Dynamic Max Algorithms in Crowdsourcing Environments

Petros Venetis
Stanford University
Stanford, CA 94305
venetis@cs.stanford.edu

Hector Garcia-Molina
Stanford University
Stanford, CA 94305
hector@cs.stanford.edu

Abstract

Our work investigates the problem of retrieving the maximum item from a set in crowdsourcing environments. We focus on tournament algorithms that can for instance select the best Facebook profile that matches a given person or the best photo that describes a given restaurant. Tournament algorithms can be tuned with parameters such as the desired difference of votes between the top-2 voted outcomes, and the maximum number of humans asked to perform a particular task. We propose a strategy for selecting appropriate tournament parameters that attempts to keep monetary cost and latency at a minimum while having quality guarantees. For our experiments, using a human model derived from psychometrics (the Thurstonian model), we provide insights on the effectiveness of our strategy in selecting appropriate tournament parameters and compare our techniques with previous work on tournament tuning.

1 Introduction

Humans are more effective than computers for many tasks, such as identifying concepts in images, translating natural language, and evaluating the usefulness of products. Thus, there has been a lot of recent interest in *crowdsourcing*, where humans perform tasks for pay or for fun [3, 19]. For any algorithm that is designed to run on a crowdsourcing environment, three quantities have to be considered: (1) quality of the results, (2) monetary cost, and (3) latency. We focus on the execution of *max algorithms*, i.e., algorithms that retrieve the maximum or best item from a set. Max algorithms are fundamental and needed in many applications. For example, a restaurant owner may wish to determine the most appealing photograph from a repository to advertise his restaurant. In another example, a traffic monitoring application collects photographs of various public places and attempts to determine the peak hours (based on the number of people or automobiles) presented in the set of photographs.

When a crowdsourcing algorithm is implemented naively, it may require significant amounts of money, may take too long to return results, or may return poor results. In our work we focus on the monetary and execution time savings when max algorithms are optimized. The task requester has limited resources and has to efficiently deal with the uncertainty in worker responses. If the requester had an infinite budget, he could compensate workers very well and his algorithm would run fast and (probably) accurately. In the realistic case that the requester has limited resources, the tradeoff between quality, cost, and latency has to be considered very carefully when designing the crowdsourcing algorithm.

Our work uses *dynamic* techniques to determine the number of worker responses that should be obtained in order to get a good balance between quality, cost, and latency. For example, say three workers are asked to select the max item from a set. If the workers do not agree on the max, then we can request that additional workers evaluate the same set, until we have enough confidence in the result, or until we reach some limit. If the initial three workers agree, we do not request additional evaluations. By balancing the number of worker responses across different comparisons, we are able to allocate questions to the comparisons that actually need the extra cost. In our experimental study we resort to simulations (since the models are not amenable to detailed analysis) and conclude that using dynamic techniques we can achieve the same quality as static techniques, with monetary (or execution time) savings of as much as 50%.

In the following, we present worker models obtained from psychometrics, a simple and yet realistic execution model, and explore the tradeoff between the quantities of interest in detail. There is a substantial body of work in the theory community that explores algorithms with "faulty" comparisons (e.g., [1, 2, 5, 6, 7, 8, 12, 20]), and that work is clearly applicable here. However, those works use relatively simple models that often do not capture all of the crowdsourcing issues (such as the latency of a crowdsourcing algorithm). For example, these works assume that operators (in our case humans) have a fixed probability of making an error. It is not obvious how to extend the proposed techniques to the common case in crowdsourcing where humans compare more than two items at a time and the error is related to how similar the items are.

Contributions In summary, our contributions are the following:

- We develop a framework for reasoning about dynamic max algorithms, including detailed worker and execution models (Section 2).
- We describe the most common max algorithms (tournament algorithms) and formally define our problem (Section 3),
- We describe a simple strategy for tuning a tournament algorithm, i.e., a strategy for selecting appropriate parameter values that optimize cost (or latency) given constraints on quality and latency (or cost respectively) (Section 4).
- We study the tradeoffs between quality, cost, and latency both for individual comparisons (Section 5) and for tournament algorithms (Section 6.1).
- We experimentally evaluate the tuning strategy and compare it against static techniques via simulations (Section 6.2, 6.3, and 6.4).

2 Preliminaries

Our problem's input is a set $\mathcal{E} = \{e_1, e_2, \ldots, e_{|\mathcal{E}|}\}$. Each item $e_i \in \mathcal{E}$ has a quality value $q(e_i) \in [0, \infty)$, that is different than the quality value of any other item, i.e., if $e_i, e_j \in \mathcal{E}$ with $i \neq j$, then $q(e_i) \neq q(e_j)$. The quality function $q : \mathcal{E} \to [0, \infty)$ forms a "less than" relation "$<$": If $q(e_i) < q(e_j)$, then $e_i < e_j$. The binary relation "$<$" forms a total strict order for the items in $\mathcal{E}$: For $e_i, e_j \in \mathcal{E}$ with $i \neq j$, either $e_i < e_j$ or $e_j < e_i$. The item $e_i \in \mathcal{E}$ for which $e_j < e_i$, $\forall\, j \in \{1, 2, \ldots, |\mathcal{E}|\} \setminus \{i\}$ (or equivalently, the item e_i for which $q(e_i) > q(e_j)$ for all $j \in \{1, 2, \ldots, |\mathcal{E}|\} \setminus \{i\}$) is called the *maximum item of* $\mathcal{E}$. Our goal is to retrieve the maximum item, given a set of constraints.

2.1 Comparisons

To determine the maximum item, max algorithms use a comparison operator Comp(). Operator $\mathrm{Comp}(S, r, R, \delta)$ asks humans to compare the items in set $S \subseteq \mathcal{E}$ and returns the item with the most votes (breaking ties arbitrarily). Each human selects an item from S that he believes has the highest quality value among the items in S. The number of humans comparing S depends on the parameters $R \in \{1, 2, \ldots\}$ and $\delta \in \{0, 1, \ldots\}$: R is the maximum number of humans that may be asked to compare S and δ is the desired difference of votes between the top-2 voted items. The parameter $r \in \{0, 1, \ldots\}$ is the number of initial questions posed to humans. We have $r \leq R$, such that no more than R questions are asked in any case.

The question-asking process for $\mathrm{Comp}(S, r, R, \delta)$ is described in Algorithm 1, where X is the total number of responses gathered, X_1 is the largest number of responses any one item has received, X_2 is the second largest number of responses, and so on. First, $X = r$ humans are asked to compare S. After receiving the r first responses, one additional human response is requested every time we have fewer than R responses in total and $X_1 - X_2 < \delta$. When we reach R responses or have $X_1 - X_2 \geq \delta$, an item with X_1 responses is returned. In Section 2.3 we discuss the latency/execution model we use for our experiments.

Algorithm 1: How humans are asked to compare items for Comp(S, r, R, δ).

```
  // $X$: total responses
  // $X_1$ is the largest number of responses one item has received, $X_2$ is the second
     largest number of responses, and so on
1 ask $r$ humans to select max item in $S$;
2 $X \leftarrow r$;
3 update $X_1$ and $X_2$ (using a heap);
4 while $X \leq R$ and $X_1 - X_2 < \delta$ do
5     request one additional human response;
6     $X \leftarrow X + 1$;
7     update $X_1$ and $X_2$ (using a heap);
  // more than one items may have $X_1$ votes
8 return an item with $X_1$ votes at random;
```

2.1.1 Error Model

When a worker compares a set of items, he may give correct or incorrect answers. To model worker errors we use a model suggested in psychometrics known as the Thurstonian model [16], but other error models can be used in our framework.

When a Thurstonian worker compares items in set S, he has some perception of the quality value of each $e_i \in S$ that may be different than its correct quality value $q(e_i)$. In particular, the worker's perception of the quality value of e_i is a number $\hat{e}_i$ that is a sample from the normal distribution $\mathcal{N}(q(e_i), \sigma(e_i)^2)$. That is, the perceived quality value of e_i is a sample from the normal distribution with mean the actual quality value of the item $q(e_i)$ and a standard deviation $\sigma(e_i)$. (Section 5.1 discusses how $\sigma(e_i)$ can be selected when experimental data from a real crowdsourcing environment are available.) Finally, the worker returns the item e_i with the highest perceived value $\hat{e}_i$ amongst the items S. To make the model clear, we see how a Thurstonian worker operates in a particular input set S in the following example.

Example 1 *Figure 1 presents an example of one comparison using the Thurstonian model. In this case, the worker compares three items e_1, e_2, and e_3 with quality values 800, 550, and 250 respectively. The figure presents the probability density functions (PDF) according to which the worker samples the values $\hat{e}_i$. For instance, he may sample $\hat{e}_1 = 650$, $\hat{e}_2 = 700$, and $\hat{e}_3 = 200$. Since the sample for e_2 is the largest sample, he answers incorrectly e_2 (the correct answer is e_1). In general, the closer the quality values of the items a worker compares are, the higher the probability that the worker will give an incorrect answer.*

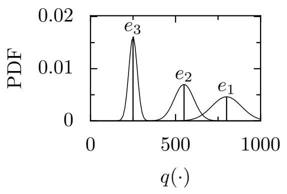


Figure 1: A set S of 3 items that a worker compares and the PDFs of the Thurstonian sampling.

The probability that a worker responds correctly, depends on the particular set S he compares. A simple and yet effective microtask difficulty metric has been defined in [17]:

$$\text{diff}(S) = \frac{q_2(S)}{q_1(S)} \tag{1}$$

where $q_i(S)$ is the i^{th} highest quality value that an item in S has. The definition only takes into account the ration of the top-2 items in S, but experimentally it has been shown to be very useful.

2.1.2 Cost Model

In the crowdsourcing environments we examine, human work is compensated: Each worker comparing items in S is compensated with 1 monetary unit. The total cost of $\text{Comp}(S, r, R, \delta)$ is the sum of its compensations and thus the cost of $\text{Comp}(S, r, R, \delta)$ is equal to the number of humans asked to compare S. In the following, we use the terms cost and questions/answers of a comparison interchangeably.

2.2 Steps

Crowdsourcing algorithms are executed in steps. A batch of microtasks (not necessarily comparisons) $\mathcal{C}_1$ is submitted during the first step to the crowdsourcing marketplace and after some time the human answers are returned. Then, depending on these answers, an appropriate set $\mathcal{C}_2$ of microtasks is selected for the second step. For the selection of the microtasks in $\mathcal{C}_i$ all human answers from previous steps $(1, 2, \ldots, i-1)$ can be considered.

2.3 Execution Model

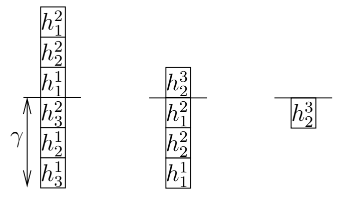
The requester's program posts a set of microtasks (not necessarily comparisons) $\mathcal{C} = \{h_1, h_2, \ldots, h_m\}$ to the marketplace and a number of human responses $r(h_i)$ are requested for microtask h_i. The crowdsourcing environment translates that into a (randomly ordered) queue of questions $\langle h_i^j \rangle$ for $i \in \{1, 2, \ldots, m\}$ and $j \in \{1, 2, \ldots, r(h_i)\}$. Within a time unit, a number of γ questions are executed by humans (or fewer if there are fewer than γ available questions in the queue). At the end of the same time unit, the requester collects all γ responses provided at that time unit and decides whether he wishes to add more questions to a microtask. The changes are reflected on the pending work queue and the cycle repeats.

In the following example we see how one step's comparison microtasks would be executed according to our model.

Example 2 *A requester's program posts to the marketplace a set $\mathcal{C}_i$ of comparisons at step i*

$$\begin{aligned} \mathcal{C}_i = \{\, & \text{Comp}(S_1, r_1, R_1, \delta_1), \text{Comp}(S_2, r_2, R_2, \delta_2), \\ & \text{Comp}(S_3, r_3, R_3, \delta_3)\} \end{aligned} \tag{2}$$

In this case, $m = 3$ and $h_i = \text{Comp}(S_i, r_i, R_i, \delta_i)$. For simplicity we assume $r_i = 2$, $R_i = 3$, and $\delta_i = 2$ for $i = 1, 2, 3$. Also, $\gamma = 3$ questions are answered at each time unit.



$t = 1$ $t = 2$ $t = 3$

Figure 2: An example of the execution of $\mathcal{C}_i$.

Figure 2 shows one possible execution of $\mathcal{C}_i$. For each time unit, the figure shows the answered questions (below the horizontal line) of that time unit and the remaining unanswered questions (above the horizontal line). The requester posts the comparisons to the marketplace and they are translated into questions. During the first time unit ($t = 1$) the first $\gamma = 3$ questions are answered by workers (h_3^1, h_2^1, and h_3^2). The two workers that performed h_3 voted the same item and since $\delta_3 = 2$ the comparison h_3 is completed and the *requester does not post any more questions for that comparison. Also, the three answered questions are removed from the queue. In the next time unit ($t = 2$), workers perform the remaining $\gamma = 3$ questions. The two workers that performed h_1 gave the same answer and thus we have a winner for S_1 (since $\delta_1 = 2$). For S_2 the requester received two different responses, and thus he posts one more question that is put in the queue by the end of the second time unit. At the third time unit ($t = 3$), one question is available in the queue and some worker performs it. Since $R_2 = 3$, a (plurality) winner is declared for S_2 and all comparisons in $\mathcal{C}_i$ have completed. In total, 3 time units were spent for the execution of the comparisons of this step.*

3 Tournament Algorithms

Our work focuses on (parameterized) tournament algorithms $\mathcal{A}_{\mathrm{T}}$, although max algorithms can be considered. Tournament algorithms operate in steps and their parameters are:

r_i**:** the initial number of human responses requested by each comparison at step i,

R_i**:** the maximum number of human responses per comparison requested at step i, and

δ_i**:** the desired difference of votes between the two most voted items of each comparison.

An algorithm A is obtained from $\mathcal{A}_{\mathrm{T}}$ when the parameters are instantiated: $A = \mathcal{A}_{\mathrm{T}}(\{r_i\}, \{R_i\}, \{\delta_i\})$, where $\{r_i\}$, $\{R_i\}$ and $\{\delta_i\}$ are sequences of r_i, R_i, and δ_i values.

Algorithm 2: $\mathcal{A}_{\mathrm{T}}(\{r_i\}, \{R_i\}, \{\delta_i\})$ operating on $\mathcal{E}$

```
1 $i \leftarrow 1$;
2 $\mathcal{E}_i \leftarrow \mathcal{E}$ ;
3 while $|\mathcal{E}_i| \neq 1$ do
4     partition $\mathcal{E}_i$ in non-overlapping sets $S_j$, with $|S_j| = 2$ (the last set can have one item);
5     $i \leftarrow i + 1$;
6     $\mathcal{E}_i \leftarrow \bigcup_j \{\mathrm{Comp}(S_j, r_i, R_i, \delta_i)\}$;
7 return $e \in \mathcal{E}_i$
```

The tournament algorithm $\mathcal{A}_{\mathrm{T}}(\{r_i\}, \{R_i\}, \{\delta_i\})$ operates on some input $\mathcal{E}$ and returns some item $e \in \mathcal{E}$, as described in Algorithm 2. In particular, $\mathcal{E}_i$ contains the items that are compared at step i ($\mathcal{E}_1 = \mathcal{E}$). At step i, $\mathcal{E}_i$ is partitioned in non-overlapping sets (S_j, for $j = 1, 2, \ldots, \lceil \frac{|\mathcal{E}_i|}{2} \rceil$) of size 2 ($|S_j| = 2$). (Sets S_j could more than 2 items each, but we only explore pairwise comparisons in this work.) The winners of each comparison $\mathrm{Comp}(S_j, r_i, R_i, \delta_i)$ form set $\mathcal{E}_{i+1}$. If $\mathcal{E}_{i+1}$ has exactly one item, this item is the output of $\mathcal{A}_{\mathrm{T}}(\{r_i\}, \{R_i\}, \{\delta_i\})$. Otherwise, the process is repeated for one more step.

3.1 Optimization Problems

There has been some work on optimizing max algorithm performance in crowdsourcing environments. For example, [18] discusses optimization strategies when there are constraints on the cost and the execution time (latency) and the maximization objective is the output quality. The framework in [18] allows different numbers of questions per comparison to be posed at different steps of a tournament algorithm. Our framework allows more flexibility, since the number of questions per comparison is dynamically determined depending on the input set S and parameters r, R and δ. We exploit the possible benefits of this flexibility in terms of the following quantities:

$\mathrm{Latency}(A, \mathcal{E})$**, the Latency:** The number of time units required for a tournament max algorithm A to complete (and return one item) for input $\mathcal{E}$.

$\mathrm{Cost}(A, \mathcal{E})$**, the (Monetary) Cost:** The total amount of money required for A to complete for input $\mathcal{E}$.

Quality$(A, \mathcal{E})$**, the Quality of the Result:** A metric that captures the effectiveness of algorithm A in returning the actual maximum item from input $\mathcal{E}$ (or an item close to the maximum). We use the MRR metric defined in Section 3.2.

Formally, we are solving the following problems

Problem 1 (Max Algorithm: Cost)

$$\begin{aligned} \underset{A=\mathcal{A}_T(\{r_i\},\{R_i\},\{\delta_i\})}{\text{minimize}} \quad & \mathbb{E}[\text{Cost}(A,\mathcal{E})] \\ \text{subject to} \quad & \mathbb{E}[\text{Latency}(A,\mathcal{E})] \leq L \\ & \text{Quality}(A,\mathcal{E}) \geq Q \end{aligned} \tag{3}$$

and

Problem 2 (Max Algorithm: Latency)

$$\begin{aligned} \underset{A=\mathcal{A}_T(\{r_i\},\{R_i\},\{\delta_i\})}{\text{minimize}} \quad & \mathbb{E}[\text{Latency}(A,\mathcal{E})] \\ \text{subject to} \quad & \mathbb{E}[\text{Cost}(A,\mathcal{E})] \leq C \\ & \text{Quality}(A,\mathcal{E}) \geq Q \end{aligned} \tag{4}$$

Solving these problems allows us to minimize the cost of running a tournament algorithm on some input, while maintaining some constraints on the quality and the latency (Problem 1) or to minimize the latency of running a tournament algorithm on some input, while maintaining constraints on the quality and the cost (Problem 2). The strategies we propose in Section 4 can be used to solve both problems.

We optimize for the average case and thus use expectations for the metrics of interest (except for Quality$(\cdot)$), but of course many different formulations of the problems could be used. Also, different metrics can be used for Quality$(\cdot)$, and we describe some in Section 3.2.

3.2 Quality Evaluation Metrics

We define the quality metric we use for our evaluation in the context of simulations. We also define one more class of metrics that can be used in practice. Before we define the metrics of interest, we define the rank of an item e with respect to a set of items S.

Definition 1 (Item's Rank) *The rank of item e with respect to set S, denoted as* rank(e, S)*, is defined as one plus the number of items in S that are greater than e.*

For example, the rank of the max item in $\mathcal{E}$ with respect to $\mathcal{E}$ is $1 + 0 = 1$.

We simulate the application $A(\mathcal{E})$ (tournament algorithm A operates on input $\mathcal{E}$) E times. At the i^{th} simulation we obtain an item $e \in \mathcal{E}$ with rank with respect to $\mathcal{E}$ rank$_i$ and define Mean Reciprocal Rank (MRR) as follows.

Definition 2 (Mean Reciprocal Rank) *The average of the reciprocal ranks of the simulations, or simply the value* $\frac{1}{E}\sum_{i=1}^{E}\frac{1}{\text{rank}_i}$.

This metric yields values in $[0, 1]$: Values close to 1 indicate that A returns items that are close to the maximum item, while values close to 0 indicate that A returns items that are not close to the maximum item.

Our evaluation described in Section 6 uses the MRR metric. Another metric that could be used is the top-k metric (the fraction of the E simulations for which rank$_i \leq k$). We have experimented with this metric also and obtained very similar results to the ones we report for MRR.

4 Optimization Algorithm

This section describes an algorithm for selecting parameters ($\{r_i\}$, $\{R_i\}$, and $\{\delta_i\}$) solving the optimization Problems 1 and 2.

Section 6.1 experimentally explores the effect that tournament parameters have on cost, quality, and latency. One of the main observations we will see is that it is beneficial for latency (and cost) to have r_i be equal to δ_i. Thus, our proposed optimization algorithm forces $r_i = \delta_i$ for all steps i of the tournament.

Algorithm 3: Simulated annealing algorithm

```
   Input  : $\{r_i^0\}$, $\{R_i^0\}$, $\{\delta_i^0\}$, $\mathcal{E}$, $k_{\max}$
   Output: $\{\hat{r}_i\}$, $\{\hat{R}_i\}$, $\{\hat{\delta}_i\}$
1  $r_i \leftarrow r_i^0$, $R_i \leftarrow R_i^0$, $\delta_i \leftarrow \delta_i^0$;
2  Questions $\leftarrow$ average questions when simulating $\mathcal{A}_{\mathrm{T}}(\{r_i\},\{R_i\},\{\delta_i\})$ on $\mathcal{E}$;
3  QuestionsBest $\leftarrow$ Questions;
4  $Q \leftarrow$ quality when simulating $\mathcal{A}_{\mathrm{T}}(\{r_i\},\{R_i\},\{\delta_i\})$ on $\mathcal{E}$;
5  $L \leftarrow$ average latency when simulating $\mathcal{A}_{\mathrm{T}}(\{r_i\},\{R_i\},\{\delta_i\})$ on $\mathcal{E}$;
6  $k \leftarrow 0$;
7  while $k < k_{\max}$ do
8      $T \leftarrow$ temperature($\frac{k}{k_{\max}}$);
9      $\delta_i^+ \leftarrow \delta_i$, $r_i^+ \leftarrow r_i$, $R_i^+ \leftarrow R_i$;
10     for (randomly selected) step $l$ add some noise to $\delta_l^+$ and $R_l^+$ (between +2 and -2) such that
       problem constraints are satisfied and estimate (via simulations) Questions$^+$;
11     if Pr $\left[\text{Questions}, \text{Questions}^+, T\right] \geq$ random() then
12         $r_i \leftarrow r_i^+$, $R_i \leftarrow R_i^+$, $\delta_i \leftarrow \delta_i^+$;
13     if Questions < QuestionsBest then
14         QuestionsBest $\leftarrow$ Questions;
15         $\hat{r}_i = r_i^+$, $\hat{R}_i = R_i^+$, $\hat{\delta}_i = \delta_i^+$;
16     $k \leftarrow k + 1$;
17 return $\{\hat{r}_i\}$, $\{\hat{R}_i\}$, $\{\hat{\delta}_i\}$
```

Our algorithm, presented in Algorithm 3, is a version of simulated annealing [10] that solves Problem 1, but very similar techniques can be used for Problem 2. The algorithm takes as input sequences $\{r_i^0\}$, $\{R_i^0\}$, $\{\delta_i^0\}$ that are a starting point for the parameter space search, the set of items on which the tournament algorithm is applied ($\mathcal{E}$), and the maximum number of iterations the algorithm will perform to find a good solution ($k_{\max}$). The algorithm's output is sequences $\{\hat{r}_i\}$, $\{\hat{R}_i\}$, and $\{\hat{\delta}_i\}$ that attempt to minimize the number of total questions asked, while giving at least the same quality as the initial sequences and at most the latency of the initial sequences.

The algorithm first initializes sequences $\{r_i\}$, $\{R_i\}$, and $\{\delta_i\}$ (line 1). Then it estimates (via simulations) the number of questions, the quality Q, and the latency L of the given sequences, and also keeps the best number of questions seen so far in variable QuestionsBest (lines 2–5). Counter variable k is initialized (line 6) and the iterations' loop begins (line 7). First, a temperature T is calculated based on how far in the process we are (line 8). High temperature values indicate that any state can be accepted as the new state, while low temperatures indicate that the new state has to be better than the previous one to be accepted. We use a linear function temperature that goes from 300 when $k = 0$ to 0 when $k = k_{\max}$. We then initialize the new sequences $\{R_i^+\}$, and $\{\delta_i^+\}$ to the previous sequences (line 9), pick a random step l and add some noise to R_l^+ and δ_l^+ such that the latency and quality constraints are satisfied, and find the number of questions $\mathcal{A}_{\mathrm{T}}(\{\delta_i^+\},\{R_i^+\},\{\delta_i^+\})$ requires to run on $\mathcal{E}$ Questions$^+$ (line 10). (We add the same noise to r_l and δ_l since $r_i = \delta_i$ for all i.) We decide to "keep" the new sequences if the transition probability Pr $\left[\text{Questions}, \text{Questions}^+, T\right]$ is greater than a random number between 0 and 1 (lines 11–12).

(The transition probability is calculated in a standard way: If $\text{Questions}^+ < \text{Questions}$ then it is 1, otherwise it is $\exp\left(\frac{\text{Questions}-\text{Questions}^+}{T}\right)$.) If the objective Questions is better than the current best QuestionsBest, the sequences and the QuestionsBest variable are updated (lines 13–15). Then k is increased by 1 (line 16) and the loop is repeated. Finally, the best sequences $\{\hat{r}_i\}$, $\{\hat{R}_i\}$, and $\{\hat{\delta}_i\}$ are returned.

The algorithm essentially attempts to refine an existing solution $\{r_i\}$, $\{R_i\}$, $\{\delta_i\}$ and to find one that gives similar (or higher) quality with lower latency and cost. In Section 6.2 we see how well the algorithm performs in practice.

5 Single Comparison Evaluation

To understand how parameter values should be selected (both for single comparisons and for whole tournament algorithms), we explore the effects that different parameter values have on cost, quality, and latency. For single pairwise comparisons, we are able to fully analyze cost, quality, and latency (Section 5.2) and we present the tradeoffs using that analysis (Section 5.3). For tournaments the analysis is significantly harder and we use simulations to understand the tradeoffs (Section 6.1). Before we delve into our examination, we first show (Section 5.1) how our error model's parameters can be selected using crowdsourcing data.

5.1 Thurstonian Model Fitting

This section describes how the Thurstonian model parameters were selected for our experiments: We matched real worker behaviour to the model. We submitted comparisons of various difficulty values to Amazon's Mechanical Turk [3] and collected worker responses. Then, we selected the Thurstonian model parameters that minimized the difference between the estimated incorrect answer probability and the observed worker error.

Dataset. For our crowdsourcing experiments we use a dataset $\mathcal{E}$ for which each item $e \in \mathcal{E}$ has a known quality value. Inspired by the jelly-beans-in-a-jar experiment [15], we created a dataset of images each containing a number $d(e)$ of colored dots, with $d(e) \in \{1, 2, \ldots, 1{,}000\}$ and define the quality value of image e as $q(e) = d(e)$. Given a set of items $S \subseteq \mathcal{E}$, a comparison microtask asks a worker to determine the image with the maximum number of dots in it. A sample of the dataset can be seen in [17].

Setting. The goal is to select the Thurstonian model parameters ($\sigma(e_i)$) that produce the same errors as real workers for the same comparisons. Using our dots dataset, we chose various comparison difficulties D (with values ranging from 0.5 to 0.97, see Equation (1)) and for each difficulty $d \in D$ we posted thousands of comparisons to Amazon's Mechanical Turk — details on these microtasks can be found in [17]. We obtained the worker responses and for each d and calculated the fraction of incorrect worker responses $F[\text{error}; d]$.

As described in Section 2.1.1, the Thurstonian worker samples values $\hat{e}_i$ for each item e_i in the comparison set S and answers the item with the highest sampled value $\hat{e}_i$. Values $\hat{e}_i$ are sampled from the normal distribution $\mathcal{N}(q(e_i), \sigma(e_i)^2)$ where $q(e_i)$ is the actual quality value of item e_i and $\sigma(e_i)$ are the standard deviation parameters we want to select. We consider standard deviation values of the form $\sigma(e_i) = \alpha_0 + \alpha_1 \cdot q(e_i) + \alpha_2 \cdot q(e_i)^2$ and denote the corresponding Thurstonian model by $\mathcal{T}(\vec{\alpha})$. We then use simulations to estimate the probability that the Thurstonian worker $\mathcal{T}(\vec{\alpha})$ will give an incorrect answer for comparisons of difficulty d and denote this probability by $\Pr[\text{error}; \mathcal{T}(\vec{\alpha}), d]$.

Our goal is to select parameters $\vec{\alpha}$ such that $F[\text{error}; d]$ and $\Pr[\text{error}; \mathcal{T}(\vec{\alpha}), d]$ are close to each other. Formally, we want to solve the problem

Problem 3 (Thurstonian Model Fitting)

$$\underset{\vec{\alpha}}{\text{minimize}} \max_{d \in D} \{ |F[\text{error}; d] - \Pr[\text{error}; \mathcal{T}(\vec{\alpha}), d]| \} \tag{5}$$

That is, we want to minimize the maximum difference between the experimentally observed worker error and the simulated Thurstonian worker error across different values of difficulty $d \in D$.

Fitting Algorithm. To solve Problem 3 we use a heuristic called differential evolution [13]. Differential evolution iteratively tries to improve a candidate solution with regard to a given objective (in our case maximum error for a difficulty value). Although this technique does not guarantee an optimal solution, the results we obtain are both intuitive and meaningful. We present in Algorithm 4 the pseudocode of the optimization process we followed.

Algorithm 4: Differential Evolution(F, CR, NP)

```
   // $C$ is a set of candidates
1  $C \leftarrow$ select $NP$ random candidates (three dimensional vectors $\vec{\alpha}$);
2  while objective improves by some candidate do
3      foreach $\vec{a} \in C$ do
4          pick three other candidates $\vec{x}, \vec{y}, \vec{z} \in C$;
5          pick $R \in \{0, 1, 2\}$;
6          foreach $i \in \{0, 1, 2\}$ do
7              $r \leftarrow U(0, 1)$;
8              if $r < CR$ or $i = R$ then
9                  $w_i \leftarrow \max\{x_i + F \cdot (y_i - z_i), 0\}$;
10             else
11                 $w_i \leftarrow \alpha_i$;
12         if $\vec{w}$ gives better objective value than $\vec{\alpha}$ then
13             $C \leftarrow (C \setminus \{\vec{\alpha}\}) \cup \{\vec{w}\}$;
14 return $\arg\min_{\vec{\alpha} \in C}$ objective
```

Results. We run Algorithm 4 with various parameters F (called differential weight), CR (crossover probability), and NP (population size) and always obtained very similar results. (We have also considered other objective functions, e.g., minimizing the average error observed across different difficulty values d and we again obtained similar solutions.) The solution we obtained was $\vec{\alpha} \cong (0.816, 0.101, 1.7 \cdot 10^{-5})$ (for which the objective was $\sim$0.04), which suggests a linear relation between σ and the quality value $q(e_i)$ of an item. This fact suggests that the perceived quality value from a worker is very accurate for low quality values and worker errors increase as the quality values increase. Figure 1 presents the PDFs for 3 items e_1, e_2, and e_3 with quality values 800, 550, and 250 respectively after the model parameters were fitted. We observe that σ is larger towards the right of the graph (i.e., for larger quality values). Also, there is significant overlap between the PDFs and thus the worker may make mistakes in his comparisons.

5.2 Single Comparison Analysis

This section analyzes the number of questions X posed when a comparison $\text{Comp}(S, r, R, \delta)$ is executed and the probability that the response given by the comparison is the correct one. We focus on pairwise comparisons, i.e., comparisons with $|S| = 2$, and in the following we assume $S = \{e_1, e_2\}$ with $e_2 < e_1$.

We denote by E_k the difference of answers e_1 and e_2 have received when k answers have been collected in total; the possible outcomes of E_k are $\{-k, -k+1, \ldots, k\}$. (For example, $E_k = k$ when all k workers voted item e_1.) The value of E_{k+1} depends only on the value of E_k and the additional vote received, and thus, the Markov property holds. Since the Markov property holds [9], we view the difference of votes between e_1 and e_2 as a Markov chain [9] with set of states $\{-R, -R+1, \ldots, R\}$ as shown in Figure 3: For each k, E_k's possible outcomes belong in this set of states.

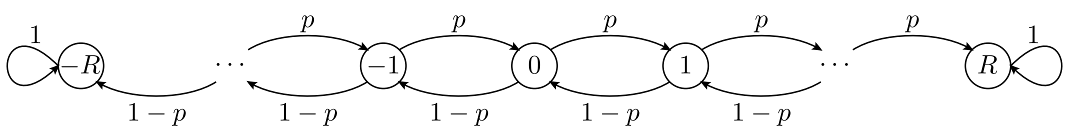


Figure 3: The states of the Markov chain E_k, $k \in \{0, 1, \ldots\}$.

We denote by p the probability that the (Thurstonian) worker answers e_1. Then, $1-p$ is the probability that the worker answers e_2. The transition probabilities $p_{i,j} = \Pr[E_k = j \mid E_{k-1} = i]$, i.e., the probabilities of moving from state i to state j with one additional response, are in our case

$$\begin{aligned}
p_{i,i+1} &= p, & \forall i \in \{-R+1, \ldots, R-1\} \\
p_{i,i-1} &= 1-p, & \forall i \in \{-R+1, \ldots, R-1\} \\
p_{i,i} &= 0, & \forall i \in \{-R+1, \ldots, R-1\} \\
p_{R,R} &= 1 & \\
p_{-R,-R} &= 1 & \\
p_{i,j} &= 0, & \forall j : |j-i| \geq 2
\end{aligned} \tag{6}$$

Values $p_{i,j}$ form the transition matrix $P = (p_{i,j})$. Using P, we can express the probability $p^k_{i,j} = \Pr[E_{n+k} = j \mid E_n = i]$ of moving from state i to state j with k extra questions as the (i,j) element of the matrix $P^k = \underbrace{P \cdot P \cdot \ldots \cdot P}_{k \text{ times}}$.

We denote by $\pi_k(i)$ the probability that the difference of votes is equal to i after k answers have been provided. Before we receive any answers, E_0 is 0, thus $\pi_0(0) = 1$ and $\pi_0(i) = 0$ for all $i \neq 0$. Probability $\pi_k(i)$ is equal to $p^k_{0,i}$ and can thus be computed as

$$\pi_k(i) = p^k_{0,i} = P^k(0,i) = \underbrace{(P \cdot P \cdot \ldots \cdot P)}_{k \text{ times}}(0,i). \tag{7}$$

5.2.1 Probability p

We first describe how probability p can be computed. For simplicity, we denote by $q_i = q(e_i)$ the quality value of e_i and $\sigma_i = \sigma(e_i)$ the standard deviation of the Thurstonian model for item e_i. For the Thurstonian model, p is the probability that the sample $\hat{e}_1$ from $\mathcal{N}(q_1, \sigma_1^2)$ is greater than the sample $\hat{e}_2$ from $\mathcal{N}(q_2, \sigma_2^2)$. Equivalently, p is the probability that $\hat{e}_1 - \hat{e}_2 \geq 0$. Since $\hat{e}_1 - \hat{e}_2$ follows $\mathcal{N}(q_1 - q_2, \sigma_1^2 + \sigma_2^2)$, we have

$$p = \int_0^\infty \frac{1}{\sqrt{2\pi(\sigma_1^2 + \sigma_2^2)}} e^{-\frac{[u-(q_1-q_2)]^2}{2(\sigma_1^2+\sigma_2^2)}} \, \mathrm{d}u$$

or using the erf function

$$p = \frac{1}{2} - \frac{1}{2} \cdot \operatorname{erf}\left(-\frac{q_1 - q_2}{\sqrt{2(\sigma_1^2 + \sigma_2^2)}} \right). \tag{8}$$

5.2.2 Cost Analysis

We now describe the distribution of required questions to complete comparison $\mathrm{Comp}(S, r, R, \delta)$. We denote the probability that (exactly) k questions are required by $\Pr[X = k; S, r, R, \delta]$ or $\Pr[X = k]$ if the context is clear. We denote states $\{-R, -R+1, \ldots, -\delta, \delta, \delta+1, \ldots, R\}$ by $\mathcal{S}_e$ and call them *ending states* (if $E_k \in \mathcal{S}_e$, the comparison ends); we denote states $\{-\delta+1, -\delta+2, \ldots, \delta-1\}$ by $\mathcal{S}_n$ and call then *non-ending states.*

In the following we examine the case where $r < R$.[1] To complete the comparison in exactly r questions, the absolute value of E_r must be greater than δ, i.e., $|E_r| \geq \delta$ or equivalently $E_r \in \mathcal{S}_e$. Thus,

$$\Pr[X = r] = \Pr[E_r \in \mathcal{S}_e] = \sum_{s \in \mathcal{S}_e} \Pr[E_r = s] = \sum_{s \in \mathcal{S}_e} \pi_r(s). \tag{9}$$

To complete the comparison in exactly k questions ($k \in \{r+1, r+2, \ldots, R-1\}$), we must have $E_k \in \mathcal{S}_e$, and $E_{k-1}, E_{k-2}, \ldots, E_r \in \mathcal{S}_n$. To simplify the analysis, we define

$$\zeta_k(x) = \Pr[E_k = x \mid E_r, E_{r+1}, \ldots, E_k \in \mathcal{S}_n].$$

For the special case $k = r$, we have $\zeta_r(x) = 0$ for $x \in \mathcal{S}_e$. For $x \in \mathcal{S}_n$ we have

$$\begin{aligned}\zeta_r(x) &= \Pr[E_r = x \mid E_r \in \mathcal{S}_n] = \\ &= \frac{\Pr[E_r = x \wedge E_r \in \mathcal{S}_n]}{\Pr[E_r \in \mathcal{S}_n]} \\ &= \frac{\Pr[E_r = x]}{\Pr[E_r \in \mathcal{S}_n]} \\ &= \frac{\pi_r(x)}{\sum_{x' \in \mathcal{S}_n} \Pr[E_r = x']} \\ &= \frac{\pi_r(x)}{\sum_{x' \in \mathcal{S}_n} \pi_r(x')}\end{aligned} \tag{10}$$

For $k > r$ we again have $\zeta_k(x) = 0$ for $x \in \mathcal{S}_e$. For $x \in \mathcal{S}_n$ we have

$$\begin{aligned}\zeta_k(x) &= \Pr[E_k = x \mid E_r, E_{r+1}, \ldots, E_k \in \mathcal{S}_n] = \\ &= \frac{\Pr[E_k = x \wedge E_k \in \mathcal{S}_n \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n]}{\Pr[E_k \in \mathcal{S}_n \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n]} \\ &= \frac{\Pr[E_k = x \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n]}{\sum_{x' \in \mathcal{S}_n} \Pr[E_k = x' \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n]}\end{aligned}$$

Using the law of total probability

$$\begin{aligned}&\Pr[E_k = x \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n] = \\ &= \sum_{z \in \mathcal{S}_n} \Pr[E_k = x \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n \wedge E_{k-1} = z] \cdot \\ &\quad \cdot \Pr[E_{k-1} = z \mid E_r, E_{r+1}, \ldots, E_{k-1} \in \mathcal{S}_n] = \\ &= \sum_{z \in \mathcal{S}_n} p_{z,x} \cdot \zeta_{k-1}(z)\end{aligned} \tag{11}$$

we have

$$\zeta_k(x) = \frac{\sum_{z \in \mathcal{S}_n} p_{z,x} \cdot \zeta_{k-1}(z)}{\sum_{x' \in \mathcal{S}_n} \sum_{z' \in \mathcal{S}_n} p_{z',x'} \zeta_{k-1}(z)}, \; x \in \mathcal{S}_n. \tag{12}$$

We can now determine $\Pr[X = k]$:

$$\begin{aligned}\Pr[X = k] &= \Pr[E_k \in \mathcal{S}_e \wedge E_{k-1}, \ldots, E_r \in \mathcal{S}_n] = \\ &= \Pr[E_k \in \mathcal{S}_e \mid E_{k-1}, \ldots, E_r \in \mathcal{S}_n] \cdot \\ &\quad \cdot \Pr[E_{k-1}, \ldots, E_r \in \mathcal{S}_n].\end{aligned} \tag{13}$$

[1]If $r = R$, then the comparison will need exactly $r = R$ questions answered and thus $\Pr[X = r] = \Pr[X = R] = 1$.

Now,

$$\Pr\left[E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=1-\sum_{n=r}^{k-1}\Pr\left[X=n\right] \tag{14}$$

and

$$\begin{aligned}&\Pr\left[E_k\in\mathcal{S}_e\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad=\sum_{y\in\mathcal{S}_e}\Pr\left[E_k=y\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad\overset{(11)}{=}\sum_{y\in\mathcal{S}_e}\sum_{x\in\mathcal{S}_n}p_{x,y}\cdot\zeta_{k-1}(x)\end{aligned}$$

From Equation (6) we can see that the only non-zero transition probabilities $p_{x,y}$ with $x\in\mathcal{S}_n$ and $y\in\mathcal{S}_e$ are $p_{\delta-1,\delta}$ and $p_{-\delta+1,-\delta}$. Thus,

$$\begin{aligned}&\Pr\left[E_k\in\mathcal{S}_e\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad=p_{-\delta+1,-\delta}\cdot\zeta_{k-1}(-\delta+1)+p_{\delta-1,\delta}\cdot\zeta_{k-1}(\delta-1)=\\&\quad=(1-p)\cdot\zeta_{k-1}(-\delta+1)+p\cdot\zeta_{k-1}(\delta-1)\end{aligned} \tag{15}$$

From Equations (10), (12), (13), (14), and (15) we can determine $\Pr\left[X=k\right]$ for any $k\in\{r+1,r+2,\ldots,R-1\}$.

Finally, $\Pr\left[X=R\right]=1-\sum_{n=r}^{R-1}\Pr\left[X=n\right]$, since in any case after R questions the comparison is completed.

5.2.3 Quality Analysis

We now determine the probability that Comp(S,r,R,δ) returns the correct result denoted by $\Pr\left[\text{returns max}; S,r,R,\delta\right]$ or simply $\Pr\left[\text{returns max}\right]$ if the context is clear. We have from the law of total probability

$$\Pr\left[\text{returns max}\right]=\sum_{k=r}^{R}\Pr\left[X=k\right]\cdot\Pr\left[\text{return max}\mid X=k\right] \tag{16}$$

The conditional probability can be calculated as follows

$$\begin{aligned}&\Pr\left[\text{return max}\mid X=k\right]=\\&\quad=\Pr\left[E_k\in\{\delta,\ldots,R\}\mid E_k\in\mathcal{S}_e,E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad=\frac{\Pr\left[E_k\in\{\delta,\ldots,R\},E_k\in\mathcal{S}_e\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]}{\Pr\left[E_k\in\mathcal{S}_e\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]}\\&\quad\overset{(15)}{=}\frac{\Pr\left[E_k\in\{\delta,\ldots,R\},E_k\in\mathcal{S}_e\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]}{(1-p)\cdot\zeta_{k-1}(-\delta+1)+p\cdot\zeta_{k-1}(\delta-1)}\end{aligned} \tag{17}$$

and finally

$$\begin{aligned}&\Pr\left[E_k\in\{\delta,\ldots,R\},E_k\in\mathcal{S}_e\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad=\Pr\left[E_k\in\{\delta,\ldots,R\}\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad=\sum_{y=\delta}^{R}\Pr\left[E_k=y\mid E_{k-1},\ldots,E_r\in\mathcal{S}_n\right]=\\&\quad=\sum_{y=\delta}^{R}\sum_{x\in\mathcal{S}_n}p_{x,y}\cdot\zeta_{k-1}(x)=\\&\quad=p_{\delta-1,\delta}\cdot\zeta_{k-1}(\delta-1)=\\&\quad=p\cdot\zeta_{k-1}(\delta-1)\end{aligned} \tag{18}$$

with a similar analysis to that of Equation (15).

The conditional probability of Equation (16) for the boundary cases ($k = r$ and $k = R$) can be calculated as:

$$\Pr\left[\text{return max} \mid X = r\right] = \frac{\sum_{y=\delta}^{R} \pi_r(y)}{\sum_{y' \in \mathcal{S}_e} \pi_r(y')} \tag{19}$$

and

$$\Pr\left[\text{return max} \mid X = R\right] = \\ = \sum_{y=1}^{R} \sum_{x=-R}^{R} \zeta_{R-1}(x) \cdot p_{x,y} + \frac{1}{2} \sum_{x=-R}^{R} \zeta_{R-1}(x) \cdot p_{x,0} \tag{20}$$

respectively.

From Equations (16), (17), (18), (19), and (20) we can calculate $\Pr\left[\text{return max}; S, r, R, \delta\right]$.

5.3 Single Comparison Tradeoffs

This section describes the parameter effects on quality and cost of $\text{Comp}(S, r, R, \delta)$. For this study we computed the results using the equations shown in the previous three Sections 5.2.1, 5.2.2, and 5.2.3. For the graphs shown in the following we use the parameter values shown in Table 1.

Parameter	**Default Value**	**Range Values**
$\text{diff}(S)$	0.95	0.70, 0.75, ..., 0.95
r	4	1, 2, ..., 10
R	10	1, 2, ..., 50
δ	4	1, 2, ..., 10

Table 1: Parameter values for comparisons tradeoff study.

Varying δ. We use the default values of Table 1 for parameters $\text{diff}(S)$ and R. We vary δ through its range values (1, 2, ..., 10) and chose $r = \delta$ for each δ value. We observe the values achieved for $\Pr\left[\text{return max}\right]$ and the distribution of X in Figure 4.

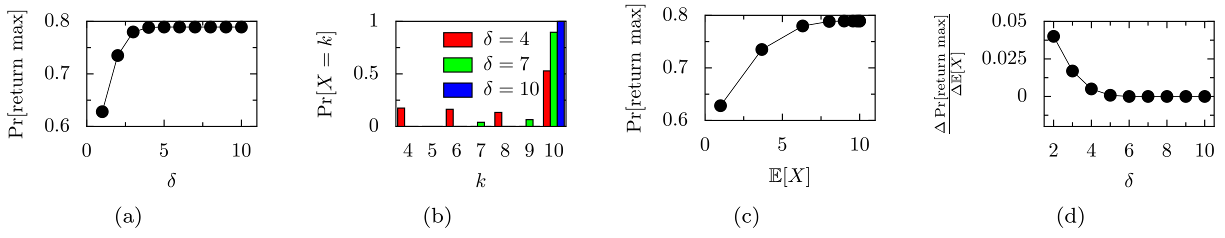


Figure 4: Exploring the effect of δ on $\Pr\left[\text{return max}\right]$ and $\Pr\left[X = k\right]$.

We observe in Figure 4(a) that the probability the max item from S is retrieved increases as δ increases, and it reaches a plateau after $\delta = 4$. Thus, forcing δ to be higher than 4 offers no clear benefit.

In Figure 4(b) we observe three different distributions for the number of questions X for various values of δ. For example, for $\delta = 4$, there are 4 bars at $k = 4, 6, 8, 10$, meaning that there is a (non-zero) probability that 4, 6, 8, or 10 questions will be asked. (Due to the choice of parameters, the number of questions asked cannot be 5, 7, or 9.) For $\delta = 7$, there are three bars at $k = 7, 9, 10$, meaning that either 7, or 9, or 10 questions will be asked. Finally, for $\delta = 10$, there is only one bar at $k = 10$ (since $R = 10$, exactly 10 answers will be asked). Thus, for δ values lower than R we may observe significantly lower values for the expected cost.

Figure 4(c) shows the tradeoff between the expected X (x-axis) and Pr [return max] (y-axis) as δ varies. The bottom left part of the graph's line refers to $\delta = 1$ and δ values increase towards the top right of the line up to $\delta = 10$. We observe that Pr [return max] increases between $\delta = 1$ and $\delta = 4$ with an increase in the expected number of questions X also. After $\delta = 4$, Pr [return max] stays practically constant, while the expected cost keeps increasing.

Figure 4(d) shows δ in the x-axis. The y-axis presents the fraction of the difference of Pr [return max] for δ and $\delta - 1$ over the difference of the expected questions X for values δ and $\delta - 1$; the fraction shows how much the probability of retrieving the max item increases over the increase in the cost/questions when we change δ by 1. Low values in the y-axis correspond to low increases in quality or/and large increases in cost, while high values correspond to large increases in quality or/and small increases in cost. We see that the fraction values decrease as δ increases and after $\delta = 4$ these values are less than 10^{-14}. The requester can use graphs such as the one we have provided to chose where to allocate his resources (in this case money). For instance, he may expect a particular return for each monetary unit he spends. Using Figure 4(d) he may stop increasing δ above 4, since he could "invest" his money to other microtasks with higher expected returns for the same extra amount of money.

Varying R. After observing that $\delta = 4$ is a good compromise, the requester may explore different values for R (in particular, the range values in Table 1, while the rest parameters have their default values). Figure 5 presents how Pr [return max] and $\mathbb{E}[X]$ are affected by the changes in R.

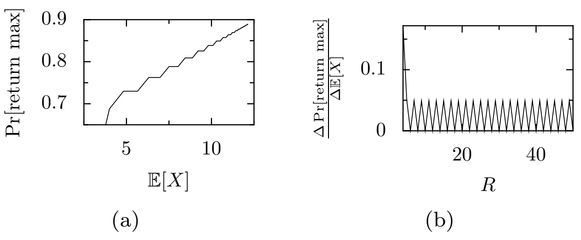


Figure 5: Exploring the effect of R on Pr [return max] and $\mathbb{E}[X]$.

Figure 5(a) presents Pr [return max] (y-axis) versus the expected X (x-axis) as the value of R changes: The bottom left point of the graph's line corresponds to $R = 4$ and the top right point of the line corresponds to $R = 50$. We observe that Pr [return max] increases (near-)linearly as $\mathbb{E}[X]$ increases. Thus, the requester can easily control the probability that the max item of S is returned by adjusting his monetary cost accordingly.

Figure 5(b) shows R in the x-axis. The y-axis presents the fraction of the difference of Pr [return max] for R and $R - 1$ over the difference of the expected cost for values R and $R - 1$; this fraction shows how much the probability of retrieving the max item increases when one increases the cost/questions (by increasing R by 1). Low values in the y-axis correspond to low increases in quality or/and large increases in cost, while high values correspond to large increases in quality or/and small increases in cost. We see that for $R = 4$ and $R = 5$ the fraction (y-axis) has a significant value. After that, the fraction oscillates between 0 and a value close to 0.05, which suggests that Pr [return max] is only increased when we increase an even R to (the odd) $R + 1$.

Varying diff(S). Until now we have assumed a fixed difficulty for S. What happens when we change diff(S) and explore Pr [return max] and $\mathbb{E}[X]$? We explore the tradeoffs when r, R, and δ take their default values (Table 1) and diff(S) takes values from the range values (shown in the same table).

Figure 6(a) presents Pr [return max] (y-axis) versus the expected cost (x-axis) as the value of diff(S) changes: The bottom right point of the graph's line corresponds to diff$(S) = 0.95$ and the top left point of the line corresponds to diff$(S) = 0.70$. We observe that when diff(S) increases from 0.70 to 0.75, the expected

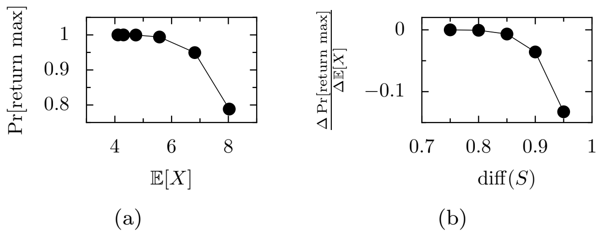


Figure 6: Exploring the effect of diff(S) on Pr [return max] and $\mathbb{E}[X]$.

cost increases slightly and the Pr [return max] decreases slightly. As diff(S) increases more and more, the changes to both cost and Pr [return max] become larger and larger. We explore the fraction of changes in Pr [return max] over the difference in expected costs as diff(S) increases in steps of 0.05 in Figure 6(b). We see that the fractions increase (in absolute values) more and more for higher diff(S). This means that as diff(S) increases, even though the cost increases significantly, the Pr [return max] decreases relatively more and thus the fraction of interest keeps decreasing.

6 Tournaments Evaluation

This section focuses on tournament algorithms and first (Section 6.1) explores the tradeoffs between the three quantities of interest:

- Cost: total number of questions asked,
- Latency: time required for the tournament to complete, and
- Quality: how accurate the tournament output is (using the MRR metric discussed in Section 3.2).

Then, we explore how effective the optimization algorithm is (Sections 6.2 and 6.3) and describe the optimization algorithm solution (Section 6.4). Since analysis is significantly harder than in the case of single comparisons, we use simulations to explore the parameter space.

6.1 Tournament Tradeoffs

We start with a simple tournament algorithm that attempts to recover the max item from a set $\mathcal{E}$ of $|\mathcal{E}| = 100$ items with quality values 10, 20, . . . , 1,000. To understand the Thurstonian model, we note that after fitting its parameters (Section 5.1), the probability p that the max item from S is answered by a worker belongs to the range $[\frac{1}{|S|}, 1]$. It takes values close to $\frac{1}{|S|}$ when the items in S have very similar values to each other; it takes values close to 1 when the max item in S has a much higher value than the rest items in S. For example, we find that

- When comparing $S = \{e_1, e_2\}$ with $q(e_1) = 20$ and $q(e_2) = 10$, $p = 0.998$ (close to the maximum possible value),
- When comparing $S = \{e_1, e_2\}$ with $q(e_1) = 1{,}000$ and $q(e_2) = 999$, $p = 0.502$ (close to the worse possible value)
- When comparison $S = \{e_1, e_2\}$ with $q(e_1) = 1{,}000$ and $q(e_2) = 990$, $p = 0.523$ (thus, the difference of the quality values is not a good indicator for p, but their relative values are).

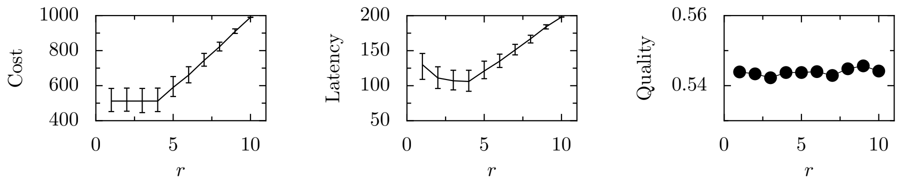
For simplicity, we used constant sequences $\{r_i\}$, $\{\delta_i\}$, and $\{R_i\}$, i.e., $r_i = r$, $\delta_i = \delta$, and $R_i = R$ for all steps i. To simulate the tournament algorithms, we execute the algorithm and simulate the outcome and the number of questions of each Comp(S, r, R, δ) and the queue for the marketplace execution schedule. For the graphs of the following sections we use the parameter values shown in Table 2.

Parameter	Default Value	Range Values
diff($\mathcal{E}$)	0.990	0.990, 0.991, ..., 0.999
R	10	1, 3, 5, 7, 10, 15, 20, 30, 50
r	δ	1, 2, ..., 10
δ	4	1, 2, ..., 10
γ	5	5, 10, 20, 50, 100, 200

Table 2: Parameter values for tournament tradeoff study.

6.1.1 Varying r

For this experiment we varied r across the range values in Table 2 and kept the rest parameters fixed at their default values.



(a) (b) (c)

Figure 7: Cost/latency/quality tradeoff when changing r for the tournament algorithms.

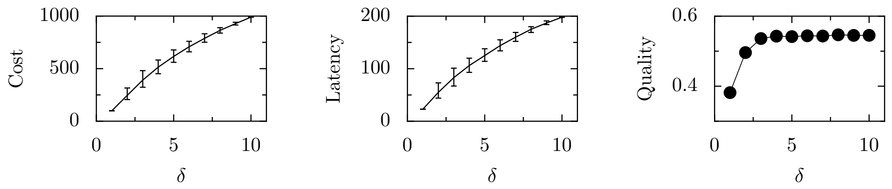
Figure 7 presents how each of the quantities of interest is affected. For each r we run 100,000 simulations and collected the observed cost, latency, and ranks of the returned items with respect to $\mathcal{E}$. We summarized the observed values in Figure 7 using errorbar plots. Each errorbar plot shows the range and the median of the y-axis values observed for a particular r. The top bar represents the maximum value observed, the bottom bar is the lowest value. The point between the two bars represents the median value observed. Figure 7 presents graphs for cost (a), latency (b), and MRR (c).

We observe that r should be selected equal to δ since

- Latency decreases as r increases towards δ and then it increases,
- Cost is the same across all values r in set $\{1, 2, \ldots, \delta\}$, and
- Quality is not affected by r.

6.1.2 Varying δ

For this experiment we used the default values in Table 2 for parameters R, r, γ, and diff($\mathcal{E}$) and range the values of δ. We selected $r = \delta$ such that the minimum possible latency.



(a) (b) (c)

Figure 8: Cost/latency/quality tradeoff when changing δ for the tournament algorithms.

We observe that

- Both cost (number of questions) and latency increase near-linearly as δ increases.
- MRR reaches a plateau at $\delta = 4$. Thus, we can achieve the (practically) the same quality (MRR) for $\delta = 4$ as for any other value $\delta > 4$, with significant savings both in cost and in latency. In particular, the cost when $\delta = 10$, i.e., when we ask all $R = 10$ questions for each comparison, is 990, while when $\delta = 4$ the median cost is $\sim$330. Thus, we can reduce cost by $\sim$70% for the same quality by changing the number of questions asked for each comparison.

6.1.3 Varying R

For this experiment we used the default values in Table 2 for parameters r, δ, γ, and diff($\mathcal{E}$) and range the values of R.

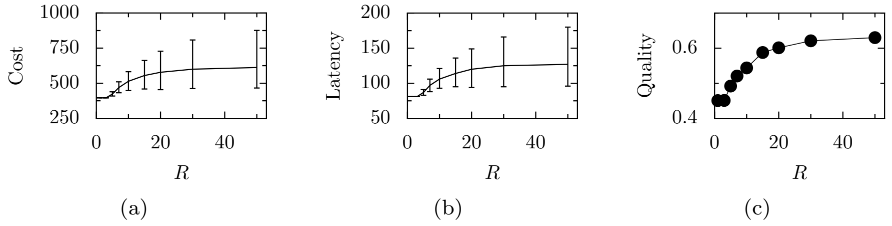


Figure 9: Cost/latency/quality tradeoff when changing R for the tournament algorithms.

Figure 9 presents how each of the quantities of interest is affected. We observe that

- Latency and cost increase as R increases, but most importantly, the variance of these quantities increases. Thus, the higher R is, the more uncertain cost and latency are.
- MRR increases as R increases and for the values of R we present there is no significant topping off observed.

6.1.4 Varying γ

For this experiment we varied γ using the range values in Table 2 and kept the rest parameters at their default values.

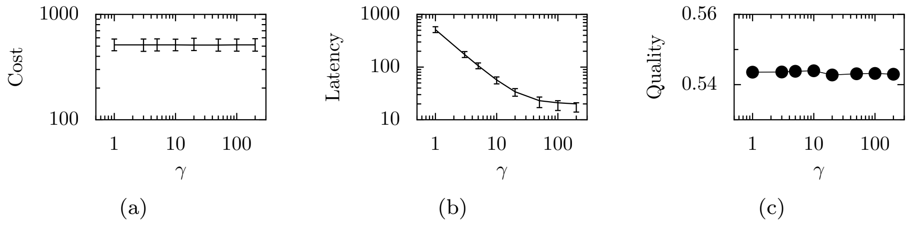


Figure 10: Cost/latency/quality tradeoff when changing γ for the tournament algorithms.

Figure 10 presents how each of the quantities of interest is affected. We observe that

- Only latency is affected: As γ increases, latency exponentially decreases at the beginning and then it reaches a plateau (i.e., when is higher than the maximum number of questions in the execution queue at all times).
- As expected, cost and MRR are not affected by changes in γ.

6.1.5 Varying Input Difficulty

For this experiment we varied the input "difficulty" as seen in the range values in Table 2 and kept the rest parameters to their default values. In particular, we chose $\mathcal{E}$ such that the max item has quality value 1,000 and the rest 99 items ($|\mathcal{E}| = 100$) were selected such that the quality value of the second one had a difference "distance" from 1,000, the third one $2 \cdot$ distance from 1,000, and so on. We varied distance from 1 to 10 with a step of 1. The higher distance is, the easier the task of retrieving the max from $\mathcal{E}$ is.

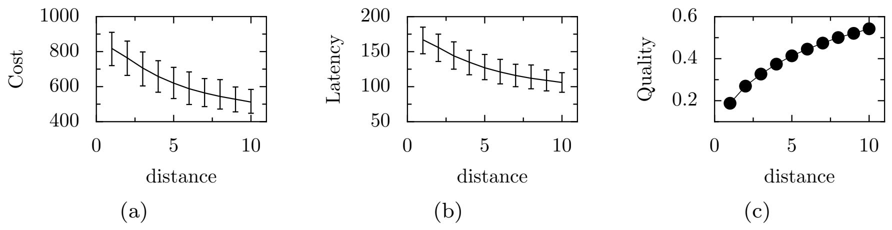


Figure 11: Cost/latency/quality tradeoff when changing distance for the tournament algorithms.

Figure 11 presents how each of the quantities of interest is affected as distance changes. We observe that

- Cost and latency decrease asymptotically as distance increases (and thus difficulty decreases). Thus, the easier the input, the faster and cheaper the tournament algorithm runs.
- Quality increases as distance increases (and thus difficulty decreases). Thus, we obtain better results when the input to the algorithm is easier to be treated by the workers.

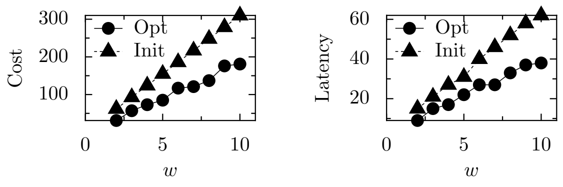
6.2 Algorithm Effectiveness

This section presents experiments that explore the usefulness of the simulated annealing algorithm (Algorithm 3). In marketplaces like Amazon's Mechanical Turk [3] the requester typically picks a constant number of worker that will perform each microtask, by setting one parameter to the marketplace interface. For example, the requester may pick to request $w = 5$ worker responses for each comparison of the tournament algorithm. In the following, we compare the case of having w responses per comparison to the case of optimizing the responses requested as described in the simulated annealing algorithm. We find that after the optimization we can achieve the same quality with many fewer total requested answers.

To understand the effect of the optimization algorithm, we chose as initial sequences for the optimization algorithm $r_i^0 = R_i^0 = \delta_i^0 = w$ with w constant across all steps i. For each w we run the optimization algorithm (both for cost and for latency) with initial sequence the one with $r_i^0 = R_i^0 = \delta_i^0 = w$. Then, we observe how the cost and the latency of the optimized tournament compares to the the cost and latency of the initial (constant) sequences for $w \in \{2, 3, \ldots, 10\}$. The input $\mathcal{E}$ contained 32 items with quality values 1,000, 990, 980, …, 690. We chose to perform $k_{\max} = 100$ iterations for the simulated annealing algorithm (we also tried higher values with practically no improvement in the optimization algorithm performance). We note that the tournament with constant sequences w and the optimized (over cost or latency) tournament give the same quality value by the design of the optimization algorithm.

The results are summarized in Figure 12: Figures 12(a) and (b) present the cost and the latency values respectively, both before and after the optimization. For both figures two lines are presented: (1) Init which represents the cost and latency values achieved by the initial sequences (the ones with w human responses per comparison at all steps) and (2) Opt represents the optimized values achieved by Algorithm 3 focused on cost or latency respectively. We note that for $w = 2$ (left part of the graphs) we achieve quality equal to 0.385 and for $w = 10$ (right part of the graphs) we achieved quality equal to 0.547; thus, w significantly affects quality.

We observe that



(a) Optimizing cost. (b) Optimizing latency.

Figure 12: Exploring simulated annealing algorithm effectiveness.

- *Opt achieves the same quality as the tournaments on the Init line, with significant savings on cost or latency.* The savings fluctuate between ∼30% and ∼50% for various w values. In practice, if a requester accepts the quality offered by a tournament with $w = 5$ for example, he can apply the optimization algorithm and save around 40% of his questions for the same quality.

The requester can use Algorithm 3 to improve tournament quality with a particular budget in mind. In particular, he may start with a tournament with constant w worker answers for each comparison. Then he may optimize over cost and obtain optimized sequences $\{\hat{r}_i\}$, $\{\hat{R}_i\}$, and $\{\hat{\delta}_i\}$. He can then start increasing these parameters till he exhausts his budget and then rerun the cost optimization algorithm, and so on. When he observes no improvement, he may stop his optimization.

6.3 Corner Case Study

This section explores some corner cases for tournament algorithms. We explore three scenarios:

Worst Case: Both the simulated annealing algorithm and previous work on tournaments [18] suggest that more human responses should be requested towards the end of the tournament. Thus, having very hard comparisons at the beginning of the tournament is not expected by these tournaments and may yield poor quality. For this scenario, we start with hard comparisons in the first step and decrease the difficulty as we get close to the end of the tournament.

In particular, at the first step, the top-1 and top-2 items are included in one comparison, the top-3 and top-4 in another one, and so on. Each of these comparisons is very difficult, since the items have very similar quality values. In the next steps, items with similar quality values are again grouped together. Thus, the winner of top-1 and top-2 is compared against the winner of top-3 and top-4, the winner of top-5 and top-6 is compared against the winner of top-7 and top-8 and so on. This type of grouping continues until there is a tournament winner.

Near-Best Case: This scenario attempts to bring easy comparisons at the first steps and to keep the difficulty of comparisons equal across the comparisons of each step. It first compares the top-1 and top-$\frac{|\mathcal{E}|}{2}$ items, the top-2 and top-$(\frac{|\mathcal{E}|}{2}+1)$ items, and so on. At the second step it compares the output of the comparison containing the top-1 and the output of the comparison containing the top-$\frac{|\mathcal{E}|}{4}$ item, and so on. The difficulty increases (on average) significantly as we get closer to the end of the tournament.

Average: This scenario explores the average case. Our cost, quality, and latency metrics are estimated via simulations. For each simulation, the input is randomly shuffled, simulating the requester who is unaware of the quality values of his input.

We used as input $\mathcal{E}$ a set of 32 items with quality values 1,000, 990, 980, ..., 690. To understand the effectiveness of the optimization algorithm, we compare three different tournaments, by applying each of them to the input 100,000 times. The tournaments we consider are

Const: A tournament algorithm with constant worker answers per comparison for all comparisons of the tournament. In particular, we report on the case where $r_i = R_i = \delta_i = w = 5$ for all i.

Static: We used the tuning strategy described in [18] with cost and latency constraints the cost and latency that Const had. Thus, Static will have better quality than Const with the same or better cost and latency. The tuning strategy returned $R_1 = 1$, $R_2 = 3$, $R_3 = 7$, $R_4 = 21$, and $R_5 = 39$. For all i, $r_i = \delta_i = R_i$ and thus the number of human responses for each comparison of each step is predetermined (hence the name Static).

Dynamic: This tournament algorithm was produced by our optimization algorithm for cost with initial parameters the ones that Static gave. Thus, the quality of Static and Dynamic should be the same, while Dynamic's cost and latency should be lower than those of Static. The parameters we obtained from the simulated annealing algorithm are $\delta_1 = 1$, $\delta_2 = 3$, $\delta_3 = 4$, $\delta_4 = 8$, $\delta_5 = 10$, $R_1 = 1$, $R_2 = 3$, $R_3 = 8$, $R_4 = 23$, $R_5 = 42$ and $r_i = \delta_i$ for all i.

Tournament	**Quality**	**Cost**	**Latency**
	Worst Case		
Const	0.400	155.0	32.0
Static	0.453	149.0	32.0
Dynamic	0.453	112.1	24.6
	Near-Best Case		
Const	0.480	155.0	32.0
Static	0.562	149.0	32.0
Dynamic	0.562	135.9	29.4
	Average		
Const	0.492	155.0	32.0
Static	0.585	149.0	32.0
Dynamic	0.585	133.4	28.8

Table 3: Corner cases study results.

Table 3 contains the results we obtained when we simulated the execution of Const, Static, and Dynamic on the three scenarios we described earlier (Worst Case, Near-Best Case, and Average). For each tournament execution we report the quality (using the MRR metric), average cost, and average latency observed.

We observe that

- *The unoptimized tournament Const has the worst cost, quality, and latency values in all cases comparing to the optimized tournaments.* Thus, optimizing tournament parameters can significantly affect a tournament's performance.
- *Both Const and Static have constant latencies and costs across all different scenarios*, because the number of worker responses requested per comparison is predetermined (and not dynamically decided, as in Dynamic).
- *Static and Dynamic have the same quality metric for all cases.* These tournaments are optimized for the Average scenario, but they perform similarly (in terms of quality) for all other scenarios we examined.
- *Dynamic saves between ∼10% and ∼25% of questions comparing to Static.* Observing the particular scenarios we see that: (1) For the Worst Case scenario, comparisons are hard in the first few tournament steps and then they get easier. Both Dynamic and Static are optimized for the average case (i.e., easier comparisons first, harder towards the end), and thus they are designed to ask few questions at the beginning and more towards the end. The fact that comparison difficulty is not as expected, significantly reduces quality for both Static and Dynamic. Also, Dynamic's cost is reduced, because

it does not exhaust the questions for the "easy" comparisons after the first step (i.e., slightly more than δ_i questions are usually enough). (2) The savings are smaller for the Near-Best Case, because the Dynamic tournament asks a few questions during the first few steps (comparisons then are easy) and then adapts to the difficulty of the comparisons as we get closer to the end of the tournament and nearly exhausts its budget. (3) On average (Average scenario), Dynamic saves more than 10% questions comparing to Static.

6.4 Optimal Solution Description

This section attempts to describe the optimal solution. As an example, we use the case of the optimized (over cost) parameters we find for the setting of Section 6.2. In particular, Figure 13 presents the $\hat{\delta}_i$ and $\hat{R}_i$ values observed across different steps of the tournament for the optimized case corresponding to $w = 5$. (We note that very similar results were obtained for all optimizations we have examined.)

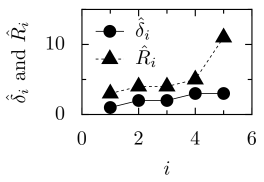


Figure 13: Optimal solution characterization: $\hat{\delta}_i$ and $\hat{R}_i$ across different steps i.

We see that both $\hat{R}_i$ and $\hat{\delta}_i$ are increasing as we get closer to the end of the tournament (i.e., i increases). The reason is that comparisons towards the end of the tournament are much harder than the comparisons at the beginning of the tournament and many fewer comparisons take place towards the final steps of tournaments. The fact that difficulty increases as we get closer to the end, explains why $\hat{\delta}_i$ increases at a smaller rate than $\hat{R}_i$: Even small increases in $\hat{\delta}_i$ will result in significantly more requests for human responses since it will be hard to get a "clear" winner. At the same time $\hat{R}_i$'s increase significantly and thus allowing more human responses to be obtained for each comparison (when needed).

This behaviour agrees with previous studies [18] which have found (for different worker models and tournament parameters) that more human responses should be requested towards the end of a tournament algorithm.

7 Related Work

A lot of work has been done in the context of our problem. In summary, comparing to previous work, our paper is different in at least two aspects:

Error models: The error models we consider are more complex than previously considered models.

Execution model: Crowdsourcing environments "force" algorithms to be executed in steps making the algorithm design harder.

In the remainder of this section we present work related to our paper.

Crowdsourcing. Marcus et al. [11] consider the problems of sorting and joining datasets in real crowdsourcing system, considering many practical issues. Their work implements max algorithms (as instances of sorting algorithms) by splitting the input set of items $\mathcal{E}$ in non-overlapping sets of equal size and asking workers to sort these sets. Moreover, in our previous work [18] we have examined static tournament algorithms with various error and compensation models. We tested many of these results in [17] to understand how a tournament algorithm works in practice.

Tournament algorithms have been used in crowdsourcing to determine the correct answer to a difficult task [14]. Answers provided by n workers to a task (not necessarily a comparison) are compared (in pairs) by workers. The winners of the pairwise comparisons are compared with each other in the next step. This process continues for a fixed number of steps, after which the majority answer is returned. The intuition is that workers are better in identifying the correct answer (when comparing it to other answers) than in producing the correct answer.

Our work mostly complements previous work with the introduction of dynamic (versus static) tournament algorithms and the error and execution model we use (both of which are more complex and realistic than previously used ones). After comparing the dynamic and the static techniques, we observed that dynamic tournaments can save significant amounts of money and execution time, and thus should be preferred in practice.

Sorting/Max algorithms with errors. One line of work similar to ours involves sorting networks, in which some comparators can be faulty. One of the first works in this area [20] proposes sorting algorithms that deal with two scenarios of faults: (1) up to k comparators are faulty and (2) each comparator has a probability δ (δ is small) of failing and yet, the result is retrieved correctly with probability greater than $1 - \varepsilon$ (ε is small). In [5] sorting networks that return the correct result with high probability are constructed from any network that uses unreliable comparators (each comparator can be faulty with some probability smaller than $\frac{1}{2}$). The depth of the resulting network is equal to a constant multiplied by the depth of the original network.

In [12] the max algorithm problem is considered under two error models: (1) up to e comparisons are wrong, and (2) all "yes" answers are correct and up to e "no" answers are wrong. The max (and the sorting) problem is also considered in [2] under a different error model: If the two items compared have very similar values (their absolute difference is below a threshold), then a random one is returned; otherwise, the correct item is returned. The max algorithm problem is solved using $2 \cdot n^{\frac{3}{2}}$ comparisons. An extension of the error model used in [12] is given in [1] where up to a fraction of p of the questions are answered erroneously at any point: If $p \geq \frac{1}{2}$, then the maximum item may not be retrieved; if $p < \frac{1}{2}$, then the maximum item can be retrieved with $\Theta\left(\frac{1}{(1-p)^n}\right)$ binary comparisons (where n is the size of the input). Even more elaborate error models are considered in [1]: For example, the number of wrong answers we have received may be at most p fraction of the number of questions *only at the end of the process* (and for example the first 5 questions may all be answered incorrectly). In this model if $p \geq \frac{1}{n-1}$, then the maximum item may not be found; if $p < \frac{1}{n-1}$, then the maximum item can be found in $n - 1$ questions. The complexity of four relevant problems is explored in [6, 7]. The problems considered are: (1) binary search, (2) sorting, (3) merging, and (4) selecting the k^{th} best item. The error model considered is the following: For any comparison of items there is a probability p that the correct result is returned and $1 - p$ that the wrong result is returned. Finally, [4] explores sorting and selection algorithms for dynamic data and attempts to always maintain a good solution for the output.

This line of work lacks the notion of steps and works with simpler (and less realistic) error and execution models than ours.

8 Conclusions

We investigated dynamic methods for retrieving the maximum item from a set in a crowdsourcing environment. We used tournament max algorithms and tuned their parameters to minimize expected cost (or latency), given constraints on the quality and latency (or cost respectively). We concluded, among other things, that

- The Thurstonian model can describe workers as comparators very accurately,
- The flexibility of selecting the number of questions to ask dynamically, allows large savings in expected cost and latency reaching 50% in some cases, and
- The optimized tournaments favor more worker responses towards the final steps.

Although not reported here, we have run full max algorithms (using the images described in Section 5.1) on Amazon's Mechanical Turk, and the results are consistent with the simulation results we have presented. Our results are preliminary, and when completed will be reported elsewhere.

Our work is an additional step in the development of rigorous max algorithms in crowdsourcing environments, but there are many other types of problems along the same lines that still need to be tackled. These problems include top-k retrieval and sorting, which we already have started considering.

References

[1] Martin Aigner. Finding the Maximum and Minimum. *Discrete Applied Mathematics*, 74(1):1–12, 1997.

[2] Miklós Ajtai, Vitaly Feldman, Avinatan Hassidim, and Jelani Nelson. Sorting and Selection with Imprecise Comparisons. In *Automata, Languages and Programming*, pages 37–48. 2009.

[3] Amazon's Mechanical Turk. www.mturk.com, June 2012.

[4] Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. Sorting and Selection on Dynamic Data. *Theor. Comput. Sci.*, 412(24):2564–2576, May 2011.

[5] Shay Assaf and Eli Upfal. Fault Tolerant Sorting Network. *FOCS*, pages 275–284, 1990.

[6] Uriel Feige, David Peleg, Prabhakar Raghavan, and Eli Upfal. Computing with Unreliable Information. In *STOC*, pages 128–137, 1990.

[7] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with Noisy Information. *SIAM J. Comput.*, pages 1001–1018, 1994.

[8] Irene Finocchi, Fabrizio Grandoni, and Giuseppe Italiano. Designing Reliable Algorithms in Unreliable Memories. In *Algorithms — ESA*, pages 1–8. 2005.

[9] Paul G. Hoel, Sidney C. Port, and Charles J. Stone. Introduction to Stochastic Processes. 1972.

[10] Scott Kirkpatrick, Charles Daniel Gelatt Jr, and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[11] Adam Marcus, Eugene Wu, David Karger, Samuel Madden, and Robert Miller. Human-Powered Sorts and Joins. *PVLDB*, 5:13–24, September 2011.

[12] Bala Ravikumar, K. Ganesan, and K. B. Lakshmanan. On Selecting the Largest Element in spite of Erroneous Information. In *STACS*, pages 88–99. 1987.

[13] Rainer Storn and Kenneth Price. Differential Evolution — A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11:341–359, 1997.

[14] Yu-An Sun, Christopher R. Dance, Shourya Roy, and Greg Little. How to Assure the Quality of Human Computation Tasks When Majority Voting Fails? Workshop on Computational Social Science and the Wisdom of Crowds (NIPS), 2011.

[15] James Surowiecki. *The Wisdom of Crowds.* Anchor, 2005.

[16] Louis Leon Thurstone. A Law of Comparative Judgment. *Psychological Review*, 34(4):273–286, 1927.

[17] Petros Venetis and Hector Garcia-Molina. Quality Control for Comparison Microtasks. In *CrowdKDD 2012*. ACM, August 2012.

[18] Petros Venetis, Hector Garcia-Molina, Kerui Huang, and Neoklis Polyzotis. Max Algorithms in Crowdsourcing Environments. In *WWW*, pages 989–998, 2012.

[19] Luis von Ahn. Games with a Purpose. *Computer*, 39:92–94, 2006.

[20] Andrew C. Yao and F. F. Yao. On Fault-Tolerant Networks for Sorting. Technical report, Stanford, CA, USA, 1979.