Set Cover Algorithms For Very Large Datasets

Graham Cormode
Howard Karloff
AT&T Labs-Research

Tony Wirth
University of Melbourne
Set Cover?

- Given a collection of sets over a universe of items
- Find smallest subcollection of sets that also cover all the items.
Why Set Cover?

The set cover problem arises in many contexts:

- **Facility location**: facility covers sites
- **Machine learning**: labeled example covers some items
- **Information Retrieval**: each document covers set of topics
- **Data mining**: finding a minimal ‘explanation’ for patterns
- **Data quality**: find a collection of rules to describe structure
How to solve it?

- Set Cover is NP-hard!
- Simple greedy algorithm:
  - Repeatedly select set with most uncovered items.
  - Logarithmic factor guarantee: $1 + \ln n$
  - No factor better than $(1 - o(1)) \ln n$ possible
- In practice, greedy very useful:
  - Better than other approximation algorithms
  - Often within 10% of optimal
Existing Algorithms

- **Greedy algorithm**: $1 + \ln n$ approximation
  - Until all $n$ elements of $X$ are in $C$ (initially empty):
    - Choose (one of) set(s) with maximum value of $|S_i - C|$
    - Let $C = C \cup S_i$

- **Naïve algorithm**: no guaranteed approximation
  - Sort the sets by their (initial) sizes, $|S_i|$, descending
  - Single pass through the sorted list:
    - If a set has an uncovered item, select it
    - Update $C$
Example greedy

ABCD

AFG

G

C

E

ABDF

BCG

EH

A

I
Optimum

ABCDE

AFG

GH

Cl

E

ABDFG

BCG

EH

A

I
What’s wrong?

- Try implementing greedy on large dataset:
  - Scales very poorly
- Millions of sets with universe of many millions of items?
- Dataset growth exceeds fast memory growth
- If forced to use disk: selecting “largest” set requires updating set sizes to account for covered items
- Even 30Mb instance required >1 minute to run on disk
Implementing greedy

- **Main step**: find set with largest $|S_i - C|$ value
- **Inverted index**:
  - Maintain updated sizes in priority queue
  - Inverted index records which sets each item is in
  - Costly to build index, no locality of reference
- **Multipass solution**:
  - Loop through all sets, calculating $|S_i - C|$ on the fly
  - Good locality of reference, but many passes!
  - If $|S_i^* - C|$ drops below a threshold:
    - Loop adds all sets with specific $|S_i^* - C|$ value
Idea for our algorithm

- Huge effort to find $\max |S_i - C|
- Instead find set close to maximum uncovered size
- If always at least factor $\alpha \times$ maximum:
  - We have $1 + \frac{(\ln n)}{\alpha}$ approximation algorithm
  - Proof similar to that for greedy

- We call it Disk-Friendly Greedy (DFG)
How to achieve this

- Select parameter $p > 1$: governs approximation and run time
- Partition sets into subcollections:
  - $S_i$ in $Z_k$ if: $p^k \leq |S_i| < p^{k+1}$
- For $k \leftarrow K$ down to 0:
  - For each set $S_i$ in $Z_k$:
    - If $|S_i - C| \geq p^k$: select $S_i$ and update $C$
    - Else: let $S_i \leftarrow S_i - C$ and add it to $Z_{k'}$: $p^{k'} \leq |S_i| < p^{k'+1}$
- For each $S_i$ in $Z_0$: select $S_i$, update $C$, if has uncovered item
Example DFG run

4–7  ABCDE  ABDFG

2–3  AFG  BCG  GH  EH  CI

1  A  E  I  

12
In-memory Cost analysis

- Each $S_i$ either selected or put in lower subcollection
- Guaranteed to shrink by factor $p$ every other pass
- Total number of items in all iterations is $(1 + 1/(p-1))|S_i|$ 
- So $1 + 1/(p-1)$ times input read time
Disk model analysis

- All file accesses are sequential!
- Initial sweep through input
- Two passes for each subcollection
  - One when sets from higher subcollections added
  - One to select or knock down sets
- Block size $B$, $K$ collections:
  - Disk accesses for reading input: $D = \sum |S_i| / B$
  - DFG requires $2D[1 + 1/(p-1)] + 2K$ disk reads
Disk-based results

- Tested on Frequent Itemset Mining Dataset Repository
- Show results on kosarak (31Mb) and webdocs (1.4Gb)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Time (s)</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>kosarak.dat</td>
<td>naive</td>
<td>8.51</td>
<td>20664</td>
</tr>
<tr>
<td></td>
<td>multipass</td>
<td>331.66</td>
<td>17746</td>
</tr>
<tr>
<td></td>
<td>greedy</td>
<td>98.66</td>
<td>17750</td>
</tr>
<tr>
<td></td>
<td>DFG</td>
<td>2.61</td>
<td>17748</td>
</tr>
<tr>
<td>webdocs.dat</td>
<td>naive</td>
<td>91.21</td>
<td>433412</td>
</tr>
<tr>
<td></td>
<td>multipass</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>greedy</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>DFG</td>
<td>86.28</td>
<td>406440</td>
</tr>
</tbody>
</table>
# Memory-based results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Time (s)</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>kosarak.dat</td>
<td>naive</td>
<td>2.20</td>
<td>20664</td>
</tr>
<tr>
<td></td>
<td>multipass</td>
<td>4.21</td>
<td>17746</td>
</tr>
<tr>
<td></td>
<td>greedy</td>
<td>2.99</td>
<td>17750</td>
</tr>
<tr>
<td></td>
<td>DFG</td>
<td>1.97</td>
<td>17741</td>
</tr>
</tbody>
</table>

| webdocs.dat     | naive   | 100.98   | 433412   |
|                 | multipass | 8049.08  | 406381   |
|                 | greedy   | 199.02   | 406351   |
|                 | DFG      | 93.38    | 406338   |
Impact of $p$

- RAM-based results for `webdocs.dat`
- Improving guaranteed accuracy only increases running time by 50% (30s)
- Observed solution size improves, though not as much
Summary

- Noted poor performance of greedy, especially on disk
- Introduced alternative algorithm to greedy:
  - Has approximation bound similar to greedy
- On each disk-resident dataset: our algorithm 10 × faster
- On largest instance: over 400 × faster
- Solution essentially as good as greedy
- Disk version almost as fast as RAM version:
  - Not disk bound!