Incremental Computation

COS 326

David Walker

Princeton University
data, data

→

recompute facts
Spreadsheets

PowerPoint Gantt Chart with Dependencies

Source
- ground
- rail
- sea
- air
- nonground

Route
- Eastern Hemisphere
- Australia
- Europe
- North America

Measurements:
- 276
- 550
- 212
- 500
- 3056

Package:
- 498
- 600
- 240
- 470
- 4050

Start:
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

WEEKS

GANTT CHART WITH DEPENDENCIES

ALGORITHNS
BY COMPLEXITY

MORE COMPLEX

LEFTPAD QUICKSORT
MERGE
GIT
SELF-DRIVING
CAR
GOOGLE
SEARCH
BACKEND

SPRAWLING EXCEL SPREADSHEET BUILT UP OVER 20 YEARS BY A CHURCH GROUP IN NEBRASKA TO COORDINATE THEIR SCHEDULING
This is your machine learning system?

Yup! You pour the data into this big pile of linear algebra, then collect the answers on the other side.

What if the answers are wrong?

Just stir the pile until they start looking right.
Computational Biology, DNA, and Mutation

- Before
- After

SARS-CoV-2

- Spike Glycoprotein (S)
- RNA and N protein
- Envelope
- M-Protein
- E-Protein
INCREMENTAL COMPUTING IN OCAMLR
Efficient Parallel Computations

Work(n) = \sim n additions to sum a vector of length n

Span(n) = \sim \log(n) additions – the length of the longest dependency chain
Efficient Parallel Computations

Work(n) = \sim n \text{ additions to sum a vector of length n}

Span(n) = \sim \log(n) \text{ additions} – the length of the longest dependency chain
Efficient Incremental Computations

out of date

+ 16
  + 9
    + 2
    + 8

change from 7 to 8

36

+ 7
  + 4
    + 3
    + 9
  + 17
    + 9
    + 8
    + 2
    + 1

+ 20
  + 3
Efficient Incremental Computations

out of date

16 + 7 + 10 + 2 = 36

20 + 17 + 3 + 8 + 2 + 1 = 53

change from 7 to 8
Efficient Incremental Computations

out of date

+ 17
+ 10
+ 2

+ 7
+ 4
+ 3

+ 20
+ 17
+ 9
+ 8
+ 2
+ 1

change from 7 to 8
Efficient Incremental Computations

Now up to date!
Work to recompute from scratch: $\sim n$
Work to recompute incrementally: $\sim \log n$

change from 7 to 8
**Similarity:** span (ie: length of the longest dependency chain) of a computation governs latency

**Difference:** we will do a parallel computation *once*. We will do an incremental computation *many times*.

- the parallel dependency graph was *implicit*
  - represented the series of function calls made in order
- the incremental dependency graphs will be *explicit*
  - we will need to create a data structure that stores the computation graph so it can be reused
Incremental Dependency Graphs

- **Nodes** have type 'a Inc.t
  - nodes store a current value with type 'a
- **Edges** are functions with type 'a -> 'b
  - if the argument 'a changes, the function recomputes 'b
Incremental Dependency Graphs

- **Nodes** have type `'a Inc.t`
  - nodes store a current value with type `'a`
- **Edges** are functions with type `'a -> 'b`
  - if the argument `'a` changes, the function recomputes `'b`
Nodes have type 'a Inc.t
- nodes store a current value with type 'a

Edges are functions with type 'a -> 'b
- if the argument 'a changes, the function recomputes 'b
Accessing Incremental Dependency Graphs

Sources of information have type 'a Var.t
You can change them.
Changes are propagated through the graph
Accessing Incremental Dependency Graphs

Sources of information have type 'a Var.t
You can change them.
Changes are propagated through the graph

Sinks have type 'a Obs.t
You can read them
let x = Var.create 3 in
let y = Var.create 7 in
let z = Inc.map2 (Var.watch x) (Var.watch y) ~f:(fun x y -> x + y) in
let z_o = Inc.observe z in
1. Create *initial sources* with Var.create

```plaintext
let x = Var.create 3 in
let y = Var.create 7 in
```
2. Create incremental nodes by *watching* sources for change.

Var.watch : 'a Var.t -> 'a Inc.t
3. Create new incremental nodes from existing incremental nodes by creating edges using map, map2, map3 ...

Inc.map : 'a Inc.t -> f:'a -> 'b) -> 'b Inc.t

let zi = Inc.map xi ~f:(fun x -> x + 1) in
3. **Create new incremental nodes** from existing incremental nodes by creating edges using map, map2, map3 ...

Inc.map2 : 'a Inc.t -> 'b Inc.t -> f:('a -> 'b -> 'c) -> 'c Inc.t

```
let ri = Inc.map2 zi yi ~f:(fun x y -> x + y) in
```
Building an Incremental Computation

4. Extract *observable* results from graph

```
let ro = Inc.observe ri
```

```
Inc.observe : 'a Inc.t -> 'a Obs.t
```

```
let ro = Inc.observe ri
```

```
11
```

```
ro : int Obs.t
```

```
11
```

```
ri : int Inc.t
```

```
4
```

```
+1
```

```
3
```

```
7
```

```
3
```

```
7
```

```
11
```

```
+1
```

```
3
```

```
7
```

```
4
```

```
7
```

```
3
```

```
7
```

```
11
```

```
ri : int Inc.t
```

```
let ro = Inc.observe ri
```
5. **Stabilize** (ie: push any pending changes through the graph)

Inc.stabilize : unit -> unit

```plaintext
Inc.stabilize ();
```

```
3 7
+1 4
+1
```

```
11
ri : int Inc.t
```

```
11
ro : int Obs.t
```

```
3
7
```
6. Get \textit{plain value} from observable after stabilizing.

\begin{align*}
\text{let } v &= \text{Obs.value\_exn \ ro} \\
\text{plain value } v &: \text{int}
\end{align*}
let x = Var.create 3 in
let y = Var.create 7 in
let xi = Var.watch x in
let yi = Var.watch y in
let zi = Inc.map xi ~f:(fun x -> x + 1) in
let ri = Inc.map2 zi yi ~f:(fun x y -> x + y) in
let ro = Inc.observe ri in
stabilize();
let v = Obs.value_exn ro in
7. *Update* source variables.

\[ \text{Var.set : 'a Inc.t -> 'a -> unit} \]
7. *Stabilize* again

```plaintext
inc.stabilize();
y : int Var.t
```

![Diagram showing the incremental computation process with nodes representing numbers and variables.](image-url)
8. Get *plain value* from observable after stabilizing.

```
let v_updated = Obs.value_exn ro in
```

```
Obs.value_exn : 'a Inc.t -> 'a Obs.t
```

```
y : int Var.t
```
9. **Repeat**: Set var --> Stabilize --> Get observed value

Each time, the subgraph that changed and on which the answer depends is recomputed.

```ocaml
Var.set x 1;
Inc.stabilize();
let v_updated2 = Obs.value_exn ro in
```
Structured Graphs

So Far: Unstructured, \textit{ad hoc} graphs

Next: Structured graphs

trees

tables/spread sheets
Structured Graphs

Implement a reduce of function f over a sequence.

Let \( \text{prev} \) be the array created in previous iteration
Each cell \( i \) of the \textit{current} array will be defined as follows:

\[
\begin{align*}
\text{curr}[i] &= f(\text{prev}[2*i])(\text{prev}[2*i+1]) & \text{if prev}[2*i+1] \text{ exists} \\
\text{curr}[i] &= \text{prev}[2*i] & \text{otherwise}
\end{align*}
\]
Let \textbf{prev} be the array created in previous iteration
Each cell \(i\) of the \textbf{current} array will be defined as follows:

\[
\begin{align*}
\text{curr}[i] &= f(\text{prev}[2*i]) (\text{prev}[2*i+1]) & \text{if prev}[2*i+1] \text{ exists} \\
\text{curr}[i] &= \text{prev}[2*i] & \text{otherwise}
\end{align*}
\]

Standard Functional Algorithm:

\[
\text{let rec merge (prev: 'a array) (f:'a -> 'a -> 'a) : 'a =}
\]
\[
\text{if Array.length prev} \leq 1 \text{ then prev.(0) else}
\]
\[
\text{let len = Array.length prev in}
\]
\[
\text{let len' = (len/2) + (len mod 2) in}
\]
\[
\text{let cell i =}
\]
\[
\text{if } i * 2 + 1 < \text{len then } f \text{ prev.(2*i)} \text{ prev.(2*i+1)}
\]
\[
\text{else } \text{prev.(2*i)}
\]
\[
in
\]
\[
\text{let curr = Array.init len' cell in}
\]
\[
\text{merge curr f}
\]

compute new cell value from previous cell values

prev array of values thrown away after its one use
Let \( \text{prev} \) be the array created in previous iteration
Each cell \( i \) of the \( \text{current} \) array will be defined as follows:

\[
\begin{align*}
\text{curr}[i] &= f(\text{prev}[2*i]) (\text{prev}[2*i+1]) \quad \text{if prev}[2*i+1] \text{ exists} \\
\text{curr}[i] &= \text{prev}[2*i] \quad \text{otherwise}
\end{align*}
\]

Standard Functional Algorithm:

\[
\begin{align*}
\text{let rec merge} (\text{prev}: 'a Inc.t array) (f:'a -> 'a -> 'a) : 'a = \\
& \quad \text{if Array.length prev <= 1 then prev.(0)} \\
& \quad \text{else}
& \quad \text{let len = Array.length prev in} \\
& \quad \text{let len' = (len/2) + (len mod 2) in} \\
& \quad \text{let cell i =}
& \quad \text{if i * 2 + 1 < len then Inc.map2 prev.(2*i) prev.(2*i+1) ~f:f} \\
& \quad \text{else prev.(2*i)} \\
& \quad \text{in} \\
& \quad \text{let curr = Array.init len' cell in} \\
& \quad \text{merge curr f}
\end{align*}
\]

pass in array of incrementals
create incremental graph:
Functional algorithms are easily transformed into incremental functional algorithms.

Stack of Functional Recursive Calls:

- $f \ x_1$
- calls $f \ x_2$
- calls $f \ x_3$

Build Incremental graph Calls:

- $f \ x_1$
- calls $\text{Inc.map} \ x_2 \sim f$
- calls $\text{Inc.map} \ x_3 \sim f$

1. convert argument from 'a to 'a Inc.t
2. convert result computed from 'b to 'b Inc.t by using Inc.map
3. fix up initial call to supply 'a Inc.t rather than 'a (use Var.create, Var.watch)
   fix up result returned to extract 'a from 'a Inc.t (use Inc.observe, Obs.value_exn)
Mutation

What happens if your algorithm is not function? Uses mutable references?

Issue 1: The output is immediately "out of date"

original run

If you run it again, you get a different answer

Very difficult to reason about (and draw!)
Avoid at almost all costs.
Mutation

What happens if your algorithm is not function? Uses mutable references?

Issue 2: An external agent modifies your reference

You usually want your inputs to have type 'a Var.t so you can watch them.
AN APPLICATION:
INCREMENTAL LONGEST COMMON
SUBSEQUENCE ALGORITHMS
Comparative Genomics
DNA Sequences

A C T G C A ....

Nucleotides

A(denine)
C(ytosine)
G(uanine)
T(hymine)
X is a *Subsequence* of Y if X can be obtained from Y by deleting some of the elements of Y.

A *Longest Common Subsequence* between Z and W is a subsequence S of both Z and W that is as long or longer than any other subsequence of Z and W.
Longest Common Subsequence: Rule 1.

Rule 1: first letters match

Input Sequences

A :: [ ... rest1 ... ]

A :: [ ... rest2 ... ]

Longest Common Subsequence

A :: LCS (rest1, rest2)

Example

A C G T

A C T A

LCS of rest is C T
Longest Common Subsequence: Rule 2.

Rule 2: first letters con't match

Input Sequences
A :: [ ... rest1 ... ]
C :: [ ... rest2 ... ]

Longest Common Subsequence
LCS (A::rest1, rest2)
or
LCS (rest1, C::rest2)
(whichever is longest)

Example
A C G T
C G A T

Longest Common Subsequence
LCS is C G T
LCS is A T (or G T)
Redundant Computation

\[ \text{LCS} (A\ G\ G\ T,\ C\ G\ A\ T) \]

\[ \text{LCS} (G\ G\ T,\ C\ G\ A\ T) \]

\[ \text{LCS} (A\ G\ G\ T,\ G\ A\ T) \]

\[ \text{LCS} (G\ G\ T,\ G\ A\ T) \]
Redundant Computation

LCS (A G G T, C G A T)

LCS (G G T, C G A T)

LCS (G T, C G A T)

LCS (G G T, G A T)

reuse

LCS (A G G T, G A T)
## Memoize Results (Dynamic Programming)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>LCS(AGGT, CGTT)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>LCS(T, GTT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>LCS(TT, TT)</td>
<td></td>
<td>LCS(T, T)</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cell \((i, j)\) contains \(\text{LCS} (\text{input}1[i..], \text{input}2[j..])\)
Cell \((i, j)\) depends on
Cell \((i-1, j-1)\), or
Cell \((i, j-1)\) and Cell\((i-1, j-1)\)
### Memoize Results (Dynamic Programming)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
### Memoize Results (Dynamic Programming)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Create a key-value map to store intermediate results:

- keys have type `dna * dna`
- values have type `dna * length`
- `Dict.find (dna1, dna2) = LCS(dna1, dna2)`

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
You will actually create a generic memoizer

- A functor generates a memoizer for *any* function!
- You'll apply it to the LCS algorithm
Build an incremental dependency graph

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

LCS(seq2, seq2)
Implementation Data Structure: Phase 2

Mutate input cells; Stabilize incremental graph; Obtain result

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AGT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>G</td>
<td>GT</td>
<td>GT</td>
<td>GT</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

LCS(seq2, seq2)
Assignment Summary: Caching N Ways

- Lazy computation with infinite data structures (streams.ml)
  - lazy results get cached
  - infinite speedup when you process infinite data structures!

- Manually Memoizing Fibonacci (memo.ml)
  - $\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$ if $n > 1$
  - recursive calls are cached to avoid exponential blow-up

- Auto-memoizing (memo.ml)
  - build a functor to cache results for any function
  - build a dictionary that maps function inputs to outputs
  - an automatic dynamic programmer
  - apply to LCS algorithm

- Incremental computation (lcs.ml)
  - build a dictionary of incrementals
  - incrementally recompute LCS
What did you get out of this course?

**HOT** (Higher-order, Typed) programming gives great code reuse.

All languages should have lambdas.

All languages should have ML data types. Concise code + exhaustiveness checks for the win.

Get work done by creating new data not always by changing old data.

Immutable data preserves invariants, simplifies reasoning about your code.

If you have inductive data, think inductively! Assume your IH and use it to compute your answer.

Functions are data structures.

Indeed, represent functions using data structures (ie: ASTs).

Lazy evaluation allows you to program with the abstraction of infinite data.

Parallelism via Map, Reduce, Scan.

Java Programs are insanely verbose. Honestly, why go back?

**OCaml**

Prove things about entire programming languages via induction over their ASTs!

Substitution model of execution defines program semantics.

But implement interpreters with an environment-based model and closures.

Check your representation invariants when data released from a module.

Trees and sequences are good for parallel (and incremental) computation.

**HOT** programming gives great code reuse.
Recursive XKCD