Visualizing Patterns with Dotplots

308-763 Advanced Seminar on Compilation and Run-time Systems for Object-Oriented Languages

David Bélanger
dbelan2@cs.mcgill.ca

McGill University
Montreal, QC


Overview

- Introduction
- Constructing a dotplot
- Patterns
- Applications
- Implementation
- Conclusion and Further Work
Introduction

- Dotplots are used to visualize large amounts of information
- Idea: Let the human eye detect interesting patterns
- Applications in several fields:
  - Software Engineering
    - Software structure
    - Duplicated code
  - Patterns in programming languages
  - Study of execution trace
  - Biology: comparing DNA sequences
  - Natural languages
  - Information retrieval
A Dotplot

Two million lines of C code in a module of a telecommunications switching program (figure from Helfman, 1995).
How it Works

- Data is tokenized: line, word, letter, n-gram etc.
- Rows and columns are labelled by the sequence of tokens
- Put a dot in entry \((i, j)\) if token \(i = \) token \(j\)

Figure from [Helfman, 1995]
How it Works

- Dots outside main diagonal indicate similarity: the darker, the more similar
- The graph is always symmetric
- Main diagonal is always full of dots
- For millions of points in hundreds of pixels, several points are mapped to the same pixel: we need some weight function and a way to assign a colour.
Basic Patterns - Diagonals

- Ordered matches (ex: copies and versions)
- Two versions of $x_{mh}$ (some program in X)

Figures from [Helfman, 1995]
Basic Patterns - Squares

- Unordered matches: “vocabulary”, “working set”
- Two distinct squares: a change in “vocabulary”
- Example: Several works of Shakespeare (1 million words). Dark squares on main diagonal are names of characters. Change of vocabulary.

Figures from [Helfman, 1995]
Diagonal Texture

- Macros in X11 manual pages
- Diagonal texture: Short sequences repeated often

Figure from [Helfman, 1995]
Diagonal Texture

Last 400 bytecodes executed. Diagonal lines in bottom-right corner correspond to a loop executed 4 times.
Comparing Data

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To compare sequences, simply concatenate them.

- $AA, BB$ - self-comparison
- $AB, BA$ - comparison of $A$ and $B$
Pattern - Shuffle

- 37 million words of parliamentary debates
- Dark squares ⇒ unique vocabulary and diagonals ⇒ a copy
- Contradiction?

Figure from [Church and Helfman, 1993]
Pattern - Shuffle

Figure from [Helfman, 1995]
Comparing Data

- Manual in Dutch, French, German, Italian, Spanish and Swedish (1 million 4-grams)
- Darker square means higher similarity

Figure from [Helfman, 1995]
Broken Diagonals

- Insertions or deletions of tokens
- Ex: Code inserted in a new version

Figure from [Helfman, 1995]
Reordered Diagonals

- Caused by reordering data
- Ex: Renaming files or reordering functions

Figures from [Helfman, 1995]
Light Cross

- Broken diagonals: insertions into ordered sequences
- Light cross: insertions into unordered sequences

Figure from [Helfman, 1995]
Example - Light Cross

- Dotplot of last 1600 bytecodes executed
- Change of instruction “vocabulary”
- A lot of integer instructions, no invokes
Some user used dotplots to find duplicated code and remove it.

After several iterations, the only remaining patterns were due to the programming language itself.

Observed that in the Switch construct breaks are more frequent than flowing through next case.

Conclusion: Make break the default behaviour. Have a `no_break` statement to continue.

Minimize syntax and maximize legibility.
A 1000-line C switch statement
Figure from [Helfman, 1995]
Resolution

- Some patterns may be “hidden” by other patterns
- To see these secondary patterns:
  1. Remove primary patterns
  2. Replot
  3. Repeat
- At different scales, different patterns will be seen
- A dark square may be composed of many interesting patterns
Execution Trace

First 400, 800, 1600 and 4000 executed bytecodes respectively.
Execution Trace

Last 400, 1600, 4000 and 20000 executed bytecodes respectively.
Order of Tokens

- Order of data is important (tokens and sequences)
- Different orders give different patterns
- Some orderings are “better”
- Some reordering algorithm can be used
- Ex: Reordering files to cluster similar ones.
Dynamic Behaviour

- File systems usage
- Program execution
  - Patterns in execution (ex: loops, frequent sequence of instructions, etc.)
  - Comparing trace files
    - 2 different programs
    - Same program with different inputs
    - 2 “versions” of same program (one with some optimization, the other not)
Implementation Overview

Simple Algorithm
- tokens $\rightarrow$ f-image $\rightarrow$ q-image
- f-image: 2-dimensional array of floating points
- q-image: 2-dimensional array of data displayable
- Compute f-image by assigning 1 to entry $(i,j)$ if $i$ = token $j$, 0 otherwise
- Compute q-image by assigning colours based on weighting (not trivial)

Problems:
1. More interested in infrequent match (ex: we already know '}' will match a lot in code)
2. $O(n^2)$ space
3. $O(n^2)$ time
Implementation Improvements

1. Weighting: Assign a larger weight to infrequent tokens that match instead of a constant weight for all matches. \((1/\text{freqOfToken} \text{ is good})\)

2. Compression: Map several points to a single point in the dotplot. Space is \(O(m^2)\) where \(m\) is the size of the f-image.

3. Approximation: Possible to speed up by ignoring frequent tokens. \(O(VT^2)\) time where \(V\) is the number of distinct tokens and \(T\) is some threshold value.
Conclusion and Further Work

- Dotplots have been proven to be an effective way of analyzing data in several applications.
- Offer an alternative way of looking at data
- Not much work has been done on applying them to execution traces

Further Work
- Implement dotplots in EVolve
- Study execution traces for common patterns