Task: Implementation of Set Similarity Join Algorithm AllPairs

November 21, 2017

In this project you will implement the set similarity join algorithm AllPairs. You will be assigned one out of three programming languages (C++/Java/Python) for your implementation.

Peter	$\{jazz, biking, swimming\}$
Katrin	$\{skiing, hiking, running, opera\}$
Astrid	$\{skiing, hiking, biking, jazz\}$

Table 1: Social Network: User interests

Many interesting problems can be represented as a set similarity join. Consider, for example, a social networking site that collects user interests. Users with similar interests should be recommended to each other. We represent the interests of each user as a set as shown in Table 1. Then, we compute the pairs of sets that are similar. Users with similar interests are recommended to each other.

1 Background

Given a collection of sets R, the set similarity join computes all pairs of sets in R that are similar. The similarity is assessed using a set similarity function $sim(r, s)$ (e.g., Jaccard, Cosine, Dice). A pair is similar if the similarity is above a user-defined threshold t. Formally, the similarity join is defined as follows:

$$simjoin(R, t) = \{(r, s) \in R \times R \mid sim(r, s) \geq t\}$$

Note that in our definition of similarity join, R is joined with itself; this kind of join is called a *self join*. In this project we only deal with self joins. R–S similarity joins, which involve two different collections R and S, use very similar techniques as similarity self joins.

In recent years, many algorithms have been proposed to compute the set similarity join. A relatively simple algorithm is AllPairs [1]. In an experimental study conducted by Mann et al. [2] AllPairs was among the three winners for computing the set similarity join.

2 Tasks

The project consists of two tasks: First, efficiently implement the AllPairs algorithm, following the pseudo-code in Algorithm 1 (along with Algorithm 2 for verification). Second, extend your implementation for weighted similarity functions (details later).

Algorithm 1 computes a similarity self join on a collection of sets R using similarity threshold t. r is a record (=sorted set) in R. Each record has a unique ID. The token at position p, $0 \leq p < |r|$, of record r is denoted as $r[p]$. I is an inverted list index, which initially is empty. The key of each inverted list is a token, the entries in the inverted list are record IDs. $I_{r[p]}$ is the list for token $r[p]$. π_r is the probing prefix length of r, π_r^I is the indexing prefix length. The length filter checks if a record s in the inverted list is long enough for r: a record s must be at least of length lb_r to match with r. The verification function $\mathit{Verify}(r, M, t)$ (Algorithm 2) verifies for each candidate set s in M if $sim(r, s) \geq t$; an efficient verification function will leverage the partial overlaps stored in M. The function $verify\_ssjoin\_paper(r, s, t, o, p_r, p_s)$ refers to the efficient verify function in Mann et al. [2]: r and s are the two sets, t is the required overlap (as computed with $eqoverlap$), o is the overlap of the prefixes, p_r and p_s are the positions where to start verification in the resp. sets.

2.1 Task 1

Efficiently implement the pseudo-code in Algorithm 1. The data has to be read from a text file, where each line represents a set containing integer tokens separated by white space. The set tokens are sorted by the token value, which represent the inverse document frequency rank of the token. The sets are already sorted by set size. Here is an example:

```
1 2 3 4 14
10 11 12 13 14
5 6 7 8 9 12 13 14
```

Your binary or script should be callable with the following parameters (including example):

```
./binary_or_script input_file jaccard_threshold
./binary_or_script bms-pos-raw.txt 0.85
```

The output of your program will contain the output size, i.e., the number of pairs in the output and the time to compute the join (without reading the input file). The time should be the CPU-time, not wallclock time. The expected format is

```
output_size
join_time_in_seconds
```

A valid output would be (if 543265 is the correct number of result pairs and 3.708 is your real CPU time):

```
543265
3.708
```

The similarity function to be implemented is Jaccard similarity, which is defined as:

$$J(r,s) = \frac{|r \cap s|}{|r \cup s|}$$

Algorithm 1: AllPairs(R, t_J)

```
input : $R$ collection of sets, $t_J$ similarity threshold
output: $res$ set of result pairs (similarity at least $t$)
1  $I = \{\}$;     /* $I$ inverted list index covering prefix of sets */
2  foreach $r$ in $R$  /* process in ascending length order of r    */
3  do
4      $M = \{\}$; /* dictionary for candidate set.  Key:  candidate,
       value:  number of intersecting tokens found so far.  */
5      for $p \leftarrow 0$ to $\pi_r - 1$   /* $\pi_r$: probing prefix length of $r$     */
6      do
7          for $s$ in $I_{r[p]}$ do
8              if $|s| < lb_r$   /* $lb_r$: length bound                   */
9              then
10                 remove index entry with $s$ from $I_{r[p]}$;
11             else
12                 if $s$ not in $M$ then
13                     $M[s] = 0$;
14                 $M[s] = M[s] + 1$;
15     for $p \leftarrow 0$ to $\pi_r^I - 1$   /* $\pi_r^I$: indexing prefix length of $r$    */
16     do
17         $I_{r[p]} = I_{r[p]} \circ r$;              /* Add set $r$ to index */
       /* Verify() verifies the candidates in $M$                  */
18     $res = res \cup Verify(r, M, t_J)$;
```

2.2 Task 2

AllPairs can be modified to allow weighted similarity functions. Each token has a particular weight associated with it. The weight is the same for each occurrence of this token, g.g., token 14 in the input example above could have weight 0.1. It has this weight in all three sets it occurs in.

Your binary or script will therefore accept another input parameter:

```
./binary_or_script input_file weight_file jaccard_threshold
```

The weight file consists of a mapping of tokens to their weight. There is one mapping per line. Each line contains the token and the weight, separated by a colon. If a token is not mapped, 1 should be assumed as its weight. It may look like this:

Algorithm 2: Verify(r, M, t_J)

```
input : $r$ probing set, $M$ candidates map (candidate $\rightarrow$ overlap), $t_J$
        Jaccard threshold
1  $res \leftarrow \emptyset$;
2  foreach $(s, o)$ in $M$  // Foreach cand. map item (cand., overlap)
3  do
4      $\pi_r \leftarrow$ probing prefix length of $r$;
5      $\pi_s \leftarrow$ indexing prefix length of $s$;
6      $w_r \leftarrow \pi_r$-th token in $r$;           // Last token of prefix in $r$
7      $w_s \leftarrow \pi_s$-th token in $s$;           // Last token of prefix in $s$
8      $t \leftarrow eqoverlap(r, s, t_J)$;
9      if $w_r < w_s$ then
10         $ret \leftarrow verify\_ssjoin\_paper(r, s, t, o, \pi_r + 1, o + 1)$;
11     else
12         $ret \leftarrow verify\_ssjoin\_paper(r, s, t, o, o + 1, \pi_s + 1)$;
13     if $ret$ is true then
14         $res \leftarrow res \cup (r, s)$;
15 return res;
```

```
3:0.1
2:0.2
14:0.1
```

The similarity function to implement is weighted Jaccard, which is defined as

$$J^W(r, s) = \frac{\sum_{w \in |r \cap s|} weight(w)}{\sum_{w \in |r \cup s|} weight(w)}$$

3 Further Readings

AllPairs combines the prefix filter and the length filter in a filter-verification framework. We recommend to start with [2]. Particularly relevant to understand AllPairs are the paragraphs “Prefix Filter” and “Length Filter” in Section 2.1 and the whole Section 2.2.

The weighted set similarity join is discussed in [3].

References

[1] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proc. WWW*, pages 131–140, 2007.

[2] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(4):360–371, May 2015.

[3] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15, Aug. 2011.