Master's Thesis Nr. 58

Systems Group, Department of Computer Science, ETH Zurich

Crowdsourced Order: Getting Top N Values From the Crowd

by

Adiya Abisheva

Supervised by

Prof. Donald Kossmann
Sukriti Ramesh
Anja Gruenheid

April 2012–October 2012

Abstract

There are several computational tasks which machines are not yet “intelligent” enough to solve and human power is required. One such task is subjective comparisons of entities, e.g. sorting images by “cuteness” or sorting restaurants by quality is hard for computers to solve without human aid. Such tasks could be crowdsourced and their results persisted. Crowdsourcing, however, a) is an expensive activity as each question to the crowd is costly, and b) returns conflicting results since people have various preferences and do mistakes. Thus, the goal is to infer as much knowledge from crowdsourced data as possible, and attempt to resolve conflicts.

This master thesis aims at exploring algorithms for performing crowdsourced order, i.e. produce ranking of objects from human observations, and tolerate contradictions by keeping cost, the number of questions asked to the crowd, low, and quality, the number of correct results, high. The results of experiments give insights on how to balance these measurements.

Contents

1 Introduction **4**

1.1 Motivation 4

1.2 Goals 5

1.3 Problem Statement 5

1.4 Thesis Overview 6

2 Background **7**

2.1 Crowdsourcing 7

2.2 Crowdsourced Databases 8

2.3 Related Work in Voting Theory 9

2.3.1 Single Winner vs. Ranking 10

2.3.2 Voter Rankings vs. Paired Comparisons 10

2.3.3 Voting Schemes 11

2.4 Related Work in Sports Tournaments 17

2.5 Related Work in Graph Theory 17

2.6 Summary 18

3 Decision Function **19**

3.1 Goal 19

3.2 Desired Decision Function 19

3.2.1 Decision 19

3.2.2 Properties 19

3.3 "Min/Max" Decision Function . . . 20
3.3.1 Definitions . . . 20
3.3.2 Deviations From Original Schulze Method . . . 22
3.3.3 Compliance With Desired Decision Function Properties . . . 22

4 Implementation **24**
4.1 Data Structures . . . 24
4.2 Algorithm . . . 25
4.3 Tolerating Contradictions . . . 26
4.3.1 Verify Random Edge Approach . . . 26
4.3.2 "Combine" Approach . . . 27
4.4 Summary . . . 28

5 Experiments **29**
5.1 Methodology . . . 29
5.1.1 Crowd Simulation . . . 29
5.1.2 Baseline Methods . . . 30
5.1.3 Parameters . . . 30
5.1.4 Hardware and Software Configurations . . . 34
5.2 Experimental Results . . . 34
5.2.1 Cost Analysis . . . 34
5.2.2 Correctness Analysis . . . 35
5.2.3 Completeness Analysis . . . 36
5.2.4 Quorum Effect . . . 37
5.2.5 Randomization Effect . . . 39
5.2.6 Error Rate Distribution Effect . . . 39

6 Conclusion **41**
6.1 Summary . . . 41
6.2 Future Work . . . 42

A Appendix 45

A.1 "Min/Max" Decision Function 45

A.1.1 Compliance With Voting System Criteria 46

A.2 Benchmarking Inference Methods 48

A.3 Experimental Results 52

A.3.1 Cost measurements 52

A.3.2 Correctness measurements 54

A.3.3 Completeness measurements 56

Introduction

In this chapter we will discuss what motivates us to do research in the field of crowdsourced databases, what the core of the problem is, as well as we provide the structure of the thesis.

1.1 Motivation

With advance of computer technology theory, many tasks are nowadays solvable by computers; however, some tasks still require human aid since machines are not "intelligent" enough to solve them. Examples of the latter ones are classification problems such as entity resolution as well as social preferences problems, for example, identifying top 10 restaurants in Zürich. Measuring quality of a restaurant is a typical crowdsourcing task. One of the famous platforms to accomplish such tasks is Amazon Mechanical Turk where users also get monetary compensation for answering questions. Tasks are known as Human Intelligent Tasks (abbrev. HITs), users are known as workers, payment for one task varies from $0.01 up to $1.00.

CrowdDB, [1], [2], is a database that asks crowd to answer queries when data or pieces of data for the query is missing in its storage. As any other traditional DBMS, CrowdDB must be able to perform data processing functions of retrieving, updating, aggregating the data, finding the minimum or the maximum, grouping, sorting etc. CrowdDB uses CrowdSQL as DML/DDL which is an extension of SQL language with the main difference that operators act not on "clean" data but rather on subjective information of HIT workers. For example, ORDER BY clause in CrowdDB will use various and in most of the cases conflicting preferences of people, and eventually must produce the order of the requested objects.

There are two main obstacles in solving the social ranking problem. By nature of the tasks, comparisons are subjective, i.e. people heavily rely on their tastes; additionally, people do mistakes introducing conflicts in crowdsourced results. For the first problem belief is in wisdom of crowds where collective opinion is as good as or in some cases even better than expert's opinion. The second problem is harder to tackle as noise in data makes it hard to exploit traditional *transitivity* property used in ordering problems. One possible way to resolve conflicts is to ask people more questions to build confidence, but every additional question costs money for crowdsourcer, initiator of the HIT. Thus, idea is to be able to rank objects in a cost-effective way by smartly *inferring* unknown information through existing crowd preferences.

This problem stems from various fields and theories – mathematics, computer science, sociology, voting theory, pairwise comparisons theory, graph theory and other sciences. The art is to be able to either integrate existing solutions for similar problem or based on related work come up with a new algorithm that solves the problem.

1.2 Goals

This master thesis aims at devising an algorithm that given the crowdsourced data of pairwise comparisons of objects with conflicts will output either the ranked objects or top N entities in a cost-effective manner and with high response time.

One of the primary goals is to create and implement **baseline** algorithmic solution, run experiments on it, report and analyze findings. As a starting point, it is helpful to look at related work done for entity resolution and adjust it for the problem of top N values, e.g. by creating new data structures, by testing various decision functions. Since experimentation gives more insights into behaviour of algorithm, next goal will be to improve the existing solution.

Following statements are extra goals that might not fit into the time limits of the master thesis, but are important to be addressed. To build confidence in correctness of the solution, it will be necessary to measure quality of produced rank, i.e. how close the resulting ranking is to the true ranking of the objects. Another challenging goal is to tackle the next votes problem: what the best questions to ask the crowd are, when information about the pair of objects is not available. Experiments for entity resolution have shown that directly asking the crowd about objects' relation is not the best solution.

1.3 Problem Statement

Followed by the motivation section, we will try to formalize the problem we are facing.

Given data

Paired-comparison data, referred to as *crowd votes*;

Conflicts in data since people do mistakes, referred to as *error rate*, and people do not always agree on each other preferences;

Varied amount of crowd votes:

- Initially, data is sparse, i.e. number of votes $\ll$ number of items, and we need to infer knowledge from sparse data,
- Later, number of votes $\geqslant$ number of items, we need to try to deal with contradictions.

Output

Ranking of items, or

Identify *Top N* items.

1.4 Thesis Overview

Background chapter, 2, gives overview of existing methods in ranking from human observations in various fields including voting theory, sport tournaments, psychology and graph theory. Also we will briefly introduce concepts of crowdsourced databases.
Decision Function chapter, 3, discusses properties of decision function needed for crowdsourced ordering relation, and describes decision function chosen for our solution, its pluses and drawbacks.
Implementation chapter, 4, explains algorithm and data structures of the chosen solution to perform crowdsourced ordering.
Experiments chapter, 5, shows results of experiments of the chosen solution with methodology and parameters of evaluation, and analysis of the results.
Conclusion chapter, 6, discusses contribution done in the field of algorithms in the crowdsourced databases and future work.

Background

In this chapter we describe related work done in the field of ranking from human observations. The problem stems from various fields such as voting theory, psychology, mathematics and computer science. Many attempts have been done to produce "perfect" ranking; these attempts can be categorized as probabilistic, graph-theoretical and combinatorial approaches. We will first start by describing concepts of crowdsourcing and, in particular, crowdsourced databases, and then proceed by explaining related work done in ranking.

2.1 Crowdsourcing

Crowdsourcing is a relatively new term that first appeared in 2006 in the "Wired" magazine article "The Rise of Crowdsourcing", [3]. Composed of two words, word "crowd" points to people participating in some activity and "sourcing" makes reference to some procurement practices. In brief, crowdsourcing is a business practice of outsourcing labor-intensive activities over the Internet by splitting them into small pieces and distributing to the hundreds of people that accomplish the tasks at their desks. Since its first introduction there has been 40 known definitions of the word "crowdsourcing" according to [4]. The most recent and probably the most all-embracing definition of the term is:

> *Crowdsourcing is a type of participative online activity in which an individual,..., or company proposes to a group of individuals of varying knowledge, heterogeneity, and number, ..., the voluntary undertaking of a task. The undertaking of a task, of a variable complexity and modularity, ..., always entails mutual benefit. The user will receive the satisfaction ... be it economic, social recognition, self-esteem, or the development of individual skills, while the crowdsourcer will obtain and utilize to their advantage that what the user has brought to the venture...*

The keywords here are "proposing activity" and "voluntary undertaking of a task". This way it is different from a known "outsourcing", since the latter one is about hiring people to perform service, similar to traditional employment concept. Crowdsourcing, on the other hand, brings in people from outside by proposing the tasks and involves them in a creative and collaborative process, [3].

Today plenty of platforms exist to perform so-called *microtasks*: Amazon Mechanical Turk, Clickworker, ShortTask and many more. These Internet marketplaces provide services for both – those who want to earn money, turkers or solvers, and those who need some online

job to be done, seekers. Seekers are able to post the jobs, and turkers provide solutions to the chosen tasks. The nature of the tasks is such in which people are outperforming computers ranging from technical to creative tasks, [5]. Examples of technical jobs are photo tagging and classification (Samasource), video and audio transcribing (CastingWords), translation (MyGenGo), retrieving business information such as URLs or contact details (CrowdFlower) and many more. Examples of creative tasks that could be crowdsourced are logo designing (crowdSpring), finding right brand name for a company or product (NamingForce) and many others.

"Crowdsourcing" can be seen as an umbrella term for different categories. Most known crowdsourcing activities which also have commercial applications are: "wisdom of crowds", "crowdfunding" and "crowdvoting".

Wisdom of crowds is a concept of sourcing knowledge or information from the crowd to gain a complete and accurate picture of the topic with the assumption that group decision is better than each individual's answer. Successful examples of this concept are Wikipedia and Yahoo!Answers.

Crowdfunding is a type of fundraising activity where crowd funds the project by contributing small amount in order to obtain project's monetary goal. Kickstarter is a well-known crowdfunding platform which allows people to pledge money for their projects.

Crowdvoting as the name suggests allows crowd to vote on a certain subject, and thus collects truly people's opinion on the matter. Some crowdvoting websites are only there for entertainment purposes, e.g. The Cutest or The Funniest; other websites, like Ranker, create consumers' "taste graph" which has more business applications. One common thing about them is utilization of some sort of algorithm for voting on the subject compared by humans. Some of those websites give out the ranking mechanism they implement, others keep it as a secret.

2.2 Crowdsourced Databases

Crowdsourcing can also be applied in the database domain; it will deal with two present features of traditional relational database systems, [1]:

1. closed-world assumption (abbreviated CWA);
2. non-tolerance towards typographic or any kind of data errors.

CWA states that if database does not contain certain information, missing information is considered to be false or non-existent. Databases with crowdsourcing capabilities make it possible for people to fill in the gaps.

The second feature of relational databases being "literal" demonstrates that given the right context people are better at solving several data comparison tasks than computers. A simple query, [1], such as:

```
SELECT * FROM company WHERE name = "I.B.M.";
```

might return empty results even when the query is correct and data for this query exists in the database. For instance, company name in the database might be spelled in full – "International Business Machine" rather than abbreviated as "I.B.M.". For people given the right context it is not difficult to identify whether two entities are the same, for computers such entity resolution task is impossible with no previous knowledge. Another simple query from entertainment website The Cutest:

```
SELECT image FROM images ORDER BY cuteness;
```

If "cuteness" score has already been previously defined and assigned to each image, the database might apply an ordering operation and return images sorted by queried field. Otherwise, subjective term "cuteness" is hard for computers to determine. One solution is to crowdsource such subjective comparison operator.

To sum up, in the database domain there are four type of operations that might be crowdsourced:

- Gathering (filling missing data, open-world assumption)
- Comparing (e.g. entity resolution)
- Ordering (e.g. ranking from human observations)
- Matching

Crowdsourced databases is a research work in progress; however, there are already implementations such as CrowdDB [1], Deco [6] and Qurk [7]. According to [8], a lot of research has been done in the field of quality control in the crowdsourced databases, e.g. assessing turkers skills and diligence; but now the focus is shifted towards devising algorithms for implementing crowdsourced operators in databases with crowdsourcing capabilities. Steps have been done in resolving entity resolution problem, i.e. to answer questions whether two objects represent the same entity [9], as well as identifying the highest-ranked object [8], not much research is known in the field of ranking. Thus, the focus of this master thesis is on implementing algorithm for ordering crowdsourced data.

2.3 Related Work in Voting Theory

This section illustrates an outstanding variety of alternatives for counting and ranking individual preferences used in elections. It is largely based on the paper "An Introduction to Vote-Counting Schemes" by Johnathan Levin and Barry Nalebuff, [10].

First, we should distinguish between the motivation of vote-counting mechanisms and the basic information on which voting schemes rely. Some methods perform better at choosing a single winner, others suit best for ranking all candidates. Some election schemes are based on voter rankings, others take as input paired comparisons data.

2.3.1 Single Winner vs. Ranking

Formally, we consider a set of alternatives (candidates) A and a set of n voters I. We denote by L the set of linear orders on A; thus, for every $\prec \in L$, $\prec$ is a total order on A. The preferences of each voter i are given by $\succ_i \in L$, where $a \succ_i b$ means that i prefers a to candidate b. Thus, we formalize two functions according to [11]:

A function $F\colon L^n \to L$ is called a *social welfare function*.

A function $f\colon L^n \to A$ is called a *social choice function*.

In other words, a social welfare function results in a ranking of candidates, i.e. a total social order on the candidates, while a social choice function yields a social choice of a single candidate.

Since we focus on ranking, we are interested in designing a social welfare function, additionally, this function should satisfy certain "good" properties described later, see Chapter 3.

2.3.2 Voter Rankings vs. Paired Comparisons

In a multi-candidate election we have two options for voters to compare the candidates, each has its own advantages and disadvantages. Thus, most of the elections schemes take as input either *voter rankings* data or *paired comparisons* data. As convention, for our solution we have chosen as an input method the latter one, since it is always possible to convert the voter rankings into paired comparisons data called *paired-comparisons matrix*, but not the other way round.

In *voter rankings* method, each voter is given a ballot with the list of all candidates where voter must assign a "rank" to each candidate in terms of preference. We probably get more information than needed to get the aggregate ranking, nevertheless this information reveals how many times candidate i has been placed jth. On the other hand, ranking all candidates creates some burden for a voter; since voters are usually not allowed to produce ties, it might be easy for a voter to select a winner on the ballot, but differentiating between other candidates might not be an easy task and scoring assignment to the remaining candidates can be done carelessly. This results in noisy input data and creates the source of ambiguity for the ranking scheme applied later.

In *paired comparisons* method, voter picks the winner among only two candidates. It is a simple yet quite a powerful way of collecting voting data and gives rise to a very basic vote-counting scheme such as making a winner this candidate who wins in every head-to-head competition, known as *Condorcet winner*. Since it is impractical to ask people to compare all candidate pairs, e.g. having only 6 candidates, a voter must make 15 comparisons, all voting schemes utilize the transitivity property of the preference relation. For example, if a is preferable to b and b is preferable to c, we infer that a is preferable to c and there is no need to ask voters to compare candidates a and c. Despite it's simplicity, paired-comparisons data is always prone to having voting cycles, e.g. a beats b, b beats c and c beats a.

Presence of conflicts in voting system with unrestricted collection of people's preferences has been proven by Arrow in 1951 in his impossibility theorem which says:

> *Every social welfare function over a set of more than 2 candidates ($|A| \geqslant 3$) that satisfies unanimity and independence of irrelevant alternatives is a dictatorship.*

For our problem it translates that no voting system that satisfies "fairness" criteria, such as unrestricted domain, non-dictatorship, Pareto efficiency and independence of irrelevant alternatives, can convert ranked preferences to the complete and transitive ranking.

2.3.3 Voting Schemes

Implication of Arrow's impossibility theorem is such that so far there is no existing voting scheme that can guarantee the total democratic order, i.e. we must at least compromise on one of the "fairness" criterion. Said that, we will look at existing voting mechanisms and try to find the "best" among them.

Condorcet Winner

A Condorcet winner is the candidate who wins in *every* comparison with all other candidates. For example, in tables 2.1, 2.2 and 2.3, we see the rankings of three voters for four candidates {A,B,C,D}.

Place	Candidate
1	B
2	C
3	A
4	D

Table 2.1: Voter 1

Place	Candidate
1	D
2	A
3	C
4	B

Table 2.2: Voter 2

Place	Candidate
1	A
2	C
3	B
4	D

Table 2.3: Voter 3

Converting voters' rankings into the paired-comparisons matrices, we get Tables 2.4, 2.5 and 2.6.

	A	B	C	D
A	×	0	0	1
B	1	×	1	1
C	1	0	×	1
D	0	0	0	×

Table 2.4: Voter 1 ranking as paired-comparisons matrix

	A	B	C	D
A	×	1	1	0
B	0	×	0	0
C	0	1	×	0
D	1	1	1	×

Table 2.5: Voter 2 ranking as paired-comparisons matrix

	A	B	C	D
A	×	1	1	1
B	0	×	0	1
C	0	1	×	1
D	0	0	0	×

Table 2.6: Voter 3 ranking as paired-comparisons matrix

Finally, summing up the matrices above we get paired-comparisons matrix 2.7.

	A	B	C	D
A	×	2	2	2
B	1	×	1	2
C	1	2	×	2
D	1	1	1	×

Table 2.7: Paired-comparisons matrix

Now, we look at the number of votes for runner (row) over opponent (column) and compare it with the number of votes for opponent over runner. Candidate that wins in every paired comparison is declared to be the Condorcet winner, in the above example, candidate A is the one.

Plurality Voting

Voters select their most preferred candidate, candidate with the most votes wins the elections. It is one of the simplest voting methods which is especially suited for elections with two serious candidates, e.g. US presidential elections. If there are more than two candidates, voters rank each candidate on their ballots, and candidate with the most first-place votes wins. In case of ties the method looks at the second-place number of votes to determine the winner. It is different from the *majority rule* since in the latter the winner must obtain more than half of the votes, i.e. more than 50% of voters must choose candidate that is to be declared the winner. Majority vote is an instance of plurality voting with only two candidates.

Approval Voting

It is one of the only methods where candidates are not ranked but rather approved or disapproved. Voters can select more than one candidate that they approve; however, selecting all candidates counts as no voting. Candidate with the most approved votes wins the elections. This method is currently used in the elections of the United Nations Secretary General, see example in Table 2.8:

Candidate	Approve	Disapprove	No opinion
Ban	14	0	1
Ghani	4	11	0
Sathirathai	4	7	4
Tharoor	10	3	2
Vike-Freiberg	5	6	2
Zeid	2	8	5

Table 2.8: United Nations Secretary General Elections in 2006, [12]

Runoff Voting

It is also known as "double election" or two-round system voting. In the first round candidates are ranked, if at this stage one candidate gets majority vote, he is declared to be the winner. Otherwise, the runoff election takes place between the two winning candidates of the first round. Advantages of such voting is correcting possibly skewed results of plurality voting. For example,

> *Suppose, we have three candidates with two similar candidates that will possibly evenly divide the 60% of votes, and one radical candidate with 40% of minority support. Under plurality voting the latter "weak" candidate will win, however, under runoff voting he will be beaten in the second round head-to-head competition by more popular candidate.*

Single Transferable Vote

Single-winner When choosing a single winner in elections, Single Transferable Vote (abbreviated STV, also known as "alternative vote") acts as an extension of runoff voting with candidates being eliminated one by one at each round. Voters rank candidates on ballots in the order of preference. If any of the candidates obtains majority vote, he is declared to be the winner. Otherwise, candidate with the least amount of first-place votes is removed from the candidates list, votes for the losing candidate are redistributed among remaining candidates, and all ballots are retabulated. The procedure of removing candidate gaining the fewest first-place votes repeats until the single winner is found.

Multiple winners In multi-winner elections STV tries to achieve the proportional representation of candidates from multiple parties. If in single-winner STV the potential winner must gain majority vote, then in multi-winner STV with w winners for a candidate to be elected, he needs to surpass some votes quota, q. Formulas to set q are *Droop quota* calculated as $\lfloor \frac{\text{number of voters}}{w+1} \rfloor + 1$, or *Hare quota* calculated as $\frac{\text{number of voters}}{w}$, or *Imperiali quota* $= \frac{\text{number of voters}}{w+2}$. Election runs in similar way as single-winner STV until selecting the first winner. If candidate exceeds the vote quota, his surplus, s, where $s = \textit{total votes received} - q$, is divided and added to voter's second place choice. This results in two scenarios: either redistribution of surplus creates another winner, the one reaching the vote quota, or no candidates will reach the quota. In the former case, we repeat the same procedure, i.e. eliminate the new winner and redistribute his surplus; in the latter case, we eliminate the losing candidate in the same fashion as electing STV single-winner, i.e. choose candidate with the fewest first-place votes and redistribute his votes among remaining candidates. The procedure stops when reaching w amount of winners.

Coombs Voting

It is a variation of STV for single winner election with the main difference that in each successive round we do not eliminate candidate with the fewest first-place votes but rather we remove candidate with the most last-place votes. This type of voting is also known as "exhaustive voting" since electing the winner with k candidates needs $k-1$ rounds,

unlike in STV voting where in case of majority vote in the first round the winner will be already found.

Borda Voting

It is a single-winner election method after which the serious study in the voting theory has begun. The method has many variations; however, in the core of the voting each candidate is assigned some score based on the amount of votes he received. In its simplest form Borda voting has the following procedure: voters rank k candidates in the order of preference on ballots, then candidates placed 1st receive $k-1$ points, candidates placed 2nd get $k-2$ points, and so forth. Candidates' points are aggregated, and finally they are ranked based on these scores. Converting voters' rankings into paired-comparisons matrix, we notice that Borda scores are sums of head-to-head wins each candidate obtained, i.e. we can sum columns of the paired-comparisons matrix to obtain the Borda score of each candidate. For example, tables below show rankings of nine voters:

Voters 1,2	
1	A
2	B
3	C

Voter 3	
1	A
2	C
3	B

Voters 4,5	
1	B
2	A
3	C

Voter 6	
1	B
2	C
3	A

Voters 7,8	
1	C
2	A
3	B

Voter 9	
1	C
2	B
3	A

Converting the rankings into paired comparison matrix, we get:

	A	B	C
A	×	5	5
B	4	×	5
C	4	4	×

Table 2.9: Paired-comparisons matrix

A	10
B	9
C	8

Table 2.10: Borda score

Copeland Voting

In this voting method we convert paired-comparisons matrix into win-loss matrix, i.e. for each head-to-head vote we assign 1 for winner in the pair, -1 for loser and 0 for tie. Similar to Borda voting we sum columns to get the Copeland score of each candidate. Taking the same example as in Borda voting, we will get the following scores:

	A	B	C
A	×	1	1
B	-1	×	1
C	-1	1	×

Table 2.11: Win-loss matrix

A	2
B	0
C	-2

Table 2.12: Copeland score

Minimum Violations

Violation is a situation when in the final ranking candidate c_i is ranked higher than candidate c_j, however in the head-to-head competition c_i is beaten by c_j. Minimum violations ranking method aims at coming up with such a permutation of candidates where number of such violations is minimum. After voters rank candidates on their ballots, votes are tabulated into paired-comparisons matrix to determine the winner in every head-to-head competition. Then we consider each permutation of candidates, e.g. $\{c_i, c_j, c_k, ...\}$, or $\{c_j, c_i, c_k, ...\}$, or $\{c_k, c_i, c_j, ...\}$, as a potential ranking, and choose the one that has the minimum number of violations. One of the ways to minimize the number of violations is to look at the win-loss matrix, e.g. recall Copeland voting. The idea is to permute rows and columns of win-loss matrix in order to maximize sum of entries above the diagonal. For example,

	a	b	c	d
a	×	51	45	58
b	49	×	53	44
c	55	47	×	49
d	42	56	51	×

Table 2.13: Paired-comparisons matrix

$\Longrightarrow$

	a	b	c	d
a	×	1	0	1
b	0	×	1	0
c	1	0	×	0
d	0	1	1	×

Table 2.14: Win-loss matrix

$\Longrightarrow$

	a	d	b	c
a	×	1	1	0
d	0	×	1	1
b	0	0	×	1
c	1	0	0	×

Table 2.15: Win-loss matrix after permutation

We obtain the following ranking in this example: $a > d > b > c$, the sum of the entries above the diagonal in the win-loss matrix after permutation is minimal. It is one of the most computationally expensive methods, since having c candidates requires evaluating $c!$ possible permutations. Another problem is non-resistance to voting cycles, i.e. the method does not know how to resolve contradictions; with the presence of voting cycles there is no zero violations ranking, and thus final ranking is not necessarily unique.

Tideman's Ranked Pairs

The idea of the method has roots in Condorcet's paper, 1785. Recall, Condoret winner is the one who wins in every head-to-head competition; however, if such candidate does not exist, in his paper Condorcet suggests that "large majorities should take precedence over small majorities in breaking cycles". In other words, in paired-comparisons matrix we find the largest margin of victory, fix the result of this pair for the final ranking and "eliminate all potential rankings that contradict this preference". If there exists the largest victory for candidate a over candidate b, later on we will not consider rankings where b beats a. One problem with such approach is that with the number of candidates greater than three, when only choosing the largest majorities, we might end up locked in a cycle. N.Tideman in 1987 suggested a fix for this problem by ignoring the head-to-head votes that will lock in a cycle. For example,

	a	b	c	d
a	×	**61**	41	**51**
b	39	×	**60**	**51**
c	**59**	40	×	**51**
d	49	49	49	×

Table 2.16: Paired-comparisons matrix

The largest margin of victory is between a and b, thus we lock a > b. Following the same logic we next lock b > c, and transitively deduce that a > c. Next pair to lock is c > a, but this results in a cycle, so according to Tideman's approach we ignore this head-to-head result. We then fix a > d, b > d and c > d, and end up with the following ranking: a > b > c > d.

Simpson-Kramer Min-Max Rule

It is another voting technique where candidates are assigned scores. Each candidate gets *max* score which is the largest number of votes against that candidate in a paired-comparisons matrix. To select a single winner we choose candidate with the minimum *max* score, or candidates are ranked in a decreasing order of their max scores. This method is considered to work well for electing single winner, though it is not suitable as a ranking technique. We will get the following min-max scores for the ranked pair example:

a	b	c	d
59	61	60	51

Table 2.17: Min-Max scores

The resulting ranking is: $d > a > c > b$

Kemeny-Young Method

This method combines idea of ranking via permutation and importance of large majorities. It is similar to minimum violations ranking, with the difference that we permute rows and columns of the paired-comparisons matrix rather than of the win-loss matrix. It also adheres to the Condorcet principle where large majorities take precedence over small majorities. Taking the same example as in minimum violations we will obtain the following ranking:

	a	b	c	d
a	×	51	45	58
b	49	×	53	44
c	55	47	×	49
d	42	56	51	×

Table 2.18: Paired-comparisons matrix

$\Longrightarrow$

	c	a	d	b
c	×	55	49	47
a	45	×	58	51
d	51	42	×	56
b	53	49	44	×

Table 2.19: Paired-comparisons matrix after permutation

Kemeny-Young method produced the ranking $c > a > d > b$ and made c the winner, while in minimum violations voting c is the loser $a > d > b > c$. The main difference is in relation between c and a. In Kemeny-Young voting, c must beat a since $c > a$

with 55 votes, and is the third largest majority after $a > d$ with 58 votes and $d > b$ with 56 votes. And since Kemeny-Young respects large majorities, $c > a$ is locked in the ranking. Furthermore, even though head-to-head c has been beaten by b and d, it still is placed ahead of them but not directly and thus does not create a violation according to it definition, i.e. "a candidate cannot be ranked **directly** above another candidate who defeated her head-to-head".

2.4 Related Work in Sports Tournaments

In sports, our problem is closely related to various competition schemes, e.g. round-robin tournament, swiss tournament and so forth. Two methods are worth looking at:

Kendall-Wei/Power Rank Method

Kendall-Wei is a ranking method via eigen-decomposition of win-loss matrix, and Power Rank is a ranking scheme via eigendecomposition of paired-comparisons matrix. Idea is similar to Copeland method, but extends it in the way that score of a candidate takes into account the strength of those candidates that he has beaten. In the first round each candidate receives the score equal to the number of pairwise wins, similar to Copeland method. In the second round each candidate receives a second score equal to the number of wins gained by the candidates she took over. In the third round each candidate's score is the sum of the two previous round scores. This process continues: next round score of a candidate is the sum of the two previous round scores. This way the sequence of scores might go to infinity, but it is proven to be convergent, and the convergent limits are called Kendall-Wei scores. One of the ways to find the limits of these sequences is through eigen-decomposition of the win-loss matrix: if W is the win-loss matrix, l is the largest positive eigenvector, we can find vector v such that $Wv = lv$, and v is the Kendall-Wei scores of candidates. Interesting feature of this method is it only works with the presence of voting cycle.

Jech Method

It is a probabilistic ranking method that not only produces ordering of alternatives, but also tells "how much different" the alternatives are from each other. This method yields probability matrix p, where probability of i winning over j in a pairwise game is $p_{ij} = \frac{T_i}{T_i+T_j}$, where T is candidate's strength. Notable feature of this method is it works even if not all teams played one another.

2.5 Related Work in Graph Theory

Finally, the last field with existing problem related to our problem is in graph theory. *Feedback arc set* is the minimum set of edges in graph that when removed makes the graph

acyclic. This is directly applicable to our problem, since if we represent our votes for alternatives as a graph, we would observe cycles due to people having various preferences or due to them making mistakes, and eliminating the cycles would make the graph free of conflicts. The key point here is finding minimum set of arcs producing the cycles which translates into our problem as removing the minimum amount of violations, or conflicts in people's votes. Unfortunately, the problem is known to be NP-Hard, and the best known approximation ratio is $\mathcal{O}(\log n \log\log n)$.

2.6 Summary

Having looked at existing background work done in the field of ranking from human observations, we concluded that voting theory offers plenty of interesting approaches. Most of the methods described here are based on scoring each individual alternative. However, we discovered a recent path-based voting scheme that satisfies most of the voting criteria and also produces preference relation between entities. This method has been proposed by M.Schulze, [13], and is known as *Schulze method* or the *Beatpath method*. It will be explained in Chapter 3.

Decision Function

In this chapter we discuss properties of decision function needed for crowdsourced ordering relation, and describe decision function chosen for our solution, its pluses and drawbacks.

3.1 Goal

Our main goal given the problem statement in Introduction 1 is to produce ranking of entities. Any ranking procedure is usually requires two steps: a) find a preference relation between objects, b) produce actual ranking of objects based on a). Having defined preference relation between entities, we ensure that whenever crowdsourced database is asked to compare two objects, it can return some meaningful answer. Thus, we will first focus on discovering decision function that takes as input two objects and as output gives an answer which of the two objects is preferred by the crowd.

3.2 Desired Decision Function

3.2.1 Decision

Intuitively, decision function will answer the following preference question: given crowd votes and two objects for comparison, which object is more preferred by the crowd.

Formally, a decision function f takes as input two nodes, n_1 and n_2, and makes the following decision on whether n_1 is preferred over n_2:

$$f(n_1, n_2) = \begin{cases} \text{Yes,} & n_1 \text{ is preferred over } n_2 \\ \text{No,} & n_1 \text{ is not preferred over } n_2 \end{cases}$$

3.2.2 Properties

Since we are interested in objects' preference, our focus is on the *order* binary relation. We can distinguish various order relations depending on their properties, e.g. total preorder, partial order, strict order and many other. For our application we require the order relation to be: *asymmetric*, *transitive* and *total*. Listed properties form *strict total order*.

Having identified the suitable order relation, we sum up all mathematical properties that desired decision function must obey:

1. *Consistency*
   The decision function should not contradict the votes.

2. *Convergence*
   For decision function to answer {Yes, No} for any pair of nodes there must exist a sequence of votes, successive "Yes" or "No" votes.

3. *Asymmetricity*
   $\forall n_1, n_2 \in$ nodes,
   $f(n_1, n_2) = \text{Yes} \implies \neg(f(n_1, n_2) = \text{No})$

4. *Transitivity*
   $\forall n_1, n_2, n_3 \in$ nodes,
   $f(n_1, n_2) = \text{Yes} \wedge f(n_2, n_3) = \text{Yes} \implies f(n_1, n_3) = \text{Yes}$

5. *Totality*
   $\forall n_1, n_2 \in$ nodes,
   $f(n_1, n_2) = \text{Yes} \vee f(n_1, n_2) = \text{No}$

3.3 "Min/Max" Decision Function

Revisiting the goal, we aim at finding the function that meets the properties listed earlier. Following conclusions drawn in the Background chapter, 2, we focused on finding a method from voting theory to solve our problem. One of the most recent election schemes that has been proposed is the *Schulze method*, or the *Beatpath method* developed by M. Schulze [13]. In the core of the method path-based "Min/Max" decision function is used. Since this method was compliant with the most of the needed "fairness" voting criteria, see Appendix A.1.1, we wanted to discover whether this method also satisfies the desired properties of decision function needed as part of our problem. It takes as input voter rankings, and outputs a) a strict partial order $\mathcal{O}$ on the set of alternatives and b) the set of winners.

path-based approach proposed scoring (decision) function is based on discovering paths in votes graph and making a decision based on the strength of the path. Other approaches involved scoring each individual entity, or assigning winning probability to alternatives.

3.3.1 Definitions

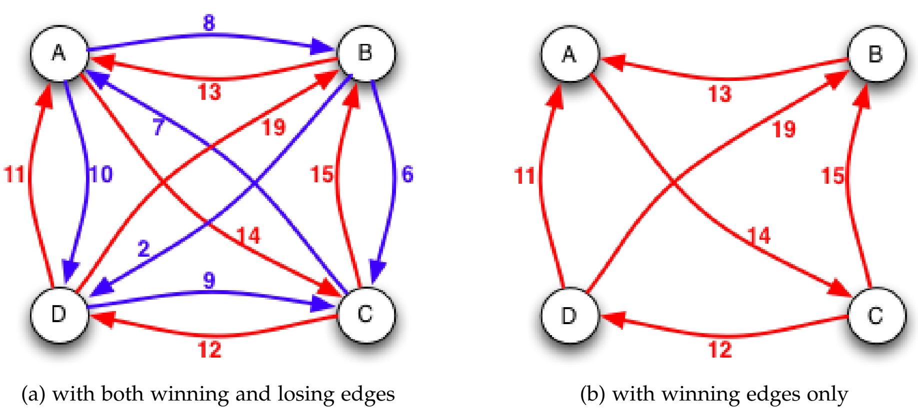
votes graph is a directed weighted graph with nodes representing objects and edges representing weighted preference of one object over the other object (or simply "vote"), for example: $a \xrightarrow{x} b$ stands for a has been preferred (voted for) x times over b. In our notation arrow head points to alternative that is beaten. See Figure 3.1 for example of the votes graph.

path is a sequence of nodes with start node and target node such that from each of the nodes there is a direct edge to the next node in the sequence. There can be multiple paths between the same nodes. Formally, having A alternatives, a path from alternative $a \in A$ to alternative $b \in A$ is a sequence of alternatives $c(1), \ldots, c(n) \in A$ where $a \equiv c(1), b \equiv c(n), 2 \leqslant n, \forall i = 1, \ldots, (n-1) : c(i) \not\equiv c(i+1)$.

path strength is the strength of its weakest link, or its the edge weight in the path with the least amount of votes. Formally, strength of the path $c(1), \ldots, c(n)$ is $min\{(N[c(i), c(i+1)], N[c(i+1), c(i)]) | i = 1, \ldots, (n-1)\}$, where $N[c(i), c(i+1)]$ stands for a weight of an edge between nodes $c(i)$ and $c(i+1)$.

strength of the strongest path since there could be more than one path existing between the two nodes, idea is to choose the strongest path being it a path with the strongest strength. The strongest path is the one with strongest weakest link. Formally, $P[a, b] = max\{min\{(N[c(i), c(i+1)], N[c(i+1), c(i)]) | i = 1, \ldots, (n-1)\} | c(1), \ldots, c(n)$ is a path from alternative a to alternative $b\}$.

direct edges if two entities have been voted for directly, according to [13], edge with less votes between two alternatives is dropped and is not considered in the path strength discovery, see Figure 3.1a vs. 3.1b.



(a) with both winning and losing edges

(b) with winning edges only

Figure 3.1: Votes graph

When calculating strengths of the strongest paths we consider votes graph with winning edges only, 3.1b.

$$P[D, A] = max\{min\{D \xrightarrow{11} A\}, min\{D \xrightarrow{19} B, B \xrightarrow{13} A\}\} = max\{11, 13\} = 13$$
$$P[A, D] = max\{min\{A \xrightarrow{14} C, C \xrightarrow{12} D\}\} = max\{12\} = 12$$
$$P[D, B] = max\{min\{D \xrightarrow{19} B\}, min\{D \xrightarrow{11} A, A \xrightarrow{14} C, C \xrightarrow{15} B\}\} = max\{19, 11\} = 19$$
$$P[B, D] = max\{min\{B \xrightarrow{13} A, A \xrightarrow{14} C, C \xrightarrow{12} D\}\} = max\{12\} = 12$$

According to original paper [13], binary relation $\mathcal{O}$ resulting from applying the method onto the votes graph is:

$$ab \in \mathcal{O} \Leftrightarrow P[a,b] > P[b,a], \text{ where } > \text{ is a } \textit{strict partial order},$$
$$S = \{a \in A | \forall b \in A \backslash \{a\} : ba \notin \mathcal{O}\} \text{ is the set of winners.}$$

A winner $x \in S$ if and only if $yx \notin \mathcal{O}, \forall y \in A\backslash\{x\}$. In the example 3.1b $S = \{D\}$. Also, it is possible to get relation between any pair of entities by looking at the members of the binary relation $\mathcal{O}$. Again, in the example 3.1b we get $\mathcal{O} = \{AB, AC, CB, DA, DB, DC\}$.

3.3.2 Deviations From Original Schulze Method

quorum we introduced the notion of quorum in order to make a final decision on whether one node is preferred over the other one. If the difference in the strengths of the strongest paths between two nodes does not reach the quorum, no preference between objects can be established, and relation between entities will be marked as "Do-not-know", or "UNKNOWN". We expect effects of the quorum will be more pronounced, when setting the quorum to values greater than 1, i.e. $q = 3, 5 \ldots$, i.e. we will get more "Do-not-know" answers, which never happens in the original Schulze method since the difference between path strengths is not considered.

Taking into account quorum, revised "Min/Max" decision function f takes as input two nodes, n_1 and n_2, of the *votes* graph and makes the following decision on whether n_1 is preferred over n_2:

$$f(n_1, n_2) = \begin{cases} \text{Yes, } n_1 \text{ is preferred over } n_2, & P[n_1,n_2] - P[n_2,n_1] \geqslant q \\ \text{No, } n_1 \text{ is not preferred over } n_2, & P[n_2,n_1] - P[n_1,n_2] \geqslant q \\ \text{Do-not-know, neither } n_1 \text{ nor } n_2 \text{ are preferred,} & \text{otherwise} \end{cases}$$

direct edges revisited we at first were not happy with the idea of dropping weak edge between nodes that have direct votes. We wanted to preserve as much information in the votes graph as possible, and tried "Min/Max" function with both winning and losing edges preserved. This way we never break the cycles on the level of an edge, for example having votes for $a \xrightarrow{5} b$ and $b \xrightarrow{3} a$, this already introduces cycle between two nodes; however, we keep history of each individual edge vote. Unfortunately, this inference method that we named *NoDrop* experimentally, see Chapter 5, showed bad performance in relation to measurements that we are interested, e.g. number of incorrect results was much higher compared to results produced by the original method. Also, with the help of small examples we could understand how local erroneous results propagate to the whole graph, see Example A.1 in Appendix.

3.3.3 Compliance With Desired Decision Function Properties

Having these differences with original Schulze method, our revised decision function introduces the new properties that are not necessarily desired:

- *Transitivity With Unknown*
  $\forall n_1, n_2, n_3 \in$ nodes,
  $f(n_1, n_2) = \text{Yes} \wedge f(n_2, n_3) = \text{Do-not-know} \implies f(n_1, n_3) = \text{Yes} \vee f(n_1, n_3) = \text{Do-not-know}$

  $\forall n_1, n_2, n_3 \in$ nodes,
  $f(n_1, n_2) = \text{Do-not-know} \wedge f(n_2, n_3) = \text{Yes} \implies f(n_1, n_3) = \text{Yes} \vee f(n_1, n_3) = \text{Do-not-know}$

- *Partial Results*
  $\forall n_1, n_2 \in$ nodes,
  $f(n_1, n_2) = \text{Yes} \vee f(n_1, n_2) = \text{No} \vee f(n_1, n_2) = \text{Do-not-know}$

With introduction of quorum we might get boost in terms of "Do-not-know" results. We also cannot guarantee the totality property anymore since binary relation $\mathcal{O}$ is partial.

Implementation

In this chapter we explain algorithm and data structures of the chosen solution to perform crowdsourced ordering.

4.1 Data Structures

We first aimed at implementing the "Min/Max" decision function for crowdsourced ordering using a straightforward approach suggested in the paper, [13]. For this we needed the following data structures to store various types of data.

nodes is a table of all alternatives that were voted for earlier. For votes graph in Example 3.1a, the nodes table is shown in Table 4.1:

node
A
B
C
D

Table 4.1: *nodes* table of alternatives

votes is a table of crowd votes for alternatives. For votes graph in Example 3.1a, the votes table is shown in Table 4.2:

path is a matrix, $N \times N$, of strengths of the strongest paths between alternatives, where N is the total number of alternatives. path[node_a,node_b] stands for the strength of the strongest path from alternative node_a to alternative node_b. For votes graph in Example 3.1b, the votes table is shown in Table 4.3:

node_a	node_b	votes(a>b)	votes(b>a)
A	B	8	13
A	C	14	7
A	D	10	11
B	C	6	15
B	D	2	19
C	D	12	9

Table 4.2: *votes* table of crowd votes

	path[*,A]	path[*,B]	path[*,C]	path[*,D]
path[A,*]	×	14	14	12
path[B,*]	13	×	13	12
path[C,*]	13	15	×	12
path[D,*]	13	19	13	×

Table 4.3: *path* matrix of strengths of the strongest paths

4.2 Algorithm

Floyd-Warshall straightforward implementation of calculating all-pairs strengths of the strongest paths is done using modification of Floyd-Warshall algorithm for calculating all-pairs shortest paths. At each update of the votes table we have to recalculate the strengths of the strongest paths between all pairs of nodes since an insert of new edge or update of existing edge may propagate to all-pairs strengths. The runtime of the algorithm is $\mathcal{O}(N^3)$, see pseudocode in Listing 1 and 2.

Algorithm 1 Pseudocode for path matrix initialization

```
function initializePathMatrix
    path = new long[N, N]
    for i ← 0, N do
        for j ← 0, N do
            path[i, j] ← UNDEFINED
        end for
    end for
end function
```

Online Floyd-Warshall recalculating all-pairs path strengths was a major issue in terms of computational cost. Thus, another solution (thanks to Anja Gruenheid) for updating strengths of the strongest paths in online fashion was eventually implemented. It required additional data structures: revised **path** matrix, $N \times N$, that stores strengths of paths in one direction, i.e. from node *a* to node *b*; **reversePath** matrix that stores strengths of the paths in the opposite direction, i.e. from node *b* to node *a*; **directEdges** matrix,

Algorithm 2 Pseudocode for calculating the all-pairs strengths of the strongest paths

```
function calculatePathStrength
    for i ← 0, N do
        for row ← 0, N do
            if i ≠ row then
                for col ← 0, N do
                    if i ≠ col ∧ row ≠ col then
                        if path[row, i] ≠ UNDEFINED ∧ path[i, col] ≠ UNDEFINED then
                            newScore ← min(path[row, i], path[i, col])
                            currScore ← path[row, col]

                            if (currScore = UNDEFINED) ∨ (newScore > currScore) then
                                path[row, col] ← newScore
                            end if
                                                    ▷ Update the path strength with "Min/Max" score
                        end if
                    end if
                end for
            end if
        end for
    end for
end function
```

$N \times N$, that holds boolean values indicating whether there exist a direct edge (vote) between two nodes; **nodePositions**, a hash map storing position of a node in a path matrix; and **nodeCounter** that keeps track of those alternatives which have already been voted for. Implementation details of revised *calculatePathStrength* method in online fashion are omitted here.

4.3 Tolerating Contradictions

Results of the first runs of the algorithm showed that implemented decision function with non-zero error rates of crowd accuracy (explained in Experiments chapter 5) produced a very high number of incorrect answers. Thus, we had to deviate from or improve our decision function in order to deal with conflicts in the votes graph and force the system to start "cleaning" itself by recognizing contradictions within stored results and between stored and new results.

4.3.1 Verify Random Edge Approach

In this approach upon query arrival to compare two nodes, we first probe the database and see whether it has any result for this query. Having stored answer for the query, probe the crowd for the same query again. If the crowd agrees with stored answer, do not change the stored answer; otherwise, ask the crowd again. If the first and second crowd answers agree with each other, update stored result with new aggregated crowd votes,

otherwise, do not change stored result. Pseudocode for verification procedure is shown in Listing 3.

Algorithm 3 Pseudocode for verifying answer

```
function verifyAnswer(answer)
    checkedAnswer = accessCrowd()                     ▷ Check current inference answer with crowd
    if checkedAnswer ≠ answer then                    ▷ Hit contradiction, improve result

        verifiedAnswer = accessCrowd()                ▷ Verify new crowd votes are indeed correct
        if verifiedAnswer = checkedAnswer then
                                                      ▷ New crowd votes are indeed correct

            pos = checkedAnswer.agreeVotes + verifiedAnswer.agreeVotes
            neg = checkedAnswer.disagreeVotes + verifiedAnswer.disagreeVotes
            inference.updateEdgeStrength(query.nodeA, query.nodeB, pos, neg)
                                                      ▷ Update data structures

            answer = inference.compareNodes(query.nodeA, query.nodeB)
                                                      ▷ Requery database to get updated answer
        end if
    end if
    return answer
end function
```

4.3.2 "Combine" Approach

Followed by moderate results of first experiments, another question raised was whether direct votes are "representable" enough as scores for paths. In the original paper, [13], strength of the direct link $a \to b$ can be measured in 4 ways:

1. by winning votes, i.e. $N[a,b]$,
2. by losing votes, i.e. $N[b,a]$,
3. by margin, i.e. $N[a,b] - N[b,a]$
4. by ratio, i.e. $\frac{N[a,b]}{N[b,a]}$

Our first implementation was based on measuring link strength by winning votes. Since this solution did not show good performance we developed inference "Combine" approach with link strength being represented by margin, or difference of weights between winning and losing edge.

4.4 Summary

We eventually tried variations of inference *Combine* and *NoDrop* approaches tied with verification of random edges and without, see Chapter 5 for experimental results.

Experiments

In this chapter we will show experimental results of the proposed solution. We also describe methodology used for experiments including crowd simulation, benchmarking framework and baseline solutions against which the proposed solution was compared.

5.1 Methodology

The purpose of running experiments is to measure the performance of proposed inference solutions. Following approach taken in entity resolution we want to measure the following:

1. **Cost** - number of questions asked to the crowd.
2. **Correctness** - number of incorrect answers returned by the system.
3. **Completeness** - number of unknown answers returned by the system.

At first we need to do some preliminary work such as modeling and simulating the crowd, building framework for benchmarking and defining baseline solutions against which the inference solution will be compared.

5.1.1 Crowd Simulation

Crowd Model has the following interface: it can be asked question to return whether one entity is better than the other, *askQuestion(nodeA,nodeB)*; it might make mistakes depending on the error rate, *twistAnswer(answer)*; it returns answer for the query, *getAnswer()*; and it returns number of votes it agrees and disagrees with "ground truth", or gold standard, *getPositiveVotes()* and *getNegativeVotes()* respectively. When returning the answer to the question whether *nodeA* is better than *nodeB*, crowd returns one of the following answers {*TRUE*, *FALSE*, *UNKNOWN*}, where *UNKNOWN* corresponds to "Do-not-know" in decision function notation.

Gold Standard represents the true ranking between objects. Main requirement for gold standard graph is it must be directed acyclic graph. For our experiments we used basic

ranking of objects, namely $1 > 2 > 3 > 4 > ... > N$. This way 1 is the best entity and beats all other objects, 2 wins all except 1, and N is the least preferable object.

Knowing the true relation between entities and error rate with which people do mistakes, we can simulate crowd giving wrong results. Listing 4 shows algorithm used for simulating answers given by the crowd whenever it is asked to compare two nodes.

Crowd Access Model At first we had a multi-accesses idea of modeling the crowd; this way we have limits on the number of times we access the crowd, maximum was set to 3, and at each access we have the bulk number of questions that can be asked to the crowd, maximum was set to 10. The idea of having crowd accesses and questions per access was introduced as a form of limitation on the edge budget, since each additional question to the crowd is costly. Over the course of running experiments multi-accesses approach did not seem feasible anymore, and we only limited number of accesses to 1, and number of questions to be asked to 30.

5.1.2 Baseline Methods

The inference solutions *Combine* and *NoDrop* were compared against two baseline solutions known as *Always Crowd* and *Always Cache*.

Always Crowd In this solution, results of previous comparisons are not persisted in database. Every new request to compare two alternatives is directed to the crowd. See Listing 5 for benchmarking *Always Crowd*.

Always Cache In this solution, results of previous comparisons are persisted in database. Thus, with arrival of new request two scenarios are possible: either results are found in the database and are returned, or if two alternatives were not compared earlier and thus no results exist in the database, the new request is redirected to the crowd. Clearly, this solution does not do inference of persisted results. See Listing 6 for benchmarking *Always Cache*.

Benchmarking Inference is done in the similar way as baseline solutions with the difference that we take randomization factor into account. See Listing 7 in Appendix.

5.1.3 Parameters

Parameters listed in Table 5.1 were varied during the experiments' runs:

numNodes is a total number of alternatives to vote for. We run experiments with *numNodes* set to 1,000.

Algorithm 4 Pseudocode for crowd simulation

```
function askQuestion(nodeA, nodeB)
    truth = goldStandard.getTruth(nodeA, nodeB)
                                        ▷ Get true answer whether nodeA > nodeB
    agreeVotes ← 0
    disagreeVotes ← 0        ▷ Count how many times crowd agrees/disagrees with true value

    for i ← 0, questionsPerAccess do
        crowdAnswer ← twistAnswer(truth, ‖nodeA − nodeB‖)
                                        ▷ Simulate wrong crowd answers
        if crowdAnswer = truth then
            agreeVotes ← agreeVotes + 1
        else
            disagreeVotes ← disagreeVotes + 1
        end if
    end for

    return (agreeVotes, disagreeVotes)
end function


function twistAnswer(answer, distance)
    if distribution = closeness then
        errorRate = errorRate ∗ (numNodes − distance)/(numNodes − 1)
                                        ▷ Decrease error rate
    end if
    twisted ← coin.toss(errorRate)
    if twisted = coin.heads then
        return ¬truth
    else
        return truth
    end if
end function
```

Algorithm 5 Pseudocode for benchmarking baseline method *crowd*

```
function benchmarkCrowd(goldStandard, querySet)
    for all query ∈ querySet do
        answer = UNKNOWN
        crowdAccessCount = 0

        while (answer = UNKNOWN) ∧ (crowdAccessCount < maxCrowdAccess) ∧
(answer ≠ BUDGET_REACHED) do

            answer = crowd.askQuestion(query.nodeA, query.nodeB)
            crowdAccessCount = crowdAccessCount + 1

        end while

        compare(truth, answer)
    end for
end function
```

Algorithm 6 Pseudocode for benchmarking baseline method *cache*

```
function benchmarkCache(goldStandard, querySet)
    for all query ∈ querySet do
        answer = database.compareNodes(query.nodeA, query.nodeB)
        crowdAccessCount = 0

        while (answer = UNKNOWN) ∧ (crowdAccessCount < maxCrowdAccess) ∧
(answer ≠ BUDGET_REACHED) do

            answer = crowd.askQuestion(query.nodeA, query.nodeB)
            crowdAccessCount = crowdAccessCount + 1

            database.insertVotes(query.nodeA, query.nodeB, answer)
            answer = database.compareNodes(query.nodeA, query.nodeB)

        end while

        compare(truth, answer)
    end for
end function
```

numQueries is a total number of queries per experiment asked for comparison of alternatives. We set this parameter for 500,000 queries per experiment.

errorRate is an accuracy of the crowd when comparing alternatives *nodeA* and *nodeB*. The following values were set for simulations {0%, 5%, 10%, 15%, 20%, 25%, 30%}. For example, with 0% error rate crowd never does mistake when comparing alternatives, with 30% error rate out of 10 queries crowd might provide 7 correct and 3 incorrect results. However, error rate may be decreased depending on the distribution used, see later.

quorum is a confidence over preference. The higher the quorum, the more confidence we gain in one alternative being better than the other. Quorum values were varied between {3,5}.

maxCrowdAccess is a maximum number of accesses to the crowd. We tested with maximum crowd accesses set to {1,3}.

questionsPerAccess is maximum number of questions asked to the crowd per one crowd access. Being a form of limitation on edge budget totally we allowed *maxCrowdAccess*questionsPerAccess* amount of questions to be asked for comparison of alternatives. In our experiments we set this parameter to 10 questions per one access when $maxCrowdAccess = 3$, and to 30 when $maxCrowdAccess = 1$, thus allowing in total 30 questions to be asked to the crowd per query.

distribution represents the distribution of error rate. The worst case scenario is if we have uniform distribution of error rate, where with 30% error rate out of 10 queries crowd might provide 7 correct and 3 incorrect results, and no other factors are taken into account. More reasonable assumption is that comparing similar objects people are more prone to make mistakes than comparing "distant" objects in terms of similarity. Thus, we introduce a factor that decreases the error rate of people depending on how similar or "close" the objects are, calling this distribution "closeness". Formula for the factor is: $errorRate = decreaseFactor * errorRate$, and $decreaseFactor = (numNodes - distanceBetweenNodes)/(numNodes - 1)$. Having gold standard in the form we have now ($1 > 2 > ... > numNodes$), we can easily define what the distance between nodes is, and smaller the distance means closer or more similar the objects are, and thus it increases the *decreaseFactor* and possibility for people to make more mistakes.

isRandom if *isRandom* is unset we do not try to improve already stored or inferred result; otherwise, if stored result is known, the inference solution tries to tolerate contradictions if any, see Appendix 7 for benchmarking inference algorithm.

random is used only when *isRandom* is set, and takes values {1,10}. With $random = 1$, inference solution will try to improve every query; with $random = 10$, every 10th query will be verified.

Parameter	Value	Symbol
Number of entities to compare	1000	*numNodes*
Number of queries asked	500,000	*numQueries*
Error rate	0%, 5%, 10%, 15%, 20%, 25%, 30%	*e*
Quorum	1, 3, 5	*q*
Upper limit on crowd access	1,3	*limit* or *maxCrowdAccess*
Number of people asked at each crowd access	10,30	*bulk* or *questionsPerAccess*
Error rate distribution	uniform, closeness	*distribution*
Randomization	none, 1, 10	*random*

Table 5.1: Experimental Parameters

5.1.4 Hardware and Software Configurations

The following hardware configurations were used when running experiments: Intel(R) Xeon(R) X3360 @2.83GHz **CPU**, 4 **Cores**, 8GB **RAM**. Software used: PostgreSQL 9.1.3 Database Management System, Java 1.6.0_23.

5.2 Experimental Results

We will report experiments' results for $quorum = 3$ and *uniform* distribution of error rate. For results with *closeness* approach and $quorum = 5$, see Appendix, A.

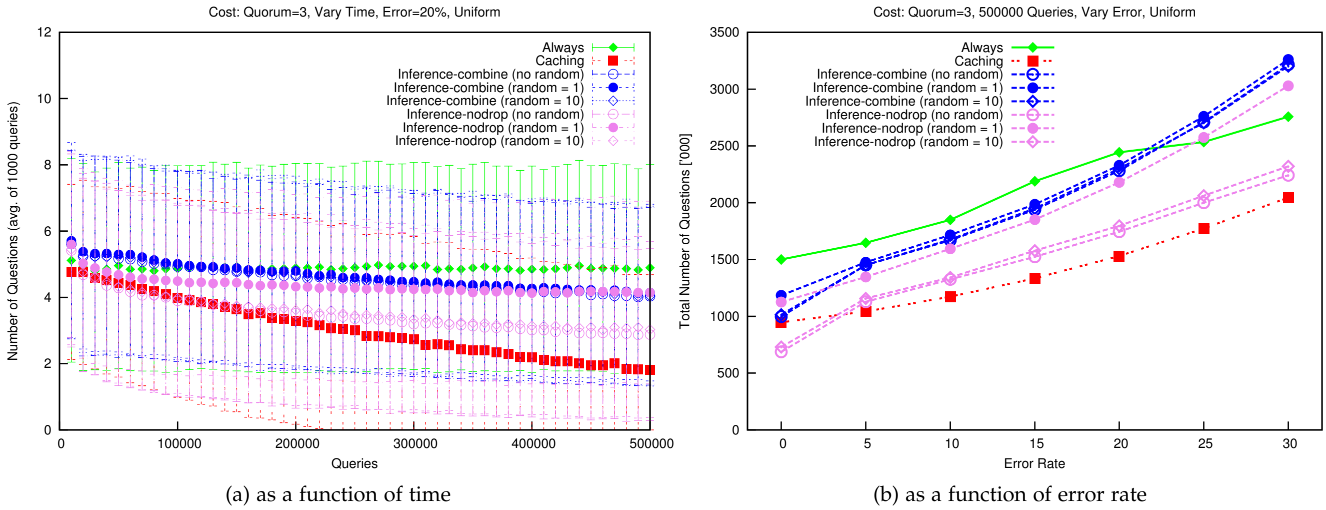
5.2.1 Cost Analysis

Since the purpose of utilizing crowdsourced database is to minimize amount of times the crowd is accessed, we measure the number of questions the crowd is asked whenever new query comes in. We do it in two ways:

Cost as a function of time where we see how the number of questions the crowd is asked varies with the number of queries asked to the crowdsourced database. We expect cost to drop since our solution is based on inference approach, i.e. we utilize transitivity property of ordering relation. When next database query is asked, there is no need to ask the crowd again – the answer to the query will either be stored already in the database (cached) or will be inferred. See Figure 5.1a. With 20% error rate we observe both inference solutions (*Combine* and *NoDrop*) have higher cost compared to baseline approach *Always Cache*, but lesser cost than *Always Crowd*. High cost is explained because with *random* set to 1 we verify every single query by asking more questions to the crowd. We will see in the correctness measure results that such tradeoff between cost and number of incorrect results is beneficial.

Cost as a function of error rate where we observe how the number of questions the crowd is asked changes when varying crowd error rates, see Figure 5.1b. We observe that

with increase of error rate the number questions is increased.



(a) as a function of time

(b) as a function of error rate

Figure 5.1: Cost measure, *uniform*, $q = 3$

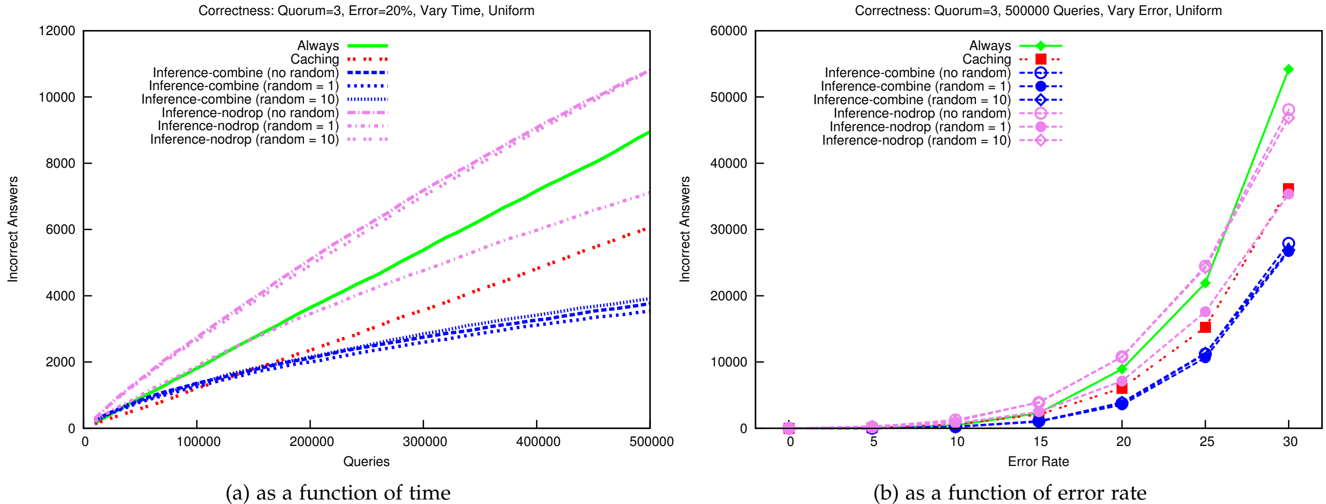
5.2.2 Correctness Analysis

Since in crowdsourced database we rely on the inferred results whenever new query comes in, we want to see whether these results are correct. Thus, we measure how many queries were answered correctly by the database. The true answer to the query is given by the gold standard which is compared against the database result. We expect crowdsourced database to deal with contradictions and "clean" itself. Similar to cost measure we will have two graphs:

Correctness as a function of time where x-axis represents the number of queries and y-axis shows the number of incorrect answers, see Figure 5.2a. We observe that number of incorrect results in the inference *Combine* approach starts dropping down much faster over the course of time. *NoDrop* solution unfortunately behaves worse than baselines solutions.

Correctness as a function of error rate where x-axis represents the error rate $\in$ [0%..30%] and y-axis shows the number of incorrect answers, see Figure 5.2b. Again the inference *Combine* method especially with higher error rates gives less incorrect results compared to baseline methods.

Thus, conclusions that can be drawn from these experiments is that inference *Combine* approach with some more adjustments does "clean" the erroneous data perviously stored in the database.



(a) as a function of time

(b) as a function of error rate

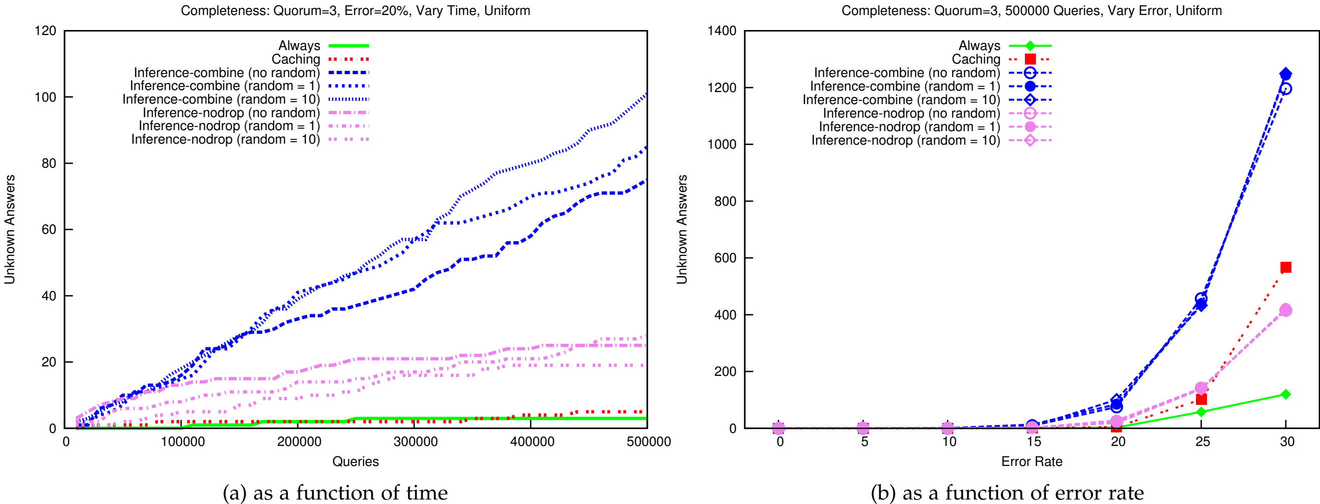
Figure 5.2: Correctness measure, *uniform*, $q = 3$

5.2.3 Completeness Analysis

Since we introduced the notion of *quorum*, we expect that database will not always be able to provide a definitive positive or negative answer to the query. Some queries with no sufficient amount of crowd votes will be marked as *UNKNOWN*. We would like to measure how many unknown results our solution provides to the queries.

Completeness as a function of time where x-axis represents the number of queries and y-axis shows the number of incomplete answers, see Figure 5.3a.

Completeness as a function of error rate where x-axis represents the error rate $\in$ [0%..30%] and y-axis shows the number of incomplete answers, see Figure 5.3b.



(a) as a function of time

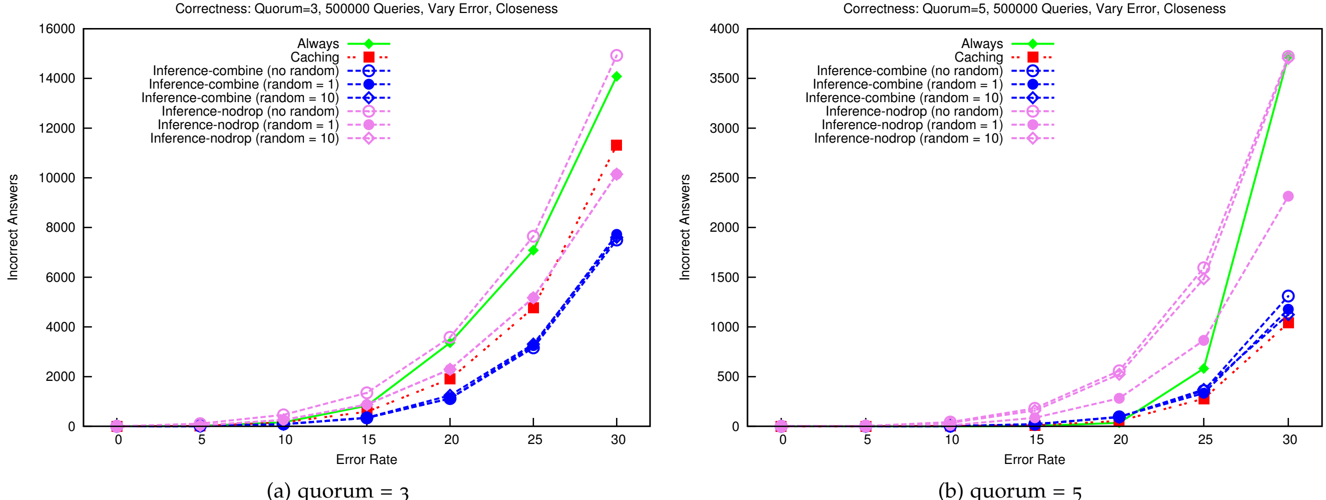
(b) as a function of error rate

Figure 5.3: Completeness measure, *uniform*, $q = 3$

We observe that number of unknown results returned by the system especially with the increase of error rate increases dramatically. Similar results were seen in entity resolution where system was behaving in a conservative way, i.e. preferring to return unknown result rather than an incorrect one.

5.2.4 Quorum Effect

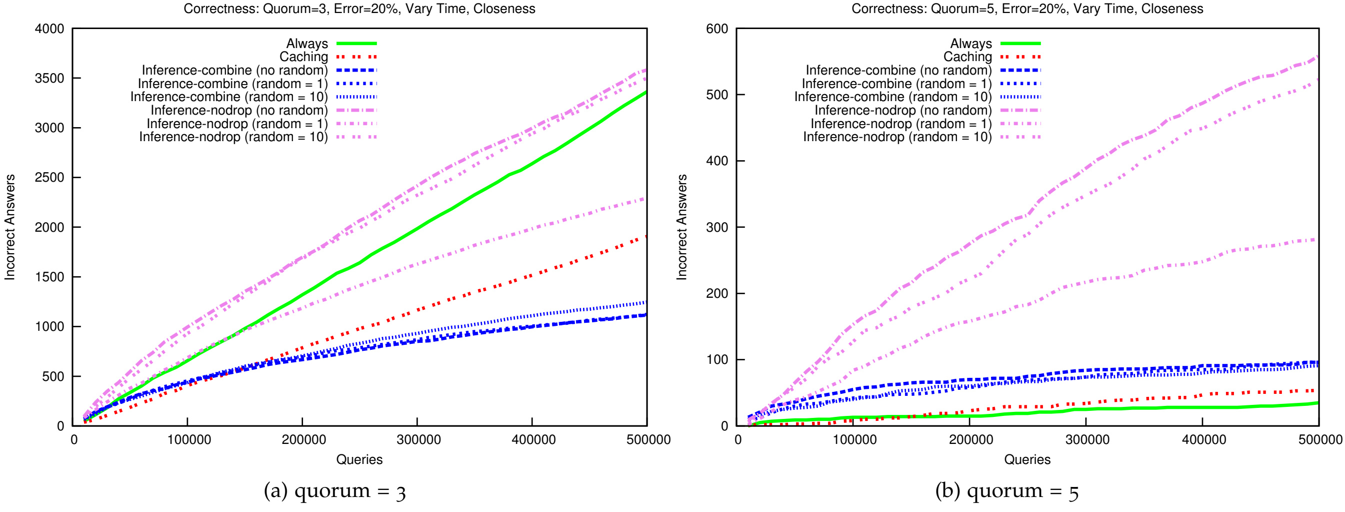
Since one of our main goals is to improve correctness measure, one of the triggers to boost the number of correct answers is to increase quorum. Intuition behind is having higher quorum gives more confidence in people's votes, and we expected number of incorrect results to be less compared to baseline approaches. However, experiments' results showed that inference solution with quorum set to 5 performs even worse than baseline methods, see Figure 5.4a vs. Figure 5.7b for *closeness* distribution, *uniform* distribution behaves analogously. Ideal setting when we observe significant drop in the number of incorrect results is with quorum set to 3.



(a) quorum = 3 (b) quorum = 5

Figure 5.4: Correctness measure as a function of error rate, *closeness*

The only advantage observed with quorum set to 5 is the rate of decrease of the number of incorrect results. When looking at the correctness graphs as a function of time, see Figure 5.5a vs. Figure 5.5b, we see that amount of wrong results drops faster, i.e. the ratio of incorrect results to the number of queries to the database decays quicker with every next batch of queries. Thus, increasing the running time of experiments by issuing more queries asked to the database is a direction from which we can make some more conclusions on benefits of setting quorum to 5.

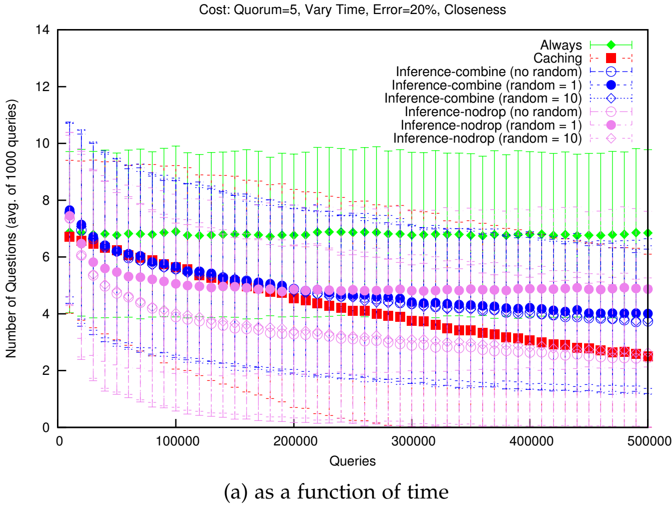


(a) quorum = 3 (b) quorum = 5

Figure 5.5: Correctness measure as a function of time, *closeness*

5.2.5 Randomization Effect

The whole idea of introducing randomization to the inference solution was to tolerate contradictions and "clean" incorrect results stored or already inferred, since in our initial inference solution the database was not improving results and number of incorrect answers was higher compared to baseline methods. Since randomly verifying results means asking more questions, we expect cost to increase and correctness to improve. Our suppositions were realized for inference *Combine* approach, and we do get significant drop of incorrect results when we do randomly verify queries' results. Interestingly, verifying every 10th or every single result does not make a huge difference i.t.o. number of incorrect results, see Figure 5.2b. Ideally, we do not want to verify every single result since it might get costly, and settings with $random = 10$ should be more favorable; though looking at the cost graph, Figure 5.1a, we see that number of questions asked to crowd with every 10th query verified compared to every single question improved is not less. However, for inference *NoDrop* approach we see the opposite picture. The number of incorrect results is not dropping significantly and performs worse than baseline solution. But, the number of questions asked to the crowd is decreasing with $random = 10$ compared to asking every single question, especially with $q = 5$ and *closeness* distribution, Figure 5.6a, the cost is better than all baseline methods. Thus, as a general observation, randomization is a good direction for improving current inference solutions, we need to understand how to improve cost of inference *Combine* by using features of *NoDrop*, or the other way round. More thoughts must be given into finding ideal settings to balance cost and correctness measures.



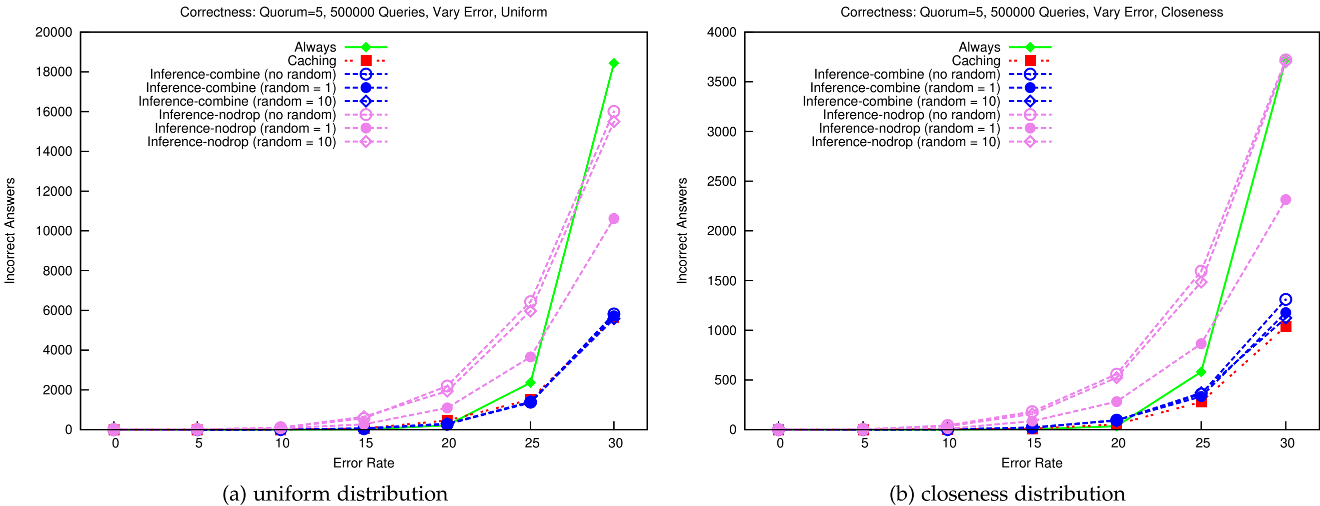
(a) as a function of time

Figure 5.6: Cost measure, *closeness*, $q = 5$

5.2.6 Error Rate Distribution Effect

We did not observe any significant differences in the patterns of behaviors of both inference solutions against baseline when varying distribution parameter. The only difference

is the number of incorrect and unknown results is much less when using *closeness* distribution compared to *uniform*. It is easily explainable, namely with *closeness* set, there are more chances to decrease error rate of the crowd, see Figure 5.7a vs. 5.7b.



(a) uniform distribution

(b) closeness distribution

Figure 5.7: Correctness measure as a function of error rate, $q = 5$

Conclusion

In this chapter we will summarize what we achieved and did not during this master thesis, and what future research in the field of crowdsourced ordering can be done.

6.1 Summary

This master thesis aimed at tackling the problem of ranking entities from crowdsourced data. Ranking procedures are generally broken down into two steps: a) be able to answer preference question "whether entity a is better than entity b" by producing strict partially ordered set, b) given such final set generate ranking of entities. We concentrated at first on subproblem a) since it is the base for moving towards subproblem b). The idea was to discover such decision function that given two entities as an input returns preference relation between them. Having done preliminary background research in voting theory, sports tournament and graph theory, we picked path based "Min/Max" scoring function from elections schemes by M.Schulze, [13]. Unfortunately, after the first run of experiments implementation of this scoring function in its original form did not show good performance – number of incorrect results was significantly higher compared to baseline solutions *Always Crowd*, where we crowdsource every incoming query to the crowd, and *Always Cache*, where we only store voting results of previous queries with no inference. After some analysis we came to conclusion that once we start introducing error rates to the crowd, and as a result get more incorrect answers from people, inference method based on original scoring function does not fight with conflicts introduced by the wrong results to the system, but rather propagates these inferred incorrect results and introduces even more contradictions. Actions were taken to tolerate contradictions in the system by first changing the definition of the link strength: originally, it was represented by the winning votes edge only, afterwards we measured link strength with the difference between the winning and losing edge, so-called margin. Second improvement was to verify randomly picked answer that has already been stored in the database with the crowd. We ran experiments with combined approach of link being represented by the difference between votes and verification of every single and every 10th query. As conclusion, combined inference approach significantly decreased the number of incorrect results compared to baseline solutions, but at the cost of more questions asked to the crowd and more unknown answers returned.

6.2 Future Work

We made a first good step in solving the problem of defining preference relation between entities, subproblem a). Certainly plenty of room is left for improvement.

Ranking Next step in ranking procedure after having defined preference relation between entities is to generate the actual ranking of entities, subproblem b). Brute-force approach will run a topological sort on the graph based on preference relation set. However, more interesting techniques, especially from sports tournament can be applied.

Rank Correlation If the ranking of the entities is done, next step is to check the quality of produced ranking. The problem of 1938 is known as "measuring the rank correlation" and some metrics already exist, such as Kendall's τ which determines how resulting ranking is close the actual one, i.e. rank correlation between actual and resulting rank, and Fleiss' κ which determines when the sorting is ambiguous. Kendall's metric is a well-established approach, but more thoughts could be put in here, and this is left as future work.

Bibliography

[1] 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*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989331

[2] "CrowdDB." [Online]. Available: http://www.crowddb.org

[3] J. Howe, "The rise of crowdsourcing," *Wired*, 2006. [Online]. Available: http://www.wired.com/wired/archive/14.06/crowds.html

[4] E. Estellés-Arolas and F. González-Ladrón-De-Guevara, "Towards an integrated crowdsourcing definition," *J. Inf. Sci.*, vol. 38, no. 2, pp. 189–200, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1177/0165551512437638

[5] A. Sherman, "18 tasks you can crowdsource," 2010. [Online]. Available: http://gigaom.com/collaboration/18-tasks-you-can-crowdsource/

[6] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom, "Deco: Declarative crowdsourcing," Stanford University, Technical Report, 2012. [Online]. Available: http://ilpubs.stanford.edu:8090/1015/

[7] A. Marcus, E. Wu, S. Madden, and R. C. Miller, "Crowdsourced databases: Query processing with people," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. CIDR, Jan. 2011, pp. 211–214. [Online]. Available: http://dspace.mit.edu/handle/1721.1/62827

[8] S. Guo, A. Parameswaran, and H. Garcia-Molina, "So who won? dynamic max discovery with the crowd," Stanford University, Technical Report, November 2011. [Online]. Available: http://ilpubs.stanford.edu:8090/1032/

[9] F. Widmer, "Memoization of crowd-sourced comparisons," 2012. [Online]. Available: http://opac.nebis.ch/F/?local_base=NEBIS&func=find-b&find_code=SYS&request=007305178

[10] J. Levin and B. Nalebuff, "An introduction to vote-counting schemes," *Journal of Economic Perspectives*, vol. 9, no. 1, pp. 3–26, Winter 1995. [Online]. Available: http://ideas.repec.org/a/aea/jecper/v9y1995i1p3-26.html

[11] N. Nisan, "Introduction to mechanism design (for computer scientists)."

[12] "Candidates for UN secretary general," 2006. [Online]. Available: http://www.unsg.org/candidates.html

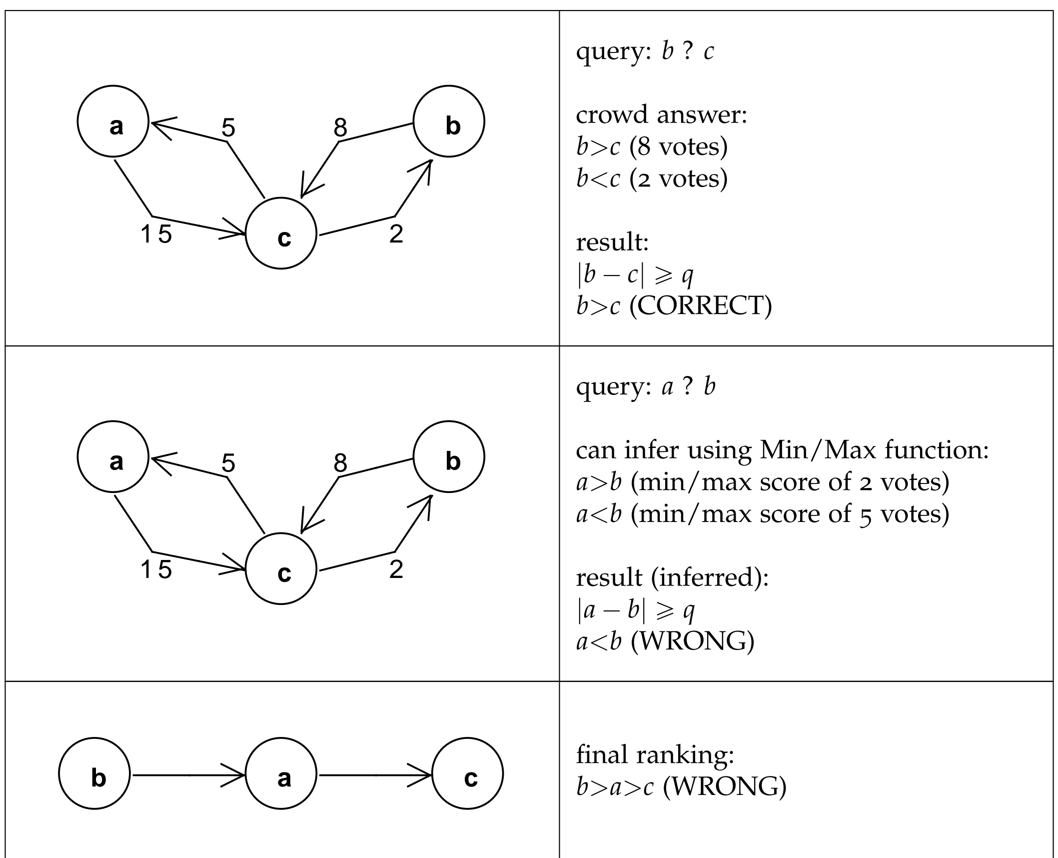
[13] M. Schulze, "A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method," *Social Choice and Welfare*, vol. 36, pp. 267–303, 2011, 10.1007/s00355-010-0475-4. [Online]. Available: http://dx.doi.org/10.1007/s00355-010-0475-4

Appendix

A.1 “Min/Max” Decision Function

Example A.1 shows “Min/Max” decision function with both winning and losing direct edges preserved produces wrong results.

Figure	Query
a b c	true ranking: $a>b>c$
a 5 5 c	query: a ? c crowd answer: $a>c$ (5 votes) $a<c$ (5 votes) result: $\|a-c\| < q$ $a>c$ (UNKNOWN)
a 5 15 c	query: a ? c crowd answer: $a>c$ (10 votes) $a<c$ (0 votes) total amount of crowd votes for a and c: $a>c$ (15 votes) $a<c$ (5 votes) result: $\|a-c\| \geqslant q$ $a>c$ (CORRECT)



Graph	Description
a, b, c; 5, 8, 15, 2	query: b ? c crowd answer: $b>c$ (8 votes) $b<c$ (2 votes) result: $\lvert b-c \rvert \geqslant q$ $b>c$ (CORRECT)
a, b, c; 5, 8, 15, 2	query: a ? b can infer using Min/Max function: $a>b$ (min/max score of 2 votes) $a<b$ (min/max score of 5 votes) result (inferred): $\lvert a-b \rvert \geqslant q$ $a<b$ (WRONG)
b, a, c	final ranking: $b>a>c$ (WRONG)

Table A.1: Example of incorrect results: quorum = 3, error rate > 0%

A.1.1 Compliance With Voting System Criteria

Any voting scheme ideally must fulfill several "fairness" criteria. Due to the Arrow's impossibility theorem it is impossible to satisfy all criteria since some desirable features contradict each other.

Unrestricted domain, or universality says that voting systems must account for **all** individual preferences. It must do so in a manner that results in a complete ranking of preferences, and it must deterministically provide the same ranking each time voters' preferences are presented in the same way.

Non-dictatorship the social welfare function cannot simply mimic the preferences of a single winner, it should account for the wishes of multiple voters.

Pareto efficiency, or unanimity says that election method must respect unanimous opinions, i.e. if **every** individual prefers a certain alternative to another, then so must the resulting societal preference order. It comes in two versions:

1. If every voter strictly prefers alterative a to alternative b, then alternative a must perform better than alternative b;
2. If no voter strictly prefers alternative b to alternative a, then alternative b must not perform better than alternative a.

Independence of irrelevant alternatives social preference between a and b must depend **only** on the individual preference between a and b.

Resolvability this criterion ensures that voting system has a low possibility of ties. The definition comes in two versions:

- By N.Tideman, resolvability says that, when there is more than winner, then, for every winning alternative, it is sufficient to add a single ballot so that the winning alternative becomes the unique winner.
- By D.Woodall, resolvability says that, the proportion of profiles without a unique winner tends to zero as the number of votes in the profile tends to infinity.

Reversal symmetry requires that if alternative a is the unique winner in the original situation, the alternative a must not be a winner in the reversed situation when all votes are reversed. It cannot happen that alternative a is the best and simultaneously the worst.

Monotonicity, or mono-raise says that when voters rank alternative a higher and without changing the order in which they rank other alternatives relatively to each other, then this must not hurt alternative a.

Independence of clones a *clone* in voting is the strategic nomination of a large number of similar alternatives. This criterion states that running an election with clones must not have any impact on the elections result.

Comparison With Other Methods

Proposed *Schulze method* satisfies all of the most important voting criteria, see Table A.2. Since it also satisfies properties of the desired decision function for our problem, it has been chosen as a decision function for the crowdsourced ordering.

Method/Criteria	Pareto-optimal?	Majority?	Condorcet?	Clone-independent?	Monotonic?	Smith set?	Reverse-symmetric?	Resolvability
Plurality	Yes	Yes	No	No	Yes	No	No	Yes
Runoff	Yes	Yes	No	Yes	Yes	No	No	Yes
Coombs	Yes	No	No	No	No	No	No	$\times$
Borda	Yes	No	No	No	Yes	No	Yes	Yes
Copeland	Yes	Yes	Yes	No	Yes	Yes	Yes	No
Tideman (Ranked Pairs)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Simpson-Kramer (Min-Max Rule)	Yes	Yes	Yes	No	Yes	No	No	Yes
Kemeny-Young	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Schulze	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table A.2: Comparison of Election Methods

A.2 Benchmarking Inference Methods

Similar to *Always Crowd* and *Always Cache* we developed a benchmarking method, Listing 7, for inference solutions. The main difference is *updateEdgeStrength* method of inference methods. See Listing 8 and 9 for pseudocodes of updating edge strength for both *Combine* and *NoDrop* methods.

Algorithm 7 Pseudocode for benchmarking inference method

```
function BENCHMARKINFERENCE(goldStandard, querySet)
    cnt ← 0
    for all query ∈ querySet do
        cnt ← cnt + 1

        truth = goldStandard.getTruth(query.nodeA, query.nodeB)
        answer = inference.compareNodes(query.nodeA, query.nodeB)

        pos ← 0
        neg ← 0
        if isRandom ∧ (((cnt − 1)%random) = 0) ∧ (answer ≠ UNKNOWN) ∧ (answer ≠
BUDGET_REACHED) then        ▷ Random approach and dealing with contradictions in case of known result

            answer = verifyAnswer(answer)        ▷ Verify stored answer, improve in case of contradictions

        else                                     ▷ Non-random approach or when stored answer is unknown
            if answer = UNKNOWN then
                answer = accessCrowd()
                pos = answer.agreeVotes
                neg = answer.disagreeVotes
                inference.updateEdgeStrength(query.nodeA, query.nodeB, pos, neg)
            end if
        end if

        compare(truth, answer)
    end for
end function


function ACCESSCROWD
    crowdAccessCount = 0
    while crowdAccessCount < maxCrowdAccess do
        answer = crowd.askQuestion(query.nodeA, query.nodeB)
        crowdAccessCount = crowdAccessCount + 1
    end while
end function
```

Algorithm 8 Pseudocode for updating edge strength of method *combine*

```
function UPDATEEDGESTRENGTH(nodeA, nodeB, positive, negative)
    pos_nodeA ← nodePositions.get(nodeA)
    pos_nodeB ← nodePositions.get(nodeB)
    difference ← positive − negative

    if (pos_nodeA = UNDEFINED) ∧ (pos_nodeB = UNDEFINED) then
        nodePositions.put(nodeA, nodeCounter++)
        nodePositions.put(nodeB, nodeCounter++)
        path[pos_nodeA, pos_nodeB] ← difference
        reversePath[pos_nodeB, pos_nodeA] ← difference ∗ (−1)

    else if (pos_nodeB = UNDEFINED) then
        nodePositions.put(nodeB, nodeCounter++)
        path[pos_nodeA, pos_nodeB] ← difference
        reversePath[pos_nodeB, pos_nodeA] ← difference ∗ (−1)
        calculatePathStrength(pos_nodeA, pos_nodeB)
        calculateReversePathStrength(pos_nodeB, pos_nodeA)

    else if (pos_nodeA = UNDEFINED) then
        nodePositions.put(nodeA, nodeCounter++)
        path[pos_nodeB, pos_nodeA] ← difference ∗ (−1)
        reversePath[pos_nodeA, pos_nodeB] ← difference
        calculatePathStrength(pos_nodeB, pos_nodeA)
        calculateReversePathStrength(pos_nodeA, pos_nodeB)

    else
        if pos_nodeA < pos_nodeB then
            if path[pos_nodeA, pos_nodeB] = UNDEFINED then
                path[pos_nodeA, pos_nodeB] ← difference
                reversePath[pos_nodeB, pos_nodeA] ← difference ∗ (−1)
            else
                path[pos_nodeA, pos_nodeB] ← path[pos_nodeA, pos_nodeB] + difference
                reversePath[pos_nodeB, pos_nodeA] ← reversePath[pos_nodeB, pos_nodeA] + difference ∗ (−1)
            end if
            calculatePathStrength(pos_nodeA, pos_nodeB)
            calculateReversePathStrength(pos_nodeB, pos_nodeA)
        else
            if path[pos_nodeB, pos_nodeA] = UNDEFINED then
                path[pos_nodeB, pos_nodeA] ← difference ∗ (−1)
                reversePath[pos_nodeA, pos_nodeB] ← difference
            else
                path[pos_nodeB, pos_nodeA] ← path[pos_nodeB, pos_nodeA] + difference ∗ (−1)
                reversePath[pos_nodeA, pos_nodeB] ← reversePath[pos_nodeA, pos_nodeB] + difference
            end if
            calculatePathStrength(pos_nodeB, pos_nodeA)
            calculateReversePathStrength(pos_nodeA, pos_nodeB)
        end if
    end if

    directEdges[pos_nodeA, pos_nodeB] ← TRUE
    if directEdgeMatrix[pos_nodeA, pos_nodeB] = UNDEFINED then
        directEdgeMatrix[pos_nodeA, pos_nodeB] = difference
    else
        directEdgeMatrix[pos_nodeA, pos_nodeB] = directEdgeMatrix[pos_nodeA, pos_nodeB] + difference
    end if
end function
```

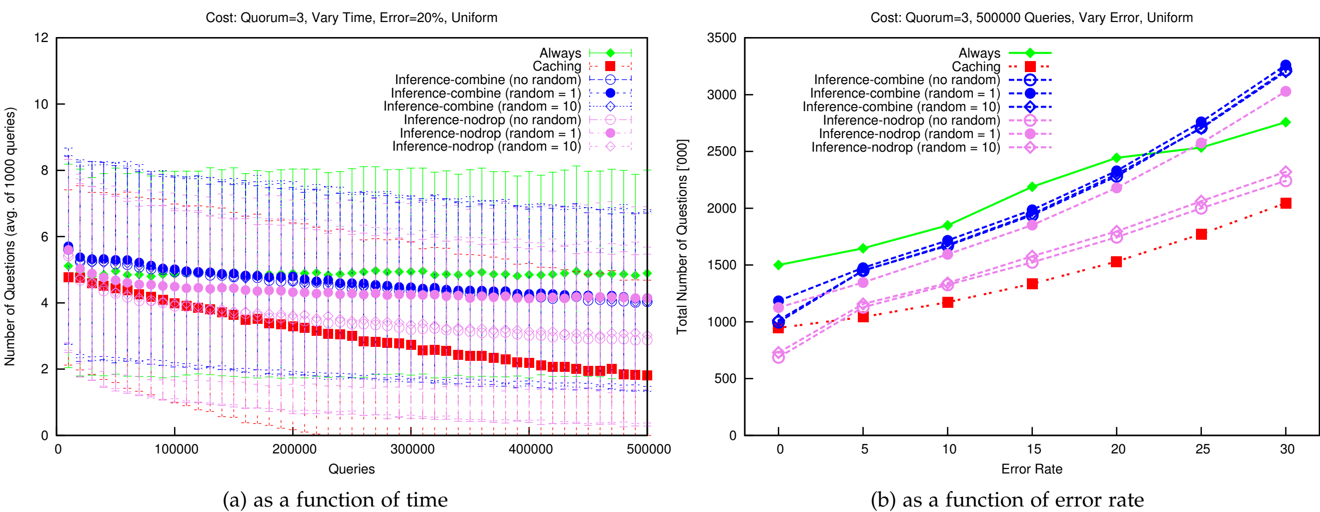
Algorithm 9 Pseudocode for updating edge strength of method *nodrop*

```
function updateEdgeStrength(nodeA, nodeB, positive, negative)
    pos_nodeA ← nodePositions.get(nodeA)
    pos_nodeB ← nodePositions.get(nodeB)

    if (pos_nodeA = UNDEFINED) ∧ (pos_nodeB = UNDEFINED) then
        nodePositions.put(nodeA, nodeCounter++)
        nodePositions.put(nodeB, nodeCounter++)
        path[pos_nodeA, pos_nodeB] ← positive
        reversePath[pos_nodeB, pos_nodeA] ← negative

    else if (pos_nodeB = UNDEFINED) then
        nodePositions.put(nodeB, nodeCounter++)
        path[pos_nodeA, pos_nodeB] ← positive
        reversePath[pos_nodeB, pos_nodeA] ← negative
        calculatePathStrength(pos_nodeA, pos_nodeB)
        calculateReversePathStrength(pos_nodeB, pos_nodeA)

    else if (pos_nodeA = UNDEFINED) then
        nodePositions.put(nodeA, nodeCounter++)
        path[pos_nodeB, pos_nodeA] ← negative
        reversePath[pos_nodeA, pos_nodeB] ← positive
        calculatePathStrength(pos_nodeB, pos_nodeA)
        calculateReversePathStrength(pos_nodeA, pos_nodeB)

    else
        if pos_nodeA < pos_nodeB then
            if path[pos_nodeA, pos_nodeB] = UNDEFINED then
                path[pos_nodeA, pos_nodeB] ← positive
                reversePath[pos_nodeB, pos_nodeA] ← negative
            else
                path[pos_nodeA, pos_nodeB] ← path[pos_nodeA, pos_nodeB] + positive
                reversePath[pos_nodeB, pos_nodeA] ← reversePath[pos_nodeB, pos_nodeA] + negative
            end if
            calculatePathStrength(pos_nodeA, pos_nodeB)
            calculateReversePathStrength(pos_nodeB, pos_nodeA)
        else
            if path[pos_nodeB, pos_nodeA] = UNDEFINED then
                path[pos_nodeB, pos_nodeA] ← negative
                reversePath[pos_nodeA, pos_nodeB] ← positive
            else
                path[pos_nodeB, pos_nodeA] ← path[pos_nodeB, pos_nodeA] + negative
                reversePath[pos_nodeA, pos_nodeB] ← reversePath[pos_nodeA, pos_nodeB] + positive
            end if
            calculatePathStrength(pos_nodeB, pos_nodeA)
            calculateReversePathStrength(pos_nodeA, pos_nodeB)
        end if
    end if

    directEdges[pos_nodeA, pos_nodeB] ← TRUE
    directEdges[pos_nodeB, pos_nodeA] ← TRUE
end function
```

A.3 Experimental Results

Here we will report experimental results with all parameters varied, error rate distribution $\in \{uniform, closeness\}$ and quorum $\in \{3, 5\}$.

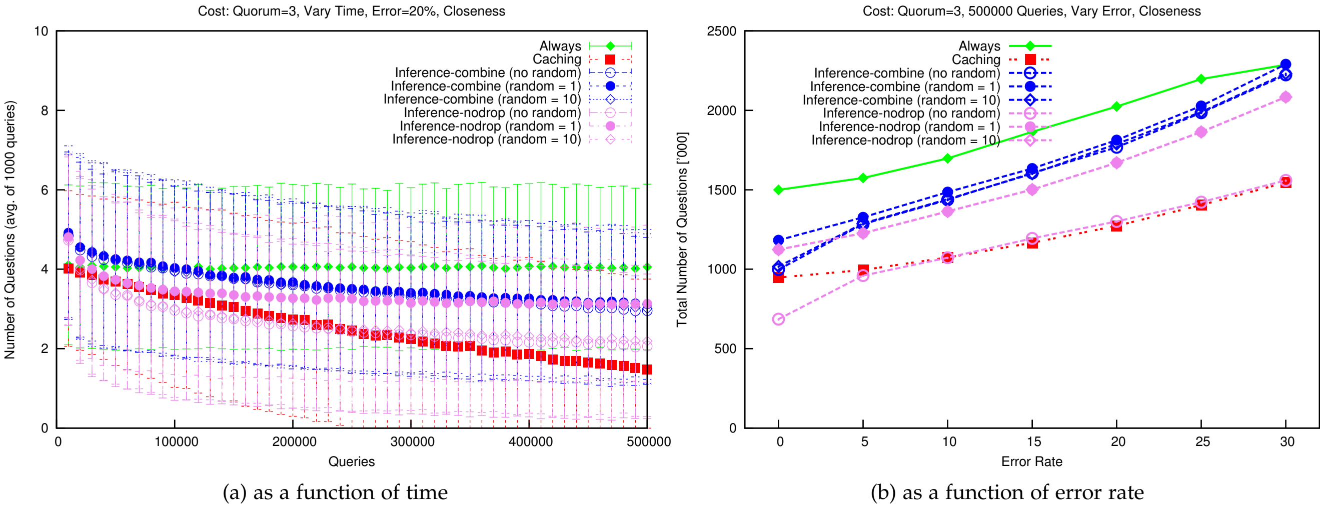
A.3.1 Cost measurements



(a) as a function of time

(b) as a function of error rate

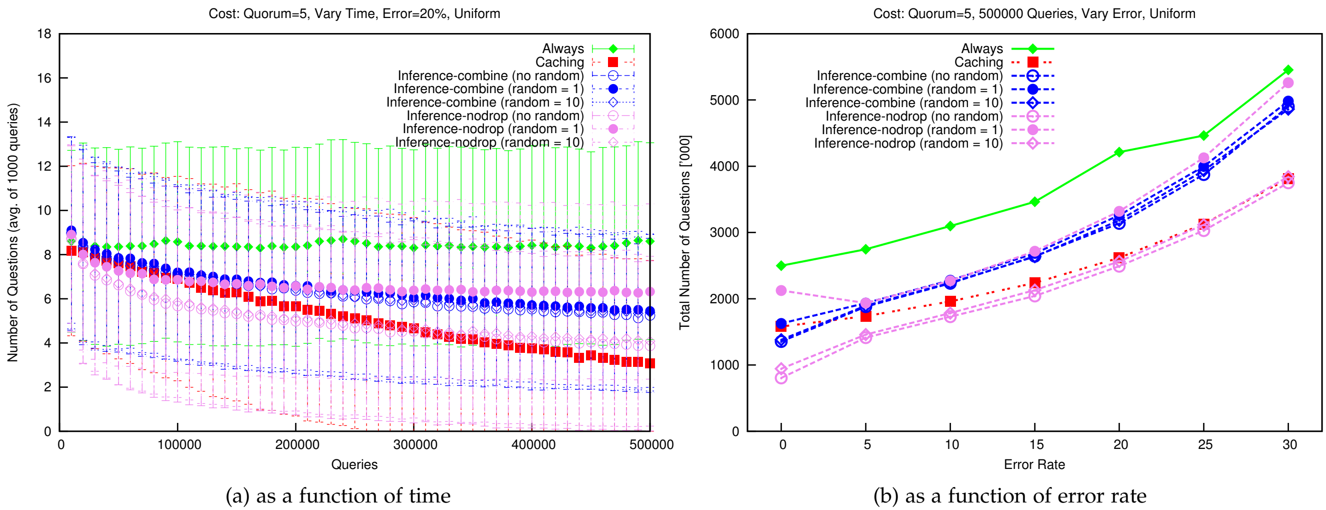
Figure A.1: Cost measure, *uniform*, $q = 3$



(a) as a function of time

(b) as a function of error rate

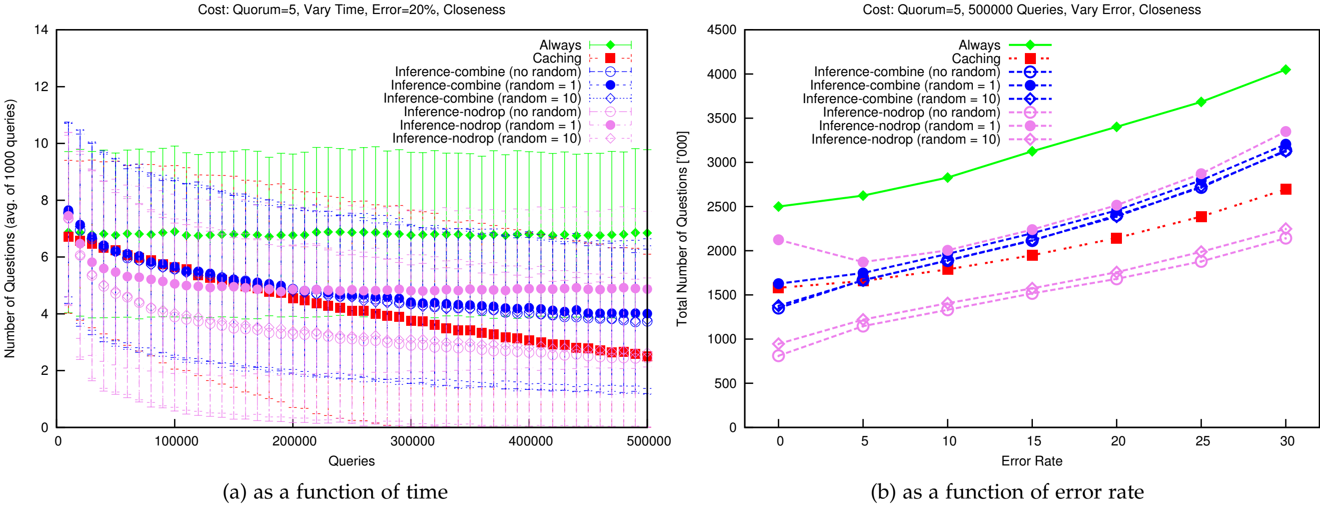
Figure A.2: Cost measure, *closeness*, $q = 3$



(a) as a function of time

(b) as a function of error rate

Figure A.3: Cost measure, *uniform*, $q = 5$

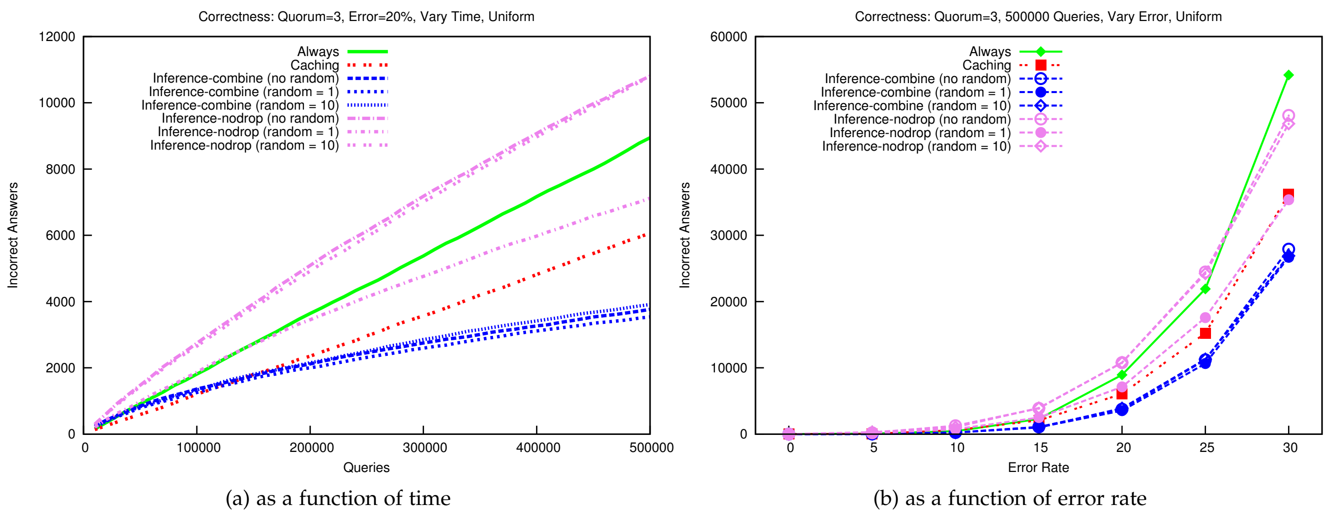


(a) as a function of time

(b) as a function of error rate

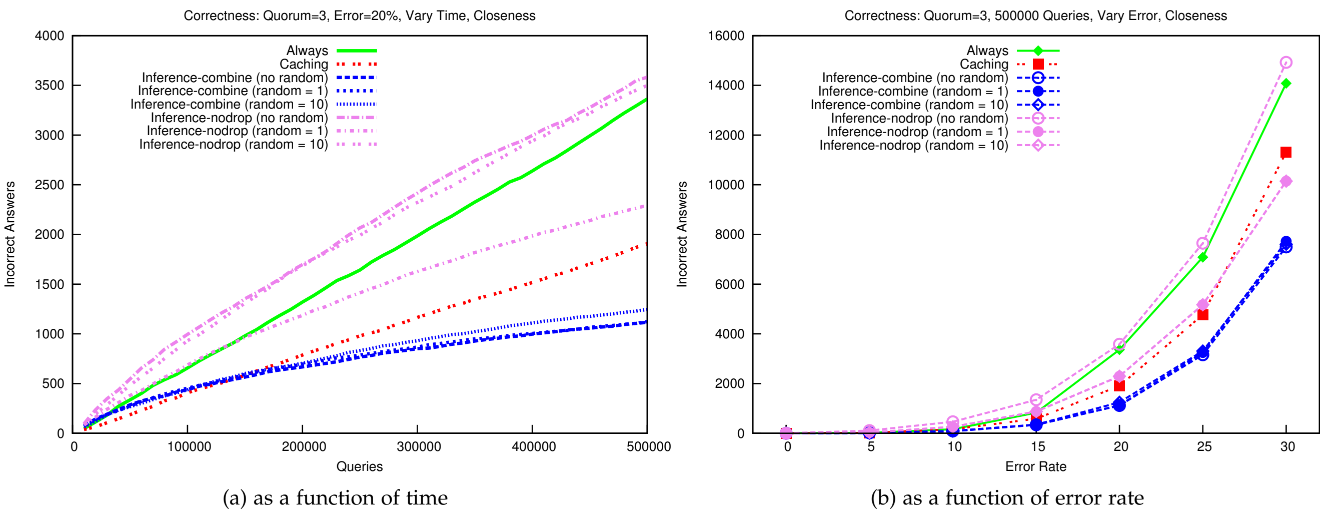
Figure A.4: Cost measure, *closeness*, $q = 5$

A.3.2 Correctness measurements



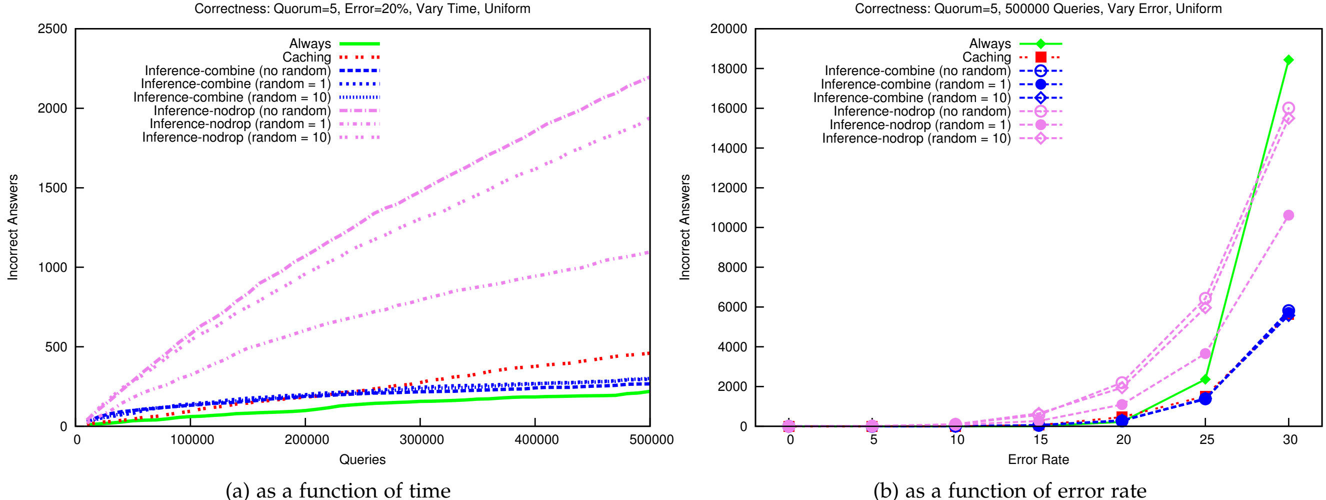
(a) as a function of time (b) as a function of error rate

Figure A.5: Correctness measure, *uniform*, $q = 3$



(a) as a function of time (b) as a function of error rate

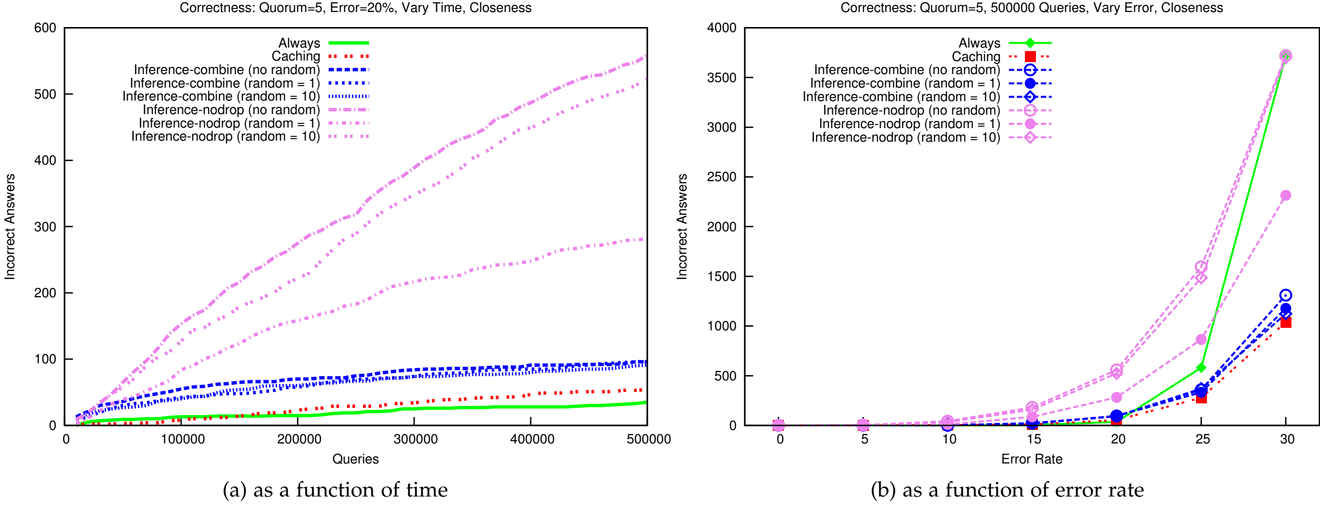
Figure A.6: Correctness measure, *closeness*, $q = 3$



(a) as a function of time

(b) as a function of error rate

Figure A.7: Correctness measure, *uniform*, $q = 5$

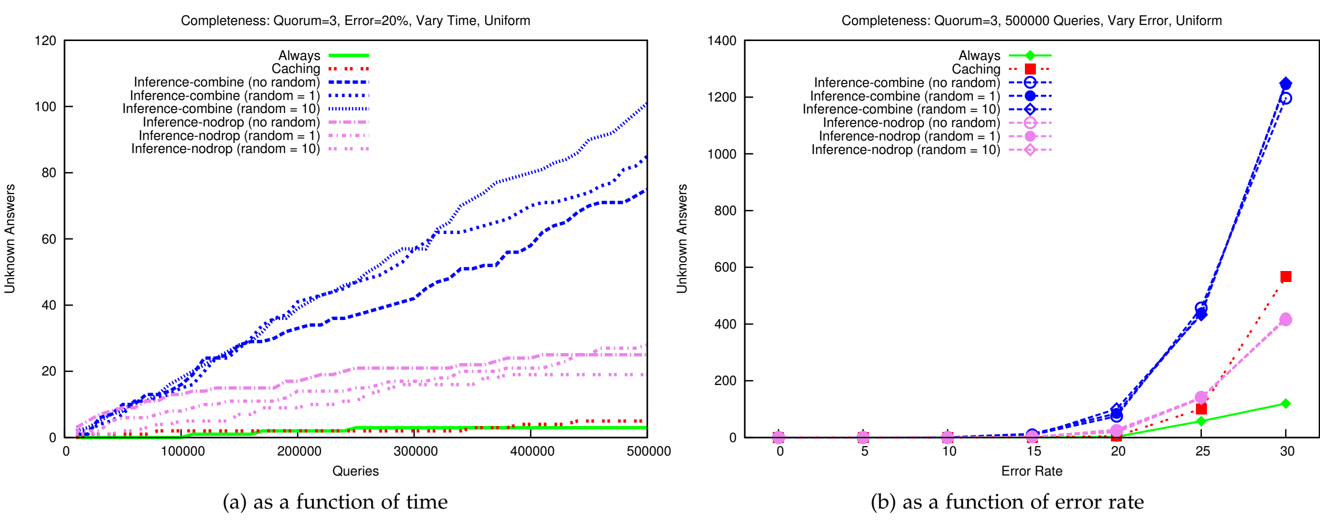


(a) as a function of time

(b) as a function of error rate

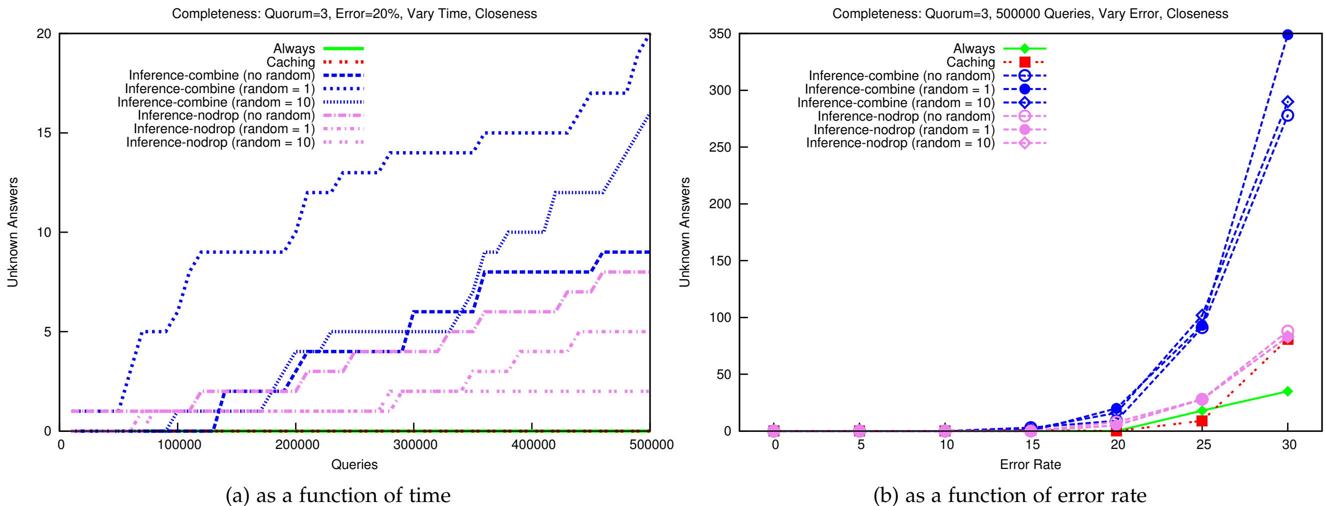
Figure A.8: Correctness measure, *closeness*, $q = 5$

A.3.3 Completeness measurements



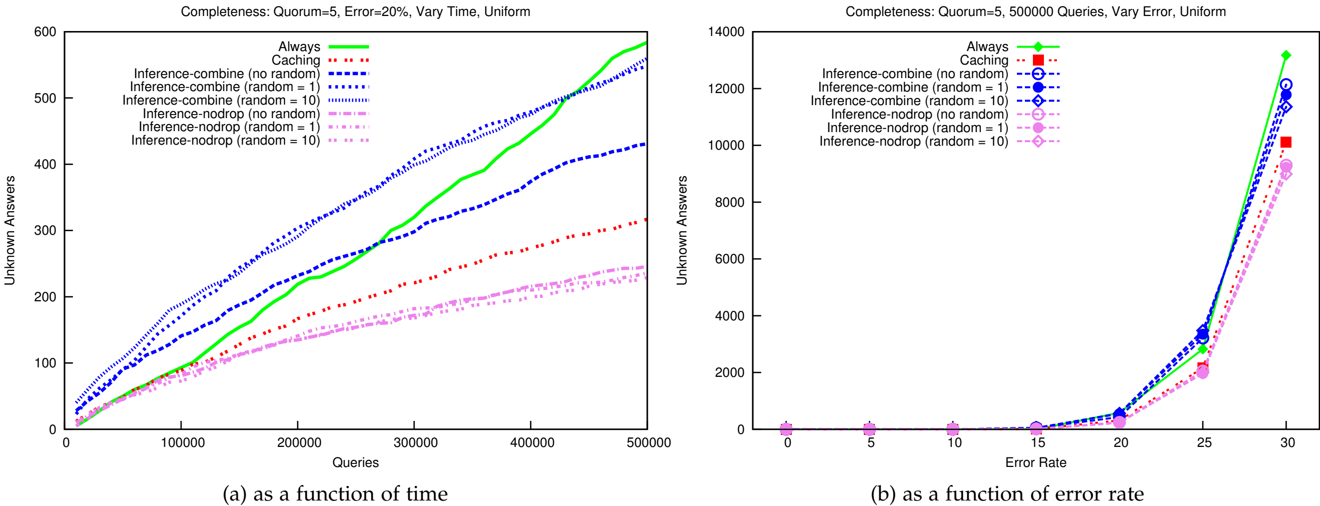
(a) as a function of time (b) as a function of error rate

Figure A.9: Completeness measure, *uniform*, $q = 3$



(a) as a function of time (b) as a function of error rate

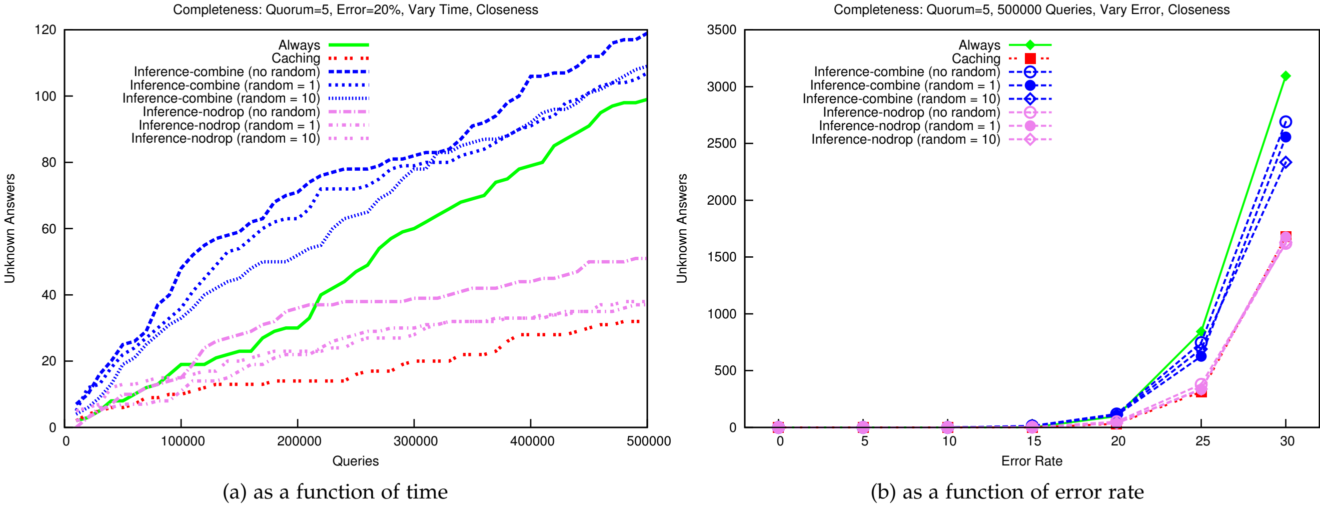
Figure A.10: Completeness measure, *closeness*, $q = 3$



(a) as a function of time

(b) as a function of error rate

Figure A.11: Completeness measure, *uniform*, $q = 5$



(a) as a function of time

(b) as a function of error rate

Figure A.12: Completeness measure, *closeness*, $q = 5$