Interactive Set Discovery

by

Md Arif Hasnat

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Md Arif Hasnat, 2021

Abstract

We study the problem of set discovery where given a few example tuples of a desired set, we want to find the set in a collection of sets. A challenge is that the example tuples may not uniquely identify a set, and a large number of candidate sets may be returned. Our focus is on interactive exploration to set discovery where additional example tuples from the candidate sets are shown and the user either accepts or rejects them as members of the target set. The goal is to find the target set with the least number of user interactions. The problem can be cast as an optimization problem where we want to find a decision tree that can guide the search to the target set with the least number of questions to be answered by the user. We propose a general algorithm that is capable of reaching an optimal solution and two variations of it that strike a balance between the quality of a solution and the running time. We also propose a novel pruning strategy that safely reduces the search space without introducing false negatives. We evaluate the efficiency and the effectiveness of our algorithms through an extensive experimental study using both real and synthetic datasets and comparing them to previous approaches in the literature. We show that our pruning strategy reduces the running time of the search algorithms by 2-5 orders of magnitude.

Acknowledgements

I would like to thank my supervisor, Professor Davood Rafiei, for his utmost support and guidance. He continuously gave me invaluable insights and was always willing and enthusiastic to assist in any way he could throughout my research. This work would not have been possible without his encouragement. It was an honor to have such a great mentor, in both a personal and professional manner.

I would also like to thank my family, especially my mother and wife, for their continuous love and support.

Contents

1 Introduction **1**
- 1.1 Motivating Examples . . . 2
- 1.2 Problem Statement . . . 2
- 1.3 Overview of Our Approach . . . 3
- 1.4 Thesis Contributions . . . 3
- 1.5 Thesis Outline . . . 4

2 Background **6**
- 2.1 Tree Terminology . . . 6
- 2.2 Decision Tree . . . 7

3 Related Work **15**
- 3.1 Example-Based Queries . . . 15
- 3.2 Interactive Learning of Queries . . . 17
- 3.3 Cost-Efficient Decision Tree Construction . . . 18
- 3.4 Set Expansion . . . 19

4 Problem Formulation **23**

5 Methodology **27**
- 5.1 Cost Lower Bounds . . . 27
- 5.2 Entity Selection . . . 29
- 5.3 Pruning . . . 32
- 5.4 Lookahead Strategies . . . 34
- 5.5 Set Discovery . . . 37
- 5.6 Optimal Strategy . . . 39

6 Experiments **44**
- 6.1 Evaluation Setup . . . 44
- 6.2 Datasets and Queries . . . 45
- 6.3 Evaluation Results on Set Discovery . . . 51
- 6.4 Evaluation Results on Query Discovery . . . 61

7 Conclusion and Future Work **64**
- 7.1 Conclusion . . . 64
- 7.2 Future Work . . . 65

References **66**

List of Tables

6.1 Query sets for the web tables dataset 46
6.2 Synthetic data by varying (a) overlap ratio α, (b) number of sets n, and (c) set size range d 47
6.3 Query sets for the TPC-H benchmark 48
6.4 Target queries for the baseball database 50
6.5 Information about selected example tuples and generated candidate queries on baseball database 51
6.6 Overlap ratios/scores in our datasets 58

List of Figures

2.1 A binary tree 8
2.2 Example of a decision tree 8
2.3 Procedure for calculating entropy-k and gain-k [12] 13

4.1 A collection of example sets 24
4.2 Example of decision tree representations of the sets in Figure 4.1 25

6.1 Tree construction time (seconds) for k-LP varying k on web tables dataset 52
6.2 Comparison of our strategies with InfoGain strategy on web tables dataset 53
6.3 Speedup of our strategies because of pruning 54
6.4 Comparison of our lookahead strategies with our optimal tree search strategy on web tables dataset (cost metric AD) 55
6.5 Comparison of our lookahead strategies with our optimal tree search strategy on web tables dataset (cost metric H) 56
6.6 Effects of set overlaps on average number of questions and tree construction time 57
6.7 Effects of increasing the number of distinct entities in a collection on average number of questions and tree construction time 59
6.8 Effects of increasing the number of sets on average number of questions and tree construction time 60
6.9 Tree construction time (seconds) for our query discovery experiment on TPC-H benchmark 62
6.10 Number of questions and query discovery time to find the target queries on baseball database 63

Chapter 1

Introduction

Consider a large collection of sets and a user who is searching for a particular set in the collection. The user may provide a few example tuples, as a small subset of the desired set, but learns that many sets in the collection contain the subset and are candidates. An interesting question is how the user can find the target set without examining all candidates.

In this thesis, we study an interactive exploration approach to set discovery where example tuples from the candidate sets are shown, and the user either accepts or rejects those tuples as members of the target set. Each interaction with the user has a cost (e.g., in terms of the time spent or the number of questions answered), and ideally, we want to retrieve the target set with the least number of interactions. Relevant research questions are: (1) what exploration strategies may be used, and how efficient are those strategies? (2) how long does an exploration take and what factors (e.g., set sizes, overlaps, etc.) do affect the exploration time? (3) how may the sets be organized to support an efficient exploration?

Interactive set discovery has many applications in enterprise settings where the users may not be familiar or willing to pose formal queries such as SQL.

Thesis statement The research hypothesis put forward is that several lower bounds on the number of interactions for a set discovery can be established and that efficient set discovery algorithms can be constructed based on those lower bounds.

1.1 Motivating Examples

Example 1.1.1. Consider searching for movie lists containing movies similar to what we have already enjoyed. The movie database allows users to create their favorite movie lists and to search movie lists created by other users using movie titles. Suppose we are searching for a movie list that contains movies similar to both *Inception* and *The Dark Knight*. We learn that those two movies are present in 10 movie lists. What is the best way of finding the desired movie list? How can this be done if there are not ten but hundreds or thousands of movie lists that match our initial search?

Example 1.1.2. Consider an enterprise database user who wants to formulate a query but is less familiar with the SQL syntax. He may provide, instead of a SQL query, a few example tuples that are expected to be in the output. The enterprise may keep past queries and each new query is likely to be one of the queries asked in the past. Under this setting, the user may be searching for a past query that has the provided tuples as a subset. Even if past queries are not recorded or do not include the desired query, a query generation tool (e.g., [18, 38]) may be employed to generate queries that include the input tuples as a subset. In both cases, we have a collection of queries that have the user-provided input as subsets, and the problem of finding the desired query becomes a set discovery problem.

1.2 Problem Statement

Given a collection C of unique sets and an initial set I, which includes a subset of the user's desired set, the goal is to find a target set G in C such that $I \subseteq G$ and G is the user's desired set. With no user interaction, the problem is under-specified and more than one such set G can contain the elements of I unless $G = I$, in which case a search is meaningless since the user has listed the full target set. We want to narrow down G to a single set through interactions with the user. Clearly, the user wants to answer as few questions as possible, following the least effort principle. Also, it should be noted that I can be an

empty set, in which case G is fully identified through interactions with the user. Although there is related work on interactive exploration and example-based queries (as reviewed next), to the best of our knowledge, the problem of set discovery is not studied in the literature.

1.3 Overview of Our Approach

We cast the problem as an optimization problem with the aim of minimizing the number of questions that the user needs to answer as well as the wait time. In each step of the exploration, the user is given questions that can reduce the number of candidate sets. Assuming that, all sets G in C that contain the initial set I are equally likely, we consider two exploration scenarios: (1) *average-case* where the average number of questions over all possible target sets is minimized, and (2) *worst-case* where the maximum number of questions over all possible target sets is minimized. Our algorithms are general and work under both exploration scenarios.

1.4 Thesis Contributions

The contributions of this thesis can be summarized as follows:

- We formalize interactive set discovery as an optimization problem, minimizing the number of questions posed to users.

- We propose cost functions to characterize the quality of a decision tree for interactive set discovery in terms of its worst-case and average-case performance and some lower bounds that are easy to compute but effective in pruning the search space.

- We propose a pruning strategy, based on our lower bounds, that allow certain choices of entities for decision tree nodes to be safely rejected if there is evidence that it cannot lead to a better tree than the one already found.

- Based on our pruning strategy, we develop an efficient lookahead algorithm that can find near-optimal trees in many cases. We also develop two variations of our lookahead algorithm to further speed up the search process by bounding the number of entities in each step of the search.

- Through an extensive experimental evaluation, we show that our pruning strategy is effective, reducing the running time by a few orders of magnitude and that our algorithms outperform competitive approaches from the literature.

1.5 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2 provides some background material for this thesis. We discuss our terminology for describing a tree and provide an overview of decision trees.

Chapter 3 reviews the literature closely related to ours, which includes the lines of work on example-based queries, interactive learning of queries, cost-efficient decision tree construction, and set expansion.

In Chapter 4, we define a few cost metrics and formulate set discovery as an optimization problem. The task is to construct a decision tree that can guide the search to the target set with minimum cost.

Our algorithms and strategies are discussed in Chapter 5. First, we define a few lower bounds on the cost of a decision tree. Then, based on the lower bounds, we develop a pruning strategy to reduce the search space during entity selection for the decision tree nodes. After that, we propose a general lookahead strategy and its two variants using the pruning strategy to make the entity selection efficient and effective. Next, we present a lookahead-based near-optimal algorithm and an efficient optimal algorithm to discover the user's target set.

The proposed strategies are evaluated in Chapter 6. It presents an extensive experimental evaluation of their efficiency and effectiveness on the tasks of set discovery and query discovery using both real and synthetic datasets while comparing them to previous approaches in the literature.

Finally, we summarize the thesis and provide remarks for future work in Chapter 7.

Chapter 2

Background

In this chapter, we discuss our terminology for describing a tree and provide an overview of decision trees.

2.1 Tree Terminology

We briefly discuss the terminology used in this thesis for describing trees [3, 30].

1. A *graph* $G = (V, E)$ consists of a finite, nonempty set of *nodes* (or *vertices)* V and a set of *edges* E. If the edges are ordered pairs (v, w) of vertices, then the graph is said to be *directed*.

2. A *path* from a node v_1 to a node v_n in a graph is a sequence of edges of the form (v_1, v_2), $(v_2, v_3), ..., (v_{n-1}, v_n)$ and is of the length n.

3. A directed graph with no cycles is called a *directed acyclic graph*. A *directed* (or *rooted*) *tree* is a directed acyclic graph satisfying the following properties:

   (a) There is exactly one node, called the root, into which no edges enter.

   (b) Every node except the root has exactly one entering edge.

   (c) There is a unique path from the root to each node.

4. If (u, v) is an edge in a tree, then u is called the *parent* of v, and v is a *child* of u. Child nodes with the same parent are *siblings*. If there is a path from u to a node w such that $u \neq w$, then u is an *ancestor* of w and w is a *descendant* of u.

5. A node with no child is called a *leaf* (or a *terminal*). All other nodes are called *internal* nodes.

6. The depth of a node v in a tree is the length of the path from the root to v. The *height* of v is the length of a largest path from v to a leaf. The height of a tree is the height of its root. The root has depth zero, leaf nodes have height zero. The *level* of v is the number of edges along the unique path between it and the root.

7. A *subtree* of a tree T is a tree consisting of a node and all of its descendants in T.

8. An *ordered* tree is a tree in which the children of each node are ordered (normally from left to right).

9. A *binary tree* is an ordered tree in which each node has at most two children.

10. A *full binary tree* is a binary tree in which each node has either 0 or 2 children.

11. A *balanced binary tree* is a binary tree in which the left and the right subtrees of every node differ in height by no more than 1.

Figure 2.1 describes various elements of a balanced full binary tree of height 2.

2.2 Decision Tree

A decision tree is a tree in which each internal node represents a “test” on an attribute (e.g., whether humidity is normal or high), each edge represents the

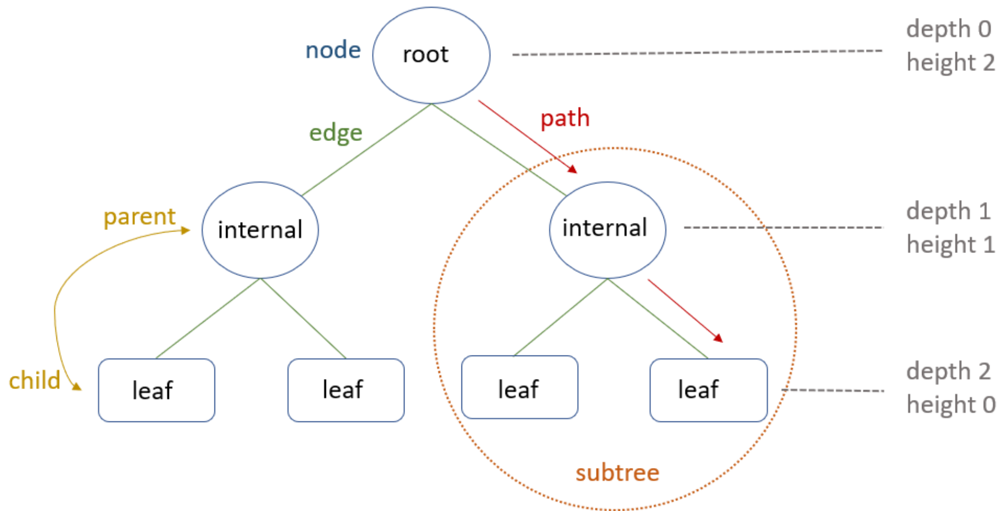
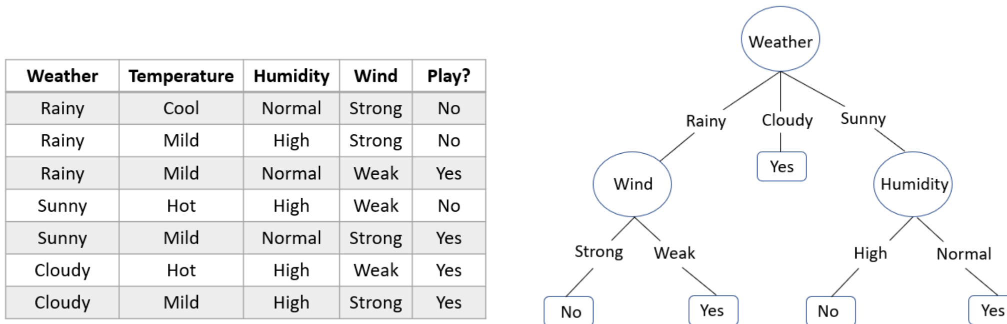


Figure 2.1: A binary tree

outcome of the test, and each leaf represents a decision taken after computing all tests. Figure 2.2 shows an example of a decision tree.



Weather	Temperature	Humidity	Wind	Play?
Rainy	Cool	Normal	Strong	No
Rainy	Mild	High	Strong	No
Rainy	Mild	Normal	Weak	Yes
Sunny	Hot	High	Weak	No
Sunny	Mild	Normal	Strong	Yes
Cloudy	Hot	High	Weak	Yes
Cloudy	Mild	High	Strong	Yes

Figure 2.2: Example of a decision tree

In statistics, data mining, and machine learning, decision trees are used to predict the target value of an item based on observations about the item. The data set is usually represented as records of the form:

$$(x, Y) = (x_1, x_2, x_3, ..., x_k, Y),$$

where the dependent variable, Y, is the target variable that is being understood, classified, predicted, or generalized and the vector x is composed of the attributes/features $x_1, x_2, x_3, ..., x_k$. For example, the records in Figure 2.2 are in the form: $(Weather, Temperature, Humidity, Wind, Play?)$,

where *Weather*, *Temperature*, *Humidity*, and *Wind* are the attributes, and *Play?* is the target variable. The target variable has two possible values- "Yes" and "No". These values are often called *classes*. Hence, each example in a data set belong to one of the classes.

Given a set of examples, the decision tree algorithms first generate a decision tree, then try to predict/classify the target variable of an unknown sample (the value of whose target variable is unknown) by testing the attribute values of the sample against the decision tree. Therefore, if the attribute values of a test sample are $Weather$ = "*Rainy*", $Temperature$ = "*Mild*", $Humidity$ = "$High$", and $Wind$ = "$Weak$", then the value of the target variable *Play?* or the class of the sample is "Yes" according to the decision tree in Figure 2.2.

Decision tree construction Earlier decision tree algorithms, such as ID3 [27], C4.5 [29], and CART [9], construct decision trees using a greedy top-down recursive procedure. The general framework followed by these algorithms for constructing a decision tree is as follows.

1. The first node is the root which considers the complete data set.

2. The best attribute is selected to split the examples at this node using a heuristic or statistical measure.

3. A child node is created for each split value of the selected attribute.

4. For each child, only the examples with the split value of the selected attribute is considered.

5. If all the examples have the same class label, or there are no remaining attributes for further partitioning, then a leaf node is created and is assigned a class label using majority voting.

6. Otherwise, Steps 2-5 are repeated for each child node until it reaches a leaf.

Various statistical measures are used for selecting the best attribute to split a data set. They generally measure the homogeneity of the target vari-

able within the subsets. They are applied to each candidate subset, and the resulting values are combined (e.g., averaged) to provide a measure of the quality of the split. These measures have been empirically evaluated for decision tree induction by Mingers [24].

Information gain This attribute selection measure utilized in ID3 [27] is based on the concept of information content and *entropy* [32], which is the degree of uncertainty, impurity, or disorder. It provides a measure to determine which attribute gives the maximum information about a class. It aims to reduce the level of entropy starting from the root node to the leave nodes. Entropy is defined as the amount of information needed to decide if an arbitrary example belongs to a class. If all the examples were to belong to the same class, the entropy would be 0. For a data set E of examples with n classes where p_i is the fraction of examples of the i-th class in the data set,

$$entropy(E) = -\sum_{i=1}^{n} p_i \log_2 p_i. \tag{2.1}$$

If an attribute a splits the data set E into m subsets $E_1, E_2, ..., E_m$, then the expected information needed to classify the examples in all the subsets E_i is

$$entropy(E, a) = \sum_{i=1}^{m} \frac{|E_i|}{|E|} entropy(E_i) \tag{2.2}$$

and the information gain by selecting attribute a is

$$InfoGain(E, a) = entropy(E) - entropy(E, a). \tag{2.3}$$

For example, the information gain for attribute *Weather* of the data set E

in Figure 2.2, where *Play?* is the target variable, can be calculated as follows:

$$\begin{aligned}
entropy(E) &= -\frac{4}{7}\log_2\frac{4}{7} - \frac{3}{7}\log_2\frac{3}{7} = 0.99,\\
entropy(E_{Weather=\text{``Rainy''}}) &= -\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3} = 0.92,\\
entropy(E_{Weather=\text{``Sunny''}}) &= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1,\\
entropy(E_{Weather=\text{``Cloudy''}}) &= -\frac{2}{2}\log_2\frac{2}{2} - 0 = 0,\\
entropy(E, Weather) &= \frac{3}{7} * 0.92 + \frac{2}{7} * 1 + \frac{2}{7} * 0 = 0.68,\\
InfoGain(E, Weather) &= 0.99 - 0.68 = 0.31.
\end{aligned}$$

The attribute that gives the maximum information gain is selected for the split.

Gain ratio Although Quinlan adopted information gain measure for ID3, he noticed that the measure is biased towards attributes with a large number of distinct values, and hence proposed a normalization, known as *gain ratio* [29]. If an attribute a splits the data set E into m subsets $E_1, E_2, ..., E_m$, then the gain ratio is defined for a as

$$GainRatio(E, a) = \frac{InfoGain(E, a)}{SplitInfo(E, a)}, \tag{2.4}$$

where

$$SplitInfo(E, a) = -\sum_{i=1}^{m} \frac{|E_i|}{|E|} log_2 \frac{|E_i|}{|E|}. \tag{2.5}$$

For example, the gain ratio for attribute *Weather* of the data set E in Figure 2.2 can be calculated as follows:

$$\begin{aligned}
InfoGain(E, Weather) &= 0.31.\\
SplitInfo(E, Weather) &= -\frac{3}{7}\log_2\frac{3}{7} - \frac{2}{7}\log_2\frac{2}{7} - \frac{2}{7}\log_2\frac{2}{7} = 1.56,\\
GainRatio(E, Weather) &= \frac{0.31}{1.56} = 0.20.
\end{aligned}$$

The attribute that gives the maximum gain ratio is selected for splitting.

Gini index Another attribute selection measure used in CART [9] is *gini index*. It measures the degree or the probability of a particular variable being

wrongly classified when it is randomly chosen. The gini index reaches its minimum (zero) when all elements belong to a single class. For a data set E of examples with n classes where p_i is the probability of the i-th class,

$$Gini(E) = 1 - \sum_{i=1}^{n} (p_i)^2. \tag{2.6}$$

If an attribute a splits the data set E into m subsets $E_1, E_2, ..., E_m$, then the gini index for the split is

$$Gini(E, a) = \sum_{i=1}^{m} \frac{|E_i|}{|E|} Gini(E_i). \tag{2.7}$$

For example, the gini index for attribute *Humidity* of the data set E in Figure 2.2, where *Play?* is the target variable, can be calculated as follows:

$$Gini(E_{Humidity=\text{``}Normal\text{''}}) = 1 - ((\frac{2}{3})^2 + (\frac{1}{3})^2) = 0.44,$$
$$Gini(E_{Humidity=\text{``}High\text{''}}) = 1 - ((\frac{2}{4})^2 + (\frac{2}{4})^2) = 0.50,$$
$$Gini(E, Humidity) = \frac{3}{7} * 0.44 + \frac{4}{7} * 0.50 = 0.47.$$

The attribute with the least gini index is chosen for splitting the examples in a node.

Gain-k The majority of the decision tree algorithms such as ID3 [27], C4.5 [29], and CART [9] use greedy heuristics to make locally optimal decisions at each node. The greedy algorithms require a fixed amount of time and are not able to generate a better tree if additional time is available. To allow trade-off between tree quality and learning time, Esmeir et al. [12] propose *entropy-k* and *gain-k* to calculate the information gain by looking at depth k below current node. Entropy-k and gain-k are defined as follows.

Let E be a set of examples at a tree node t. Let $P_E(c_i)$ be the probability of an example in E to belong to class c_i. The entropy at t can be calculated using Shannon's information measure:

$$entropy(E) = I(P_E(c_1), ..., P_E(c_n)) = -\sum_{i=1}^{n} P_E(c_i) \log_2 P_E(c_i) \tag{2.8}$$

If an attribute a with values $\{v_1, ..., v_m\}$ partitions E into $\{E_1, ..., E_m\}$, then the new entropy is the weighted average of the entropies for each subset:

$$entropy\text{-}1(E, a) = \sum_{i=1}^{m} \frac{|E_i|}{|E|} entropy(E_i) \tag{2.9}$$

Thus, the expected information gain for a is given by

$$gain\text{-}1(E, a) = entropy(E) - entropy\text{-}1(E, a) \tag{2.10}$$

The suffix "1" of entropy-1 indicates that the effect of using the attribute was tested one level below the current node. This definition is extended in Figure 2.3 for calculating entropy-k and its associated gain-k. The recursive definition minimizes the $k - 1$ entropy for each child and computes their weighted average. Note that the information gain computed by ID3 is equivalent to gain-k for $k = 1$.

Procedure ENTROPY-K(E, A, a, k)
  If $k = 0$
    Return $\mathrm{I}(\mathrm{P}_E(c_1), \ldots, \mathrm{P}_E(c_n))$
  $V \leftarrow \mathrm{domain}(a)$
  Foreach $v_i \in V$
    $E_i \leftarrow \{e \in E \mid a(e) = v_i\}$
    Foreach $a' \in A$
      $A' \leftarrow A - \{a'\}$
      $h_i(a') \leftarrow$ ENTROPY-K$(E_i, A', a', k - 1))$
  Return $\sum_{i=1}^{|V|} \frac{|E_i|}{|E|} \min_{a' \in A} (h_i(a'))$

Procedure GAIN-K(E, A, a, k)
  Return $\mathrm{I}(\mathrm{P}_E(c_1), \ldots, \mathrm{P}_E(c_n)) -$
      ENTROPY-K(E, A, a, k)

Figure 2.3: Procedure for calculating entropy-k and gain-k [12]

Tree pruning Once a decision tree has been built, some type of pruning is then usually carried out. Pruning is a data compression technique that reduces the size of decision trees by removing subtrees of the tree that are non-critical and redundant to classify instances. There are three main reasons for pruning [22]. One is that it helps to reduce the complexity of a decision tree,

which would otherwise make it very difficult to understand [28], resulting in a faster, possibly less costly classification. Another reason is to help prevent the problem of over-fitting the training data. The third reason is that noisy, sparse, or incomplete data sets can cause very complex decision trees, so pruning is a good way to simplify them [28]. There are several ways to calculate whether a subtree should be pruned or not. A comprehensive review of pruning methods has been carried out by Frank et al. [13].

Rules extraction Decision trees can sometimes be hard to read and interpret especially when trees get large. To improve the readability, a decision tree may be reformulated as a set of production rules [28]. One rule is created for each path from the root to a leaf. Each attribute-value pair along a path forms a condition, and the leaf node holds the class prediction. In general, the rules have the form: IF *condition*1 AND *condition*2 AND *condition*3 THEN outcome. For example, IF $Weather$ = "*Rainy*" AND $Wind$ = "*Strong*" THEN *Play*? = 'No" is a rule extracted from the decision tree in Figure 2.2.

Decision tree evaluation The most commonly used performance criterion for a decision tree is the *classification accuracy rate* which is the percentage of test set samples that are correctly classified. For decision trees with binary target variables (positive/negative class), various combinations of *sensitivity* (the number of correctly predicted positives divided by the total number of positives) and *specificity* (the number of correctly predicted negatives divided by the total number of negatives) are also considered as measures of accuracy [8].

Chapter 3

Related Work

Our work is related to the lines of work on (a) example-based queries, (b) interactive learning of queries, (c) cost-efficient decision tree construction, and (d) set expansion.

3.1 Example-Based Queries

The problem of discovering queries based on examples has its root in QBE [46] and has been lately studied for reverse engineering queries in various domains (e.g., relational [18] and graph data [4, 25]). Our work is related to this line of work in that it can be applied to discover target queries based on example tuples if the candidate queries are known or can be enumerated.

Tran et al. [37] devise a data classification-based technique to generate an instance equivalent query Q' whose output is equivalent to the output of a given query Q on a database D. The generated query is a select-project-join (SPJ) query where all the join predicates are foreign-key joins. Considering the example tuples as positive tuples and all other tuples in the database as negative tuples, they construct a binary decision tree to classify the tuples using the gini index as the goodness criterion for splitting. The final decision tree is translated into a disjunctive query by adding the conjunctions of the splitting conditions in the path from the root to each leaf node that contains the positive examples. In a later work [38], they provide an extension by supporting more expressive instance equivalent queries (IEQs), such as SPJ with union operators (SPJU-IEQs) and SPJ with group-by aggregation operators

(SPJA-IEQs), and supporting multiple versions of the input database.

Zhang et al. [45] study the query reverse engineering problem where given a database instance and the output of a query on that instance, they find a join query that generates the same output on the same instance. Their main insight is that any query graph can be characterized as a union of disjoint paths connecting its projection tables to a center table and a series of merge steps over the disjoint paths. They refer to this union as a star and the center table as the star center. Using the insight, they use a lattice structure to filter out the potential candidates that need not be tested, where each vertex represents a star and the edges represent the merge steps.

Shen et al. [34] study a slightly relaxed version of the problem where a minimal project-join query that contains the given example tuples on its output is sought. They develop three different algorithms that can be used to verify the generated candidate queries. Their "VERIFYALL" algorithm simply executes each of the candidate queries and verify whether its result contains all the example tuples. The "SIMPLEPRUNE" algorithm prunes candidate queries using subtree-supertree relationship without verifying for any of the rows. Their final algorithm is "FILTER" which is used to quickly prune candidate queries using several well-defined filters.

Weiss et al. [43] investigates the complexity of learning an SPJ query that returns all positive examples but none of the negative examples on a given database. They consider different factors in their complexity analysis such as the size of the query to be learned, the size of the schema, the number of examples, etc., and show that the problem of determining whether such query exists (satisfiability) is NP-hard when there is a bound on the size of the query.

Tan et al. [35] study the reverse engineering problem for OLAP (OnLine Analytical Processing) queries with group-by and aggregation. Their three-phase algorithm named REGAL(Reverse Engineering Group-bys, Aggregates, and seLections) first identifies a set of group-by candidate columns for the target query based on a lattice structure. Then it discovers candidate combinations of those columns and aggregations that are consistent with the given output by applying a set of aggregation constraints. Finally, it finds the nec-

essary conjunction of selection predicates over the table columns, which is used as the selection condition in the target query to generate the exact given output.

An underlying assumption in many of these works is that the queries can be discovered on small instances (where the answer tuples can be easily listed) before being applied to larger instances. Moreover, all the aforementioned works focus on specific query types to keep the complexity of query generation under control, whereas in our work there is no restriction on query types but the assumption is that the sets (or queries) are known or can be enumerated.

3.2 Interactive Learning of Queries

There have been also works on interactive exploration to learn a desired query. Abouzied et al. [1] study the problem of learning quantified boolean queries, based on membership questions answered by users. They show that learning quantified boolean queries is intractable and that efficient solutions for a smaller class, referred to as role-preserving qhorn queries, are reachable.

Bonifati et al. [6, 7] infer join queries based on interactions in the form of simple yes/no answers about the presence of tuples in the final output. To keep the user interaction as minimal as possible, they propose a few local strategies that explore the lattice of join predicates and two lookahead strategies for presenting a tuple to the user. The local strategies work by navigating through the lattice from most specific to most generic join predicates (top-down strategy) or vice versa (bottom-up strategy) while pruning uninformative nodes, whereas the lookahead strategies work by looking one-step or two-step ahead to select a tuple that guarantees the maximum number of uninformative tuples if is labeled by the user.

Dimitriadou et al. [10] predict a DNF (disjunctive normal form) query, based on relevance feedback on sample tuples produced by a decision tree, aiming at reducing the number of samples and the exploration time. Initially, the user is presented with a few sample tuples to characterize as relevant or not. Then, the labeled samples are used to train a decision tree classifier

that predicts the samples relevant to the user interest. In each iteration, more samples are extracted using the classifier and presented to the user. With each feedback, the decision tree is revised, and the final tree is mapped to a query.

Li et al. [21] take a sample database and a query result on that sample, as input, and generate candidate SPJ queries, using the approach of Tran et al. [38]. In each follow-up interaction, the user is provided with a modified database and a collection of query results on that database and chooses one query result as the correct one. With each user feedback, some of the candidate queries are removed until a single query emerges.

These approaches as well are only applicable when the target query is of a specific type or falls within a specific class (e.g., join queries, conjunctive queries, etc.).

3.3 Cost-Efficient Decision Tree Construction

The problem of cost-efficient decision tree construction is well studied in the literature, and several popular algorithms (e.g., ID3 [27] and C4.5 [29]) have been developed. A popular heuristic used by these algorithms for selecting the next feature or the splitting attribute is the information gain measure. The feature with the largest information gain is selected to split the dataset. Though these algorithms often target the problem of classification where the maximum accuracy is sought, the heuristic may also help to construct a full decision tree that minimizes the average and the maximum root-to-leafs path lengths.

Since the problem of constructing a cost-efficient optimal binary tree is NP-complete [16], Adler et al. [2] propose a greedy algorithm that achieves $(\ln n + 1)$-approximation, by simply choosing an entity at each decision node that most evenly partitions the collection of items.

Another greedy tree construction algorithm in the context of finding desired tuples in a structured database is used by Roy et al. [5]. It selects an attribute that minimizes the number of indistinguishable pairs of sets.

In contrast to these greedy approaches that are considered as 1-step looka-

head strategies, our work provides an efficient k-steps lookahead algorithm and two variations of it, which are not only more effective in finding a better solution than 1-step lookahead approaches but also efficient because of our pruning strategy.

To allow trade-off between tree quality and learning time, Esmeir et al. [12] also propose lookahead-based algorithms for anytime induction of decision trees by developing k-steps entropy and k-steps information gain. However, our lookahead algorithms are 2 to 5 orders of magnitude faster than theirs, thanks to our pruning strategy.

3.4 Set Expansion

Set expansion refers to the problem of expanding a small set of "seed" entities into a complete set by discovering other entities that also belong to the same "concept set". For example, if a given seed set consists of a few cities in Canada, such as Edmonton, Toronto, and Vancouver, then set expansion should return a more complete set with the other entities in the same semantic class, such as Montreal, Calgary, Ottawa, Sudbury, Victoria, Winnipeg, etc., that are also the cities in Canada. This problem is related to ours in that it aims at retrieving a complete set based on a few given examples.

To address the set expansion problem, Wang et al. [40–42] propose the SEAL (Set Expander for Any Language) system using a two-phase extraction and ranking architecture. In the extraction phase, they construct a customized wrapper for each semi-structured page that contains the seeds using the maximally long left and right contexts that encloses all the seed entities. These wrappers are applied on the corresponding pages to extract the candidate entities. In the ranking phase, a graph is built containing all the seeds, the constructed wrappers, and the extracted candidate entities. Then, the entities are ranked using the aggregated similarity of the entities to the seeds based on many random walks through the graph.

Sarmento et al. [31] present a corpus-based approach using the co-occurrence statistics of the entities in a text collection. They define a membership func-

tion that computes the similarity between an entity and a set of seed entities using the vector space model (VSM) where each entity is represented by a vector of numerical features and can be compared using a standard distance measure such as the cosine similarity measure. Then, the set of n entities with the highest scores computed by the membership function is selected as the expanded set.

He et al. [14] propose a class of algorithms called SEISA (Set Expansion by Iterative Similarity Aggregation) using the jaccard or the cosine similarity between the entities calculated based on their co-occurrences in a collection of web lists or query logs. Their *static thresholding* algorithm estimates a threshold k based on the similarity score distribution of the entities to the given seeds and selects top k similar entities as the initial elements of the expanded set (the concept set). Then, it iteratively updates the expanded set based on the aggregated similarity score of an entity to the seeds (relevance score) and the entities (coherence score) in the current expanded set until no further changes can be made. On the other hand, their *dynamic thresholding* algorithm estimates the threshold k at every iteration. Therefore, the convergence is not guaranteed and the algorithm stops after a fixed number of iterations.

Shen et al. [33] present a framework called SetExpan for the set expansion problem on a text corpus based on a denoised context feature selection and a ranking-based unsupervised ensemble method. The framework uses two types of context features obtained from the text corpus: skip-grams and coarse-grained types. They form a bipartite graph and assign weights on the edges between each pair of entities and context features using the tf-idf scoring, considering each entity as a "document" and each of its context features as a "term". In the context feature selection step, it calculates a score for each context feature based on its accumulated strength with currently expanded entities and selects the top Q features with the highest scores. In the entity selection step, the algorithm first generates T subsets of the context features by sampling without replacement. Then, it calculates the rank of each entity based on its similarity score to the currently expanded entities conditioned on each subset of the features using a weighted jaccard similarity measure.

It obtains the mean reciprocal rank for each entity by summing up all the reciprocal ranks for that entity based on each feature subset. Finally, all the entities with the mean reciprocal ranks above some threshold are added into the expanded entity set. The algorithm iterates the context feature selection and the entity selection steps until the expanded entity set reaches the expected output size.

Jindal et al. [17] present an inference-based method for set-expansion by allowing both positive and negative examples in the seed set. Their method computes the similarity scores of each entity to every positive example and negative example in the seed set and ranks the entities based on those scores by considering the scores to the positive examples as rewards and the scores to the negative examples as penalties. The intuition is that the entities that have high similarity to the positive examples are more likely to belong to the concept set, while entities that have high similarity to the negative examples are less likely to belong to the concept set.

Huang et al. [15] propose a framework called Set-CoExpan that generates multiple auxiliary sets as negative sets and expands the target set and the auxiliary sets simultaneously, where each set tries to avoid elements in other sets. The auxiliary sets are closely related to but different from the target semantic class and are generated using hierarchical clustering. In the auxiliary sets generation step, related words to each seed element in the embedding space are grouped according to their semantic types using intra-seed clustering. Then, inter-seed clustering is applied to those groups to form auxiliary sets. In the expansion step, the target set and the auxiliary sets are expanded by selecting the context features that make each set cohesive while distinguishing from other sets using a greedy approach. Finally, the entities are ranked using the selected features based on their similarity to the current expanded set and are added to the set.

All the aforementioned works and the other works on set expansion ([26], [23], [44], [39], etc.) are related to our work in that these works aim at retrieving a complete set based on examples. However, there are two key differences between this line of work and ours: (1) the target set in these works may not

be in the collection, and the relevant entities are collected (potentially from multiple sets in the collection) based on their similarities to the seed set; (2) these approaches are approximate one-shot solutions that may or may not retrieve the desired set. Whereas in our case, the target set must be in the collection and is guaranteed to be found.

Chapter 4

Problem Formulation

Consider a collection of n candidate sets and a target set in the collection that needs to be identified. Without loss of generality, we assume the sets are all unique; if not, duplicates can be removed without affecting the search task. We want to find a desired set through a set of membership questions that the user answers (e.g., Is A in the target set?). At a high level, we want to minimize the number of interactions.

A general approach to the search problem is to construct a decision tree with the candidate sets placed at the leaves and each internal node representing a question. With interactions limited to yes/no membership questions, the decision tree will be a full binary tree with n leaves and $n-1$ internal nodes. The number of such decision trees that can be constructed is huge[1], and some of those trees are more efficient for finding the target set than others.

Let m denote the size of the universe from which the sets are drawn. For a collection C of finite sets, $m = |\bigcup_{s \in C} s|$. In our presentation, we may refer to the members of the universe as entities, though our approach is applicable to any sets of tuples (e.g., sets of relationships). For a fixed tree shape with $n-1$ internal nodes, the number of possible placements of m entities or tuples on internal nodes will be $m(m-1)\ldots(m-n+2) = \frac{m!}{(m-n+1)!}$, assuming that each entity appears at most once in the tree. Otherwise, this number is even larger. Searching for an efficient decision tree among all these tree shapes and possible placements of entities on internal nodes is a major computational challenge,

[1]The actual number is the $(n-1)$th Catalan number, i.e., $\frac{1}{n}\binom{2(n-1)}{n-1} = \frac{(2(n-1))!}{n!(n-1)!}$.

and that is the problem studied in this thesis.

To alleviate the problem, one may group entities in C into *informative* and *uninformative*. An entity that is either present in all sets in C or none is not informative, since a membership question about that entity does not reduce the search space. The rest of the entities can be considered as informative. Clearly we want to limit our questions to informative entities, and only place those entities on the internal nodes.

Example 4.0.1. Consider the collection of seven sets, as shown in Figure 4.1. Entity a is uninformative since it is present in all sets. All the other entities $b, c, ..., k$ are informative. Figure 4.2 shows three possible decision trees that represent the sets in the collection. All the trees are full binary decision trees with 6 internal nodes and 7 leaves. The root node corresponds to all sets of the collection. In Figure 4.2a, the left branch corresponds to the sub-collection $\{S1, S2, S3\}$ where entity d is present and the right branch corresponds to the sub-collection $\{S4, S5, S6, S7\}$ where d is not present. Each branch is further broken down based on the presence or absence of entities.

$$\begin{array}{lll} S1 = \{a, b, c, d\} & S2 = \{a, d, e\} & S3 = \{a, b, c, d, f\} \\ S4 = \{a, b, c, g, h\} & S5 = \{a, b, h, i\} & S6 = \{a, b, j, k\} \\ S7 = \{a, b, g\} & & \end{array}$$

Figure 4.1: A collection of example sets

Given a decision tree, the number of questions that are required to find a set is determined by the depth at which the set is placed. For example, in Figure 4.2a, $S2$ can be detected using two questions whereas one will need three questions to find any other set. Since we do not know the target set in advance, and assuming that all sets are equally likely, the cost of a tree can be defined as the average depth of the leaves which equivalently represents the expected number of questions required to find the target set.

Definition 4.0.2. Let T be a full binary decision tree over a collection C of unique sets, i.e., T has exactly $|C|$ leaves and each leaf is labelled with a set in C. If *depth(s,T)* denote the depth of a set s in T, then the cost of T is defined

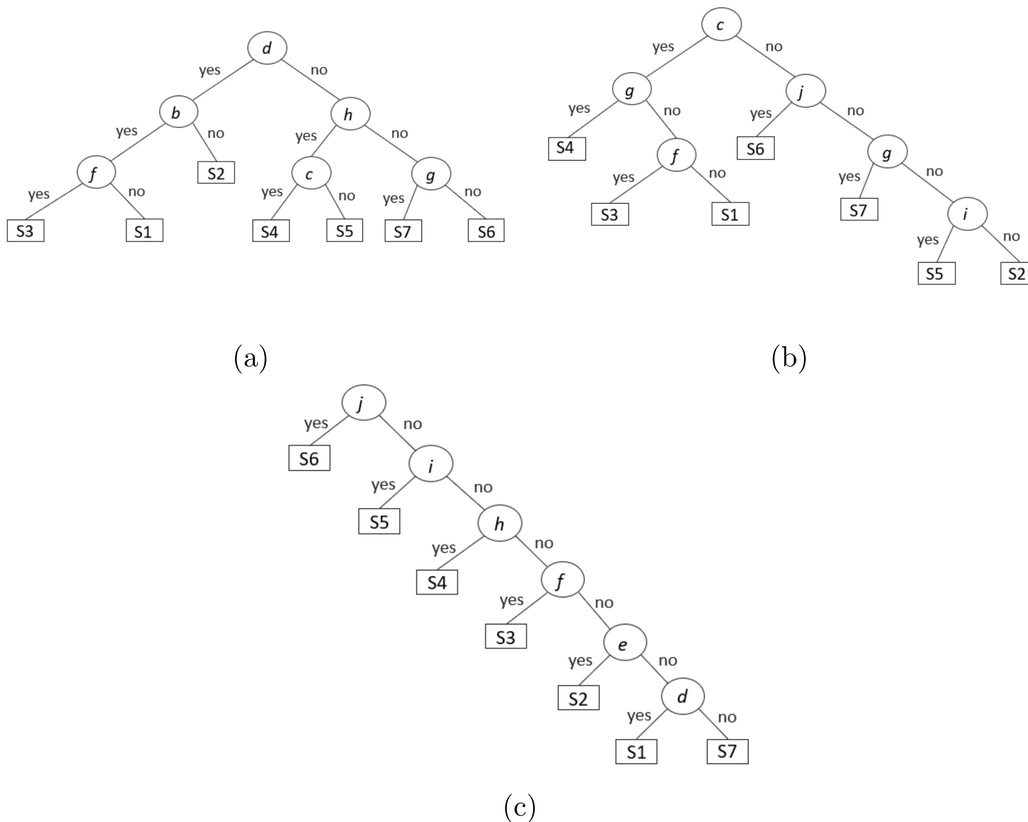


Figure 4.2: Example of decision tree representations of the sets in Figure 4.1

as

$$cost(T) = \frac{\sum_{s \in C} depth(s, T)}{|C|}.$$

Alternatively, one can also define the cost of a tree as the height[2] of the tree.

For a collection with n unique sets, the height of a full binary decision tree cannot be less than $\lceil \log_2 n \rceil$ for $n > 0$. This sets a lower bound on the height (H) of an optimal tree, which we refer to as $LB_H(n)$. The next lemma gives the lower bound on the average depth of the leaves.

Lemma 4.0.3. *Given a collection of n unique sets such that $n > 0$, a lower bound on the average depth of the leaves (AD) of a full binary decision tree representing the collection, denoted as $LB_AD(n)$, is $\frac{\lceil n \log_2 n \rceil}{n}$.*

[2]Here height refers to the depth of the leaf with the longest distance from the root, i.e., the number of questions to be answered to reach the deepest leaf.

Proof. The average depth of the leaf nodes of a full binary decision tree representing n sets cannot be less than $\log_2 n$. Hence, the sum of depth of the n leaf nodes cannot be less than $\lceil n \log_2 n \rceil$ since it must be an integer number. Therefore, the lower bound on AD for n unique sets can be expressed as $\frac{\lceil n \log_2 n \rceil}{n}$. □

Now let's examine the trees in Figure 4.2 again. The lower bound on AD of any full binary decision tree representing a collection of 7 sets, according to Lemma 4.0.3, is 2.857. The AD of the tree in Figure 4.2a is 2.857, Figure 4.2b is 3.0 and that of the tree in Figure 4.2c is 3.857, meaning the first tree is optimal but the others are not.

Given a collection of n unique sets, our goal can be stated as finding a full binary decision tree representation of the collection with the least cost where the cost metric is either AD or H.

Chapter 5

Methodology

Given a collection C of unique sets, our set discovery process constructs a decision tree with the sets in C placed at the leaves. Since there are many possible trees that can be constructed, and some are more efficient than others, our goal is to find a tree that leads to the least number of interactions with the user. Before presenting our algorithms, we develop a few lower bounds on cost, which will be used in pruning the search space of our algorithms.

5.1 Cost Lower Bounds

Given a collection C of unique sets, and without considering the entities that are being selected, two lower bounds on cost (as discussed in Section 4) are

$$LB_AD_0(C) = \frac{\lceil |C| * \log_2 |C| \rceil}{|C|}, and \tag{5.1}$$

$$LB_H_0(C) = \lceil \log_2 |C| \rceil \tag{5.2}$$

for cost metrics AD and H respectively. Now consider an entity e that partitions C into two sub-collections $C1$ and $C2$. Our cost lower bounds after looking 1-step ahead with e as the first question can be written as

$$LB_AD_1(C, e) = \frac{|C1| * LB_AD_0(C1) + |C2| * LB_AD_0(C2)}{|C|} + 1, and \tag{5.3}$$

$$LB_H_1(C, e) = max(LB_H_0(C1), LB_H_0(C2)) + 1 \tag{5.4}$$

for cost metrics AD and H respectively. We use the general term LB to refer to any lower bound (including LB_AD and LB_H) when a distinction in the cost metric is not important.

Let E denotes the set of entities in collection C. A lower bound on cost over all entities with 1-step look ahead is

$$LB_1(C) = min_{e \epsilon E} LB_1(C, e). \tag{5.5}$$

Given a collection C and entity e, similar lower bounds for k-steps look ahead with $k > 0$ can be expressed as

$$LB_AD_k(C, e) = \frac{|C1| * LB_AD_{k-1}(C1) + |C2| * LB_AD_{k-1}(C2)}{|C|} + 1, \tag{5.6}$$

$$LB_H_k(C, e) = max(LB_H_{k-1}(C1), LB_H_{k-1}(C2)) + 1 \tag{5.7}$$

for cost metrics AD and H respectively. A lower bound over all entities is

$$LB_k(C) = min_{e \epsilon E} LB_k(C, e). \tag{5.8}$$

A desirable property of these lower bounds is their monotonicity and that the cost lower bounds never decrease. This means the lower bounds can get tighter but not looser, as we look more and more steps ahead. The next two lemmas state this more formally.

Lemma 5.1.1. *For any collection C, $LB_k(C)$ is a monotone non-decreasing function of k, i.e., for non-negative integers $k1$ and $k2$, if $k2 > k1$, then $LB_{k2}(C) \geq LB_{k1}(C)$.*

Proof. The proof is by induction on k. For the basis, the lowest possible cost of a binary decision tree on C, defined as $LB_0(C)$, is calculated assuming that the entity in each node of the tree partitions the sub-collection as evenly as possible. But, $LB_1(C)$ is calculated after an actual entity from the collection is assigned to the root node and assuming that all other nodes partitions the sub-collections as evenly as possible. If the entity at the root partitions the collection as evenly as possible then $LB_1(C)$ is equal to $LB_0(C)$. Otherwise, $LB_1(C)$ is greater than $LB_0(C)$. For the induction step, suppose the claim

holds at step $k1$. In each additional step $k2 = k1+1$ of the lower bound calculation, an additional level of nodes are assigned with the best entities recursively. If any of those entities does not partition the corresponding sub-collections as evenly as possible then $LB_{k2}(C) > LB_{k1}(C)$, otherwise, $LB_{k2}(C) = LB_{k1}(C)$. Therefore, the statement holds. □

Lemma 5.1.2. *For any collection C and entity e in the collection, $LB_k(C, e)$ is a monotone non-decreasing function of k, i.e., for positive integers $k1$ and $k2$, if $k2 > k1$, then $LB_{k2}(C, e) \geq LB_{k1}(C, e)$.*

The proof follows the same reasoning as the one in Lemma 5.1.1.

5.2 Entity Selection

The problem of constructing an optimal binary decision tree, minimizing the cost to discover an unknown target set, is NP-complete [16], hence various greedy entity selection strategies have been studied in the literature. In this section, we briefly review these strategies and compare them with ours.

Most even partitioning A greedy approximation algorithm that achieves $(\ln n + 1)$-approximation for the decision tree problem on a collection C with n sets is simply to choose an entity at each internal node that most evenly partitions the collection of sets in that node [2]. In other words, it selects an entity that minimizes the difference between the size of the two sub-collections, $C1$ where the entity is present and $C2$ where the entity is absent, of a collection C.

Information gain Decision tree construction is a very well-understood process in machine learning and data mining. A popular heuristic used by the decision tree algorithms (such as ID3 [27] and C4.5 [29]) for selecting the next feature or entity is the information gain. The entity with the largest information gain is selected to split the collection. If we treat each set in C as a class and each distinct entity e as a feature, then the information gain of e that partitions C into sub-collections $C1$ and $C2$ can be written as

$$InfoGain(C,e) = \log_2|C| - \frac{|C1| * \log_2|C1| + |C2| * \log_2|C2|}{|C|}. \quad (5.9)$$

Gain ratio Information gain is biased towards attributes with a large number of distinct values. Hence, Quinlan [29] proposed a normalization, known as the gain ratio. However, all attributes/entities have the same number of distinct values (yes/no) in our set discovery problem. Therefore, this normalization is not needed.

Gini index Another attribute selection measure used in CART [9] is the gini index. For an entity e that partitions a collection C of distinct sets, where each set is equally likely, into sub-collections $C1$ and $C2$, the gini index of e can be written as

$$Gini(C,e) = \frac{|C1|}{|C|} * (1 - \frac{1}{|C1|}) + \frac{|C2|}{|C|} * (1 - \frac{1}{|C2|}) = 1 - \frac{2}{|C|}. \quad (5.10)$$

According to Equation 5.10, all entities have the same gini index regardless of how they partition a collection. Therefore, the gini index is not applicable in our set discovery problem.

Indistinguishable pairs Another entity selection strategy (used by Roy et al. [5]) selects an attribute or entity that minimizes the number of indistinguishable pairs of sets. For an entity e that partitions a collection C into sub-collections $C1$ and $C2$, the number of indistinguishable pairs is given as

$$Indg(C,e) = \frac{|C1| * (|C1| - 1) + |C2| * (|C2| - 1)}{2}. \quad (5.11)$$

Cost lower bound Entity selection can be done using our cost lower bounds, as discussed in Section 5.1, with the entity that minimizes a cost lower bound of a collection being selected as the next question and being presented to the user. This is the strategy we use in this paper because of some of the desirable properties of those lower bounds. However, in some cases, two entities that do not partition a collection in the same way may have the same value of a lower bound. For example, suppose entity a partitions a collection of 16 sets

into 9 and 7 sets, and entity b partitions the same collection into 10 and 6 sets. With $\lceil \log_2 9 \rceil = \lceil \log_2 10 \rceil = 4$, both entities will have the same value of the lower bound on height. When there are such ties, we select an entity that most evenly partitions the collection to differentiate between entities with the same value of cost lower bound.

Though these strategies seem different from each other, it can be shown that they select the same entity for the binary decision tree problem. Hence, they all achieve the same $(\ln n + 1)$-approximation factor.

Lemma 5.2.1. *Given a collection C, the strategies (a) information gain, (b) indistinguishable pairs, and (c) 1-step cost lower bound select the same entity that partitions C most evenly into two sub-collections.*

Proof. (a) In Equation 5.9, since $|C|$ is constant and $|C1| + |C2| = |C|$, the quantity $|C1| * \log_2 |C1| + |C2| * \log_2 |C2|$ is minimum when C is most evenly partitioned into $C1$ and $C2$. Hence, the entity that partitions C most evenly has the largest information gain and is selected by *information gain* strategy.
(b) Similarly, in Equation 5.11, $|C1| * (|C1| - 1) + |C2| * (|C2| - 1)$ is minimum when C is most evenly partitioned. Therefore, the entity that partitions C most evenly has the minimum value of Indg() and is selected by *indistinguishable pairs* strategy.
(c) It can also easily be seen from Equations 5.3 and 5.4 (after replacing LB_AD_0 and LB_H_0 with their respective values from Equations 5.1 and 5.2) that, an entity that most evenly partitions the collection C into $C1$ and $C2$, gives the minimum value of $LB_1()$ in Equation 5.5, thus is selected by our *1-step cost lower bound* strategy. □

These approaches all can be considered as 1-step lookahead. However, due to the monotonicity of cost lower bound, we can develop more efficient k-steps lookahead strategies. We cannot say the same about other entity selection strategies. In particular, the k-steps cost lower bound (as defined in Section 5.1) allows us to develop a simple but effective pruning strategy that significantly reduces the runtime of lookahead strategies without affecting the cost. Although k-steps lookahead strategies have been studied for entropy [6]

and information gain [12], we are not aware of similar pruning strategies for these measures.

5.3 Pruning

We want to find a decision tree that requires the least number of interactions with the user for a set discovery hence has the least cost. However, exhaustive searching the space of possible trees for the one with the least cost is computationally intensive and not always feasible. In this section, we propose a *pruning* strategy to reduce the size of the search space without affecting the correctness.

Lemma 5.3.1. *Let $LB_k(C, e)$ denote our lower bound of cost for entity e in collection C by looking k-steps ahead, and suppose entity selection is done based on $LB_k()$, k-steps lookahead for some k. Now consider entities e_1 and e_2, both in C. If $LB_l(C, e_2) \geq LB_k(C, e_1)$ for $l \leq k$, then e_2 can be pruned without affecting the correctness of the search.*

Proof. When $LB_l(C, e_2) \geq LB_k(C, e_1)$ for $l \leq k$, then $LB_k(C, e_2)$ cannot be smaller than $LB_k(C, e_1)$ based on Lemmas 5.1.1 and 5.1.2, and e_2 can be pruned without affecting the correctness of the search. □

As an example, consider the collection of sets shown in Figure 4.1, denoted as $C1$, and let H be our cost metric. The 1-step lower bound, $LB_H_1()$, for entities c and d is 3, and that for all other informative entities is 4. Suppose, we are interested in 3-steps lower bound and it is already calculated for d as $LB_H_3(C1, d) = 3$. Since $LB_H_1()$ for all other entities is not less than 3, any further calculation for them can be pruned safely.

Now, consider another collection where the sets are the same as in collection $C1$ except $S1 = \{a, b, c\}$ and $S4 = \{a, b, c, d, g, h\}$, and let us denote this collection with $C2$. The set counts for all entities are as before, and the 1-step lower bound, $LB_H_1()$, for the informative entities remain the same as in collection $C1$. But, the 3-steps lower bound for d, $LB_H_3(C2, d)$, is 4 now. Therefore, we cannot prune the 3-steps lower bound calculation for entity

c using the 1-step lower bound, $LB_H_1(C2, c)$. However, the 2-steps lower bound for c, $LB_H_2(C2, c)$, is 4, hence any further calculation for c can be pruned now using the 2-steps lower bound since it is not less than the already calculated least 3-steps lower bound for entity d.

Implementation There are several places where our pruning is applied. First, entities are sorted based on their 1-step lower bounds in non-decreasing order, the k-steps lower bounds for entities are calculated in that order, and the least value found so far is updated accordingly. If the 1-step lower bound of an entity e is not less than the already found least k-steps lower bound, then the k-steps lower bound calculations of entity e and all the subsequent entities in the sorted order are pruned.

Second, when calculating the k-steps lower bound for an entity, the already found least value is used to set an upper limit for each of the recursive steps of the calculation. Whenever the upper limit is reached, the rest of the k-steps lower bound calculation for the current entity is pruned. Since, for an entity e to be selected, $LB_k(C, e)$ needs to be less than the already found least value of the lower bound (AFLV), if e partitions a collection C into $C1$ and $C2$, the upper limit (UL) for the accepted value of $LB_{k-1}(C1)$ can be calculated using Equation 5.6, for the cost metric AD, by replacing $LB_AD_k(C, e)$ with AFLV and $LB_AD_{k-1}(C2)$ with the least possible value $LB_AD_0(C2)$ as

$$UL(C1) = \frac{(AFLV - 1) * |C| - |C2| * LB_AD_0(C2)}{|C1|}, \tag{5.12}$$

and similarly using Equation 5.7, for the cost metric H, as

$$UL(C1) = AFLV - 1. \tag{5.13}$$

Once the actual value of $LB_{k-1}(C1)$ is calculated, the upper limit (UL) for the accepted value of $LB_{k-1}(C2)$ can be calculated, for the cost metric AD, as

$$UL(C2) = \frac{(AFLV - 1) * |C| - |C1| * LB_AD_{k-1}(C1)}{|C2|}, \tag{5.14}$$

and for the cost metric H, as

$$UL(C2) = AFLV - 1. \tag{5.15}$$

5.4 Lookahead Strategies

Now that we have covered our cost functions and pruning strategies, we present our k-steps lookahead strategy and two variations of it, for selecting the next question to ask by looking k-steps ahead. These strategies choose an entity based on the k-steps lower bound for the cost of a collection, as discussed in Section 5.1. When there are ties between two or more entities in terms of cost, the entity that partitions the collection most evenly is chosen. If there are still ties for the choice of entities, then an entity is selected randomly from the set of candidates. The algorithm applies our pruning strategy in every step where the search space can be cut without compromising the required number of questions, as discussed in Section 5.3.

k-Lookahead with Pruning (k-LP) Algorithm 1 presents our *k-lookahead with pruning* strategy. It takes a collection C of unique sets, the number of steps k to look ahead, and an upper limit ul of the k-steps lower bound for an entity to be selected, as input. Initially, the upper limit is set to a large number. Then, it sorts the entities in the collection based on their partitioning capability from the most even to the least even (Line 11). Since, the entity that partitions a collection most evenly has the minimum value of the 1-step cost lower bound, the entities will also be sorted based on their 1-step lower bound of cost in non-decreasing order. This way, an entity with both the minimum lower bound and also the most even partitioning capability is considered first, breaking possible ties on the cost lower bound. For each entity in the sorted order that partitions the collection C into two sub-collections C^+ and C^-, the $(k-1)$-steps lower bounds for C^+ and C^- are calculated by recursively calling the algorithm (Lines 16-31). Those quantities are plugged into Equation 5.6 or 5.7 (depending on the cost metric used) to obtain the k-steps lower bound l for each entity (Line 32). The algorithm keeps track of the entity with the least k-steps lower bound l and sets it as the upper limit ul for the next entity to be considered (Lines 33-35). Since the entities are sorted, if it finds an entity with an equal or larger 1-step lower bound than the upper limit ul, the algorithm stops early and prunes all the remaining entities (Lines 14-15). To

further reduce the search space, it calculates the upper limits for C^+ (using Equation 5.12 or 5.13) and C^- (using Equation 5.14 or 5.15) and passes them to the recursive call (Lines 21-22 and 28-29). If no entity can be selected with a lower value of $(k-1)$-steps lower bound than the calculated upper limit, then it stops processing the current entity and moves to the next entity (Lines 23-24 and 30-31). Finally, the algorithm returns an entity e with the minimum k-steps lower bound of cost (Lines 7-10 or 37). To speed up the calculations, the algorithm uses memoization by storing and reusing the results for different inputs of collection C and steps, k (Lines 1-6, 9, and 36).

Two important observations can be made about our k-LP algorithm. First, it can be shown that the algorithm finds *an optimal solution* if k is set to the height of an optimal tree or a greater value. Second, the early stopping opportunities, which are based on our pruning strategy presented earlier, sets apart our lookahead strategies from the existing lookaheads in literature [6, 12].

For a collection of n unique sets and m distinct entities, the runtime of Algorithm 1 is $O(m^k n)$ since finding 1-step lower bounds for m entities is $O(mn)$ and in each recursive step, there will be $O(m)$ calls to the next step. Our next two strategies further reduce the time by setting bounds on the number of candidate entities.

k-LP with Limited Entities (k-LPLE) Despite all the pruning done using our lower bounds, the runtime of our k-LP algorithm increases as a polynomial function of m (e.g., quadratic for $k = 2$), and the algorithm becomes very inefficient for large values of k. On the other hand, the chance of constructing a better tree increases as we increase k. One good trade-off between the lookahead steps k and the number of candidate entities m is to limit the candidate entities in each step of the lower bound calculation to a few top entities, say q, ranked in terms of the 1-step lower bound of cost. For $q << m$, the reduction in runtime can be significant. This can be implemented by adding an extra input q to Algorithm 1, modifying Line 11 so that SE contains only the first q sorted entities, and passing q on the recursive calls to the algorithm in Lines 22 and 29.

Algorithm 1 K-Lookahead with Pruning (K-LP)

Input: collection C of unique sets, steps k, and upper limit ul of the k-steps cost lower bound for an entity to be selected
Output: selected entity and it's k-steps cost lower bound

```
 1: if (C, k) ∈ Cache then
 2:     e, l ← Cache[(C, k)]
 3:     if ul ≤ l then
 4:         return null, l
 5:     else if e ≠ null then
 6:         return e, l
 7: if k = 1 then
 8:     let entity e to most evenly partition C
 9:     Cache[(C, k)] ← (e, LB_1(C, e))
10:     return Cache[(C, k)]
11: SE ← sort entities according to most even partitioning of C
12: e ← null
13: for each entity e_i ∈ SE do
14:     if LB_1(C, e_i) ≥ ul then
15:         break
16:     let C^+ be the collection {S_j ∈ C | e_i ∈ S_j}
17:     C^- ← C − C^+
18:     if |C^+| = 1 then
19:         l^+ ← 0
20:     else
21:         ul^+ ← Upper-Limit (ul, |C^+|, LB_0(C^-), |C^-|, |C|)
22:         e^+, l^+ ← K-LP (C^+, k − 1, ul^+)
23:         if e^+ = null then
24:             continue
25:     if |C^-| = 1 then
26:         l^- ← 0
27:     else
28:         ul^- ← Upper-Limit (ul, |C^-|, l^+, |C^+|, |C|)
29:         e^-, l^- ← K-LP (C^-, k − 1, ul^-)
30:         if e^- = null then
31:             continue
32:     l ← K-Steps-Lower-Bound (|C^+|, l^+, |C^-|, l^-, |C|)
33:     if l < ul then
34:         ul ← l
35:         e ← e_i
36: Cache[(C, k)] ← (e, ul)
37: return e, ul
```

k-LP with Limited but Variable number of Entities(k-LPLVE) The runtime of k-LPLE may further be reduced by greedily considering only a single entity in each recursive step of the k-steps lower bound calculation for an entity. The intuition here is that an entity with the smallest 1-step lower bound is more probable to be the best choice. Hence, our k-LPLVE strategy limits the number of candidate entities to only one (with the least 1-step lower bound) during each step of the k-steps lower bound calculation for the q entities. With this strategy, the search time is expected to reduce further, but the quality of the results is not expected to change much (see Section 6 for evaluation results). This strategy can be implemented by performing the same modifications as k-LPLE in Algorithm 1 except that the constant value of 1 is passed as q on the recursive calls in Lines 22 and 29. This means the function is called from outside with q, and SE in Line 11 takes the first q sorted entities during that call and only the first entity during the subsequent recursive calls to the function made by itself.

5.5 Set Discovery

The set discovery scheme studied in this paper is an interactive process that starts with an initial question posed to the user and continues with follow-up questions based on the user's answers. The lookahead strategies choose an entity to be the next question, which is expected to minimize the cost of discovering the user's desired set. With each user feedback, the same selection process continues until the user's desired set is discovered or the user is satisfied with the refined sub-collection of sets and does not want to answer more questions.

The general approach for *set discovery* is presented in Algorithm 2. It takes the entire collection C of unique sets and a user-provided initial set I as inputs and finds the sub-collection CS containing all the supersets of I in C (Lines 2-4). It then iteratively, selects the best entity e according to the entity selection strategy denoted by $\mathbf{\Upsilon}$, asks the user a question about the presence of that entity in the desired set, and re-calculates the sub-collection CS of

candidate sets based on the user feedback until a single set is left or the halt condition $\boldsymbol{\Gamma}$ (e.g., the user does not want to answer more questions) is met (Lines 5-12). Finally, it returns the remaining sets that are consistent with the user's answers (Line 13).

The runtime of Algorithm 2 depends on the number of questions required to discover the desired set and the strategy used. In the worst case, the number of questions can be $n - 1$ for a collection of n sets.

Algorithm 2 Set Discovery

Inputs: collection C of unique sets and initial set I
Output: sets that are consistent with the user's answers
Parameter: entity selection strategy $\boldsymbol{\Upsilon}$ and halt condition Γ

```
1: CS ← ∅
2: for each set S_i ∈ C do
3:     if I ⊆ S_i then
4:         CS ← CS ∪ {S_i}
5: while |CS| > 1 and Γ is false do
6:     e ← ϒ(CS)
7:     α ← query the user about the presence of e in target set
8:     let P be the collection {S_i ∈ CS | e ∈ S_i}
9:     if α is true then
10:        CS ← P
11:    else
12:        CS ← CS − P
13: return CS
```

Offline tree construction Our tree construction may be done offline for static collections, for example, when the initial query sets are known in advance or are always empty. Algorithm 3 provides the steps for precomputing a decision tree on a collection of sets. With the decision tree constructed offline, a set discovery can be efficiently performed by asking questions and following only a single path through the tree in real-time.

Algorithm 3 takes a collection C of unique sets as input. If the collection has only one set, then it constructs a tree T consisting of a single node with the only set G (Lines 1-3). Otherwise, the algorithm selects the best entity e using the entity selection strategy denoted by $\boldsymbol{\Upsilon}$ (Line 5). It recursively

constructs the subtrees T^+ and T^- for the two sub-collections $C+$ and $C-$ respectively (Lines 6-9). Finally, a tree T, consisting of a root node e and two child subtrees (T^+, T^-), is constructed and returned (Lines 10-11).

There are $n-1$ internal nodes in a full binary decision tree representing a collection of n sets and m entities, and each internal node requires a k-steps lookahead, which costs $O(m^k n)$. Hence the runtime of Algorithm 3 is $O(m^k n^2)$.

Algorithm 3 Tree Construction

Input: collection C of unique sets
Output: a decision tree representation of the input collection
Parameters: entity selection strategy $\boldsymbol{\Upsilon}$

1: **if** $|C| = 1$ **then**
2: **let** G be the only element of C
3: $T \leftarrow$ **Tree** $(G, null, null)$
4: **else**
5: $e \leftarrow \boldsymbol{\Upsilon}(C)$
6: **let** CS^+ be the collection $\{S_i \in C \mid e \in S_i\}$
7: $CS^- \leftarrow C - CS^+$
8: $T^+ \leftarrow$ **Tree-Construction** (CS^+)
9: $T^- \leftarrow$ **Tree-Construction** (CS^-)
10: $T \leftarrow$ **Tree** (e, T^+, T^-)
11: **return** T

Allowing "don't know" answers Sometimes the user is uncertain about the membership of an entity in the target set and may reply "don't know" to the membership question. In such cases, the entity selection strategy can be called again using the same collection of candidate sets but excluding the entities that the user is not sure about. When there is no more entity to choose from the candidate sets, all candidate sets can be returned.

5.6 Optimal Strategy

The lookahead strategies presented in Section 5.4 are efficient in that they construct a tree by looking only a few steps ahead at each node of the tree. The discovered trees may also be close to optimal in many cases, but there is

no guarantee that a discovered tree is optimal or even close to optimal. To guarantee that an optimal tree is constructed, one needs to look at all the steps ahead before selecting an entity as the next question. That can be achieved by setting k to the height of an optimal tree or a greater value in Algorithm 1. However, constructing an optimal tree using Algorithm 3 and Algorithm 1 with optimal k is not efficient and can be improved in two ways. First, while selecting an optimal entity as the root of the tree, the optimal entity choices for the successor nodes are also explored. Thus, it is not necessary to execute the entity selection algorithm for the successor nodes in Algorithm 3. Instead, those already explored choices can be stored as tree nodes and linked together to construct an optimal tree in a bottom-up manner. Second, the parameter k is not necessary since we look at all the steps ahead and can stop if only one candidate set is left. Hence, instead of storing all the k-steps solutions of a collection for memoization, only an optimal solution can be stored, which can speed up the calculation and save some memory space. Next, we present a single algorithm to select optimal entities and to construct an optimal tree representation of a given collection.

Algorithm 4 presents our optimal tree construction (*optimal tree search*) strategy. It takes a collection C of unique sets and an upper limit ul of the cost for the optimal tree to be constructed as inputs. Note that the Tree-Node function takes a collection C, an optimal entity e as the first question for C, an optimal tree representation T^+ of the sub-collection where e is present, an optimal tree representation T^- of the sub-collection where e is absent, and the optimal cost m for the tree representation of C, respectively as inputs. Then, it creates an optimal tree representation of the collection C. If the collection has only one set, then a tree consisting of a single node with only that set is constructed (Lines 7-8). Otherwise, the algorithm sorts the entities based on their partitioning capability from the most even to the least even (Line 10). For each entity e in the sorted order, an optimal tree T^+ for the sub-collection C^+ where e is present and T^- for the sub-collection C^- where e is absent are constructed by recursively calling the algorithm (Lines 15-24). Based on the cost of those subtrees, the cost m for an optimal tree representation of C

with e as the root is calculated (Line 25). The algorithm keeps track of the tree representation with the least cost (Lines 26-28). It applies the pruning similar to Algorithm 1 in Lines 13-14, 19-20, and 23-24 but uses the optimal costs instead of the k-steps lower bounds. Finally, it returns an optimal tree representation of the input collection C if it is possible to construct a tree within the given upper limit ul of the cost (Line 30). Otherwise, an invalid tree (the tree is null or the representing collection is null) is returned (Lines 4 and 11). To speed up the calculations, the algorithm uses memoization by storing and reusing the constructed optimal trees for the different input collections C (Lines 1-6 and 29). Since it may not be possible to construct an optimal tree for a collection within the given upper limit of the cost, the algorithm also stores an invalid tree for a collection until a valid tree is constructed, to keep track of the largest upper limit tried with so far for that collection (Lines 11 and 29). Thus, the algorithm only proceeds to construct a valid tree again for that collection if the given input upper limit is greater than the stored upper limit (Lines 3-4).

For a collection of n unique sets and m distinct entities, the partitioning score (or 1-step lower bound) calculation for all entities requires $O(mn)$ and sorting them requires $O(m \log m)$ runtime. Since the number of unique sub-collections is at most 2^n (the power set of the given collection), the runtime of Algorithm 4 is $O((mn + m \log m) * 2^n)$. The additional space requirement is $O(2^n)$ for memoization. However, they are usually much less in practice due to the pruning.

Once the optimal tree is constructed, the target set can be discovered by traversing the tree and asking questions about the entities in the tree nodes. Algorithm 5 shows how to traverse the constructed tree to find the target set. First, it projects C to a sub-collection CS that includes the initial set I (Lines 1-4). Then, with an optimal tree constructed on CS (Line 6), a question is asked from the user about the presence of the entity assigned to the root node, and one of the child subtrees T^+ and T^- is selected for the next traversal (Lines 8-13). The algorithm iteratively traverses a sequence of nodes starting from the root node of the optimal tree until it finds a leaf node ($|C| = 1$) or

Algorithm 4 Optimal Tree Search

Input: collection C of unique sets and upper limit ul of the cost for the tree to be constructed
Output: optimal tree representation of the input collection

```
1: if C ∈ Cache then
2:     T ← Cache[C]
3:     if ul ≤ T.m then
4:         return null
5:     else if T.C ≠ null then
6:         return T
7: if |C| = 1 then
8:     T ← Tree-Node (C, null, null, null, 0)
9: else
10:    SE ← sort entities according to most even partitioning of C
11:    T ← Tree-Node (null, null, null, null, ul)
12:    for each entity e_i ∈ SE do
13:        if LB_1(C, e_i) ≥ ul then
14:            break
15:        let C+ be the collection {S_j ∈ C | e_i ∈ S_j}
16:        C− ← C − C+
17:        ul+ ← Upper-Limit (ul, |C+|, LB_0(C−), |C−|, |C|)
18:        T+ ← Optimal-Tree-Search (C+, ul+)
19:        if T+ = null or T+.C = null then
20:            continue
21:        ul− ← Upper-Limit (ul, |C−|, T+.m, |C+|, |C|)
22:        T− ← Optimal-Tree-Search (C−, ul−)
23:        if T− = null or T−.C = null then
24:            continue
25:        m ← Cost (|C+|, T+.m, |C−|, T−.m, |C|)
26:        if m < ul then
27:            ul ← m
28:            T ← Tree-Node (C, e_i, T+, T−, m)
29: Cache[C] ← T
30: return T
```

the halt condition $\mathbf{\Gamma}$ (e.g., the user does not want to answer more questions) is met. Finally, it returns the remaining sets that are consistent with the user's answers.

Algorithm 5 Optimal Set Discovery

Inputs: collection C of unique sets and initial set I
Output: sets that are consistent with the user's answers
Parameter: optimal tree construction strategy **OS** and halt condition Γ

1: $CS \leftarrow \varnothing$
2: **for** each set $S_i \in C$ **do**
3:     **if** $I \subseteq S_i$ **then**
4:         $CS \leftarrow CS \cup \{S_i\}$
5: **if** $|CS| > 1$ **then**
6:     $T \leftarrow \mathbf{OS}(CS)$
7:     **while** $|T.C| > 1$ and Γ is $false$ **do**
8:         $e \leftarrow T.e$
9:         $\alpha \leftarrow$ **query** the user about the presence of e in the target set
10:         **if** α is $true$ **then**
11:             $T \leftarrow T.T^+$
12:         **else**
13:             $T \leftarrow T.T^-$
14:     $CS \leftarrow T.C$
15: **return** CS

Chapter 6

Experiments

This section reports an experimental evaluation of our algorithms and pruning strategy on both real and synthetic data and under different parameter settings.

6.1 Evaluation Setup

As our evaluation measures, we study (a) the effectiveness of our algorithms in finding a "good" solution for the problem of set discovery, (b) the effectiveness of our pruning strategy in reducing the size of the search space, (c) the efficiency of our algorithms in terms of the running time, and (d) the scalability of our algorithms with both the number and the size of sets. Our results are compared to the relevant algorithms in the literature (when applicable).

The effectiveness of our set discovery is measured in terms of the *number of questions* to be answered by a user looking for a target set. Without knowing much about the target set of a user, we assume all sets that contain an initially provided set are equally likely. With this, the effectiveness may be defined in terms of the *average number of questions* or the *maximum number of questions* to be answered by a user. These quantities also represent the average depth of the leafs (AD) and the height (H) of a decision tree that is constructed.

The efficiency of an entity selection algorithm is measured in terms of the *tree construction time*, which is the time needed to construct a decision tree using the selection strategy. It can be noted that the tree construction time is different from the time spent when searching for a specific set (*discovery time*).

For the former, Algorithm 3 constructs a whole tree with all sets placed at the leaves and the internal nodes giving the paths to all sets at the leaves, whereas for the latter, Algorithm 2 only constructs a path from the root to the target set. The latter is much less if the wait time for user responses is excluded.

The algorithms being evaluated include entity selection using k-LP, k-LPLE, and k-LPLVE strategies. For a comparison with entity selection strategies from the literature, our evaluation also includes *information gain* (InfoGain) [27] and *gain-k* [12]. Our reported result for *information gain* holds for *indistinguishable pairs* [5] and *1-step lookahead* since they all select the same entity, as shown earlier (Lemma 5.2.1).

Our algorithms have been implemented in Python 3. All our experiments were run on a 64-bit Windows machine with Intel(R) Core i5-9300H @2.40 GHz processor and 8 GB RAM.

6.2 Datasets and Queries

We conduct our experiments on two datasets for set discovery, including *web tables*, which consists of a collection of entity sets extracted from the columns of various web tables, and *synthetic data*, where large collections of sets are generated following some distributions. The former evaluates our algorithms on a real dataset, whereas the latter assesses the scalability of our strategies under different collection sizes and parameter settings. We also evaluate our algorithms on the task of query discovery, based on the TPC-H benchmark and a *baseball* database.

Web tables This is a collection of sets extracted from the textual columns of web tables. The entities in each column are considered a set after removing duplicates. The collection includes 1,407,178 sets containing in total 6,312,409 distinct entities. The sets in the collection are all unique, meaning all duplicate sets are removed. For one experiment, 14,491 unique sub-collections were selected using all possible sets of size 2 as the initial seed sets and constraining each sub-collection to include at least 100 sets. The choice of size-2 initial sets was based on the observation that at least two entities from a semantic class

are required to unambiguously represent a semantic class. As an example, Liverpool may represent both a "City" and a "Football Club" whereas Liverpool and Arsenal together do not represent the "Cities" semantic class. The number of sets in the selected sub-collections was in the range of $[100, 11219]$ with an average of 390 and a standard deviation of 478, and the number of distinct entities was in the range of $[15, 15186]$ with an average of 3,112 and a standard deviation of 2,379. For another experiment, three general but different concept sets are selected as the reference sets: "States in India", "Cities in Canada", and "Countries in Asia". Six query sets (referred to as Q1 to Q6) are constructed, each from one of our three concept sets, and each had an initial set size of 2 or 3. The choices of the concept sets and the initial sets are driven by their variations in the number of candidate sets, the number of entities, and the entity distribution to ensure different levels of query complexity and at the same time, the attainability of an optimal solution within a considerable running time. Table 6.1 presents our selected queries with some statistics about each query, including the number of candidate sets that match the initial set and the total number of distinct entities in those candidate sets.

Query set	Initial set	Number of candidate sets	Number of distinct entities
Q1	{Kerala, Goa}	136	3469
Q2	{Maharashtra, Tamil Nadu}	247	3740
Q3	{Ottawa, Edmonton, Montreal}	200	2456
Q4	{Vancouver, Winnipeg, Edmonton}	228	2566
Q5	{China, India, Myanmar}	249	934
Q6	{Indonesia, Iran, Bhutan}	319	3805

Table 6.1: Query sets for the web tables dataset

Synthetic data To study the performance of our entity selection strategies under different data distributions as well as the scalability with the number of entities and sets in the collection, we generated a few synthetic set collections. The set generation follows a copy-add preferential mechanism where some elements are copied from an existing set and the rest of the elements are added from a universe of elements. Similar copying models are used in other

domains (e.g., the dynamics of the web graph [19], the copying and publishing relationships between data sources [11], etc.). Each set has two parameters: a set size s, chosen randomly from a range of values (e.g., $[50, 100]$), and an overlap ratio $\alpha \in [0, 1)$. For each set, we choose a size s from the range of possible sizes randomly and an overlap ratio α. Then $\alpha * s$ elements are copied from a previously generated set and $(1-\alpha) * s$ elements are added from the entity universe. If a previously generated set does not exist or does not have enough elements, then additional elements are selected from the entity universe to bump up the set size to s. We generated 19 synthetic collections by varying the overlap ratio α, the number of sets n, and the range of set sizes d. Table 6.2 gives some information about these collections including the number of distinct entities in each collection. For this dataset, no entities were selected as query entities (i.e., the user-provided initial set is considered empty), and all the sets in each collection are considered as possible target sets.

Overlap ratio α	Number of distinct entities
0.99	23k
0.95	36k
0.90	59k
0.85	83k
0.80	108k
0.75	132k
0.70	156k
0.65	178k

(a) Varying the overlap ratio α with n fixed to $10k$ and d fixed to $[50, 60]$

Number of sets n	Number of distinct entities
10k	59k
20k	125k
40k	216k
80k	385k
160k	622k

(b) Varying the number of sets n with α fixed to 0.9 and d fixed to $[50, 60]$

Set size range d	Number of distinct entities
50-100	119k
100-150	150k
150-200	180k
200-250	214k
250-300	249k
300-350	283k

(c) Varying the set size range d with α fixed to 0.9 and n fixed to $10k$

Table 6.2: Synthetic data by varying (a) overlap ratio α, (b) number of sets n, and (c) set size range d

TPC-H benchmark We generated an instance of the TPC-H benchmark dataset [36], considering the benchmark queries as our reference queries for the experiments. We adapted the query reverse engineering approach of Zhang et

al. [45] to compute join queries that generate a superset of the example tuples. Their approach discovers only project-join queries without any aggregates, arithmetic expressions, group by statements, selection conditions, etc. Therefore, we removed those operators from the benchmark queries to make the queries suitable for the query reverse engineering tool. For each query, we took a few example tuples from its output and ran the reverse engineering tool to generate queries that output a superset of the examples. To increase the size of our query set, we further generated additional queries by varying the selection conditions in our base queries, by either including or excluding each condition. Since the selection conditions in each query were chosen from the power set of the selection conditions of the reference TPC-H query, this also produced overlaps between different query results. This resulted in 224 unique queries for our query collection. We ran all those queries, generating a collection with 224 sets and 570,558 distinct tuples in those sets. Finally, we conducted each experiment by selecting a few output tuples, as the initial set, from each benchmark query and constructing the tree representations of the candidate queries selected by the initial set of tuples, according to our different strategies. Here we report the result only for five benchmark queries with the highest number of candidate sets where a query discovery is more meaningful. Table 6.3 shows the benchmark queries chosen, the number of candidate queries that output a superset of the initial set, and the total number of unique tuples in each case.

Query set	TPC-H query no.	Number of candidate queries	Number of unique tuples
Q1	6	16	5960
Q2	10	8	6003
Q3	12	32	5989
Q4	19	63	174649
Q5	22	65	150

Table 6.3: Query sets for the TPC-H benchmark

Baseball database The baseball database [20] is a complex, multi-relation

database that contains batting, pitching, and fielding statistics plus standings, team stats, player information, and more for Major League Baseball (MLB) covering the years between 1871 and 2020. Our experiment is based on the *People* table which contains information about name, birth, death, height, weight, batting and throwing hand, etc., of 20,185 baseball players. For our experiment, we considered only CNF (conjunctive normal form) queries with conditions on columns *birthCountry*, *birthState*, *birthCity*, *birthYear*, *birthMonth*, *birthDay*, *height*, *weight*, *bats*, and *throws* of the *People* table. At first, we constructed 10 target queries that could be interesting to a user. Table 6.4 describes the target queries and the number of tuples in their outputs. Then, for each target query, we randomly selected 2 output tuples as the example tuples and generated candidate CNF queries that contain the example tuples in their output. The candidate queries are generated using the following simple steps:

1. The columns are grouped into categorical and numerical columns. In our experiment, we treated *birthCountry*, *birthState*, *birthCity*, *birthMonth*, *birthDay*, *bats*, and *throws* as categorical columns whereas *birthYear*, *height*, and *weight* as numerical columns.

2. A few reference values are defined for each numerical column. For examples, *height*: {60, 65, 70, 75, 80}, *weight*: {120, 140, 160, 180, 200, 220, 240, 260, 280, 300}, and *birthYear*: {1850, 1870, 1890, 1910, 1930, 1950, 1970, 1990}.

3. A selection condition on each categorical column is constructed as the disjunctions of the unique values of the example tuples for that column. For example, if the birth city of an example player is Chicago and that of another player is Seattle, then the selection condition for *birthCity* is $birthCity = \text{“}Chicago\text{”} \vee birthCity = \text{“}Seattle\text{”}$. On the other hand, if the birth city of all example players is Chicago, then the selection condition is $birthCity = \text{“}Chicago\text{”}$.

4. A few selection conditions on each numerical column are constructed

using the possible intervals of the reference values that contain the values of all example tuples. For example, if the height of an example player is 62 and that of another player is 73, then the possible selection conditions on *height* are $height > 60 \wedge height < 75$, $height > 60 \wedge height < 80$, $height > 60$, $height < 75$, and $height < 80$.

5. Each selection condition on a column yields a candidate query. Furthermore, the conjunctions of any two selection conditions on two different columns provide additional candidate queries. For example, $\sigma_{birthCity=\text{“}LosAngeles\text{”}}(People)$ is a query with selection condition on a single column, whereas $\sigma_{birthCity=\text{“}LosAngeles\text{”} \wedge height>70 \wedge height<80}(People)$ is a query with selection conditions on two columns. Similarly, candidate queries with selection conditions on more columns can be generated. In our experiment, we considered queries with selection conditions on up to two columns.

Target query	Query description	Number of output tuples
T1	$\sigma_{birthCountry=\text{“}USA\text{”} \wedge birthYear>1990}(People)$	892
T2	$\sigma_{birthCity=\text{“}LosAngeles\text{”} \wedge height>70 \wedge height<80}(People)$	201
T3	$\sigma_{birthState=\text{“}TX\text{”} \wedge bats=\text{“}L\text{”}}(People)$	267
T4	$\sigma_{bats=\text{“}R\text{”} \wedge throws=\text{“}L\text{”}}(People)$	599
T5	$\sigma_{bats=\text{“}L\text{”} \wedge throws=\text{“}R\text{”}}(People)$	2179
T6	$\sigma_{birthCountry=\text{“}USA\text{”} \wedge bats=\text{“}B\text{”}}(People)$	939
T7	$\sigma_{birthCountry=\text{“}USA\text{”} \wedge height<70}(People)$	1901
T8	$\sigma_{birthMonth=12 \wedge birthDay=25}(People)$	65
T9	$\sigma_{height>75 \wedge weight>260}(People)$	49
T10	$\sigma_{height<65 \wedge weight<160}(People)$	26

Table 6.4: Target queries for the baseball database

Once the candidate queries were generated, we applied our set discovery strategy to discover the target query. The user answers about the membership of the presented tuples were simulated by verifying them against the output of the target query. Table 6.5 provides information about the selected example tuples for each target query, the number of generated candidate queries from

the example tuples, and the average number of tuples in the output of those candidate queries.

Target query	Player ids of example tuples	Number of candidate queries	Average number of output tuples
T1	baragca01, phillev01	776	9404.24
T2	ryanbr01, edwarda01	987	11254.35
T3	jonesru01, gurkaja01	1003	9629.40
T4	masaoon01, cyrer01	951	10079.17
T5	ellioal01, drumrke01	940	10612.07
T6	dashnle01, craigro02	916	10957.30
T7	riceha01, bentora01	1148	9202.28
T8	brownll01, ellerfr01	1339	9772.70
T9	evansde01, fulchje01	600	7187.00
T10	emmerbo01, gearidi01	1189	7795.78

Table 6.5: Information about selected example tuples and generated candidate queries on baseball database

6.3 Evaluation Results on Set Discovery

Choosing the parameters k and q To set the parameters k and q for our algorithms, we did run some experiments on our web tables dataset. Figure 6.1 shows that the runtime of k-LP increases by one to two orders of magnitude when the number of lookahead steps k is increased from 2 to 3. At the same time, the average number of questions usually becomes less with higher k. To balance the runtime with the quality of the trees that are constructed, we set $k = 2$ for our experiments with the k-LP strategy. The runtime may also be kept low, while increasing k, using the k-LPLE strategy, which limits the number of entities in each step. For our experiments with k-LPLE and k-LPLVE strategies, we set $k = 3$ and experiment with different values (up to 50) of the number of entities q. The average number of questions that are required remains almost the same when the value of q exceeds 10, but the runtime increases significantly. Therefore, we set $q = 10$ for the k-LPLE and k-LPLVE strategies. The average numbers of questions for larger values of q are almost the same hence are not reported here.

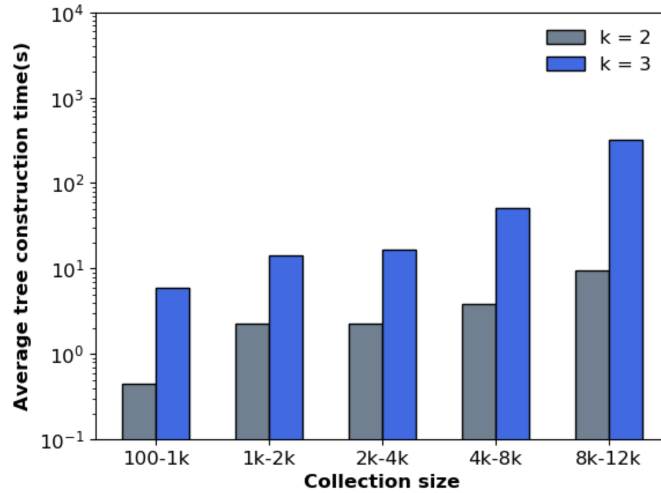


Figure 6.1: Tree construction time (seconds) for k-LP varying k on web tables dataset

Comparison to strategies in the literature Our baselines for comparison include *information gain* [29], *indistinguishable pairs* [5], and *gain-k* [12]. However, *information gain*, *indistinguishable pairs*, *gain-k* with $k = 1$, and our k-LP with $k = 1$ can be considered as 1-step lookahead strategies, and they all select the same entity, as discussed in Section 5.2 (and also verified in our experiments). Moreover, *gain-k* with $k = 2$ performs similarly to our k-LP with $k = 2$. Therefore, we show our comparison results with *information gain* (InfoGain) only and not other baseline strategies with identical results. Figure 6.2 shows our improvements over InfoGain in the average number of questions with the cost metric AD and the maximum number of questions with the cost metric H. The results are based on set discovery in our *web tables* dataset with 14,491 sub-collections. In 53% to 67% of sub-collections with cost metric AD and 7% to 16% of sub-collections with cost metric H, our strategies find a better tree with less number of questions, whereas InfoGain only finds a better tree in 5% to 17% of sub-collections with cost metric AD and in only 1% of sub-collections with cost metric H. In the rest of the sub-collections (e.g. 30% for k-LP with k=2 and cost metric AD), our strategies and InfoGain result in the same trees. The mean improvement in the maximum number of questions (H) is close to one, whereas the mean improvement for the average number of

questions is less due to the facts that the improvement is averaged over all sets in each sub-collection and that the average number of questions for InfoGain is already very close to the optimal (the average difference in the average number of questions with optimal solution for InfoGain is only about 0.048) and there is less room for improvement. However, a little improvement in the number of questions can also be significant for some real-life scenarios. For example, if the questions are medical tests required to identify a disease, then a small reduction even in the average number of tests could save the patients a large amount of money and time to complete the tests. Among our strategies, the performance of k-LPLVE and k-LPLE is close to each other but much better than k-LP in terms of required questions, since a larger value of k is used. Moreover, their running times are also close to k-LP because of the use of a limited number of entities ($q = 10$).

Strategy	Less questions percentage	More questions percentage	Improvement mean(std)
k-LP (k=2)	53	17	0.014 (0.029)
k-LPLE (k=3, q=10)	67	5	0.025 (0.032)
k-LPLVE (k=3, q=10)	64	7	0.023 (0.031)

(a) Cost Metric AD

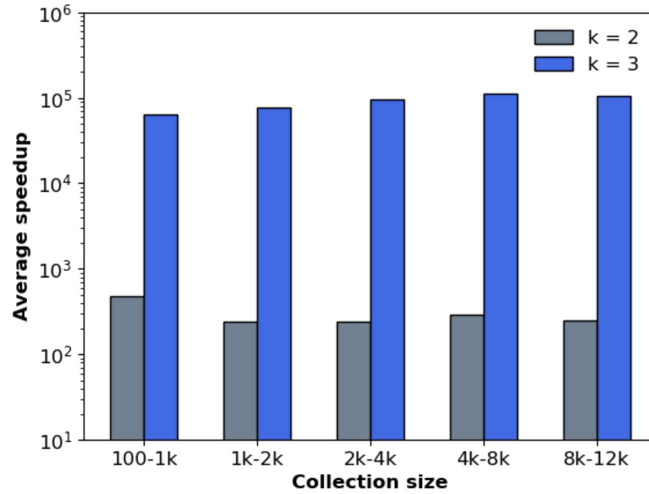
Strategy	Less questions percentage	More questions percentage	Improvement mean(std)
k-LP (k=2)	7	1	0.749 (0.796)
k-LPLE (k=3, q=10)	14	1	0.884 (0.647)
k-LPLVE (k=3, q=10)	16	1	0.901 (0.640)

(b) Cost Metric H

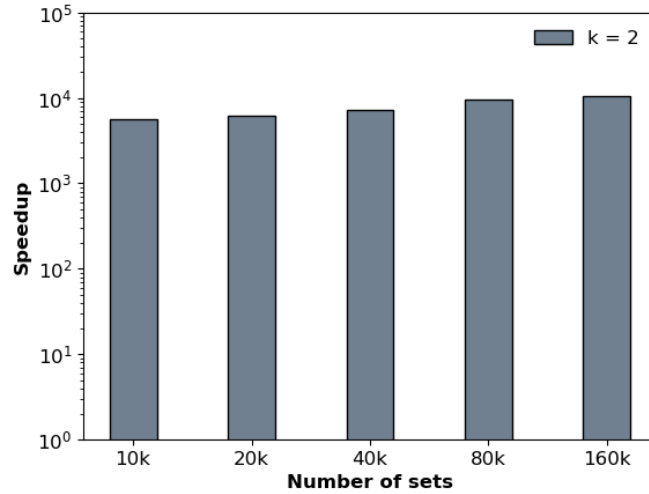
Figure 6.2: Comparison of our strategies with InfoGain strategy on web tables dataset

Effectiveness of our pruning The pruning proposed in this thesis makes a huge difference in the tree construction time of all our strategies. Figure 6.3a shows the speedup on the web tables dataset, and Figure 6.3b shows the same on the synthetic datasets. The average speedup in runtime on the web tables dataset is in the range of two to three orders of magnitude for $k = 2$ and up to five orders of magnitude when $k = 3$. Since the runtime of gain-k

increases polynomially with the number of entities and exponentially with k, the speedup is more for larger values of k and on datasets with a large number of entities and sets. This can be seen in Figure 6.3a for the web tables dataset where k is varied from 2 to 3 and in Figure 6.3b for the synthetic dataset with a fixed k and varying the number of sets.



(a) k-LP vs gain-k on web tables data



(b) k-LP vs gain-k on synthetic data

Figure 6.3: Speedup of our strategies because of pruning

Comparison with our optimal strategy The results of the experiments

are shown in Figure 6.4 for the cost metric AD and in Figure 6.5 for the cost metric H. For both metrics, our lookahead strategies find solutions that are very close to an optimal solution returned by the *optimal tree search* strategy but within significantly less tree construction time. However, for the query sets Q1 and Q6 with the cost metric H, *optimal tree search* requires less tree construction time than the other strategies. Since, the first entity that *optimal tree search* considers in each node during its bottom-up tree construction is optimal, all other entities are pruned. Although similar pruning is applied by the other strategies, they still need to look a few steps ahead because of their top-down approach to tree construction. Moreover, the tree construction time of *optimal tree search* is much less than the time required by a brute force algorithm that does not find an optimal solution within the cut-off time of twelve hours for the experiments.

Query set	optimal tree search	k-LP ($k = 2$)	k-LPLE ($k = 3$, $q = 10$)	k-LPLVE ($k = 3$, $q = 10$)
Q1	7.15	7.18	7.18	7.17
Q2	8.02	8.06	8.04	8.04
Q3	7.74	7.76	7.74	7.74
Q4	7.91	7.93	7.91	7.91
Q5	7.97	7.98	7.97	7.98
Q6	8.39	8.39	8.39	8.39

(a) Average number of questions

Query set	optimal tree search	k-LP ($k = 2$)	k-LPLE ($k = 3$, $q = 10$)	k-LPLVE ($k = 3$, $q = 10$)
Q1	0.727	0.087	0.169	0.102
Q2	77.953	0.081	0.252	0.159
Q3	96.544	0.209	0.607	0.222
Q4	205.738	0.193	0.491	0.209
Q5	2.536	0.137	0.281	0.249
Q6	3646.862	0.455	2.179	0.808

(b) Tree construction time (seconds)

Figure 6.4: Comparison of our lookahead strategies with our optimal tree search strategy on web tables dataset (cost metric AD)

Performance varying the overlap between sets One factor that affects the performance of a set discovery is the amount of overlap between sets.

Query set	optimal tree search	k-LP ($k = 2$)	k-LPLE ($k = 3$, $q = 10$)	k-LPLVE ($k = 3$, $q = 10$)
Q1	8	8	8	8
Q2	9	10	10	10
Q3	10	11	12	11
Q4	10	11	11	11
Q5	8	10	10	9
Q6	9	9	9	9

(a) Maximum number of questions

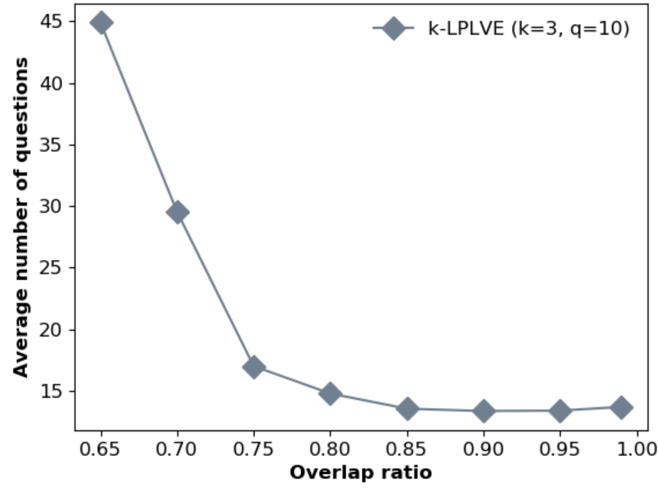
Query set	optimal tree search	k-LP ($k = 2$)	k-LPLE ($k = 3$, $q = 10$)	k-LPLVE ($k = 3$, $q = 10$)
Q1	0.027	0.040	0.076	0.073
Q2	10.568	0.060	0.175	0.105
Q3	1236.189	0.064	0.163	0.119
Q4	253.299	0.077	0.226	0.135
Q5	0.148	0.081	0.136	0.144
Q6	0.163	0.177	0.291	0.297

(b) Tree construction time (seconds)

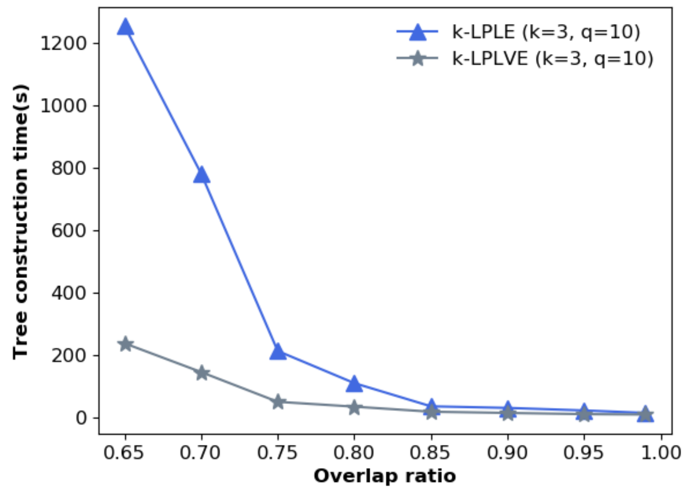
Figure 6.5: Comparison of our lookahead strategies with our optimal tree search strategy on web tables dataset (cost metric H)

Consider an extreme case where there is no overlap between sets. With n sets, one needs to ask roughly $n/2$ questions on average ($n-1$ questions in the worst case) to find a target set. As the overlap between sets increases, there is more chance to filter more than one set with each question. To better understand this relationship between the overlap and the search performance, we varied the overlap ratio as in Table 6.2a for our synthetic dataset and measured the number of questions that were needed to discover each set. Figure 6.6 shows the average number of questions that were needed as the overlap ratio varied from 0.65 to 0.99. As the overlap ratio increases, both the average number of questions and the tree construction time decrease. When the overlap ratio becomes less than 0.90, the average number of questions starts showing an upward trend. This upward trend is expected to continue to the point where one needs to ask roughly $n/2$ questions on average ($n - 1$ questions in the worst case) to find a target set. This happens, for example, when all sets have the same elements except at least one more element that distinguishes each

set from the rest.



(a) Average number of questions



(b) Tree construction time (seconds)

Figure 6.6: Effects of set overlaps on average number of questions and tree construction time

One question is if some notion of overlap can be calculated for our other datasets and if that can provide some hint on the structure of the sets and maybe the expected number of questions when searching for a set. The overlap ratio, as defined above for data generation, cannot be easily computed for real datasets, but similar measures of overlap may be used. For example, we may

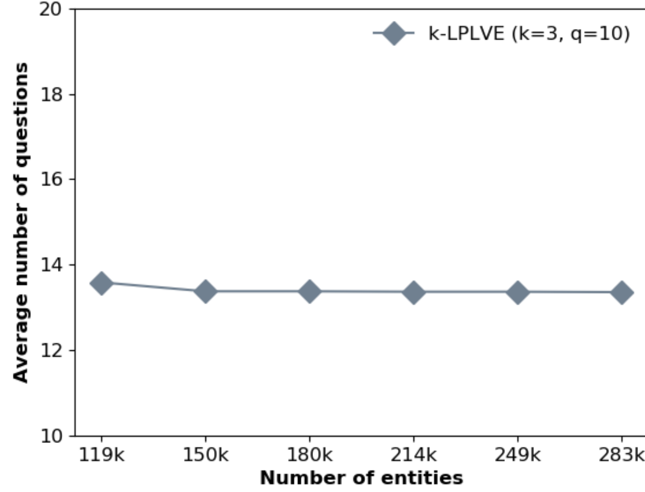
find for each set the largest fraction of elements that appear in a single other set or the union of all other sets. We refer to the former as max-overlap and the latter as union-overlap. Table 6.6 shows these quantities (as well as minimum overlap) averaged over all sets in a collection for our three of the datasets. On our synthetic dataset, max-overlap and union-overlap are very close to the overlap ratio, but these overlaps are not sufficient statistics and any prediction of performance should be treated with a grain of salt.

Dataset	Query set	Min-overlap	Max-overlap	Union-overlap
Web tables	Q1	0.09	0.91	0.94
	Q2	0.13	0.95	0.97
	Q3	0.11	0.74	0.92
	Q4	0.11	0.74	0.94
	Q5	0.08	0.96	0.98
	Q6	0.13	0.96	0.99
Synthetic	$\alpha = 0.99$	0.24	0.97	0.98
	$\alpha = 0.95$	0.12	0.95	0.97
	$\alpha = 0.90$	0.01	0.91	0.94
	$\alpha = 0.85$	0	0.86	0.92
	$\alpha = 0.80$	0	0.81	0.90
	$\alpha = 0.75$	0	0.76	0.87
TPC-H	Q1	0.12	0.99	1.0
	Q2	0.19	0.97	1.0
	Q3	0.11	0.99	1.0
	Q4	012	0.98	0.93
	Q5	0.40	0.99	0.99

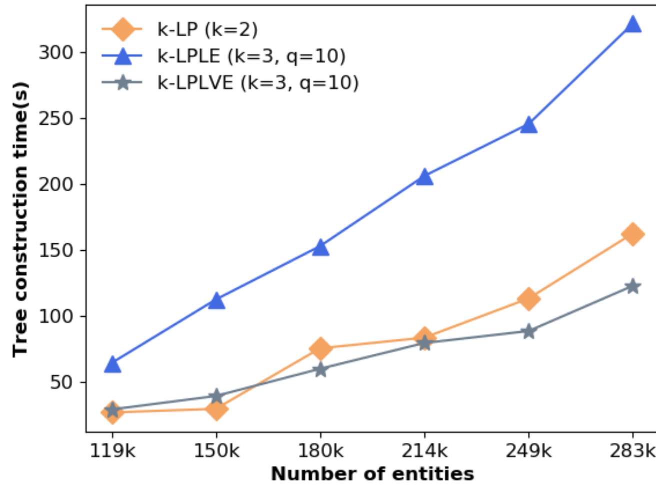
Table 6.6: Overlap ratios/scores in our datasets

Scalability with the number of entities and the collection size To evaluate the scalability of our algorithms on larger datasets, we conducted some experiments using our synthetic data. In one experiment, we varied the number of distinct entities in a collection, while keeping the number of sets and the overlap ratio fixed at 10k and 0.9 respectively. The number of distinct entities changes (as shown in Table 6.2c) with the set size varied. As can be seen in Figure 6.7, the average number of questions is not affected much, but the tree construction time increases because of the larger number of candidate entities that are considered during the lower bounds calculation. The increase

in running time is linear for k-LPLE and k-LPLVE, and the running time of k-LP increases quadratically with $k = 2$.



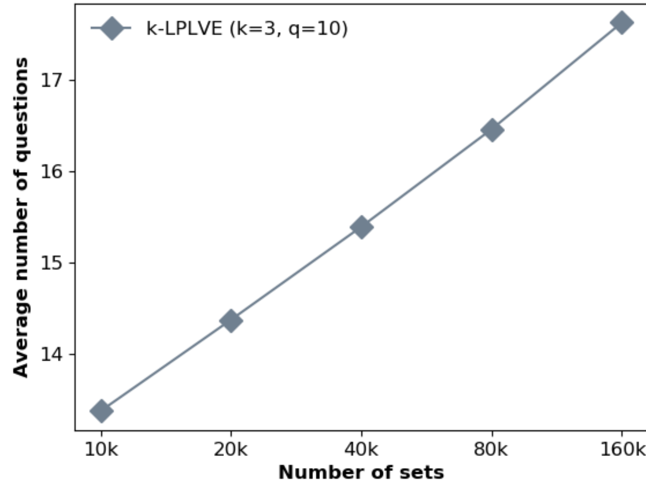
(a) Average number of questions



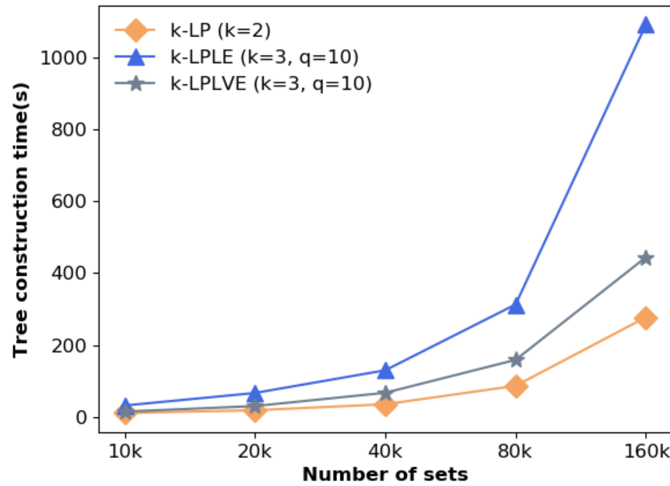
(b) Tree construction time (seconds)

Figure 6.7: Effects of increasing the number of distinct entities in a collection on average number of questions and tree construction time

In another experiment, we varied the number of sets in the collection while keeping the set size in $[50, 60]$ and the overlap ratio fixed at 0.9. The number of distinct entities m increases as well (as shown in Table 6.2b), when we increase the number of sets n. As shown in Figure 6.8, with each doubling



(a) Average number of questions



(b) Tree construction time (seconds)

Figure 6.8: Effects of increasing the number of sets on average number of questions and tree construction time

of the input size, the average number of questions increases roughly by 1. The tree construction time is expected to increase linearly with the number of sets if the number of distinct entities is fixed. In our experiment, the tree construction time looks a bit far from linear (and more quadratic) because of the increase in m as n increases.

6.4 Evaluation Results on Query Discovery

Experiment on TPC-H benchmark In our query discovery experiment on the TPC-H benchmark, all our strategies find an optimal solution, which is the lower bound for discovering each query. This turns out to be an easier problem for set discovery. Since each query was chosen from the power set of the selection conditions of a reference TPC-H query, there is a clean containment relationship between many queries in the collection, which turns out to be ideal for set discovery using 1-step strategies. Hence, we do not report the number of questions but only the tree construction time in Figure 6.9. Since 1-step strategies (e.g., InfoGain) find an optimal solution, the first tuple, considered by our k-step strategies (i.e., k-LP, k-LPLE, and k-LPLVE) when constructing each tree node, is also an optimal choice and all other tuples are easily pruned. Thus, the tree construction time for those strategies is very close to each other. Moreover, the little differences are only because of the use of different values for k. However, the tree construction time for gain-k is quite high compared to all other strategies, since it does not apply any pruning, and the runtime increases as a quadratic function of the number of distinct entities (for $k = 2$). It is clear that, for the cases where a 1-step strategy is capable of finding an optimal solution, the use of our other strategies with higher values of k also does not cost much in terms of runtime because of our pruning strategy.

Experiment on baseball database Figure 6.10 shows both the number of questions and the system processing time (query discovery time) required to discover the target queries on the baseball database for the baseline InfoGain strategy and our lookahead strategies. It can be seen that the number of questions for k-LP, k-LPLE, and k-LPLVE is less than or equal to InfoGain (except T10 for k-LP). Since none of the strategies are optimal, our strategies may sometimes require more questions than InfoGain, but that probability is very low as shown in Figure 6.2. Moreover, the query discovery time of our strategies is considerably less, provided that the candidate queries have a large number of tuples on average (7000 to 12000) in their outputs, as shown in Table 6.5, which could result in higher processing time. Finally, an important

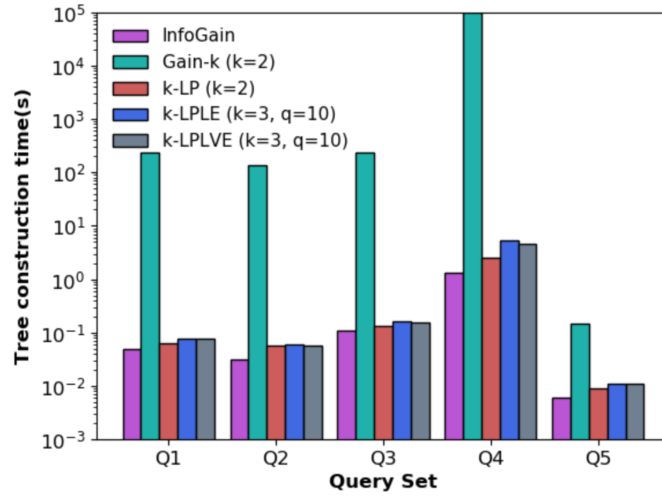


Figure 6.9: Tree construction time (seconds) for our query discovery experiment on TPC-H benchmark

observation can be made about our query discovery strategy. The user is required to confirm the membership of only a few tuples (9 to 11) to find the target query among a large number of candidate queries (600 to 1200) which is more convenient than listing all the possible output tuples (close to 2000 for some of our target queries) of a target query.

Target query	InfoGain	k-LP ($k=2$)	k-LPLE ($k=3$, $q=10$)	k-LPLVE ($k=3$, $q=10$)
T1	10	10	10	10
T2	10	9	10	10
T3	10	9	10	10
T4	10	9	10	10
T5	10	10	9	9
T6	10	10	9	9
T7	11	11	10	10
T8	11	11	10	10
T9	10	9	9	9
T10	10	11	10	10

(a) Number of questions

Target query	InfoGain	k-LP ($k=2$)	k-LPLE ($k=3$, $q=10$)	k-LPLVE ($k=3$, $q=10$)
T1	1.798	163.097	11.662	7.999
T2	3.234	17.880	37.867	26.060
T3	2.343	12.847	27.993	17.574
T4	3.418	21.151	34.077	27.173
T5	2.921	31.499	31.589	19.453
T6	2.796	20.548	20.944	15.894
T7	2.711	19.602	19.813	13.419
T8	3.687	19.124	23.314	18.690
T9	0.906	10.747	10.395	4.806
T10	2.187	7.108	16.257	17.685

(b) Query discovery time (seconds)

Figure 6.10: Number of questions and query discovery time to find the target queries on baseball database

Chapter 7

Conclusion and Future Work

In this chapter, we present some concluding remarks about our work and discuss a few possible future directions.

7.1 Conclusion

In this thesis, we have studied the problem of set discovery using an interactive approach where example entities from candidate sets are presented to the user, and the search is narrowed down based on the feedback about the presence of entities in the target set. We have represented the collection of candidate sets as full binary decision trees and formulated the problem as decision tree optimization where the height of the tree denotes the maximum number of questions (worst-case), and the average depth of the leaf nodes denotes the average number of questions (average-case) needed to discover a set from the collection. We have established some lower bounds on the number of questions in average-case and worst-case, which are easy to compute but effective in pruning the search space. Based on the lower bounds, we have developed a pruning strategy that allows certain choices of entities for decision tree nodes to be safely rejected. Finally, we have proposed several effective and efficient k-steps lookahead algorithms to construct a tree that results in a near-optimal number of questions for set discovery.

In an extensive experimental evaluation, we have evaluated the performance and scalability of our algorithms on set discovery over the *web tables* dataset and synthetic data and on query discovery over the TPC-H benchmark

and the *baseball* database. We show that our algorithms are more effective than the greedy approaches in the literature and that they find a near-optimal solution. Moreover, our pruning strategy makes our k-steps lookahead algorithms 2-5 orders of magnitude faster.

7.2 Future Work

Our work can be extended or improved in a few directions. One direction is to study the distribution of entities in a collection. Better understanding the distribution may provide some insight to develop other strategies. Another direction is to study the scenarios where the sets to be discovered are not equally likely. The User may prefer some sets over others. Extending our algorithms to the cases where the sets are noisy or have errors is another direction.

Moreover, our established lower bounds on the height and the average depth of a binary tree may be generalized for all types of decision trees, then similar pruning may be applied to improve the existing decision tree learning algorithms. Finally, our approach to set discovery may provide some insight into the problem of set expansion and its expected output.

References

[1] A. Abouzied, D. Angluin, C. Papadimitriou, J. Hellerstein, and A. Silberschatz, "Learning and verifying quantified boolean queries by example," *Proceedings of the PODS Conference*, Apr. 2013. DOI: 10.1145/2463664.2465220. 17

[2] M. Adler and B. Heeringa, "Approximating optimal binary decision trees," in *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, A. Goel, K. Jansen, J. D. P. Rolim, and R. Rubinfeld, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–9, ISBN: 978-3-540-85363-3. 18, 29

[3] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*, 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1974, ISBN: 0201000296. 6

[4] M. Arenas, G. I. Diaz, and E. V. Kostylev, "Reverse engineering sparql queries," in *Proceedings of the WWW Conference*, 2016, pp. 239–249. 15

[5] S. Basu Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania, "Minimum-effort driven dynamic faceted search in structured databases," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, ser. CIKM '08, Napa Valley, California, USA: Association for Computing Machinery, 2008, pp. 13–22, ISBN: 9781595939913. DOI: 10.1145/1458082.1458088. [Online]. Available: https://doi.org/10.1145/1458082.1458088. 18, 30, 45, 52

[6] A. Bonifati, R. Ciucanu, and S. Stawork, "Interactive inference of join queries," in *In EDBT*, 2014, pp. 451–462. 17, 31, 35

[7] A. Bonifati, R. Ciucanu, and S. Staworko, "Learning join queries from user examples," *ACM Trans. Database Syst.*, vol. 40, no. 4, Jan. 2016, ISSN: 0362-5915. DOI: 10.1145/2818637. [Online]. Available: https://doi.org/10.1145/2818637. 17

[8] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern Recogn.*, vol. 30, no. 7, pp. 1145–1159, Jul. 1997, ISSN: 0031-3203. DOI: 10.1016/S0031-3203(96)00142-2. [Online]. Available: https://doi.org/10.1016/S0031-3203(96)00142-2. 14

[9] L. Breiman, J. Friedman, R. Olshen, and C. J. Stone, "Classification and regression trees," 1983. 9, 11, 12, 30

[10] K. Dimitriadou, O. Papaemmanouil, and Y. Diao, "Explore-by-example: An automatic query steering framework for interactive data exploration," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 517–528, ISBN: 9781450323765. DOI: `10.1145/2588555.2610523`. [Online]. Available: `https://doi.org/10.1145/2588555.2610523`. 17

[11] X. L. Dong, L. Berti-Equille, and D. Srivastava, "Truth discovery and copying detection in a dynamic world," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 562–573, 2009. 47

[12] S. Esmeir and S. Markovitch, "Lookahead-based algorithms for anytime induction of decision trees," in *Proceedings of the Twenty-First International Conference on Machine Learning*, ser. ICML '04, Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 33, ISBN: 1581138385. DOI: `10.1145/1015330.1015373`. [Online]. Available: `https://doi.org/10.1145/1015330.1015373`. 12, 13, 19, 32, 35, 45, 52

[13] E. Frank and I. H. Witten, "Reduced-error pruning with significance tests," in *Available: http://libra.msra.cn/paperdetail.aspx?id=305368*, 1998, p. 98. 14

[14] Y. He and D. Xin, "Seisa: Set expansion by iterative similarity aggregation," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11, Hyderabad, India: Association for Computing Machinery, 2011, pp. 427–436, ISBN: 9781450306324. DOI: `10.1145/1963405.1963467`. [Online]. Available: `https://doi.org/10.1145/1963405.1963467`. 20

[15] J. Huang, Y. Xie, Y. Meng, J. Shen, Y. Zhang, and J. Han, "Guiding corpus-based set expansion by auxiliary sets generation and co-expansion," in *Proceedings of The Web Conference 2020*, ser. WWW '20, Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 2188–2198, ISBN: 9781450370233. DOI: `10.1145/3366423.3380284`. [Online]. Available: `https://doi.org/10.1145/3366423.3380284`. 21

[16] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is np-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976, ISSN: 0020-0190. DOI: `https://doi.org/10.1016/0020-0190(76)90095-8`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/0020019076900958`. 18, 29

[17] P. Jindal and D. Roth, "Learning from negative examples in set-expansion," in *2011 IEEE 11th International Conference on Data Mining*, 2011, pp. 1110–1115. 21

[18] D. V. Kalashnikov, L. V. Lakshmanan, and D. Srivastava, "Fastqre: Fast query reverse engineering," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 337–350. 2, 15

[19] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal, "Stochastic models for the web graph," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, IEEE, 2000, pp. 57–65. 47

[20] S. Lahman, *Baseball database*, 2020. [Online]. Available: `http://www.seanlahman.com/baseball-archive/statistics/`. 48

[21] H. Li, C.-Y. Chan, and D. Maier, "Query from examples: An iterative, data-driven approach to query construction," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2158–2169, Sep. 2015, ISSN: 2150-8097. DOI: `10.14778/2831360.2831369`. [Online]. Available: `https://doi.org/10.14778/2831360.2831369`. 18

[22] S. Lomax and S. Vadera, "A survey of cost-sensitive decision tree induction algorithms," *ACM Comput. Surv.*, vol. 45, no. 2, Mar. 2013, ISSN: 0360-0300. DOI: `10.1145/2431211.2431215`. [Online]. Available: `https://doi.org/10.1145/2431211.2431215`. 13

[23] J. Mamou, O. Pereg, M. Wasserblat, A. Eirew, Y. Green, S. Guskin, P. Izsak, and D. Korat, "Term set expansion based NLP architect by Intel AI lab," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 19–24. DOI: `10.18653/v1/D18-2004`. [Online]. Available: `https://www.aclweb.org/anthology/D18-2004`. 21

[24] J. Mingers, "An empirical comparison of selection measures for decision-tree induction," *Mach. Learn.*, vol. 3, no. 4, pp. 319–342, Mar. 1989, ISSN: 0885-6125. DOI: `10.1023/A:1022645801436`. [Online]. Available: `https://doi.org/10.1023/A:1022645801436`. 10

[25] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar queries: Give me an example of what you need," *Proceedings of the VLDB Endowment*, vol. 7, no. 5, pp. 365–376, 2014. 15

[26] P. Pantel, E. Crestan, A. Borkovsky, A.-M. Popescu, and V. Vyas, "Web-scale distributional similarity and entity set expansion," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2 - Volume 2*, ser. EMNLP '09, Singapore: Association for Computational Linguistics, 2009, pp. 938–947, ISBN: 9781932432626. 21

[27] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986, ISSN: 0885-6125. DOI: `10.1023/A:1022643204877`. [Online]. Available: `https://doi.org/10.1023/A:1022643204877`. 9, 10, 12, 18, 29, 45

[28] J. R. Quinlan, "Simplifying decision trees," *Int. J. Man-Mach. Stud.*, vol. 27, no. 3, pp. 221–234, Sep. 1987, ISSN: 0020-7373. DOI: 10.1016/S0020-7373(87)80053-6. [Online]. Available: https://doi.org/10.1016/S0020-7373(87)80053-6. 14

[29] J. R. Quinlan, *C4.5: Programs for machine learning*. San Mateo, CA: Morgan Kaufmann, 1993. 9, 11, 12, 18, 29, 30, 52

[30] S. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991. DOI: 10.1109/21.97458. 6

[31] L. Sarmento, V. Jijkoun, M. Rijke, and E. Oliveira, ""more like these": Growing entity classes from seeds," Jan. 2007, pp. 959–962. DOI: 10.1145/1321440.1321585. 19

[32] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x. 10

[33] J. Shen, Z. Wu, D. Lei, J. Shang, X. Ren, and J. Han, "Setexpan: Corpus-based set expansion via context feature selection and rank ensemble," in *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2017, Skopje, Macedonia, September 18-22, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 10534, Springer, 2017, pp. 288–304. DOI: 10.1007/978-3-319-71249-9_18. [Online]. Available: https://doi.org/10.1007/978-3-319-71249-9%5C_18. 20

[34] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik, "Discovering queries based on example tuples," ser. SIGMOD '14, Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 493–504, ISBN: 9781450323765. DOI: 10.1145/2588555.2593664. [Online]. Available: https://doi.org/10.1145/2588555.2593664. 16

[35] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava, "Reverse engineering aggregation queries," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1394–1405, Aug. 2017, ISSN: 2150-8097. DOI: 10.14778/3137628.3137648. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.14778/3137628.3137648. 16

[36] TPC, *Tpc-h benchmark*, 2020. [Online]. Available: http://www.tpc.org/tpch/. 47

[37] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, "Query by output," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 535–548, ISBN: 9781605585512. DOI: 10.1145/1559845.1559902. [Online]. Available: https://doi.org/10.1145/1559845.1559902. 15

[38] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, “Query reverse engineering,” *The VLDB Journal*, vol. 23, no. 5, pp. 721–746, Oct. 2014, ISSN: 1066-8888. DOI: 10.1007/s00778-013-0349-3. [Online]. Available: https://doi.org/10.1007/s00778-013-0349-3. 2, 15, 18

[39] D. Vickrey, O. Kipersztok, and D. Koller, “An active learning approach to finding related terms.,” Jan. 2010, pp. 371–376. 21

[40] R. Wang and W. Cohen, “Iterative set expansion of named entities using the web,” Dec. 2008, pp. 1091–1096. DOI: 10.1109/ICDM.2008.145. 19

[41] R. Wang and W. Cohen, “Language-independent set expansion of named entities using the web,” Nov. 2007, pp. 342–350, ISBN: 978-0-7695-3018-5. DOI: 10.1109/ICDM.2007.104. 19

[42] R. C. Wang and W. W. Cohen, “Character-level analysis of semi-structured documents for set expansion,” in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3 - Volume 3*, ser. EMNLP ’09, Singapore: Association for Computational Linguistics, 2009, pp. 1503–1512, ISBN: 9781932432633. 19

[43] Y. Y. Weiss and S. Cohen, “Reverse engineering spj-queries from examples,” in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS ’17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 151–166, ISBN: 9781450341981. DOI: 10.1145/3034786.3056112. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3034786.3056112. 16

[44] P. Yu, Z. Huang, R. Rahimi, and J. Allan, “Corpus-based set expansion with lexical features and distributed representations,” in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM, Jul. 2019, pp. 1153–1156, ISBN: 978-1-4503-6172-9. DOI: 10.1145/3331184.3331359. 21

[45] M. Zhang, H. Elmeleegy, C. Procopiuc, and D. Srivastava, “Reverse engineering complex join queries,” Jun. 2013, pp. 809–820. DOI: 10.1145/2463676.2465320. 16, 48

[46] M. M. Zloof, “Query by example,” in *Proceedings of the national computer conference and exposition*, 1975, pp. 431–438. 15