain \( v \).

Data-flow problem:  
\[
M(v \rightarrow u) = \lambda x. \{ y \mid x \leq y \}
\]
\[
S(\text{source}) = \emptyset
\]
\[
S(u) = \bigcap M(v \rightarrow u)(S(v)) \
\text{reprod}(u)
\]

Identity function not allowed on an edge.
Dominators is Unbounded

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
  - $O(f(n, \|\delta\|))$

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

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

$\text{Apply}[1] B = [1, 1, 1, 1, 1, 1, 1]$

$\text{Reduce}[+](\text{Apply}[1] B) = 7$

Example: Bag Equality

$[1, 1, 2, 2] \quad [1, 1, 2, 2]$
Differential Updating in INC

- Produce smallest message
  \[ B_{old} = \{1, 2, \cdots, 50\} \]
  \[ B_{new} = \{49, 50, \cdots, 100\} \]
  \[ \Delta^+ = \{51, \cdots, 100\} \]
  \[ \Delta^- = \{1, 2, \cdots, 49\} \]
  \[ \text{Remainder} = \{49, 50\} \]

- \( T_{Inc} = O(T_{Batch}) \)
  at most a constant factor to maintain
  structures used during updating

Incremental Computation via Function Caching

[Pugh & Teitelbaum - POPL 1991]

Applicability: Decomposable list problems

\[ L = \text{All } B \]
\[ f(L) = \Omega(f(A), f(B)) \]

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)
\[
sl(L) = \begin{cases} 
\text{null}(L) \rightarrow L \\
\text{cons}(hd(L) + 2, sl(tl(L)))
\end{cases}
\]

Idea: Cache previous values

**Original problem**

\[
\begin{array}{c}
sl(1 2 3 4 5 6 7 8) \\
(1 4 9 16 25 36 49 64)
\end{array}
\]

**Change last elt.**

\[
\begin{array}{c}
sl(1 2 3 4 5 6 7 9) \\
(1 4 9 16 25 36 49 81)
\end{array}
\]
\[ ss(S) = \begin{cases} \text{if length}(S) = 1 & \text{then} \quad \text{hd}(S) \quad \text{nil} \\ \text{else} & \text{append}\left( ss(\text{first-half}(S)), ss(\text{second-half}(S)) \right) \end{cases} \]
Goal: Stable decomposition

* Given \( x \), \( \text{hash}(x) \) is a positive int
* \( \text{level}(x) \) is the largest \( i \) s.t. \( \text{hash}(x) \) is a multiple of \( 2^i \)
  
  E.g., \( \text{hash}(x) = 011010000 \)
  
  \( \text{level}(x) = 3 \)

* Require: \( \frac{1}{2} \) of all els. are level 0 (avg.)
  
  \( \frac{1}{4} \) of all els. are level 1 (avg.)

\[ \Rightarrow \text{with high probability there will be only } O(\log n) \text{ cache misses.} \]
**Dynamization**

Solution for static problem  
[queries only]  
\[
\downarrow
\]
Solution for dynamic problem  
[queries + insert + delete]  
(semi-dynamic = queries + insert)

Search problems  
Query: point \(\times\) set \(\rightarrow\) answer

Ex. 1: Membership in a set  
Member: element \(\times\) set \(\rightarrow\) Boolean

Ex. 2: Nearest neighbor  
Nearest: point \(\times\) point-set \(\rightarrow\) distance

**Decomposable Search Problem**

Query: point \(\times\) set \(\rightarrow\) answer

\[S \in \text{set}\]

\[\Box: \text{answer} \times \text{answer} \rightarrow \text{answer}\]

computable in constant time

\[\text{Query}(x, S) = \Box(\text{Query}(x, A), \text{Query}(x, B))\]
Block Partitioning

Let $|S_{\text{current}}| = n$

Expand $n$ in binary

e.g. $110101$

$\leq \log_2 n$

Static Structures

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

Query time: at most $\log n$ lookups

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

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 $|S_{\text{current}}|$ 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 $\Delta$'s in INC similar to passing of $\Delta$'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
Fork Nodes

\[ \lambda xy.((I I)x) \]
\[ \lambda xy.(I x) \]
\[ \lambda xy.x \]

Distribution Rule for \( \Delta \)

Ex: \( \lambda x.((I I)x) \) \( \rightarrow \) \( \lambda x.\Delta \)

\( (I I)I \) \( \rightarrow \) \( (I I)x \)
<table>
<thead>
<tr>
<th>Graph</th>
<th>Op.</th>
<th>Corresponding Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>dist.</td>
<td>(\text{(\lambda)zy.(\text{y})})(II)(II) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>(\beta)</td>
<td>(\text{(\lambda)zy.(\text{y})})(II)(II) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>dist.</td>
<td>(\text{(\lambda)y.(\text{(\text{y})})(II)})(II) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>(\beta)</td>
<td>(\text{(\lambda)y.(\text{y})}(II)(II) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>(\beta)</td>
<td>((I(II)) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>(\beta)</td>
<td>((I(II)) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>(\beta)</td>
<td>((I(II)) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
<tr>
<td>((\text{(\lambda)zy} \cdot \text{(\text{y})}))</td>
<td>(\beta)</td>
<td>((I(II)) \quad \text{(\lambda)zy.(\text{y})}(II)(II)</td>
</tr>
</tbody>
</table>

**Figure 8.1:** Reduction of \(M_1\)
**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]
Caching in Alphonse

\[ \text{proc } v(z) \]
\[ \text{type } \text{ObjType} = \text{meth } t = \rho \]
\[ \text{obj.meth}(x, y) \]

\[ \text{proc } p(0, a, b) \]
\[ ...g... \]

Caching in Alphonse

\[ \text{proc } v(z) \]
\[ \text{type } \text{ObjType} = \text{meth } t = \rho \]
\[ \text{obj.meth}(x, y) \]

\[ \text{proc } p(0, a, b) \]
\[ ...g... \]
**Alphonse Checklist**

- Batch/incremental 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]
Projections

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

\[
\text{proj}: D \to D
\]
\[
\text{proj} \subseteq \text{ID}
\]
\[
\text{proj} \circ \text{proj} = \text{proj}
\]

(idempotence)

Without projections:

\[
\begin{align*}
l_{d_1} &= P \ p \ l_{d_1} \\
l (l_{d_1}, d_2) &= L \ l_{d_1} d_2
\end{align*}
\]

Using projections:

\[
\begin{align*}
l_{\text{proj}, a} &= P \ p \ l_{\text{proj}, a} \\
l \ l a &= L \ l_{\text{proj}, a} (\text{apply } \overline{\text{proj}} a)
\end{align*}
\]

Incremental Computation via Partial Evaluation

[ Sundaresh & Hudak - POPL 1991 ]

\[
\begin{align*}
\text{Result} &= \bigsqcup_{i \in \{a,b,c,d\}} (L \ f_i \ \text{proj}_{i} (a \oplus b \oplus c \oplus d)) \\
\text{Result'} &= \bigsqcup_{i \in \{a,b,c,d\}} (L \ f_i \ \text{proj}_{i} (a' \oplus b' \oplus c' \oplus d'))
\end{align*}
\]

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)