Kernel-based Similarity Search in Massive Graph Databases with Wavelet Trees

Yasuo Tabei
(JST ERATO Minato Project)
Joint work with Koji Tsuda (AIST)
Outline

■ Introduction
  ● Recent development of graph databases
  ● Needs for graph similarity search
  ● Bag-of-words representation of a graph
  ● Semi-conjunctive query

■ Method
  ● Scalable similarity search with wavelet trees

■ Experiments
  ● Use a large-scale graph dataset
  ● 25 million chemical compounds
Graphs are everywhere

Gene co-expression network

Protein 3D structure

Chemical compound

RNA 2D structure

Social Network
Graph Similarity Search

- Retrieve graphs similar to the query
- Large databases
  - More than 20 million chemical compounds in PubChem database
- Bag-of-words representation of graphs
  - WL procedure (NIPS 2009)
- Why not use document retrieval methods?
  - Inverted index
  - Not that easy (explained later)
Weisfeiler-Lehman Procedure (NIPS, 09)

Convert a graph into a set of words (bag-of-words)

- Make a label set of adjacent vertices: \{E, A, D\}
- Sort: A, D, E
- Add the vertex label as a prefix: B, A, D, E
- Map the label sequence to a unique value: B, A, D, E → R
- Assign the value as the new vertex label

Bag-of-words: \{A, B, D, E, …, R, …\}
Search by cosine similarity

- Identify all graphs in the database whose cosine is larger than a threshold $1-\varepsilon$

$$W_i \text{ s.t } K_N(W_i, Q) = \frac{|W_i \cap Q|}{\sqrt{|W_i|} \sqrt{|Q|}} \geq 1 - \varepsilon$$

- $W_i, Q$: bag-of-words of graphs

- The above solution can be relaxed as follows,

If $K_N(Q, W) \geq 1 - \varepsilon$, then

$$(1 - \varepsilon)^2 |Q| \leq |W| \leq \frac{|Q|}{(1 - \varepsilon)^2}$$

- Can be used for fast search
Semi-conjunctive query

Cosine query can be relaxed to the following form

\[ W_i \ s.t \ |W_i \cap Q| \geq k \]

- The number of common words between two bag-of-words \( W_i \) and \( Q \)
  
  \[ |W_i \cap Q| = |(A,C,E,F,H) \cap (A,E,I,J,L)| = |(A,E)| = 2 \]

- \( k = (1-\varepsilon)^2 |Q| \)
- No false negatives
- False positives can easily be filtered out by cosine calculations
Inverted index

In natural language processing, inverted index has been used to solve semi-conjunctive query

Inverted Index

<table>
<thead>
<tr>
<th>Word</th>
<th>Graph ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2, 8, 13, 15</td>
</tr>
<tr>
<td>B</td>
<td>4, 9, 13</td>
</tr>
<tr>
<td>C</td>
<td>8, 10, 16</td>
</tr>
<tr>
<td>D</td>
<td>1, 3, 11</td>
</tr>
<tr>
<td>E</td>
<td>4, 9, 13, 14</td>
</tr>
</tbody>
</table>

Associative map

- key ⇔ word
- value ⇔ graph identifiers including a word
Bottom-up search

Inverted Index

<table>
<thead>
<tr>
<th>Word</th>
<th>Graph ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2,8,13,15</td>
</tr>
<tr>
<td>B</td>
<td>4,9,13</td>
</tr>
<tr>
<td>C</td>
<td>8,10,16</td>
</tr>
<tr>
<td>D</td>
<td>1,3,11</td>
</tr>
<tr>
<td>E</td>
<td>4,9,13,14</td>
</tr>
</tbody>
</table>

Query: (A, C, E)

i) Look the index up with query bag-of-words
ii) Aggregate all the lists of graph indices
iii) Sort
iv) Scan

Aggregation

(2,8,13,15,8,10,16,4,9,13,14)

Sort

(2,4,8,8,9,10,13,13,14,15,16)

k=2
Search time of inverted index on 25 million graphs

- Search time of inverted index is not so different from that of sequential scan

**Graph:**
- X-axis: Number of graphs
- Y-axis: Average search time (sec)
- Line colors and markers:
  - Inverted index (blue triangles)
  - Sequential scan (green pluses)

**Data Points:**
- 25 million graphs:
  - Inverted index: 40 sec
  - Sequential scan: 38 sec
Each word is not specific enough

Query contains 1000s of words

Aggregated array is VERY long

Sorting takes $O(C \log C)$ in time
Overview of our method

- Top-down search in a tree over the series of graphs
- Huge memory, if tree is implemented with pointers
- Wavelet Tree: Succinct data structure
- The smaller the similarity threshold is, the quicker the algorithm finishes
  - Not the case in inverted index
Binary tree over graphs

- **leaf** ⇔ **graph**
- **node** ⇔ **interval**
- Each node is identified by a bit string \( v = \{01\} \)
- At the leaves, the graph indices correspond to \( \text{int}(v)+1 \)
  (e.g., \( \text{int}(010)+1 = 2 + 1 = 3 \))
Summarization of bag-of-words

- Represent bag-of-words as a bit array
  \[ 1 2 3 4 5 6 7 8 \]
  Ex) \( W_i = (1,3,4,7,8) \rightarrow x_i = (1,0,1,1,0,0,1,1) \)

- Take disjunction \( \lor \) of all bit arrays in the interval of a node \( v \)

Ex) For an interval \([1,4]\)
\[ X_1 = (0,1,0,0,0,0,1,0) \]
\[ X_2 = (1,0,1,1,0,0,0,0) \]
\[ X_3 = (1,0,0,0,0,0,1,1) \]
\[ X_4 = (1,0,0,0,0,0,0,1) \]
\[ y_v = x_1 \lor x_2 \lor x_3 \lor x_4 \]
\[ = (1,1,1,1,0,0,1,1) \]
Binary tree over graphs

- Assign to each node $v$ a bit array $y_v$
- $y_v$: bit array
  - $i$-th bit is 1 if graphs in an interval have the corresponding word.
Top-down traversal

- **Q**: bag-of-words of a query
- Perform top-down traversal
- Prune the search space if \( \sum_{j \in Q} y_v[j] \leq k \)
- The larger \( k \) is, the smaller the search space is
Huge Memory

- Time is $O(\tau m)$: Very fast
  - $\tau$: the number of traversed node
  - $m$: the number of bag-of-words in a query

- Space is $O(Mn\log n)$ bit
  - $M$: the number of unique words
  - $n$: the number of graphs
Wavelet Tree! (SODA,03)

- Replace $y_v$ in each node $v$ by a rank dictionary
  - explained in next slides
- Implement a tree without using pointers
- Only 60% memory overhead compared to the inverted index (Vigna,08)
- Access to the summary information in any internal node
Rank dictionary (Raman, 02)

- Give bit array $B[1, n]$ the following operation:
  - $\text{rank}_c(B, i)$: return the number of $c \in \{0, 1\}$ in $B[1 \ldots i]$

**Ex)** $B = 0110011100$

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank$_1(B, 8) = 5$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rank$_0(B, 5) = 3$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Implementation of rank dictionary

- Divide the bit array $B$ into large blocks of length $\ell = \log^2 n$
  
  $R_L =$ Ranks of large blocks

- Divide each large block to small blocks of length $s = \log n / 2$

  $R_s =$ Ranks of small blocks relative to the large block

 rank$_1(B, i) = R_L \lfloor i / \ell \rfloor + R_s \lfloor i / s \rfloor + \text{(remaining rank)}$

  
  Time: $O(1)$
  
  Memory: $n + o(n)$ bits
Restricted inverted index

- Concatenate graph ids for words in the root
- Restrict the inverted index for the interval $[s_v, t_v]$ of a node $v$

<table>
<thead>
<tr>
<th>Word</th>
<th>Graph ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1,3,6,8</td>
</tr>
<tr>
<td>B</td>
<td>2,5,7</td>
</tr>
<tr>
<td>C</td>
<td>1,2,7</td>
</tr>
<tr>
<td>D</td>
<td>4,5</td>
</tr>
</tbody>
</table>
Whole structure of restricted inverted index

[1,8] A B C D
1 3 6 8 2 5 7 1 2 7 4 5

[1,4] A B C D
[3,4] A D
[5,6] A B D
[6,7] A B C

[5,8] A B C D
Similarity search

To retrieve graphs similar to a query $Q=(A,C)$, the tree is traversed in the top-down manner.
Similarity search

- To retrieve graphs similar to a query $Q= (A, C)$, the tree is traversed in a top-down manner.

Observation

- To perform top-down traversal, only intervals of words in each node are necessary.
Replace restricted inverted index $A_v$ in each node $v$ with a bit array $b_v$.

- $b_v[i]=1$ if $A_v[i]$ goes to the right child
Similarity search

- Intervals of child nodes can be computed by rank operations
  - \( s_{left(v),j} = \text{rank}_0(bv, s_{vj}-1) + 1, t_{left(v),j} = \text{rank}_0(bv, t_{vj}) \)
  - \( s_{right(v),j} = \text{rank}_1(bv, s_{vj}-1) + 1, t_{right(v),j} = \text{rank}_1(bv, t_{vj}) \)

Ex)

<table>
<thead>
<tr>
<th></th>
<th>broot</th>
<th>0 0 1 1 0 1 1 0 0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aroot</td>
<td>1 3 6 8 2 5 7 1 2 7 4 5</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{rank}_0(broot, 8-1) + 1 = 4, \text{rank}_0(broot, 10) = 5 \)

<table>
<thead>
<tr>
<th></th>
<th>bleft</th>
<th>0 1 0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aleft</td>
<td>1 3 2 1 2 4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>bright</th>
<th>0 1 0 1 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aright</td>
<td>6 8 5 7 7 5</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{rank}_1(broot, 8-1) + 1 = 5, \text{rank}_1(broot, 10) = 5 \)
Wavelet Tree

- Wavelet tree can be obtained to replace the restricted Inverted indices with bit arrays.
- Wavelet tree consists of bit arrays $bv$ and initial intervals $Croot$.
Wavelet Tree

Graph ids can be recovered from bit strings on the path from the root to leaves.
Memory

- \((1+\alpha)N \log n + M \log N\) bits
  - Bit arrays bv
  - Initial intervals Croot
- \(N\): the number of all words in the database
- \(n\): the number of graphs
- \(\alpha\): overhead for rank dictionary (\(\alpha=0.6\))
- For inverted index, \(N \log n\) bits
- About 60% overhead to inverted index!!
Experiments

- 25 million chemical compounds from PubChem database
- Use search time and memory as evaluation measures
- Compare our method gWT to
  - inverted index
  - sequential scan implemented in G-Hash (Wang et al, 2009)
Search time on 25 million graphs

![Graph showing search time on 25 million graphs](image)
Memory usage

![Graph showing memory usage vs. number of graphs]

- **Wavelet Tree**
- **Croot**

The graph illustrates the memory usage in megabytes as a function of the number of graphs. The wavelet tree uses significantly more memory compared to Croot, especially as the number of graphs increases.
Overhead of rank dictionary

![Graph showing overhead vs. number of graphs]
Construction time

![Graph showing construction time](image)

- **time (sec)**
- **# of graphs**
Related work

- A lot of methods have been proposed so far.
  1. gIndex [Yan et al., 04]
  2. Graph Grep [Shasha et al., 07]
  3. Tree+Delta [Zhao et al., 07]
  4. TreePi [Zhang et al., 07]
  5. Gstring [Jiang et al., 07]
  6. FG-Index [Cheng et al., 07]
  7. GDIndex [Williams et al., 07]
  etc
Related work

- Decompose graphs into a set of substructures
  - subgraphs, trees, paths etc
- Build a substructure-based index
Drawbacks

• Require frequent subgraph mining
• Do not scale to millions of graphs
Efficient similarity search method for massive graph databases

Solve semi-conjunctive query efficiently

Built on wavelet trees

Use Weisfeiler-Lehman procedure to convert graphs into bag-of-words

Applicable to more than 20 million graphs

Software

http://code.google.com/p/gwt/
Acknowledgements

• Prof. Shin-ichi Minato (Hokkaido Univ.)
• Dr. Daisuke Okanohara (PFI)
• Members in ERATO Minato Project