My long path towards $O(n)$ longest-path in 2-trees

JORDAN BISERKOV

ClojuTRE
Helsinki, Finland
September 14th 2018
Jordan Biserkov

- Programming professionally since 2001
- Found Lisp in 2005 via pg essays & books
- Found Clojure on HN in 2010, fell in love
- Independent contractor for Cognitect since 2018
- Biserkov.com
My epic journey in the 2-trees forests

- End goal: implement the Big $O(n)$ boss
- but first $O(k)$ bosses in the Bottom-level
  - First use of my superpower
- The $O(n \sqrt{n})$ boss
  - Side quest: Find 5 bugs in a 3rd party library
  - The ancient Structural tree
- The $O(n \log n)$ boss
  - A wild stack overflow appears
- The final fight
2-trees are NOT ...

- Binary trees
- Even trees

2-trees are ...

- A class of undirected graphs
- Used to model electric circuits
- Recursively structured
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
2-tree recursive construction demo
Background

➢ The 90’s algorithm to compute the length of the longest path in a 2-tree has colossal hidden constants and is “linear” in purely abstract sense
  • Never implemented

➢ In 2013 Markov, Vassilev and Manev published a novel algorithm
  • Implemented as pseudo-code in the paper

➢ **Goal:** Implement the MVM algorithm in O(n) time
Overview

➢ Recursively split the 2-tree into sub-2-trees
  • Only a few nodes change
  • Perfect fit for Clojure’s persistent data structures

➢ Boundary cond.: Leaf edges, label [1 1 0 0 0 0 0 0]

➢ Combine labels of subtrees to compute parent tree label

➢ The first element of the label is the result – the length of the longest-path
Code structure

Top level
- Compute-label

Middle level
- Combine-on-face
- Combine-on-edge

Bottom level – helper functions
- max-2-distinct
- max-3-distinct
a and b are vectors with k elements each

\[ \max\{a_i + b_j \mid i \neq j\} \]

(defn naive-max2DistinctFolios [a b n]
  (reduce max
    (for [i (range 0 k)
          j (range 0 k)
          :when (not= i j)]
      (+ (nth a i) (nth b j)))))

Problem: 2 Nested for-loops $\rightarrow O(k^2)$ runtime

$$a = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}, \quad b = \begin{bmatrix} 6 & 7 & 8 & 9 & 10 \end{bmatrix}$$

<table>
<thead>
<tr>
<th>+</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
Optimization: $O(k)$

- Iterate each vector separately, keeping track of:
  - the maximum
  - the second largest
  - the index of the maximum

- Check whether we can use both maxima (different indices) and if not - which alternative is larger

\[
\max \left( + \text{ maxA secondB} \right) \\
\left( + \text{ maxB secondA} \right)
\]
\( \max \{ a_i + b_j + c_t \mid i \neq j \neq t \neq i \} \)

\(a, b\) and \(c\) are vectors with \(k\) elements
Problem: 3 Nested for-loops $\rightarrow O(k^3)$ runtime
Optimization: $O(k)$

- Iterate each vector separately, keeping track of:
  - the maximum
  - the second largest
  - the third largest
  - the index of the maximum and the second largest

- Check which of the 36 combos are valid and which sum is the largest

- Terrible complexity, many bugs
Generative testing to the rescue

- Also called property-based testing
- Finds complex bugs immediately
- Difficult to come up with a useful property
- Shrinks input to minimal case which triggers the bug, in this case often vectors with 0 and 1
- Use \( (= \text{(naïve ...)} \ (\text{faster ...})) \) as testing property
Previous implementation

- Java
- 2-tree represented as a matrix
- Sub-2-tree = submatrix = tons of copying
- $O(n^2)$ runtime
- $O(n^2)$ memory usage
My first implementation

- Clojure
- as close to the paper as possible
- 2-tree represented as map from int to set of int
- $O(n\sqrt{n})$ runtime
- Perhaps Clojure’s dynamic typing is the problem?
Optimization: use Zach Tellman’s int-map and int-set

{0 #{1 2 3 4} 1 #{0 2} 2 #{0 1 3 4} 3 #{0 2} 4 #{0 2}}

Runtime is faster, but complexity still $O(n\sqrt{n})$
Sidequest: find 5 bugs in 3rd-party library

➢ The problem manifests as a NullPointerException

➢ Cursive’s debugger is awesome
  • Breakpoint on exception

➢ Zach Tellman is a great guy, fixed bug quickly

➢ Problem has evolved: infinite looping in subgraph-walk during multiple-recursion?!? How? Why?

➢ 5 times in a row, same-day bug delivery, what sorcery is this?
The root cause of the slowdown?

- Splitting into sub-2-trees
- Persistent data structure are fast enough, actual updates not the problem
- Computing which vertices need updating is the problem
- The authors told me to seek the ancient Structural tree
Representation: map from edge to [vertices]

$$\{[0 \ 1] \ [2]\}
\{[0 \ 2] \ [3 \ 4 \ 10]\}
\{[1 \ 2] \ [5]\}
\{[1 \ 5] \ [8 \ 9]\}
\{[2 \ 5] \ [6]\}
\{[5 \ 6] \ [7]\}$$

Blue nodes represented implicitly: parent edge + vertex
External edge nodes represented implicitly as nil
My second implementation

➢ Iterative preprocessing step: builds structural tree
➢ Recursive part operates on structural tree
➢ $O(n \log n)$ runtime
➢ More complex, unexplored territory
➢ Generative testing saves the day again
➢ Best of both implementations
  • Straightforward and correct, but slow one
  • Complex and unproven, but faster one
Suddenly wild stack overflow appears

- But how?
- Infinite recursion?
- Another bug?
- No, all tests pass. What?
- A genuine stack overflow due to one benchmark using ultra-tall 2-trees
Workaround?

- Increase the call stack size via JVM options, but the problem reappears when you double N a few times

**Solution:** Every recursive algorithm can be made iterative, by using an explicit stack parameter, instead of the call stack.

Then it hit me – there is a data structure in my program that holds all the information it needs – the EdgesVertices map.

With some modifications the recursive calls can be removed completely and all the work can be done during the preprocessing (bottom-up) phase.
My third implementation

- Iterative, dynamic programming, no recursive part
- O(n) runtime!!
- Millions of vertices without overflow
- Map from edge to vector of labels
- Generative testing saves the day yet again
The result

Projected $O(n \log n)$
Projected $O(n)$
Actual time

Benchmarks via Criterium by Hugo Duncan
## Implementations recap

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Direction</th>
<th>Data structure</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Recursive</td>
<td>🛡️ ➡️</td>
<td>Matrix</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Direct</td>
<td>Recursive</td>
<td>🛡️ ➡️</td>
<td>int-map, int-set</td>
<td>$O(n\sqrt{n})$</td>
</tr>
<tr>
<td>Indirect</td>
<td>Iterative</td>
<td>➡️</td>
<td>int-map, int-set</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td></td>
<td>Recursive</td>
<td>🛡️ ➡️</td>
<td>EdgeVertices map</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>Iterative</td>
<td>➡️</td>
<td>int-map, int-set, EdgeLabels map</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Transient variants of persistent data structures

➢ If the original value is never used after modification, it’s safe to modify it in place, while still presenting an immutable interface to the outside world.

➢ Add complexity, so make your program work without them, then add:
  • a call to transient in the beginning
  • ! to assoc, dissoc, conj and friends
  • a call to persistent! at the end
Further optimization of middle level functions

- Higher level decision making – 2 simpler, faster functions instead of 1 complex, mathematically pure
- Proper case simplified greatly, removed branching
- Degenerate cases handled by specialized variant
  - Simplified greatly, removed branching
  - When \(a = 1\) the expression \((+ \ a \ b)\) becomes \((\text{inc} \ b)\)
  - When \(c = 0\) the expression \((\text{max} \ c \ d)\) becomes \(d\)
- Frequent trivial case handled directly
  - No function call cost, no unnecessary computation
Memoization

➢ The function remembers the result for given parameters to avoid costly recomputation

➢ Useful whenever a big problem is divided into smaller ones

➢ The built-in memoize returns a variable argument function, which adds overhead.

➢ If we know the number of arguments, we can build our own version which is simpler and faster
Resources

➢ The algorithm
https://sites.google.com/site/minkommarkov/longest-2-tree--draft.pdf

➢ My implementations
https://github.com/Biserkov/twotree-longest-path

➢ Understanding Clojure’s transients
http://www.hypirion.com/musings/understanding-clojure-transients
Thank you!
Questions?