Cost-Effective Crowdsourced Entity Resolution: A Partial-Order Approach

Chengliang Chai†, Guoliang Li†, Jian Li‡, Dong Deng†, Jianhua Feng†
†Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China
‡Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
{liguoliang, lijian83,fengjh}@tsinghua.edu.cn; {chaicl15, dd11}@mails.tsinghua.edu.cn

ABSTRACT

Crowdsourced entity resolution has recently attracted significant attentions because it can harness the wisdom of crowd to improve the quality of entity resolution. However existing techniques either cannot achieve high quality or incur huge monetary costs. To address these problems, we propose a cost-effective crowdsourced entity resolution framework, which significantly reduces the monetary cost while keeping high quality. We first define a partial order on the pairs of records. Then we select a pair as a question and ask the crowd to check whether the records in the pair refer to the same entity. After getting the answer of this pair, we infer the answers of other pairs based on the partial order. Next we iteratively select pairs without answers to ask until we get the answers of all pairs. We devise effective algorithms to judiciously select the pairs to ask in order to minimize the number of asked pairs. To further reduce the cost, we propose a grouping technique to group the pairs and we only ask one pair instead of all pairs in each group. We develop error-tolerant techniques to tolerate the errors introduced by the partial order and the crowd. Experimental results show that our method reduces the cost to 1.25% of existing approaches (or existing approaches take 80× monetary cost of our method) while not sacrificing the quality.

Categories and Subject Descriptors
H.2 [Database Management]: Database applications
Keywords: Crowdsourcing;Entity Resolution;Partial Order

1. INTRODUCTION

Entity resolution aims to find records that refer to the same entity from a collection of records. For example, consider the 11 records in Table 1. r_1, r_2 and r_3 refer to the same entity. r_4, r_5, r_6 and r_7 refer to the same entity. Entity resolution has many real-world applications, particularly in health data integration, knowledge-base construction, web search, comparison shopping, and law enforcement.

However existing machine-based methods are still far from perfect[22, 24], because the same entity may have many unpredictable representations. Crowdsourced entity resolution that leverages the crowd's ability to solve this problem has attracted significant attentions[12, 21, 23, 24, 25].

A brute-force method enumerates every pair of records and asks the crowd to check whether they refer to the same entity. This method involves huge monetary costs, especially for large datasets. To address this problem, several algorithms have been proposed to reduce the cost by pruning some pairs that do not need to be asked. Wang et al. [23] utilized the transitivity to reduce the cost, but the quality may not be guaranteed. This is because the transitivity may not hold for some records, which leads to incorrect deduction and uncontrollable error propagation. Wang et al. [24] proposed a correlation-clustering method, which adaptively assigned the records referring to the same entity into the same cluster. This method improves the quality at the expense of asking many more questions and thus involves high monetary costs. In summary, existing methods either cannot achieve high quality or involve huge monetary costs.

To address these problems, we propose `Power`, a partial-order based crowdsourced entity resolution framework, which significantly reduces the monetary cost while keeping high quality. The basic idea is that we define a partial order on the record pairs and prune many pairs that do not need to be asked based on the partial order. Specifically, we first define a partial order: (1) If a pair of records refer to the same entity, then the pairs preceding this pair also refer to the same entity; (2) If a pair of records refer to different entities, then the pairs succeeding this pair refer to different entities. Then we select a pair as a question and ask the crowd to check whether the records in the pair refer to the same entity. Based on the answer of this pair, we infer the answers of other pairs based on the partial order. Thus our goal is to judiciously select the pairs to ask in order to minimize the number of asked pairs. To this end, we devise effective algorithms to iteratively select pairs without answers to ask until we get the answers of all the pairs. To further reduce the cost, we propose a grouping technique to group the pairs such that we only need to ask one pair instead of all pairs in each group. Since asking only one pair in each iteration leads to a high latency, we propose effective techniques to select multiple pairs in each iteration. As both the partial order and the crowd may introduce errors, we develop error-tolerant techniques to tolerate the errors.

To summarize, we make the following contributions.
(1) We propose a partial-order based crowdsourced entity resolution framework. We define a partial order on record pairs and utilize the partial order to infer the answers of some unasked pairs so as to reduce the monetary cost.

SIGMOD'16, June 26 - July 1, 2016, San Francisco, CA, USA.

DOI: http://dx.doi.org/10.1145/2882903.2915252

(2) We construct a graph based on the partial order and utilize the graph to ask questions and infer answers. We devise efficient algorithms to construct the graph. We develop a grouping technique to group the record pairs, which can further reduce the cost. We prove that the optimal grouping is NP-hard and propose approximation algorithms.
(3) We judiciously select pairs to ask in order to minimize the number of asked pairs. We propose a path-based algorithm that asks one question in each iteration and prove that the algorithm is optimal in general. To reduce the latency, we devise a topological-sorting-based algorithm that asks multiple questions in parallel in each iteration.
(4) We develop a probability-based method to tolerate the errors introduced by the crowd and the partial order.
(5) We conduct experiments using real-world datasets on a real crowdsourcing platform. Experimental results show that our method reduces the cost to 1.25% of existing approaches (or existing approaches take more than 80 times money of our method) while not sacrificing the quality.

The rest of this paper is structured as follows. We first define the problem and review related work in Section 2 and then propose our framework in Section 3. The grouping strategy, question selection, and error-tolerant techniques are discussed in Sections 4, 5, 6 respectively. We report experimental results in Section 7 and conclude in Section 8.

2. PRELIMINARIES

2.1 Problem Definition

Definition 1 (Crowdsourced Entity Resolution). *Consider a table $\mathcal{T}$ with m attributes $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m\}$ and n records $\{r_1, r_2, \ldots, r_n\}$, where each record denotes an entity. The entity resolution aims to identify the records that refer to the same entity. Crowdsourced entity resolution leverages the crowd's ability to address this problem.*

For example, Table 1 shows a table with 4 attributes and 11 records. r_1, r_2, and r_3 refer to the same entity. r_4, r_5, r_6, and r_7 refer to the same entity. Each of r_8, r_9, r_{10}, r_{11} represents a different entity. Crowdsourced entity resolution asks questions to the crowd (or workers) for identifying the records referring to the same entity. As we need to pay the workers for answering a question, the objective is to reduce the number of questions while keeping high quality.

2.2 Related Work

2.2.1 Crowdsourced Entity Resolution

Generating Questions for Workers. An important problem in crowdsourced entity resolution is to design questions for workers. A straightforward method is to generate *pair-comparison-based questions*, where each question is a pair of two records and asks workers to check whether the two records refer to the same entity. This method may generate a large number of questions. To address this problem, *clustering-based questions* are proposed [12, 22], where each question is a group of records and asks workers to classify the records into different clusters such that records in the same cluster refer to the same entity and records in different clusters refer to different entities. As the clustering-based method does not need to enumerate every pair, it can reduce the monetary cost. However, workers prefer the pair-comparison question as it is much easier to answer.
Pruning Dissimilar Pairs. Intuitively, we do not need to ask the *dissimilar pairs* that have low probabilities referring to the same entity. Wang et al. [22] proposed a similarity-based method, which computed the similarity of record pairs and pruned the pairs with small similarities. As this method can prune many dissimilar pairs without sacrificing the quality of final answers, most of existing studies used this technique to reduce the cost.
Leveraging Transitivity to Reduce The Cost. Transitivity can be used to reduce the cost: Given three records, r_1, r_2, r_3, if $r_1 = r_2$ (r_1 and r_2 refer to the same entity) and $r_2 = r_3$, we can deduce that $r_1 = r_3$ and do not need to ask whether $r_1 = r_3$. Wang et al. [23] and Vesdapunt et al. [21] studied how to utilize the transitivity to reduce the number of questions. Although this method can reduce the cost, the quality may be reduced. For example, suppose $r_1 = r_2$ and $r_2 \neq r_3$, but the crowd returns $r_1 = r_2$ and $r_2 = r_3$. Then it introduces an incorrect deduction $r_1 = r_3$.
Improving The Quality. Wang et al. [24] proposed a correlation-clustering method, which includes three steps. It first prunes dissimilar pairs with small similarities. Then, it selects some pairs to ask and divides the records into a set of clusters based on the workers' results of these asked pairs. Finally, it refines the clusters by selecting more pairs to ask, checking whether their answers are consistent with the initial clusters, and adjusting the clusters based on the inconsistencies. This method improves the accuracy at the expense of huge monetary costs.
Question Selection. A natural problem is how to select next questions to ask in order to improve the quality. Whang et al. [25] proposed a probabilistic model to select high-quality questions. Verroios et al. [20] improved the model by tolerating workers' errors. Gokhale et al. [7] studied the crowdsourced record linkage problem, which linked two records from two tables, which is different from ours as we focus on linking multiple records in the same table.

Compared with existing techniques, our model can significantly reduce the cost while not sacrificing the quality.

2.2.2 Other Related Work

Crowdsourced Operators. There are many studies on leveraging crowd's ability to improve database operators, e.g., crowdsourced selection [1, 26], crowdsourced sort[18, 2], crowdsourced max/top-k [8, 19]. They focus on trading-off monetary cost, quality and latency.
Crowdsourced Systems. Several crowdsourced databases, e.g. Deco[16, 17], Quak[13], CrowdDB[5], were proposed, aiming to implement and optimize crowdsourced operators.
Crowdsourced Quality Control. Many methods are proposed to improve the quality[9, 15, 3, 11, 27]. Most of these studies focus on devising a worker model to capture worker's quality, computing the worker's model, eliminating bad workers, assigning questions to appropriate workers, and aggregating the results from multiple workers.

3. PARTIAL-ORDER-BASED FRAMEWORK

We first define a partial order (Section 3.1) and then propose a partial-order-based algorithm (Section 3.2).

3.1 Partial Order

Record Similarity. Given two records r_i and r_j, we use p_{ij} to denote the pair (r_i, r_j) and use s_{ij}^k to denote the similarity of p_{ij} on attribute $\mathcal{A}_k$. We can utilize any similarity function to compute the similarity, e.g., edit distance, Jaccard, Euclidean distance. Here we take Jaccard and edit similarity as examples. Let $r_i[k]$ denote the value of r_i on attribute $\mathcal{A}_k$. For Jaccard, we tokenize $r_i[k]$ into a set of

	Name ($\mathcal{A}_1$)	Address ($\mathcal{A}_2$)	City ($\mathcal{A}_3$)	Flavor ($\mathcal{A}_4$)
r_1	ritz-carlton restaurant (atlanta)	181 w. peachtree st.	atlanta	european french
r_2	ritz-carlton restaurant	181 peachtree dr	atlanta	european(french)
r_3	ritz-carlton restaurant Georgia	181 peachtree st.	city of atlanta	european France
r_4	cafe ritz-carlton buckhead	3434 peachtree rd.	city of atlanta	american
r_5	cafe ritz-carlton (buckhead)	3434 peachtree rd.	city of atlanta	american
r_6	dining room ritz-carlton buckhead	3434 peachtree ave.	atlanta	international
r_7	dining room ritz-carlton (buckhead)	3434 peachtree ave.	atlanta	international
r_8	cafe claude	201 83rd st.	new york	cafe
r_9	cafe bizou (american)	13 54th st.	new york	american food
r_{10}	gotham bar & grill	12th rd.	new york	american(new)
r_{11}	mesa grill	102 5th rd.	new york	southwestern

Table 1: Eleven Records In A Real Restaurant Dataset.

p_{ij}	s^1_{ij}	s^2_{ij}	s^3_{ij}	s^4_{ij}	p_{ij}	s^1_{ij}	s^2_{ij}	s^3_{ij}	s^4_{ij}
p_{12}	0.72	0.4	1	0.88	p_{37}	0.28	0.2	0.33	0
p_{13}	0.75	0.75	0.33	0.8	p_{45}	0.92	1	1	1
p_{23}	0.77	0.5	0.33	0.69	p_{46}	0.69	0.5	0.33	0
p_{24}	0.51	0.2	0.33	0	p_{47}	0.65	0.5	0.33	0
p_{25}	0.53	0.2	0.33	0	p_{56}	0.63	0.5	0.33	0
p_{26}	0.42	0.2	1	0	p_{57}	0.71	0.5	0.33	0
p_{27}	0.45	0.2	1	0	p_{67}	0.94	1	1	1
p_{34}	0.39	0.2	1	0	p_{89}	0.33	0.2	1	0
p_{35}	0.39	0.2	1	0	$p_{10,11}$	0.5	0.25	1	0

Table 2: Record Similarity.

tokens and compute Jaccard on token sets as below.

$$s^k_{ij} = \text{JAC}(r_i[k], r_j[k]) = \frac{|r_i[k] \cap r_j[k]|}{|r_i[k] \cup r_j[k]|}, \quad (1)$$

where $|r_i[k]|$ is the token-set size of $r_i[k]$.

For edit similarity, we first compute their edit distance, which is the minimum number of edit operations (insertion, deletion, substitution) required to transform one string to the other, and then compute the edit similarity as below.

$$s^k_{ij} = \text{EDS}(r_i[k], r_j[k])) = 1 - \frac{\text{ED}(r_i[k], r_j[k])}{\max(|r_i[k]|, |r_j[k]|)}, \quad (2)$$

where EDS(ED) is the edit similarity (distance) function.

For example, we use the edit similarity on attributes $\mathcal{A}_1$ and $\mathcal{A}_4$, and Jaccard on attributes $\mathcal{A}_2$ and $\mathcal{A}_3$. For instance, $s^1_{12} = 1 - \frac{9}{33} = 0.72$, and $s^2_{12} = \frac{2}{5} = 0.4$. As discussed in Section 2.2, we do not need to consider pairs whose similarities are smaller than a similarity bound τ, as they have small probabilities to be the same entity. Formally, we only consider the *similar pair* p_{ij} such that (1) $s_{ij} = \text{JAC}(r_i, r_j) \geq \tau$ for Jaccard, where $\text{JAC}(r_i, r_j)$ is the Jaccard similarity on the token sets of r_i and r_j ; or (2) $s_{ij} = \text{EDS}(r_i, r_j) \geq \tau$ for edit similarity, where $\text{EDS}(r_i, r_j)$ is the edit similarity on records r_i and r_j. The similar record pairs with $\tau = 0.2$ are shown in Table 2. If $s^k_{ij} < \tau$, we set $s^k_{ij} = 0$ for simplicity.

Partial Order. We define a partial order on record pairs. Given two pairs $p_{ij} = (r_i, r_j)$, $p_{i'j'} = (r_{i'}, r_{j'})$, $p_{ij} \succeq p_{i'j'}$, if (r_i, r_j) has no smaller similarities than $(r_{i'}, r_{j'})$ on every attribute. $p_{ij} \succ p_{i'j'}$, if $p_{ij} \succeq p_{i'j'}$ and (r_i, r_j) has larger similarities on at least one attribute than $(r_{i'}, r_{j'})$. Formally,

$$p_{ij} \succeq p_{i'j'} \quad \text{if } s^k_{ij} \geq s^k_{i'j'} \text{ for } 1 \leq k \leq m \quad (3)$$

$$p_{ij} \succ p_{i'j'} \quad \text{if } p_{ij} \succeq p_{i'j'} \text{ and } \exists k, s^k_{ij} > s^k_{i'j'} \quad (4)$$

For example, in Table 2, $p_{34} \succeq p_{35}$, $p_{27} \succ p_{34}$, and $p_{27} \succ p_{35}$.

3.2 Graph-Based Algorithm

We model the pairs as a graph based on the partial order.

DEFINITION 2 (GRAPH MODEL). *Given a table $\mathcal{T}$, we build a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each vertex in $\mathcal{V}$ is a similar record pair. Given two pairs p_{ij} and $p_{i'j'}$, if $p_{ij} \succ p_{i'j'}$, there is a directed edge in $\mathcal{E}$ from p_{ij} to $p_{i'j'}$.*

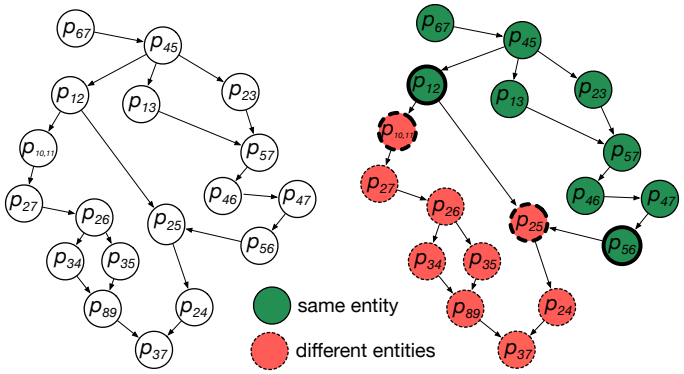


Figure 1: Partial Order and Graph Model.

Figure 1 shows the graph for the pairs in Table 1. In the figure, we do not show all the edges for illustration purpose: given two vertices, if there is already a path between them, we do not show the direct edge between them. For example, there should be an edge between p_{67} and p_{12}, but we omit it as there is already a path from p_{67} to p_{12}.

Graph Coloring. Each vertex in $\mathcal{G}$ has two possibilities: (1) they refer to the same entity and we color it GREEN; (2) they refer to different entities and we color it RED. Initially each vertex is uncolored. Our goal is to utilize the crowd to color all vertices. A straightforward method is to take the record pair on each vertex as a question and ask workers to answer the question, i.e. whether the two records in the pair refer to the same entity. If a worker thinks that the two records on the vertex refer to the same entity, the worker returns `Yes`; `No` otherwise. For each pair, to tolerate the noisy results from workers, we assign it to multiple workers, say 5. Based on the workers' results, we get a voted answer on each vertex. If majority workers vote `Yes`, we color it GREEN; otherwise we color it RED. Next, we interchangeably use vertex, pair and question if the context is clear.

Obviously this method is rather expensive as there are many vertices on the graph. To address this issue, we propose an effective coloring framework to reduce the number of questions. Algorithm 1 shows the pseudo code. It first computes the partial orders between pairs and constructs a graph (line 1). Then it selects an uncolored vertex p_{ij} (line 3) and asks workers to answer `Yes` or `No` on the vertex, (1) If majority workers vote `Yes`, we not only color p_{ij} GREEN, but also color all of its ancestors GREEN (line 5). In other words, for $p_{i'j'} \succ p_{ij}$, we also take $r_{i'}$ and $r_{j'}$ as the same entity. This is because $p_{i'j'}$ has larger similarity on every attribute than p_{ij}, and since r_i and r_j refer to the same entity (denoted by $r_i = r_j$), we deduce that $r_{i'} = r_{j'}$.
(2) If majority workers vote `No`, we not only color p_{ij} RED, but also color all of its descendants RED (line 7). In other words, for $p_{ij} \succ p_{i'j'}$, we also take $r_{i'}$ and $r_{j'}$ as different

Algorithm 1: A Partial-Order-Based Framework

Input: $\mathcal{T} = \{r_1, r_2 \cdots, r_n\}$
Output: All vertices are colored as GREEN or RED

```
1 Construct G = (V, E) based on partial orders;
2 while there exist uncolored vertices in V do
3     Select an uncolored vertex p_ij to ask workers;
4     if majority workers vote Yes then
5         color p_ij and p_i'j' (p_i'j' ≻ p_ij) GREEN;
6     else
7         color p_ij and p_i'j' (p_ij ≻ p_i'j') RED;
8 return colored V;
```

entities. This is because $p_{i'j'}$ has smaller similarity on every attribute than p_{ij}, and since r_i and r_j refer to different entities (denoted by $r_i \neq r_j$), we deduce that $r_{i'} \neq r_{j'}$.

If all the vertices have been colored, the algorithm terminates (line 7); otherwise, it selects an uncolored vertex and repeats the above steps (lines 2-7).

Obviously, this method can reduce the cost as we can avoid asking many unnecessary vertices. For example, consider the constructed graph in Figure 1. A naive method is to ask all eighteen pairs. However, if we first ask $p_{10,11}$, as majority workers vote `No`, we can color $p_{10,11}$ and its descendants p_{27}, p_{26}, p_{34}, p_{35}, p_{89} and p_{37} RED without needing to ask these descendants. Then if we select p_{56}, as majority workers vote `Yes`, we color p_{56} and its ancestors p_{46}, p_{47}, p_{57}, p_{23}, p_{45}, p_{67} and p_{13} GREEN without needing to ask them. In Section 5, we will show that we need to ask at least 4 questions (e.g., $p_{12}, p_{10,11}, p_{25}, p_{56}$) to color all vertices.

There are several challenges in this algorithm.
(1) Graph Construction. As there are large numbers of pairs, how to efficiently construct the graph? Can we reduce the graph size so as to reduce the number of questions?
(2) Question Selection. How to select the minimum number of vertices to ask in order to color all vertices?
(3) Error Tolerant. The coloring strategy and the workers may introduce errors. So how to tolerate the errors?

We address these challenges in the following sections.

4. GRAPH CONSTRUCTION

We first propose efficient graph-construction algorithms (Section 4.1) and then present grouping methods (Section 4.2).

4.1 Graph Construction Algorithms

Brute-Force Method. It enumerates every pair of vertices and checks whether they satisfy the partial order. If so, the algorithm adds an edge between them. The complexity of this method is $\mathcal{O}(|\mathcal{V}|^2)$. Obviously this method is rather expensive, especially if there are a large number of vertices.

Quicksort-Based Method. Quicksort is an efficient algorithm for the sorting problem and it can be extended to construct the graph. We first randomly select a vertex p_{ij} as pivot, and then split other vertices into three disjoint parts by comparing them with p_{ij}:
(1) *Parent Vertex Set:* $\mathcal{P}(p_{ij}) = \{p_{i'j'} | p_{i'j'} \succ p_{ij}\}$. For each $p_{i'j'}$ in $\mathcal{P}(p_{ij})$, we add an edge from $p_{i'j'}$ to p_{ij};
(2) *Child Vertex Set:* $\mathcal{C}(p_{ij}) = \{p_{i'j'} | p_{ij} \succ p_{i'j'}\}$. For each $p_{i'j'}$ in $\mathcal{C}(p_{ij})$, we add an edge from p_{ij} to $p_{i'j'}$;
(3) *Incomparable Vertex Set:* $\mathcal{U}(p_{ij}) = \mathcal{V} - \mathcal{P}(p_{ij}) - \mathcal{C}(p_{ij}) = \{p_{i'j'} | p_{ij} \not\succ p_{i'j'} \ \& \ p_{i'j'} \not\succ p_{ij}\}$. For each $p_{i'j'}$, there is no edge between p_{ij} and $p_{i'j'}$, as they are incomparable.

Obviously, $\forall p \in \mathcal{P}(p_{ij}), p' \in \mathcal{C}(p_{ij}), p \succ p'$, and thus we do not need to compare the pairs in $\mathcal{P}(p_{ij}) \times \mathcal{C}(p_{ij})$. Then, we consider the pairs in $(\mathcal{P}(p_{ij}) \cup \mathcal{U}(p_{ij})) \times (\mathcal{P}(p_{ij}) \cup \mathcal{U}(p_{ij}))$ and $(\mathcal{C}(p_{ij}) \cup \mathcal{U}(p_{ij})) \times (\mathcal{C}(p_{ij}) \cup \mathcal{U}(p_{ij}))$. To add edges between these pairs, we can recursively utilize the above method.[1] The worst-case complexity of this method is also $\mathcal{O}(|\mathcal{V}|^2)$ if all the vertices are incomparable. However, this method has better performance than brute-force in practice, because it can prune many unnecessary pairs (e.g., $\mathcal{P}(p_{ij}) \times \mathcal{C}(p_{ij})$).

Index-Based Method. The quicksort-based method still has poor performance for large datasets. To address this issue, we propose an index-based method. As the similarity s^k_{ij} is a numerical value, we can utilize geometric relationship to compare two pairs. For simplicity, we first assume there are two attributes ($m = 2$). So the similarity of p_{ij} has two components s^1_{ij} and s^2_{ij}. Therefore, we can map each vertex to a point in a two-dimensional coordinate as shown in Figure 2(a).

If we want to find the child set of p_{ij}, $\mathcal{C}(p_{ij}) = \{p_{i'j'} | p_{ij} \succ p_{i'j'}\}$, we report the left-bottom vertices (i.e., vertices in the rectangle). Similarity, if we compute $\mathcal{P}(p_{ij}) = \{p_{i'j'} | p_{i'j'} \succ p_{ij}\}$, we report the top-right vertices. We can utilize the 2-dimensional range trees to achieve this goal [10].

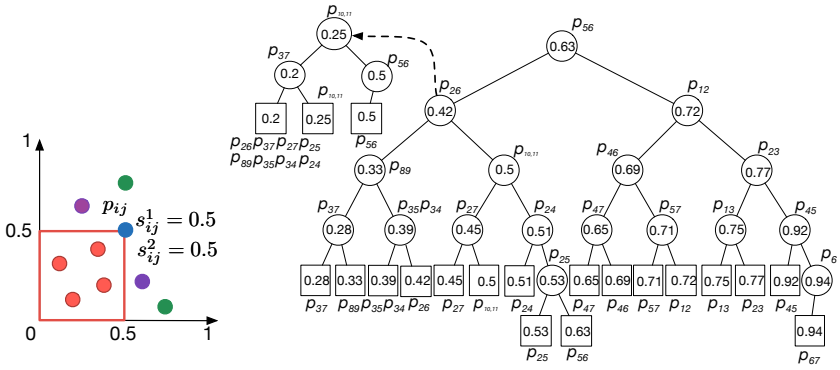
Range Search Tree Construction. We first construct a first-level balanced binary tree based on s^1_{ij} for all vertices as shown in Figure 2(b), where leaves are vertices in $\mathcal{V}$ and the internal nodes are guided search values. (There are multiple pairs in a node because they have the same similarity. For example, p_{34}, p_{35} are in the same node, because $s^1_{34} = s^1_{35} = 0.39$.) The value of a node is the largest s^1_{ij} for all vertices in its left subtree, and thus the s^1_{ij} values of vertices under the left subtree are not larger than the value of this node; while the s^1_{ij} values of vertices under its right subtree are larger than the value. We can build the binary tree in a bottom-up way. For each internal node, we construct the second-level balanced binary tree based on s^2_{ij} for vertices under this node.

Reporting $\mathcal{C}(p_{ij})$ with Range Search Tree. Given a vertex p_{ij}, we use the range search tree to report $\mathcal{C}(p_{ij})$. We first find the tree nodes whose descendants' similarities on $\mathcal{A}_1$ are not larger than s^1_{ij} using the first-level tree. For each of such *qualified* nodes on $\mathcal{A}_1$, we visit its second-level tree and find the nodes whose descendants' similarities on $\mathcal{A}_2$ are not larger than s^2_{ij}. Then the vertices under these nodes are added into $\mathcal{C}(p_{ij})$. Next we discuss how to find such qualified nodes in the first-level tree and the same techniques can be used to search the second-level tree.

To find the qualified nodes on $\mathcal{A}_1$, we search the first-level tree from the root. For each node, (1) If its value is not larger than s^1_{ij}, (1.1) if it is a leaf, it is a qualified node; (1.2) if it is not a leaf, the similarities of all the vertices under its left child on attribute $\mathcal{A}_1$ are not larger than s^1_{ij}, and its left child is a qualified node. Next we recursively process its right child; (2) If its value is larger than s^1_{ij}, (2.1) if it is a leaf, we prune it; (2.2) if it is not a leaf, we prune its right subtree as the similarities of all the vertices under its right child on attribute $\mathcal{A}_1$ must be larger than s^1_{ij}. Next we recursively process its left child. Iteratively, we can identify all qualified nodes on $\mathcal{A}_1$. This method accesses at most $\log(|\mathcal{V}|)$ nodes in the first-level tree.

For example, suppose we want to compute $\mathcal{C}(p_{12})$ where $s^1_{12} = 0.72$ and $s^2_{12} = 0.4$. We first compare s^1_{12} with the root $s^1_{56} = 0.63$. As $s^1_{12} > s^1_{56}$, its left child (i.e., p_{26}) is

[1] Note to avoid duplicately comparing two pairs in $\mathcal{U}(p_{ij})$, we can only select pivots from $\mathcal{C}(p_{ij})$ and $\mathcal{P}(p_{ij})$.



(a) $2d$ coordinate (b) Rang Search Tree

Figure 2: Index-based Graph Construction.

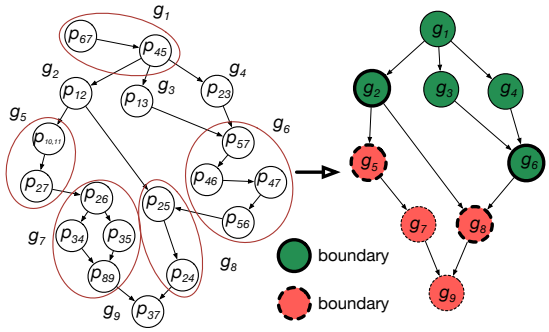


Figure 3: Vertex Grouping.

a qualified node. Next we go to the right child p_{12}. As $s^1_{12} = s^1_{12}$, we visit its left child p_{46}. As $s^1_{12} > s^1_{46}$, its left child p_{47} is a qualified node. Next we go to the right child p_{57}. As $s^1_{12} > s^1_{57}$, its left child p_{57} is a qualified node and we go to its right node p_{12}. As p_{12} is a leaf, it is a qualified node. Next for each qualified node $(p_{26}, p_{47}, p_{57}, p_{12})$, we check it on the second attribute. Take p_{26} as an example. As $s^2_{12} = 0.4$ is larger than the root's value, its left child p_{37} is a qualified node. We then visit its right child p_{56}. As $s^2_{12} \leq s^2_{56}$, we go to its left child which is a leaf. As the value is larger than s^2_{12}, we prune it. Thus the pairs under node p_{37} are added into $\mathcal{C}(p_{12})$.

Building The Graph with Range Search Tree. For each vertex p_{ij}, we use the range search tree to find $\mathcal{C}(p_{ij})$ and add vertices in $\mathcal{C}(p_{ij})$ as the children of p_{ij}. Then we can build the graph. It is straightforward to generalize 2-dimensional range trees to m-dimensional range trees.

Complexity. Both the time and space complexities of constructing the tree is $\mathcal{O}(|\mathcal{V}| \log^{m-1} |\mathcal{V}|)$. The time complexity of computing $\mathcal{C}(p_{ij})$ is $\mathcal{O}(\log^m |\mathcal{V}| + |\mathcal{C}(p_{ij})|)$, where $|\mathcal{C}(p_{ij})|$ is the size of $\mathcal{C}(p_{ij})$. After using the fractional cascading technique [10], the complexity is reduced to $\mathcal{O}(\log^{m-1} |\mathcal{V}| + |\mathcal{C}(p_{ij})|)$. Thus the overall time complexity of constructing the graph is $\mathcal{O}(|\mathcal{V}| \log^{m-1} |\mathcal{V}| + |\mathcal{E}|)$.

4.2 Vertex Grouping

Note that some vertices have very close similarities and we can combine them to reduce the graph size, which not only reduces the cost but also saves the graph construction cost. For example, p_{67} and p_{45} have close similarities on the four attributes, i.e., p_{67}:(0.94, 1, 1, 1) and p_{45}:(0.92, 1, 1, 1) as shown in Table 2. Thus we can combine them as a single vertex. Next we formulate the problem.

Definition 3 (Vertex Group). *Given a threshold ε, a subset $g \subseteq \mathcal{V}$ is called a vertex group, if for any pairs p_{ij} and $p_{i'j'}$ in g, $|s^k_{ij} - s^k_{i'j'}| \leq \varepsilon$ for $1 \leq k \leq m$.*

As the similarities between different pairs in a group should not have large gap, we use ε to set a constraint. For example, suppose $\varepsilon = 0.1$. $\{p_{26}, p_{34}, p_{35}\}$ is a group as the difference of their similarities on every attribute is smaller than 0.1 (p_{26}:(0.42,0.2,1,0), p_{34}:(0.39,0.2,1,0), p_{35}:(0.39,0.2,1,0)).

Next we partition the vertices into different groups.

Definition 4 (Grouping Strategy). *Given a set of vertices $\mathcal{V}$, a grouping strategy is a partition of $\mathcal{V}$ to generate a set of groups $g_1, g_2, \ldots, g_x$, which satisfies,*
(1) Complete: For any $p_{ij} \in \mathcal{V}$, $\exists g_t, p_{ij} \in g_t$; and
(2) Disjoint: For any two groups g_i, g_j, $g_i \cap g_j = \phi$.

For example, consider the eighteen pairs in Table 1. Given threshold $\varepsilon = 0.1$, the groups $\{p_{67}$,$p_{45}\}$, $\{p_{12}\}$, $\{p_{13}\}$, $\{p_{23}\}$, $\{p_{10,11},\ p_{27}\}$, $\{p_{57}$,p_{47},p_{46},$p_{56}\}$,$\{p_{24}$,$p_{25}\}$, $\{p_{26}$,p_{34},p_{89},$p_{35}\}$, $\{p_{37}\}$ satisfy the two constraints.

Partial Order on Groups. We can define the partial order on the groups. For any two groups g_i and g_j,

$$g_i \succeq g_j \quad \text{if } \forall p \in g_i, p' \in g_j, p \succeq p' \tag{5}$$
$$g_i \succ g_j \quad \text{if } \forall p \in g_i, p' \in g_j, p \succ p' \tag{6}$$

Let $g^k.l / g^k.u$ denote the smallest/largest similarity of pairs in g on $\mathcal{A}_k$, i.e., $g^k.l = \min_{p_{ij} \in g} s^k_{ij}$ and $g^k.u = \max_{p_{ij} \in g} s^k_{ij}$. We can prove that if $g^k_i.l \geq g^k_j.u$ for $1 \leq k \leq m$, $g_i \succeq g_j$; if $g^k_i.l \geq g^k_j.u$ and $\exists k\ g^k_i.l > g^k_j.u$, $g_i \succ g_j$. Thus we can use $g^k_i.l$ and $g^k_j.u$ to easily determine the partial orders of two groups. Given a set of groups, if $g_i \succ g_j$, we add an edge from g_i to g_j. Then we can construct a grouped graph.

Definition 5 (Grouped Graph). *Given a set of vertices $\mathcal{V}$ and a set of groups $g_1, g_2, \ldots, g_x$ generated using the grouping strategy, we construct a grouped graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$, where each vertex in $\mathcal{V}'$ is a group, and there is an edge in $\mathcal{E}'$ from g_i to g_j if $g_i \succ g_j$.*

Coloring The Grouped Graph. We ask workers to color the grouped graph. If a group is selected to ask, we randomly select a pair in the group and take the answer of this pair as the answer of the group. Then we utilize our coloring algorithm (Section 3.2) to color the grouped graph.

Optimal Group Generation. There are multiple grouping strategies. We quantify how good a grouping strategy is. Obviously, the smaller the number of vertices in the grouped graph is, the lower the cost is. Thus we aim to generate the minimum number of groups.

Definition 6. *(Optimal Group Generation). Given a set of vertices $\mathcal{V}$ and a threshold ε, we aim to generate the minimum number of groups.*

We can prove that the optimal group generation problem is NP-hard as proved in Theorem 1.

Theorem 1. *The optimal group generation problem is NP-Hard. (See Appendix D.1 for the proof.)*

We propose a greedy algorithm and a heuristic algorithm.

Greedy Algorithm. The basic idea is that we first generate all the maximal groups, which are defined as below.

Definition 7 (Maximal Group). *A group g is called a maximal group if $\forall p_{ij} \in \mathcal{V} - g$, $g \cup \{p_{ij}\}$ is not a group (i.e., it does not satisfy the ε-constraint in Definition 3.).*

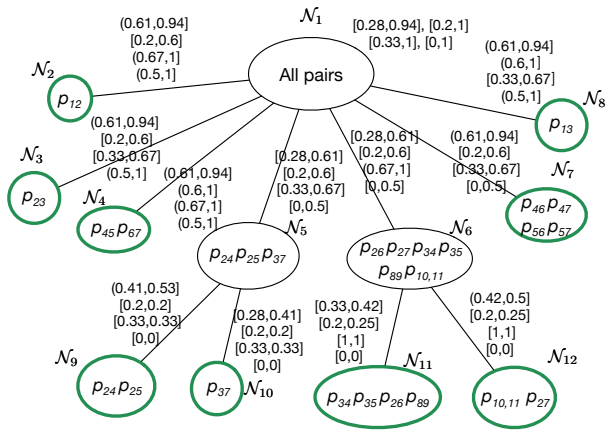


Figure 4: The Group Tree.

For example, $\{p_{26}, p_{34}, p_{35}\}$ is a group, but it is not a maximal group, because if we add p_{89}, $\{p_{26}, p_{34}, p_{35}, p_{89}\}$ is still a group satisfying Definition 3, which contradicts with Definition 7. $\{p_{26}, p_{34}, p_{35}, p_{89}\}$ is a maximal group, as we cannot add any pair to form a new group.

Next we introduce a greedy algorithm (see more details in Appendix A). We first generate the set of all maximal groups, denoted by $\mathcal{M}$. Then we greedily pick the largest group g in $\mathcal{M}$ with the maximum number of vertices. For each g_i in $\mathcal{M}$, we remove the vertices in g from g_i and update g_i to g_i-g. (If g_i-g is empty, we remove it from $\mathcal{M}$.) Next we iteratively pick the largest group from $\mathcal{M}$ until $\mathcal{M}$ is empty. This greedy algorithm has a $\ln(|\mathcal{V}|)$ approximation ratio. However it is expensive to generate the maximal groups and the complexity of this greedy algorithm is $\mathcal{O}(|\mathcal{V}|^m)$.

For example, we want to group the vertices in Figure 1. Firstly, we generate all the maximal groups $\mathcal{M}$ ={$\{p_{67}, p_{45}\}$, $\{p_{12}\}$, $\{p_{13}\}$, $\{p_{23}\}$, $\{p_{10,11}, p_{27}, p_{26}\}$, $\{p_{27}, p_{26}, p_{34}, p_{35}\}$, $\{p_{26}, p_{34}, p_{35}, p_{89}\}$, $\{p_{47}, p_{57}, p_{46}, p_{56}\}$, $\{p_{24}, p_{25}\}$, $\{p_{37}\}$}. Then we select the largest group $\{p_{27}, p_{26}, p_{34}, p_{35}\}$ from the maximal group set as a group. Next we remove vertices in it from other maximal groups. Now $\mathcal{M}$ ={$\{p_{67}, p_{45}\}$, $\{p_{12}\}$, $\{p_{13}\}$, $\{p_{23}\}$, $\{p_{10,11}\}$, $\{p_{89}\}$, $\{p_{47}, p_{57}, p_{46}, p_{56}\}$, $\{p_{24}, p_{25}\}$, $\{p_{37}\}$}. Then we select the largest group. Finally the groups are $\mathcal{M}$ ={$\{p_{67}, p_{45}\}$, $\{p_{12}\}$, $\{p_{13}\}$, $\{p_{23}\}$, $\{p_{10,11}\}$, $\{p_{27}, p_{26}, p_{34}, p_{35}\}$, $\{p_{89}\}$, $\{p_{47}, p_{57}, p_{46}, p_{56}\}$, $\{p_{24}, p_{25}\}$, $\{p_{37}\}$}.

Split-Based Algorithm. As the greedy algorithm is expensive, we propose an efficient algorithm. The basic idea is that we first take all the pairs as a group, and if any attribute does not satisfy the threshold constraint, we partition the group based on this attribute. The pseudo code is shown in Algorithm 2. Formally, we build a tree structure and the root is $\mathcal{N}_1 = \mathcal{V}$. Let $\mathcal{N}_1^i.l/\mathcal{N}_1^i.u$ denote the minimal/maximal similarity of pairs in $\mathcal{N}_1$ on attribute $\mathcal{A}_i$. If $\mathcal{N}_1^i.u - \mathcal{N}_1^i.l > \varepsilon$, we split $\mathcal{N}_1$ based on $\mathcal{A}_i$ and generate two ranges $[\mathcal{N}_1^i.l, \frac{\mathcal{N}_1^i.l+\mathcal{N}_1^i.u}{2}]$, $(\frac{\mathcal{N}_1^i.l+\mathcal{N}_1^i.u}{2}, \mathcal{N}_1^i.u]$; otherwise, we do not split $\mathcal{N}_1$ based on this attribute. Suppose we split $\mathcal{N}_1$ based on $\mathcal{A}_{i_1}, \mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_t}$. We generate 2^t children of $\mathcal{N}_1$ by enumerating the two ranges of these attributes. For each node, we add the pairs that fall in the corresponding ranges into the node. If a node cannot be split on any attribute, it is a leaf. Finally the groups on leaves are the result.

For example, we walk through our algorithm on the records in Table 2. Suppose $\varepsilon = 0.1$. Figure 4 shows the group tree. Firstly, the root $\mathcal{N}_1$ ($[\mathcal{N}_1^1.l, \mathcal{N}_1^1.u]$,$[\mathcal{N}_1^2.l, \mathcal{N}_1^2.u]$, $[\mathcal{N}_1^3.l, \mathcal{N}_1^3.u]$, $[\mathcal{N}_1^4.l, \mathcal{N}_1^4.u]$) is denoted as ($[0.28, 0.94]$, $[0.2, 1]$, $[0.33, 1]$, $[0, 1]$) in Figure 4. As $\mathcal{N}_1^i.u - \mathcal{N}_1^i.l > \varepsilon$ for $i \in [1, 4]$, we split $[\mathcal{N}_1^1.l, \mathcal{N}_1^1.u]$, $[\mathcal{N}_1^2.l, \mathcal{N}_1^2.u]$, $[\mathcal{N}_1^3.l, \mathcal{N}_1^3.u]$ and $[\mathcal{N}_1^4.l, \mathcal{N}_1^4.u]$ into

Algorithm 2: Vertex Grouping: `Split`

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: A set of groups $g_1, g_2, ..., g_x$

```
N_1 ← V; Priority queue Q = {N_1};
while Q is not empty do
    Pop node N_i from Q;
    for k ∈ [1, m] do
        if N_i^k.u − N_i^k.l > ε then
            Split N_i based on A_k ;
    if N_i is split by A_{i_1}, A_{i_2}, ..., A_{i_t} then
        Generate 2^t children of N_i;
        Move pairs in N_i into corresponding children;
        Add these children into Q;
    else
        N_i is a leaf and taken as a group g;
return the groups on the leaves;
```

$\langle[0.28, 0.61], (0.61, 0.94]\rangle$, $\langle[0.2, 0.6], (0.6, 1]\rangle$, $\langle[0.33, 0.67], (0.67, 1]\rangle$ and $\langle[0, 0.5]\ (0.5, 1]\rangle$ respectively. Then we move each pair in $\mathcal{N}_1$ into the 2^4 children (empty children are removed). For $i \in [1, 4]$, s_{45}^i and s_{67}^i are in the range of $(0.61, 0.94]$, $(0.6, 1]$, $(0.67, 1]$, $(0.5, 1]$, and p_{45} and p_{67} are added into $\mathcal{N}_4$. Then we calculate $\mathcal{N}_4^i.l, \mathcal{N}_4^i.u$ and get ($[0.92, 0.94]$, $[1, 1]$, $[1, 1]$, $[1, 1]$). As each range is smaller than ε, $\mathcal{N}_4 = \{p_{45}, p_{67}\}$ is a leaf. Next, we move $\{p_{24}, p_{25}, p_{37}\}$ into $\mathcal{N}_5$ ($[0.28, 0.53]$, $[0.2, 0.2]$, $[0.33, 0.33]$, $[0, 0]$). It is not a group and split again. As $[\mathcal{N}_5^2.u - \mathcal{N}_5^2.l] < \varepsilon$, $[\mathcal{N}_5^3.u - \mathcal{N}_5^3.l] < \varepsilon$ and $[\mathcal{N}_5^4.u - \mathcal{N}_5^4.l] < \varepsilon$, we split $\mathcal{N}_5^1$ and get two leaves $\mathcal{N}_9$ and $\mathcal{N}_{10}$. At last, we get 9 groups (as shown in Figure 3).

Complexity. The tree has at most $\log \frac{1}{\varepsilon}$ levels. Thus the time complexity of constructing the tree is $\mathcal{O}(|\mathcal{V}| \log \frac{1}{\varepsilon})$.

5. QUESTION SELECTION

An important problem is to select the minimum number of vertices as questions to color all vertices. We first formulate the question-selection problem (Section 5.1,) and then propose a serial algorithm that selects one vertex in each iteration (Section 5.2) and parallel algorithms that select multiple vertices in each iteration (Section 5.3).

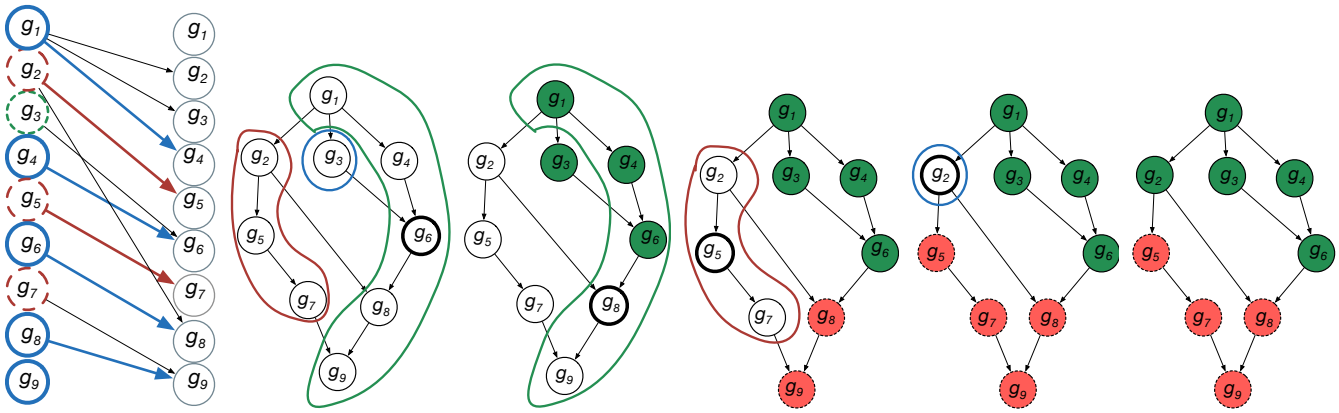
5.1 Optimal Vertex Selection

We first assume that (1) if a vertex is Green, then all of its ancestors are Green; and (2) if a vertex is Red, then all of its descendants are Red. We will discuss how to support the case that the two conditions do not hold in Section 6.

Definition 8 (Optimal Graph Coloring). *Given a graph, the optimal graph coloring problem aims to select the minimum number of vertices as questions to color all the vertices using the coloring strategy.*

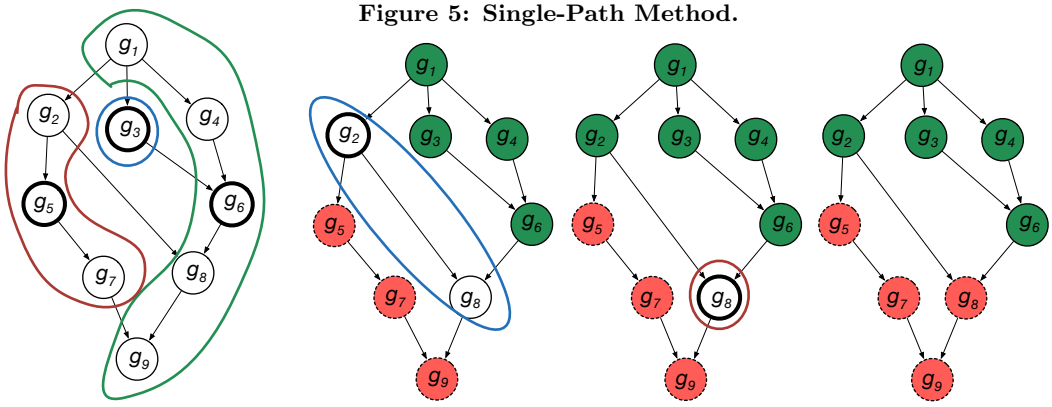
For example, in Figure 3, if we sequetially select vertices g_8, g_7, g_5, g_2, g_3, g_4 and g_6, we ask 7 questions. The optimal crowdsourced vertices are g_2, g_5, g_6 and g_8 (highlighted by bold circles), because the colors of these vertices cannot be deduced based on the colors of other vertices. Next we study how to identify the optimal vertices. We first introduce a notation for ease of presentation.

Definition 9 (Boundary Vertex). *A vertex is a boundary vertex if its color cannot be deduced based on other vertices' colors. There are four cases: (1) all of its parents have different colors with the vertex; (2) all of its children have different colors with the vertex; (3) it has no child and its color is* Green*; or (4) it has no parent and its color is* Red*.*



(a) Disjoint Paths (b) 3 Paths, Ask g_6 (c) Ask g_8 (d) Ask g_5 (e) Ask g_2 (f) All Colored

Figure 5: Single-Path Method.



(a) Ask g_5, g_3, g_6 (b) Ask g_2 (c) Ask g_8 (d) All Colored

Figure 6: Multi-Path Method.

For example, g_6 is a boundary vertex as its child g_8 has different color with g_6. g_4 is not a boundary vertex as its child g_6 has the same color and g_4's color can be deduced based on g_6's color.

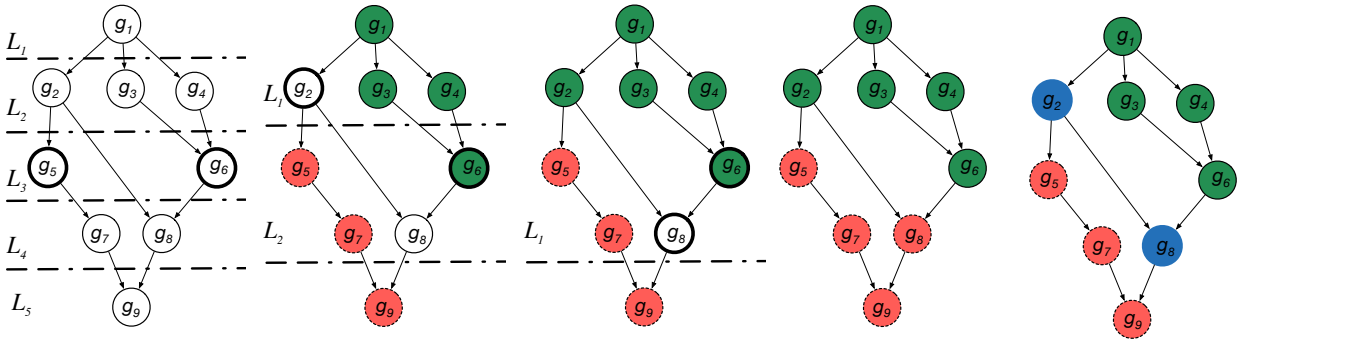
We can prove that all the boundary vertices must be asked, because their colors cannot be deduced. Thus the number of asked vertices using any algorithm is not smaller than the number of boundary vertices. However, as we do not know the ground truth, we cannot identify the boundary vertices in advance. To address this problem, we propose effective algorithms to identify the boundary vertices with theoretical guarantee.

5.2 Serial Algorithm

Comparable Vertices. Given any two vertices p_{ij}, $p_{i'j'}$, if they are *comparable*, i.e., $p_{ij} \succ p_{i'j'}$ or $p_{i'j'} \succ p_{ij}$, we may deduce p_{ij}'s color based on $p_{i'j'}$'s color, and vice versa. Obviously, two comparable vertices must be on a (directed) path in the graph, and the vertices on a path are totally ordered (i.e., any two vertices are comparable). Given a path, we can use a binary-search method to select the boundary vertices. Formally given a path, we first ask the mid-vertex on the path. (1) If the vertex is colored Green, its ancestors' colors can be deduced but its descendants' colors cannot be deduced, and thus we ask the mid-vertex between this vertex and the destination vertex of the path; (2) If the vertex is colored Red, its descendants' colors can be deduced but its ancestors' colors cannot be deduced, and thus we ask the mid-vertex between this vertex and the source vertex of the path. Iteratively, we can find the boundary vertices. For the path P with $|P|$ vertices, the number of asked vertices is $\mathcal{O}(\log |P|)$. This is optimal and cannot be improved in general. For example, $g_1 \rightsquigarrow g_4 \rightsquigarrow g_6 \rightsquigarrow g_8 \rightsquigarrow g_9$ is a path. We first ask the mid-vertex g_6. As g_6 is Green, we ask the mid-vertex between g_6 and g_9, i.e., g_8. As g_8 is Red, all the vertices are colored in the path.

Incomparable Vertices. If two vertices are incomparable, we cannot deduce one vertex's color based on the other vertex's color. Suppose there are $\mathcal{B}$ incomparable vertices (any two vertices are incomparable). We can divide the graph into $\mathcal{B}$ disjoint paths (i.e., any two paths have no common vertices). Then we can ask each path using the binary search method. As the maximum length of a path is $|\mathcal{V}|$, the number of asked vertices is $\mathcal{O}(\mathcal{B} \log |\mathcal{V}|)$. This is optimal and cannot be improved in general. This is because if $\mathcal{B} = 1$, we need to ask $\log |\mathcal{V}|$ vertices. For example, in Figure 5, we have 3 disjoint paths $g_1 \rightsquigarrow g_4 \rightsquigarrow g_6 \rightsquigarrow g_8 \rightsquigarrow g_9$, $g_2 \rightsquigarrow g_5 \rightsquigarrow g_7$, and g_3. We need to ask these paths using the binary-search algorithm.

Finding $\mathcal{B}$ Disjoint Paths. We transform the graph $\mathcal{G}$ into a bipartite graph $\mathcal{G}^b = ((\mathcal{V}_1^b, \mathcal{V}_2^b), \mathcal{E}^b)$, where $\mathcal{V}_1^b = \mathcal{V}_2^b = \mathcal{V}$ and there is an edge between $v_1 \in \mathcal{V}_1^b$ and $v_2 \in \mathcal{V}_2^b$ if there is an edge $(v_1, v_2) \in \mathcal{V}$. We find a maximal matching in $\mathcal{G}^b = ((\mathcal{V}_1^b, \mathcal{V}_2^b), \mathcal{E}^b)$, which is a maximal set of edges in $\mathcal{G}^b = ((\mathcal{V}_1^b, \mathcal{V}_2^b), \mathcal{E}^b)$ where any two edges do not share a common vertex in $\mathcal{V}_1^b$ and $\mathcal{V}_2^b$, i.e., for any two edges (v, v'), (u, u') in the matching, $v \neq u$ and $v' \neq u'$. Obviously any two edges in the matching sharing the same vertex in $\mathcal{V}$ must be on the same path, i.e., for any two edges (v, v'), (u, u') in the matching, if $v' = u$, then $v \rightsquigarrow v' = u \rightsquigarrow u'$ must be on the same path based on the partial order. Note that the maximal matching can be computed in $\mathcal{O}(\mathcal{B}|\mathcal{V}|^2)$[4]. Based



(a) Ask g_5, g_6 (b) Ask g_2 (c) Ask g_8 (d) All Colored

Figure 7: Topological-Sorting Based Method.

Figure 8: Error-Tolerant.

Algorithm 3: Question Selection: `SinglePath`

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: All vertices in $\mathcal{V}$ are colored as Green or Red

```
while there exist uncolored vertices in V do
    Compute disjoint paths using maximal matching;
    Color the longest path using binary search;
    Remove the colored vertices;
return colored V;
```

on this idea, we utilize the maximal matching to find the $\mathcal{B}$ disjoint paths as follows.

Let $\mathcal{Y}$ denote the maximal matching, $\mathcal{Y}_1$ denote the set of the first vertices in $\mathcal{Y}$ and $\mathcal{Y}_2$ denote the set of the second vertices in $\mathcal{Y}$. Then $\mathcal{V}_2^b - \mathcal{Y}$ is the set of vertices that have no in-edges, and we can take them as the first vertex of a path. For each such vertex v, if it has an edge (v, v'), we take v' as the second vertex in the path. Then we check whether v' has an edge (v', v''). Iteratively, we can find the path starting at v. The paths computed in our method satisfy: disjoint, complete and minimal, and the correctness is guaranteed by the following theorem.

Theorem 2. *The set of paths found by the maximal matching of $\mathcal{G}^b$ satisfy:*
(1) Disjoint: any two paths do not share a vertex;
(2) Complete: the paths contain all the vertices;
(3) Minimal: the size is exactly $\mathcal{B}$ and is not larger than the size of any other set of paths satisfying (1) and (2).

Proof. See Appendix D.2. The proof essentially follows the Fulkerson's proof of Dilworth theorem [6]. □

For example, consider the graph in Figure 3. We construct a bipartite graph as shown in Figure 5(a). As there is an edge from g_1 to g_3 in $\mathcal{G}$, there is an edge from g_1 in $\mathcal{V}_1^b$ to g_3 in $\mathcal{V}_2^b$. Thus $\mathcal{G}$ and $\mathcal{G}^b$ have the same number of edges. Then we find a maximal matching which is the set of the colored edges. The vertices g_1, g_2 and g_3 in $\mathcal{V}_2^b$ have no in-edges in the maximal matching. We compute the disjoint paths starting from them. From g_1 we get path $g_1 \rightsquigarrow g_4 \rightsquigarrow g_6 \rightsquigarrow g_8 \rightsquigarrow g_9$; from g_2 we get $g_2 \rightsquigarrow g_5 \rightsquigarrow g_7$; and g_3 itself is a path. Thus we get 3 disjoint paths.

SinglePath Algorithm. Then we propose a path-based question-selection algorithm. The pseudo code is shown in Algorithm 3. It first computes the $\mathcal{B}$ disjoint paths. Then it asks the longest path using the binary-search method, colors the graphs, and then removes the colored vertices. Next it recomputes the disjoint paths and asks the next longest path. Iteratively it can color all vertices. The complexity of this algorithm is $\mathcal{O}(\mathcal{B}|\mathcal{V}|^2)$.

For example, in Figure 5, we first identify the minimal disjoint paths as shown in Figure 5(a). Then we select the longest path (Figure 5(b)), ask the path using binary search. We first ask g_6 and color the graph based on the answers of asked vertices (Figure 5(c)). Next we ask g_8 and get Figure 5(d). Then we recompute the disjoint paths, ask mid-vertex of the longest path $g_2 \rightsquigarrow g_5 \rightsquigarrow g_7$ (Figure 5(d)), and color the graph (Figure 5(e)). Next as there is only one vertex left, we ask it and get the final result (Figure 5(f)). This method totally asks 4 vertices and involves 4 iterations.

5.3 Parallel Algorithm

If users do not care about the latency, the single-path algorithm is a good choice. However if the latency is very crucial, the single-path algorithm is not acceptable as it needs to post one question at a time on crowdsourcing platforms, which would result in a long time latency. To address this issue, we design parallel algorithms, which select multiple vertices and ask them together in each iteration.

5.3.1 Multi-Path Algorithm

We extend the path-based algorithm to support the parallel setting. The pseudo code is illustrated in Appendix B. We first identify the $\mathcal{B}$ disjoint paths and then ask their mid-vertices in parallel. Based on the answers on these vertices, we color the graph. Next we remove the colored vertices and repeat the above step until all the vertices are colored. Figure 6 shows an example. Note that the parallel algorithm may generate conflicts. For example, if g_i is colored Green and g_j is colored Red, then there is a conflict on g where $g \succ g_i$ and $g_j \succ g$, because g is deduced as Green based on g_i and deduced as Red based on g_j. To address this issue, we can use majority voting to vote g's color.

5.3.2 Topological-Sorting-Based Algorithm

In the multi-path algorithm, the asked vertices may have ancestor-descendent relationships, and thus it may ask unnecessary questions. For example, in Figure 6(a), we do not need to ask g_3 and g_6 together, as the color of g_3 can be deduced based on the color of g_6. To address this issue, we aim to ask independent vertices in each iteration.

To this end, we perform a topological sorting on the vertices. We first identify the set of vertices with zero in-degree, denoted by $\mathcal{L}_1$. Then we delete them from the graph and find another set of vertices whose in-degrees are zero, denoted by $\mathcal{L}_2$. We repeat this step until all vertices are deleted. Suppose there are $|\mathcal{L}|$ sets, $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_{|\mathcal{L}|}$. Obviously vertices in each $\mathcal{L}_i$ have no in-edges (as their in-degrees are 0) and thus can be taken as an independent set. Moreover, the vertices in the sets with small subscripts (e.g., $\mathcal{L}_1, \mathcal{L}_2$) are more

Algorithm 4: Question Selection:`TopologicalSorting`

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: All vertices in $\mathcal{V}$ are colored as GREEN or RED
while *there exist uncolored vertices in* $\mathcal{V}$ **do**
  Do a topological sorting on the uncolored vertices in $\mathcal{G}$ and obtain $|\mathcal{L}|$ sets, $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_{|\mathcal{L}|}$;
  Ask workers to color vertices in $\mathcal{L}_{\frac{|\mathcal{L}|+1}{2}}$;
return *colored* $\mathcal{V}$;

likely to be colored GREEN and the vertices in the sets with large subscripts (e.g., $\mathcal{L}_{|\mathcal{L}|}$) are more likely to be colored RED, and thus we cannot deduce the colors of many uncolored vertices based on them. In other words, the boundary vertices are more likely to be in the middle sets. To this end, we first ask vertices in $\mathcal{L}_{\frac{|\mathcal{L}|+1}{2}}$.

Next we design a topological-sorting-based algorithm and Algorithm 4 illustrates the pseudo code. It first computes topological-sorted sets $\mathcal{L}_1, \mathcal{L}_2, \cdots, \mathcal{L}_{|\mathcal{L}|}$. Then it asks vertices in $\mathcal{L}_{\frac{|\mathcal{L}|+1}{2}}$ in parallel. Based on the results of these vertices, it colors the graph, removes the colored vertices, and repeats the above step. Iteratively it colors all vertices.

For example, we construct the topological structure as shown in Figure 7(a). $\mathcal{L}_1 = \{g_1\}$,$\mathcal{L}_2 = \{g_2, g_3, g_4\}$,$\mathcal{L}_3 = \{g_5, g_6\}$, $\mathcal{L}_4 = \{g_7, g_8\}$,$\mathcal{L}_5 = \{g_9\}$ and $|\mathcal{L}| = 5$. So we select $\mathcal{L}_3 = \{g_5, g_6\}$ and ask the vertices. After getting their answers, we obtain Figure 7(b). Then we compute the topological sorting on the graph of the uncolored vertices. Next, $\mathcal{L}_1 = \{g_2\}, \mathcal{L}_2 = \{g_8\}$. We ask g_2. After this iteration, only g_8 is uncolored. We ask it and get the final result (Figure 7(d)). This method asks 4 vertices and has 3 iterations.

6. TOLERATING ERRORS

There are two types of possible errors in our framework. The first is caused by workers' errors and the second is introduced by our coloring strategy. For example, suppose a vertex p_{ij} is actually RED. However the workers wrongly color it GREEN. This error is caused by workers. Consider p_{ij}'s ancestor, $p_{i'j'}$, whose color is RED. Our coloring strategy will wrongly color it GREEN based on partial order. This error is caused by our coloring strategy. Next we discuss how to address these errors.

Confidence of Workers' Answers. To tolerate workers' errors, we assign each vertex to multiple workers and aggregate their answers. There are many methods to compute the confidence of workers' answers, and we take majority voting as an example and any other techniques can be integrated into our method. Suppose each vertex is assigned to z workers and $y > \frac{z}{2}$ workers vote a consensus answer (e.g, `Yes`) and $z - y$ workers vote the other answer (e.g., `No`). The confidence of the voted answer is $c = \frac{y}{z}$.

Error-Tolerant Coloring Strategy. For each crowdsourced vertex, if the confidence of workers on this vertex is high, e.g, ≥ 0.8, we use our coloring strategy to color its ancestors or descendants; otherwise, we color it BLUE and do not color its ancestors or descendants. For the GREEN and RED pairs, we take them as ground truth as their answers have large confidences. Next we utilize them to color BLUE pairs.

We first need to compute the weights of different attributes which reflect the importance in determining the colors of each pair. Let P^g denote the set of GREEN pairs. For every $p_{ij} \in P^g$, if s_{ij}^k is large, then attribute $\mathcal{A}_k$ plays an important role to determine the color of p_{ij}, and we should assign it with a large weight; otherwise it is insignificant to determine the color of p_{ij}. To this end, we assign a weight ω_k for

Algorithm 5: Error-Tolerant

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: All vertices in $\mathcal{V}$ are colored as GREEN or RED
1 **while** *there exist uncolored vertices in* $\mathcal{V}$ **do**
2 Select a set of uncolored vertices to ask workers;
3 **for** *each asked* p_{ij} *with an answer* **do**
4 **if** $confidence \geq 0.8$ **then**
5 color p_{ij} and its ancestors or descendents;
6 **else** color p_{ij} BLUE;
7 Generate histogram h_i and compute $\mathtt{Pr}_i$;
8 **for** *each* $p_{i'j'}$ *colored* BLUE *in* h_i **do**
9 **if** $\mathtt{Pr}_i > 0.5$ **then** color $p_{i'j'}$ GREEN;
10 **else** color $p_{i'j'}$ RED;
11 **return** *colored* $\mathcal{V}$;

each attribute $\mathcal{A}_k$ as below

$$\omega_k = \frac{\sum_{p_{ij} \in P^g} s_{ij}^k}{\sum_{p_{ij} \in P^g} \sum_{1 \leq t \leq m} s_{ij}^t}. \quad (7)$$

Then we compute a weighted similarity of p_{ij},

$$\hat{s}_{ij} = \sum_{k \in [1,m]} \omega_k \cdot s_{ij}^k. \quad (8)$$

Coloring The Pairs in Low-Confidence Groups. We use a histogram based method to color pairs in BLUE vertices [22, 25]. We first generate equi-depth histograms based on the weighted similarities of pairs in GREEN and RED vertices. Each histogram h_i contains a set of pairs within a similarity range. We count the number of GREEN pairs in h_i and compute the probability $\mathtt{Pr}_i$ that pairs in h_i should be colored GREEN, i.e., the number of GREEN pairs to the total number of pairs in h_i. Then we assign the pairs in BLUE vertices into the histograms and color them based on probability $\mathtt{Pr}_i$. For example, if a pair falls in a histogram with high probability of GREEN, the vertex is colored GREEN; otherwise RED. Algorithm 5 shows the pseudo code. It uses the coloring strategy only for the vertices with high-confidence answers (line 5) and utilizes the histograms to color the vertices with low-confidence answers (lines 7-10).

Recall the topological-sorting method in Figure 7(b). The workers return the answer of g_2 with a low confidence, and we color it BLUE and do another topological sorting among the rest groups, i.e., g_8. g_8 is colored BLUE as workers give a low confidence answer. We get Figure 8. Then we need to color pairs in g_2 and g_8 (i.e., p_{12}, p_{24}, p_{25}) based on the colored pairs. First, we calculate the attribute weight ω based on the pairs P^g =$\{p_{12}, p_{67}, p_{45}, p_{23}, p_{46}, p_{56}, p_{47}, p_{57}\}$ in the colored groups. Using Equation 7, we obtain ω =$\{0.32, 0.28, 0.21, 0.19\}$. Then we build 5 histograms with width 0.2 (see Appendix C). We compute $\hat{s}_{ij}$ of each colored pair by Equation 8 and assign it into the corresponding histogram. $\{p_{67}, p_{45}\}$ are assigned into h_5 ([0.8,1]). As all of them are colored GREEN, $\mathtt{Pr}_5 = 1$. $\{p_{23}, p_{13}\}$ are assigned into h_4 ([0.6,0.8)), and $\mathtt{Pr}_4 = 1$. $\{p_{46}, p_{57}, p_{47}, p_{56}, p_{10,11}, p_{26}, p_{27}\}$ are assigned into h_3 ([0.4,0.6)), and $\mathtt{Pr}_3 = \frac{4}{7} = 0.57$. $\{p_{37}, p_{89}, p_{34}, p_{35}\}$ are assigned into h_2 ([0.2,0.4)), and $\mathtt{Pr}_2 = 0$. Next we compute $\hat{s}_{ij}$ of p_{12}, p_{24} and p_{25}. For instance, $\hat{s}_{12} = 0.32 \times 0.72 + 0.28 \times 0.4 + 0.21 \times 1 + 0.19 \times 0.88 = 0.72$, so we assign it into h_4 and color it GREEN due to $\mathtt{Pr}_4 > 0.5$. Similarly, we color p_{24} and p_{25} RED.

7. EXPERIMENT

In this section, we evaluate our methods and report experimental results. The goals of our experiments include (1)

	#Records	#Attr	#Pairs	#Workers/Pair
Restaurant	858	4	5010	5
Cora	997	8	29510	5
ACMPub	66,879	4	204,000	5

Table 3: Three real-world Datasets.

evaluating our proposed techniques and (2) comparing our method with state-of-the-art approaches.

7.1 Experimental Setting

Datasets. We use three real-world datasets which are widely adopted by existing works [7, 22, 23, 24]. (1) Restaurant[2] is a restaurant dataset consisting of 858 restaurants with 752 different entities. The dataset has four attributes, Name, Address, City and Flavor. (2) Cora[3] is a dataset of research papers, which contains 997 records with 191 different entities. The dataset has 8 attributes: Author, Title, Journal, Year, Pages, Publisher, Type and Editor. (3) ACMPub[4] is a larger publication dataset consisting of 66,879 records with 5347 different entities. It has four attributes: Author, Title, Conference and Year. Table 3 shows the details.

Similarity Functions. We use three similarity functions, Jaccard, edit similarity and bigram Jaccard. For bigram, we generate bigrams for every attribute and compute Jaccard on bigram sets as the similarity, where a bigram is a substring with length 2 and a bigram set contains all the bigrams in an attribute. We use bigram by default.

Pruning. As ACMPub has 66,879 records, it will generate $\frac{66879*66878}{2} = 2,236,366,881$ pairs and it is rather expensive to consider all of them. Following previous work [24, 23], we compute a similarity score for each pair of records by Jaccard and prune pairs whose similarity scores are bellow 0.3. After pruning, there are 5010, 29510 and 204000 pairs left in Restaurant, Cora and ACMPub datasets respectively.

AMT Setting. We use Amazon Mechanical Turk (AMT). To ensure fair comparison between different algorithms, each question should be answered by the same workers. To this end, we crowdsource all pairs in each dataset to AMT and get their answers. If different algorithms ask the same pair, they will use the same answer. We assign each question to five workers and use the weighted majority voting to integrate the answers. We pack every ten pairs in a HIT and pay 10 cents for each HIT. We vary workers' accuracy which can be specified on AMT, where the worker accuracy is computed based on workers' approval rate in history at AMT.

Comparison. We compare with state-of-the-art methods, ACD [23], Trans [24] and GCER [25] on the same experimental setting. We get the source codes of ACD and Trans from the authors and implement GCER by ourselves.

Evaluation Metrics. For different methods, we compare the quality, the number of questions, the number of iterations, and the assignment time. For quality, we use F-measure, which is a combination of precision and recall. Suppose the set of pairs that refer to the same entity is S_T, and the set of pairs that an algorithm reports as the same entity is S_P. Then the precision is $p = \frac{|S_T \cap S_P|}{|S_P|}$, the recall is $r = \frac{|S_T \cap S_P|}{|S_T|}$, and the F-measure is $\frac{2pr}{p+r}$.

7.2 Evaluating Worker Accuracy

We compare our methods (Power without error-tolerant techniques and Power+ with error-tolerant techniques) with state-of-the-art approaches ACD [23], Trans [24] and GCER [25]. We compare the number of iterations, the number of questions, and the quality. As GCER requires a parameter to tune the number of asked pairs, we set this parameter the same as ACD, i.e., the maximum number of questions among these algorithms. GCER asks 100 questions in each iteration. For our two algorithms, we use the split-based grouping algorithm to group the pairs and set the grouping threshold ε as 0.1, utilize the index-based method to construct the graph, and employ the topological-sorting algorithm to select questions. We evaluate our graph construction, grouping and question-selection techniques in Appendix E.

7.2.1 Real Exp: Evaluating Worker Accuracy

Existing studies [24, 22, 7] select the workers with approval rate above 95% or passing a qualification test to avoid malicious workers. To evaluate the robustness of the algorithms, we vary the workers' accuracy. In the real crowdsourcing platforms AMT, we can specify the worker accuracy by selecting the approval rate. We select three groups of workers, 70%-80% (70% in the figure), 80%-90% (80% in the figure) and above 90% (90% in the figure) respectively. For each group of workers, we ask them to answer our questions and compare different algorithms. Figures 9-11 show the results. We make the following observations.

Quality. Firstly, Power+ outperforms Power, because Power+ can tolerate workers' errors. With the increase of worker accuracy, the improvement decreases. This is because for higher worker accuracy, there are fewer errors and Power+ has limited room to further improve the quality. Secondly, Power+ achieves the same quality as state-of-the-art studies and even higher. Even for low-quality workers, our methods still achieve high quality, because (1) Power+ can tolerate errors by not coloring unconfident vertices (and thus avoid enlarging the errors by a wrong coloring vertex); and (2) few pairs invalidate the partial order. Specifically, on the Restaurant dataset, as the tasks are very easy, most workers can correctly compare each pair, and thus all the methods achieve high quality. On the Cora dataset, Power+ and ACD achieve much higher quality than Trans and GCER on all three groups of workers, because this dataset is harder and workers may return noisy results. Trans and GCER cannot tolerate workers' errors and moreover they may expand the error propagation due to the transitivity rules. On the ACMPub dataset, Power+ and ACD still outperform other methods because both of them consider crowd's errors. Thirdly, with the increase of worker accuracy, the quality of all the algorithms increase, because workers return higher quality answers. Fourthly, even for workers with different accuracy, the algorithms achieve similar quality. This is because the worker accuracy on AMT is computed based on their accuracy on history tasks but not on our tasks. A worker will give higher quality on easy datasets, e.g., Restaurant, and lower quality on hard datasets, e.g., Cora. To address this issue, we conduct a simulation experiment in Section 7.2.2.

#Questions. Firstly, our two methods Power and Power+ ask fewer questions than state-of-the-art methods, even by 2 orders of magnitude. This is because we can utilize the partial order to prune many pairs that do not need to be asked and use the grouping techniques to reduce the graph size. The partial order can prune the pairs with larger similarities than a Green vertex and the pairs with smaller similarities than a Red vertex, while the grouping technique can prune many pairs with similar similarities close to the

[2] http://www.cs.utexas.edu/users/ml/riddle/data/restaurant.tar.gz
[3] https://www.cics.umass.edu/smccallum/data/cora-refs.tar.gz
[4] http://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution

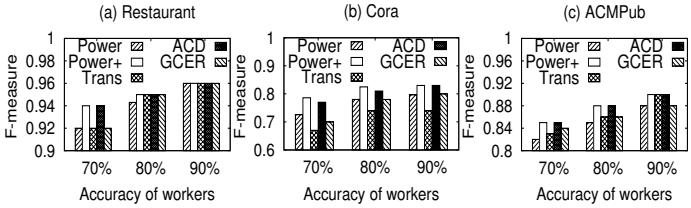


Figure 9: Quality Comparison by Varying Worker Accuracy (Real Experiments).

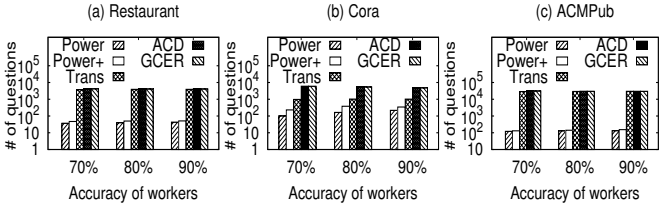


Figure 10: #Question Comparison by Varying Worker Accuracy (Real Experiments).

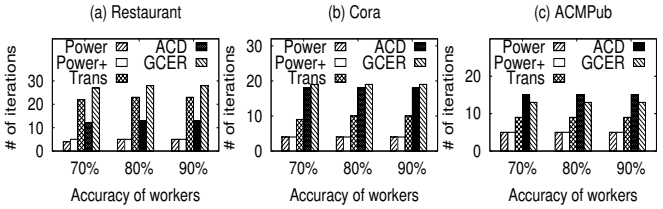


Figure 11: #Iteration Comparison by Varying Worker Accuracy (Real Experiments).

asked pairs. `Trans` can also reduce the number of questions based on transitivity at the expense of lowering down the quality. `ACD` and `GCER` achieve high quality at the expense of asking many more questions. For example, on `ACMPub`, `ACD`, `GCER`, and `Trans` ask 30,000 questions, and our methods ask 150 questions. Thus our methods can save 200× monetary cost than `ACD`, `GCER`, and `Trans`. On `Restaurant`, `ACD` and `GCER` ask 4100 questions, `Trans` asks 3900 questions while `Power` only asks 51 questions. Thus our methods save 80× monetary cost than `ACD`, `GCER`, and `Trans`. On `Cora`, `ACD` and `GCER` ask 4800 questions, `Trans` asks 1020 questions while our methods only ask 354 questions. `Trans` saves a little cost on the `Restaurant` dataset because only few pairs satisfy the transitivity rules. Secondly, `Power+` asks a few more questions than `Power` to tolerate the unconfident vertices and avoid coloring their ancestors and descendants. As there are few unconfident vertices, the gap between `Power+` and `Power` is trivial. Thirdly, the worker accuracy has little effect on the number of questions, because (1) our methods ask few questions and the question number is determined by the graph structure but not worker accuracy and (2) other methods do not consider worker accuracy to select questions.

#Iterations. Firstly, our methods involve fewer iterations than state-of-the-art approaches. This is because (1) our methods ask smaller number of questions and (2) our methods can ask many questions in parallel. For example, on the `Restaurant` dataset, `ACD` involves 13 iterations, `GCER` involves 28 iterations, `Trans` involves 23 iterations, while `Power+` only involves 5 iterations. On the `Cora` dataset, `ACD` involves 18 iterations, `Trans` involves 10 iterations, `GCER` involves 19 iterations, while `Power+` only involves 4 iterations. On the `ACMPub` dataset, `ACD` involves 15 iterations, `Trans` involves 9 iterations, `GCER` involves 13 iterations, while `Power+` only involves 5 iterations. Thus our method saves 2-5× latency cost on the `Cora` dataset. Secondly, `Power` and `Power+` nearly have the same number of iterations, because they have little difference on the number of asked questions. Thirdly,

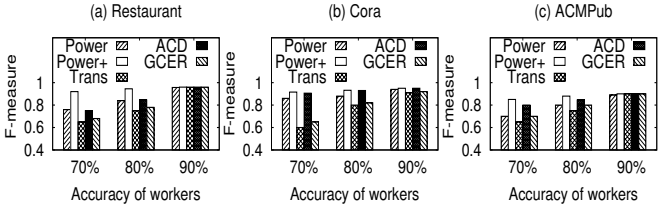


Figure 12: Quality Comparison by Varying Worker Accuracy (Simulation Experiments).

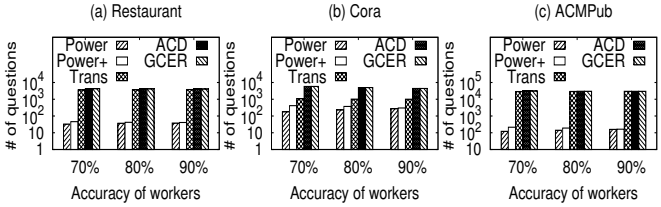


Figure 13: #Question Comparison by Varying Worker Accuracy (Simulation Experiments).

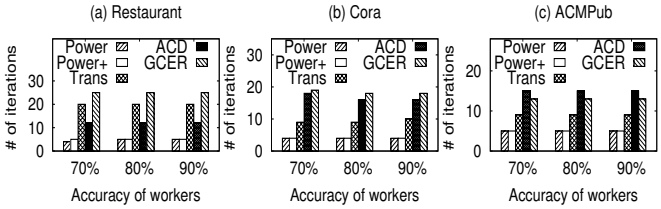


Figure 14: #Iteration Comparison by Varying Worker Accuracy (Simulation Experiments).

the worker accuracy has little impact on the number of iterations, because existing studies do not consider worker accuracy and our methods ask few questions.

7.2.2 *Simulation Exp: Evaluating Worker Accuracy*

In real experiments, we select workers based on their history accuracy on AMT. However the approval rates of workers only reflect their history accuracy when they answered other questions in history but not the accuracy on our questions. Workers may have different quality on different datasets and time. For example, on the `Restaurant` dataset, the problem is easy, and most workers give a correct answer even though they have a low history accuracy. On the `Cora` dataset, because the dataset is relative dirty and professional, many workers return wrong answers even though they have a high history accuracy. Therefore, the reason why many previous studies[7, 22, 23, 24] set a high approval rate is to filter these malicious workers, but this does not mean that most workers can give right answers under the high approval rate guarantee. To address this issue, we conduct a simulation experiment. We assume the ground truth is known and generate workers with quality in 70%-80%, 80%-90%, and above 90% respectively. Figures 12-14 show the results on the simulation experiments.

Quality. Firstly, `Power+` significantly outperforms other methods for low-quality workers, e.g., 70% and 80%. This is because (1) `Power+` can tolerate more errors by postponing coloring the unconfident vertices, and for low-quality workers, there are many more unconfident vertices. For example, for those wrongly answered vertices, we do not color them Green or Red immediately and also do not color their ancestors or descendants, which avoid many errors; (2) `Power+` can tolerate the malicious answers by first asking middle-level vertices in the graph and thus has low possibility to wrongly label some pairs (because a high-level vertex will affect many vertices if it is colored Red and a low-level vertex will affect many vertices if it is colored Green). `ACD` outperforms other baselines, because it tolerates errors based

on the clusters refinement (each cluster contains records referring to the same entity). However, on the Restaurant dataset, ACD has lower quality, because there are few records in each cluster and ACD cannot utilize this limited information to infer the answers. Trans and GCER cannot tolerate errors, and thus they have rather low quality for low worker accuracy. Secondly, for high worker accuracy, e.g., 90%, all the algorithms achieve rather high quality as there are few errors in the workers' answers. Thirdly, with the increase of worker accuracy, all the methods achieve higher quality as they can utilize high quality answers. Power+ outperforms ACD, which in turns is better than other methods. For example, on the Restaurant dataset, in real experiments, all methods have more than 92% F-measure whatever the workers' accuracy is, because workers have high quality on this easy dataset. However, in our simulation experiment, for 70% accuracy, Power+ achieves 92% F-measure while Power, Trans, ACD, and GCER have 76%, 65%, 77% and 75% F-measure respectively. For 80% accuracy, Power+ still outperforms other methods, and all the methods have improvement on quality compared with 70%. For 90% accuracy, all methods can achieve high quality, e.g., 95%, because the workers return high-quality answers. On the Cora dataset, for 70% accuracy, Power+ and ACD have high F-measure and reach 91% due to tolerating crowd's errors while Power has F-measure 86%. Trans and GCER expand the error propagation due to the transitivity rule, whose F-measures are 60% and 65%. For 80% accuracy, Power+ and ACD improve to 93%. And Power, Trans and GCER are 88%, 80% and 82% respectively. For 90% accuracy, all methods improve the quality to above 90%. Power+ and ACD still outperform others. On the ACMPub dataset, similarly to the Restaurant and Cora datasets, Power+ and ACD outperform other methods when worker accuracy is low. If worker accuracy is high, all methods achieve nearly the same quality.

#Questions. Since the worker accuracy has little effect on the number of questions, there is little difference between real experiments and simulation experiments. Power+ saves 80×, 10×, 200× monetary cost than ACD, GCER, and Trans on the Restaurant, Cora, and ACMPub datasets respectively.

#Iterations. Since the worker accuracy has little impact on the number of iterations, there is little difference between real experiments and simulation experiments. Our method still saves 2-5× latency cost on the three datasets.

7.3 Evaluating Similarity Functions

We evaluate the effect of different similarity functions. On each dataset, we respectively use Jaccard, edit similarity, and bigram on every attribute to generate the graph and compare the results for the three similarity functions. Note that on the Restaurant dataset, Jaccard is not a good similarity function for the Name attribute as there are only 1-2 words in the restaurant name; while on the ACMPub dataset, edit similarity is not a good choice for the Title attribute as there are many words in the paper title. We want to test whether our methods and state-of-the-art approaches can tolerate the noisy results generated by different similarity functions. We use the worker accuracy of 90%. Figures 15-17 show the results. We make the following observations.

Firstly, different similarity functions have little impact on the quality among all the methods, because all of these methods use a property that the pairs with large similarities have large possibility to refer to the same entity. On the real datasets, most of the similar functions satisfy this

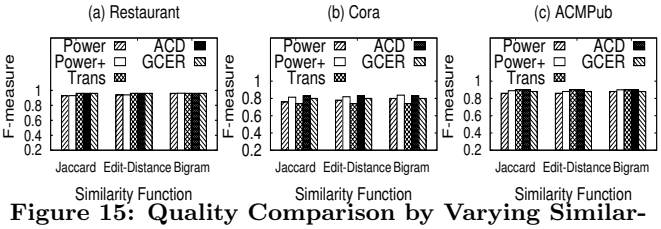


Figure 15: Quality Comparison by Varying Similarity Functions (Real Experiments).

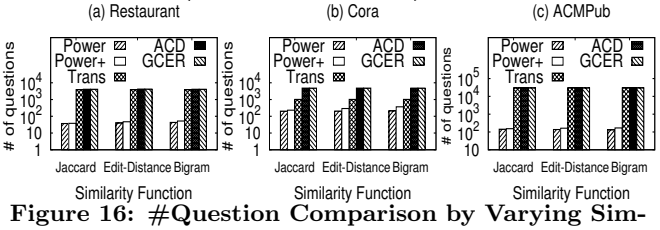


Figure 16: #Question Comparison by Varying Similarity Functions (Real Experiments).

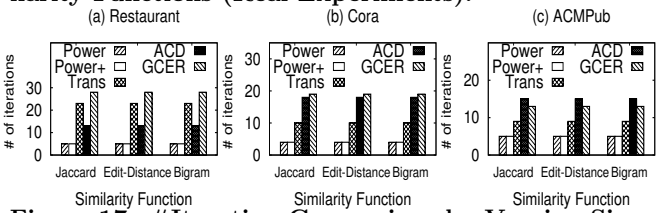


Figure 17: #Iteration Comparison by Varying Similarity Functions (Real Experiments).

property. For our methods, if the functions do not significantly invalidate the partial order, they can be used in our methods. In addition, existing methods use record-level similarity while our methods utilize attribute-level similarity to evaluate different pairs. Thus our methods can use more information to determine the partial order. Even if there exist some noisy functions on some attributes (e.g., Jaccard for Name and edit similarity for Title), our methods can utilize other similarity functions to obtain a good partial order and thus can correct the errors caused by the noisy similarity functions. Therefore our methods are robust on real datasets, even for some noisy functions. Secondly, the similarity functions have little effect on the number of questions, as the number of questions is determined by the graph structure and the graphs generated by different functions have no much difference. Thirdly, the similarity functions have little impact on the number of iterations, as the number of iterations is determined by the number of questions but not by the similarity functions.

8. CONCLUSION

We proposed a partial-order based crowdsourced entity resolution framework. We defined a partial order on record pairs based on their similarities on every attribute. We proposed a graph-based coloring strategy to deduce the answer of some pairs based on the answers of asked pairs. We devised two algorithms to construct the graph and proposed two grouping methods to reduce the graph size. We proposed effective algorithms to judiciously select pairs to ask in order to minimize the number of asked pairs. We developed error-tolerant techniques to tolerate the errors. Experimental results show that our method saves more money than existing approaches while keeping the same quality.

Acknowledgements. This work was partly supported by the 973 Program of China (2015CB358700), NSF of China (61422205, 61472198), Huawei, Shenzhou, Tencent, FDCT/116/2013/A3, MYRG105(Y1-L3)-FST13-GZ, 863 Program(2012AA012600), and Chinese Special Project of Science & Technology(2013zx01039-002-002).

9. REFERENCES

[1] C. C. Cao, J. She, Y. Tong, and L. Chen. Whom to ask? jury selection for decision making tasks on micro-blog services. *PVLDB*, 5(11):1495–1506, 2012.

[2] X. Chen, P. N. Bennett, K. Collins-Thompson, and E. Horvitz. Pairwise ranking aggregation in a crowdsourced setting. In *WSDM*, pages 193–202, 2013.

[3] J. Fan, G. Li, B. C. Ooi, K. Tan, and J. Feng. icrowd: An adaptive crowdsourcing framework. In *SIGMOD*, pages 1015–1030, 2015.

[4] S. Felsner, V. Raghavan, and J. P. Spinrad. Recognition algorithms for orders of small width and graphs of small dilworth number. *Order*, 20(4):351–364, 2003.

[5] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[6] D. R. Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. *American Mathematical Society*, 7(4):701–702, 1956.

[7] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD*, pages 601–612, 2014.

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

[9] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *SIGKDD workshop on human computation*, pages 64–67. ACM, 2010.

[10] M. Kreveld, M. Overmars, O. Schwarzkopf, M. d. Berg, and O. Schwartskopf. Computational geometry: algorithms and applications, 1997.

[11] 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.

[12] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

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

[14] N. Megiddo and K. J. Supowit. On the complexity of some common geometric location problems. *SIAM journal on computing*, 13(1):182–196, 1984.

[15] W. R. Ouyang, L. M. Kaplan, and et al. Debiasing crowdsourced quantitative characteristics in local businesses and services. In *IPSN*, pages 190–201, 2015.

[16] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, pages 1203–1212, 2012.

[17] H. Park and J. Widom. Query optimization over crowdsourced data. *PVLDB*, 6(10):781–792, 2013.

[18] T. Pfeiffer, X. A. Gao, Y. Chen, A. Mao, and D. G. Rand. Adaptive polling for information aggregation. In *AAAI*, 2012.

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

[20] V. Verroios and H. Garcia-Molina. Entity resolution with crowd errors. In *ICDE*, pages 219–230, 2015.

[21] N. Vesdapunt, K. Bellare, and N. N. Dalvi. Crowdsourcing algorithms for entity resolution. *PVLDB*, 2014.

[22] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 2012.

[23] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, pages 229–240, 2013.

[24] S. Wang, X. Xiao, and C. Lee. Crowd-based deduplication: An adaptive approach. In *SIGMOD*, pages 1263–1277, 2015.

[25] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 2013.

[26] Y. Zheng, R. Cheng, S. Maniu, and L. Mo. On optimality of jury selection in crowdsourcing. In *EDBT*, pages 193–204, 2015.

[27] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD*, pages 1031–1046, 2015.

APPENDIX

A. VERTEX GROUPING: GREEDY

Generating Maximal Groups. We first consider the one-dimensional case, i.e., $m = 1$. We generate all the maximal groups based on s_{ij}^1. We first sort p_{ij} based on s_{ij}^1 in a descending order, denoted by $p_1, p_2, \ldots, p_n$. For the first pair p_1, we generate a longest group $\{p_1, p_2, \ldots, p_t\}$ where $p_1 - p_t \leq \varepsilon$ and $p_1 - p_{t+1} > \varepsilon$. Obviously this longest group is a maximal group. Next we generate the longest group for p_2. If the longest group of p_2 is not contained by that of p_1, it is a maximal group. Iteratively we can generate all the maximal groups. The complexity is $\mathcal{O}(|\mathcal{V}|^2)$.

For the m-dimensional case, we first generate the maximal groups $\mathcal{M}^i$ on every attribute $\mathcal{A}_i$. Then we join them to generate the maximal groups, i.e., $\mathcal{M}^1 \bowtie \mathcal{M}^2 \bowtie \cdots \bowtie \mathcal{M}^m = \{\mathcal{M}_{i_1}^1 \cap \mathcal{M}_{i_2}^2 \cap \cdots \cap \mathcal{M}_{i_m}^m\}$ where $1 \leq i_j \leq |\mathcal{M}^j|$. We prove that the generated groups contain all maximal groups. Then we utilize these groups to run the greedy algorithm.

Notation	Description
$\mathcal{T} = \{r_1, r_2, ..., r_n\}$	a set or records
$\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_m\}$	a set of attributes
$r_i[k]$	value of attribute $\mathcal{A}_k$ in record r_i
p_{ij}	(r_i, r_j)
s_{ij}^k	similarity between $r_i[k]$ and $r_j[k]$
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	a DAG of pairs in $\mathcal{T}$
$\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$	a grouped DAG of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
$\succ$	partial order
$\mathcal{C}(p_{ij})$	the child vertex set of p_{ij}
$\mathcal{P}(p_{ij})$	the parent vertex set of p_{ij}
$\mathcal{B}$	the number of incomparable vertices

Table 4: Notations Used In This Paper.

THEOREM 3. $\mathcal{M}^1 \bowtie \mathcal{M}^2 \bowtie \cdots \bowtie \mathcal{M}^m = \{\mathcal{M}_{i_1}^1 \cap \mathcal{M}_{i_2}^2 \cap \cdots \cap \mathcal{M}_{i_m}^m\}$ *contains all maximal groups.*

PROOF. We prove that for any maximal group, g, there exist $\mathcal{M}_{i_1}^1$, $\mathcal{M}_{i_2}^2$, $\cdots$, $\mathcal{M}_{i_m}^m$, $g = \mathcal{M}_{i_1}^1 \cap \mathcal{M}_{i_2}^2 \cap \cdots \cap \mathcal{M}_{i_m}^m$. As g is a maximal group, $g_i^k.u - g_i^k.l \leq \varepsilon$ for any attribute $\mathcal{A}_k$. Let $s_{ij}^k = g_i^k.l$. We generate the maximal group $\mathcal{M}_{i_k}^k$ on attribute $\mathcal{A}_k$ based on s_{ij}^k. Obviously $g \subseteq \mathcal{M}_{i_k}^k$. Thus $g \subseteq \mathcal{M}_{i_1}^1 \cap \mathcal{M}_{i_2}^2 \cap \cdots \cap \mathcal{M}_{i_m}^m$. As g is a maximal group, $g = \mathcal{M}_{i_1}^1 \cap \mathcal{M}_{i_2}^2 \cap \cdots \cap \mathcal{M}_{i_m}^m$. □

Algorithm 6 shows the pseudo code. It first generates all the maximal groups (line 1), and then greedily picks the largest group (line 3). Finally it updates other groups by removing the vertices in the largest group (line 5).

B. MULTI-PATH SELECTION ALGORITHM

Algorithm 7 shows the pseudo code. It first finds the minimal disjoint paths (line 2) and then asks their mid-vertices in parallel (lines 4-5). Next it colors the graph based on the answers and removes the colored vertices (line 6). Finally, it repeats the above step if there exist uncolored vertices in $\mathcal{V}$. For example, we first compute the three disjoint paths and asks their mid-vertices g_5, g_3 and g_6 together in Figure 6. We get the answers: g_5 is RED, and g_3 and g_6 are GREEN. We color the graph based on these three answers(Figure 6(b)). Next we generate a path: $g_2 \rightsquigarrow g_8$ and we ask g_2. The answer is: g_2 is GREEN, and we color the graph (Figure 6(c)). Iteratively we color all vertices (Figure 6(d)). This method asks 5 vertices and involves 3 iterations.

C. EXAMPLE OF ERROR TOLERANT

Recall the example in Figure 8 where g_2 and g_8 are colored BLUE as workers return low-confident answers to them. We calculate the attribute weight ω based on the pairs P^g $=\{p_{13}, p_{67}, p_{45}, p_{23}, p_{46}, p_{56}, p_{47}, p_{57}\}$ in the colored groups. Using Equation 7, we obtain ω $=\{0.32, 0.28, 0.21, 0.19\}$.

Algorithm 6: Vertex Grouping: `Greedy`

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: A set of groups $g_1, g_2, ..., g_x$

```
Generate maximal groups M;
while M is not empty do
    Pick the largest group g from M;
    for each g_i in M do
        g_i = g_i − g;
```

Algorithm 7: Question Selection: `Multi-Path`

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: All vertices in $\mathcal{V}$ are colored as GREEN or RED

```
while there exist uncolored vertices in V do
    Compute B disjoint paths;
    for each path of these disjoint paths do
        N ← mid-vertex of the path;
    Ask N to workers in parallel and color G;
    Removed colored vertices;
return colored V;
```

Then we compute the estimated similarities based on the weight and Table 18 shows the estimated similarities. Next we divide the pairs into different histograms and Figure 19 shows the histograms. p_{12} falls in h_4 and is colored GREEN. p_{24} and p_{25} fall in h_2 and are colored RED.

p_{ij}	$\hat{s}_{ij}$	p_{ij}	$\hat{s}_{ij}$
p_{12}	0.72	p_{37}	0.21
p_{13}	0.68	p_{45}	0.97
p_{23}	0.60	p_{46}	0.43
p_{24}	0.28	p_{47}	0.42
p_{25}	0.29	p_{56}	0.41
p_{26}	0.40	p_{57}	0.44
p_{27}	0.41	p_{67}	0.98
p_{34}	0.39	p_{89}	0.37
p_{35}	0.39	$p_{10,11}$	0.44

Figure 18: Estimated Similarity $\hat{s}_{ij}$.

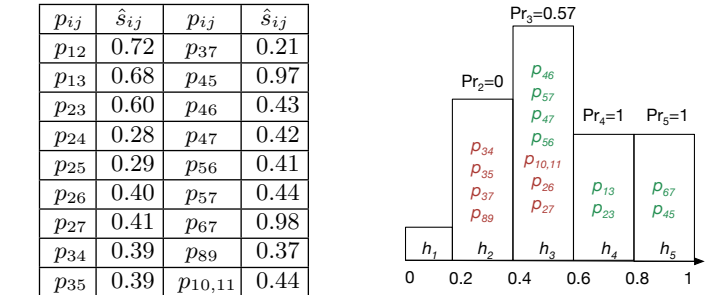


Figure 19: Equi-depth Histograms.

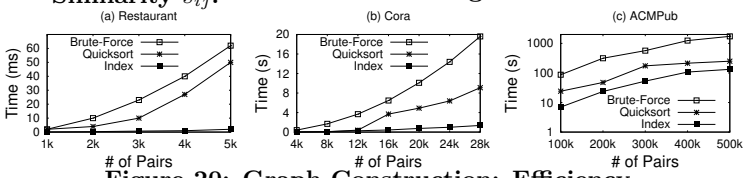


Figure 20: Graph Construction: Efficiency.

D. PROOF OF THEOREMS

D.1 Proof of Theorem 1

We prove the problem is NP-Hard even $m = 2$ by a reduction from the following rectangle cover problem. In a rectangle cover instance, we are given a set of points in the Euclidean plane $\mathbb{R}^2$. Our goal is to use the minimum number of unit squares to cover all points. The problem is known to be NP-Hard [14]. In our problem, it is easy to see a vertex group can be covered by a square of side length ϵ. We can partition the set of vertices into k groups, if and only if all vertices can be covered by k squares of side length ϵ. Therefore, our problem is equivalent to the rectangle cover problem, thus is NP-Hard as well.

D.2 Proof of Theorem 2

(1) Disjoint: If there exist two paths with common vertices, this vertex has at least two edges in the maximal matching, which contradicts with the definition of maximal matching.
(2) Complete: Consider any vertex v. If its in-degree is 0, it must be covered by a path. If its in-degree is not 0, it has an in-edge (v', v). We call v' the parent of v. If the in-degree of v' is 0, v' and v will be covered by the same path starting at v'; otherwise we find the parent of v'. Iteratively we find an ancestor of v whose in-degree is 0, and then v is covered by the path starting at this ancestor.
(3) Minimal: Let J denote the number of edges in a matching and D denote the number of disjoint paths in the graph. Fulkerson et al. [6] proved that $J + D = |\mathcal{V}|$. As $|\mathcal{V}|$ is fixed, if we find the maximal matching, then D is minimal.

E. MORE EXPERIMENTAL RESULTS

E.1 Evaluation on Graph Construction

E.1.1 Evaluation on Graph Construction Algorithms

We compare the efficiency of the three graph construction algorithms (proposed in Section 4.1). (1) `Brute-Force`: the brute-force method that compares every two pairs. (2) `QuickSort`: the quicksort-based method. (3) `Index`: the index-based method.[5] To test the scalability, on `ACMPub`, we set the bound τ as 0.18 and generate 500K pairs. Figure 20 shows the results by varying the number of pairs. We can see that `Index` significantly outperforms the other two methods, even by 1 order of magnitude. For example, on the `Cora` dataset with 28k pairs, `Brute-Force` takes 20 seconds, `QuickSort` improves it to 10 seconds, while `Index` only takes 1 second. On the larger dataset `ACMPub`, `Index` still outperforms other methods and achieves higher performance. This is because `Index` can utilize the range search tree index to efficiently find the children of a pair and can prune many unnecessary pairs (e.g., incomparable pairs). `QuickSort` outperforms `Brute-Force` because it can also remove some unnecessary pairs based on the partial order. However the improvement is not signifiant, as many vertices in the graph are not comparable based on the partial order and thus many pairs cannot be pruned. For example, in `Restaurant`, 70% pairs of records are not comparable. In `Cora`, 84% pairs of records are not comparable. In `ACMPub`, 80% pairs of records are not comparable.

E.1.2 Evaluation on Grouping

We first evaluate our two techniques `Greedy` and `Split` (proposed in Section 4.2). (1) `Greedy`: it greedily groups the vertices. (2) `Split`: it uses the split-based technique. We first compare the number of groups generated by them. Figure 21 shows the number of groups and Figure 22 shows the running time. Note that on the `ACMPub` dataset, `Greedy` **cannot report the results within 10 hours and thus we do not show `Greedy` in the figure**. We have several observations on the number of groups. Firstly, compared with the total number of pairs in `Restaurant` (5,010 pairs), `Cora` (29,510 pairs) and `ACMPub` (204,000 pairs), `Split` and `Greedy` only generate less than 150, 1300 and 700 groups. Thus the grouping technique can significantly reduce the number of vertices, and thus can reduce the time latency and the crowd cost. Secondly, `Split` generates a few more groups than `Greedy`, because `Split` uses heuristics to generate groups while `Greedy` utilizes a greedy strategy to generate high-quality groups. For example, on the `Cora` dataset with

[5]The three datasets have 4-8 attributes. As it is too complicated to construct a high dimensional range tree, we use a heuristics: we choose two important attributes in each dataset to construct 2-dimensional indexes. When we search the children of a pair, the pairs reported by the index are a superset as they may not satisfy other attributes. To addres this issue, we only need to verify them to remove the false positives based on other non-indexed attributes. In our experiment, we choose attributes Name and Address for `Restaurant`, Author and Title for `Cora`, and Author and Title for `ACMPub`.

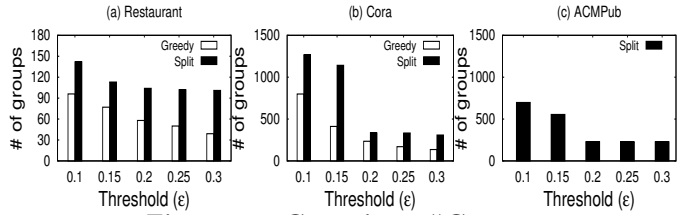


Figure 21: Grouping: #Groups.

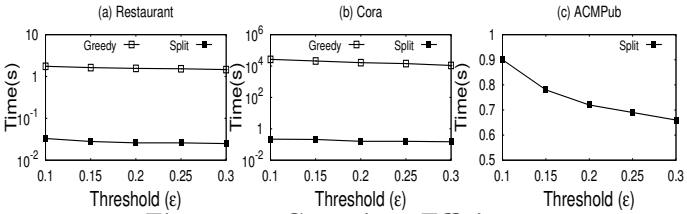


Figure 22: Grouping: Efficiency.

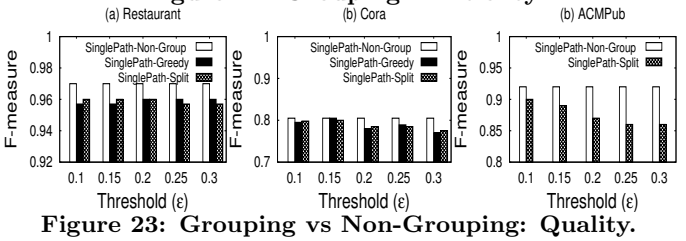


Figure 23: Grouping vs Non-Grouping: Quality.

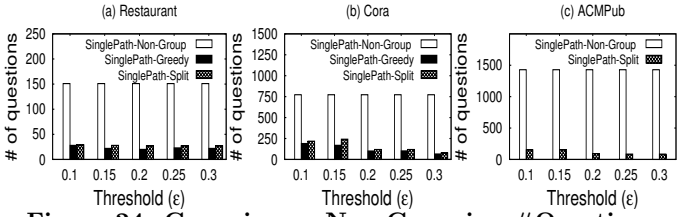


Figure 24: Grouping vs Non-Grouping:#Questions.

grouping threshold $\varepsilon = 0.1$, `Greedy` generates 800 groups and `Split` generates 1200 groups. Thus if we focus on reducing the number of groups, we can select the `Greedy` algorithm. Thirdly, with the increase of the thresholds, the number of groups decreases, because groups with larger thresholds contain more vertices. On the other hand, we can see that `Greedy` takes much longer time than `Split`, even 10000× slower on larger datasets. For example on the `Cora` dataset, `Greedy` takes more than 10000 seconds while `Split` only takes less than 1 second. Thus if we focus on high efficiency, we recommend the `Split` algorithm.

We then compare grouping with non-grouping in terms of quality and the number of questions. We compare three algorithms. (1) `SinglePath-Non-Group`, which utilizes `SinglePath` to ask questions on the original graph without grouping. (2) `SinglePath-Greedy`, which utilizes `SinglePath` to ask questions on the grouped graph generated by the `Greedy` algorithm. (3) `SinglePath-Split`, which utilizes `SinglePath` to ask questions on the grouped graph generated by the `Split` algorithm. Figure 23 shows the quality and Figure 24 shows the number of questions. Note that we do not show `SinglePath-Greedy` on the `ACMPub` dataset as it is too slow.

We have the following observations. (1) The grouping technique slightly reduces the quality by 2% than the non-grouping method. The reasons are twofold. Firstly, many pairs are grouped and we only ask one pair and utilize its answer to deduce the answer of other pairs in the group. If the pairs in the same group have different colors, this method may involve errors. Secondly, there are smaller number of edges in the grouped graph and we ask fewer questions. (2) The grouping technique significantly reduces

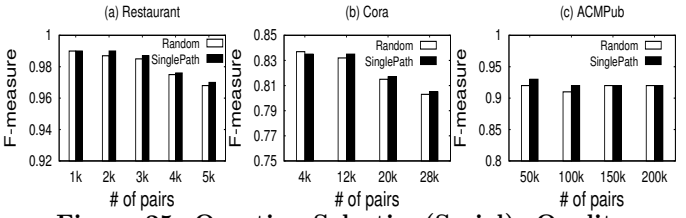


Figure 25: Question Selection(Serial): Quality.

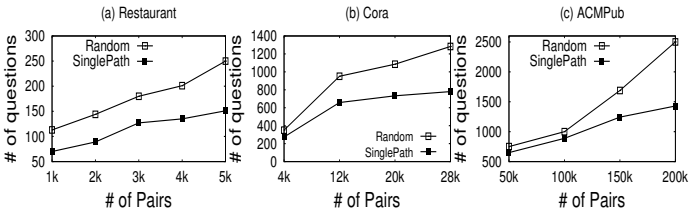


Figure 26: Question Selection(Serial): #Questions.

the number of questions. For example, on the `Cora` dataset with $\varepsilon = 0.1$, the non-grouping method asks 800 questions while the grouping method only asks 80 questions. On the `ACMPub` dataset, the non-grouping method asks 1400 questions while the grouping method asks 163 questions. This is because the grouping technique can significantly reduce the graph size. Thus we can utilize grouping techniques to reduce the cost. (3) The two grouping techniques have no large difference on the number of question as their graph sizes are very close. This verifies that we can use `Split` to generate the groups. (4) The number of questions is not only determined by the number of groups, but also the number of edges. First, fewer groups will lead to fewer questions. Second, fewer edges may lead to more questions, because the answers of many groups cannot be deduced based on the answers of other groups. With the increase of the grouping threshold, the number of groups decreases, and thus the number of questions should decrease intuitively. However, with the increase of the grouping threshold, the groups become larger and it is more difficult to add an edge between two groups. Thus there may be fewer edges in the graph and the number of questions may decrease.

E.2 Evaluation on Question Selection

E.2.1 Evaluation on Serial Algorithms

We first evaluate the serial question-selection algorithms and compare two algorithms (proposed in Section 5.2). (1) `Random`: which randomly selects a vertex in each iteration. (2) `SinglePath`: which selects a vertex from the longest path in each iteration. We compare the two algorithms on the non-grouping graphs. Figure 25 shows the quality and Figure 26 shows the number of questions. We can see that `SinglePath` outperforms `Random` and reduces the number of questions. For example, on the `Restaurant` dataset with 5000 pairs, `Random` asks 250 pairs while `SinglePath` only asks 150 pairs. On the `ACMPub` dataset, `Random` asks 2500 pairs while `SinglePath` only asks 1400 pairs. This is because `SinglePath` can effectively identify the boundary pairs using a binary search strategy. On the other hand, `SinglePath` achieves similar quality with `Random` as the question order does not significantly affect the quality. Thus we can utilize the `SinglePath` to select questions.

E.2.2 Evaluation on Parallel Algorithms

We then evaluate the parallel question-selection algorithms (proposed in Section 5.3). We compare three algorithms: (1) `SinglePath`: which selects a vertex from the longest path in each iteration. (2) `Multi-Path`: which selects multiple

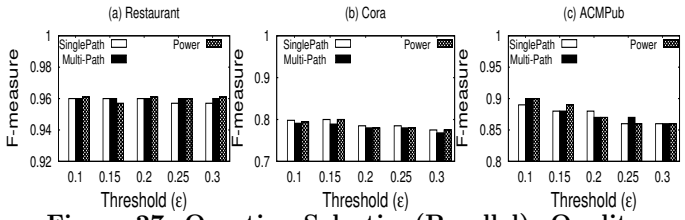


Figure 27: Question Selection(Parallel): Quality.

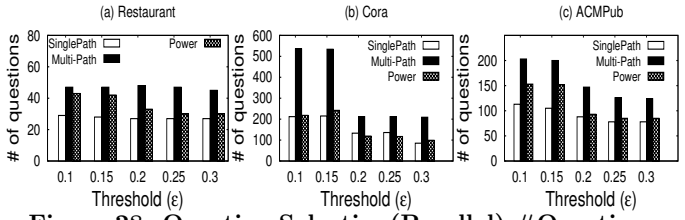


Figure 28: Question Selection(Parallel):#Questions.

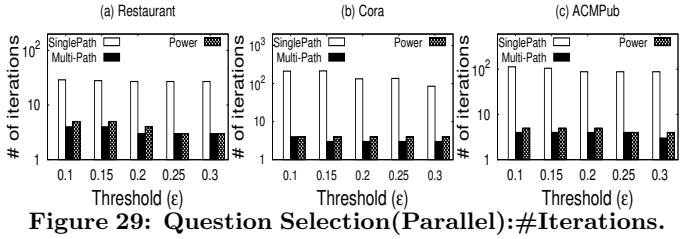


Figure 29: Question Selection(Parallel):#Iterations.

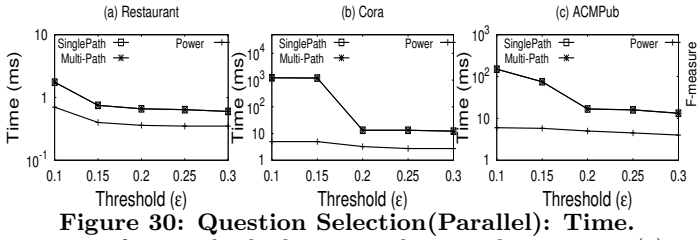


Figure 30: Question Selection(Parallel): Time.

vertices from multiple disjoint paths in each iteration. (3) `Power`: which selects multiple independent vertices based on topological sorting in each iteration. We compare the quality, the number of questions, the number of iterations, and the assignment time in each iteration to select the questions for workers. Figures 27-30 show the results.

(1) For quality, we can see that the three methods achieve similar quality, because different question orders will not affect the quality based on the partial order. (2) For the number of questions, we can see that the two parallel algorithms `Multi-Path` and `Power` ask a few more questions than `SinglePath`. The reason is evident that `Multi-Path` may ask pairs with ancestor-descendent relationships and `Power` may ask pairs with the same descendants which can be avoided by serial algorithms based on the partial order. `Power` outperforms `Multi-Path` because `Power` asks independent questions in each iteration while `Multi-Path` may ask dependent questions. (3) For the number of iterations, the two parallel algorithms `Multi-Path` and `Power` significantly outperform `SinglePath` as they ask questions in parallel. For example, on the `Cora` dataset, `Power` and `Multi-Path` only have 4 iterations while `SinglePath` involves 200 iterations. On the `ACMPub` dataset, `Power` and `Multi-Path` have 5 iterations while `SinglePath` involves 113 iterations. Thus `Power` and `Multi-Path` can significantly reduce the latency. In practice, we need to use the parallel algorithms. (4) For assignment time, all the three algorithms can assign tasks within 1 second. `Multi-Path` and `SinglePath` take longer time than `Power` as they are expensive to find multiple independent paths using the graph matching algorithm, which is consistent with the complexity analysis, while `Power` only needs to compute the topological sorting which is efficient.

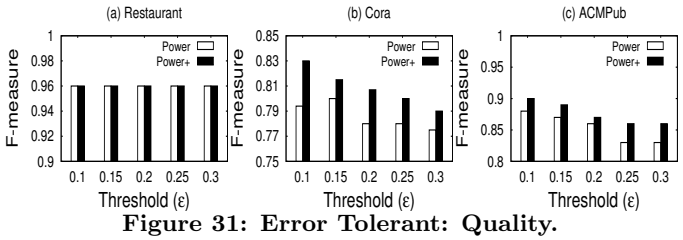


Figure 31: Error Tolerant: Quality.

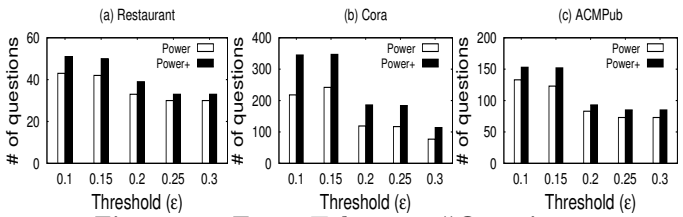


Figure 32: Error Tolerant: #Questions.

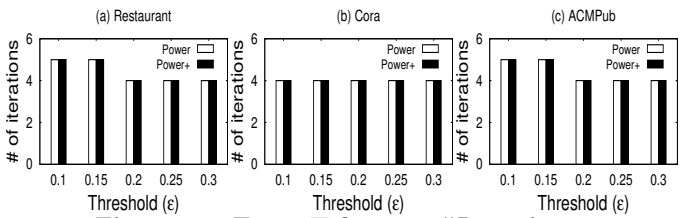


Figure 33: Error Tolerant: #Iterations.

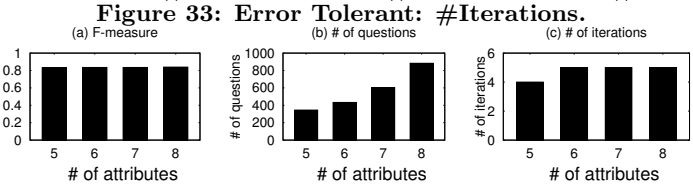


Figure 34: Evaluation by Varying #Attributes(`Cora`).

E.3 Evaluation on Error-Tolerant Techniques

We evaluate the error-tolerant techniques (proposed in Section 6) and compare two algorithms. (1) `Power`: which does not consider errors. (2) `Power+`: which extends `Power` to tolerate errors. We compare quality, the number of questions, and the number of iterations. As they have the same assignment time, we do not compare the assignment time. We build 20 histograms. Figures 31-33 show the results.

`Power+` achieves better quality than `Power`, especially on the `Cora` dataset, because it can tolerate the errors introduced by workers and the partial order. For example, on the `Cora` dataset with $\varepsilon = 0.1$, `Power` only has 79% F-measure while `Power+` improves the quality to 83%. On the `ACMPub` dataset, `Power` has 87% F-measure while `Power+` improves to 90%. On the `Restaurant` dataset, the improvement is not signifiant because the dataset is easy and `Power` already achieves 96% F-measure. On the other hand, `Power+` asks a little more questions than `Power` as `Power+` does not utilize the partial order for some pairs and thus reduces the number of deduced pairs. The two methods have the same number of iterations, because the only difference is that `Power+` does not deduce the answers for some unconfident pairs. Thus we can use the error-tolerant technique to improve the quality.

E.4 Evaluation on The Number of Attributes

We vary the number of attributes on the `Cora` dataset and Figure 34 shows the results. We can see that with the increase of attribute numbers, the number of questions increases, because it is harder to add edges between pairs for more attributes and thus the number of edges decreases. Similar to the number of questions, the number of iterations slightly increases. The quality is not affected as it is determined by the partial order and the crowd error.