Save Time and Money!

Leveraging Transitive Relations for Crowdsourced Joins

Overview

How can we save both time and money?

By reducing the number of crowdsourced joins, as crowdsourced parts are usually much slower and more expensive than machine solutions.

We leverage **transitive property** of entity resolution to reduce the number of crowdsourced pairs.

Eg. If A = B and B = C, then A = C

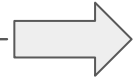
Transitivity

Positive Transitive Relation

If o1 = o2 and o2 = o3, then o1 = o3

Negative Transitive Relation

If o1 = o2 and o2 ≠ o3, then o1 ≠ o3



1. If there exists a path from o to o’ which consists of matching pairs, then (o, o’) is matching
2. If there exists a path from o to o’ which contains a single non-matching pair, then (o, o’) is non matching
3. If any path from o to o’ contains more than one non-matching pair, (o, o’) cannot be deduced

How to implement the Framework?

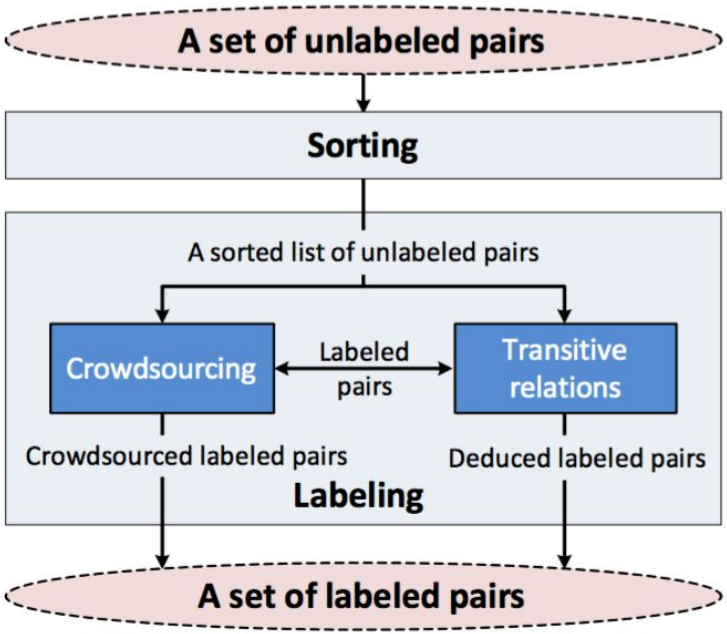


Figure 4: Hybrid transitive-relations and crowdsourcing labeling framework.

VERY important, as getting the right order can reduce the number of crowdsourced joins!

WHY?

Labelling….. Any ideas as to how to do this?

Sorting

Ordering affects the number of crowdsourced pairs!

Eg. p1 = (o1, o2),
p2 = (o2,o3),
p3 = (o1, o3)

If we choose w = (p3, p2, p1), we need 3 pairs crowdsourced
If we choose w' = (p1, p2, p3), we only need 2 pairs crowdsourced

What is the best way to order the pairs?!?!?
What is the **Optimal Labeling Order**?!?!?!

Sorting

Optimal Labeling Order

Given a set of object pairs, the optimal labeling order is to first label all the matching pairs, and then label the other non-matching pairs.

BUT!

Since we don’t normally have ground truth about what pairs are matching before we start crowdsourcing the pairs…..

Any ideas as to how to use this result to optimize labeling order in real life???

Sorting

Expected Optimal Labeling Order*

Given a set of object pairs, and each object pair is assigned with a probability that they are matching, the problem of identifying the expected optimal labeling order is to compute a sorted list of pairs such that the expected number of crowdsourced pairs is minimal using the order.

Labeling

Input: sorted list of unlabeled pairs

Output: list of labeled pairs

NaÏvely, we can send one pair at a time to be crowed sourced, then check based on transitivity whether we send the next!

BUT this is not very efficient.

How do we make it faster?

Labeling

Parallel Labeling Algorithm

Eg. w = <(o1, o2), (o2, o3), (o3, o4), (o2, o4)>

- First pair we must crowdsource
- Second pair also must crowdsource
- Third pair can’t be deduced from previous pairs, thus crowdsourced
- Last pair CAN be deduced based on 2nd and 3rd pair, thus we DON’T crowdsource in this batch
- When crowdsourced results returned, we use transitivity to deduce pairs, then find another list to send to crowdsource
- Repeat till all pairs are labeled

Labeling

Can we make parallel labeling even faster?

Instant Decision (ID)
Imagine instead of waiting for each iteration to return the entire step, before we send the next set of crowdsourced pairs, we take whatever is returned and try to send the next partial set of the next iteration.

Non-matching First (NF)
For each iteration, we crowdsource the ones that are least likely to match first. We assumed that all pairs are matching, so only non matching pairs gives us more information

Experiment

Data:

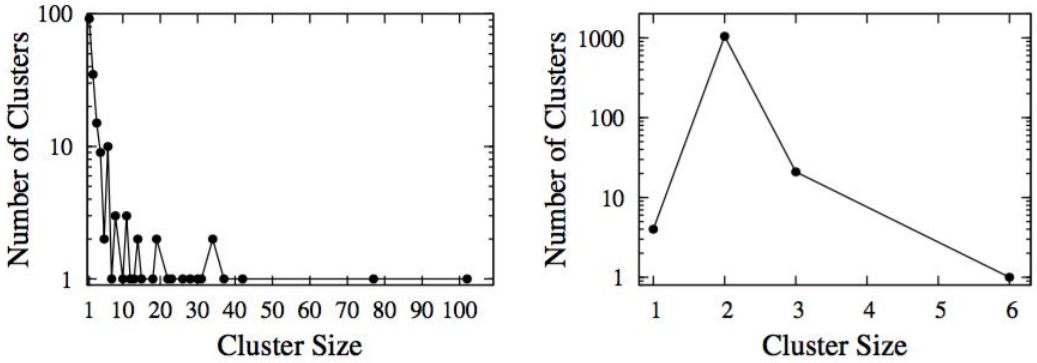


Figure 10: Cluster-size distribution.

Setup:

- Majority vote (best of 3)
- Prequalified workers (need to match all 3 test pairs correctly)
- Non-Transitive just sends it all
- Transitive uses Parallel algorithm with Instant Decision

Results

Sorting Order:

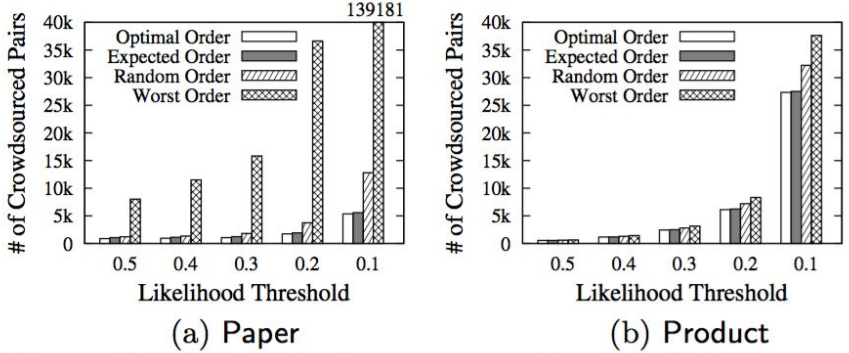


Figure 12: The number of crowdsourced pairs required by different labeling orders.

Effectiveness:

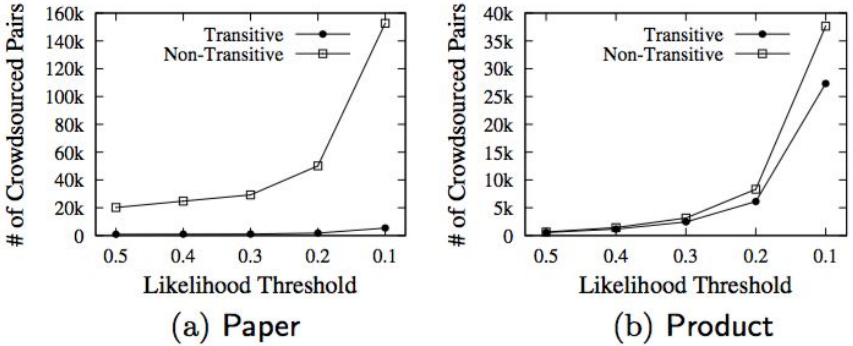


Figure 11: Effectiveness of transitive relations.

Results

Parallel(ID)/Non-Parallel:

Table 1: Comparing Parallel(ID) with Non-Parallel in AMT (likelihood threshold = 0.3).

Dataset	# of HITs	Non-Parallel	Parallel(ID)
Paper	68	78 hours	8 hours
Product	144	97 hours	14 hours

Transitive Speed/Quality:

Table 2: Comparing Transitive with Non-Transitive in AMT (likelihood threshold = 0.3)

(a) Paper

	# of HITs	Time	Quality		
			Precision	Recall	F-measure
Non-Transitive	1465	755 hours	68.82%	95.03%	79.83%
Transitive	52	32 hours	62.96%	90.47%	74.25%

(b) Product

	# of HITs	Time	Quality		
			Precision	Recall	F-measure
Non-Transitive	158	22 hours	95.69%	68.94%	80.14%
Transitive	144	30 hours	94.70%	68.82%	79.71%

Results

Optimizations:

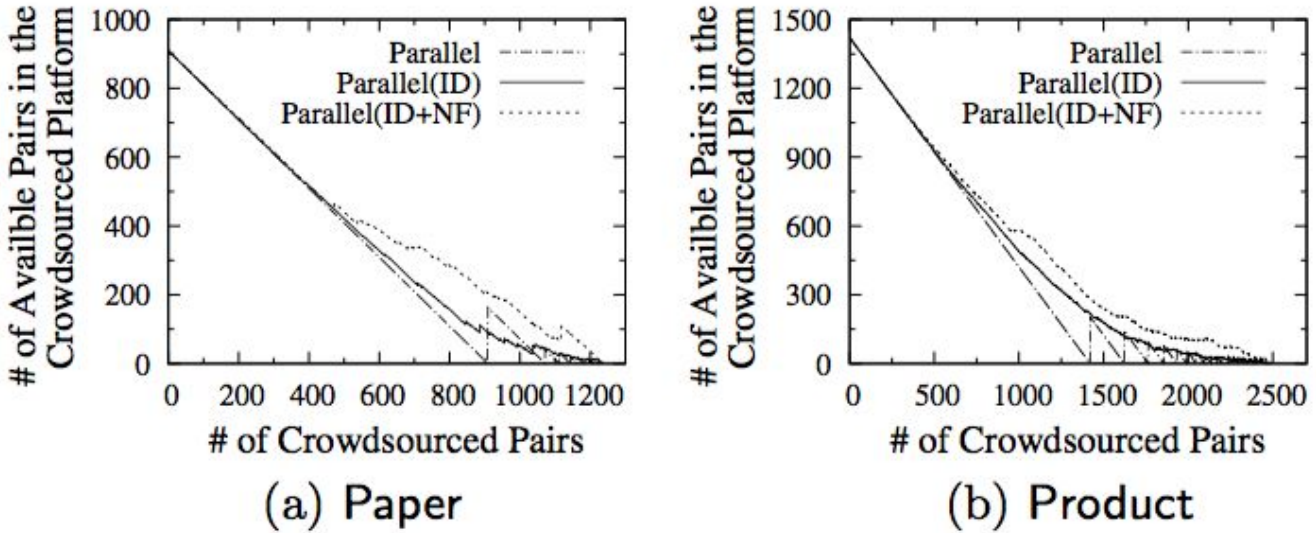


Figure 15: Optimization techniques for the parallel labeling algorithm (likelihood threshold = 0.3).

Summary

Transitivity can reduce time/costs by reducing the number of crowdsourced joins without much loss in quality!

Parallel algorithm is essential to leveraging transitivity in practice

The larger the cluster size, the better your results with transitivity

Transitivity can be used with other existing frameworks

Questions?

Thank You!

Parallel Labeling

Algorithm 2: ParallelLabeling(ω)

Input: $\omega = \langle p_1, p_2, \cdots, p_n \rangle$: a sorted list of unlabeled pairs
Output: $\mathcal{L} = \{(p_i, \ell) \mid 1 \leq i \leq n\}$: a set of labeled pairs

```
1  begin
2      L = {};
3      while there is an unlabeled pair in ω do
4          P = ParallelCrowdsourcedPairs(ω);
5          L ∪= CrowdsourceLabels(P);
6          for each unlabeled pair p ∈ ω do
7              if DeducedLabel(p, L) then
8                  Add (p, ℓ) into L;
9      return L;
10 end
```

Figure 7: Parallel labeling algorithm.

Getting Crowdsourced Pairs

Algorithm 3: ParallelCrowdsourcedPairs(ω, $\mathcal{L}$)

Input: $\omega = \langle p_1, p_2, \cdots, p_n \rangle$: a sorted list of unlabeled pairs
$\mathcal{L}$: a set of labeled pairs
Output: $\mathcal{P}$: a set of pairs that can be crowdsourced in parallel

```
1  begin
2      P = {};
3      Initialize an empty ClusterGraph;
4      for i = 1 to n do
5          if (p_i, ℓ) ∈ L then
6              Insert (p_i, ℓ) into ClusterGraph;
7          else
8              p_i = (o, o');
9              if cluster(o) ≠ cluster(o') and there is no edge
               between cluster(o) and cluster(o') then
10                 Add p_i into P;
11             Insert (p_i, "matching") into ClusterGraph;
12     return P;
13 end
```

Figure 8: ParallelCrowdsourcedPairs algorithm.

Deduce Label

Algorithm 1: DEDUCELABEL(p, $\mathcal{L}$)

Input: $p = (o, o')$: an object pair; $\mathcal{L}$: a set of labeled pairs
Output: ℓ: the deduced label

```
1  begin
2      Build a ClusterGraph for L;
3      Let cluster(o) and cluster(o') denote the cluster of
       objects o and o' respectively;
4      if cluster(o) = cluster(o') then
5          ℓ = "matching";
6      else
7          if there is an edge between cluster(o) and cluster(o')
           then
8              ℓ = "non-matching";
9          else
10             ℓ = "undeduced";
11     return ℓ;
12 end
```

Figure 5: DeduceLabel algorithm.

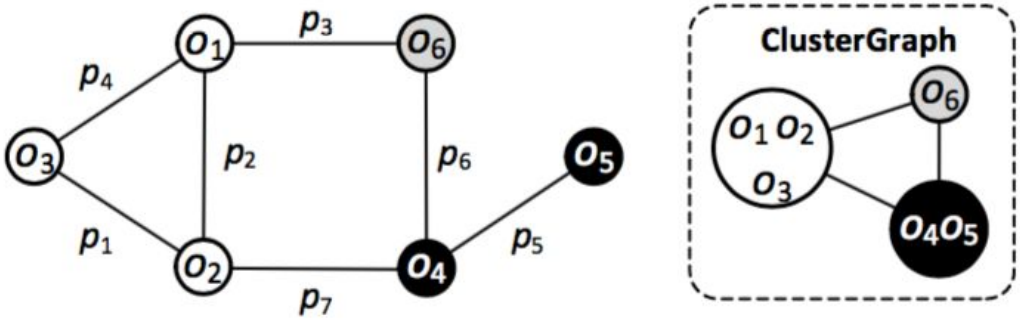


Figure 6: A ClusterGraph built for the first seven labeled pairs $\{p_1, p_2, \cdots, p_7\}$ in Figure 3.