Robust Entity Resolution Using a CrowdOracle

Donatella Firmani¹, Sainyam Galhotra², Barna Saha², Divesh Srivastava³

¹ Roma Tre University, donatella.firmani@uniroma3.it
² UMass Amherst, {sainyam, barna}@cs.umass.edu
³ AT&T Labs – Research, divesh@research.att.com

Abstract

Entity resolution (ER) seeks to identify which records in a data set refer to the same real-world entity. Given the diversity of ways in which entities can be represented, ER is a challenging task for automated strategies, but relatively easier for expert humans. We abstract the knowledge of experts with the notion of a boolean oracle, that can answer questions of the form “do records u and v refer to the same entity?”, and formally address the problem of maximizing progressive recall and F-measure in an online setting.

1 Introduction

Humans naturally represent information about real-world entities in very diverse ways. Entity resolution (ER) seeks to identify which records in a data set refer to the same underlying real-world entity [4, 8]. ER is an intricate problem. For example, collecting profiles of people and businesses, or specifications of products and services from websites and social media sites can result in billions of records that need to be resolved. Furthermore, these entities are represented in a wide variety of ways that humans can match and distinguish based on domain knowledge, but would be challenging for automated strategies. For these reasons, many frameworks have been developed to leverage humans for performing entity resolution tasks [17, 9].

The problem of designing human-machine ER strategies in a formal framework was studied by [18, 16, 6]. These works introduce the notion of an Oracle that correctly answers questions of the form “do records u and v refer to the same entity?”, showing how different ER logics can achieve different performance having access to such a “virtual” tool and a set of machine-generated pairwise matching probabilities. In this setting, the knowledge of experts is abstracted with the notion of a boolean oracle. However, certain questions can be difficult to answer correctly even for humans experts. To this end, the work in [7] formalizes a robust version of the above ER problem, based on a “Noisy oracle” that can incorrectly label some queried matching and non-matching pairs. The same paper describes a general error correction tool, based on a formal way for selecting indirect “control queries”, that can be plugged into any correction-less oracle strategy while preserving the original ER logic. We refer to both the perfect and the noisy boolean oracle models as CrowdOracle.

Earlier CrowdOracle strategies, such as [18], consider ER to be an off-line task that needs to be completed before results can be used. Since it can be extremely expensive in resolving billions of records, more recent strategies [6, 7] focus on an on-line view of ER, which enables more complete results in the event of early
termination or if there is limited resolution time available. On-line strategies consider progressive recall and F-measure as the metrics to be maximized in this setting. If one plots a curve of recall (or, analogously, F-measure) as a function of the number of oracle queries, progressive recall is quantified as the area under this curve.

Contributions and outline We describe the CrowdOracle pipelines introduced by [6, 7, 16, 18] by using a common framework. The building blocks of the framework enable the definition of new strategies, which can perform even better than those in the most recent works in certain applications. Problem formulation, error correction and some examples are taken from [6, 7]. However, framework formulation, strategy categorization, and an illustrative experiment are original contributions of this paper. This paper is organized as follows.

- In Section 2, we describe our ER problem leveraging the formal notion of a CrowdOracle that can (possibly incorrectly) label some queried matching and non-matching pairs.
- In Section 3, we discuss the components of our descriptive framework for CrowdOracle strategies. Key techniques and theoretical results of [7] are also given in this section.
- In Section 4, we describe previous CrowdOracle strategies leveraging our framework, and show how combining its building blocks in new ways can lead to more efficient algorithms for specific applications.
- Finally, related work is discussed in Section 5.

2 Preliminaries

Let \( V = \{v_1, \ldots, v_n\} \) be a set of \( n \) records. Given \( u, v \in V \), we say that \( u \) matches \( v \) when they refer to the same real-world entity. Let \( H = (V, A, p_m) \), \( A \subseteq V \times V \), be a graph with pairwise machine-generated matching probabilities \( p_m : A \to [0, 1] \). We may not have probabilities of all record pairs, and we may have \( |A| \ll \binom{n}{2} \).

Consider a graph \( C = (V, E^+) \), where \( E^+ \) is a subset of \( V \times V \) and \((u, v) \in E^+\) represents that \( u \) matches with \( v \). \( C \) is transitively closed, that is, it partitions \( V \) into cliques representing distinct entities. We call the nodes in each clique a cluster of \( V \), and we refer to the clustering \( C \) as the ground truth for the ER problem. We refer to the cluster including a given node \( u \), as \( c(u) \in C \). Consider a black box which can answer questions of the form “are \( u \) and \( v \) matching?”.

Edges in \( C \) can be either asked to the black box or inferred leveraging previous answers. If the black box always tells the truth, a user can reconstruct \( C \) exactly with a reasonable number of queries [18, 16]. In real crowdsourcing applications, however, some answers can be erroneous and we can only build a noisy version of \( C \), which we refer to as \( C' \). \( c'(u) \) refers to the cluster in \( C' \) including a given node \( u \).

Definition 1: A CrowdOracle for \( C \) is a function \( q : V \times V \to \{YES, NO\} \times [0, 0.5] \). If \( q(u, v) = (a, e) \), with \( a \in \{YES, NO\} \) and \( e \in [0, 0.5] \), then \( Pr[(u, v) \in E^+] = 1 - e \) if \( a = \text{YES} \), and \( e \) otherwise. In the ideal case, when \( e = 0 \) for any pair \((u, v)\), we refer to the CrowdOracle as perfect oracle.

For instance, if \( q(u, v) = (\text{YES}, 0.15) \), then \((u, v) \in E^+\) with probability 0.85, and if \( q(u, v) = (\text{NO}, 0.21) \), then probability of \((u, v) \in E^+\) is 0.21. We refer to the probability of a specific answer for the pair \((u, v)\) being erroneous, conditioned on the answer being YES or NO, as its error probability \( p_e(u, v) \). Let \( Q = Q_+ \cup Q_- \) be a graph containing all the edges that have been queried until a given moment, along with the oracle answers, we state \( p_e : Q \to [0, 0.5] \). An ER strategy \( s \) takes as input matching probability graph \( H \) and grows a clustering \( C' \) by asking edges as queries to the noisy oracle. We call inference the process of building a clustering \( C' \) from \( Q \). \( C' \) initially consists of singleton clusters: \( s \) can either merge existing clusters into larger clusters, or split an already established cluster. Note that the sub-graph of \( Q_- \) induced by \( c'(u) \) (that is, \( Q_- \cap c'(u) \)) can be non-empty, because of wrong answers. We refer to such a sub-graph as \( Q_-[c'(u)] \).

92
There are many ways of estimating the matching probability function \( p_m \). For instance, automated classifier methods can provide pairwise similarities, which can be mapped to matching probabilities, as in Section 3.1 of [20]. Analogously, there are many ways of accessing error probabilities. For instance, the crowd platform could return a confidence score associated with each answer. Another option is to learn a function mapping similarity scores to error probabilities, akin to matching probabilities [20]. However, computing all the \( \binom{n}{2} \) pairwise matching probabilities may not be feasible when the number of records \( n \) is large, and adopting a crowd-only approach may be prohibitively expensive. To this end, people often remove obvious non-matching pairs during a pre-processing phase. Then, they ask the crowd to examine the remaining pairs, which lead to a relatively sparse graph. One approach is to put obviously non-matching nodes (e.g., watches and dishwashers in an e-commerce dataset) in separate “domains” and consistently remove cross-domain edges. The result, similarly to what is done in [15], is a collection of disconnected complete sub-graphs that can be resolved independently. Another approach, exploited for instance in [3, 14], is to remove obviously non-matching edges either by a matching probability threshold or other cheap procedures such as (overlapping) blocking. The result is a sparse graph, possibly consisting of several connected components (not necessarily cliques). In the traditional (non-crowdsourcing) setting, there is an extensive literature about blocking (see for instance [22]).

3 CrowdOracle Framework

We now define a conceptual framework for describing recent CrowdOracle strategies in terms of basic operations. The input of each CrowdOracle strategy includes the CrowdOracle answers \( Q \), the matching probability function \( p_m \), and the error probabilities \( p_e \), as in definition of Problem 1. In our framework, a strategy is a mechanism for selecting non-exhaustive sequence of CrowdOracle queries (i.e., less than \( \binom{n}{2} \) queries) and inferring clusters according to answers. An oracle strategy also uses the following shared data and methods.

- The partition \( C' \), which can be updated upon the arrival of new answers.
- The method \( \text{query\_pair}(u, v) \), which returns a \{YES, NO\} oracle answer for the pair \((u, v)\). Every invocation of such method contributes to the cost of the ER process and each strategy can be thought of determining a different sequence of \( \text{query\_pair()} \) invocations. We note that given two partially grown clusters \( c_1, c_2 \in C' \) in the perfect oracle setting, the result of \( \text{query\_pair}(u, v) \) is consistently the same for any \((u, v) \in c_1 \times c_2\). For sake of simplicity, then, we sometimes use notations such as \( \text{query\_pair}(u, c_2) \) or \( \text{query\_pair}(c_1, c_2) \) for referring to an arbitrary inter-cluster pair-wise query.

Our framework consists of three aspects, useful for describing a variety of CrowdOracle strategies:

- the cost model of the ER task, which can represent off-line or on-line ER;
- the CrowdOracle model and the selection criteria for issued queries;
- the algorithms for updating the partition \( C' \), which we refer to as “building blocks”.

We discuss each of the above components in the following sub-sections.

3.1 Cost model

Recall and F-measure denote, respectively, the fraction of positive edges found among those of the unknown clustering \( C \), and the harmonic mean of this value and precision, which is the fraction of positive edges found that truly belong to \( C \). Specifically, F-measure is defined as \( \frac{2 \cdot \text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}} \). Recall and F-measure naturally represent the “amount” of the information that is available to the user at a given point. However, they cannot distinguish the dynamic behaviour of different strategies. In other words, they cannot represent whether a strategy achieves
The theory in [18] yields that both problems require at least oracle setting, optimizing F-measure is the same as optimizing recall. Problem 2 (Off-line): CrowdOracle problems

Given a set of records \( V \), a CrowdOracle access to \( C \), and a matching probability function \( p_m \) (possibly defined on a subset of \( V \times V \)), find the strategy that maximizes progressive F-measure.

Problem 2 (Off-line): Given a set of records \( V \), a CrowdOracle access to \( C \), and a matching probability function \( p_m \) (possibly defined on a subset of \( V \times V \)), find the strategy that maximizes F-measure and minimizes queries.

An optimal strategy for Problem 1 is also optimal for Problem 2, making Problem 1 more general. For instance, strategy \( S_1 \) in Example 1 is optimal for both problems, whereas \( S_2 \) is optimal only for Problem 2. The theory in [18] yields that both problems require at least \( n - k \) questions for growing all \( k \) clusters (i.e., the size of a spanning forest of \( C^+ \)) and at least \( \binom{k}{2} \) extra questions for proving that clusters represent different entities. Intuitively, a strategy for Problem 2 tries to ask positive queries before negative ones, whereas a strategy for Problem 1 also does this in a connected fashion, growing larger clusters first. We call ideal() the optimal

---

1https://en.wikipedia.org/wiki/Category:Universities_and_colleges_in_Rome
2In the perfect oracle setting, precision is 1 and F-measure is equal to \( \frac{\text{recall+1}}{2} \).
3In practice, some strategy can have great performance in the on-line setting at the cost of slightly worse final recall-queries ratio.
4For sake of completeness, we also give an example of sub-optimal strategy for Problem 2 and Example 1. Consider \( S_3 \) as \((r_a, r_d), (r_b, r_d), (r_a, r_f), (r_d, r_f), (r_a, r_e), (r_a, r_e)\). The second query \((r_a, r_d)\) is somewhat “wasted” in the perfect oracle setting as the corresponding negative edge would have been inferred after \((r_a, r_e)\) by leveraging transitivity.
strategy for the on-line problem. Consider Example 1, \texttt{ideal()} first grows the largest cluster \( C_1 = \{r_a, r_b, r_c\} \) by asking adjacent edges belonging to a spanning tree of \( C_1 \) (that is, every asked edge shares one of its endpoints with previously asked edges). After \( C_1 \) is grown, \texttt{ideal()} grows \( C_2 = \{r_d, r_e\} \) in a similar fashion. Finally, \texttt{ideal()} asks negative edges in any order, until all the labels are known, and also \( C_3 = \{r_f\} \) can be identified\(^5\).

### 3.2 Oracle model and query selection

A strategy \( S \) can be thought of as a \textit{sequence}\(^6\) of queries. While in the perfect oracle setting \( S \) can leverage transitivity, in real CrowdOracle applications, \( S \) can only trade-off queries for F-measure. Let \( T \) be the set of positive answers collected by \( S \) at a given point of the ER process. We can think of two extreme behaviors.

- \( S \) skips all the queries that can be inferred by transitivity, that is, \( T \) is a \textit{spanning forest} of \( V \). This is necessary and sufficient to resolve the clusters in the absence of answer error, as shown in Example 1.
- \( S \) asks exhaustively all the queries in \( V \times V \), that is, \( T \) is a noisy collection of cliques, and infers clusters by minimizing disagreements (for instance, via correlation clustering).

In the middle of the spectrum, the work in \cite{7} shows that the error of resolution can be minimized if we strengthen the min-cuts of \( T \) with “control queries”\(^7\), exploiting the notion of \textit{expander graphs}. Expander graphs are sparse graphs with strong connectivity properties. We first describe spanning forest approaches for error correction and then we summarize the methods in \cite{7}.

#### Spanning forest

As discussed at the end of Section 3.1, the goal of a perfect oracle strategy \( S \) is two-fold: promoting positive queries and growing clusters sequentially (only for Problem 1). In order to achieve this goal, the query selection of \( S \) can be driven by the \textit{recall gain} of discovering that two specific clusters refer to the same entity. Depending on how \( S \) estimates the recall gain of a cluster pair \( c_u, c_v \in C' \), we can have \textit{optimistic} and \textit{realistic} query selection. The optimistic approach only considers the maximum inter-cluster matching probability, that is, it estimates the recall gain of \( c_u \) and \( c_v \) as \( \max_{u \in C_u, v \in C_v} p_m(u, v) | c_u | \cdot | c_v |. \) Selecting queries optimistically can be computationally efficient, and can give good results if matching probabilities are accurate. However, it can perform badly in presence of non-matching pairs having higher probability than matching pairs \cite{6}. To this end, realistic approach uses a robust estimate, based on the notion of \textit{cluster benefit} \( \text{cbn}(c_u, c_v) = \sum_{u, v \in C_u \times C_v} p_m(u, v). \) We note that if \( |c_u| = |c_v| = 1 \) then \( \text{cbn}(c_u, c_v) = p_m(u, v) \), as in the optimistic approach.\(^8\) Difference between optimistic and realistic is illustrated below.

#### Example 2 (Optimistic and realistic):

Consider clusters grown by \( S_2 \) of Example 1 after 4 queries \( C' = \{r_a, r_b\}, \{r_c\}, \{r_d, r_e\}, \{r_f\} \). Optimistic estimate of recall gain between non-matching \( \{r_a, r_b\} \) and \( \{r_f\} \) is \( 2 \cdot 1 \cdot 0.55 = 1.1 \), which is comparable to matching \( \{r_a, r_b\} \) and \( \{r_c\} \), i.e., \( 2 \cdot 1 \cdot 0.60 = 1.2 \). By switching to realistic, we get \( \text{cbn}(\{r_a, r_b\}, \{r_f\}) = 0.55 + 0.29 = 0.84 \) as opposed to \( \text{cbn}(\{r_a, r_b\}, \{r_c\}) = 0.60 + 0.54 = 1.14 \).

#### Expander graph

Control queries for handling CrowdOracle errors can be selected among those that provide strongest connectivity between records of each cluster, based on the concept of \textit{graph expanders}, which are sparse graphs with formal connectivity properties. Expansion properties of clusters translate into (i) \textit{robustness} since the joint error probability of each cut is small, all the subsets of nodes are likely matching pair-wise;\

\(^5\)We note that recall is 1 as soon as \( c_2 \) is fully grown.

\(^6\)Partially ordered if parallel.

\(^7\)In addition to the spanning forest.

\(^8\)One may wonder why not take the average. We note that the recall gain is larger for large clusters. However the average would be a robust estimate of the probability that the two clusters are matching.
(ii) **cost-effectiveness**: the total number of queries is small, as edge density of expander is small. Technically, different formalizations of connectivity give rise to different notions of expanders. We focus on edge expansion.

**Definition 2 (Edge expansion):** The edge expansion parameter \( h(G) \) of a graph \( G = (V,E) \) is defined as the minimum cut size \( \left(V', V \setminus V'\right) \) for every subset of nodes \( V' \subseteq V \), \( |V'| \leq |V|/2 \). Cut size is defined as the number of edges crossing the cut. If \( h(G) \) is small, we say that the graph has at least one **weak cut**.

\( G \) is a \( \gamma \)-expander if \( h(G) \geq \gamma \). Intuitively, every subset of the nodes of \( G \) that is not “too large” has a “large” boundary. In our ER setting, the boundary of a subset of nodes of the same entity provides evidence of how the subset relates to the rest of the graph. A disconnected graph is not an expander (the boundary of a connected component is empty), and every connected graph is an expander. However, different connected graphs have different edge expansion parameters. Consider an entity \( C \) being partitioned in two clusters \( A \) and \( B \) at some point of the ER process, like in Figure 1(a). Upon a positive answer for the pair \((u,v)\) with error probability 0.3, the two clusters are connected but the probability of the whole cluster \( C = A \cup B \) of being correct is only 0.7. At this point, whatever answers have been collected inside \( A \), the boundary of \( A \) consists indeed of a mere edge. Upon two positive answers for the pairs \((u_1,v_1)\), and \((u_2,v_2)\)\(^9\) with error respective probabilities 0.2 and 0.4, the connection between \( A \) and \( B \) becomes more robust. The probability of the whole cluster \( C = A \cup B \) can be quantified with the product of the cut error probabilities under the assumption of independent error, that is \( 1 - 0.3 \cdot 0.2 \cdot 0.4 = 0.976 \). At this point, the boundary of \( A \) consists indeed of three edges. The larger the boundary the higher the success probability. Suppose at this point the ER algorithm decides to merge \( A \) and \( B \). Since \( C \) is unknown a priori, in order to trust the result, the large boundary property has to hold for all the possible partitions \( A \) and \( B \). Therefore, the structure we are looking for is an expander. The weights are introduced because the product value, which is related to the weight of the cut, matters more than the cut cardinality (i.e., number of edges). The complete graph has the best expansion property, but it also has largest possible degree, and it would be prohibitively expensive to build. Informally, a graph is a good expander if it has low degree and high expansion parameters. In the following example and in Figure 1(c) we show the possible result of having an expander \((\gamma = 1)\) for data in Example 1, compared with spanning forests of Figure 1(b).

**Example 3 (Spanning forest and expander graph):** Consider the six places of Example 1, plus three extra names \((r_1)\) Studium Urbis, \((r_2)\) Città Universitaria, \((r_3)\) Uniroma 3. Correct clustering is \( \{r_a, r_b, r_c, r_1, r_2\}, \{r_d, r_e, r_3\}, \) and \( \{r_f\} \). Both connected components of Figure 1(b) (i.e., trees in the spanning forest) and expander graphs of Figure 1(c) yield the same, correct, clustering. While connected components only work in absence of errors, expander produces the correct clustering also in presence of plausible human errors such as \((r_a, r_f)\)\(^11\),

---

\(^9\)Other expansion notions include node expanders, and **spectral** expanders. We refer the interested reader to [1] for more discussion.

\(^10\)\(u_1, u_2 \in A\), and \(v_1, v_2 \in B\)

\(^11\)“Università di Roma” means “University of Rome”.


\((r_c, r_3)\) (false positives) and \((r_c, r_2)\) (false negative). Even though expansion requires more queries than building a spanning forest, it is far from being exhaustive: in the larger clusters, only 5 queries are asked out of \(\binom{5}{2} = 10\).

The method `query_cluster()` in [7] is meant to be called in place of `query_pair()` with the purpose of growing clusters with good expansion properties. Given a query \((u, v)\) selected by a strategy (represented with the two corresponding clusters \(c_u\) and \(c_v\)), `query_cluster()` provides an intermediate layer between the ER logic and the CrowdOracle. Similarly to `query_pair()`, indeed, `query_cluster()` provides functionalities for deciding when two clusters (or any two sets of nodes) are matching. However, instead of asking the selected query \((u, v)\) as `query_pair(u, v)` would do, `query_cluster(c_u, c_v, \beta)` selects a bunch of random queries between \(c_u = c'(u)\) and \(c_v = c'(v)\), and returns a YES answer only if the estimated precision of the cluster \(c_u \cup c_v\) is high. The parameter \(\beta\) controls the edge expansion value \(\gamma\) trading-off queries for precision.\(^{12}\) Smaller values of \(\beta\) correspond to sparser clusters, and therefore to less queries (\(\beta = 0\) asks a single positive question, yielding result similar to `query_pair()`). Greater values of \(\beta\) correspond to denser clusters and to higher precision. Formally, expected precision increases exponentially with \(\beta\). We refer the interested reader to Theorem 3 in [7].

**Discussion** By analogy with the optimistic and realistic spanning forest approaches, we refer to graph expanders as pessimistic approach. We note that a CrowdOracle strategy \(S\) can leverage multiple approaches together, as discussed later in Section 4. For instance, \(S\) can select a cluster pair to compare by using the realistic cluster benefit but then use `query_cluster()` as a substitute of plain connectivity, and so on. Finally, experiments in [7] show that \(\beta = 1\) achieves the best progressive F-measure, and we set this as default value.

### 3.3 Building block algorithms

We now describe the basic operations of our framework. The operations can be implemented either with the simple `query_pair()` oracle interface, or can be modified to apply random expansion with `query_cluster()`.

- **Insert node** This operation grows already established clusters by adding a new node \(u\). Possible outcomes are success, when \(u\) is recognized as part of one of the clusters, or fail, in which case \(\{u\}\) is established as a new singleton cluster. Specifically, `insert-node(u)` compares \(u\) to cluster \(c_i\), \(i = 1, 2, \ldots\) until `query_pair(u, c_i)` (or `query_cluster(\{u\}, c_i)`), when using the error-correction layer) returns a positive answer. The main lemma in [16] proves that an insert-node-only strategy in the perfect oracle setting requires at most \(n - k + \binom{k}{2}\) queries. We can do better by introducing an “early termination” condition (e.g., at most \(\tau\) comparisons before establishing a new cluster) at the price of possible loss in recall.

- **Merge clusters** Recall of an insert-node-only algorithm can be smaller than 1 for two reasons: (i) positive-to-negative errors of CrowdOracle, and (ii) positive questions “deferred” for early termination. This operation can boost recall by merging pairs of partially grown clusters that represent the same entity. To this end, `merge-clusters(c_i, c_j)` can rely upon a single intra-cluster query `query_pair(u, v)` with arbitrary \(u, v \in c_i \times c_j\), or leverage `query_cluster(c_i, c_j)` for expander-like control queries.

In real CrowdOracle applications, the above methods can make mistakes – even when equipped with the pessimistic random expansion toolkit – by adding a node to the wrong cluster or by putting together clusters referring to different entities. Specifically, false negatives can separate in different clusters nodes referring to the same entity, while false positives can include in the same cluster nodes referring to different entities. This is more likely to happen early in the ER process, when we have collected few CrowdOracle answers. Luckily, mistakes can become evident later on, upon the arrival of new answers. The methods below are useful for identifying and correcting `insert-node()` and `merge-clusters()` mistakes.

\(^{12}\)Technically, \(\beta\) controls the ratio between the edge expansion parameter and the log of the given cluster size.
Table 1: Recent strategies categorization. We note that spanning forest strategies do not use delete-node(), and that edge expansion strategies can leverage optimistic and realistic approaches in some of their phases.

<table>
<thead>
<tr>
<th>STRATEGY</th>
<th>COST MODEL</th>
<th>QUERIES</th>
<th>insert-node()</th>
<th>delete-node()</th>
<th>merge-clusters()</th>
</tr>
</thead>
<tbody>
<tr>
<td>wang()</td>
<td>off-line</td>
<td>spanning forest</td>
<td></td>
<td></td>
<td>optimistic</td>
</tr>
<tr>
<td>vesd()</td>
<td>on-line</td>
<td></td>
<td></td>
<td></td>
<td>optimistic</td>
</tr>
<tr>
<td>hybrid()</td>
<td>on-line</td>
<td></td>
<td></td>
<td></td>
<td>realistic</td>
</tr>
<tr>
<td>mixed()</td>
<td>on-line</td>
<td></td>
<td></td>
<td></td>
<td>realistic</td>
</tr>
<tr>
<td>lazy()</td>
<td>off-line</td>
<td>edge expansion</td>
<td>realistic</td>
<td>pessimistic</td>
<td>realistic phase + pessimistic phase</td>
</tr>
<tr>
<td>eager()</td>
<td>on-line</td>
<td></td>
<td>realistic/pessimistic</td>
<td>pessimistic</td>
<td>realistic/pessimistic</td>
</tr>
<tr>
<td>adaptive()</td>
<td>on-line</td>
<td>both</td>
<td>realistic/pessimistic</td>
<td>pessimistic</td>
<td>realistic/pessimistic</td>
</tr>
</tbody>
</table>

- **Delete node** This operation removes erroneous nodes from the clusters. Specifically, delete-node(u) triggers query_cluster(u, c’(u)) and whenever it returns NO, it pulls out the node and sets up a new singleton cluster {u}. We note that delete-node() can successfully fix both single-node errors due to insert-node() failure and larger merge-clusters() errors if one of the clusters c is sufficiently small. In the latter case, we can indeed repeatedly apply delete-node() to remove the smaller cluster, and then put it back together with insert-node() and merge-clusters() operations.

- **Split cluster** In case of severe merge-clusters() errors, we can try to recover by identifying “weak cuts” (see Definition 2) and splitting low confidence clusters into high confidence sub-graphs. To this end, split-cluster(c) selects the minimum cut (c_u, c_v) of a given cluster c (which corresponds to maximum joint error probability) and tries to expand it with query_cluster(c_u, c_v). If it succeeds, no changes to c need do be done. Otherwise, the cluster is split into the two sides of the cut.

In the next section, we will illustrate how combining the above operations leads to various strategies.

## 4 CrowdOracle Strategies

We now describe prior CrowdOracle strategies using the framework in Section 3, as summarized in Table 1. Then, we summarize the experimental results of the original papers [6, 7] and provide intuitions and illustrative experiments for a new strategy – mixed() – that can outperform hybrid() in specific application scenarios.

### 4.1 Strategy description

We consider wang(), vesd() and hybrid() from [6] in the perfect oracle setting, and lazy(), eager() and adaptive() from [7] in the general CrowdOracle model. (We refer the interested reader to original papers for more discussion.) For the perfect oracle – or, equivalently, spanning forest – strategies, we also report the key approximation results in [6] for our two problems 1 and 2, under a realistic edge noise model for $p_m$.

**Wang** The strategy in [18], which we refer to as wang(), is purely based on optimistic merge clusters operations. Every node starts as a singleton cluster. Then, cluster pairs are possibly merged in non-increasing order of matching probability, leveraging transitivity. Consider Example 1, where $(r_a, r_d)$ is the edge with highest $p_m$ value. The first operation is merge-clusters($\{r_a\}, \{r_d\}$) – which is equivalent to query_pair(r_a, r_d) in this setting – yielding a negative outcome. The next edge in non-increasing order of matching probability is $(r_d, r_f)$, thus, the second operation is merge-clusters($\{r_d\}, \{r_f\}$), yielding another negative result. Something different happens upon the third operation, which is merge-clusters($\{r_d\}, \{r_e\}$), because
query_cluster(r_d, r_e) yields a positive answer. The singleton clusters r_a and r_d are merged into a “doubleton” cluster \{r_d, r_e\}, and – given that the next edge in the ordering is \( (r_a, r_e) \) – the fourth operation is merge-clusters(\{r_a\}, \{r_d, r_e\}) and so on. We note that, being based on spanning forests, the optimistic merge-clusters() variant used by wang() asks only one of the two inter-cluster edges \((r_a, r_d)\) and \((r_a, r_e)\).

We can make the strategy pessimistic by replacing the query_pair() step with query_cluster().

The wang() strategy has an approximation factor of \( \Omega(n) \) for Problem 1 and \( O(\log^2 n) \) for Problem 2.

**Vesd**  The strategy in [16], which we refer to as vesd(), leverages optimistic insert node operations. Differently from wang(), vesd() starts with a unique singleton cluster, corresponding to the node with highest expected cluster size. In our running example, this is the cluster \{r_d\}.\(^{14}\) Nodes are possibly inserted in current clusters in non-increasing order of expected cluster size: the next node in the ordering is \( r_e \), thus the first operation is insert-node(\( r_e \)). This operation triggers query_pair(\( r_e, r_d \)) as the first query (differently from wang()) and yields a positive answer, upon which the initial cluster \{r_d\} is “grown” to \{r_d, r_e\}. After processing \( r_a \) and \( r_e \), the current clusters are \( c_1 = \{r_d, r_e\} \) and \( c_2 = \{r_a, r_c\} \). When we consider \( r_f \) (which represents a different entity) we have two candidate clusters for insertion, and four intra-cluster edges connecting \( r_f \) to nodes in \( c_1 \) and \( c_2 \). Since optimistic insert node is driven by the highest matching probability edge among those (i.e., \( (r_a, r_f) \)) the first cluster selected for comparison – with no success – is \( c = \{r_a, r_c\} \). Similarly to wang(), vesd() requires one query for comparing \( r_f \) and \( c \), which can be arbitrarily chosen between query_pair(\( r_a, r_f \)) and query_pair(\( r_c, r_f \)). Before moving to node \( r_b \), insert-node(\( r_f \)) compares \( r_f \) to the other clusters (i.e., \( c_2 \)) until possibly success. Otherwise, a new cluster is created. Analogously to wang(), the strategy can be made pessimistic by replacing query_pair() with query_cluster().

The vesd() strategy has a better approximation factor of \( \Omega(\sqrt{n}) \) than wang() for Problem 1 and same \( O(\log^2 n) \) for Problem 2. Without assumptions on matching probabilities, vesd() is shown to be \( O(k) \) (which is usually \( O(n) \)) for the off-line setting\(^{15}\), while wang() can be arbitrarily bad.

**Hybrid**  The hybrid() strategy in [6] combines wang() and vesd(), and can be modified to apply random expansion similarly. Specifically, it first applies a variant of vesd() where insert-node() is modified with a parametric early termination option.\(^{16}\) That is, some edges between established clusters may be non-resolved (i.e., non-inferable) at some point. In addition, nodes to add are selected based on their singleton-cluster benefit (i.e., sum of incident matching probabilities) with respect to established clusters, making hybrid() a realistic insert node strategy. After this phase, recall can be smaller than 1. To this end, hybrid() applies wang() taking care of “deferred” questions due to early termination. The early termination’s parameters can be set such that i) hybrid() becomes a realistic variant of vesd(), by letting insert-node() terminate only in case there are no more clusters to consider (that is, no further merge-clusters() operations are needed); 2) hybrid() works like wang(), by inhibiting insert-node() completely; 3) anything in between.

In the worst case, hybrid() provides an \( O(\sqrt{n}) \)-approximation to the on-line Problem 1. If matching probabilities are such that we can pick two representatives from any cluster \( A \) before elements of a smaller cluster \( B \), then (no matter what the matching probability noise is) hybrid() performs like ideal1(), because the benefit of a third node in the same cluster is higher than any other node in a different cluster, and so on.

**Lazy**  This pipeline is described in [7] and can be thought of as the simplest edge expansion strategy in the framework. lazy() is indeed focused towards optimizing the progressive F-measure at the cost of lower precision at the start. It does so by following a mix of perfect oracle strategies vesd() and wang() – similarly to what

\(^{13}\)parameters correspond to the clusters of the edge endpoints

\(^{14}\)Expected cluster size of \( r_d \) is 3.55

\(^{15}\)In this setting, we only need to minimize the number of queries.

\(^{16}\)Intuitively, insert-node(\( u \)) fails not only if there are no more clusters to examine to, but also if \( \text{cbn}(u, c) \) drops below a given threshold \( \theta \) or the number of questions related to node \( u \) exceeds a given amount of trials \( \tau \).
hybrid() does – in the beginning to avoid asking extra queries as required to form expanders. However, at the end, lazy() runs merge-clusters() with query_cluster() over all cluster pairs and delete-node() over all the nodes, aiming at the correction of recall and precision errors, respectively. We note that the final error correction phase may be challenged by large merge-clusters() errors\footnote{Only removing singleton nodes of one of two erroneously merged clusters, without putting them back together.}, and delete-node() could give better results. This was not considered in [7], where lazy() is used as a baseline. In the beginning, the only difference with hybrid() is merge-clusters(), because cluster pairs are possibly merged in non-increasing order of cluster benefit (rather than matching probability), making lazy() a realistic merge cluster strategy.

**Eager** The strategy eager() in [7] has “orthogonal” behaviour with respect to lazy(). Indeed, it maintains high precision at the cost of low progressive F-score. It is a pessimistic version of lazy() where both insert-node() and merge-clusters() use query_cluster() as a substitute of query_pair(). Since expander properties are maintained throughout the execution, large cluster merge errors are unlikely. Therefore, split-cluster() is not used and the final error correction phase of eager() is the same as lazy().

**Adaptive** The strategy adaptive() in [7] achieves the best of eager() and lazy() in real CrowdOracle applications. It provides the same final F-measure of eager() earlier in the querying procedure, along with the high progressive F-measure of lazy(). The intuition is to switch between query_pair() and query_cluster() depending on the current answer. We compare clusters with query_pair() as in lazy(), but we use our robust comparison tool query_cluster() if the result is in “disagreement” with matching probabilities. Formally, a disagreement can be: (i) a positive answer in case of low average matching probability ($< 0.5$); (ii) a negative answer in case of high average matching probability ($\geq 0.5$). hybrid() runs two executions of the error correction merge-clusters()+delete-node() procedure, one at the end (similarly to what lazy() and eager() do) and another when switching from the insert node phase to the merge cluster phase. Such extra-execution is useful for correcting early errors due to the adaptive nature of the initial insert-node phase.

### 4.2 Empirical evaluation

Depending on matching and error probabilities (i.e., how accurate machine-based methods and crowd can be on the specific application), the considered strategies may have different performances. hybrid() and adaptive() are shown to be comparable or better than other strategies in their respective settings. However, when comparable, a user may prefer simpler strategies. Next, we report the main takeaways from [6, 7].
1. Suppose there are no erroneous answers, and matching probabilities have uniform edge noise. If the size distribution of clusters is skewed (i.e., there are few large clusters and a long tail of small clusters), we expect vesd() to be better than wang() and hybrid() to be comparable or better than vesd().

2. In the same setting, but with small clusters (for instance, in Clean-Clean ER tasks where most clusters have size 2), we expect wang() to be much better than vesd() and hybrid() to perform like wang().

3. Suppose now the error rate of answers is high and matching probabilities are correlated with the ground truth, that is, truly positive edges have high probability and truly negative edges have low probability. We expect eager() to perform better than lazy(), and adaptive() to perform like eager().

4. Similarly, if the error is high and matching probabilities are uncorrelated with the ground truth, we expect eager() to be better than lazy(), and adaptive() to be like eager().

5. Instead, when the error is low and matching probabilities are correlated with the ground truth, we expect lazy() to be better than eager(), and adaptive() to be like lazy().

6. In the less realistic case where the error is low and matching probabilities are uncorrelated with the ground truth, we still expect lazy() to be better, but we expect adaptive() to be like eager().

7. There can be mixed cases of reasonable error rate and matching probability noise. We expect the different edge expansion strategies to have similar progressive F-measure in such cases.

Figures 2(a) and 2(b) report an experimental comparison of the considered strategies, against the popular cora bibliographic dataset.\textsuperscript{18} cora is an example of a dataset where matching probabilities are correlated with the ground truth and the cluster size distribution is skewed (the top three clusters account for more than one third of the records), thus matching with application scenarios 1 and 3. The plots confirm the expected behaviour of strategies, with adaptive(), eager(), hybrid(), and vesd() being close to ideal().

Mixed Consider the perfect oracle setting, for sake of simplicity. Suppose that the matching probabilities have, in addition to the noise observed in the experiments of Figures 2(a) and 2(b), also a new, systematic noise, which only affects specific clusters and “splits” them in two parts. This can happen in many applications, where real-world entities are represented in well-defined variations. Examples include different sweetness – dry, extra dry – of the same wine entity, or the ArXiv and conference versions of the same paper. They are not really different entities, but we can expect lower $p_m$ values between records of the two variations, than between records of the same variation. Systematic “split” error is correlated with entity variations rather than ground truth, and is a challenging scenario for hybrid() strategy. After growing the first variation, indeed, if inter-variation $p_m$ values are such that corresponding questions are skipped by insert-node() and deferred to the merge-clusters() phase, hybrid() would seed the new variation as a separate cluster and grow it as if it was a different entity, until the start of the second phase. Our framework enables the design of a new strategy, that we call mixed(), that at every step selects insert-node() or merge-clusters() depending on the realistic cluster benefit, rather than having two separated phases. Experimental comparison of hybrid() and mixed() is shown in Figure 2(c), against a synthetic version of the cora dataset. In the synthetic cora, we artificially add the systematic error in the largest cluster $c$ and set $p_m(u, v)$ to 0.001, $u, v \in c$, if $u$ is odd and $v$ is even.\textsuperscript{19} The mixed() strategy consists of a sequence of realistic merge clusters operations, with sporadic realistic insert node (with the same early termination as hybrid()). Specifically, whenever the next pair of clusters in non-increasing order of benefit (see Section 2) corresponds to two singleton nodes $\{ u \}$ and

\textsuperscript{18} We refer the reader to the original papers [6, 7] for more details about the dataset and the experimental methodology.

\textsuperscript{19} We also augment inter-variation $p_m$ values and re-scale all the other scores in the graph, so that ranking of nodes by expected cluster size does not change for the purpose of the experiment.
\{v\} never seen before, \texttt{mixed() substitutes merge-clusters(\{u\}, \{v\})} with \texttt{insert-node(w)}, where \texttt{w} is the largest unprocessed node in non-increasing order of expected cluster size. We observed that \texttt{mixed()} performs like \texttt{hybrid()} with the original \texttt{cora} dataset. However, in the presence of systematic error, after growing the first entity variation \texttt{ca}, as soon as the second variation of \texttt{c} – which we refer to as \texttt{cb} – grows large enough that the \texttt{cbn(ca, cb)} becomes relevant, the two sub-clusters are merged early by \texttt{merge-clusters()} into \texttt{c}.

5 Related Work

Entity Resolution (ER) has a long history (see, e.g., [4, 8] for surveys), from the seminal paper by Fellegi and Sunter in 1969 [5], which proposed the use of a learning-based approach, to rule-based and distance-based approaches (see, e.g., [4]), to the recently proposed hybrid human-machine approaches (see, e.g., [17, 9]). We focus on the latter line of work, which we refer to as \texttt{crowdsourced ER}, where we typically have access to machine-generated probabilities that two records represent the same real-world entity, and can ask “are \texttt{u} and \texttt{v} matching?” questions to humans.

The strategies described in [16, 18, 6] abstract the crowd as a whole by an \texttt{oracle}, which can provide a correct YES/NO answer for a given pair of items. Traditional ER strategies consider ER to be an offline task that needs to be completed before results can be used, which can be extremely expensive in resolving billions of records. To address this concern, recent strategies [21, 13] propose to identify more duplicate records early in the resolution process. Such \texttt{online} strategies are empirically shown to enable higher recall (i.e., more complete results) in the event of early termination or if there is limited resolution time available. The strategies focus on different ER logics and their performances, which can be formally compared in terms of the number of questions asked for 100% recall [6, 12]. Unfortunately, the strategies do not apply to low quality of answers: if an answer involving two clusters \texttt{C1}, and \texttt{C2}, is wrong, the error propagates to all the pairs in \texttt{C1 × C2}.

The oracle errors issue raised by [18, 16, 6] is addressed by recent works such as [10, 14, 15, 7]. The solution provided by these works consists of a brand new set of techniques for replicating the same question (i.e. about the same pair) and submitting the replicas to \texttt{multiple} humans, until enough evidence is collected for labeling the pair as matching or non-matching (see Section 5). New techniques include voting mechanisms [10], robust clustering methods [14], and query-saving strategies such as classifying questions into easy and difficult [15]. These algorithms show how to make effective use of machine-generated probabilities for generating replicas and correcting errors, sometimes for the specific pair-wise answers and sometimes for the entities as a whole. In this setting, each YES/NO answer for a given pair of items can be interpreted as a matching probability: 1 – \texttt{pE} if the pair is supposed to be matching, and \texttt{pE} otherwise. (An information theoretic perspective of it is provided in [11].) General purpose answer-quality mechanisms are described in the crowd-sourcing literature [2, 19].

6 Conclusions

In this paper, we considered the pipelines described in [18, 16, 6, 7] in the general CrowdOracle model. We summarized their goals, provable guarantees and application scenarios. Specifically, we described a common framework consisting of simple operations that combined together lead to the considered strategies. This framework raised the issue of a specific scenario, which can be challenging for strategies in [18, 16, 6, 7]. To this end, we leveraged the simple operations introduced in this paper for defining an original strategy, which is better than \texttt{hybrid()} in the challenging scenario (and comparable in the traditional setting).

References


