Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 38

Systems Group, Department of Computer Science, ETH Zurich

Memoization of Crowd-sourced Comparisons

by

Florian Widmer

Supervised by

Prof. Donald Kossmann
Sukriti Ramesh

September 2011 - February 2012

Informatik
Computer Science

Abstract

Databases generally adhere to the “closed-world” assumption. If data is not in the database, the database treats it as non-existent. This model works well for things like financial data or inventory. For other data types such as addresses, the data may exist but not be in the database. New systems called crowd-sourced databases now assume an “open-world” and allow a schema to contain columns or even entire tables that are filled with information that is crowd-sourced.

Crowd-sourcing relations between entities is the next step in this development. This allows joins and orderings of data that is difficult to compare computationally but easily compared by humans. This master thesis investigates data-structures and algorithms to make the most out of crowd-sourced relations by exploiting the transitivity inherent to the equality and order relations. Along the way, ambiguities in the data have to be tolerated and resolved. After all, humans are far from perfect and so is the data that crowd-sourcing provides.

Contents

1 Introduction

This introduction provides an example that motivates research in this area as well as a problem statement and an overview over the structure of the thesis.

1.1 Motivation

Imagine you are an enormously wealthy art collector with an insanely large museum (e.g. the Uffizi in Florence). You would like to exhibit all paintings of the same painters in a room each. Unfortunately you've lost all documentation which painting is from which painter.

A very simple method to sort all these paintings into the rooms is to ask an art expert whether two paintings are from the same painter. Of course the expert will expect to get paid and you may want to get your answer as fast as possible. Certainly, you do not want the expert to compare every painting with every other painting.

The nice thing about the equality relation (in this example equality is the painters being the same) is that it is transitive. Utilizing transitivity we can reduce the number of questions that we have to ask the expert by inferring that two paintings are by the same artist.

Instead of using an expert you could also ask several art students whether two paintings are the same and still pay less than for the expert. Most art students will give you a correct answer except for a few rotten apples. By asking several students the same question, you can detect contradictions and weed out wrong answers and still saving money.

1.2 Problem statement

This thesis aims at finding ways to do knowledge inference over crowd sourced equality and ordering relations. Both in the absence and presence of errors introduced by the crowd. The proposed algorithms have to be evaluated for performance in several terms (speed, correctness, cost).

1.3 Structure of the thesis

We've divided the work into three tasks, inference in the absence of errors, inference while tolerating errors and implementation. These correspond to a chapter each after a brief overview of crowd-sourcing and crowd-sourced databases. The second to last chapter presents an experimental evaluation of our work and the last chapter summarizes the insights we have gained and the contributions made by this master thesis.

Equality and Ordering We started out with both equality and ordering in mind but due to time constraints, we had to stop working on the ordering part. The chapter about inference in the absence of errors presents an algorithm that can infer orders but adapting this to tolerate errors was left out as well as experiments investigating the algorithm beyond correctness.

2 Background: Crowd-sourcing

Since not everyone is familiar with the relatively new field of crowd-sourcing, this section gives a short overview of crowd-sourcing and crowd-sourced databases.

2.1 Crowd-sourcing

Crowd-sourcing is a model to solve a problem by breaking it down into smaller parts and distributing each to multiple people (the crowd) to solve, then consolidating the results. The term was first coined by Jeff Howe in Wired Magazine [1] as a portmanteau of *Crowd* and *Outsourcing*. The underlying idea has existed for a long time but with the arrival of the Internet it became possible to distribute such tasks worldwide at a low cost.

Today, marketplaces like *Amazon Mechanical Turk*[1] offer a service to handle distribution and payment of so called *Human Intelligence Tasks*. Registered users can obtain tasks and solve them for a specified payment. Tasks can be of any nature and vary in complexity.

Interestingly, crowd-sourcing is also being used towards illicit ends. The *Koobface* botnet described by Thomas in [2] utilizes Zombies to solve Captchas in order for the Trojan to create fake accounts on social networks and spread further. The Zombie computer is locked and the user is forced to solve the Captcha before being able to keep working. Similarly, spammers may also employ crowd-sourcing to gain access to a large number of fake accounts in social networks and forums that are protected by Captchas or similar means.

Besides distributing the analysis of data, there are also efforts to create algorithms based on *crowd-sourcing*. The Turk-IT toolkit [3] can be used to program crowd-algorithms and to combine human- and computer powered computation.

2.1.1 Caveats

Information that has been crowd-sourced must be taken with a grain of salt and that grain's name is subjectivity. People in the crowd might give different answers based on any number of factors (experience, stress level, payment, good will). Usually tasks are presented to several people and statistical methods are employed to identify a majority answer.

2.2 Crowd-sourced databases

Using people to gather and process information can also be applied to the domain of databases. In this context, there are several possibilities to employ the human mind.

[1]http://www.mturk.com

- Gathering
- Ordering
- Comparing
- Matching

Several database systems have been created to outsource tasks that are traditionally notoriously hard to solve for databases. Examples are Deco [4], Qurk [5] and CrowdDB [6]. CrowdDB allows crowd-sourced columns and tables as well as crowd-ordering and matching. *CrowdDB* uses human input via crowd-sourcing to process queries that neither database systems nor search engines can adequately answer. Computationally difficult functions as well as matching, ranking and aggregating on fuzzy criteria are deferred to the crowd. *SQL* is used in *CrowdDB* to program queries and model data.

The possibility to gather incomplete data from a crowd removes the closed-world assumption of traditional databases. *CrowdDB* is able to generate interfaces to solicit information from human operators automatically. A complex query may be answered in part by regular computation and human input which is combined and returned as the result.

In the case of matchings, rankings and comparisons, *CrowdDB* currently does not retain any information. If the same query is run twice, the information needs to be gathered from the crowd twice. The only data that is retained is crowd-sourced tables and columns.

2.3 The advantages of inference

Crowd-sourcing information is expensive both in terms of time and money (human operators typically expect to be paid). It would therefore be advantageous, if the database was able to infer certain information from already existing knowledge that was possibly obtained through crowd-sourcing. Mathematically speaking, we want to make use of transitivity.

- $(A = B) \wedge (B = C) \rightarrow (A = C)$
- $(A = B) \wedge (B \neq C) \rightarrow (A \neq C)$
- $(A < B) \wedge (B < C) \rightarrow (A < C)$

Using existing knowledge in this way should reduce the number of questions that need to be asked of the crowd. As a part of this process, we might be able to identify interesting questions to ask the crowd to maximize our knowledge.

The next chapter will focus on data-structures that facilitate inference of equality and order assuming perfect data (without contradictions). The chapter after that will focus on how to deal with contradicting answers.

3 First task: inferring knowledge

This chapter deals with the inference of knowledge assuming that the data is "clean" in the sense that there are no contradictions as they would naturally arise from using human operators. It presents data-structures and algorithms to infer knowledge from vote-graphs.

3.1 Order

We are only concerned with "strict greater than" ordering. "Less than" works exactly the same of course. In the context of crowd-sourcing we will have to deal with partial orders. One way to imagine associations between entities being compared is a directed acyclic graph (DAG).

$$\begin{tikzcd}
 & & A \arrow[dl] \arrow[dr] & & \\
 & D \arrow[dl] & & B \arrow[dlll] \arrow[dr] & \\
E & & & & C
\end{tikzcd}$$

Figure 1: Example DAG

In the DAG in figure 1, the entities that are being compared are the nodes. There is an edge from A to B if the crowd told us that $A > B$. An entity C is smaller than A if C is reachable from A as the relationship is transitive. DAG reachability and transitive closure has been studied and algorithms exist to solve this problem. See [7, 8, 9, 1, 10] and many others. The ordering problem maps nicely to the question of reachability in the DAG.

3.1.1 DAG reachability algorithm

We've implemented a very simple and intuitive algorithm developed by Agrawal, Borgida and Jagadish [11] using database tables as data storage.

This algorithm uses a spanning tree and post-ordering of the tree's nodes. In a nutshell it works like this:

1. Construct a spanning tree over all nodes of the DAG (a virtual root node can be used if the graph has several components).
2. Assign post-order numbers to each node in the tree.
3. Each node is assigned an interval starting with the lowest post-order id in its subtree and ending with its own post-order id.

4. For each edge that is not in the spanning tree, the source node and its ancestors inherit all intervals assigned to the target node (do this in topological order).

5. Overlapping intervals can be merged.

See figure 2 for the same DAG as in figure 1 after this procedure. The edge BE is not in the spanning tree and node B has inherited the interval of node E. Node A also inherits this interval but it was merged since they overlap.

A reachability query is answered by checking the post-order id of the node that is the target of the query, and then checking whether this id is in any of the intervals of the source node. If neither node is reachable from the other one, the answer is “unknown”. Section 3.2.1 explains how “unknown” ties into binary logic.

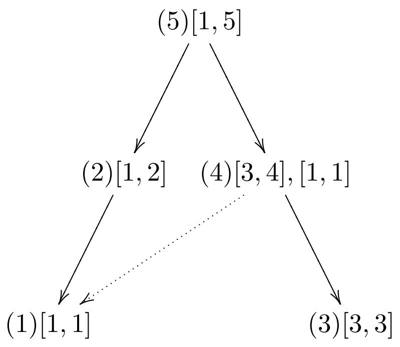


Figure 2: Post-order ids and intervals.

Insertion of edges can be accommodated quite easily using a larger increment in assigning post-order numbers.

3.1.2 Data structures

The data-structures for this algorithm are very simple, please refer to table 1 for an example. There can be several intervals for a node in table 1b that encode directed edges that were not part of the spanning tree.

3.2 Equality

As with orders, equality in the realm of crowd-sourcing is only partially defined. Two objects can either be equal or unequal and as a third option, their relation might be unknown. Keeping groups of equal objects in disjoint sets and storing identified inequalities between these sets solves the problem.

Source	Target
A	D
D	E
B	E
B	C

(a) Edge table

Node	Start	End
A	1	5
D	1	2
E	1	1
B	3	4
B	1	1
C	3	3

(b) Intervals table

Node	Post-order ID
A	5
B	4
C	3
D	2
E	1

(c) Nodes table

Table 1: Data structures of the ordering algorithm for the example in figure 2

3.2.1 Logic

The answers "equal" and "unequal" need to be supplemented by a third "unknown". We will use the symbol $\star$ to denote "unknown". This extends the definition of transitivity:

- $A = B \wedge B = C \rightarrow A = C$
- $A = B \wedge B \star C \rightarrow A \star C$
- $A = B \wedge B \neq C \rightarrow A \neq C$
- $A \star B \wedge B \neq C \rightarrow A \star C$

3.2.2 Data-structure

A data-structure to store disjoint sets is needed, while a simple two column table is enough to store inequalities in a database. The indexes serve the same purpose as trees in other implementations. A discussion of disjoint set data structures can be found in Cormen's Introduction to Algorithms [12].

Object	Representative
A	A
B	A
C	C
D	C
E	E

(a) Representative table

Representative 1	Representative 2
A	C
C	E

(b) Inequality table

Table 2: Data-structure for equality

Equality In table 2a each row represents an entity. The entity belongs to a group that is represented by a "representative". If two entities have the same representative, they are in the same group and therefore equal.

Inequality Each row in table 2b represents two groups that are unequal to each other. In the example, the groups with representatives A and C are unequal. There is no information about the relationship of groups A and E, it is therefore “unknown”. The lexicographically smaller representative is stored in the left column to make retrieval of rows easier.

3.2.3 Query

A query whether $A = B$ is as simple:

1. Check the representatives table for both entities

   If their representatives are the same, they are equal

2. Check if the two representatives of the entities that are being compared are in a row of the inequalities table

   If such a row exists, the objects are unequal

3. If you reach this point, the answer is unknown

4 Second task: tolerating contradictions

The previous section only dealt with perfect data. The algorithms presented there would not work if there was a single contradiction in the data. Using crowd-sourcing, there will be ambiguity. This section explains how to do inference in the presence of errors but only for equality, ordering did not fit in the time-frame of this master thesis.

4.1 Semantics of inference

There are two ways to look at crowd-sourcing equality relations, on-demand and opportunistic.

On-demand If the system is in control of which questions to ask to the crowd, it will only ask questions to which it cannot infer the answer. If we can infer an answer, we will never have to ask the crowd directly for it and will never get a contradicting answer from the crowd. This means that the system maintains a conflict free graph.

Opportunistic If data is loaded in bulk, we cannot afford this luxury and have to tolerate the inconsistencies. Figure 3a and the corresponding example in section 4.3.1 show that different paths exist along which one can infer whether two objects are equal or not. The only additional information available is how many people assert that each step is correct.

4.2 Scoring functions

A scoring functions is needed as a measure to judge how good an inference is. For our purpose it has to satisfy several criteria:

1. Consistency: The scoring function should not contradict the votes.
2. Convergence: For every comparison there should be a sequence of votes so that the scoring function decides ”Yes” (e.g., three successive ”yes” votes”). There should also be a sequence of votes so that the scoring function decides ”No” (e.g., three successive ”no” votes).
3. Symmetry: $a = b \to b = a$ and $a \neq b \to b \neq a$.
4. Transitivity: $a = b \land b = c \to a = c$
5. Anti-transitivity: $a = b \land b \neq c \to a \neq c$
6. Resilience: A set of malicious votes cannot derail the system if we have sufficient correct votes.

Path-based We investigated path-based scoring functions. This means we imagine all votes as an undirected graph with the entities being compared as nodes and two types of weighted edges ("equal" and "unequal").See figure 3a for an example. To evaluate an inference we look at all possible paths between the two nodes.

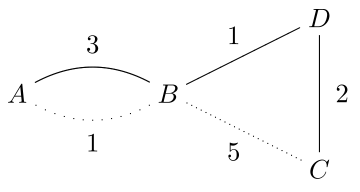
Finding a scoring function As a first scoring function we considered the sum of the score of all paths, where the score of a path is the minimum weight of its edges, the *min-sum* scoring function. Fagin and Wimmers [13] explain why this is a weighted scoring function; in this case the weight was the number of paths. We decided against this, because a large number of low-weighted paths (presumably minority votes) can overrule a single high-weighted path (majority vote). Also, if circles were allowed *min-sum* even broke the condition of transitivity as we were able to draw inferences of the form $A = B \wedge B = C \wedge A \neq C$.

4.3 Minimum-Maximum scoring function

We finally decided on a scoring function that we think reflects intuitively the way one would reason about a graph of votes. Unfortunately this function does not satisfy the definition of transitivity fully.

4.3.1 Definition of paths

In order to deal with the problems imposed by contradictions in the original data, we modeled the information that is given by the crowd as a graph. Objects are nodes, assertions that two objects are equal are weighted edges and assertions of inequality are weighted edges of a different type. Figure 3a shows a very simple graph with the associated assertions in figure 3b. All weights are positive but edges representing inequality assertions will be referred to as "negative" edges while the others are called "positive".



(a) Example graph

Votes	Assertion
3	$A = B$
1	$A \neq B$
5	$B \neq C$
2	$C = D$
1	$B = D$

(b) Votes

Figure 3: Crowd-sourced equality relations

The information in figure 3 gives us several paths along which we can infer new knowledge (the list is not complete):

- The path ABD using the edges with weights 1 is referred to as a negative path, as it contains an inequality edge (AB) that asserts inequality.
- The path ABD using the edges with weights 3 and 1 is referred to as a positive path, as it only contains equality assertions.
- The path ABC using the edges with weights 1 and 5 is an illegal path as it contains more than two edges asserting inequality. This is mathematically obvious as $A \neq B \wedge B \neq C \nrightarrow A \neq B \vee A = B$

Illegal paths will not be used by the scoring function, as they contain no information. Positive paths are inferences for equality, negative paths are inferences for inequality. The choice of this function is based on several considerations.

Path strength Any path can only be as strong as its weakest link. Therefore, a path is scored by its lowest ranking vote. The positive path ABDC in Figure 3a has weights 3,1 and 2 so the entire path is scored with 1.

Inference strength If there are more than one paths of each type (positive and negative) we select the score of the one with the highest strength among those, so we have two scores; one for the positive paths and one for the negative paths.

Circles An interesting property of this approach is that it disregards circles in a path. If a path has a circle there must also be the same path without the circle. Either their path strength is the same, or if there is a lower edge in the circle, then the direct path wins the comparison.

Final verdict The final step to score an inference is to compare the scores for equality and inequality. We use a quorum to have a sufficiently large majority, anything else yields the answer “unknown”.

To recapitulate:

1. Find all legal paths between the two nodes that are part of the inference
2. Score each path with its lowest edge weight
3. Pick the positive and negative path with the highest scores each
4. If one of the scores is above quorum that’s your answer, else it’s “unknown”

4.3.2 Philosophical observations

A very interesting aspect of scoring functions is how they reflect the way we reason about knowledge. Our choice of scoring function was made not only because of mathematical properties but also because we think that this is the way a person would intuitively look at the vote-graph and compare the entities.

Path scoring As explained above, each path is scored by its minimum weight. This represents the idea that a chain is only as strong as its weakest link. Contrary to a logical calculus like natural reasoning where every step has the same value, we have a weight associated with each step that represents our confidence in it.

Picking the best path Only the two highest ranking paths supporting equality and inequality are evaluated against each other. The underlying idea is that these are the two majority votes. These are the two paths where most people voted in support. We do not need to evaluate the minority votes, even though they could be numerous. The one path most people agreed on makes more sense than a lot of different paths proposed by a small number of people each.

4.4 Relaxing transitivity

The Minimum-Maximum scoring function does not model transitivity perfectly. We call the new definition of transitivity "weak transitivity" because it gives weaker guarantees.

- $A = B \wedge B = C \rightarrow A = C \vee A \star C$
- $A = B \wedge B \neq C \rightarrow A \neq C \vee A \star C$

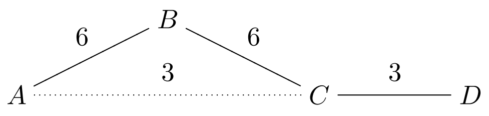


Figure 4: Weak transitivity: $A = C$ $C = D$ $A \star D$

In figure 4 the comparison of A and C yields a positive score of 6 and a negative score of 3. Imagine the quorum to be 3. Our scoring function gives us $A = C$ as a result. For C and D we only have a positive score of 3, which also gives us $C = D$. For A and D however, the scoring function gets a score of 3 for both positive and negative paths (the reason being that the edge of weight 3 between C and D is the lowest edge in both paths) therefore $A \star D$.

4.4.1 Ways to deal with weak transitivity

Adapting the scoring function to the data-structure from section 3.2.2 does not work so well however. This data-structure only stores data that satisfies "strong" (traditional) transitivity. Therefore, we now have two options to implement a system using our scoring function.

Accurate This will produce exactly the result of the Minimum-Maximum scoring function. To achieve this, either a data-structure using more space (to fit memoized paths) has to be used or we have to invest more time and compute the scoring every time (discovering paths).

Fuzzy We chose another direction for implementation. This will use the data-structure from section 3.2.2 as a cache and modify it as new information is available. Weak transitivity is ignored, only information that satisfies (traditional) strong transitivity is retained and given as answers to queries (with the inclusion of "unknown"). It will only put entities into groups that are clearly all equal to each other while the *fuzzy* edges that are only equal to some entities and not others are treated as unknown.

The implementation in the next section is based on this approach. It will return "unknown" if the entities are in two different groups even if the direct comparison of the edges would suggest that they are the same. This is fundamentally biased towards giving"unknown" as an answer unless other edges than the one being queried are also improved. The approach has to be improved by a method to identify the right questions to ask to the crowd. This will be presented in section 5.4.

Example Imagine figure 4 without the edge AC. The scoring function will tell you that all the entities are equal to each other ($A = D, C = D$). With the introduction of the edge AC the *accurate* implementation would give the answer that $A \star D$ but $C = D$. On the other hand the answer of the *fuzzy* implementation will be that $A \star D, C \star D$

5 Implementation: trading accuracy for speed

Implementing the min-max scoring function is difficult as all paths have to be evaluated. We settled for a slightly inaccurate version that we called fuzzy in the previous section. During the experiments we discovered that the implementation has a fundamental flaw, if the reaction to a query being answered with “unknown” is to just crowd-source that edge. Section 5.4 addresses these problems and proposes solutions.

5.1 Approximation

Faced with the high cost of discovering all possible paths we had to find an alternative way to implement the Minimum-Maximum scoring function. The fuzzy variant from the previous section which would use the data-structure from section 3.2.2 as a cache looked promising. The challenge is to find a way to store the necessary information so that we can give the same result as the scoring function in most cases.

Shortcomings As mentioned in the previous section, the mapping from the Minimum-Maximum scoring function to the data-structure is not perfect. By enforcing strong transitivity, we lose some information. The information that is lost is exemplified by figure 4. By requiring that all entities that are said to be equal satisfy strong transitivity, we lose the information that the scoring function also considers $C = D$.

5.1.1 Breaking down the scoring function

Our breakdown of the scoring function is not perfect. We identified several conditions that reflect what the scoring function does, but as mentioned before, this has some shortcomings.

A condition for group membership The scoring function sets those entities equal that have a positive path to each other which is at least higher than the best negative path plus a quorum. The union-find data-structure stores groups of entities that are all equal to each other. If we can identify entities that would be ruled equal to each other by the scoring function, we can store them in the data-structure. We will refer to these groups of entities as Groups.

Instead of verifying all paths in a group we pick an entity in the group and verify that the scoring function rates all other entities in the group equal to it. We do this by ensuring that there is positive path from each entity to the group representative. We conclude from this that the entity with the lowest scoring path to the representative can be reached by all other entities in the group using a positive path of the same score. The highest negative path inside the group then has to be at least quorum lower than this path to guarantee that all entities are ruled equal by the *min-max* scoring function.

In order to avoid the high cost of discovering the best paths each time, we utilize a spanning tree to quickly identify the best paths from an entity to its group representative. A spanning tree is also easy to update if new edges arrive. Also, all highest-ranking negative paths between group members will use these best paths plus one negative edge. The negative edge in figure 5 shows that using the tree of best positive paths and a negative edge, we can construct all negative paths inside a group that would be considered by the scoring function.

Definition: Group tree The spanning tree composed of the best path from group members to the representative.

Definition: Group positive Minimum The lowest weighted edge in the group tree. Also referred to as *minpos*.

Definition: Group negative maximum The highest weighted inequality edge between two nodes in the same group. Also referred to as *maxneg*.

Condition: Group Condition A group must satisfy the following condition:

$$minpos - quorum \geq maxneg \tag{1}$$

If the group condition condition is no longer met by a group due to a change of weight in an edge, the group is dissolved. Entities are added to groups if the group condition is still satisfied afterward.

Example Figure 5 shows an example of a group tree. The dotted edge is negative. The group positive minimum is 4 while the group negative maximum is 1. For each entity, there is a positive path of at least score 4 and a negative path of at most score 1 to each other entity. Other edges between the entities may exist, but they are not part of the best paths (they are of lower weight) This means that the Minimum-Maximum scoring function will call all of these entities equal to each other. Imagine now that the weight of the negative edge is increased to 3. The group condition is broken, because the scoring function would rate $Y \neq W$.

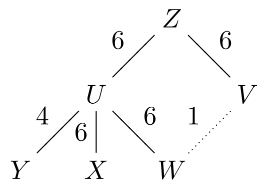


Figure 5: A Group tree

A condition for inequality Inequalities between groups are understood as all entities of one group being unequal to all entities of another group. We need a condition that will ensure that the scoring function will give us this result.

Definition: e^+ The highest weighted positive edge between two groups is referred to as e^+.

Definition: e^- The highest weighted negative edge between two groups is referred to as e^-.

Condition: **Inequality condition** Two groups A and B are considered unequal if the following condition is satisfied:

$$\begin{aligned} \min(\text{minpos}(A), \text{minpos}(B), e^-) - \textit{quorum} \geq \\ \min(\text{maxneg}(A), \text{maxneg}(B), e^+) \end{aligned} \tag{2}$$

The inequality condition ensures that a negative path exists from every entity in both groups to every other entity in both groups that is at least by quorum higher than the best positive path from each entity to each other entity. The paths are constructed by using *minpos* and *maxneg* and the best connecting edge of either type. Minimum is used on both sides of the condition because we mimic the behavior of the scoring function in evaluating the two paths. Their score is the minimum of the edges along the path.

Example Figure 6 shows several groups. Between groups A and D we have a negative edge of weight 9 and a positive edge of weight 1. These edges do not necessarily have to be connected to the same nodes. Inside group A we have a positive path between any two points of at least $\text{minpos}(A) = 5$ also, we know that any positive path inside group D is at least $\text{minpos}(D) = 8$ we can conclude that a negative path from anywhere inside A to inside D will use the negative edge with weight 9 and be the minimum of the three values. The same goes for the positive path.

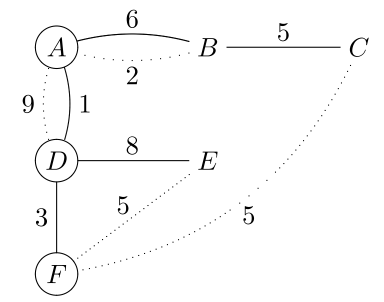


Figure 6: Three groups, represented by A, D and F

Condition: **Merge condition** Two groups A and B are considered equal and can be merged if the following condition is satisfied:

$$\begin{aligned} \min(\text{minpos}(A), \text{minpos}(B), e^+) - \textit{quorum} \geq \\ \max(\text{maxneg}(A), \text{maxneg}(B), e^-) \end{aligned} \tag{3}$$

The merge condition is only true when a positive path exists between any node in either group that is higher by at least quorum than any of the negative paths. Maximum is used in this condition because the group condition (1) considers the highest negative edge inside the group. We take the maximum of three maxima (Group A, Group B and the edges in between) and thereby achieve the combined maximum. The minimum on the left side of the condition is used to construct the lowest path to the new representative much in the same way as in the inequality condition (2).

Example Imagine quorum to be 3. The groups D and F in figure 6 could satisfy the merge condition if the positive edge between them rose to 8. Currently these groups do not satisfy neither the merge nor the inequality condition, so they are referred to as *candidates*. It is also possible that two groups do not satisfy the merge condition because one of them has an internal negative edge whose weight is too high.

Candidates During the following discussion of data-structures and algorithms we will repeatedly refer to candidates. These are edges between groups where neither equality nor inequality can be established.

5.2 Data-structures

We need to store several different types of information:

- The actual votes provided by the crowd
- Group membership of entities
- Inequalities between groups
- Data simplifying the evaluation of our conditions

The following structures store the necessary information.

5.2.1 Votes table

Votes are stored as edges in a graph, for simple retrieval, the lexicographically lower entity is stored in the first column. Table 3 shows an example.

5.2.2 Groups table

Groups are stored in a table much in the same way as described in section 3.2.2. Table 5 shows the data for the example in figure 6. The first two columns store the node and its representative while the last column stores the score of the best positive path to the representative.

5.2.3 Best path tree

The best path tree stores the tree of best paths for a group. Table **??** shows an example.

Entity 1	Entity 2	Votes	Type
A	B	6	=
A	B	2	≠
B	C	5	≠
A	D	1	=
A	D	9	≠
D	E	8	=
D	F	3	=
E	F	5	≠
C	F	5	≠

Table 3: Votes table for figure 6

Parent	Child
Z	U
Z	V
U	Y
U	X
U	W

Table 4: Best path table for figure 5

Node	Representative	Best Path
A	A	inf
B	A	6
C	A	5
D	D	inf
E	D	8
F	F	inf

Table 5: Group table for figure 6

5.2.4 Inequalities and Candidates tables

To store all necessary information about inequalities and candidates that enable us to evaluate the inequality and merge conditions we need two tables. Table 2b is an example for an inequality table, the candidates table 7 has exactly the same structure. The "negative" and "positive" column store the highest positive and negative edge weight between the groups. The last two columns are used to keep track of multiple edges with the same weight.

Group A	Group B	positive	negative	# positive	# negative
A	D	1	9	1	1
A	F	$\emptyset$	5	0	1

Table 6: Inequalities table for figure 6

Group A	Group B	positive	negative	# positive	# negative
D	F	3	5	1	1

Table 7: Candidates table for figure 6

5.3 Algorithm

All computation is done eager, but also lazy implementations are possible.

5.3.1 Query

The data-structures can be queried for the equality of two entities exactly in the same way as is described in section 3.2.3. Only the groups and inequalities table need to be queried.

5.3.2 Inserting new information

In the current version of the implementation, the data-structures are updated immediately after insertions. All conditions are checked and dealt with accordingly.

Group condition broken If the group condition is broken, the entities in the group are no longer all equal to each other and the group has to be broken into several groups. This is most easily achieved by removing all information about the entities in the group from all tables and reinserting the votes again, starting with the inequality assertions. Reinserting the votes including the one that broke the group condition rebuilds the groups in a legal state.

Merge condition satisfied If the merge condition is satisfied, this means that also the group condition is satisfied for both groups now. The groups are merged by attaching the best-path-tree of one group to the other group using the best edge between them. Of course the tree has to be updated.

Inequality condition satisfied When a row in the candidates table satisfies the inequality condition due to new votes, this means that all entities of one group would be rated unequal to all entities in the other group by the scoring function. To reflect this, the row is moved from the candidates to the inequality table.

Inequality condition broken If the inequality condition is broken for a row in the inequality table, the row is moved to the candidates table.

5.4 Problems and proposed solutions

A fundamental problem of this algorithm is that it can run into a situation where groups with low edges cannot merge with another group even though the positive value of the candidate is very high.

Example In figure 7: There are two groups with representatives K and H. Whatever the value of the edge KH, the groups will not be merged even though the scoring function would tell us that $L = H$ because the merge condition cannot be satisfied. The negative edge of weight 10 is not a problem in group H because the positive edge is much higher. However, for the groups to be able to merge, the edge KL needs to have a higher weight. The algorithm and data-structures do not recognize this situation. A query whether $K = L$ will be answered as “true” and no one will see the need to crowd-source that edge. A query whether $K = H$ will return “unknown” but curtsying that edge will not help the situation.

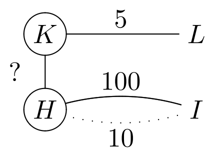


Figure 7: Problematic graph

Improvements It is not always best to just crowd-source the queries that are answered with “unknown”. We need hints as to which questions to ask next. Our current implementation only returns “unknown” if it does not know the answer, and the benchmark keeps asking the crowd the same question again. We need to identify the edges that break the conditions (group-, inequality- and merge-condition) and crowd- source these edges.

Examples

- After the update of a candidate, the positive and negative values are further than quorum apart. This indicates that we should check why the merge- or inequality-condition are not satisfied. The check is rather simple, we know what values are part of the condition, and whether they are evaluated to maximum or minimum. The dominating value is the one that is prohibiting a merge or inequality. All edges that are in the best path tree and have a weight of this value should be crowd-sourced to maximize our knowledge.
- If a query returns “unknown” but there is an entry in the candidates table, we should ask the crowd about the edges in both groups that have the same weight as those considered by the inequality and merge conditions (equations (2) and (3)). This way we increase our knowledge about the edges that matter.

6 Measurements

To measure the performance of the algorithm that is presented in chapter 5, we had to invent our own benchmark as there is neither a given benchmark for this type of problem nor data-sets modeling crowd responses for entity resolution. This chapter explains how we model the crowd and measure the relevant properties of our system.

Please also keep in mind that the algorithm for *inference* has problems which are more closely outlined in section 5.4. This results in a very high incompleteness rating.

6.1 Methodology

We want to investigate the following properties of our system:

1. Cost - How many questions did we ask to the crowd?
2. Correctness - How many wrong answers did we give?
3. Completeness - How often were we not able to give an answer?
4. computational cost - How much space and time did it take to get the answers?

A model for the crowd We model answers from the crowd with a given error rate. Our benchmarking program has a stored "ground-truth" that represents the world as it really is. Every time the crowd is asked, a coin toss determines whether the crowd gives the correct or the wrong answer. Of course the coin toss is not 50-50 but can be tuned to a specific value. Every answer from the crowd is inserted into the database. It can happen that there are questions which cannot be answered by asking more questions to the crowd. For this reason, we also enforce an upper limit of votes on an edge. Once this limit is exhausted the database returns "unknown and don't ask the crowd anymore" as an answer. This is also logged as an unknown answer.

A model for queries We are not concerned with specific access patterns so we pick two entities at random and ask the database to tell us whether they are equal or not.

Sequence of events If the database does not know an answer, we have to ask the crowd. The pseudocode in algorithms 1 and 2 explains how we coupled the system with the crowd. Of course one could go back to ask the crowd indefinitely, but that will possibly take forever. We've decided to stop after a fixed amount of tries.

Algorithm 1 Pseudo code for the crowd

```
function ASKQUESTION(query)
    $correct \leftarrow$ truth. getCorrectAnswer($query$)
    for $i = 1 \rightarrow bulk$ do
        $d \leftarrow$ twistanswer($correct$)
        db. insertVote($d$)
    end for
end function

function TWISTANSWER(query)
    $t \leftarrow$ coin. toss($errorrate$)
    if $t = head$ then
        return $\neg correct$
    else
        return $correct$
    end if
end function
```

Algorithm 2 Pseudo code for the benchmark

```
while benchmarking do
    $query \leftarrow$ generateQuery()
    $answer \leftarrow$ db. askQuery($query$)
    $counter \leftarrow 1$
    while $answer = unknown \wedge counter < limit$ do
        crowd. askQuestion($query$)
        db. askQuery($query$)
        $counter \leftarrow counter + 1$
    end while
end while
```

6.1.1 Parameters

The above description of the methodology gives us the following parameters to vary:

Parameter	Values (Default)	Symbol
Number of entities	5000	S
Distribution of entities among groups	Zipf ($s = 0.5$), Uniform	dist
Error rate	0, 5,10,15	e
Quorum	3,5	q
Upper limit on crowd access	5	l (limit)
Number of people asked simultaneously	5	b (bulk)
Number of queries	5 Million	QN

Table 8: Parameters

6.1.2 System under test

The system which we are measuring is the database equipped with the algorithm to do equality inference in the presence of errors and contradictions. Time measurements are taken while a query is executed on this database by the benchmark driver (JDBC connection on local machine). The benchmark driver is single threaded waits for the answer before issuing another query, making this a closed system.

Hardware Intel Xeon 64 bit, 2.8 GHz, 4 cores, 8GB Ram

Software Ubuntu 10.04 , Postgres 9.1

6.1.3 Baselines

As baselines we used two other implementations: “Always” and “Caching”.

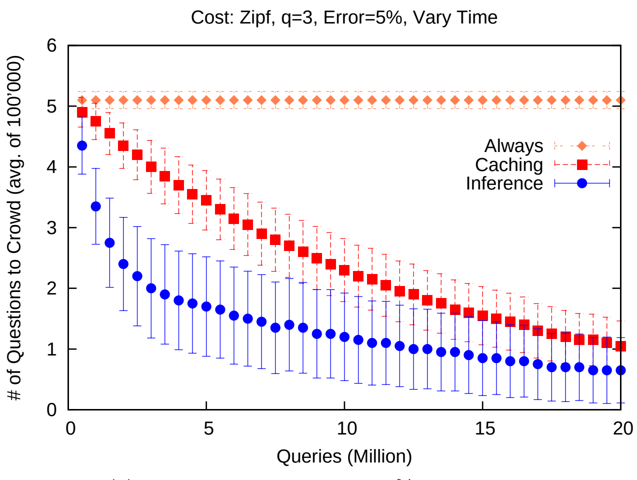
Always crowd This implementation always asks the crowd until it has a quorum or exceeds the upper limit of votes and then gives the answer accordingly.

Caching This implementation works the same as always crowd, except that it remembers the answer and does not ask the crowd if it gets the same query a second time.

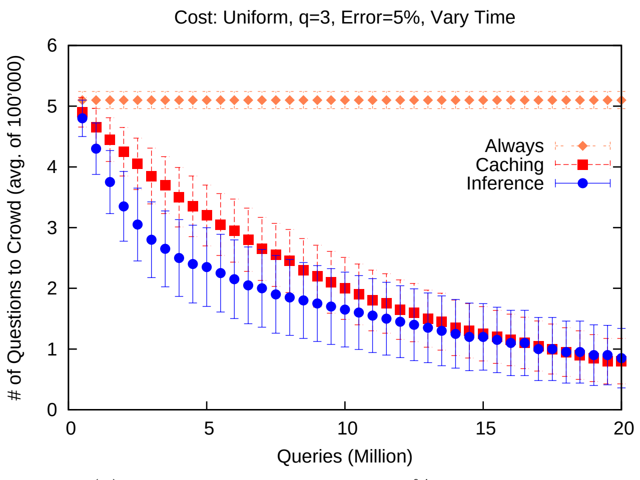
6.2 Experimental results

This section presents the results of our measurements. Each group of experiments is followed by a short discussion of the impact of the results. The first section shows graphs for both Zipf and uniform distribution, the later sections leave that out because the results are roughly the same.

6.2.1 Cost



(a) Cost: Zipf, q=3, Error=5%, Vary Time



(b) Cost:Uniform, q=3, Error=5%, Vary Time

Figure 8: Cost vs Time

Cost over time Figures 8b and 8a show the cost of answering queries in relation to time (represented as number of queries asked so far. With a low error rate such as 5% this cost goes down both for the inference and caching approach. The always crowd method has the same high cost in all cases which is not surprising.

Cost for different error rates Figure 9 shows the cost as the total number of questions to the crowd after 5 million queries. The effect of a higher error rate is an increase of cost in all methods as expected. However the increase for inference is much more dramatic. This can be attributed to more conflicts that have to be resolved by asking more questions.

6.2.2 Correctness

Correctness was measured in terms of how many incorrect results were returned by the database. This doe snot include unknown answers as they are treated in the next section. Figure 10a shows the effect of the error rate on the total number of wrong answers for the different methods. *always* and *caching* have similarly high rates at error rates above 5% while *inference* manages to stay much lower. The conclusion to draw from this experiment is that *inference* is worth the investment in terms of questions to the crowd.

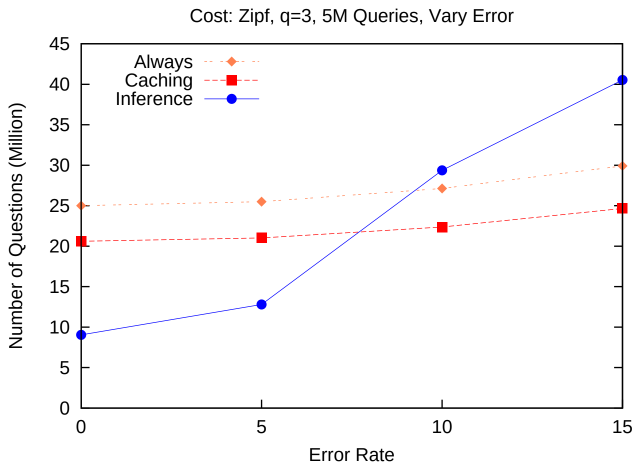


Figure 9: Cost: Zipf, q=3, 5M Queries, Vary Error

6.2.3 Completeness

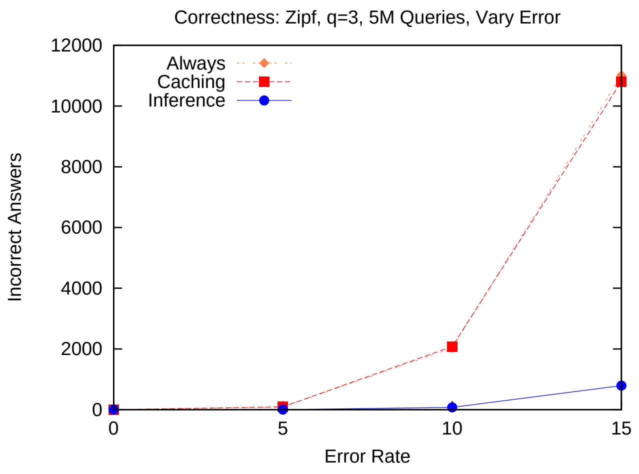
While the *inference* approach was shown to have a much better correctness rating than the other approaches in figure 10a, there is another side to that medal. Completeness was measured in terms of how many "unknown" answers were given by the system. As figure 10b shows, *inference* is much more incomplete. For high error rates the incompleteness can go up to 20% of the queries. On the upside we can say that *inference* is very conservative. It refuses to give you answers when there is confusion, but it could do much better in resolving the conflicts.

This high incompleteness rate is due to problems in the algorithm outlined in section 5.4.

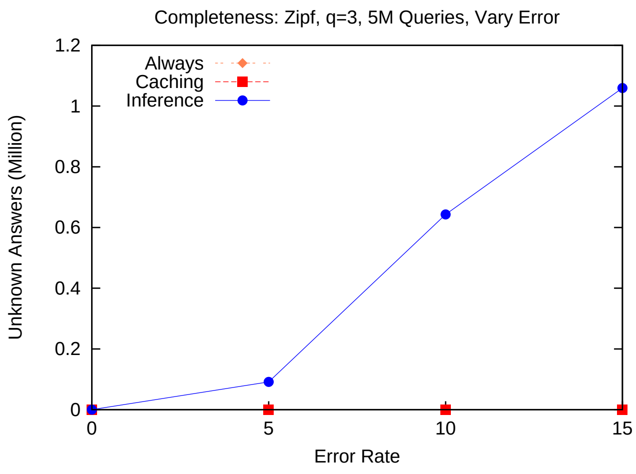
6.2.4 Effect of quorum

The effect of quorum turned out to be significant for the *caching* and *always* approaches. After we ran a first series of experiments with quorum 3 and a second series of experiments with quorum 5 we saw an improvement in correctness but a loss in completeness for these two methods while *inference* took away better correctness at a higher cost for all approaches. The completeness even improved slightly for *inference*.

Figure 11 shows the correctness of *inference* and *caching* at varying error rates for quorum values of 3 and 5. A massive improvement in the correctness of the *caching* approach is apparent in figure 11b, the number of incorrect queries dropped to 44. The number for inference is only marginally better at 35 but with far less improvement.



(a) Correctness: Zipf, q=3, 5M Queries, Vary Error



(b) Completeness: Zipf, q=3, 5M Queries, Vary Error

Figure 10: Correctness and Completeness vs. Error rate

	Always		*Caching*		*Inference*	
Error Rate	*avg*	*var*	*avg*	*var*	*avg*	*var*
0%	0.001	0.001	0.21	0.46	0.29	18.18
5%	0.001	0.001	0.21	0.23	2.76	70951.11
10%	0.001	0.001	0.22	0.30	13.84	175122.16
15%	0.001	0.001	0.23	0.28	93.21	2557591.59

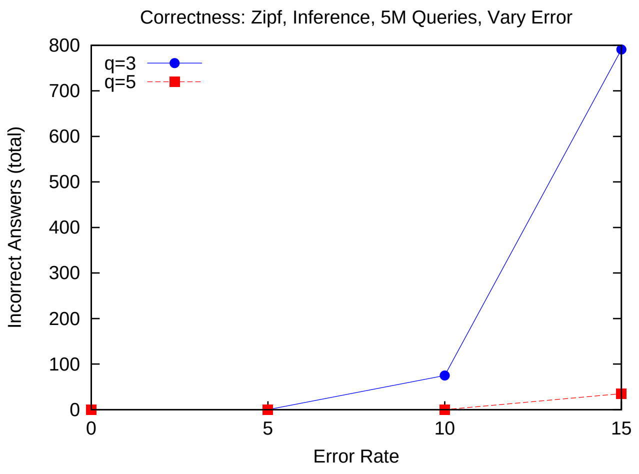
Table 9: Response Time [msecs]: Zipf,q=5, 5M queries, Vary Error Rate

Incompleteness is still very high for *inference* with quorum 5. For *caching* it rose from 152 to 612 unknown queries but this is no relation to the around 870'000 unknown queries for *inference*

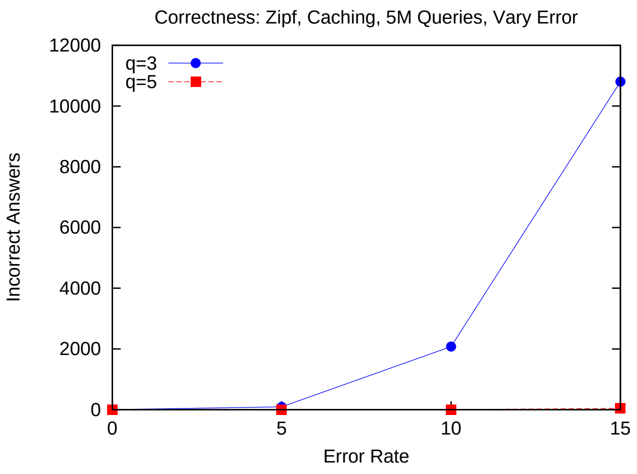
6.2.5 Computational cost

The previous section about cost measured questions to the crowd. Of course, every algorithm can also be evaluated for complexity. It is important to keep in mind that crowd-sourcing has an inherently high run time, as the crowd is very slow in comparison to traditional database systems. The effect of the crowd was left out in these experiments deliberately, as we only wanted to evaluate the algorithms and data-structures themselves. The time to simulate the crowd however is included but marginal.

Response time (table 9) is very good for *always* and *caching* while *inference* sees an increase with increased error. This was to be expected as more errors mean more contradictions to resolve. The run times are still very small however,



(a) Correctness: Zipf, 5M Queries, Inference, Vary Error



(b) Correctness: Zipf, 5M Queries, Caching, Vary Error

Figure 11: Correctness vs. Error rate for different quorum

Error Rate	*Always*	*Caching*	*Inference*
0%	0	602	662
5%	0	603	891
10%	0	605	1903
15%	0	605	5501

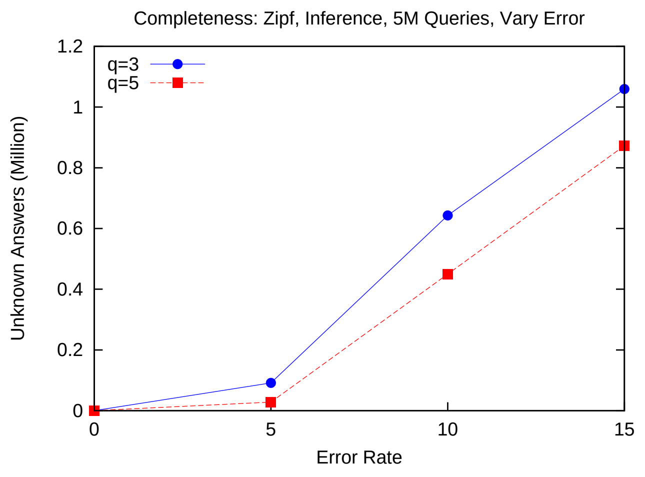
Table 10: Storage [MB]: Zipf, q=5, 5M queries, Vary Error Rate

compared to the time it takes to get answers form a crowd. The high variance is explained by the difference

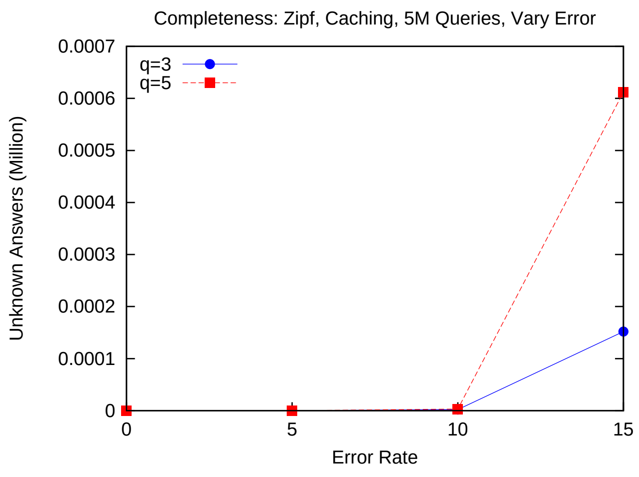
The *always* approach does not use any space to store data while table 10 shows constant space usage for all error rates for the *caching* approach. The *inference* approach uses a lot more space as soon as the error rates go higher. This data includes the votes tables and the support structures.

6.3 Summary

To summarize the experiments, there is no clear winner. Your choice of implementation depends on what you want. *Inference* has problems with completeness to the point where it gives 20% incomplete answers. However, *inference* is better at correctness while *cache* and *always* have better completeness but give you the occasional wrong answer. When it comes to cost in terms of asking the crowd, *caching* is definitely better than *always*.



(a) Completeness: Zipf, Inference, 5M Queries, Vary Error



(b) Completeness: Zipf, Caching, 5M Queries, Vary Error

Figure 12: Completeness vs. Error rate for different quorum

When to use inference Low error rates yield the highest quality results from inference while the cost is not much higher or even lower than the other approaches. For higher error rates, *inference* is conservative in the sense that it still has high correctness but gives fewer complete answers. *Caching* and *always* are more aggressive in giving answers but of course hit a few wrong ones along the way. If your application is error-sensitive, then *inference* is the way to go.

Caching *Caching* is always better than the *always crowd* approach. It produces the same quality of results at lower cost (space aside).

Inherent problems of the *inference* algorithm As mentioned before, the algorithm does not exactly map the *min-max* function and does not work in certain cases. This results in a very high incompleteness. The *inference* algorithm has to be improved as proposed in section 5.4.

7 Conclusion

As a short wrap-up, this chapter reiterates the contributions and observations and also lists the unresolved issues and problems.

7.1 Contributions

Data-structures and algorithms to infer knowledge exist and work. We've presented solutions based on existing work for the order and equality relations in the absence of errors.

Tolerating errors The main contribution is the scoring function to tolerate errors and the implementation approximating the computationally expensive function into a more affordable data-structure. The implementation delivers an approximation of the result of the function but has some problems as it does not map the *min-max* scoring function perfectly to the data-structures..

Benchmarking No benchmark exists to measure the performance of crowd-sourced inference algorithms. This thesis proposes one possible way to do so.

7.2 Observations

Inference is not expensive, as long as there are no errors to tolerate. Tolerating errors in crowd-sourcing is not only computationally expensive, we need to ask more questions to the crowd which directly translates into spending money on human operators. Our choice of implementation is not always the best option depending on the needs of the application. If correctness is key, *inference* performs very good, even at high error rates. But this is achieved at the price of more questions to the crowd and a massive completeness penalty.

Correctness The algorithm that was presented in chapter 5 detects errors in the crowd-sourced data. Lacking more data, the algorithm then returns "unknown" as an answer instead of a wrong answer. It is conservative in the sense that it rather not returns a result than a wrong one.

Completeness As the error rate grows, the answers of the algorithm become less complete in the sense that it gives "unknown" as an answer more often. This can be alleviated by asking the right questions to the crowd (it's not always the direct comparison of two entities).

7.3 Future Topics of Research

Several open issues remain. Our implementation has a flaw that can be fixed by asking the right questions to the crowd.

Scoring function The scoring function min-max is not strongly transitive. It is unknown whether a scoring function exists that is strongly transitive or whether such a scoring function does not exist.

Parallelization The presented implementation was not created with parallelism in mind. A real system will have to deal with queries and input in parallel.

Ordering We did not investigate how to infer ordering knowledge in the presence of errors. This is definitely a topic that is worth investigating, especially whether the *min-max* scoring function can be used here as well.

A Bibliography

[1] J. Howe, "The rise of crowdsourcing," *Wired Magazine*, June 2006.

[2] K. Thomas and D. Nicol, "The koobface botnet and the rise of social malware," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pp. 63 –70, oct. 2010.

[3] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "Turkit: tools for iterative tasks on mechanical turk," in *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, (New York, NY, USA), pp. 29–30, ACM, 2009.

[4] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom, "Deco: Declarative crowdsourcing," technical report, Stanford University, 2011.

[5] A. Marcus, E. Wu, S. Madden, and R. C. Miller, "Crowdsourced databases: Query processing with people," in *CIDR*, pp. 211–214, www.crdrdb.org, 2011.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "CrowdDB: answering queries with crowdsourcing," in *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 61–72, ACM, 2011.

[7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, (New York, NY, USA), pp. 193–204, ACM, 2008.

[8] S. K. Dey and H. Jamil, "A hierarchical approach to reachability query answering in very large graph databases," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, (New York, NY, USA), pp. 1377–1380, ACM, 2010.

[9] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, (New York, NY, USA), pp. 595–608, ACM, 2008.

[10] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, p. 75, Apr. 2006.

[11] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," *SIGMOD Rec.*, vol. 18, pp. 253–262, June 1989.

[12] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, ch. 21: Data structures for Disjoint Sets, pp. 498–524. McGraw-Hill Higher Education, 2nd ed., 2001.

[13] R. Fagin and E. L. Wimmers, "A formula for incorporating weights into scoring rules," *Theoretical Computer Science*, vol. 239, no. 2, pp. 309–338, 2000.