Answering Planning Queries with the Crowd (Technical report)

Haim Kaplan Ilia Lotosh Tova Milo Slava Novgorodov

School of Computer Science
Tel-Aviv University
{haimk,ilialoto,milo,slavanov}@post.tau.ac.il

March 1, 2013

Abstract

Recent research has shown that crowd sourcing can be used effectively to solve problems that are difficult for computers, e.g., optical character recognition and identification of the structural configuration of natural proteins [34]. In this paper we propose to use the power of the crowd to address yet another difficult problem that frequently occurs in a daily life - answering planning queries whose output is a sequence of objects/actions, when the goal, i.e, the notion of "best output", is hard to formalize. For example, planning the sequence of places/attractions to visit in the course of a vacation, where the goal is to enjoy the resulting vacation the most, or planning the sequence of courses to take in an academic schedule planning, where the goal is to obtain solid knowledge of a given subject domain. Such goals may be easily understandable by humans, but hard or even impossible to formalize for a computer.

We present a novel algorithm for efficiently harnessing the crowd to assist in answering such planning queries. The algorithm builds the desired plans incrementally, choosing at each step the 'best' questions so that the overall number of questions that need to be asked is minimized. We prove the algorithm to be optimal within its class and demonstrate experimentally its effectiveness and efficiency.

Chapter 1

Introduction

A planning query is a query whose output is a sequence of objects or actions that gets one from some initial state to some ideal goal state. Automated planning is a branch of artificial intelligence that tries to solve this problem using a computer [14]. However, there is a large class of planning queries that we meet in our daily life that is difficult for a computer to solve, not only because of the involved computational complexity, but because the goal state (as well as the consequence of individual actions) is hard or even impossible to formalize. In contrast, in many of these problems, the goal (and the effect of actions) is intuitively understandable by humans, making the planning humanly possible.

As a simple example, consider a vacation trip planning. A person may have some tentative start and end dates for her vacation, a preference of what she likes to do and a geographic area where she wants to travel. Based on this data she now needs to compile a potential set of places and attractions to visit and, from this set build a vacation schedule (essentially an ordered subset of the original set). A typical goal here may be to enjoy the vacation the most and/or to expand horizons. Such a goal is naturally subjective and hard to formalize (relevant factors may include total travel distances, attractions along the way, price and many more). However, people sharing similar taste/interests are likely to have the same notion of objective function and their experience and opinion can assist in the planning.

In general we are targeting here problems where one has a large set of items from which she needs to choose a subset and then order this subset in a sequence that will give the best value. The "value" definition is domain-specific, hard to formalize but easy to comprehend by humans. The vacation planning example above is one such instance. Another example is academic schedule planning, where the goal for instance is to obtain solid knowledge of a given subject area.

Answering such planning queries requires *expertise* in the domain of the problem, which is often gained by experience, solving instances of the same (or similar) problems. Since many people deal with similar planning problems, it is reasonable to assume that the *crowd* may provide useful insight here. Indeed, several attempts were made in this direction. For example, for academic schedule planning, the *CourseRank* system [5] allows students to rate courses and provides a convenient tool to compile recommended courses into schedule. Another example is the *Cross-Service Travel Engine for Trip Planning* [11] that allows harvesting POIs (points of interest) from various traveling recommen-

dation sites and provides a tool to compile a trip schedule from these POIs. Theses systems however focus on identifying the *set* of relevant items (courses, POIs), but the non-trivial task of *ordering* them in an ideal way, to form an actual plan, is left to the user.

Assisting the user in this fairly challenging task is the goal of the present work. We refer below to an ordered list of items as a *plan* and present CrowdPlanr, a system that employs the crowd to build "good" plans (w.r.t some abstract quality criteria) for specific tasks. It takes as input a set of relevant items (that can be retrieved from the existing systems mentioned above) and intelligently asks users from the crowd series of simple questions (about possible 1-step continuations of given partial plans), using the answers to identify the plans preferred by the crowd.

Intuitively, the set of all possible plans (ordered lists) that can be built from a given set of items can be modeled as a tree, where each node is an item, its ancestors are the items preceding it in the plan and its children are the items that may follow it. A root-to-leaf path in this tree represents a plan. One may rate (and correspondingly rank) plans by the probability of a person to consider a given plan as the best (w.r.t to the given abstract criteria). As the size of this tree may be extremely large (exponential in the size of the items set), it is clearly impractical to ask the crowd about each possible plan. Instead, we employ in CrowdPlanr a novel efficient algorithm that traverses this tree incrementally. It carefully restricts attention to the more promising plans - ones with highest maximum potential score (to be formally defined in the sequel) and optimally chooses at each step the 'best' questions (about possibly continuation), so that the overall number of questions that the crowd needs to be asked is minimized.

Note that the problem we are solving here can be viewed as a particular type of *sorting*. Using the crowd for implementing a sort-by operator is a problem that received much attention in recent crowdsourcing research [39, 7]. A key difference is that all these previous works assume the order between two elements to be independent of preceding elements, and thus the developed algorithms are based on the assumption that users can be asked to compared pairs of individual elements (e.g be asked if $A < B$). This is not the case here: the order in which plan items are selected depend not only on their individual value/properties but also on what precedes them in the plan (e.g. city A may be more attractive than B, but if in a trip a user first visits C, then A (being rather similar to it) may be skipped altogether and B should be visited instead. Consequently a new algorithm that efficiently provides users with the *context* relevant for their choice had to be developed here.

A first prototype of CrowdPlanr will be demonstrated in [26]. The demonstration gives only a high level overview of the the system capabilities and user interface. The present paper provides a comprehensive description of the formal model and algorithmic solutions underlying CrowdPlanr.

1.1 Technical contributions

- We introduce a simple generic model for modeling plans and interpreting crowd's answers to questions about them. Based on this model, we develop a formal definition of the planning problem and the identification of (approximated) best answer.

- We present an effective algorithm for identifying the (approximately) best answer using the crowd. As the search space may be extremely large, and consequently the number of questions that may be posed to user excessively high, the algorithm builds the desired plan incrementally, choosing at each step the 'best' questions so that the overall number of questions that need to be asked is minimized.
- We study formally the efficiency of our algorithm. Following common practice [12], we employ the notion of instance-optimality, that reflects how well a given algorithm performs compared to all other possible algorithms in its class and show our algorithm to be instance-optimal for a large common class of planning queries and data instances. Moreover, we show that the optimality ratio that our algorithm achieves (to be formally defined in the sequel) is far by at most a factor of two from the lowest possible optimality ratio.
- Finally, we discuss the implementation of the CrowdPlanr and demonstrate, by means of an extensive experimental evaluation, on both synthetic and real life data, that our algorithm consistently outperforms alternative baseline algorithms.

1.2 Work organization

In Chapter 2 we describe our data model and formally define the planning problem. In Chapter 3 we present the algorithm we developed to solve this problem. In Chapter 4 we discuss the algorithm performance, define the notion of instance-optimality and prove our algorithm to be instance-optimal for a large class of inputs. In Chapter 5 we discuss two extensions to our algorithm - finding top-k answers and working with multiple users in parallel. In Chapter 6 we present experimental results on both synthetic and real-world datasets. In Chapter 7 we survey the related work. We conclude and consider future work in Chapter 8.

Chapter 2

Preliminaries

We start with an intuitive description of our model, then proceed to the formal definitions.

We assume that we are given an initial finite set $\mathbb{S}$ of potential items to build a plan from. This set already reflects the preferences the user has defined when she requested a plan. There are multiple domain-specific tools that can be used for identifying this initial set $\mathbb{S}$ of items, e.g. *TripAdvisor* [38] for vacation trip planning, and we assume that one such tool has been employed. We will use this set to suggest to the user possible answers when we ask a question. Some of these items may become irrelevant as we progress, which will be reflected by the users not selecting them as answers.

CrowdPlanr allows users to build plans at different levels of granularity, zooming in and out between levels. For instance, in a trip to Europe, one can start by planning the countries to visit, then the cities in each country and the attractions within/between cities. Different granularity levels are often independent and we thus focus below, for simplicity, on a single level and explain things in this simplified context. The model extends naturally to the nested case, by allowing users, when dependencies do exist, to view the full detailed plan constructed so far, when considering its continuation.

As a simple running example we will use below the planning of a vacation in Italy (at the city granularity), starting from Rome. The set of items $\mathbb{S}$ in this case includes commonly visited Italian cities, e.g., {*Milan, Venice, Verona, Florence, Pisa, Trento, Bologna, Naples, ...*}. Note that, in general, not every user can answer every question. Indeed users that have never visited/read/heard vacation stories about Italy cannot help much in planning a vacation there. The targeting of questions to relevant users is by itself a challenging problem that may be addressed by a variety of methods (e.g. using semantic knowledge about users [2], employing collaborative-filtering based techniques [2, 3], etc.). In principle, any such black-box algorithm can be plugged into our system and we will assume below that the set of relevant crowd users has been identified.

Model

Given a set $\mathbb{S}$ of items, a *plan* is an ordered subset of $\mathbb{S}$. We will assume, and use, two special items in $\mathbb{S}$ - † to mark the beginning of the plan and ‡ to mark an end

of the plan. A *complete* plan is an ordered sequence of items $(\dagger, a_1, \ldots, a_k, \ddagger)$, with no repetitions, starting with a beginning marker and ending with an end marker. We also consider partial plans - prefixes that can be expanded by adding new items; these do not have an end marker. The set of all possible plans may be represented by a tree, called *a decision tree*, where the root is labeled by the start marker, internal nodes are labeled by items from $\mathbb{S}$, leaves are labeled by the end marker, and each internal node v_i represents a partial plan $p_i = (\dagger, a_1, \ldots, a_i)$, corresponding to the labels of nodes on the path from the root to v_i. The reason for using a tree (over a graph) is to allow for capturing the dependency between each item choice and the (partial) plan preceding it.(See below).

More generally, one may also want to consider plans where some items are unordered. For instance, when planning an academic schedule, the set of courses taken in a given semester may be unordered. This may be naturally incorporated into our model by having tree nodes that correspond to sets of items rather than individual ones. For a sake of clarity of the presentation we ignore it here.

The decision tree is built iteratively by asking users questions on its nodes. The question on a node v_k is of the form *"Given a sequence* $(\dagger, \ldots, a_k)$ *what should be the next item?"*, where $(\dagger, \ldots, a_k)$ are the labels on the nodes on a path from the root to v_k. To answer the user selects an item from $\mathbb{S}$. Thus, with each question a user is presented with a context of an existing sequence. Answers to these questions define a probability distribution on the children of every node. We use these distributions to define a score for every node - a *score of a node* is its probability to follow its parent in node's partial plan

Formally we define the decision tree as follows:

Definition 2.1 (Decision Tree)**.** *A* Decision tree T *is a labeled tree* $T(V, E)$ *with node labels from* $\mathbb{S}$*. The root of the tree is labeled by* $\dagger$*, leaves may be labeled by* $\ddagger$*, and all other node labels are from* $\mathbb{S} \setminus \{\dagger, \ddagger\}$*. For every node* $v \in V$ *the set of its children is denoted as:*

$$Children(v) = \{u | u \in V, (v, u) \in E\}$$

In addition, two functions are defined on the nodes of tree:

- $d_T : V \to \mathbb{N}$ *is a display counter.* $d_T(v)$ *counts number of questions asked on* v*.*
- $c_T : V \to \mathbb{N}$ *is a choice counter.* $c_T(v)$ *counts how many times* v *was chosen as an answer. For every node* v *it must hold that* $\sum_{u \in Children(v)} c_T(u) = d_T(v)$*.*

For each node v in the tree the combination of its display counter and the choice counters of its children define a conditional probability distribution of users choosing a particular child to follow v in a sequence. Thus we can easily define a probability of a sequence to be an optimal one by combining the conditional probabilities of the nodes composing it. Formally it can be defined as follows:

Definition 2.2 (Node score)**.** *We define node score in a tree* T *recursively:*

- *For the root (node labeled with* $\dagger$*):* $score_T(v) = 1$

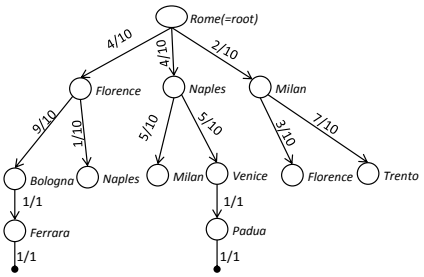


Figure 2.1: An example of a tree representing a set of plans

- *For a node u with a parent v, $score_T(u) = \frac{c_T(u)}{d_T(v)} score_T(v)$*

Example: To continue with our running example, a portion of the tree describing (partial) Italy vacation plans is depicted in Figure 2.1. The display and choice counts are depicted as labels on the edges incoming the nodes (for example let v, u, w be the nodes labeled with "Florence", "Bologna" and "Naples" respectively, then $d(v) = 10, c(u) = 9$ and $c(w) = 1$). In this figure, 10 questions were asked on most of the nodes, and 1 question on some. Black dots represent leafs marked with ‡. The scores of the leaves corresponding to some of the sequences are:

- *(Rome, Florence, Bologna, Ferrara, ‡)* - $\frac{4}{10} \cdot \frac{9}{10} \cdot \frac{1}{1} \cdot \frac{1}{1} = 0.36$
- *(Rome, Naples, Milan)* - $\frac{4}{10} \cdot \frac{5}{10} = 0.2$
- *(Rome, Milan, Trento)* - $\frac{2}{10} \cdot \frac{7}{10} = 0.14$

The previous definitions do not place an upper bound on the number of users that need to be asked in order to compute the probability distribution for a given node. In principle we could ask all available users for each node, but this exhaustive approach can be prohibitively expensive in practice. Instead, we expect applications to place a limit on the number of obtained answers. For this purpose, we define a threshold $\mathcal{N}$ that denotes the desired number of users to be probed for a node. (This may be determined, e.g., based on the desired sampling error bounds [17].) Thus, in principle, by asking $\mathcal{N}$ questions on all of the (incrementally added) nodes (until no more new nodes are added) we can obtain a complete tree.

Definition 2.3 (Complete tree)**.** A complete tree $\mathcal{T}$ *is a decision tree in which all leaves are labeled by ‡ and for each internal node the display counter equals $\mathcal{N}$.*

From the user perspective there is a semantic difference between a complete and partial sequence - a complete sequence cannot not be extended further (i.e. the users building it determined that this plan ends here). It makes sense to rank only complete sequences. This difference is naturally reflected in our model where ‡ markers are used to distinguish complete sequences:

Definition 2.4. *A sequence $p = (u_1, \ldots, u_k)$ is a* complete sequence *if and only if u_1 is marked with † and u_k is marked with ‡. All other sequences are* partial*. A set of all complete sequences in a tree T will be denoted as $\mathbb{P}(T)$, and the set of all partial sequences as $\mathcal{P}(T)$.*
The set of complete sequences in T containing node v will be denoted as $\mathbb{P}_v(T)$.

For example, in Figure 2.1 the sequence $(Rome, Florence, Bologna, Ferrara)$ is a complete sequence, while sequence $(Rome, Naples, Milan)$ is a partial one.

Now we can formally define a set of top-k sequences as:

Definition 2.5 (Top-k sequences)**.** *A set A of complete sequences is a* top-k set *if $|A| = k$ and for every complete path p' in $\mathbb{P}(\mathcal{T}) \setminus A$:*

$$\forall p \in A : score_{\mathcal{T}}(p) \geq score_{\mathcal{T}}(p')$$

We call the top-1 sequence an optimal sequence*.*

By nature, the scores computed by sampling a crowd of users are imprecise, in the sense that they only capture general trends: Sequences having similar scores are likely to have a similar "value" for the user. Consequently when two sequences have almost the same score it practically does not matter which one is returned as answer. Namely, it suffices to return a sequence whose score is *approximately* the best. Two types of approximations are common in the literature: relative approximation (i.e. approximation up to a constant *multiplicative* factor) and absolute approximation (i.e. approximation up to an *added* constant). Since we consider here probabilities and when plan scores get very low they become by nature not very interesting, we chose to use additive approximation. Formally we define:

Definition 2.6 (Approximated top-k)**.** *A set A of complete sequences is an* approximated top-k set *if $|A| = k$ and for every complete path p' in $\mathbb{P}(\mathcal{T}) \setminus A$:*

$$\forall p \in A : score_{\mathcal{T}}(p) \geq score_{\mathcal{T}}(p') - \varepsilon$$

The above definitions define a set of optimal (up to a constant) sequences in terms of the complete tree. Note however that, since the size of this tree may be extremely large (exponential in the size of the items set S), it is clearly impractical to build it fully and ask the crowd about each of its nodes. Instead, we employ an efficient algorithm that intelligently traverses the tree and processes only the minimal necessary parts. The algorithm discovers only a partial, small as possible, decision tree T, that contains sufficient information to guarantee that the set of k highest ranked (up to ε) sequences A in it remains the same in every possible complete tree that can be built by extending T. We call such T *a proof of correctness for* A. We will show in the sequel that the size of the proof of correctness found by our algorithm is $O(\frac{1}{\varepsilon} \cdot |S|)$. Formally, we define *a proof of correctness* as follows.

Definition 2.7 (Possible completion)**.** *A complete tree T' is* a possible completion of a decision tree T *if the following conditions hold:*

1. *T is a subtree of $\mathcal{T}$*
2. *$\forall v \in V_T : d_{T'}(v) \geqslant d_T(v)$*
3. *$\forall v \in V_T : c_{T'}(v) \geqslant c_T(v)$*

We denote a set of possible completions of T as $Compl(T)$.

Definition 2.8 (Top-K Proof of Correctness)**.** *A decision tree T is* a proof of a set A being a top-k set *if for all $\mathcal{T} \in Compl(T)$:*

$$\forall p \in A : \forall p' \in \mathbb{P}(\mathcal{T}) \setminus A : score_{\mathcal{T}}(p) \geq score_{\mathcal{T}}(p') - \varepsilon$$

Example: In the tree presented in figure 2.1 the sequence *p=(Rome, Florence, Bologna, Ferrara)* is the highest ranked sequence, however if we take $\varepsilon = 0.01$ then this tree is not a proof of correctness for the set $\{p\}$ - indeed, there is a possible continuation of this tree - T', where additional 9 questions are asked (recall that $\mathcal{N} = 10$) on *Bologna* node, and for all these questions we get *"Trento"* as an answer. Then, in T', the sequence p will have a score of $\frac{4}{10} \cdot \frac{9}{10} \cdot \frac{1}{10} = 0.036$, while a sequence *p' = (Rome, Florence, Bologna, Trento)* will have a score of $\frac{4}{10} \cdot \frac{9}{10} \cdot \frac{9}{10} = 0.324$.

Chapter 3

Planning using crowd

We are now ready to present our algorithm (Algorithm 2) for finding the optimal, up to ε plan. More generally, one may want want to find the top-k such plans. For simplicity we focus here on the top-1 plan and explain in Section 5.1 how the same approach may be extended for the top-k case. For brevity we will omit below the words "up to a constant" and whenever refer to an optimal plan we mean optimal up to a constant.

Our algorithm for finding an optimal plan will hold a decision tree (initially containing only the root) and will expand it by asking questions on its nodes. To achieve its goal the algorithm has to solve the two following sub-problems:

- Checking stop condition - i.e. checking whether the current tree is a proof of correctness for the current optimal plan
- Deciding which next questions to ask in order to reach stop condition as fast as possible

The algorithm is inspired by the well-known A^* algorithm [20] and the key challenge was to find the appropriate solution for these two points, that guarantee optimality.

3.1 Stop condition

To solve the first sub-problem we define a notion of uncertainty for a sequence. Uncertainty is the maximum possible difference between a given sequence score and the highest sequence score in all possible completions of the current state of the tree. Formally it is defined as follows.

Definition 3.1 (Uncertainty)**.** *In a tree T, an uncertainty for a complete sequence p is given by:*

$$U(T,p) = \max_{T' \in Compl(T)} \left[\max_{p' \in \mathbb{P}(T')} score_{T'}(p') - score_{T'}(p) \right]$$

Following definitions 3.1 and 2.8 we can use the uncertainty notion to check whether a decision tree is a proof of correctness for a sequence in it:

Lemma 3.2. *A tree T is a proof of correctness for a complete sequence p iff $U(T,p) \leq \varepsilon$.*

A naïve approach to calculating the value of uncertainty of a given decision tree would be to enumerate its possible completions. However, this approach is ineffective since every incomplete node of a tree can be extended with an arbitrary sub-tree. Instead, we use an efficient algorithm (Algorithm 1) that traverses the current decision tree only once in order to calculate the uncertainty.

Algorithm 1 Calculating $U(T,p)$

Assuming $p = (u_1, \ldots, u_m)$

1: $Deltas \leftarrow \emptyset$

2: **for all** $\{v | v \in V, v \neq m, d(v) < \mathcal{N}\}$ **do**

Assuming v is a part of a path
$p' = (u_1, \ldots, u_k, v_1, \ldots, v_n = v)$ *and*
$(u_1, \ldots, u_k)$ *is a common prefix of p and p'*

3: $maxCommon \leftarrow \prod_{i=2}^{k} \frac{c_T(u_i) + \mathcal{N} - d_T(u_{i-1})}{\mathcal{N}}$

4: $maxPPrime \leftarrow \frac{\mathcal{N} - d_T(v)}{\mathcal{N}} \prod_{i=1}^{n} \frac{c_T(v_i) + \mathcal{N} - d_T(u_{i-1})}{\mathcal{N}}$

5: $minP \leftarrow \prod_{i=1}^{m} \frac{c_T(w_i)}{\mathcal{N}}$

6: $\delta \leftarrow maxCommon \cdot (maxPPrime - minP)$

7: $Deltas \leftarrow Deltas \cup \{\delta\}$

8: **end for**

9: **return** $\max_{\delta \in Deltas} \delta$

This algorithm exploits the fact that the maximum difference in scores is achieved when one of the sequences gets its lowest possible score, while some other sequence gets its highest possible score. The algorithm goes iteratively over all nodes in T that we can ask more questions on and for every node v builds a sequence p' that ends one step after v (i.e. a shortest complete sequence that contains v). The algorithm then calculates maximum possible score difference between p' and p (line 3-6). The maximal common prefix of the two sequences is designated as $u_1, \ldots, u_k$. To achieve the maximum possible difference the algorithm assigns: highest possible score to the common part of p and p' (line 3), highest possible score to the reminder of p' (line 4) and lowest possible score to the reminder of p (line 5). At the end, the algorithm returns the maximum of the calculated differences.

Example: While calculating the uncertainty of a path ending by a node labeled "Padua" in a tree presented in Figure 2.1, the algorithm will build a sequence p' for a node "Bologna": $p' = (Rome, Florence, Bologna)$. The common prefix contains only the root, thus $maxCommon = score(root) = 1$, $maxPPrime = \frac{9}{10} \cdot \frac{4}{10} \cdot \frac{9}{10} = \frac{324}{1000}$ and $minP = \frac{4}{10} \cdot \frac{5}{10} \cdot \frac{1}{10} = \frac{20}{1000}$, and finally $\delta = \frac{304}{1000}$. The same calculation will be performed for all other nodes of the tree and the maximum δ will be returned.

Theorem 3.3 formally proves the algorithm correctness.

Theorem 3.3. *Given a decision tree T and a complete sequence p in it Algorithm 1 calculates $U(T,p)$.*

Proof. The maximum possible score of a complete sequence p in any $T' \in Compl(T)$ is upper bounded by T's current state - indeed there are only two options for p:

1. $p \in \mathbb{P}(T)$, then $p = (u_1, \ldots, u_k)$ will get a maximum score if for all remaining questions for every u_i, u_{u+1} will be chosen as an answer. In this case,
$$\max_{T' \in Compl(T)} score_{T'}(p) = \prod_{i=2}^{k} \frac{c_T(u_i) + \mathcal{N} - d_T(u_{i-1})}{\mathcal{N}}$$

2. *p is a continuation of some partial sequence $p' \in \mathcal{P}(T)$* (i.e. p' is a prefix of p), then the maximum score of p in $T' \in Compl(T)$ is exactly the maximum score of p' in T' (the maximum is achieved if all users select p as the only continuation of p'), and thus it can be calculated as in previous case.

On the other hand, the minimal possible score for a sequence $p = (u_1, \ldots, u_k)$ is achieved if for all the remaining questions on node u_i all the answers will be something other than u_{i+1}. And its minimal score would be:

$$\min_{T' \in Compl(T)} score_{T'}(p) = \prod_{i=2}^{k} \frac{c_T(u_i)}{\mathcal{N}}$$

Finally, if we have two sequences $p_1 = (u_1, \ldots, u_k, v_1, \ldots, v_m)$ and $p_2 = (u_1, \ldots, u_k, w_1, \ldots, w_m)$ ($u_1, \ldots, u_k$ is the common prefix of the two sequences) then the difference in their scores in a possible continuation T' is given by:

$$score_{T'}(p_1) - score_{T'}(p_2) = \left(\prod_{i=2}^{k} \frac{c_{T'}(u_i)}{\mathcal{N}} \right) \cdot \left(\prod_{i=1}^{n} \frac{c_{T'}(v_i)}{\mathcal{N}} - \prod_{i=1}^{m} \frac{c_{T'}(w_i)}{\mathcal{N}} \right)$$

And thus, it is maximized when one of the sequences gets all of the remaining votes (including the common prefix part of the sequence) and the second sequence (except for the common prefix) gets no more votes. □

3.2 What questions to ask

The second sub-problem any algorithm for finding an optimal sequence has to solve is deciding what questions to ask next. For simplicity we will present an algorithm that asks one question at a time, in Section 5.2 we will show how it can be extended to allow asking multiple questions in parallel. We employ a greedy approach to solve this problem - we ask questions on a sequence with the highest "potential", i.e. a sequence with a highest potential score. This approach is effective (as we will show in section 6) and can can further be extended for identifying a bulk of 'best questions', e.g. when multiple questions may be posed to users in parallel. For clarity we explain next in details how to choose a single next question, then briefly consider the selection of multiple questions.

Formally, the notion of sequence potential is defined as follows:

Definition 3.4 (Potential score)**.** *Given a tree T and a sequence p (partial or complete) in it:*

$$M_T(p) = \max_{T' \in Compl(T)} score_{T'}(p)$$

In general, there may be several sequences that have the highest potential. Since we do not have any additional information that allows us to prefer one over the other, we will consider all of them in a round-robin.

Algorithm 2 Finding the optimal sequence

1: $T \leftarrow origin$
2: $i \leftarrow 0$
3: **while** $\mathbb{P}(T) = \emptyset$ OR $\min_{p \in \mathbb{P}(T)} U(p, T) \geq \varepsilon$ **do**
4: $\quad allLeafs \leftarrow Leafs(T)$
5: $\quad bestLeaf \leftarrow \text{argmax}_{p \in allLeafs} M(p)$
6: $\quad Candidates \leftarrow \{bestLeaf\}$
7: $\quad TopNodes \leftarrow \{tN(bestLeaf)\}$
8: $\quad maxScore \leftarrow M(bestLeaf)$
9: $\quad$ **if** $maxScore > \varepsilon$ **then**
10: $\quad\quad$ **for all** $p \in allLeafs$ **do**
11: $\quad\quad\quad$ **if** $M(p) = maxScore$ **then**
12: $\quad\quad\quad\quad$ **if** $tN(p) \notin TopNodes$ **then**
13: $\quad\quad\quad\quad\quad Candidates \leftarrow Candidates \cup \{p\}$
14: $\quad\quad\quad\quad\quad TopNodes \leftarrow TopNodes \cup \{tN(p)\}$
15: $\quad\quad\quad\quad$ **end if**
16: $\quad\quad\quad$ **end if**
17: $\quad\quad$ **end for**
18: $\quad\quad p \leftarrow Candidates[i \mod |Candidates|]$
19: $\quad\quad$ Ask a question on $tN(p)$
20: $\quad\quad i \leftarrow (i+1) \mod |Candidates|$
21: $\quad$ **else**
22: $\quad\quad p \leftarrow Candidates[0]$
23: $\quad\quad$ Ask a question on lowest node of p
24: $\quad$ **end if**
25: **end while**
26: **return** $\text{argmin}_{p \in \mathbb{P}(T)} U(p, T)$

If there is more than one minimum(maximum) item, argmin*(*argmax*) shall return one at random*

Each iteration of this algorithm finds a node in the current decision tree T and asks a question on it. The algorithm stops (condition on line 3) when the uncertainty of some node p in the tree drops below ε. When this happens, following Lemma 3.2 T is the proof of correctness for p.
On line 4 we find a sequence (partial or complete) with the maximum potential score. Following the proof of Theorem 3.3, max potential score can be calculated using a simple formula in linear time (in the length of the sequence).

Given a sequence we have also to choose a node in it to ask a question, this is done on line 5. We prefer to ask questions on higher nodes as they affect more paths in the tree. Formally we define:

Definition 3.5 (Top-node)**.** *For a path* $p = (v_1, \ldots, v_k)$ a top node - $tN(p)$ *is a node* v_i*, s.t.* i *is the minimum item in the set* $\{i | 1 \leq i \leq k, d(v_i) < \mathcal{N}\}$ *(i.e. a topmost node that was not yet exhausted).*

The loop in lines 8-15 selects all the sequences that have the maximum potential score. We ask questions on all of them in the round-robin manner. Finally, on line 7 we check for a special condition - if all of the sequences in the tree can not have score greater than ε, then it does not matter which sequence we return - all of them are optimal by definition, thus we just need to discover one complete sequence and return it. The easiest way to do it is by asking questions on the lowest possible node - take any sequence and ask a question on its last node, if the answer terminates the sequence return it, otherwise ask a question on a newly discovered node.

Example: given a decision tree presented in Figure 2.1 our algorithm will ask a question on a node labeled "Bologna" since it's the highest non-exhausted node of a sequence with the highest potential (the potential of a sequence $(Rome, Florence, Bologna, Ferrara)$ *is* $\frac{4}{10} \cdot \frac{9}{10} \cdot \frac{10}{10} = \frac{36}{100}$*).*

It is clear that the algorithm eventually halts - the number of questions we can ask is bounded by the size of $\mathcal{T}$ (times $\mathcal{N}$) which is finite. It is also clear that when it does, there is a sequence p in T for which $U(T, p) < \varepsilon$ and this is the sequence that is returned (lines 3 and 24). Thus, following Lemma 3.2 the algorithm returns an optimal sequence. In Section 4 we perform a detailed analysis of the algorithm's efficiency.

Chapter 4

Efficiency and optimality

In this chapter we will discuss the efficiency and the optimality of the algorithm presented above and will also provide a lower-bound for the possible optimality ratio. To discuss optimality we need to define a cost measure and a set of inputs, based on which we will compare different algorithms.

We use the number of visited nodes in a tree (i.e. number of nodes we asked questions about) as our *cost measure*. This number is in direct correlation to the actual number of questions asked - indeed we assumed that in order to learn a probability distribution of a continuation from a node one has to ask $\mathcal{N}$ questions on this node. Furthermore, using number of nodes as a cost measure, rather than the actual number of asked questions, makes reasoning about optimality much simpler.

The class of inputs we consider is the set of all possible complete trees composed from items in $\mathbb{S}$ where the difference between the shortest and the longest sequence is at most $\mathbf{k}$, for some predefined constant $\mathbf{k}$. For a given $\mathbf{k}$ we denote the corresponding class of inputs as $\mathbf{I_k}$. As we saw in our experiments, typical real-world inputs fall into $\mathbf{I_k}$ for fairly small value of $\mathbf{k}$ (comparable plans of the same granularity usually contain similar number of items).

We use an instance-optimality notion as it appears in [12]:

Definition 4.1 (c-Optimality)**.** *For a class of inputs* $\mathbf{I}$ *and a class of algorithms* $\mathbf{A}$*, algorithm* $\mathcal{A} \in \mathbf{A}$ *is* c-optimal *if for every input* $\mathcal{I} \in \mathbf{I}$ *and for every algorithm* $\mathcal{B} \in \mathbf{A}$*:*

$$cost(\mathcal{A}, \mathcal{I}) \leq c \cdot cost(\mathcal{B}, \mathcal{I}) + c'$$

We refer to c *as the* optimality ratio*.* c' *is also a constant.*

We will prove next the following two results. The first proves the instance optimality of our algorithm and the second shows its optimality ratio is far by at most a factor of two from the lower bound.

Theorem 4.2. *Algorithm 2 (*$\mathcal{A}$*) is* $\frac{1}{\varepsilon} - optimal$ *on trees from* $\mathbf{I_k}$*.*

Theorem 4.3. *Let* $\mathcal{B}$ *be a deterministic algorithm for finding an optimal (up to* ε*) sequence which is* $c - optimal$ *on trees from* $\mathbf{I_k}$*. Then* $c \geq \frac{2}{\varepsilon}$*.*

To prove Theorem 4.2 we will first analyze the performance of Algorithm 2 and show that it asks questions about at most $\frac{1}{\varepsilon} - 1$ different sequences (later,

in Section 6, we show that in real-life cases our algorithm considers even less sequences). To do this we will introduce a concept of *"Candidates pool"*, a set of nodes that our algorithm considers to ask questions about. We will show that during the run of the algorithm, only a limited number of nodes can enter the pool, thus limiting the total number of different paths considered by our algorithm.

Theorem 4.4. *Algorithm 2 asks questions about at most* $\frac{1}{\varepsilon} - 1$ *different sequences during its run.*

Proof. Let's call Algorithm 2 A. Only sequences with potential maximum score greater than ε are considered by A, A asks questions only on the top-nodes of the sequences in $Candidates$. Let P_i be the set (pool) of all top-nodes that are part of a path with potential maximum greater than ε after question i. It is clear that the node for question $i+1$ is chosen only from P_i. If a node v was in P_i but is not in P_{i+1} we say that node v has *left the pool*. Node v can leave the pool only in one of the following cases:

1. v is no longer a top-node, i.e. all possible questions on it were asked, but it still is a part of a path with potential max score greater than ε. In this case one or more children of v will be in P_{i+1}.
2. v is no longer a part of path with potential max score greater than ε, it means that no descendants of v will be in P_j for $\forall j > i$.

It is clear that once a node has left the pool, it can not return. Let Out be the set of all nodes that left the pool for the reason 2, formally $Out = \{v | \exists 1 \leq i \leq m : v \in P_i \wedge v \notin P_{i+1} \wedge (\forall u \in Children(V) : u \notin P_{i+1})\}$.
After question m every path that was ever considered by A has a node that is a part of it in either P_m or Out.
$M(p) \leq score_T(tN(p))$: For every possible completion T' of T, $p \in \mathbb{P}_{tN(p)}(T')$, thus following Lemma 4.6 $score_{T'}(p) \leq score_{T'}(tN(p))$, by the definition of top-node, its score is final (since the display count of its parent equals $\mathcal{N}$), thus $score_{T'}(tN(p) = score_T(tN(p))$, hence following the definition of potentially maximum score we get that $M(p) \leq score_T(tN(p))$.
This means that for every $v \in P_m$, $score(v) > \varepsilon$ and also for every $u \in Out$, $score(u) > \varepsilon$ (since every node in Out was once in the pool and its score was already final then).
No two nodes in P_m are a part of a same path (by the definition of $top-node$). The same is true for nodes in Out (indeed if v has moved to Out, its children can not even enter the pool so they can not be moved to Out either). There are also no $v \in P_m$ and $u \in Out$ such that u, v are parts of the same path (for the same reason). Thus all the nodes in $P_m \cup Out$ are parts of a different sequences. Hence (by Lemma 4.7) $\sum_{v \in P_m \cup Out} score(v) \leq 1$. And since for every $v \in P_m \cup Out$, $score(v) > \varepsilon$ we have that $\sum_{v \in P_m \cup Out} score(v) > |P_m \cup Out| \cdot \varepsilon$.
From these two facts we get that $|P_m \cup Out| < \frac{1}{\varepsilon}$.
Thus algorithm A considers at most $\frac{1}{\varepsilon} - 1$ different sequences during its run. □

Corollary 4.5. *The size of the proof of correctness tree found by Algorithm 2 is* $O(\frac{1}{\varepsilon} \cdot |S|)$.

Proof. The proof follows from the fact that the tree contains at most $\frac{1}{\varepsilon}$ different paths, and each path contains at most $|S|$ nodes. □

To complete the proof of Theorem 4.4 we prove the following two lemmas.

Lemma 4.6. *For every node $v \in V$, $\sum_{p \in \mathbb{P}_v(T)} score(p) = score(v)$. In particular $\sum_{p \in \mathbb{P}(T)} score(p) = 1$.*

Proof. By induction on the tree structure. For leaves the claim is true since $\mathbb{P}_v(T)$ contains exactly one path - from the root to v, and thus $\sum_{p \in \mathbb{P}_v(T)} score(p) = score(v)$ by definition.
Let v be an internal node and assume the claim is true for its children. Let $\{u_1, u_2, ..., u_n\}$ be the set of v's children. For every u_i there is a single path from the root to u_i - $\{w_1, w_2, ...w_k\}$ (where w_k is u_i and w_{k-1} is v). Thus:

$$\begin{aligned} score(u_i) = \prod_{j=2}^{k} \frac{c(w_j)}{d(w_{j-1})} &= \frac{c(w_k)}{d(w_{k-1})} \prod_{j=2}^{k-1} \frac{c(w_j)}{d(w_{j-1})} \\ &= \frac{c(u_i)}{d(v)} score(v) \end{aligned}$$

Since a path containing u_i can not contain u_j we can split $\mathbb{P}_v(T)$ into $\bigcup_{i=1}^{n} \mathbb{P}_{u_i}(T)$, and thus:

$$\begin{aligned} \sum_{p \in \mathbb{P}_v(T)} score(p) = \sum_{i=1}^{n} \sum_{p \in \mathbb{P}_{u_i}(T)} score(p) = \\ = \sum_{i=1}^{n} score(u_i) = score(v) \sum_{i=1}^{n} \frac{c(u_i)}{d(v)} \end{aligned}$$

Choice counters of the children sum up to a display counter of the parent, thus $\sum_{i=1}^{n} \frac{c(u_i)}{d(v)} = 1$, and hence $\sum_{p \in \mathbb{P}_v(T)} score(p) = score(v)$. □

Lemma 4.7. *Let A be a set of nodes of a tree T, s.t. no two nodes in A are a part of a same sequence. Then $\sum_{v \in A} score_T(v) \leq 1$.*

Proof. Let's build a tree T' from T by terminating a path at each $v \in A$, the rest of the tree remains as is. Now, for each $v \in A$ there is a path p_v in T' (since all of the nodes in A are parts of different paths, there is no conflict). Also, $score_{T'}(p_v) = score_T(v)$ (since we left the rest of the tree as is, all the nodes from the root to v in T and T' have the same counter values). $\{p_v | v \in A\} \subseteq \mathbb{P}(T')$, thus by Lemma 4.6 $\sum_{v \in A} score_T(v) = \sum_{p \in \{p | v \in A\}} score_{T'}(p) \leq \sum_{p \in \mathbb{P}(T')} score_{T'}(p) = 1$. □

Now we can prove the optimality ratio of Algorithm 2.

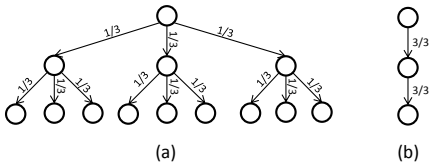


Figure 4.1: (a) Scattered-tree of depth 2 ($\mathcal{N} = 3$). (b) Chain-tree of length 2 ($\mathcal{N} = 3$).

of Theorem 4.2. Let $\mathcal{T} \in \mathbf{I}$ be some ground truth tree and let p be the shortest optimal (up to ε) path in it, with length l. $\mathcal{A}$ will consider at most $\frac{1}{\varepsilon}$ different paths during its run, one of them will be p. Each one of these paths is at most $l + \mathbf{k}$ nodes long (following the assumption that paths length varies by at most $\mathbf{k}$ nodes), thus in total the algorithm will visit at most $\frac{l+\mathbf{k}}{\varepsilon}$ nodes. On the other hand any other algorithm $\mathcal{B}$ will have to visit at least l nodes to discover and return p (or any other optimal path, since p is the shortest optimal path). Thus:

$$cost(\mathcal{A}, \mathcal{T}) \leq \frac{l + \mathbf{k}}{\varepsilon}$$
$$cost(\mathcal{B}, \mathcal{T}) \geq l$$

Combining this we get:

$$cost(\mathcal{A}, \mathcal{T}) \leq \frac{1}{\varepsilon} \cdot cost(\mathcal{B}, \mathcal{T}) + \frac{\mathbf{k}}{\varepsilon}$$

$\frac{\mathbf{k}}{\varepsilon}$ is a constant independent of the input, thus following the definition, Algorithm 2 is $\frac{1}{\varepsilon} - optimal$. □

And finally we prove that there is no deterministic algorithm that has optimality ratio better than $\frac{2}{\varepsilon}$. To do so, we will show that for every deterministic algorithm we can construct an input that will require from it to consider $\frac{2}{\varepsilon}$ different paths, while an optimal algorithm will have to consider only one path.

of Theorem 4.3. For the sake of the proof we will first define 2 special types of subtrees:

- A *Chain-tree* of length k is a subtree consisting of k nodes $u_1, \ldots, u_k$, where u_i is the only child of u_{i-1} for every $1 < i \leq k$. In this subtree $score(u_k) = score(u_1)$.
- A *Scattered-tree* of depth k is a subtree rooted in u of depth k where every node has exactly $\mathcal{N}$ children. Every leaf in this subtree has a score of $\frac{score(u)}{\mathcal{N}^k}$.

Figure 4.1 illustrates these types of subtrees.

Let's analyze the performance of $\mathcal{B}$ when running on a set of inputs $\{\mathcal{T}_x | x \geq 1\}$. A tree $\mathcal{T}_x$ will be constructed as follows:

- Let k be the largest integer such that $\left(\frac{1}{2}\right)^k > \varepsilon$

- Starting from the root, which is considered to be on level 0, every node on level $0 \leq i \leq k$ will have 2 children, every child will have choice count of $\lfloor \frac{\mathcal{N}}{2} \rfloor$ (If $\mathcal{N}$ is odd, additional child will be added to every node with choice count of 1 and a scattered- tree underneath it).
- After this, on the level k we will have c leaves, each one of them with a score of $s = \left(\frac{1}{2}\right)^k$.
- By the way of selection of k we ensured that $s > \varepsilon$ and $\frac{1}{2}s \leq \varepsilon$. Since all the leaves have the same score and their score sum up to 1 (Lemma 4.6) we have that $c \geq \frac{1}{2\varepsilon}$ and since c is an integer, $c \geq \lceil \frac{1}{2\varepsilon} \rceil$. Let's denote these leaves as $u_1 \dots u_c$.
- Under each one of the u_i's we will put a chain-tree of length M, where M is an arbitrarily large number, we will denote the end of each chain-tree as v_i
- Under each one of the v_i's except for v_x we will put a scattered- tree of depth T, such that every leaf will have a score less than $s - \varepsilon$, under v_x we will put a chain-tree of length T.
- Every leaf we have now will be labeled with ‡, making the tree complete

This tree has exactly one correct answer (under v_x). Now, suppose $\mathcal{B}$ does not consider a path under u_i during its run, for some i. Then, $\mathcal{T}_i$ is indistinguishable from $\mathcal{T}_x$ up until the discovery of u_i and the correct answer is under u_i, thus when running against $\mathcal{T}_i$, $\mathcal{B}$ will not discover a correct answer, contradicting the assumption that $\mathcal{B}$ is a correct algorithm. So $\mathcal{B}$ considers at least c different paths during its run. On the other hand, algorithm $\mathcal{B}_x$ that discovers all of the u_i's and then proceeds asking questions only about u_x can return after asking $\mathcal{N}$ questions on every node of the u_x chain-tree (since under each u_i all paths have a score of at most s, discovering one path with score s is enough to return a correct answer), thus considering only 1 path during its run. Length of every path in $\mathcal{T}_x$ tree equals $k + M + T$. $\mathcal{B}$ asks questions on at least $k + \frac{1}{2\varepsilon} \cdot M + T$ nodes, while $\mathcal{B}_x$ asks questions on $k + M + T$ nodes. Since M can be arbitrarily large the optimality ratio between $\mathcal{B}$ and $\mathcal{B}_x$ is at least $\frac{1}{2\varepsilon}$. □

Chapter 5

Natural extensions

There are two natural extensions that come to mind when solving a problem of finding a best result using crowd: finding top-k answers instead of just the top-1, and asking questions in bulk (since there might be several users ready to answer questions and we don't want them to waste their time). In this chapter we will discuss both these extensions - their effect on the model that we presented and the results that we achieved.

5.1 Finding top-k answers

First, we will discuss the extension of the problem to finding top-k answers. Recall Definitions 2.6 and 2.8 which define a top-k (up to an error) set of sequences and the proof of correctness for this set. Given these definitions we can define a generalized optimization problem as: *Find an approximated top-k set by asking as less questions as possible.* We assume in the sequel that $k < \frac{1}{\varepsilon}$, since otherwise the problem becomes trivial - every set of sequences of size k will be a top-k set.

Next, we will describe and present our algorithm for solving the top-k finding problem. This algorithm employs the same techniques as the algorithm presented in Section 3.2: asking questions about sequences with the highest potential, and asking questions on the highest possible nodes. To find the top-k set we will execute Algorithm 2 iteratively while saving the discovered tree. In the first iteration it will find the top-1 sequence (as proven in Section 3.2), in the second iteration it will find the second best sequence and so on - after k iterations we will have the top-k set (it will proven formally in the sequel).

For us to be able to perform several iterations of Algorithm 2 on the same tree (and not get the same result every time) we have to make it ignore the results of the previous iterations (this is relevant for both the stop condition and the selection of the next question). To do this we will place these results into a special set. This set, designated $PrevResults$, will hold the sequences returned by the previous iterations of Algorithm 2.

5.1.1 Generalized stop condition

The notion of uncertainty (Definition 3.1) has to be generalized to allow ignoring a given set of sequences (previous results in our case). Here is the formal generalized definition:

Definition 5.1 (Generalized Uncertainty)**.** *Given a set of complete sequences $PrevResults$, an uncertainty of a complete sequence p in a tree T is given by:*

$$U(T,p,PrevResults) = \max_{T' \in Compl(T)} \left[\max_{p' \in \mathbb{P}(T') \setminus PrevResults} score_{T'}(p') - score_{T'}(p) \right]$$

The algorithm for calculating the generalized uncertainty is almost identical to Algorithm 1, the only thing that is different is the set of nodes we test. Algorithm 3 is the modified algorithm, with the modifications highlighted:

Algorithm 3 Calculating $U(T,p,PrevResults)$

1: $Deltas \leftarrow \emptyset$
2: **for all** $\{p' | p' \in \mathcal{P}(T) \setminus PrevResults, p' \neq p\}$ **do**
   Assuming $p' = (u_1,\ldots,u_k,v_1,\ldots,v_m)$ *and*
   $p = (u_1,\ldots,u_k,w_1,\ldots,w_m)$
3: $\quad maxCommon \leftarrow \prod_{i=2}^{k} \frac{c_T(u_i)+\mathcal{N}-d_T(u_{i-1})}{\mathcal{N}}$
4: $\quad maxPPrime \leftarrow \prod_{i=1}^{n} \frac{c_T(v_i)+\mathcal{N}-d_T(u_{i-1})}{\mathcal{N}}$
5: $\quad minP \leftarrow \prod_{i=1}^{m} \frac{c_T(w_i)}{\mathcal{N}}$
6: $\quad \delta \leftarrow maxCommon \cdot (maxPPrime - minP)$
7: $\quad Deltas \leftarrow Deltas \cup \{\delta\}$
8: **end for**
9: **return** $\max_{\delta \in Deltas} \delta$

Since only the search domain was changed and not the algorithm operation, the proof of Theorem 3.3 works for this algorithm also.

Following definitions 5.1 and 2.8 we can use the modified notion of uncertainty to check whether a decision tree is a proof of correctness for a set of sequences in it:

Lemma 5.2. *A decision tree T is* a proof of a set A being a top-k set *iff for every $p \in A$:*

$$U(T,p,A \setminus \{p\}) \leq \varepsilon$$

5.1.2 Deciding what question to ask

As we said before, we will employ the same two techniques to decide what question to ask next as we used in Algorithm 2: asking questions on the most promising sequence, and asking questions as high as possible in the tree. However the search domain for the most promising sequence has to modified to exclude sequences from $PrevResults$. In Algorithm 2 we go over all the leaves of the discovered tree (since we always ask questions on the highest possible

node, leaves describe all possible single-step continuations of T). In the generalized algorithm we would like to exclude leaves from $PrevResults$ from the search. However, this is not enough. Indeed, if a leaf from $PrevResults$ is a single child on node v in T and node v is not exhausted yet (more questions can be asked on it), then there is another possible single-step continuation of T described by v, and thus we should include it in our search space. We call such nodes *quasi-leafs*.

Definition 5.3. *Given a tree T and a set of leafs A in it, an internal node v in T is a* quasi-leaf *if $d_T(v) < \mathcal{N}$ and all of v's children are in A.*

5.1.3 Algorithm for finding top-k sequences

The algorithm for finding a set of top-k sequences will execute a generalized version of Algorithm 2 for k iterations. Each new iteration will use the results of the previous ones. Algorithm 4 is the generalized version of Algorithm 2 and it is presented below, the differences from the original algorithm are highlighted.

Algorithm 4 FindTopKBase: Finding the optimal sequence, while ignoring previous results

Input: $T, PrevResults$
Output: $TopSequence, T$

1: $i \leftarrow 0$
2: **while** $\mathbb{P}(T) = \emptyset$ OR $\min_{p \in \mathbb{P}(T) \setminus PrevResults} U(p, T, PrevResults) \geq \varepsilon$ **do**
3: $allLeafs \leftarrow Leafs(T) \cup QuasiLeafs(T, PrevResults)$
4: $bestLeaf \leftarrow \text{argmax}_{p \in allLeafs} M(p)$
5: $Candidates \leftarrow \{bestLeaf\}$
6: $TopNodes \leftarrow \{tN(bestLeaf)\}$
7: $maxScore \leftarrow M(bestLeaf)$
8: **if** $maxScore > \varepsilon$ **then**
9: **for all** $p \in allLeafs$ **do**
10: **if** $M(p) = maxScore$ **then**
11: **if** $tN(p) \notin TopNodes$ **then**
12: $Candidates \leftarrow Candidates \cup \{p\}$
13: $TopNodes \leftarrow TopNodes \cup \{tN(p)\}$
14: **end if**
15: **end if**
16: **end for**
17: $p \leftarrow Candidates[i \mod |Candidates|]$
18: Ask a question on $tN(p)$
19: $i \leftarrow (i+1) \mod |Candidates|$
20: **else**
21: $p \leftarrow Candidates[0]$
22: Ask a question on lowest node of p
23: **end if**
24: **end while**
25: **return** $\text{argmin}_{p \in \mathbb{P}(T) \setminus PrevResults} U(p, T, PrevResults), T$

If there is more than one maximum item, argmax *shall return one at random*

Finally, the algorithm for finding a set of top-k sequences is presented below:

Algorithm 5 Finding a set of top-k sequences

```
1: T ← Origin
2: PrevResults ← ∅
3: for 0 ≤ i < k do
4:     p, T ← FindTopKBase(T, PrevResults)
5:     PrevResults ← PrevResults ∪ {p}
6: end for
7: return PrevResults
```

5.1.4 Efficiency and optimality

In this subsection we will show that the results we proved in chapter 4 regarding the efficiency and the optimality of our algorithm for finding a top-1 sequence are also true for the top-k extension. To do so, we will show that Algorithm 5 considers at most $\frac{1}{\varepsilon}$ different sequences (just like our original algorithm), while any other algorithm for finding a set of top-k sequences will have to consider at least k different sequences.

Theorem 5.4. *Algorithm 5 considers at most $\frac{1}{\varepsilon}$ different sequences during it's run.*

Proof. Algorithm 5 executes a variant of Algorithm 2. This variant has the same complexity (indeed, the only change is the reduction of search space for the next question). In addition all the executions of the algorithm share the discovered tree, and thus they share a *pool of candidates* (recall from the proof of theorem 4.4, a pool of candidates is a set of sequences that can be considered by the algorithm). As we shown in the proof of theorem 4.4, the size of this pool is limited by $\frac{1}{\varepsilon}$, and thus at most $\frac{1}{\varepsilon}$ different sequences will be considered during the executions of Algorithm 4 on the same tree. □

Since any other algorithm has to consider at least k different sequences in order to return a set of top-k answer, one may expect that the optimality ratio of Algorithm 5 is at most $\frac{k}{\varepsilon}$. Unfortunately, this is not true, since the k different sequences that some algorithm has to consider may have many nodes in common, hence finding the top-k answers does not necessarily incur factor k penalty in performance. Thus, proven upper bound for the optimality ratio of Algorithm 5 is $\frac{1}{\varepsilon}$ (the same bound we have provenfor the original algorithm).

In Chapter 6 we will show that in practice looking for top-k sequence instead of top-1 does not incur a severe penalty in required questions count, moreover the performance of our algorithm (relative to baseline algorithms) improves as k grows.

5.2 Asking questions in bulk

Up until now we discussed how to select a next question to ask under the assumption that the questions are asked one at a time. However, in a real-life applications there usually will be multiple users working with the system

simultaneously. This requires issuing multiple questions in parallel. In this section we will discuss an extensions to the algorithm we have developed that will allow to do so. First, we present the problem of asking questions in parallel in detail, then we will present several heuristic approaches to its solution. Formal reasoning about asking questions in parallel will require modeling the interaction of the users with the system (how many users are online at each moment) and is out of scope of this work. We will, however, show empirically that our heuristics work very well in practice.

In general, the problem of asking questions in parallel can be formulated as follows: *Given a system state S and a budget of B questions, assign as much questions as possible (out of the budget). The goal is to reach a final system state while minimizing the total number of asked questions.* In our model, the system state is the decision tree discovered so far and the final state of the system is a proof of correctness. Note that assigning exactly B questions is not always possible (for example, in our model when the decision tree consists only of the root, we can assign at most $\mathcal{N}$ questions).

In our model, questions may be assigned to nodes in a current discovered decision tree that are not exhausted yet. Such nodes can be ranked by some criteria, and then there are two general approaches to assigning the questions:

1. Assign as much questions as possible on the highest ranked node, then move to the second highest rank and so on. This is called the "deep" approach.

2. Distribute questions evenly on all available nodes. This is called the "broad" approach.

We implemented both of these approaches and compared their performance empirically. The evaluation results will be discussed in Chapter 6.

The implementation was done by modifying Algorithm 4. Instead of finding one most promising node, the modified algorithm will return a list of all nodes that we can ask questions on, sorted by potential. Questions are assign to some of these nodes basing on the approach (either "deep" or "broad"). The list of nodes is built using the following method:

1. Order all leafs and quasi-leafs by max potential score in descending order.

2. Replace every leaf in the resulting list with its top-node (while removing duplicates).

3. Throw out all nodes with a max potential score less than ε

5.2.1 Optimality

Even though when asking questions in bulk we will inevitably have to ask more unneeded questions, we will show in this subsection that we will still consider at most $\frac{1}{\varepsilon}$ different sequences, and thus optimality results from Chapter 4 hold.

Theorem 5.5. *Both approaches to asking questions in bulk lead to considering at most $\frac{1}{\varepsilon}$ different sequences.*

Proof. We only consider nodes that have a maximum potential score of at least ε (step 3), thus only nodes that are members of the pool are considered (recall from the proof of theorem 4.4, the "pool" is the set of all nodes with a max potential score of at least ε). And as was shown in the proof of theorem 4.4 at most $\frac{1}{\varepsilon}$ different paths can have nodes in the pool during the run of the algorithm. □

Chapter 6

Experimental Evaluation

In this chapter we will present the results of the experimental evaluation of our algorithm. During the evaluation we explored its behavior on different data sets (both synthetic and real) with different parameters. We also explored the effect of the algorithms parameters (such as allowed error, batch size and the required number of answers per node). We compared our algorithm to several baseline algorithms.

6.1 Evaluation setup

To conduct the experiments we have implemented CrowdPlanr in C# and PHP while using MySQL as the database engine. Its architecture is presented in Figure 6.1. For the evaluation purposes the User Interface was replaced with a simulator (called *the Oracle* in the sequel) that returned answers to queries either from a synthetically generated dataset or a dataset recorded from the interaction with real users. Other parts of the system are: *Plan Builder* which executes the algorithm for finding the optimal sequence, *Crowd Manager* which formulates the questions for users (by preparing a set of possible answers) and *Database* which holds all the information gathered by the system.

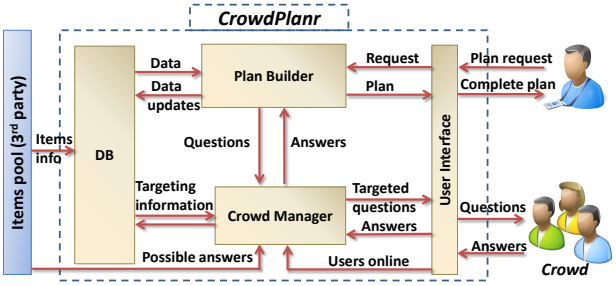


Figure 6.1: CrowdPlanr architecture

Algorithms In our experiments we compared the number of visited nodes (and the number of questions asked) by our algorithm (as defined in Chapter 3, from now on it will called CrowdPlanr) to the number of nodes visited by the

baseline algorithms running on the same input (since the problem presented in this work has not been studied before, we can't compare our algorithm to other solutions). The baseline algorithms that were considered are:

- *Random* - a naïve algorithm that randomly chooses which question to ask next, from all possible questions (nodes of the tree that are not exhausted yet). This algorithm showed extremely poor performance (asked significantly more questions than all other algorithm), and thus we will not include this algorithm in our comparisons
- *Greedy* - an algorithm that employs a trivial greedy approach: ask a question on a sequence that currently has the maximal score, try to extend it as much as possible (i.e. ask a question on a lowest node possible of the selected sequence).
- *CrowdPlanr*$^-$ - compared to the greedy, our algorithm is different in two ways, first we choose a sequence with the highest potential score (not the highest current score) and second we ask questions on a highest node possible (not always trying to extend the selected sequence). CrowdPlanr$^-$ is an algorithm that is half-way between the *Greedy* and the CrowdPlanr algorithms: it selects a sequence with the highest current score (like the *Greedy*) and asks questions on a highest node possible of that sequence (like the CrowdPlanr).

The halting condition for all the algorithms is the same: they can return a set of sequences only if the tree they had discovered so far forms a proof of correctness (recall Definition 2.8 for that set. The algorithms use uncertainty calculation we presented in Chapter 3 to check this condition.

Synthetic Data For evaluating the effect of various properties of the input on the number of questions asked by the algorithms we generated a synthetic datasets simulating an input with desired properties. These datasets were represented by complete trees accessible by *the Oracle*. In each experiment we changed only one property of the input while all the others had a default value. We considered the following properties of the input:

- *Tree depth* is ranged from 5 to 10, with default value of 7. Trees that have more than 10 levels are less important for two main reasons: first, human factor reason, in the real world scenarios it is hard for the user from the crowd to hold in her mind more than 10 items as a context and give a good recommendation for continuation of the sequence (usually when one wants to plan a longer sequence, she will do the planning on different granularity levels). Second, since we use probabilities and the final score of the answer is a multiplication of the probabilities of the nodes, for sequences longer than 10 nodes the scores get very small in our setup and hence all the sequences will be optimal up to ε (Definition 2.6).
- *Number of possible descendants* of each node is a parameter that is responsible for the width of the tree. We range values for this parameter from 2 to $\frac{\mathcal{N}}{2}$, with a default value of 5. We saw in our experiments that this parameter does not affect the number of questions asked by the algorithms, and thus we will not discuss these experiments in detail.

- *Depth difference (***k***)* is the difference in levels between the highest and the lowest leaf in the tree. This property shows the balance of the tree. The default value is 0, which means that all the sequences have the same length. We also ranged **k** values up to $\frac{TreeDepth}{3}$. Our experiments showed that this parameter also does not affect the number of questions asked by the algorithms, thus we will not discuss these experiments in detail.

- *Skewness* is a percent of votes that go in favor a specific child of each node in tree (for example if the *Skewness* is 60% then for every node that will be a child that gets 60% of the votes). The default value for skewness is 60%, we also checked skewness of 50% and 70%.

Real Data To ensure that our algorithm performs well in real life we evaluated the number of questions asked by it (and its baseline competitors) on two datasets coming from different real-world applications:

1. A Large dataset containing a record of 20,000 vacation trips in Europe. The trips included 10 different cities and were approximately of the same length (in terms of visited cities). This dataset was obtained from a traveling agency, we omit its name for privacy reasons.

2. A Medium size dataset containing answers to a question "In which order to watch Star Wars films?". It was obtained by comparing the popularity of the proposed orders on various web sites. This question is asked frequently on the internet and has 100,000,000 results in web search engines. It has 6! (=720) possble answers (some of them, of course, completely wrong). An important property of this dataset is that there are only small number of "good" orders, while others have very little support.

All the datasets were initially contained the ranking of the sequences and were translated into a complete tree that was used by the *Oracle* to answer queries. We believe this is a good approximation of a real-life interaction with the users, because we assume that the users know the rating of a complete sequence and thus their rating of the partial sequence (i.e. the answer to our questions) will be consistent with it.

Algorithm parameters In addition to the properties of the input we evaluated how the parameters of the algorithm affect the number of nodes visited by it (and the number of questions asked by it). We evaluated the effect of the following parameters:

- *Allowed error (ε)* The default value of an allowed error in our experimens was 0.01, in addition we ranged it from 0.002 to 0.1.

- *The number of sequences in the answer (k)* By default we were looking for the single best sequence. In addition we ranged k from 1 to 10.

- *Number of questions per node ($\mathcal{N}$)* is chosen based on a statistical data, reliability of the crowd and budget constraints. We use a default value of 10, but also run experiments with $\mathcal{N} = 50$ and $\mathcal{N} = 100$.

- *The number of questions asked in parallel (batch size)* By default we let the algorithms ask one question at a time. In addition we ranged the batch size from 1 to 50.

All the parameters (both of the input and of the algorithms) used in our experiments are summarized in Table 6.1.

Parameter	*Default value*	*Tested values*
Tree depth	7	5 - 10
Questions per node ($\mathcal{N}$)	10	10, 50, 100
Data skewness	60%	50%, 60%, 70%
Maximum children per node	5	5
Depth difference	0	0
Allowed error (ε)	0.01	0.002 - 0.1
k of Top-k	1	1-10
Batch size	1	5-50

Table 6.1: Summary of the parameters

6.2 Evaluation results

In every experiment for each of the algorithms we measured the number of nodes visited (node is considered visited if there was at least one question asked about it) and the total number of questions asked. For each experiment we executed the algorithms on 3 datasets with the same properties and averaged the results.

The experiments were modeled in the following way: generated datasets were read by the *Oracle* and all the questions asked by the algorithms were redirected to the *Oracle* which answered them basing on an input data set in a deterministic way (i.e. several run using the same oracle would yield the same results). The *Oracle* also collected statistics about asked questions from which we derived the results of the experiments.

6.2.1 Effects of the data parameters

In this set of experiments we evaluated the effect of the parameters of the input data on the number of questions asked by each of the algorithms. For this we used synthetic data sets. We also verified achieved results on a real-world datasets.

Varying the depth of the tree In this experiment we evaluated the effect of a tree depth. All other parameters remained default (see Table 6.1). The results are summarized in Figure 6.2.

Note that the Y axis has a logarithmic scale and it represents the number of questions that were asked. The X axis represents the depth of the tree. From Figure 6.2 we can see that the *Greedy* algorithm performs significantly worse than both the CrowdPlanr and the CrowdPlanr$^-$ algorithms. Also, we can see that the difference between the CrowdPlanr and the CrowdPlanr$^-$ grows with the depth of the tree.

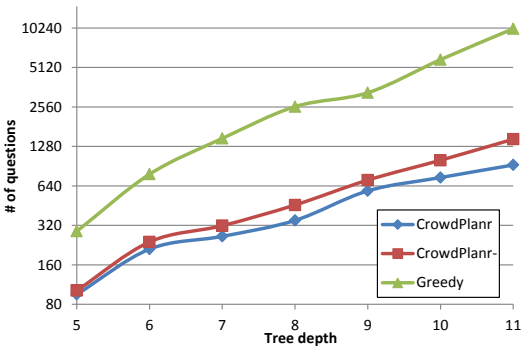


Figure 6.2: Varying the tree depth

Varying the data skewness In this experiment we examined the effect of data skewness on the number of questions asked by the algorithms. We tested skewness levels of 50%, 60% and 70%. All other parameters had default values (see Table 6.1). The results of the experiment are presented in Figure 6.3.

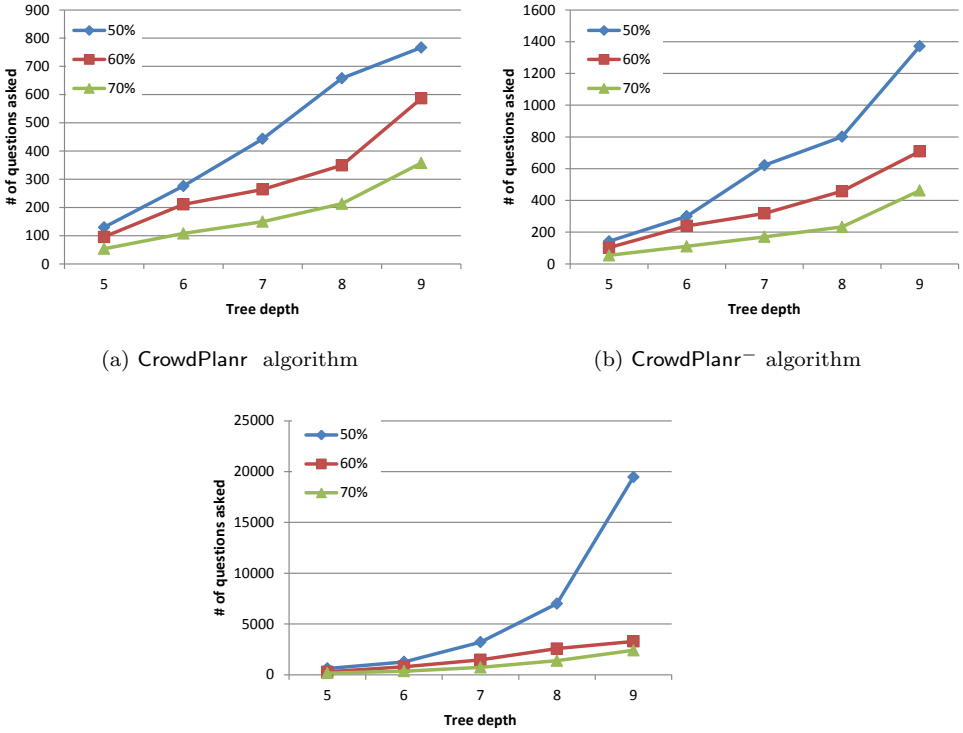
As can be seen in this graph, the more data is skewed the easier it is for the algorithm to find a correct answer. The most prominent effect skewness has on the *Greedy* algorithm.

Varying the number of questions per node In this experiment we evaluated the impact of different values for $\mathcal{N}$ (10, 25, 50, 100) on the number of nodes visited by the algorithms (the total number of questions is less interesting since it is expected to be linearly dependent on $\mathcal{N}$). All other parameters had a default value (see Table 6.1). Theoretically we have proven that our algorithm is instance-optimal and its optimality ratio does not depend on $\mathcal{N}$, that means that CrowdPlanr algorithm is expected to visit the same number of nodes for any value of $\mathcal{N}$. In practice it means that CrowdPlanr can be used for different approaches with different audience, crowd size and system needs. Figure 6.4 shows that the reality meets the expectation.

As we can see number of visited nodes is almost constant for various values of $\mathcal{N}$. Note that Y axis has a logarithmic scale. This empirically proves that the number of nodes visited by the CrowdPlanr is independent of the value of $\mathcal{N}$.

Real world data In this experiment we analyzed the number of nodes visited by each one of the algorithms when executed on a real-world data sets. The parameters of the algorithms were set to default values (see Table 6.1). The experiment was performed on two real-world data sets: trip planning and star wars watching order. The results are shown in Figure 6.5.

Here the Y axis represents the number of visited nodes. As we can see, the CrowdPlanr algorithm is slightly better than the CrowdPlanr$^-$ algorithm and both are significtanly better than the *Greedy* algorithm. We conclude from this experiment that the behavior of all the algorithms seems to be the same as on synthetic data.



(a) CrowdPlanr algorithm

(b) CrowdPlanr$^-$ algorithm

(c) *Greedy* algorithm

Figure 6.3: The effect of the data skewness

6.2.2 Effects of the algorithm parameters

In this set of experiments we examined the effect of different parameters of the algorithm (like the required number of ansers or the allowed error) on the number of questions asked by the algorithm and the number of nodes visited by the algorithm. We performed these experiments both on the real and synthetic data and the results were the same. The results presented below were achieved from the synthetic data.

Varying allowed error In this experiment we evaluated the effect of the allowed error value on the number of questions asked by the algorithms. For this we left all the parameters with default values (see Table 6.1) and ranged ε from 0.01 to 0.1. The results are summarized in Figure 6.6.

Here again, the Y axis has a logarithmic scale and represents the number of questions asked by the algorithm. The X axis represents the value of ε. We can see that the effect of the allowed error is much larger on the *Greedy* algorithm than on the other two. One possible explanation to this is that the Greedy strategy causes the algorithm to "jump" all over tree without really reducing the uncertainty. Increasing the allowed error sets the uncertainty bar lower, thus saving the algorithm work. Another interesting property of this graph is that the number of questions asked by the CrowdPlanr$^-$ algorithm decreases

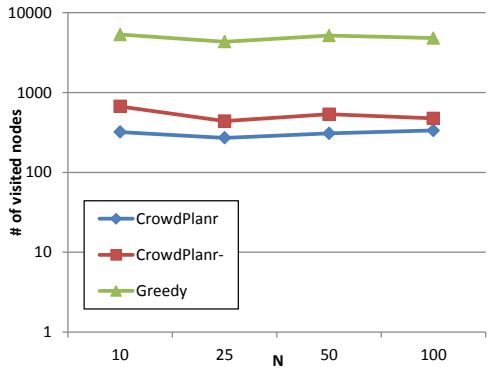


Figure 6.4: Visited nodes when varying $\mathcal{N}$

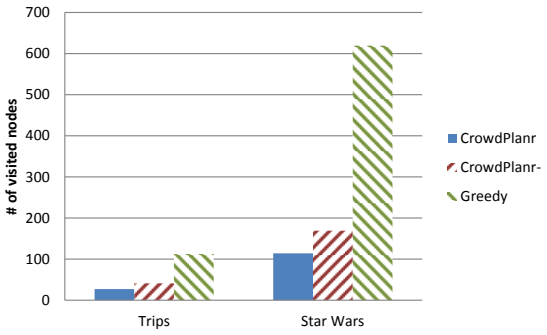


Figure 6.5: Real-world datasets

relatively slow with the increase of ε. A possible explanation for this can be that the selection criteria of the sequence to work on is wrong, which causes the algorithm to ask questions on a wrong sequence, and since it asks questions on a highest possible node, the mistake is not revealed fast enough.

Finding top-k answers It is expected that as we increase the k (the number of desired answers) the more questions we will have to ask (and more nodes we will have to visit) in order to find the top-k set. However, when we are looking for the second best sequence we can use the information we discovered while we were looking for the top-1 sequence, thus the growth in the required questions count is expected to be sub-linear. For the next experiment we fixed the allowed error to be $\varepsilon = 0.002$, ranged k from 1 to 10 and all other parameters were set to default values (see Table 6.1). The results are shown in Figure 6.7.

The results match the expectation. In addition, the graph in Figure 6.7 displays several interesting properties. First, the advantage of our algorithm over the baseline algorithms[1] grows with k. Second, the number of additional questions we have to ask decreases with k. This can be explained by the fact

[1]Recall that CrowdPlanr$^-$ uses only one heuristic from CrowdPlanr: asking questions on the highest available node, while *Greedy* always tries to go with the sequence that currently has the highest score

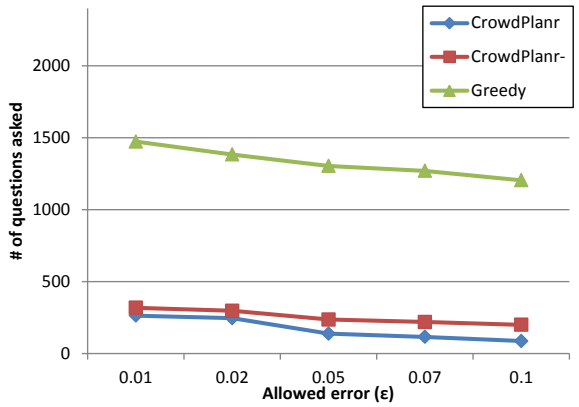
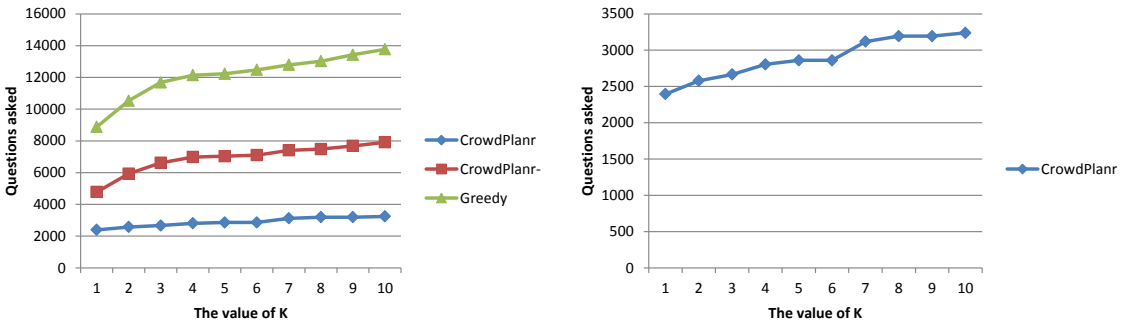


Figure 6.6: Varying ε



(a) All algorithms compared

(b) Zoom into CrowdPlanr behavior

Figure 6.7: Number of questions as a function of K

that the number of sequences with a score higher than ε is relatively small, and thus with k big enough it doesn't matter what sequence we add to the set.

This behavior is further explored in Figure 6.8. This figure contains several graphs, each graph describes the number of nodes that CrowdPlanr algorithm had to visit in order to find a top-k set for a ceratin value of k as a function of ε (all other parameters are, again, set to default values). All the graphs in this figure are declining and converge to a certain point as the value of ε increases. This demonstrates once again that when the number of sequences with a score greater than ε is less than the desired k, then it becomes an easy task to find the top-k set - just find all the sequences with scores above ε and fill the rest of the set with any other sequences.

Asking questions in parallel As explained in Section 5.2 there are two general approaches to asking questions in parallel given a current state of a decision tree:

- Sort nodes by their "potential" (the definition of "potential" depends on the algorithm) and then divide the questions evenly between the suitable

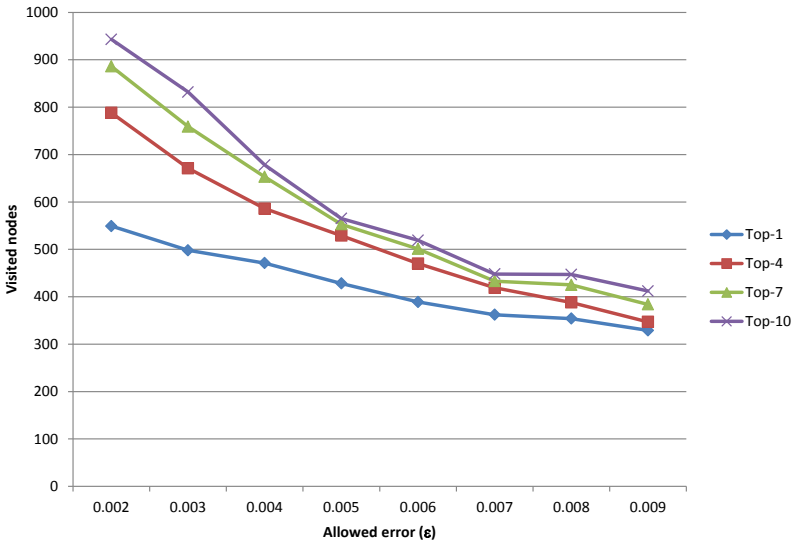


Figure 6.8: Number of questions as a function of ε

nodes (nodes that have a maximum potential score of at least ε). We call this approach *a broad approach*

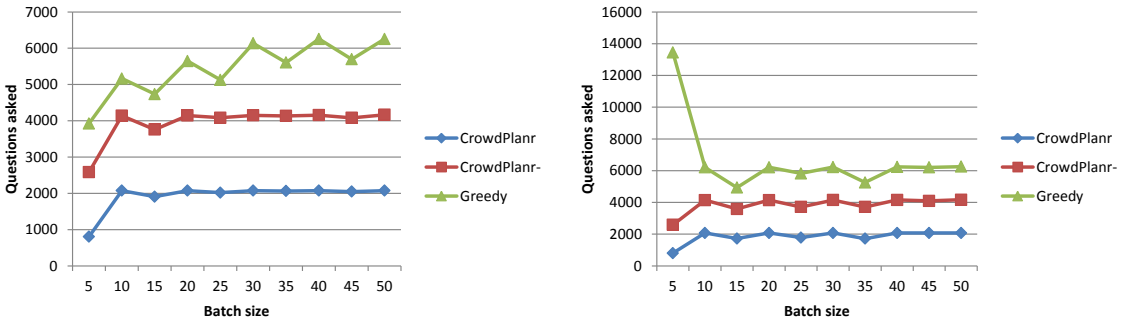
- Ask as many questions as possible on a node with the highest "potential", then move to the second best and so on. We call this approach *a deep approach*

We implemented both these approaches for our CrowdPlanralgorithm and for every one of our baseline algorithms (CrowdPlanr$^-$ and *Greedy*) as well. We then compared their performance while limiting the batch size (the number of questions asked simultaneously). For this experiment the allowed error was fixed to be $\varepsilon = 0.03$ and the batch size was ranged from 5 to 50, all other parameters were set to default values (see Table 6.1). The results of the experiment are presented in Figure 6.9.

As can be seen in this figure, the behavior of CrowdPlanr and CrowdPlanr$^-$ is very similar - the total number of questions raises with the batch size, until the batch size becomes greater than the questions limit, then the total number of questions remains almost constant. CrowdPlanr algorithm is almost twice more efficient than the CrowdPlanr$^-$ algorithm, while *Greedy* algorithm shows somewhat erratic behavior. The behavior of "deep" and "broad" approaches seems to be almost identical, however there is a difference. To see it, we put each algorithm on a different graph (Figure 6.10).

The behavior of the "broad" approach is more stable, but in general requires more questions, while "deep" approach can save questions in some settings. This may suggest that our method for ranking the "potential" of the node is correct, and thus when we concentrate on the nodes with the highest potential we ask less questions in total. This claim is indirectly supported by the behavior of the *Greedy* algorithm, whose ranking method is clearly wrong (see Figure 6.11).

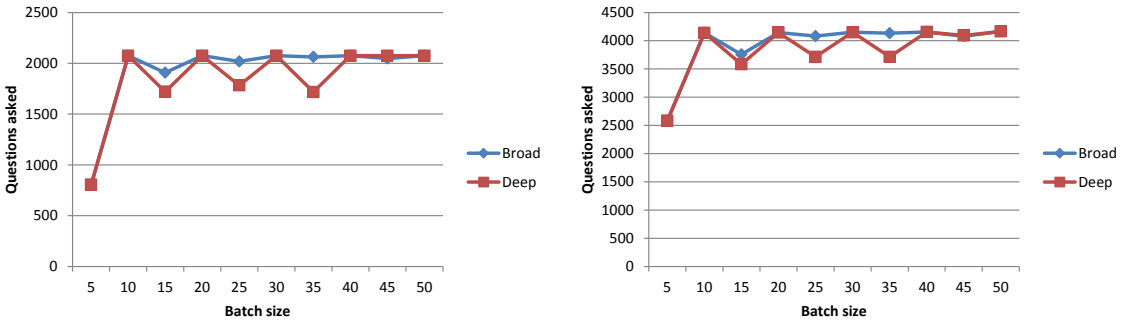
For the *Greedy* algorithm the "deep" behavior is worse than the "broad" one (especially for small batch sizes), this suggests incorrect nodes ranking method.



(a) Broad approach

(b) Deep approach

Figure 6.9: Asking questions in parallel



(a) CrowdPlanr algorithm

(b) CrowdPlanr$^-$ algorithm

Figure 6.10: Comparing "Broad" and "Deep" approaches

6.2.3 Additional experiments

In our last experiment we explored the behavior of all of the algorithms - how many questions were asked (or nodes visited) on every level of a decision tree. This tells us how "focused" each one of the algorithms is. For this experiment, we fixed the tree depth to be 10 and left all other paramters to have default values (see Table 6.1). The results are presented in Figure 6.12.

In this graph the X axis represents the level in the tree (0 represents the root, 9 represent leafs) and Y axis represents the normalized number of nodes discovered by the algorithms (the normalization is done by dividing the result of each algorithm by the result of the worst algorithm). This can show how many unnecessary nodes were visited. In this example we compare CrowdPlanr$^-$ and CrowdPlanr, as you can see the ratio of visited node linearly decreasing with the depth of the tree. On the first three levels the both algorithms show the same performance and the performance of CrowdPlanrimproves as we go deeper into the tree. This means that CrowdPlanrbetter utilizes the information recieved from the crowd and asks less questions in the long run. When we measured asked questions instead of visited nodes we got very similar similar results and

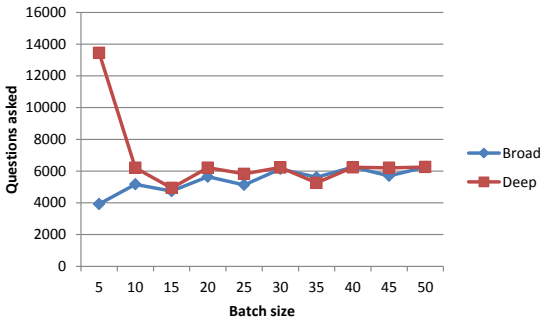
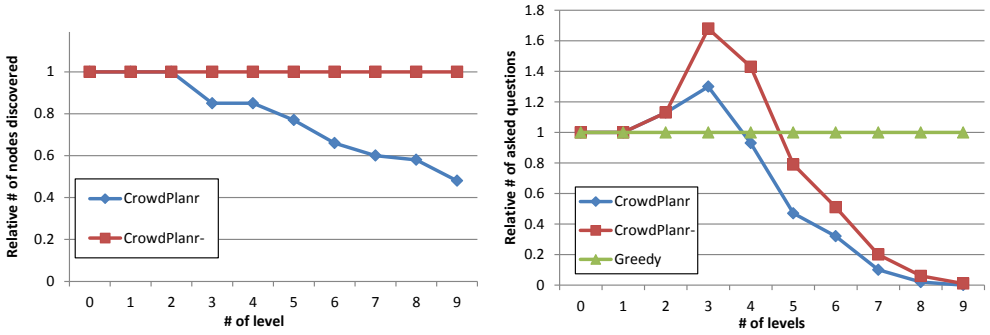


Figure 6.11: Comparing "Broad" and "Deep" approaches for the *Greedy* algorithm



(a) CrowdPlanr vs. CrowdPlanr$^-$ (b) CrowdPlanr, CrowdPlanr$^-$ vs. *Greedy*

Figure 6.12: Nodes discovered on each level

hence we omit them here. When CrowdPlanr and CrowdPlanr$^-$ are compared to the *Greedy* we can see that on the higher levels *Greedy* performs better than CrowdPlanr and CrowdPlanr$^-$. In this case the greedy strategy to go after the local maximum seems to be a good choice, however starting from the level 4, *Greedy*'s performance gets dramatically worse.

6.3 Summary

First, we have empirically shown that our algorithm CrowdPlanr has a better performance than the base-line algorithms in both number of asked questions and the number of visited nodes. Next, we have demonstrated that CrowdPlanr performs good on both synthetic and real-world data of different sizes and types. In addition, we have shown that as we go deeper into the decision tree the benefits of CrowdPlanr and CrowdPlanr$^-$ over a greedy approach become more evident. We also have studied the impact of the different input data parameters like the depth of a tree, skewness of the data and number of questions per node. Finally, we have shown that extending our algorithm to

finding the set of top-k answers or asking many questions in parallel does not incur heavy penalty in the total number of asked questions. For both these extensions our algorithm significantly outperforms basic baseline algorithms.

Chapter 7

Related Work

Since the emergence of the Web 2.0 using the crowd as a source of information has become common in the Internet. In the beginning the information was provided directly by the users (e.g. Wikipedia [1] and other user-generated sites). In the recent years much research has been conducted on more sophisticated ways to extract bit of knowndelge from the crowd. In particular a *crowd computing* paradigm has emerged to solve problems that are computationally hard to solve but are easy for humans [10]. The planning problem that we consider here is such an instance, as the goal, i.e, the notion of "best output", is hard to formalize. Another examples of such problems are pattern recognition and image tagging.

There are several approaches to extracting information from the crowd. One approach suggests using games (e.g. [9], [40], [19], [41], [27]), while another approach suggests extracting information from the interaction of users with a certain system (e.g. [42]). Also, there is a more direct approach - paying the users to perform small and easy tasks the results of which can be used in computations. Several platforms exist that connect users willing to perform such micro-tasks with task providers, the most popular such platform is the Amazon Turk ([30]). Finally, there are attemps [4] to use social services, like Facebook and Twitter, that are getting more and more popular in the last years as means to access the crowd.

Many applications has been developed that take advantage of the Amazon Turk platform. Instead of answering all requests with computer algorithms, some human-expert tasks are published on crowdsourcing platforms for human workers to process. Typical tasks include image annotation ([35]), information retrieval ([16]) and natural language processing ([23]). These are tasks that even state-of-the-art technologies cannot accomplish with satisfactory accuracy, but could be easily and correctly performed by humans.

Others considered the development of a unified model to allow uniform data collection from both humans and machines [32]. In particular, research has been directed in the Databases community to development of DB systems that allow to specify which parts of the data should be crowdsourced (e.g. CrowDB [13], Deco [33], Qurk [28]). These systems provide declarative language support and ability to define what data will be retrieved from the crowd, and further allows to employ the crowd as data-processing units. Crowdsourcing was also suggested as method for data cleaning, integration and analytics, entity resolution, schema

expansion (e.g. [18], [36], [43], [25]).

There are two main concerns when working with the crowd - how to minimize the number of questions (or tasks) one needs to ask the crowd (the are two reasons for this concern - time required to obtain the answers and the cost of payments to the workers) and how to validate the quality of the received results (studies has shown that users exhibit different behaviors in micro-task markets [22], they can provide incorrect answers or even lie dilebirately).

The minimization of the cost (measured in terms of the number of questions that are posed to the crowd) and of the expected error are important goals in crowd-based query processing ([3], [31]). The optimal choice of questions to pose to the crowd has also been considered in [3] to reduce the uncertainty/error in aggregation functions over crowd answers. Here again a key difference from our work is the independence assumption among the aggregated data items. The dependency exhibited in planning problems requires the development of corresponding (different) uncertainty measures, and consequently different algorithms. For a specific applications more focused approaches are developed, for example [8] suggests a hybrid approach to entity linking problem - initial rough processing is done by a computer and the crowd is then used dealing with hard cases where the computer did not provide a decisive answer.

Not all workers produce results of the same quality, estimating the reliability and using it to asses the quality of the data is an additional research direction. In [25] validation questions (questions for which the system knows the correct answers) are asked along with the real ones, others try to rank the qualities of the workers [21] or accurately identify abusive content [15]. An interesting observation was made in [29] - increasing the price of a single task does not help getting results of a better quality.

Closest to our work are the works that consider max and top-k query processing with the crowds, that involve ordering of the query results using the crowd. For example, the problem of finding the maximum element has been investigated in [18] that considers two problems: the *judgment problem*, that defines how given a set of comparison results one determines an element which is most likely to be the maximum, and the *next vote problem*, that determines which future comparisons will be most effective. Another example is [39] that provides efficient heuristics that can be tuned based on parameters like execution time, cost and quality of result. A key difference between max/top-k query processing and ours is the *inherent dependency that exists between the items in the plan*. Unlike max/top-k processing where users can be asked to compare pairs of individual elements, planning requires a global view of the (preceding sub-)plan, and its possible/ideal completions.

An adjacent to crowd-sourcing is a field of active learning [37], which also studies the problem of requesting human input. The goal is to request input from an expert in order to enrich the training data set for a specific machine-learning task.

Crowdsourcing attracted also interest from the AI community with research aiming at dynamic workflow executions that optimally use the crowd for accomplishing a given complex task (e.g. [6], [24]). This is complementary to our work where users are used to identify and order the items (potentially the to-be-executed workflow components) needed to best accomplish an informally specified goal.

Chapter 8

Conclusion

In this paper we propose to use the power of the crowd for answering planning queries, when the goal, i.e, the notion of best plan, is hard to formalize. We introduce a simple generic model for modeling plans and for interpreting crowd's answers to questions about them. Based on this model, we present an effective algorithm for identifying the (approximately) best answer using the crowd. The algorithm builds the desired plans incrementally, choosing at each step the best questions so that the overall number of questions that need to be asked is minimized. We prove the algorithm to be instance-optimal for a large common class of planning queries and data instances, showing that the optimality ratio that it achieves is the best possible, and demonstrate experimentally the algorithm's effectiveness and efficiency.

We focused here on identifying the (approximated) best plan. More generally, one may want to identify top-k best answers. Our algorithm naturally generalizes to this context by continuing the execution after a top-1 sequence is found. Intuitively, nodes that are part of the returned sequence should be marked in the tree and ignored when candidates are considered. An interesting challenge for future research is identifying heuristics that can be applied when some prior knowledge about the expected answer distribution or tree structure is available. How to obtain such information is also an interesting questions. Prior knowledge on users may also be used for weighting answers non-informally and for targeting questions to the most appropriate users.

Another possible extension to our algorithm, to be considered in the future research, could be to allow creating plans not necessarily in a successive order (for example when parts of the plan are known and one wants to use the crowd to fill in the gaps).

Bibliography

[1] Wikipedia. http://www.wikipedia.org/.

[2] G. Adomavicius and A. Tuzhilin. Towards the next generation of recommender systems. *IEEE TKDE*, 2005.

[3] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W.-C. Tan. Asking the right questions in crowd data sourcing. In *ICDE*, pages 1261–1264, 2012.

[4] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 1009–1018, New York, NY, USA, 2012. ACM.

[5] Courserank. http://courserank.stanford.edu/.

[6] P. Dai, Mausam, and D. S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI*, 2010.

[7] S. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, 2013.

[8] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Zencrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 469–478, New York, NY, USA, 2012. ACM.

[9] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Declarative platform for data sourcing games. In *WWW*, pages 779–788, 2012.

[10] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, 2011.

[11] G. C. et al. A cross-service travel engine for trip planning. In *SIGMOD*, pages 1251–1254, 2011.

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[13] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, 2011.

[14] M. Ghallab, D. S. Nau, and P. Traverso. *Automated planning - theory and practice*. Elsevier, 2004.

[15] A. Ghosh, S. Kale, and P. McAfee. Who moderates the moderators?: crowdsourcing abuse detection in user-generated content. In *Proceedings of the 12th ACM conference on Electronic commerce*, EC '11, pages 167–176, New York, NY, USA, 2011. ACM.

[16] C. Grady and M. Lease. Crowdsourcing document relevance assessment with mechanical turk. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, CSLDAMT '10, pages 172–179, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[17] R. Groves, F. J. Fowler, M. Couper, J. Lepkowski, E. Singer, and R. Tourangeau. *Survey Methodology*. John Wiley and Sons, 2009.

[18] S. Guo, A. G. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD Conference*, pages 385–396, 2012.

[19] S. Hacker and L. von Ahn. Matchin: eliciting user preferences with an online game. In *CHI*, pages 1207–1216, 2009.

[20] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 1968.

[21] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 64–67, New York, NY, USA, 2010. ACM.

[22] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 453–456, New York, NY, USA, 2008. ACM.

[23] J. Ledlie, B. Odero, E. Minkov, I. Kiss, and J. Polifroni. Crowd translator: on building localized speech recognizers through micropayments. *Operating Systems Review*, 43(4):84–89, 2009.

[24] C. H. Lin, Mausam, and D. S. Weld. Crowdsourcing control: Moving beyond multiple choice. In *UAI*, pages 491–500, 2012.

[25] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: A crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.

[26] I. Lotosh, T. Milo, and S. Novgorodov. Crowdplanr: Planning made easy with crowd. In *ICDE*, 2013.

[27] H. Ma, R. Chandrasekar, C. Quirk, and A. Gupta. Improving search engines using human computation games. In *CIKM*, pages 275–284, 2009.

[28] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.

[29] W. Mason and D. J. Watts. Financial incentives and the "performance of crowds". In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 77–85, New York, NY, USA, 2009. ACM.

[30] Amazon's mechanical turk. https://www.mturk.com/.

[31] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.

[32] A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR*, pages 160–166, 2011.

[33] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.

[34] A. J. Quinn and B. B. Bederson. Human computation: a survey and taxonomy of a growing field. In *CHI*, pages 1403–1412, 2011.

[35] C. Rashtchian, P. Young, M. Hodosh, and J. Hockenmaier. Collecting image annotations using amazon's mechanical turk. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, CSLDAMT '10, pages 139–147, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[36] J. Selke, C. Lofi, and W.-T. Balke. Pushing the boundaries of crowd-enabled databases with query-driven schema expansion. *PVLDB*, 5(6):538–549, 2012.

[37] B. Settles. Active learning literature survey. *Computer Sciences Technical Report 1648, University of Wisconsin*, 2009.

[38] Tripadvisor. http://www.tripadvisor.com/.

[39] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, pages 989–998, 2012.

[40] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *CHI*, pages 319–326, 2004.

[41] L. von Ahn and L. Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, 2008.

[42] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. re-captcha: Human-based character recognition via web security measures. *Science*, 2008.

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