Crowd-Powered Find Algorithms

Anish Das Sarma [#1], Aditya Parameswaran [†2], Hector Garcia-Molina [*3], Alon Halevy [⊕4]

[#] *ClearList Inc., USA*
[1] anish.dassarma@gmail.com

[†] *Computer Science Dept., U. Illinois (UIUC)*
Urbana IL 61801, USA
[2] adityagp@illinois.edu

[*] *Computer Science Dept., Stanford University*
Stanford CA 94305, USA
[3] hector@cs.stanford.edu

[⊕] *Google Inc., USA*
[4] halevy@google.com

***Abstract*—We consider the problem of using humans to find a bounded number of items satisfying certain properties, from a data set. For instance, we may want humans to identify a select number of travel photos from a data set of photos to display on a travel website, or a candidate set of resumes that meet certain requirements from a large pool of applicants. Since data sets can be enormous, and since monetary cost and latency of data processing with humans can be large, optimizing the use of humans for finding items is an important challenge. We formally define the problem using the metrics of cost and time, and design optimal algorithms that span the skyline of cost and time, i.e., we provide designers the ability to control the cost vs. time trade-off. We study the deterministic as well as error-prone human answer settings, along with multiplicative and additive approximations. Lastly, we study how we may design algorithms with specific expected cost and time measures.**

I. Introduction

The field of *crowdsourcing*, which deals with solving hard problems by combining the power of machine and human computation, has gained a lot of traction in the last few years. Crowdsourcing techniques are used for solving problems that are hard for machines, such as comprehending and analyzing abstract concepts as well as media such as images, video and text. There have been several pieces of work on developing crowdsourced solutions for conventional algorithms, such as clustering [29], sorting [22], finding the best item [11], filtering [26] and categorization [27]. Other work has studied interface and workflow design [19], [21], [32], [34], obtaining the most informative training data for machine learning [31], designing algorithms with noisy comparators [10], as well as other marketplace aspects [12], [17], [23].

In this paper, we study a fundamental algorithmic problem we call CrowdFind, which arises in multiple crowdsourcing contexts. While we consider more general variants, the simplest version of CrowdFind is the following:

> CrowdFind: Given a (large) set of items, a boolean property P, and a number k, use humans to find k items that satisfy P.

In the general version, the property might not be boolean and we might have a more broad objective—for example, find three red items OR two blue items—instead of a prespecified number of desired items.

CrowdFind arises in several applications. As examples, a company may want to build a team of 20 young Java programmers from a large set of pre-screened resumes, or a travel website may want to identify 10 photos containing the Eiffel Tower from a dataset of travel photos. Of course, one could apply filtering [26], e.g., find all photos of Eiffel Tower in the set of photos. But adapting [26] would result in considering the entire set of items, and hence would be highly inefficient when we only need a small number of items satisfying the predicate, e.g., only 10 Eiffel Tower photos are desired. Therefore, we focus our efforts on identifying a subset of input items with desired properties.

Addressing the CrowdFind problem reveals a fundamental **cost-time tradeoff** which this paper studies in detail: At one end of the spectrum, there are solutions that require high monetary cost, but the overall latency of solving the problem is low, while at the other end, there are solutions that minimize cost but incur heavy latency. The following example illustrates this tradeoff.

Example 1.1: Consider a data set $\mathcal{I}$ of images, from which we want to find 10 images that satisfy a property P, i.e., whether it is a photo of a cat. We pick each image, and ask humans a question about it, i.e., "does the image show a cat?". Suppose on average that 20% of the photos are of cats. For the purposes of this example, we assume that humans do not make mistakes while answering questions.

Since crowdsourcing marketplaces have high latencies, we consider a space of algorithms where we are allowed to ask a batch of questions *in parallel*, and we do so over multiple "phases", where in each phase a number of questions are asked in parallel and answered together in one unit of time. Consider the following algorithms:

1) **Sequential:** Pick one image at a time, ask a human if the image shows a cat (in one phase), and then stop as soon as enough images satisfying P are gathered. The expected number of questions and phases for this

algorithm is $\frac{10}{0.2} = 50$. Note that this algorithm is *cost-optimal*, i.e., it only asks as many questions as strictly necessary.

2) **Parallel:** Ask all images in parallel in a single phase, incurring a heavy monetary cost depending on the size of $\mathcal{I}$. If $|\mathcal{I}| = 10000$, the number of questions for this algorithm is 10000, while the number of phases is 1.
3) **Hybrid-1:** Ask multiple images in the same phase, but no more than necessary: Ask 10 images in parallel in the first phase, and if x_1 images were found that satisfied P, ask $10 - x_1$ in the next phase, and if x_2 were found in the second phase, ask $10 - x_1 - x_2$ in the third and so on. In this case, the expected number of phases for this algorithm is much smaller than 50, while the expected number of questions is the same as the sequential algorithm.
4) **Hybrid-2:** A slight modification of the algorithm above might ask more than 10 images in the first phase, hoping to get all 10 images in the first phase, but without incurring much more cost than necessary. For instance, we may ask $\frac{10}{0.2} = 50$ images in the first phase, with an expected number of 10 images satisfying P. The expected phases for this one is less than Hybrid-1, while the expected number of questions is larger.

This paper proposes algorithms that balance the two factors—cost and time—in the most desirable way. We measure time or latency by the number of *phases* of crowdsourcing we need to perform, with one or more items asked in each phase, and measure cost as the total number of questions asked. Our paper effectively provides algorithms that generate solutions for CROWDFIND problem instances on-the-fly that are "optimal", i.e., they lie along the *skyline of solutions* of the cost-time tradeoff, and the balance between cost and time can be configured by the application. In other words, we provide knobs that let application designers control the point along the skyline that is desired.

We explore the cost-time tradeoff under two models of human responses:

- *Deterministic:* In this setting, every human gives an accurate response to every question, which is a reasonable assumption for properties that are easily evaluated by humans, e.g., whether a resume mentions a date of birth before 1990.
- *Uncertain:* In this setting, humans may give erroneous responses, which is often a more realistic assumption for properties that may be hard to evaluate, e.g., if a blurry photo contains the Eiffel Tower, or when we have humans providing lower-quality or spam responses.

Given the setting, we make the following contributions.

- **Deterministic Setting (Section III):** We study the cost-time tradeoff for the deterministic setting:
  - Given the cost-optimal sequential algorithm A, we find an algorithm A' that asks the same questions as A, but minimizes the number of phases of the algorithm. Essentially, we find the best parallelization A' of A.
  - Given the cost-optimal sequential algorithm A, and an (additive or multiplicative) approximation bound α, we find an algorithm A' that asks at most α more questions than A for every instance, but minimizes the number of phases. Essentially, we can use this algorithm to find any optimal point in the cost-time tradeoff.
- **Uncertain Setting (Section IV):** We study the cost-time tradeoff when humans may give erroneous responses:
  - We find a cost-optimal sequential algorithm A that minimizes cost. Unlike in the deterministic setting, this algorithm is non-trivial.
  - Given the sequential algorithm A, we find an algorithm A' that asks the same questions as A, but minimizes the number of phases of the algorithm.
  - Given the sequential algorithm A and an approximation bound α, we present two algorithms that ask at most α more questions than A, but have provable guarantees on the number of phases.
  - We formally show that adaptations of filtering [26]—which addresses a different problem and does not optimize for latency—for the uncertain setting are arbitrarily worse in both cost and number of phases than our algorithms.
- **Specific Points on the Skyline (Section V):** We show that our techniques can be extended to when we desire bounds on expected cost and expected number of phases.

  In the extended technical report [30], we also examine the case when we know that there is exactly one item satisfying the property, and study how the number of phases and cost is impacted. For instance, we may want to identify a specific suspect in a database of images of known criminals.

II. DEFINITIONS

We start by defining the class of CROWDFIND problems we tackle (Section II-A), then present our model of human computation (Section II-B), and then our metrics for comparing solutions to each problem (Section II-C). A table of symbols can be found in our technical report [30].

A. Class of Problems

The input to each problem consists of a (possibly infinite) set $\mathcal{I}$ of *items*. *Properties* of items capture attribute values: each property P has a finite domain D, and for every item I, $P(I) \in D$ denotes the value of the property for item I. Without loss of generality, we assume that $D = \{0, 1, 2, \ldots, m\}$, for some m. We define an item I to be *evaluated*, $e(I) = 1$, if $P(I)$ is known, else $e(I) = 0$.

An *output condition* $\mathcal{O}$ specifies when a set of evaluated items from $\mathcal{I}$ constitutes a solution to the CROWDFIND problem. Formally, $\mathcal{O}$ takes as input a set $\mathcal{F}$ of pairs of integers of the form: $\{(0, a_0), (1, a_1), \ldots, (m, a_m)\}$ and returns a value in $\{0, 1\}$. Each a_i is the number of items that are known to have $P(I) = i$, and the output condition returns 1 when we

have enough evaluated items for each value in the domain D. For convenience, we define $\mathcal{F}(i)$ to be equal to a_i of $\mathcal{F}$. We illustrate these definitions through an example.

Example 2.1: For a set of images, we may define a property P="contains cat", where the domain of P is $\{0, 1\}$, with 0 denoting an image without a cat, and 1 denoting one with. An output condition in this case might specify that we need to find two or more images that contain a cat. Thus, $\mathcal{O}(\mathcal{F}) = 1$ iff $\mathcal{F}(1) \geq 2$.

We may also define a property P="dominant color", whose domain would be a set of colors. In this case, the output condition may be to find two red items and three blue items, and we may stop as soon as we discover two or more red items and three or more blue items. Here, if red is 0 and blue is 1, then $\mathcal{O}(\mathcal{F}) = 1$ iff $\mathcal{F}(0) \geq 2 \wedge \mathcal{F}(1) \geq 3$.

A *problem* is specified by a set of input items $\mathcal{I}$, a property P on the set of items and an *output condition* $\mathcal{O}$. The goal is to discover the properties of as many items as is required to satisfy the output condition. More formally:

Definition 2.2 (Class of Problems and Solutions): Each problem $Q(\mathcal{I}, P, \mathcal{O})$ is specified by:

1) A (possibly infinite) database $\mathcal{I}$ of *items*.
2) A property P, which evaluates to a value in a domain $D = \{0, 1, \ldots, m\}$ for every item I.
3) A deterministic output condition, which is a function $\mathcal{O}$, taking as input a set $\mathcal{F} = \{(0, a_0), (1, a_1), \ldots (m, a_m)\}$, evaluating to a value in $\{0, 1\}$.

Given a problem $Q(\mathcal{I}, P, \mathcal{O})$, we say that a set of evaluated items $\mathcal{I}_s, \forall I \in \mathcal{I}_s, e(I) = 1$ is a *solution* to Q if and only if $\mathcal{O}(\mathcal{F}_s) = 1$ where $\forall i \in D, \mathcal{F}_s(i) = |\{I \in \mathcal{I}_s : P(I) = i\}|$.

In this paper, since we are dealing with search problems, we assume $\mathcal{O}$ is monotonic: that is, if we are given $\mathcal{F}_1$ and $\mathcal{F}_2$, where $\forall d \in D : \mathcal{F}_1(d) \leq \mathcal{F}_2(d)$, then $\mathcal{O}(\mathcal{F}_1) \leq \mathcal{O}(\mathcal{F}_2)$. In other words, the more we know about items, the closer we may be to satisfying the output condition. As an example, if we find three photographs that contain cats, and one that doesn't, we may be closer to satisfying the output condition than if we have just found two photographs containing cats. We note that the output condition $\mathcal{O}$ doesn't need to be specified by enumerating all sets of $\mathcal{F}$ that result in a solution; in fact, all our techniques apply when $\mathcal{O}$ is a black-box function.

We shall start by considering the case when the property P is boolean, i.e., evaluates to one of $D = \{0, 1\}$ (equivalent to a "yes/no question"). In this case, $\mathcal{F}$ is of the form $\{(0, a_0), (1, a_1)\}$. Our techniques also apply to the more general case when $|D| > 2$.

B. Computation Model

We now describe the components of our computation model.

Humans: The main component of any crowd-sourcing system, including ours, is asking questions to humans. Humans are asked questions about items, and we assume that all humans are identical and independent. Effectively, we model humans as (potentially noisy) logical processing elements. These are reasonable assumptions to make, since human workers on microtask platforms like Mechanical Turk are transient and work when they desire. Therefore, we may not be able to learn worker error rates for workers, or recruit only reliable workers for tasks: this is certainly true in many crowd-sourcing marketplaces.

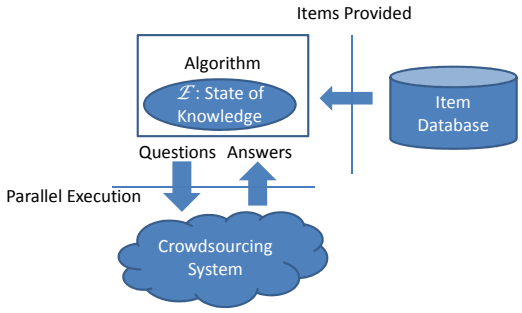

Fig. 1: Outline of a Search Algorithm

Question: We assume that for the property P, there is a corresponding *human question* H_P that can be asked on any item I to get an estimate of $P(I)$. We let $R(I)$ denote the multiset of answers to H_P on I obtained from humans. We define $(I, R(I))$ to be a *partially evaluated item*.

Given $R(I)$, a *strategy* $\mathcal{S}$ is used to estimate the value of P on I. Examples of strategies include taking the majority over a fixed number of repeated evaluations of H_P, or a threshold on a probability estimate. Our techniques apply to any strategy—we elaborate on strategies in Section IV. A strategy either estimates $P(I)$ to be 0/1, or returns "unknown", if $P(I)$ cannot be estimated (i.e., $\mathcal{S}(R(I)) = 0/1/\text{unknown}$.) Moreover, $e(I) = 1$ if $\mathcal{S}(R(I)) = 0$ or 1, and $e(I) = 0$ if $\mathcal{S}(R(I)) =$ unknown.

We distinguish two cases of computation in this paper:

- **Deterministic:** In the deterministic setting, humans don't make errors while answering questions. Thus, the result of H_P evaluated on item I, i.e., $H_P(I)$, is $P(I)$. Here, our strategy $\mathcal{S}$ is rather simple: $\mathcal{S}(R(I))$ = unknown if no question has been asked on I yet, and $\mathcal{S}(R(I)) = P(I)$ as soon as $R(I)$ has one or more answers.
- **Uncertain:** In the uncertain setting, humans may make errors, i.e., $H_P(I)$ may be any value in the domain D. Then, we may want to evaluate H_P multiple times on the same item I, and $\mathcal{S}$ takes into account the various $R(I)$ values.

We assume that the items are indistinguishable to begin with. Our techniques also generalize to the case when the items are provided scores a-priori based on how likely they are to satisfy the predicate (by, say, a machine learning algorithm), however, we will not discuss that further in this paper.

CROWDFIND **Algorithms:** Next we describe the model in which any CROWDFIND algorithm works in conjunction with a crowdsourcing system and the database of items $\mathcal{I}$, as illustrated in Figure 1. Any algorithm A is given a problem instance, i.e. a database of items $\mathcal{I}$, property P, and the output condition $\mathcal{O}$. The algorithm A maintains a *state of knowledge*, which is the set of partially evaluated items, denoted

$\mathcal{E} = \{(I_i, R(I_i))\}$. As the algorithm asks more questions to humans, this information grows by either gathering new property information for existing items (only necessary when the human answers are uncertain), or finding out properties of some new items. Algorithm A proceeds in *phases*. In phase i, the algorithm performs the following operations: (The pseudocode for any algorithm A is given in Algorithm 1.)

1) **Item Selection:** (lines 1-6 in Algorithm 1) The algorithm picks a multiset of items to ask human questions for in phase i. That is, Algorithm A picks a multiset $\mathcal{CQ} = \{I_j\}$, where $I_j \in \mathcal{I}$. If the algorithm decides to ask a question for an item that has not been seen before, then a new item is provided from the database of items $\mathcal{I}$.
2) **Human Questions:** (lines 7-10 in Algorithm 1) Algorithm A communicates the multiset of items $\mathcal{CQ}$ to ask questions to the crowdsourcing system. The crowdsourcing system *in parallel* asks the human question H_P on item I_j, for each $I_j \in \mathcal{CQ}$ to different humans. The corresponding set of newly obtained answers for items are added to the state of knowledge of the algorithm for each item (i.e., $R(I_j)$.)
3) **Test for Solution:** (lines 11-15 in Algorithm 1) The algorithm tests whether the current set of evaluated items constitutes a solution to the problem. The number of evaluated items that have property values for each value in the domain D is computed first, followed by an evaluation of the output condition $\mathcal{O}$. If the output condition is not satisfied, the algorithm moves on to the next phase, else the algorithm halts.

Algorithm 1: Algorithm Outline

```
   Data: P, O, E
   Result: E, T, M
   T ← 0;
   M ← 0;
   v ← 0;
   while v == 0 do
1      CQ ← set of questions to ask in current phase;
2      H ← ∅;
3      for I ∈ CQ do
4          if I ∉ E then
5              I ← get new item from I;
6          H ← H ∪ {I};
7      ask H in parallel using crowdsourcing service;
8      E ← E ∪ answers of H;
9      T ← T + 1;
10     M ← M + |CQ|;
11     F_s ← ∅;
12     // compute output condition;
13     for (I, R(I)) ∈ E do
14         if e(I) == 1 then
               F_s(S(R(I))) ← F_s(S(R(I))) + 1;
15     v ← O(F_s);
```

Naturally, the core logic of any Algorithm A rests in Line 1, wherein the multiset of questions $\mathcal{CQ}$ is selected. That logic will be the focus of our paper.

To enable us to compare the executions of various algorithms on the database, we make a few natural assumptions on the execution of algorithms: (1) We assume that the order in which "new" items are presented to algorithms is identical. (2) For each item, different algorithms receive the same answer for each question. For instance, if one algorithm receives an answer a when a question on item I is asked for the k'th time, then every algorithm would get the answer a for the k'th question on item I.

C. Performance Metrics

We assume that each human question takes one unit of cost to answer, and that a batch of questions being asked in parallel take one unit of time to be answered. The performance of any algorithm as described in Section II-B above can be expressed in terms of two quantities on a fixed instance I. (Note that in this paper, we focus on deterministic algorithms, rather than randomized ones.)

- **Latency** $\mathcal{T}$: The total number of phases $\mathcal{T}$ of an algorithm on an instance of the problem is the latency of the algorithm. We ignore any other computation time, such as testing whether the current set of items satisfies the output condition, since that is an automated test that is negligible compared to human tasks. Notice that we are implicitly assuming that all humans take around the same amount of time to answer questions. While this assumption may not hold exactly in real scenarios, in practice, we find that a bulk of the answers arrive at the same time, while a small number of answers arrive much later; a common strategy, therefore, is to cancel the outstanding answers and then issue a new batch of questions. Our algorithms also apply to the case where we cancel outstanding answers.
- **Monetary Cost** $\mathcal{C}$: The total monetary cost $\mathcal{C}$ is the number of human questions asked by the algorithm. If x_i is the number of human questions in the ith phase, then: $\mathcal{C} = \sum_{i=1}^{i=\mathcal{T}} x_i$

D. Optimization Problems

This section describes the optimization problems addressed in the rest of this paper; in all definitions below P refers to a property and $\mathcal{O}$ to the output condition. First, we describe the sequential problem, which is trivial for the case when humans do not make mistakes, but non-trivial for the uncertain setting:

Problem 1 (Sequential): Given a problem $(P, \mathcal{O})$, find an algorithm A that asks one question in each phase and returns the correct solution incurring the least monetary cost for each problem instance J.

We define the *optimal cost* on a certain problem instance J, $\mathcal{C}_{opt}(J)$ to be the cost taken by the sequential algorithm (i.e., the solution to Problem 1).

Then, we have the following problem which finds the maximum parallelism possible while ensuring optimal cost on every instance.

Problem 2 (Cost-optimal Max-parallel): Given a problem $(P, \mathcal{O})$, find an algorithm A such that for each problem instance J:

- Algorithm A returns a correct solution and has $\mathcal{C}(J) = \mathcal{C}_{opt}(J)$

- No other algorithm A' (which for every instance J, returns a solution and has $\mathcal{C}(J) = \mathcal{C}_{opt}(J)$) has lower latency.

However, by trading off some cost, we may be able to achieve better parallelism.

Problem 3 (α-multiplicative-approx. MP): Given a problem $(P, \mathcal{O})$, find an algorithm A such that for each instance J:

- Algorithm A returns a correct solution and has $\mathcal{C}(J) \leq \alpha\mathcal{C}_{opt}(J)$
- No other algorithm A' (which for every instance J, returns a solution and has $\mathcal{C}(J) \leq \alpha\mathcal{C}_{opt}(J)$) has lower latency.

In other words, we provide an α-approximate cost on every instance, and increase parallelism as much as possible. We can also define the problem based on additive approximation.

Problem 4 (α-additive-approx. MP): Given a problem $(P, \mathcal{O})$, find an algorithm A such that for each instance J:

- Algorithm A returns a correct solution and has $\mathcal{C}(J) \leq \alpha + \mathcal{C}_{opt}(J)$
- No other algorithm A' (which for every instance J, returns a solution and has $\mathcal{C}(J) \leq \alpha + \mathcal{C}_{opt}(J)$) has smaller latency.

The problems described so far capture *instance-specific guarantees*, i.e., the goal is to design algorithms that provide approximation guarantees per instance, relative to the sequential algorithm. Next, we describe a problem wherein the goal is to quantify *expected monetary cost* and *expected latency* of algorithms. Note that while expected monetary cost and latency guarantees do not translate to instance-specific guarantees—i.e., there may be algorithms whose expected costs and latencies are low, but may do extremely poorly on some instances—having expected cost and latency guarantees allows us to place and compare algorithms relative to each other on the two dimensional plane of cost and time.

Problem 5 (Expected Monetary Cost and Latency): Given an algorithm for a problem $(P, \mathcal{O})$, find its expected monetary cost and latency.

III. Deterministic Setting

This section describes solutions to the problems defined in Section 2 for the case in which humans do not make mistakes. Recall from the previous section that the main logic of Algorithm 1 lies in Line 1, where the algorithm selects a set of items $\mathcal{CQ}$ to be asked questions in parallel, based on the current state of execution $\mathcal{E}$. Moreover, since humans do not make mistakes, each item may be asked as part of $\mathcal{CQ}$ at most once. Thus, we can simply represent $\mathcal{CQ}$ using a single integer x, which signifies the number of new items to be asked in each phase.

When there is a single predicate with no errors in human answers, the state of execution $\mathcal{E}$ can be simply represented using a pair (a, b), where a is the number of items that have been verified to satisfy the single predicate P, and b is the number of items that have been verified to not satisfy the predicate.

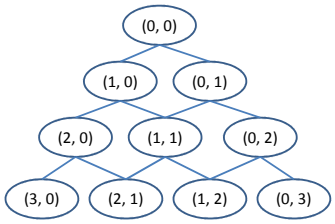

Fig. 2: Possible states of the system on asking up to 3 questions.

Example 3.1: Figure 2 depicts the reachable states on asking up to 3 questions when we have not asked any questions yet (i.e., state (0, 0)). For the rest of this section, we use an example output condition $\mathcal{O}$ which specifies that two or more items are desired that satisfy the predicate. Thus, states (2, 0), (2, 1) and (3, 0) are states that satisfy the output condition.

A. Problem 2: Optimal Cost

We first consider the problem of minimizing phases, while keeping the cost the same as the sequential algorithm. In our example, it is clear that 2 questions may be asked in the first phase since there are no states when one question is asked that satisfy the output condition. Subsequently, if the resulting state after the answers are obtained is $\mathcal{E} = (1, 1)$, only one question may be asked in the next phase, while if the state is $(0, 2)$, then two questions may be asked in the next phase.

We now design an online algorithm, called OptCost, that solves Problem 2. The algorithm proceeds as follows: Let the current state be (a, b). Then, in the next phase, the algorithm asks $x_m + 1$ questions where x_m is the largest x such that:

$$\forall i, j \geq 0, 0 \leq i + j \leq x, \mathcal{O}(\{(0, a + i), (1, b + j)\}) \neq 1 \quad (1)$$

In other words, the algorithm asks precisely as many questions in each phase until one of the reachable states is one for which the output condition is met.

We have the following theorem:

Theorem 3.2: Algorithm OptCost solves Problem 2.

The proof of this and other theorems in this section can be found in the extended technical report [30].

B. Problem 3: Multiplicative Approximation

The algorithm, called α-MultApprox, proceeds as follows: Let the current state be (a, b), and the number of questions asked so far be $y = a+b$. Then, the algorithm asks $\alpha \times (y + x_m + 1) - y$ questions in the next phase, where x_m is as defined in Section III-A. Thus, the algorithm asks up to $(\alpha - 1)(y + x_m + 1)$ additional questions beyond what OptCost asks.

Consider our example with $\alpha = 2$. Consider the case when the resulting state after the first phase is $(1, 3)$. (Thus, $y = 1 + 3 = 4$.) In this case, on asking one question, we can reach a state for which the output condition is satisfied $(2, 3)$ – thus OptCost will ask one question. However, α-MultApprox will ask $\alpha(y + x_m + 1) - y = \alpha(4 + 1) - 4 = 6$ questions in parallel in the next phase.

We have the following theorem:

Theorem 3.3: Algorithm α-MultApprox solves Problem 3.

C. Problem 4: Additive Approximation

The algorithm, called α-AddApprox, proceeds as follows: Let the current state be (a, b), and the current phase be y. Then, the algorithm asks $x_m + \alpha + 1$ questions corresponding to the largest x_m satisfying equation 1. Thus, the algorithm asks up to α additional questions beyond what OptCost asks.

We have the following theorem, whose proof is similar to that of Theorem 3.3:

Theorem 3.4: Algorithm α-AddApprox solves Problem 4.

D. Discussion

The previous subsections gave us optimal online algorithms for the deterministic variant of the CROWDFIND problem, given α, and whether we would like to use additive or multiplicative approximations. For a given value of α, the additive approximation algorithm uses the same "aggressiveness", i.e., it asks up to α additional questions in a given phase, independent of how many questions have been asked so far. On the other hand, the multiplicative approximation algorithm becomes progressively more aggressive as more questions are asked. For instance, for α = 2, if 3 questions are required to satisfy the output condition, the multiplicative algorithm asks up to 6 questions, but if 1000 questions are required to satisfy the output condition, the algorithm may ask 2000 questions.

An important hallmark of our algorithms is that as long as the output condition is not satisfied, we may switch from an additive to a multiplicative approximation or vice-versa. For instance, we may employ multiplicative approximation until a certain phase, and then switch to an additive approximation (thereby allowing the user of the algorithm to force the algorithm to be more conservative.) The reason why switching is possible is that we do not really "commit" to a specific approximation until the last phase when the output condition is eventually met. Until that phase, any questions used count towards satisfying the output condition, rather than being extra questions over and above the optimal number of questions. However, note that we may no longer have latency guarantees if we switch between strategies. The only guarantee we have is that the latency would be at most the sum of the latencies if we used only additive approximations, and if we used only multiplicative approximations.

IV. UNCERTAIN HUMAN ANSWERS

This section considers the case in which humans may make errors. We formally define the uncertainty setting in Section IV-A. Under the uncertainty model, even the best sequential algorithm is not obvious; we introduce the main sequential and crowd-sourced strategies in Section IV-B. We also consider finding only 1 items in Section IV-B (or only 0 items). We also show that adaptations of filtering [26] to CROWDFIND are arbitrarily worse than our algorithms. We present a brief description of extending to the case when we desire both 1 and 0 items in Section IV-C.

A. Model of Uncertainty

Questions and Answers: As before, there is a single predicate P that evaluates to 1 or 0 on each item and we may ask a human the value of P on any item I. However, human answers may not always be correct. To distinguish the item's true 1/0 predicate value from human responses, we label each human response as YES or NO: A YES value stands for a positive response indicating that the human thinks that the value of P on I is 1, while a NO value indicates that the human thinks that the value of P on I is 0.

To reason about probabilistic behavior, we assume that the error rates of the human workers (i.e., the probabilities of humans answering incorrectly—false positive and false negative error probabilities) and selectivity of the predicate P (i.e., the a-priori probability of any item satisfying the predicate) are both known in advance. These parameters are estimated using a *gold standard* sample, as is typical in crowdsourcing applications. The gold standard sample is evaluated by experts (to estimate selectivity), and by human workers (to estimate the error probabilities).

Strategies: With each item receiving a set of YES and NO responses, we need a *strategy* $\mathcal{S}$ to infer the 1/0 value for that item based on the set of human responses. Recall that a strategy simply takes as input the multiset of human answers $R(I)$ for a given item I, represented as a pair (n_1, n_2): where n_1 = number of YES answers and n_2 = number of NO answers, and returns 1/0/unknown. A strategy may then be represented using a two-dimensional grid, where a point (n_1, n_2) in the grid corresponds to the case when an item I has received n_1 YES answers and n_2 NO answers. Each point (n_1, n_2) in this grid is annotated with a value 1/0/unknown (i.e., $\mathcal{S}(n_1, n_2)$). Moreover, the grid is finite, i.e., there are a bounded number of answers required to arrive at a decision regarding any given item. Previous work [26] has considered many strategies for this inference (including algorithms to find optimal strategies), and our techniques apply to *all strategies*, examples include:

- *Majority(M):* For an odd natural number M, an M-majority strategy requires for every item I, precisely M human questions may be asked on it. I is declared a 1 item if and only it receives $\frac{M+1}{2}$ YES responses, and a 0 item if and only if it receives $\frac{M+1}{2}$ NO responses.
- *Rectangular(M_1, M_2):* Continue asking questions for any item until M_1 YES answers are obtained (in which case the item is declared to be 1) or until M_2 NO answers are obtained (in which case the item is declared to be 0).
- *Threshold(τ_1, τ_2, M):* (Similar to Wald's statistical test [36]). Given a set of n_1 YES responses and n_2 NO responses on an item I, $\Pr(1|n_1, n_2)$ denotes the probability of I being a 1. Given any specific error rate on humans, a-priori 1/0 selectivities, we may easily compute Pr using standard probability theory. A (τ_1, τ_2, M) probability threshold strategy, $\tau_1 \geq \tau_2$ defines a strategy where an item I with n_1 YES and n_2 NO responses is declared to be a 1 item if $\Pr(I = 1|n_1, n_2) \geq \tau_1$, and a

0 item if $\Pr(I = 0|n_1, n_2) < \tau_1$, or if M questions are asked, is declared a 1 if it receives $\frac{M+1}{2}$ YES responses, and 0 otherwise.

Assumptions: We assume that the output condition is specified by a pair (K_1, K_2): Given a set $\mathcal{I}$ of items, our goal is to find $\mathcal{I}_1, \mathcal{I}_2 \subseteq \mathcal{I}$ such that: (1) all items in $\mathcal{I}_1$ are inferred to be 1 items, all items in $\mathcal{I}_2$ are inferred to be 0 items (based on a chosen strategy); (2) $|\mathcal{I}_1| \geq K_1$, $|\mathcal{I}_2| \geq K_2$. (Here, unlike the deterministic setting, we do not consider output conditions $\mathcal{O}$ such as "find two 0 items or two 1 items".)

The key ideas of our algorithms are best presented by assuming $K_2 = 0$; i.e., assuming we are only looking for 1 items. So for most of the rest of the section we assume $K_2 = 0$, and consider extension to $K_2 > 0$ in Section IV-C. Additionally, to begin with, we make the assumption that the number of items is large (in comparison with K_1 or K_2); our algorithms generalize to the case when there is a bounded set of items. We describe this generalization in Section IV-E.

Expected Cost Computation: Given a set of n_1 YES responses and n_2 NO responses on an item I, we define $Y(n_1, n_2)$ to be the *expected cost to finding the next item that satisfies the predicate*; i.e., $Y(n_1, n_2)$ includes two cases:

- **Item I becomes a 1 item:** Let the probability of this event be PR_1. In this case, we want the expected number of questions required to declare I to be a 1 item; let this number of questions be denoted N_1.
- **Item I becomes worse than pursuing a new item:** Let the probability of this event be PR_2. In this case, we want $Y(0,0)$ plus the the number of questions required to make the cost higher than $Y(0,0)$. Let the number of questions required to make the cost higher than $Y(0,0)$ be denoted N_2.

The following expression combines the two cases above for $Y(n_1, n_2)$: $Y(n_1, n_2) = PR_1 * N_1 + PR_2 * (Y(0,0) + N_2)$. Given a strategy, we describe how to compute Y for each grid point in the strategy in Section IV-D.

B. Algorithms for $K_2 = 0$

We start by studying the case where we are only required to find K_1 1 items, and $K_2 = 0$ items of type 0 the converse case of $K_1 = 0$ is solved identically.

1) Problem 1: Sequential Algorithm: Consider the simple cost-optimal algorithm OptSeq shown in Algorithm 2 for any strategy. As the first step, we precompute $Y(n_1, n_2)$ for each point in the grid representation of the strategy. The algorithm simply picks an undecided item I (from set $\mathcal{U}$) with the lowest cost $Y(R(I))$ at each phase and asks a human if the item satisfies the predicate. We may maintain a priority queue of unprocessed items and their cost to pick the best candidate at each phase. Notice that all items have the same value $Y(0,0)$ to begin with, so we only need to sort the Y values of the items for which at least one question has been asked, rather than all items in $\mathcal{U}$. Whenever an item is inferred to be a 1 or a 0 (according to the strategy), it is removed from consideration (i.e., removed from set $\mathcal{U}$).

Algorithm 2: Algorithm OptSeq, a cost-optimal sequential algorithm for the $(K_1, 0)$ uncertainty problem.

```
Data: I, K_1, Pr(·), strategy S
Result: Set L of K_1 1 items
L ← ∅;
U ← I;
Compute Y for each point in the strategy S;
while |L| < K_1 do
    Pick I ∈ U with lowest Y(R(I));
    Ask a question on I to the crowdsourcing service;
    Add answer to R(I);
    if S(R(I)) = 1 then
        Remove I from U and add it to L;
    if S(R(I)) = 0 then
        Remove I from U;
```

Algorithm 3: Algorithm UncOptCost, a cost-optimal, phase-optimal algorithm for the $(K_1, 0)$ uncertainty problem.

```
Data: I, K_1, Pr(·), strategy S
Result: Set L of K_1 1 items
L ← ∅;
U = I;
Compute Y for each point in the strategy R;
while |L| < K_1 do
    Pick I' ⊆ U, |I'| = (K_1 − |L|) items with the lowest Y(R(I));
    CQ ← {};
    for each I ∈ I' do
        Let n^+ be the fewest YES responses required to make Y(R(I)) = 0;
        Let n^− be the fewest NO responses required to make
        Y(R(I)) = Y(0,0);
        Add min{n^+, n^−} questions on I to CQ;
    Ask CQ in parallel to the crowdsourcing service;
    Update R(I) for each I ∈ CQ based on answers;
    for each I ∈ CQ do
        if S(R(I)) = 1 then
            Remove I from U and add it to L;
        if S(R(I)) = 0 then
            Remove I from U;
```

2) Problem 2: Min-Cost Phase Optimal ($\alpha = 1$): Next we present a cost-optimal and phase-optimal algorithm UncOptCost, which uses an idea similar to Algorithm 2, but combines as many questions into a phase as will be definitely asked in Algorithm 2. (This enables us to give the guarantee of low number of phases while keeping the cost the same as Algorithm 2.) The pseudocode of UncOptCost is presented in Algorithm 3. Intuitively, the algorithm asks questions on at most K_1 items in each phase; these are the items with the lowest expected cost of a 1 decision based on the current number of YES and NO responses. To ensure that no unnecessary question is asked on any of these items, we ask the minimum number of questions that may: (a) either have the strategy $\mathcal{S}$ confirm the item as a 1 item, or (b) reduce the expected cost $Y(n_1, n_2)$ below the next highest item, removing it from the set of top items being considered. Based on the criteria for the set of items and number of questions asked at each phase, we can show that UncOptCost is a min-cost phase-optimal algorithm.

Theorem 4.1:
- UncOptCost asks all and only questions asked by OptSeq. Therefore, UncOptCost is a

min-cost solution to the $(K_1, 0)$ uncertainty problem.

- UncOptCost is phase-optimal.

Proof: (Sketch) For any algorithm that asks more questions than UncOptCost at any phase, it is easy to construct an input for which that algorithm asks strictly more questions than OptSeq. ∎

3) Problem 3: α-cost approximation ($\alpha > 1$): Next we consider α-multiplicative cost approximate algorithms. For any $\alpha > 1$, intuitively we are allowed to ask more questions in each phase than asked by UncOptCost. Broadly there are two ways to increase the number of questions we ask: We could expand the set of items and ask a similar number of questions on each item, or we could ask more questions on the set of items asked by UncOptCost. In the following, we consider both these approaches to asking more questions. For ease of presentation, we shall assume that α is an integer. In practice, all our algorithms can be extended to non-integer α; a trivial (and non-optimal) way to do this is by simply considering $\lfloor \alpha \rfloor$. Further, our ideas may be adapted for α-additive cost approximations, but details are omitted from this study.

Expand Set of Entities: We present Algorithm α-Expand, which is an α-cost approximate algorithm for the $(K_1, 0)$ uncertainty problem: Given an $\alpha > 1$, α-Expand runs α simultaneous instances of UncOptCost as follows. The first instance, a "master instance", of UncOptCost proceeds identically to the $\alpha = 1$ case, except that it keeps track of the total number of 1 items that have been found across all simultaneous instances of the algorithm. Therefore, every instance stops as soon as a total of K_1 1 items have been found. Simultaneously, we have a set of $(\alpha - 1)$ "slave instances" that mimic the master instance: Each slave instance maintains a running set of items on which to ask questions at each stage. The total number of items asked in each stage is identical to the master instance, and there is a one-to-one correspondence in these sets of items: If the master instance asks n_i questions on the i'th item, even the slave instance asks exactly n_i items on the i'th item. At the end of each phase, the total number of 1 and 0 items are computed, and removed from the working set of items.

Our next theorem summarizes the result of the α-Expand algorithm. Intuitively, the guarantee that we aim to get is that each of the instances contribute $\frac{K_1}{\alpha}$ 1 items. If we were to run the slave instances independent of the master, this will allow us to get to K_1 1 items in the same number of phases it takes for one slave to get $\frac{K_1}{\alpha}$ 1 items. However, since the slaves mimic the master exactly (in order to ensure that the algorithm is a α-multiplicative cost approximation), we multiply the number of phases by a "delaying factor", as can be seen in the following.

Theorem 4.2: Assuming an infinite set of input items, where each item is drawn independently and uniformly at random from some distribution, let the expected number of phases required by Algorithm UncOptCost to find K items by $\mathcal{ET}_{\texttt{opt}}(K)$, and let the expected number of phases required by Algorithm α-Expand be $\mathcal{ET}_{\alpha\texttt{-Expand}}(K)$. Suppose n^{max} is the maximum number of questions asked on any slave instance in one phase, and n^{min} is the minimum number of questions asked in any phase in the master instance, we have:

- α-Expand is an α-multiplicative cost approximation.
- $\mathcal{ET}_{\alpha\texttt{-Expand}}(K) \leq \min\{\frac{n^{max}\mathcal{ET}_{\texttt{opt}}(\frac{K}{\alpha})}{n^{min}}, \mathcal{ET}_{\texttt{opt}}(K)\}$

Proof: Since α-Expand maintains a one-to-one correspondence between the set of items in the master instance, and items in each slave instance, the total number of questions asked is α times the number of questions asked in the master instance. Since the master instance mimics UncOptCost, we have that α-Expand is an α-multiplicative approximation.

Let the random variable denoting the number of items obtained by α-Expand after T phases be $X_\alpha(T)$. α-Expand runs α copies of UncOptCost, and let the random variable denoting number of items obtained by each of these instances be $X_i(T)$, $i = 1..\alpha$. We have: $X_\alpha(T) = \sum_{i=1}^{\alpha} X_i(T)$. Therefore:

$$E[X_\alpha(T)] = E[\sum_{i=1}^{\alpha} X_i(T)] = \sum_{i=1}^{\alpha} E[X_i(T)]$$

Let us set $T = \mathcal{ET}_{\texttt{opt}}(\frac{K}{\alpha})$. Further, since each slave phase runs exactly the same number of questions as the master, for each item, each slave may require up to a multiplicative factor of $\frac{n^{max}}{n^{min}}$ more questions. Therefore, we have that:

$$\begin{aligned} E[X_\alpha(\frac{n^{max}}{n^{min}}\mathcal{ET}_{\texttt{opt}}(\frac{K}{\alpha}))] &\geq \sum_{i=1}^{\alpha} E[X_i(\mathcal{ET}_{\texttt{opt}}(\frac{K}{\alpha}))] \\ &= \sum_{i=1}^{\alpha} \frac{K}{\alpha} = K = E[X_\alpha(\mathcal{ET}_{\alpha\texttt{-Expand}}(K))] \\ \Rightarrow \mathcal{ET}_{\alpha\texttt{-Expand}}(K) &\leq \frac{n^{max}}{n^{min}}\mathcal{ET}_{\texttt{opt}}(\frac{K}{\alpha}) \end{aligned}$$

∎

Multiply Number of Questions: Our next approach, Algorithm α-Multiply proceeds by asking questions on exactly the same set of items as UncOptCost, but asking α-times as many questions on each item. If UncOptCost asked n_i questions in the i'th phase on a particular item I, then α-Multiply asks $\alpha \times n_i$ questions on I in the same phase. (Note that UncOptCost picks at most K_1 items based on probability order at the beginning of each phase; the ordering of items may be different for α-Multiply, but we still pick exactly the same set of items as UncOptCost to ensure α-cost approximation.)

Intuitively, our guarantee states that we may speed up the processing of items by a factor of α, while processing items in the same order. Once again, we need to add a "delaying factor" to account for the fact that we may end up asking extra unnecessary questions.

Theorem 4.3: Given an input set $\mathcal{I}$ of items, and K_1, let $T_{\texttt{opt}}$ be the number of phases required by `UncOptCost`; further, for each output item I_i ($i = 1..K_1$), let n^{max} and n^{min} be the maximum and minimum number of questions asked by UncOptCost in any phase. Let $\rho = \min\{\max_{i=1..K_1} \frac{n_i^{max}}{n_i^{min}}, \alpha\}$. We have that:

- α-Multiply is an α-multiplicative cost approximation.
- α-Multiply takes at most $\frac{\rho T_{\texttt{opt}}}{\alpha} + \frac{n^{max}}{n^{min}}$ phases to solve the $(K_1, 0)$ problem on $\mathcal{I}$.

Proof: Since each phase of α-Multiply asks at most α times as many questions as UncOptCost, the algorithm is an α-multiplicative approximation.

Let X be the number of phases taken by α-Multiply. Let $T_{\texttt{opt}}(i)$ be the number of phases for which I_i is active in UncOptCost, and let $X(i)$ be the number of phases I_i is active in α-Multiply. For each item I_i, UncOptCost asks at most $n_i^{max}T_{\texttt{opt}}(i)$ questions, and α-Multiply asks at least $\alpha X(i)n_i^{min}$ questions.

Since the total number of questions required to resolve each item is independent of the specific algorithm, we have that the number of questions asked by α-Multiply is at most αn_i^{max} more than that asked by the sequential algorithm. Therefore, we have that

$$\begin{aligned} \alpha X(i)n_i^{min} &\leq n_i^{max}T_{\texttt{opt}}(i) + \alpha n_i^{max}. \\ \implies X(i) &\leq \frac{\rho T_{\texttt{opt}}(i)}{\alpha} + \frac{n_i^{max}}{n_i^{min}}. \end{aligned}$$

Since this holds for each item i, we have: $X \leq \frac{\rho T_{\texttt{opt}}}{\alpha} + \frac{n^{max}}{n^{min}}$ ■

4) Comparison with Filtering: We now compare our algorithms against those developed in the prior work on filtering [26]. There are a number of key differences between [26] and our work: First, [26] designs algorithms for filtering and not CROWDFIND: the goal of filtering is to find *all* items in a data set that satisfy a given predicate. Second, [26] *does not optimize latency*—the focus is on designing strategies that optimize *monetary cost* and *accuracy*. Due to these reasons, our algorithms, that are tailored to CROWDFIND, and optimized for latency, perform much better.

Next, we formally show that adaptations of filtering algorithms to CROWDFIND have provably (a) much higher cost (b) much higher latency than our CROWDFIND algorithms. For simplicity, we consider the case in which the output condition requires K_1 items satisfying the predicate, i.e., we have a problem instance of the type $(K_1, 0)$. Our discussion also applies to the general (K_1, K_2) variant.

One obvious way of adapting [26] to CROWDFIND is to filter all items simultaneously; naturally, this algorithm has arbitrarily high $O(n)$ cost compared to our algorithms, all of which use cost proportional to $O(K_1/\sigma)$, independent of n.

The other way of using [26] for our problem is to filter in sequence; asking questions on one item until it is resolved to be a 1 or a 0. Once an item is resolved, we operate on the next item, and so on, until K_1 items are resolved to 1, thereby satisfying the output condition $\mathcal{O}$. We call this algorithm OptSeqFilter.

We first present a comparison of our sequential algorithm OptSeq with the sequential filtering algorithm OptSeqFilter. Note that the key difference between OptSeq and OptSeqFilter is that OptSeq may abandon an item even before it is resolved to a 0 or 1 (if there is another item with a better chance of being resolved to 1, at a lower cost) — while the filtering algorithm always continues filtering until we resolve to a 0 or 1. The following result establishes that: (1) The expected cost of OptSeqFilter is at least as much as OptSeq, (2) OptSeqFilter may incur arbitrarily higher cost compared to OptSeq.

Lemma 4.4:
- Given any problem instance P of the $(K_1, 0)$ problem over a set $\mathcal{I}$ of items and any strategy, the expected cost of solving P using OptSeq is at most as much as the expected cost of solving P using OptSeqFilter.
- Given any $M > 0$, we can construct a $(K_1, 0)$ problem P such that the expected cost of solving P with OptSeqFilter is $\Omega(M)$ and that of solving P with OptSeq is $\mathcal{O}(1)$.

The proof of this and the next lemma can be found in the extended technical report [30].

The previous lemma demonstrates that OptSeqFilter can have much higher expected cost than the cost optimal algorithm OptSeq. We next show how OptSeqFilter can have much higher expected latency as well as expected cost than our phase-optimal and cost-optimal algorithm UncOptSeq. Basically, the lemma states that our method of parallelizing as much as possible while retaining the same cost can result in significant gains in latency.

Lemma 4.5:
- Given any problem instance P of the $(K_1, 0)$ problem over a set $\mathcal{I}$ of items and any strategy, the expected cost and latency of solving P using UncOptCost is at most as much as the expected cost and latency of solving P using OptSeqFilter.
- We can construct a $(K_1, 0)$ problem P such that the expected latency of solving P with OptSeqFilter is $\Omega(K_1)$, and that of solving P with UncOptCost is $\mathcal{O}(1)$.

C. 1 and 0 Items

So far we only considered algorithms where the goal is to find K_1 1 items and zero 0 items. We show that for the general (K_1, K_2) problem of finding K_1 1 items and K_2 0 items, there is a simple 2-approximation algorithm in the number of phases. Consider Algorithm 2Step that operates as follows: (1) In the first step it solves the $(K_1, 0)$ problem using techniques destribed above, (2) In the second step it solves the $(0, K_2)$ problem, again using techniques described above, then combines the items obtained. It can be seen easily that 2Step is a 2-approximation to the latency of an optimal α-approximation algorithm $\mathcal{A}^*$.

Lemma 4.6: Given a problem instance for the (K_1, K_2) problem, and input α denoting the required cost-approximation. Let $\mathcal{A}^*$ be an optimal α-cost multiplicative approximation algorithm for an output condition that can be expressed as (K_1, K_2). We have that 2Step that applies the optimal α-cost approximation techniques on $(K_1, 0)$ and $(0, K_2)$ problems respectively is a 2-approximation of $\mathcal{A}^*$.

Proof: Let $T(A)$ be the number of phases required by algorithm A. Let the two stages of 2Step be A_1 and A_2. Since A_1 and A_2 are phase-optimal for the $(K_1, 0)$ and $(0, K_2)$ problems respectively, we have that:

$$\begin{aligned} T(\mathcal{A}^*(K_1, K_2)) &\geq T(A^*(K_1, 0)) \geq T(A_1(K_1, 0)) \\ T(\mathcal{A}^*(K_1, K_2)) &\geq T(A^*(0, K_2)) \geq T(A_2(0, K_2)) \end{aligned}$$

Adding the two equations gives our result:

$$\begin{aligned} T(\text{2Step}(K_1, K_2)) &= T(A_1(K_1, 0)) + T(A_2(0, K_2)) \\ &\leq 2T(\mathcal{A}^*(K_1, K_2)) \end{aligned}$$

■

Note that the 2-approximation above applies to either a min-cost phase optimal algorithm $\mathcal{A}^*$, or to any α-cost phase optimal algorithm. Effectively, given any algorithm $\mathcal{A}^*$, we have that 2Step achieves the same result as $\mathcal{A}^*$ in at most twice the cost.

D. Cost Computation

We now discuss how to compute the expected cost $Y(n_1, n_2)$ given a strategy. As in [26], a strategy can be represented as a *closed* region in the X-Y plane, i.e., there is a certain value m such that no item ever reaches $x + y = m$ (where x, y are the number of YES/NO answers from humans) during processing. Note that for most commonly used strategies, even a very small m (say $m = 5$) will suffice.

Given a strategy, there is a recursive dynamic programming algorithm that computes $Y(n_1, n_2)$ for all points within the strategy, starting at the boundary of the strategy, and proceeding towards $(0, 0)$. Consider a point (n_1, n_2). Let the probability of getting a YES answer at (n_1, n_2) be $p_1(n_1, n_2)$ and the probability of getting a negative answer at (n_1, n_2) be $p_0(n_1, n_2)$. $p_0(n_1, n_2)$ evaluates to the following expression: $\Pr(1|(n_1, n_2))\Pr(\text{NO}|1) + \Pr(0|(n_1, n_2))\Pr(\text{NO}|0)$. (There is a similar expression for $p_1(n_1, n_2)$.) These expressions do not depend on the strategy, but instead depend on n_1 and n_2, the selectivity of the predicate, and the error rates of human workers which depend on the item being 1 or 0.

At each point (n_1, n_2), we either have the option of starting afresh (i.e., starting with a new item), or proceeding with the current item by asking an additional question. If we ask an additional question and end up at $(n_1, n_2 + 1)$, then our expected cost from that point on is $Y(n_1, n_2 + 1)$, while if we end up at $(n_1 + 1, n_2)$, our expected cost from that point on is $Y(n_1 + 1, n_2)$. We have the following:

$$Y(n_1, n_2) = \min \begin{cases} Y(0,0) \\ p_1(n_1, n_2)Y(n_1 + 1, n_2) + p_0(n_1, n_2)Y(n_1, n_2 + 1) + 1 \end{cases}$$

If the first case is used, then we return to $(0, 0)$ (and start with a new item) if we reach (n_1, n_2), and in the second case, we ask an additional question. Our corner cases are the following: At all points of the strategy (n_1, n_2) where 0 is returned, we set $Y(n_1, n_2) = Y(0, 0)$, while at all points of the strategy (n_1, n_2) at which 1 is returned, we set $Y(n_1, n_2) = 0$. To compute $Y(n_1, n_2)$ for any pair of values n_1, n_2, we recursively unfold the equation above, until the base case of $(0, 0)$. Note, however, that the value of $Y(0, 0)$ is unknown, thus the expression keeps increasing in size as we move towards the origin. More precisely, the size of the expression could be as large as the number of points in the strategy (or exponential in m).

However, we may use the following theorem to guide our search for the right value of $Y(0, 0)$:

Theorem 4.7: The values of $Y(n_1, n_2)$ for all n_1, n_2 (not both 0) monotonically decreases as $Y(0, 0)$ decreases. Furthermore, if we substitute α for $Y(0, 0)$ and compute $Y(n_1, n_2)$ for all points (n_1, n_2) (not both 0), then $Y(0, 0) \geq \alpha$ iff $\alpha \leq p_1(0, 0)Y(1, 0) + p_0(0, 0)Y(0, 1) + 1$ (and vice versa.)

Thus, our approximate decision procedure is the following: We use a binary search for α between 0 and m. We start with a value of $Y(0, 0) = \alpha$, and use the equations above to compute the values of $Y(n_1, n_2)$ for all points where n_1, n_2 are not both zero, then compare α and $p_1(0, 0)Y(1, 0)+p_0(0, 0)Y(0, 1)+1$. If α is larger, then $Y(0, 0) < \alpha$ (and so α must be reduced), while if α is smaller, then $Y(0, 0) > \alpha$ (thus α must be increased.) We stop when $Y(0, 0) \approx \alpha$. Each iteration of the search procedure would take $O(m^2)$.

E. Discussion

So far, we assumed that $\mathcal{I}$ has an infinite number of items. Our algorithms continue to have the same worst-case guarantees as the infinite case, when $\mathcal{I}$ has a bounded number of items, as long as we can satisfy the output condition without examining all items (which is certainly true when $|\mathcal{I}|$ is much larger than K_1 or K_2.)

Once we end up examining all items, then our main issue is in the cost computation, where $Y(0, 0)$ is no longer the expected cost of a new item satisfying the predicate. (Specifically, $Y(0, 0)$ will be larger, and as a result, Y values for all items will increase.) We run our algorithms as before until we examine all items. At that point, we set $Y(0, 0)$ to be ∞, and recompute $Y(n_1, n_2)$. (This recomputation has the effect of setting Y for each item to be the cost of confirming that item to be a 1 or a 0 item.) Then, we continue execution of our algorithms. While approximate, this approach provides a practical solution when we end up examining all items.

In this section, we developed two approximation algorithms (one with expanding the set of items under consideration α-Expand, and one with a look-ahead for the same set of items α-Multiply). One could envision designing a hybrid algorithm that does both expansion as well as lookahead; we leave deriving bounds on performance of such an algorithm as future work. However, we may certainly use the worst case bounds of the two algorithms to select, for a given CROWDFIND problem, the algorithm which has better bounds and therefore will perform better.

Lastly, while we considered multiplicative approximation in this section, one could imagine an analog of the α-Multiply algorithm for additive approximation. We omit details of the derivation of performance bounds for such an algorithm.

V. PROBLEM 5: EXPECTED COST AND LATENCY

In this paper, so far, we focused on problems that provide *instance-specific guarantees*, i.e., we designed algorithms with approximation guarantees *for each instance*, relative to the sequential algorithm for the same instance. In this section, we derive *expected* cost and latency across all instances, for the algorithms that we devised. Deriving such guarantees will enable us to directly compare algorithms on cost and time.

We make simplifying assumptions now: we assume that our output condition $\mathcal{O}$ asks for k items that satisfy the predicate (recall that in earlier sections, we considered more general output conditions), and that humans do not make mistakes.

We begin by introducing some notation: We denote the selectivity of our predicate to be σ. (Recall that we used selectivity for computing expected cost Y in Section IV.) Given that our items satisfy the predicate with probability σ, we denote the expected monetary cost of an algorithm as $\mathcal{EC}$ and the expected latency or expected time as $\mathcal{ET}$ (where the expectation is over instances randomly generated using the selectivity information). We use N to denote the total number of items $|\mathcal{I}|$. (Our analysis would also apply if $\mathcal{I}$ was infinite.) Next we analyze $\mathcal{EC}$ and $\mathcal{ET}$ of completely sequential, completely parallel, and intermediate algorithms.

Sequential and Parallel Extremes: As in Section III, in the sequential case, we can ask one question at a time, and stop once we have k items satisfying the predicate. This approach has $\mathcal{EC} = \mathcal{ET} = \frac{k}{\sigma}$, since we examine k/σ items before we find k that satisfy the predicate.

The straightforward parallel algorithm that asks all items in a single phase is guaranteed to find k items that satisfy the predicate. This approach has $\mathcal{EC} = N, \mathcal{ET} = 1$.

Intermediate Solutions: We may now design algorithms to "parallelize" the sequential algorithm and potentially speed it up. For instance, we simply ask k questions in the first phase, ask only as many as necessary in the second phase (i.e., only as many items as strictly necessary to satisfy the output condition), and so on. To analyze algorithms of this type, let us first assume that when we ask questions, we get precisely what we expect. Subsequently, we will show how to ensure that with high probability we get what we expect.

In this setting, if we ask k questions in the first phase, the number of items that satisfy the predicate would be precisely σk. Then, we may ask $k - \sigma k$ in the next phase, and we get $\sigma(k - \sigma k)$ items that satisfy the predicate in the second phase. In the ith phase, generalizing this computation, it can be shown that we ask $k(1-\sigma)^{i-1}$ questions. We would like this number to be < 1, which guarantees that we have found enough items. That is, $i > 1 - \frac{\log k}{\log(1-\sigma)}$. Moreover, the total number of questions asked can be computed as follows: $\sum_{j=1}^{j=i} k(1-\sigma)^{j-1} = k\frac{1-(1-\sigma)^i}{\sigma}$

We can generalize the algorithm above by asking α times the remaining number of items required at each phase. That is, we begin by asking αk questions, then, if $k' \leq k$ items are still to be found by the second phase, we ask $\alpha k'$, and so on. Using similar ideas to the computation in the previous algorithm, we have: $i > 1 - \frac{\log(\alpha k)}{\log(1-\alpha\sigma)}$. Moreover, the total number of questions asked can be computed as follows:

$$\sum_{j=1}^{j=i} \alpha k(1-\alpha\sigma)^{j-1} = k\frac{1-(1-\alpha\sigma)^i}{\sigma}$$

However, note that in a given phase, we may not exactly obtain the number of items that we expect to satisfy the predicate. To ensure that with high probability we get *at least* the number of items that we expect in each round, we scale up the number of questions asked at each phase by a factor β, and we are guaranteed to get expected monetary cost and latency lower than those computed above:

$$\mathcal{ET} \leq 2 - \frac{\log(\alpha k)}{\log(1-\alpha\sigma)} \qquad \mathcal{EC} \leq \beta k \frac{1-(1-\alpha\sigma)^i}{\sigma}$$

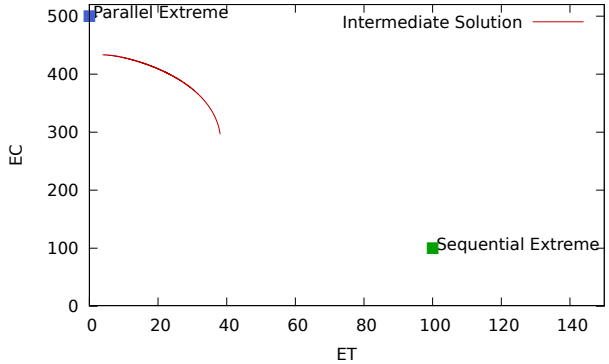

Fig. 3: Comparing algorithms on $\mathcal{ET}$, $\mathcal{EC}$; $k = 30, \sigma = 0.3, N = 500$

The following theorem formalizes this result.

Theorem 5.1: Given the set $\mathcal{I}$ of items, each with an independent probability σ of satisfying the input predicate, and a required number k of items to be found, the parallel algorithm of asking $\beta\alpha k_i$ items in the ith phase ($\alpha \leq 1/\sigma$), where k_i is the remaining number of items to be found has the following guarantees with $\beta = \lceil 2 + \frac{m^2(1-\sigma)}{k} \rceil$:

$$\Pr(M \geq \beta k \frac{1-(1-\alpha\sigma)^i}{\sigma}) \leq \frac{T}{m^2}$$

where T is the actual number of phases, whose expectation is: $\mathcal{ET} \leq 2 - \frac{\log(\alpha k)}{\log(1-\alpha\sigma)}$

The proof can be found in the extended technical report [30].

2D Plane of Algorithms: In figure 3, we depict the sequential and parallel extremes, along with the intermediate solutions on varying α for a scenario where $N = 500, k = 30, \sigma = 0.3, m^2 = 100$, and $\beta = \lceil 2 + \frac{m^2(1-\sigma)}{k} \rceil$. As can be seen, intermediate solutions provide valuable alternatives to sequential or parallel extremes (with at least one of $\mathcal{EC}, \mathcal{ET}$ lower).

VI. Related Work

In recent years, there has been a lot of work on designing fundamental algorithms using human computation, such as filtering [26], identifying the best item [11], sorts and joins [22], categorization [27], top-k [6], clustering [29], entity resolution [2], [35], debugging provenance [5], gathering knowledge [8]. Our work is unique in that not only does it study a different problem, CrowdFind, but also provides a principled way of reasoning about the skyline of solutions. Reference [9] gives a good overview of crowdsourcing, while [28] provides a taxonomy of the field.

Our work is related to the work on filtering [26]; however, as we proved in Section IV-B4, adapting filtering algorithms to CrowdFind can have arbitrarily worse cost and latency.

In [37], the goal is to use humans to assist in finding one image relevant to an image search query, which maps to an instance of our problem. The work develops a machine-learned model of the delay and accuracy of human workers, and uses it to heuristically determine whether to pursue the given image. Our work is more general in that it considers not only a broader class of problems, but also develops algorithms to determine all solutions that lie on the skyline of cost and latency.

Our work (especially when humans are assumed to not make mistakes) is also related to the vast field of parallel

computation [3], [4], [16], [33] wherein several models of parallel computation have been proposed, including PRAMs [16], Bulk-Synchronous Parallel processing [33] and LogP [4], and more recently, ones based on MapReduce [1], [15], [18]. There are a couple of key differences between this field and our work: First, this field typically assumes a fixed number of processors operating in parallel (in each phase)–while there are practical implementations that could vary the number of processors. On the other hand, we can dynamically vary the number of humans working on our tasks at any point. There is direct no notion of monetary cost in their setting, while in our setting the total number of human operations or questions is the total cost. Second, the main consideration in this field is to model and understand the tradeoffs between local computation and data storage at each processor and communication between processors. In our setting we have a central coordinator which does the computation between phases (assumed negligible in comparison with the latency of crowdsourcing), and leveraging human processors on demand to do simple tasks (Thus, the human processors don't communicate with each other.) Of course, parallel computation has no counterparts for the case when humans make mistakes.

In our work, we make the simplifying assumption that humans are equally likely to make errors, like in [26]. Recent work on crowdsourcing has tried to identify which workers to ask which questions [24], [25], as well as learning characteristics of workers while asking questions [7], [13], [14], [20]. We plan to incorporate more fine-grained error models in future work.

VII. Conclusions

In this paper, we studied the fundamental CrowdFind problem, relevant in many crowdsourcing applications. We developed solutions that lies on the skyline of cost and latency for two settings: when humans answer correctly, and when they may make errors.

Our focus was on a single predicate; in future work, we plan to consider multiple predicates, specifically, when the output condition may be represented as a boolean formula over predicates. In addition, we made the simplifying assumption that all workers are equally capable, identifying spam workers and learning accuracies of workers over time while solving CrowdFind problems are also interesting extensions.

References

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, 2012.

[2] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD Conference*, pages 783–794, 2010.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93*, pages 1–12, 1993.

[5] N. Dalvi, A. Parameswaran, and V. Rastogi. Minimizing uncertainty in pipelines. In *NIPS*, 2012.

[6] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. ICDT '13, pages 225–236, 2013.

[7] O. Dekel and O. Shamir. Vox populi: Collecting high-quality labels from a crowd. In *COLT*, 2009.

[8] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Using markov chain monte carlo to play trivia. In *ICDE*, pages 1308–1311, 2011.

[9] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 2011.

[10] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23:1001–1018, October 1994.

[11] S. Guo, A. Parameswaran, and H. Garcia-Molina. So Who Won? Dynamic Max Discovery with the Crowd. In *SIGMOD*, 2012.

[12] J. Horton and L. Chilton. The labor economics of paid crowdsourcing. *CoRR*, abs/1001.0627, 2010.

[13] J. Whitehill et al. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *NIPS*. 2009.

[14] D. R. Karger, S. Oh, and D. Shah. Budget-optimal task allocation for reliable crowdsourcing systems. *CoRR*, abs/1110.3564, 2011.

[15] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, 2010.

[16] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD-88-408, EECS Department, University of California, Berkeley, Mar 1988.

[17] S. Kochhar, S. Mazzocchi, and P. Paritosh. The anatomy of a large-scale human computation engine. In *HCOMP*, New York, NY, USA, 2010.

[18] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. PODS '11, pages 223–234, New York, NY, USA, 2011. ACM.

[19] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP*, 2009.

[20] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: a crowdsourcing data analytics system. *Proc. VLDB Endow.*, 5(10):1040–1051, June 2012.

[21] M. Bernstein et al. Soylent: a word processor with a crowd inside. UIST '10, pages 313–322, 2010.

[22] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. In *VLDB*, 2012.

[23] W. Mason and D. J. Watts. Financial incentives and the "performance of crowds". In *HCOMP*, 2009.

[24] P. Donmez et al. Efficiently learning the accuracy of labeling sources for selective sampling. In *KDD*, 2009.

[25] P. Welinder, P. Perona. Online crowdsourcing: rating annotators and obtaining cost-effective labels. In *CVPR*, 2010.

[26] A. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *SIGMOD*, 2012.

[27] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: it's okay to ask questions. In *VLDB*, 2011.

[28] A. Quinn and B. Bederson. Human computation: a survey and taxonomy of a growing field. In *CHI*, 2011.

[29] R. Gomes et al. Crowdclustering. In *NIPS*, 2011.

[30] A. D. Sarma, A. Parameswaran, H. Garcia-Molina, and A. Halevy. Finding with the crowd. In *Infolab Technical Report, 2013*.

[31] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[32] M. Toomim, T. Kriplean, C. Pörtner, and J. Landay. Utility of human-computer interactions: toward a science of preference measurement. CHI '11, pages 2275–2284, New York, NY, USA, 2011. ACM.

[33] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[34] L. von Ahn and L. Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8), 2008.

[35] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[36] L. Wasserman. *All of Statistics*. Springer, 2003.

[37] T. Yan, V. Kumar, and D. Ganesan. Crowdsearch: exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys*, 2010.