A data structure that is semantically equivalent to Turing machine can be embedded into a constant-dimensional Euclidean space

Abstract: The dimension of a graph, particularly a complete graph, is commonly considered to increase linearly with the size of the graph. In this paper, we describe the Enterprise-Participant (EP) data model, a recursive language and equivalently a data structure that is semantically equivalent to Turing machine. We show that the data, including graphs and other transitive relations, in an EP database can be embedded into and has an isomorphic image in a constant-dimensional Euclidean space. Embedding the discrete EP data into a continuous geometric space provides an opportunity to better enrich statistical machine learning by integrating symbolic computing.

***Keywords*—** Euclidean space embedding, computability, PAC learnability, graph, transitive relation.

1 Introduction

The dimension of a graph G, denoted as $dim\ G$, is defined as the minimum number n such that G can be embedded into an Euclidean n-space $\boldsymbol{E_n}$ with every edge of G having the unit length 1, [Erdős 1965, Soifer 2009, Kavangh 2018]. Certain graphs have a constant $dim\ G$, such as a path graph has a dimension of 1, a tree has a dimension of 2, and a complete-bicolored (bipartite) graph, denoted as $\boldsymbol{K_{m,n}}$, has $dim\ (\boldsymbol{K_{m,n}}) \leq 4$. An arbitrary graph is in general considered to have an unbounded dimension, i.e., it grows linearly with the number n of its vertices, particularly $dim\ \boldsymbol{K_n} = n - 1$ for a complete graph, denoted as $\boldsymbol{K_n}$.

For decades, inventorying a large volume of data and supporting queries on transitive relations have been a long and hot research topic. Logic program languages such as Prolog on the top of relational databases were used to support queries over transitive relation for knowledge management applications in the deductive database research in 1980s and 1990s, e.g., Datalog as a known deductive database system [Agrawal 1989], [Ludwig 2009]. Graph, a network-oriented data structure, was used to represent transitive relations to manage massive volume of web pages in the Internet in such as Semantic Web Technologies, e.g., OWL (Web Ontology Language) designed to manage complex web contents, in 2000s and 2010s [Bizer 2009, Halpin 2010]. Graph again has been the data structure carrying transitive relations being embedded to a Euclidean space to inject desired knowledge to statistical machine learning models since the 2010s [Nickel 2012, Nickel 2017, Bordes 2011, Berant 2012, Berant 2014, Cai 2017]. However, these approaches have not been successful with a symptom of an unacceptable system performance, i.e., in response time in the symbolic approaches [Liu 1999, Brass 2019] and in the accuracy of prediction in machine learning due to the high dimensionality of graph data [Seshadhri 2020, Nickel 2017, Apidianaki 2022]. To avoid the challenge of the high dimensionality reduction for graphs, embedding hierarchical data into a geometric space for machine learning has become popular. The study has shown its outperformance over the graph embedding because of its low dimensionality [Berant 2014, Nickel 2017, Chamil 2020, Lin 2023].

EP is a type and variable free language system and equivalently a data structure with which an EP database can be constructed. EP can represent various transitive relations including cyclical data like a graph and hierarchical data. These transitive relations are actually the effect of EP's reduction rules on the top of physical data that are arranged in three kinds of hierarchical structures. In this paper, we will show that an EP database can be embedded into, and further has an isomorphic image in, a constant dimensional Euclidean space because of its underlying hierarchical structures.

An EP database can be syntactically converted from a finite approximation to the lambda calculus and is interpreted as a bounded function, i.e., recursive with infinite domain while guaranteed with a finite co-domain [Xu 2017]. As a result, the union of all (infinite) EP databases and equivalently the class of all bounded functions are semantically equivalent to the lambda calculus, i.e., equivalent to Turing Machines. In other words, we can use EP to accumulate information, formally a proper subset of the properties of the class of partially computable functions, as much as the finitely available time and space were allowed. EP

is more expressive than the contemporary data structures, including relational data, tree structures, and network structures (graphs).

Embedding discrete EP data to a continuous space with a constant dimension should help enrich statistical machine learning through symbolic computing and prior-knowledge, i.e., in the area of informed machine learning [Daniel 2025]. As an example, if we can use EP to embed a set of natural language utterances that strictly follow a set of standard syntactic structural rules, into a constant-dimensional geometric space through EP, we will be able to use statistical machine learning to map those utterances not following the standard rules onto the smooth and continuous geometrical space that is featured by the EP-enabled discrete points representing the standard utterances. When a non-standard utterance is very close to a standard utterance in distance, we may predict the former is actually meant to be the latter. This approach is an alternative to the natural language process solely using statistical machine learning.

In [Xu 2025], a class of bounded functions that is represented by a corresponding class of EP databases is found to be Probably Approximately Correct (PAC) learnable. In other words, receiving a finite set of sample EP expressions that are randomly selected from its infinite domain, an algorithm exists to automatically construct an EP database that produces fresh EP expressions that are not originally received as samples. With the unique capacity of a semantically Turing machine equivalent computing power, the PAC learnability, the embeddability to a constant dimensional Euclidean space, we believe the EP data structure is the foundation of a new generation of machine learning that integrates learnability, statistics, and symbolic computing for advanced applications such as a natural language process with more intelligence and accuracy. The EP data structure would be similar to the vector data structure for today's statistical based machine learning with an exception that using the EP data structure doesn't need dimensionality reduction.

In Sections 2, we discuss the differences in the approaches and the results of graph embeddings between the traditional graph embedding method and the method using EP. In Section 3, we introduce terms and assignments with which an EP database is formed, and additionally the three kinds of trees underlying EP databases that enable the embedding of EP data to a constant dimensional space. In Section 4, we introduce the reduction rules of EP that enable views of transitive data, particularly cyclical data. In Section 5, we give an isomorphic image of EP in a constant dimensional Euclidean space. This material in Section 5 appears to be tedious but logically is straightforward: mapping EP data to a 6-dimensional Euclidean space according to the three kinds of physical tree structures one EP databases.

2. Related work

The conclusion that a graph can be embedded to a constant dimensional space using EP doesn't have a simple correlation with the common understanding on a graph's linearly growing dimension. First, the embedding methods of the two approaches are different. When a graph is represented into an EP database, the basic notions of vertex and edge are transformed to different notions in EP. The resulting data structure, with its own notions of "vertex" and three kinds of "edges", can be further embedded to a constant dimensional Euclidean space, during which the embedding method obeys the rule of a graph embedding that the length of an edge maintain to be unit 1. However, the length unit of 1 is no longer applied to the edges in the original graph but to the new "edges" in the EP database that transformed the graph.

Secondly, all the properties of a graph are preserved when it is embedded into a constant dimensional space no matter what an approach is taken.

Thirdly, being able to embed a graph to a constant dimensional space may not be interesting at all if the embed method doesn't keep the length of the edges in the original graph to be the unit of 1. Given a graph, for example, we can add a vertex right into the middle of an edge and mark the new edge with a color while marking the two end vertices of the edge with a second color. The resulting graph would be a bipartite graph, which has a constant dimension of at most 4. However, this embedding doesn't add a

value even though the properties of the original graph are preserved, because we cannot gain anything new from such an embedding.

With that said, we continue being interested in discussing the constant dimensional embeddings because we believe the constant dimensional space embedding should help informed machine learning [Daniel 2025].

3. EP terms and databases for tree-structured relations

The Enterprise-Participant (EP) data model is a language system and equivalently a data structure with which an EP database can be constructed. The idea behind EP is that The Enterprise-Participant (EP) data model is a language system and equivalently a data structure with which an EP database can be constructed. The idea behind EP is that we treat all objects to be represented as functions. Given a function f that produces a value m when it is applied to an argument n, denoted as $f(n) = m$, let's think of an exercise in which we inventory the behavior of f in a database. We can rewrite $f(n) = m$ as $\{f\ n := m\}$, reading it as: applying f to n is assigned a value m. The set $\{f\ n := m\}$, called a database, is an approximation of f. When we apply f to an additional argument n', we would obtain a better approximation $\{f\ n := m, f\ n' := m'\}$ where $f(n') = m'$. In addition, m could be another function such that $m\ (p) = q$ for a given input p. So we can exhibit more properties of f with the accumulated approximation $\{f\ n := m, f\ n' := m', m\ p := q\}$ or equivalently $\{f\ n\ p := q, f\ n' := m'\}$. From the database $\{f\ n := m, f\ n' := m', m\ p := q\}$, we can derive: $(f\ (n))\ (p) = q$.

The EP data model is described as a language system ($\boldsymbol{F}$, $\boldsymbol{C}$, *null*, ·, (,), :=, D) where

1) $\boldsymbol{F}$ is a set of identifiers (function names),
2) $\boldsymbol{C}$ is a set of constants, disjoint from $\boldsymbol{F}$. It could include infinite domains such as strings, integers, reals, and timestamps. $\boldsymbol{C}$ includes a special constant *null*.
3) · is a binary operation that produces a set $\boldsymbol{E}$ such that

$m \in \boldsymbol{F} \Rightarrow m \in \boldsymbol{E}$

$m \in \boldsymbol{E}, (n \in \boldsymbol{C} \cup \boldsymbol{E}) \Rightarrow (m \cdot n) \in \boldsymbol{E}$

Here we simply write $(m \cdot n)$ as $(m\ \ n)$ and further $m\ n$ when $(m\ n)$ is implied, where m, n, and $m\ n$ are called a function, an argument, and the corresponding application. For a $x \in \boldsymbol{E}$, we call x a term. We further use SUB(x) to denote all the subterms of x, where $x \in$ SUB(x) and $m, n \in$ SUB(x) when $x \equiv m\ n$. We use SUB$^+$(p) to denote all the proper subterms of x, i.e., SUB$^+$ (x) = SUB(x) – {x}.

Identifiers are the most basic building blocks in EP. Like in programming languages, we can choose alphanumeric tokens as identifiers, such as *abc123*, *_abc*, and more commonly we take words from a natural language vocabulary as identifiers, such as *hello*, *John*, *sport*, *law*, and *person*.

A term is either an identifier $x \in \boldsymbol{F}$ or an application $x\ y \in \boldsymbol{E}$ where $x \in \boldsymbol{E}$, y $\in \boldsymbol{C} \cup \boldsymbol{E}$, such as $x\ x$, $x\ 3.14$, $(a\ b\ c)$ $(d\ e\ 3\ (d\ t\ 3))$ are legitimate terms where $x, a, b, c, d, e, t \in \boldsymbol{F}$.

By terms alone, we can represent containment relationships. For example, the hierarchical structure of geographical locations can be expressed, such as *USA Florida Miami*. The terms embed transitive relations, such as we can infer *Miami* is part of *USA* because *Miami* is part of *Florida* and *Florida* is part of *USA*.

The containment relationships embedded in a term can be formally defined with a built-in tree-structured relation. Given a term $m\ n \in \boldsymbol{E}$, a relation, denoted as {+ [1], is used to represent the relationships between the application and its function, i.e.,

$m\ n \in \boldsymbol{E} \Rightarrow\ m\ n\ \{+\ m$

[1] All the built-in transitive relations defined in the paper are denoted with symbols without given a name. See all the operators and their proved properties in [Xu 2010].

The containment relationships are also defined with a transitive relation. Given a term $m_0 \ldots m_{n-1} m_n \in \boldsymbol{E}$, where $n \geq 0$, the transitive relation, denoted as {=+, represents the relationships between the given term and its left-most terms, i.e.,

$m_0 \ldots m_{n-1} m_n \in \boldsymbol{E} \Rightarrow$
$m_0 \ldots m_{n-1} m_n \{=+ m_0 \ldots m_{n-1} m_n$
$m_0 \ldots m_{n-1} m_n \{=+ m_0 \ldots m_{n-1}$
…
$m_0 \ldots m_{n-1} m_n \{=+ m_0$

For the earlier geographical containment example, we have *USA Florida Miami* {=+ *USA* because *USA Florida Miami* {=+ *USA Florida* and *USA Florida* {=+ *USA*. Given $a, b, c \in \boldsymbol{E}$, in general, we have a {=+ c if a {=+ b and b {=+ c.

We just defined one tree-structured relational operator {+ and a transitive relational operator {=+, which take the most left identifier of a term in D as a root. When we take the right most identifier of a term in D as a root, we develop another tree-structured relation, denoted as {- and the corresponding transitive relation {=-. Given a term $m\ n \in \boldsymbol{E}$, a relation, denoted as {-, is used to represent the relationship between an application and its argument, i.e.,

$m\ n \in \boldsymbol{E} \Rightarrow\ m\ n \{- n.$

Given a term $m_0\ (m_1 \ldots m_n) \in \boldsymbol{E}$, where $n \geq 0$, a relation, denoted as {=-, represents the relationships between the given term and its right-most terms, i.e.,

$m_0\ (m_1 \ldots m_n) \in \boldsymbol{E} \Rightarrow$
$m_0\ (m_1 \ldots m_n) \{=- m_0\ (m_1 \ldots m_n)$
$m_0\ (m_1 \ldots m_n) \{=- m_1 \ldots m_n$
…
$m_0\ (m_1 \ldots m_n) \{=- m_n$

For example, we have *college.edu math math100* (*college.edu admin* (*SSA.gov John*)) {=- *John*. This transitive relation is a way to infer that *John* is associated with *college.edu*. Given terms a, b, c, in general, we have a {=- c if a {=- b and b {=- c.

A term can be assigned with another term to form a database. The EP data model is further consists of:

4) := is the Cartesian product $\boldsymbol{E} \times (\boldsymbol{C} \cup \boldsymbol{E})$, i.e., := = $\boldsymbol{E} \times (\boldsymbol{C} \cup \boldsymbol{E})$. When a pair $(p, q) \in$:=, we denote it as $p := q$, which is called an assignment, where p and q are the assignee and assigner respectively.

5) D, called a database, is a finite set of terms and a finite set of assignments, i.e., $D \subset (\boldsymbol{E} \cup :=)$, such that for each assignment $p := q \in D$, where $p, q \in \boldsymbol{E}$, the following constraints are met:

1. p has only one assigner, i.e., $p := q$ and $p := q' \in D \Rightarrow q \equiv q'$
2. A proper subterm of p cannot be an assignee, i.e., $p := q \in D \Rightarrow \forall x \in \mathrm{SUB}^{+}(p)\ [\forall m \in \boldsymbol{E}\ [x := m \notin D]]$
3. q can not be an assignee, i.e., $p := q \in D \Rightarrow \forall a \in (\boldsymbol{C} \cup \boldsymbol{E})\ [q := a \notin D]$

A database can be considered as a program representing data and business logic. Here are a few examples: 1) {$u\ v := v$; $v\ u := u$} for the directed graph including a cycle; 2) {$a\ b\ c := d$; $d\ (e\ f) := g$}, for a random database having a non-cyclic but transitive relation; 3) {*Joe birth date = '03/20/1980'*; *Joe height = 6*; *Joe weight = 180*; *joe male*} for a random database; and 4) {*SSA.gov John SSN := 123456789; SSA.gov John birth := '6/1/1996'; college.edu admin (SSA.gov John) enroll := '9/1/2014'; college.edu admin (SSA.gov John) major := college.edu math; college.edu math math100 (college.edu admin (SSA.gov John)) grade := A; GroceryStore (SSA.gov John);*}, for a sample school administration database.

To this end, we give Fig. 1 showing the data structures for a few sample databases.

Fig. 1.1. The data structure for a sample database

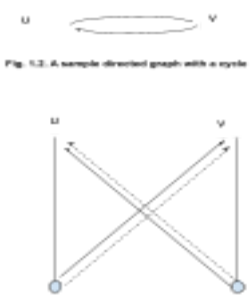

Fig. 1.3. The data structure for the graph in Fig. 1.2

Before moving on to the next section, we introduce a syntactical order among terms $\boldsymbol{E}$, designated by one of its operators $<_{EP}$, that is used for physical data storage and referenced in Section 5. Given any $m, n \in \boldsymbol{E}$,

1 If $m \equiv n$, then m is equal to n, denoted as $m =_{EP} n$;

2 If m and n are identifiers or constants and m is alphanumerically smaller than m, then we say m is smaller than n, denoted as $m <_{EP} n$.

3 If $m \equiv s_0 \ldots s_k\, m_{k+1} \ldots m_i$ and $n \equiv s_0 \ldots s_k\, n_{k+1} \ldots n_j$, where $k \geq 0$, i and $j > 1$, $m_{k+1} <_{EP} n_{k+1}$, then $m <_{EP} n$.

For example, we have *jessie* $<_{EP}$ *joe*, *Florida Miami (South Beach)* $<_{EP}$ *Florida Tampa*.

4 EP Reductions for pre-ordering relations

As part of the EP data model, a set of reduction rules are available on an EP database. Given a database D, a term $n \in \boldsymbol{C} \cup \boldsymbol{E}$ is in EP normal form (or normal form in brief) if and only if

1. It is a constant $c \in \boldsymbol{C}$, or
2. It is a term $n \in D$ and n is not an assignee, i.e., $n \in D$ and $\forall\, b \in \boldsymbol{E}\ [n := b \notin D]$.

Let NF(D) denote the entire set of normal forms under a database D, where $null \in$ NF(D) and other constants $c \in$ NF(D) only if $c \in D$.

Definition 4.1 Given a database D, we have one-step *reduction* rules, denoted as $\rightarrow_D$:

1. An assignee is reduced to the assigner, i.e., $a := b \in D \Rightarrow\ a \rightarrow_D b$
2. An identifier not in the database is reduced to *null*, i.e., $a \in \boldsymbol{F}, a \notin D \Rightarrow\ a \rightarrow_D null$
3. If a and b are normal forms and $a\ b \notin D$, then $a\ b$ is reduced to *null*, i.e.,

   $a, b \in NF(D), a\ b \notin D \Rightarrow a\ b \rightarrow_D null$

4. $a \rightarrow_D a', b \rightarrow_D b' \Rightarrow a\ b \rightarrow_D a'\ b'$

If we have a sequence of reductions: $a \rightarrow_D a_0$, $a_0 \rightarrow_D a_1$, ..., $a_{n-1} \rightarrow_D a_n$, where $n \geq 0$, we say that a is effectively, i.e., in finite steps, reduced to a_n, denoted as $a \rightarrow_D a_n$. If $a_1 \rightarrow_D b$ and $a_2 \rightarrow_D b$, then we say that b, a_1 and a_2 are equal (equivalence relation), denoted as $b == a_1 == a_2$. We also define $a == a$ for any term $a \in D$. Note that given an application, e.g., $a\ b\ c$, the sequence of the reductions toward its normal form is unique because the restricted syntactical form of an application only allows one unique reduction sequence, e.g., $a\ b\ c$ is restrictedly written as $((a\ b)\ c)$.

A term a has a normal form b if b is in normal form and $a \rightarrow_D b$. Any term $a \in \boldsymbol{C} \cup \boldsymbol{E}$ under a database D has one and only one normal form and can be effectively reduced in finite steps. A constant, such as *null*, *3.14*, or "Hello", is always in normal form.

With the EP reduction rules introduced, we give two sample databases representing graphs. First, we give a graph with a single directed link with two end vertices: $D = \{u_1\ u_2 := u_2\}$. EP has a reduction on D: $u_1\ u_2 \rightarrow_D u_2$. Here the vertex u_1 can be viewed as a function that yields to u_2 when it is applied to u_2, which simulates that one from u_1 can walk over to u_2. Because the database is only defined with the single pair $\{u_1\ u_2 := u_2\}$, applying u_2 to anything else would yield to meaningless, denoted as *null* in EP: $u_2\ y$

$\rightarrow_D$ *null* for any *y*. This reduction says that one from u_2 cannot reach out to anything else. In this database, NF(D) = {*null*, u_1, u_2}.

Now, let's give another EP database representing a graph with a triangle: D = {$v_1\ v_2 := v_2$; $v_2\ v_1 := v_1$; $v_2\ v_3 := v_3$; $v_3\ v_2 := v_2$; $v_3\ v_1 := v_1$; $v_1\ v_3 := v_3$}, where each undirected edge is expressed by a pair of directed edges, for example, $v_1\ v_2 := v_2$ and $v_2\ v_1 := v_1$ for the edge between vertices v_1 and v_2. With the database D, the system has the following infinitely possible reductions:

$v_1\ v_2\ v_1 \rightarrow_D v_1$

$v_3\ v_2\ v_1\ v_2\ \ldots\ v_1 \rightarrow_D v_1$

…

The sample reductions above simulate how one can walk from one vertex to another along the edges of the triangle. In this database, NF(D) = {*null*, v_1, v_2, v_3}.

Given a database D, there is a function Y_D: $\boldsymbol{E} \rightarrow$ NF(D) that is defined as:

Y_D = {(m, n) | $m \in \boldsymbol{E}$, $n \in$ NF(D), and $m \rightarrow_D n$}.

We call such a function Y_D bounded because $\boldsymbol{E}$ is infinite and NF(D) is finite. To support the discussion in the coming section, we further introduce another set similar to but semantically equivalent to Y_D, where only meaningful terms (reduced to non-*null* normal forms) are included:

W_D = {(m, n) | $m \in \boldsymbol{E}$, $n \in$ NF(D) \ {*null*}, and $m \rightarrow_D n$}.

Given a database, potentially an arbitrary number of terms can be reduced to a non-*null* normal form. For example, D = {$x\ x := x$} has W_D = {(x, x), ($x\ x$, x), ($x\ x\ x$, x), …}, where except for the first two elements, the others are derived by reductions. Note all databases have infinite elements in W_D. For example, D = {$a\ b\ c := d$; $d\ e := f$;} has only one derived element ($a\ b\ c\ e$, f) in W_D and D = {$a\ b\ c$:= c;} has no derived elements in W_D.

Given an equivalence relationship $m\ n == q$ under a database D, a binary relation, denoted as (+, is defined to represent the relationship between m and q, i.e.,

$m\ n == q \Rightarrow q$ (+ m

Given $m_0\ \ldots\ m_{n-1}\ m_n == q$ under a database D, where $n \geq 0$, we define a pre-ordering relation, denoted as (=+, such that

q (=+ $m_0\ \ldots\ m_{n-1}\ m_n$

q (=+ $m_0\ \ldots\ m_{n-1}$

…

q (=+ m_0

Given a database {$a\ b\ c$: = d; $d\ e := f$}, for example, we have f (=+ a because f (=+ d and d (=+ a. Given $a \rightarrow_D b$ and $b \rightarrow_D c$, we also have c (=+ a. Given a graph with two vertices u and v connected by two bi-directional connections, represented as D = {$u\ v := v$; $v\ u := u$}, a path from u to v is expressed as v (=+ u; and a cycle between u and v is expressed as (v (=+ u *and* u (=+ v). We demonstrate these relationships in Fig. 1.2 and 1.3. The relations ==, {=+, {=-, and (=+ are all pre-ordered. We are more interested in calling (=+ a pre-ordered relation because it can express paths and cycles in a graph and other cyclical data.

5 An isomorphic mapping between EP and a substructure in a constant-dimensional Euclidean space

Given a database D over **E** and an Euclidean space with a dimension of 6, denoted as $\mathcal{R}$, we develop a mapping from D to $\mathcal{R}$, from which we obtain an image of D, denoted as $\mathcal{D}$. On the top of $\mathcal{D}$, we develop an image of EP's one-step reduction rule $\rightarrow_D$, denoted as $\rightarrow_{\mathcal{D}}$. With the reduction rule $\rightarrow_{\mathcal{D}}$, we expand $\mathcal{D}$ to $\mathcal{W}_{\mathcal{D}} \subset \mathcal{R}$, attempted to be an image of W_D in EP. We further show the structures D and $\rightarrow_D$ are

isomorphic to the corresponding images $\mathcal{D}$ and $\rightarrow_{\mathcal{D}}$ respectively while W_D is an injection to $\mathcal{W}_{\mathcal{D}}$, where $\mathcal{W}_{\mathcal{D}}$ is added with many intermediate results that do not impact the reserve surjection from $\mathcal{W}_{\mathcal{D}}$ to W_D.

Before we start the discussion, we give alternative notations in a data structure for the notations in an EP database so that we can exchangeably use all of them. Fig. 1.1 and 1.3 give the data structures for two sample databases introduced in Section 3, where a dot (node) represents an application that is connected to the function using an up-down link and to the argument using a dashed arrow. A dot doesn't have a label but it can be named by the corresponding application, e.g, *u v* in Fig. 1.3, according to the data structure having an upward link to the function and a dashed arrow to the argument from the dot. Given an assignment, the assignee connects to the assigner using a solid arrow, e.g., the dot *u v* points to *v*. Any term in *D* is called a node in the corresponding data structure, e.g., the label *v*, the dot named as *u v* in Fig. 1.3. See Appendix A for a formal discussion that builds an isomorphism between an EP database and the corresponding EP data structure.

5.1 Database image $\mathcal{D}$ in $\mathcal{R}$

While other and likely better embedding methods are available, such as Hyperbolic embeddings [Nickel 2017, Chamil 2020, Lin 2023], we simply choose a Euclidean space $\mathcal{R}$ of a dimension of 6 for a demonstration purpose in this section. The idea is very simple: let each of the 3 categories of the hierarchical structures under {=+, {=-, and := take one plane in $\mathcal{R}$. Specifically, the trees under {=+ are in the first plane, denoted P1, the trees under {=- are in the second plane, P2, and the trees under := (and later extended to ==) are in the third one, P3.

For a given node in a database, e.g., $m \in \mathcal{D}$, we reserve a space on a plane, e.g., P1. The space is defined by 1) an open interval (x_1, x_2) where x_1 and x_2 are fractions along X-Axis, and 2) a semi-closed interval $[y, \infty)$ where y is a non-negative integer along Y-Axis. For example, in Fig. 3.1, the term *d* is reserved in P1 with the area surrounded by dashed lines right below the letter "*d*" at the center of the figure. It is precisely defined by the intervals (⅓, ⅔) and [1, ∞).

To maintain EP's physical storage constraints, e.g., nodes must be stored in an ordered sequence (e.g., implemented in a B-tree) while a database manages to record randomly entered nodes, the width of an open interval along X-Axis to be reserved varies for a given node, depending on the time when the node is entered. When a rectangle area with an open internal (x_1, x_2) and an semi-closed internal $[y, \infty)$ is initially reserved for a node *m*, the interval (x_1, x_2) is divided to three equal open intervals: $(x_1, x_1 + \Delta)$, $(x_1 + \Delta, x_1 + 2\Delta)$, and $(x_1 + 2\Delta, x_2)$, where $\Delta = (x_2 - x_1)/3$. When an identifier that is first ever to be mapped to a plane such as P1, it will be mapped to the center sub rectangle with the horizontal interval $(x_1 + \Delta, x_1 + 2\Delta)$ and the vertical internal [y +1, ∞). For example, *d* in Fig. 1.3 took the center sub interval (⅓, ⅔) as it was the first ever node being entered in P1 before other nodes. A second identifier will take either the first or the third sub rectangle, depending on its alphabetic order. For example, the term *b*, after *d* was entered, is reserved within the left interval (0, ⅓). To leave space for identifiers that are later to be entered, the left interval is recursively divided into 3 sub intervals and the node *b* actually takes only the center sub interval (1/9, 2/9). For all identifiers, the interval along Y-Axis in a plane like P1 is always [1, ∞). See the area defined by (1/9, 2/9) and [1, ∞) that is reserved for the *b* node in Fig. 1.3.

Given an area defined by (x_1, x_2) and $[y, \infty)$ for a term *m*, a subordinate of *m*, such as *m n* for *m n* {+ *m* in P1 or *n m* for *n m* {- *m* in P2, must be placed within *m*'s area in the given plane with the y-coordinate of $y + 1$. For example, the term *b c* is reserved with the area defined by the intervals (⅛ - 1/54, ⅛ + 1/54) and [2, ∞) within the term *b* reserved area defined by intervals (1/9, 2/9) and [1, ∞) in P1 as depicted in Fig. 1.3.

For an assignment, e.g., *m* := *n*, or an equivalence relation, e.g., *m* == *n*, we use the plane P3, where *n* takes the Y-Axis interval [1, ∞) and *m* takes the Y-Axis interval [2, ∞) all the time. See Fig. 3.3 for some examples.

For a reserved X-Axis interval in a given plane, if a node *m* has already reserved the center sub interval and if another node *m*' needs to be inserted into the same interval beside the *m* reserved center sub interval, the operator $<_{EP}$ given at the end of Section 3 determines if *m*' takes either the left subarea or the right subarea: if *m*' $<_{EP}$ *m*, then *m*' will take the left subarea, otherwise if it is larger, it will take the right subarea. For example, because *a e* (*b c f*) $<_{EP}$ *d* (*b c f*), the node *a e* (*b c f*) takes the left subarea under *g* in the plane P3 as depicted in Fig. 3.3.

Because a X-Axis interval is open, the two end points of the interval will never be reserved for an EP node. With more EP nodes are added into a plane, the gap between an open point of a X-Axis interval, e.g., the left open point of the interval for *b*, and an open point of an inner X-Axis interval, e.g., the left open point of the interval for *b c*, is getting smaller. However, the continuous plane guarantees more and more nodes can be continuously inserted between the shrinking gap.

For each given plane, we only utilize the open interval between 0 and 1, i.e., (0, 1), along the X-Axis for potentially infinite EP nodes. The top level starts at 1 along Y-Axis. Therefore a point *p* (0, 0) in a given plane signifies that *p* is not in the plane. The first ever node entered into a plane is mapped to the point (½, 1) in $\mathcal{R}$, which is the first point entered in the given plane.

We use a vector to capture a mapped point in $\mathcal{R}$. For example the term *d* is mapped to a point with the vector (½, 1, ⅓, 0, 0, 0, ⅛, 1, 1/9), as depicted in Tab. 1. In the vector, the first two numbers are its x- and y-coordinates in P1, the third is the width of its open interval centered at its x-coordinate in P1. The middle three 0s are about P2, signifying that *d* is absent. The last three numbers are about *d*'s x- and y-coordinates and its open interval width in P3. A vector is also associated with the corresponding EP term, e.g., *d*, and a name, e.g., $\mathcal{d}$, named after the EP term in the **Pacifico** font, for the point in R. A collection of such vectors is denoted as "Matrix" in this paper.

With the protocol laid out up to this point, we give detailed steps below to map an EP database to $\mathcal{R}$.

Definition 5.1 Term mapping. Given a database *D*, we initiate a corresponding subspace $\mathcal{D} = \varnothing$ in $\mathcal{R}$. $\mathcal{D}$ is physically represented by a sequence of vectors, denoted as Matrix. For each term $t \in D$, call the following function, denoted as *map-term*(*t*):

1. If *t* is an identifier *i*, we search P1's top layer sequence to see if *i* has already been mapped to a point in the sequence that is ordered by $<_{EP}$ (i.e., the alphabetical order at the identifier level). The search is done against Matrix, where *i* and $\mathcal{i}$ are recorded together when *i* is mapped to $\mathcal{i}$ in $\mathcal{R}$. See Tab. 1 for an example. If it has already inserted to $\mathcal{R}$, i.e., *i* has a vector in Matrix, then do nothing and return. If it is not yet, find out the geometric information about the left and right neighbors between which $\mathcal{i}$ needs to be inserted, create $\mathcal{i}$ by calculating $\mathcal{i}$'s x-coordinate, say *x*, and $\mathcal{i}$'s open interval with on X-Axis, say *r*, and assigning 1 as its y-coordinate in P1. At this point, we leave 0 for all the others in P2 and P3 for now. (The calculation is done by the protocol laid out at the earlier of this section.) This is the end of Step 1 and the *map-term*(*t*) call. Return to the caller.

2. If *t* is an application, e.g., $t \equiv m\ n$. We call *map-term*(*m*) and *map-term*(*n*) to obtain the corresponding $\mathcal{m}$ and $\mathcal{n}$ in $\mathcal{R}$ first, and then,

   a) Search $\mathcal{m}$'s subordinates in P1 to see if (*m n*) has already been mapped to a corresponding ($\mathcal{m}$ $\mathcal{n}$).

      If yes, move to Step b) below. Otherwise, create a point in P1, denoted as ($\mathcal{m}$ $\mathcal{n}$), under $\mathcal{m}$'s subordinate sequence using the protocol described at the beginning of this section. A new entry for ($\mathcal{m}$ $\mathcal{n}$) would have been created in Matrix. We fill in 0s for all the other vector elements in P2 and P3 for now in the Matrix.

b) Search to see if $\boldsymbol{n}$ has already been in P2. If $\boldsymbol{n}$ is not in P2, insert it into P2. Note if n is an identifier, $\boldsymbol{n}$ will be inserted into the top layer of P2. Otherwise, we insert $\boldsymbol{n}$, when $\boldsymbol{n} \equiv \boldsymbol{n}_1\ \boldsymbol{n}_2$, to the subordinate sequence of $\boldsymbol{n}_2$ (see d (b c f) under b c f in Tab. 1 for an example). ($\boldsymbol{n}$'s location and the reserved area information in P2 would have been recorded in Matrix.)

c) Search to see if ($\boldsymbol{m}$ $\boldsymbol{n}$) has already in the subordinate sequence of $\boldsymbol{n}$ in P2. If not, insert ($\boldsymbol{m}$ $\boldsymbol{n}$) to the subordinate sequence of $\boldsymbol{n}$ in P2. (($\boldsymbol{m}$ $\boldsymbol{n}$)'s location and the reserved area information in P2 would have been recorded into Matrix.)

Definition 5.2 Assignment mapping. For each term $t \in D$, call *map-assignment*(t): if t doesn't have an assigner, do nothing and return as it has already been assigned with (0, 0) as the coordinate of the corresponding $\boldsymbol{\ell}$ in P3 through a *map-term*() call. Otherwise, for $t := s \in D$, we search if s is in P3. If not, insert s into the top layer of P3 (note that s would be at the top layer of P3 always). Once s is in P3, we insert t into the subordinate sequence of s according to the protocol laid out at the beginning of this section.

See the EP terms and the corresponding points in Fig. 3.1, 3.2, 3.3, and Tab. 1 for the example database depicted in Fig. 2.

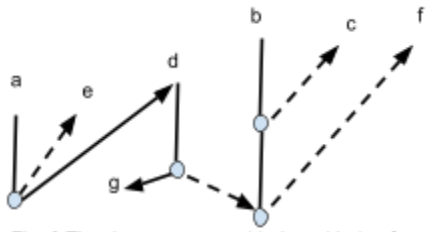

Fig. 2 The data structure with three kinds of trees for
D = {d (b c f) := g; a e := d}

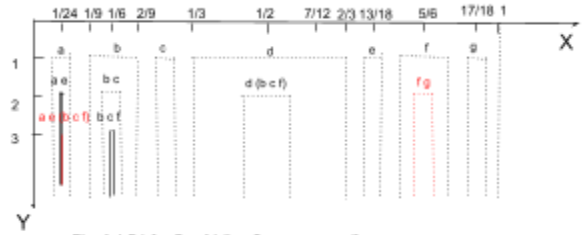

Fig. 3.1 P1 for D = {d (b c f) := g; a e := d}

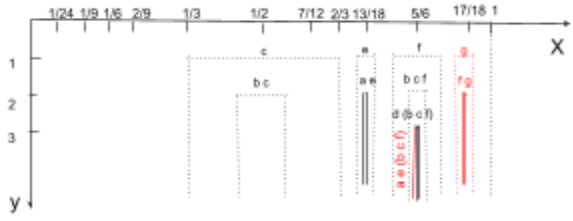

Fig. 3.2 P2 for D = {d (b c f) := g; a e := d}

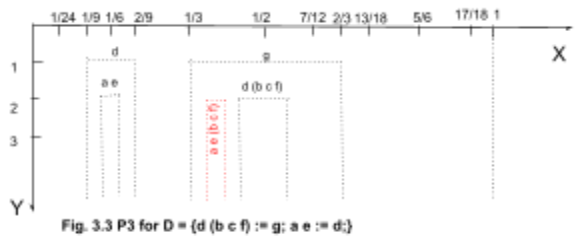

Fig. 3.3 P3 for D = {d (b c f) := g; a e := d;}

Point in $\mathcal{R}$	EP term	P1			P2			P3		
		X	Y	r	X	Y	r	X	Y	r
a	*a*	1/24	1	1/27	0	0	0	0	0	0
b	*b*	1/6	1	1/9	0	0	0	0	0	0
c	*c*	5/18	1	1/27	1/2	1	1/3	0	0	0
d	*d*	1/2	1	1/3	0	0	0	1/6	1	1/9
e	*e*	13/18	1	1/27	13/18	1	1/27	0	0	0
f	*f*	5/6	1	1/9	5/6	1	1/9	0	0	0
g	*g*	17/18	1	1/27	0	0	0	1/2	1	1/3
$a\ e$	*a e*	1/24	2	1/81	13/18	2	1/81	1/6	2	1/27
$b\ c$	*b c*	1/6	2	1/27	1/2	2	1/9	0	0	0
$b\ c\ f$	*b c f*	1/6	3	1/81	5/6	2	1/27	0	0	0
$d\ (b\ c\ f)$	*d (b c f)*	1/2	2	1/9	5/6	3	1/81	1/2	2	1/9
$a\ e\ (b\ c\ f)$	*a e (b c f)*	1/24	3	1/243	399/486	3	1/243	25/54	2	1/27

Tab. 1: The Matrix for $D = \{d\ (b\ c\ f) : = g; a\ e := d\}$, where P1 is the plane for the trees under {=+, P2 is for the trees under {=-, and P3 is for the trees under == including :=. In each plane, the columns x and y are the x- and y-coordinate of a point m in the plane mapped from a term *m*, and the column r is the open interval width reserved for *m*'s subordinates (in correspondence to dashed horizontal lines in Fig. 3.1, 3.2, and 3.3).

So far, we have mapped *D* to $\mathcal{D}$ in $\mathcal{R}$. It is clear that we have the following results:

Lemma 5.3 1 *D* and $\mathcal{D}$ are isomorphic.

2 The tree structures under {+, {-, and := are reserved in $\mathcal{D}$.

Proof: 1 *D* and $\mathcal{D}$ are isomorphic.

a) for every $t \in D$, there is a $\ell \in \mathcal{D}$. The function call *map-term*() in Definition 5.1 maps every term *t* in *D* to a point ℓ in $\mathcal{R}$.
b) for every $\ell \in \mathcal{D}$, there is a $t \in D$. In Matrix, each vector is associated with the corresponding EP term from which the vector is transformed from. We simply choose the EP term as the reverse mapping for a given vector.

2) The tree structures under {+, {-, and := are reserved in $\mathcal{D}$.

a) Given *m* and *n* in ***E***, the relations {+ and {- are defined as *m n* {+ *m* and *n m* {- *m*. When *m*, *n*, *n m*, and *m n* are mapped to $\mathcal{R}$ in Definition 5.1, $m\ n$ and $n\ m$ are located in the immediate

subordinate sequences of m in P1 and P2 respectively, i.e., $|(m\ n).P1.x - m.P1.x| < m.P1.r/2$ and $(m\ n).P1.y - m.P1.y = 1$ for P1, and $|(n\ m).P2.x - m.P2.x| < m.P2.r/2$ and $(n\ m).P2.y - m.P2.y = 1$.

b) The tree structure under := has the correspondence in P3 with only two layers, as defined in Definition 5.2. It is a similar situation as those for {+ and {- in P1 and P2 respectively, as shown above. □

Note the transitive relations {=+ and {=- are clearly reserved in $\mathcal{D}$ as well. (No further discussion is necessary.)

5.2 W_d in EP vs. $\mathcal{W}_\mathcal{D}$ in $\mathcal{R}$

In this section, we expand $\mathcal{D}$ to $\mathcal{W}_\mathcal{D}$ in $\mathcal{R}$ in a correspondence to W_D by introducing a one-step reduction rule $\rightarrow_\mathcal{D}$ in $\mathcal{R}$ that mirrors $\rightarrow_D$ in EP. This expansion will have the multi-step reduction of EP to be reflected in $\mathcal{R}$ (and clearly so are those EP pre-ordering relations).

Definition 5.4 Given $m, n \in \mathcal{R}$, if m is a subordinate under n in P3, i.e., $m.P3.y - 1 = n.P3.y$ and $|m.P3.x - n.P3.x| < n.P3.r / 2$, we say m is one-step reducible to n, denoted as $m \rightarrow_\mathcal{D} n$. Otherwise, we say m is one-step reducible to $null$, denoted as $m \rightarrow_\mathcal{D} null$. For convenience, we also use $m \rightarrow_\mathcal{D} m$ to denote no action taken during a reduction process as m cannot be further reduced.

In Definition 5.4, we allow m and n to be in $\mathcal{R}$ in order to make the reduction $\rightarrow_\mathcal{D}$ rule applicable to points in $\mathcal{W}_\mathcal{D}$ that are beyond $\mathcal{D}$. But up to now, only the points in $\mathcal{D}$ will make sense under the $\rightarrow_\mathcal{D}$ reduction rule, i.e.,

Lemma 5.5 $m := n \in D \Rightarrow m \rightarrow_\mathcal{D} n$

Proof. When $m := n \in D$, Definition 5.2 maps m under n's immediate subordinate sequence in P3, i.e., $m.P3.y - 1 = n.P3.y$ and $|m.P3.x - n.P3.x| < n.p3.r / 2$. By Definition 5.4, we have $m \rightarrow_\mathcal{D} n$. □

Using the $\rightarrow_\mathcal{D}$ reduction rule, we expand $\mathcal{D}$ to include images reflecting meaningful terms in $\boldsymbol{E}$ under a D, i.e., for every term m , $n \in \boldsymbol{E}$, $m \neq null$, and $m \rightarrow_D n$, we expand $\mathcal{D}$ to include m and n in $\mathcal{R}$ such that $m \rightarrow_\mathcal{D} n$. Actually the construction of $\mathcal{W}_\mathcal{D}$ starts with the one-step reduction rule $\rightarrow_\mathcal{D}$:

Definition 5.6 The construction operator • is a binary operation that produces a set $\mathcal{W}_\mathcal{D} \subset \mathcal{R}$ such that

I) $m \in \mathcal{D} \Rightarrow m \in \mathcal{W}_\mathcal{D}$

II) if $m, n \in \mathcal{W}_\mathcal{D}$, $m \rightarrow_\mathcal{D} m'$, $n \rightarrow_\mathcal{D} n'$, and $(m'\ n')$ is in $\mathcal{W}_\mathcal{D}$, i.e., there is a third point $(m'\ n') \equiv p \in \mathcal{W}_\mathcal{D}$ such that $|m'.P1.x - p.P1.x| < m'.P1.r /2$, $m'.P1.y + 1 = p.P1.y$, $|n'.P2.x - p.P2.x| < n'.P2.r/2$, and $n'.P2.y + 1 = p.P2.y$, then we construct a new point, denoted by $m\ n \in \mathcal{W}_\mathcal{D}$, into $\mathcal{R}$, and further extend the definition of the one-step reduction $\rightarrow_\mathcal{D}$ to include $m\ n \rightarrow_\mathcal{D} p$. Specifically,

1) call *map-term*($m\ n$) to create points in P1 and P2 for $m\ n$, where $m\ n$ is the EP term in correspondence to $m\ n$ (according to Lemma 5.3.1). (This expansion brings meaningful terms, even not in D, into $\mathcal{W}_\mathcal{D}$ in a correspondence to W_D.)

2) insert ($m\ n$) to the subordinate sequence of p in P3 by calling *map-assignment*() as given in Definition 5.2. Such an addition in P3 allows $m\ n \rightarrow_D p$, where *m n* is not in *D*. (This expansion brings the reduction rule $\rightarrow_D$ in EP , more than an assignment in *D*, into W_D in a correspondence to W_D.)

III) if $m, n \in W_D$, $m \rightarrow_D m'$, $n \rightarrow_D n'$, and $(m'\ n')$ is not in W_D, we continue to construct $m\ n \in W_D$, such that $m\ n \rightarrow_D null$.

Definition 5.6.III above introduces intermediate results, such as $a\ b \rightarrow_D null$, into W_D. This step is not desired but necessary because an EP database allows an assignment like $a\ null := c$, where we may have another term $d\ e \rightarrow_D null$ to be an intermediate result for the reduction from (a ($d\ e$)) to c. In this case, if we don't keep $d\ e$ to be a point in W_D, we will not be able to map the EP one-step reduction rule $a \rightarrow_D a'$, $b \rightarrow_D b' \Rightarrow a\ b \rightarrow_D a'\ b'$ that is defined in Definition 4.1.4 when the special symbol *null* is involved.

Lemma 5.7 The relations $\rightarrow_D$ and $\rightarrow_D$ are isomorphic.

I). $\rightarrow_D \Rightarrow \rightarrow_D$

Given $a, b \in E$ and $a \rightarrow_D b$, there are the following possibilities according to the definition of $\rightarrow_D$ in Section 4:

Case 1: An assignee is reduced to the assigner, i.e., $a := b \in D \Rightarrow a \rightarrow_D b$. We have $a \rightarrow_D b$ from Lemma 5.5.

Case 2: An identifier not in the database is reduced to *null*, i.e., $a \in F, a \notin D \Rightarrow a \rightarrow_D null$. Because a is not in D, neither is a in R according to the Definition 5.1. Because a is not in R, we have $a \rightarrow_D null$ according to Definition 5.4.

Case 3: If a and b are normal forms and $a\ b \notin D$, then $a\ b$ is reduced to *null*, i.e., $a, b \in NF(D), a\ b \notin D \Rightarrow a\ b \rightarrow_D null$. Because a and b are normal forms, a and b are in D, so are a and b in R. However, since $a\ b$ is not in D, neither is $a\ b$ in R. According to Definition 5.4, we have $a\ b \rightarrow_D null$.

Case 4: $a \rightarrow_D a', b \rightarrow_D b' \Rightarrow a\ b \rightarrow_D a'\ b'$.

a) When $a'\ b' \in W_D$, `Definition 5.6.II makes` $a\ b \in W_D$ `with` $a\ b \rightarrow_D a'\ b'$.

b) When $a'\ b' \notin W_D$, `Definition 5.6.III makes` $a\ b \in W_D$ `with` $a\ b \rightarrow_D null$.

II). $\rightarrow_D \Rightarrow \rightarrow_D$

When we have $m \rightarrow_D n$, we choose the exact reverse sequence of the mappings defined in Definition 5.1, 5.2, and 5.6, i.e., take the EP terms *m* and *n* associated with the vectors m and n, and immediately we have $m \rightarrow_D n$. □

The construction process adds all derivable information to W_D. For example, we have the vector $a\ e\ (b\ c\ f) \in W_D$, as shown the entry in red color in Tab. 1, which is the only piece of meaningfully derivable information for $D = \{d\ (b\ c\ f) := g; a\ e := d\}$. Also see the image W_D with infinite vectors in Tab. 2 for $D = \{u\ v := v; v\ u := u\}$, representing infinite walks between two vertices with two directed edges.

Point in R	EP term	P1			P2			P3		
		X	Y	r	X	Y	r	X	Y	r
u	*u*	1/2	1	1/3	1/6	1	1/9	1/6	1	1/9

v	v	5/6	1	1/9	1/2	1	1/3	1/2	1	1/3
$u\ v$	$u\ v$	1.2	2	1/9	1/2	2	1/9	1/2	2	1/9
$v\ u$	$v\ u$	5/6	2	1/27	1/6	2	1/27	1/6	2	1/27
$u\ v\ u$	$u\ v\ u$	1/2	3	1/27	79/486	2	1/81	1/6	3	1/81
…	…	…	…	…	…	…	…	…	…	…

Tab. 2: Matrix for $D = \{u\ v := v;\ v\ u := u;\}$, where infinitely possible walks like $u\ v\ u$, $v\ u\ v$, … $u\ v\ u$ ... u, … have vectors in Matrix.

Theorem 5.8 W_D is an injective to $\mathcal{W}_D$ and there is a subjective from $\mathcal{W}_D$ to W_D.

I) W_D is an injective to $\mathcal{W}_D$

This is because of Definition 5.6.III, which introduces some intermediate results like $a\ b \rightarrowtail_D null$ into $\mathcal{W}_D$ while it is not in W_D. Otherwise, W_D would be isomorphic to $\mathcal{W}_D$.

II) there is a subjective from $\mathcal{W}_D$ to W_D

a) Given $m \in \mathcal{W}_D$, if $m \rightarrowtail_D null$, we choose m in the m vector to be its reverse image and we have $m \rightarrowtail_D null$ and $m \notin W_D$ according to Definition 5.1, 5.2, and 5.6.

b) Given $m \in \mathcal{W}_D$, if $m \rightarrowtail_D n$ and n is not identical to $null$, we choose m and n in the m *and* n vectors to be their reverse images and we have $m \rightarrowtail_D n$ and $m \in W_D$ according to Definition 5.1, 5.2, and 5.6.□

Now, we are ready to introduce the final reduction rule $\rightarrow_D$ in $\mathcal{R}$ which reflects the multiple step reduction rules $\rightarrow_D$ in EP as given in Section 4:

Definition 5.9 Given an arbitrary point $p \in \mathcal{R}$, a reduction rule, denoted as $\rightarrow_D$, is defined as following:

1) if p cannot be found in $\mathcal{W}_D$ (in the vector Matrix), we define p is reduced to $null$, i.e., $p \rightarrow_D null$, where $null$ is one of the points on the top layer of the sequences on P1, reflecting *null*, a special identifier (or constant) in $\boldsymbol{F}$ (or $\boldsymbol{C}$).
2) if p's y-coordinate in P3 is 0 or 1, then p is reduced to itself, denoted as $p \rightarrow_D p$.
3) if p's y-coordinate in P3 is larger than 1, and if we find another point n such that n's y-coordinate in P3 is 1 and p is within n's X-Axis open-interval in P3, i.e., $|n.\text{P3}.x - p.\text{P3}.x| < n.\text{P3}.r/2, p.\text{P3}.y > n.\text{P3}.y = 1$, then we reduce p to n, denoted as $p \rightarrow_D n$. Otherwise, we reduce p to $null$, denoted as $p \rightarrow_D null$.

Theorem 5.10 $\rightarrow_D$ and $\rightarrow_D$ are isomorphic

Proof. This is done by Lemma 5.7, Theorem 5.8, and Definition 5.9.□

As an potential application, we may define an approximation: Given an arbitrary point $p \in \mathcal{R}$ with coordinates (x_1, y_1) and (x_2, y_2) on P1 and P2 respectively, 1) if we find a point $m \in \mathcal{W}_D$ with

coordinates (x'_1, y'_1) and (x'_2, y'_2) that $\mathbf{p}$ is very close to, i.e., $|x'_1 - x_1| < \mathbf{m}.P1.r/ (2 * C)$, $|x'_2 - x_2| < \mathbf{m}.P2.r/ (2 * C)$, $|y'_1 - y_1| < 1 / (2 * C)$, and $|y'_2 - y_2| < 1 / (2 * C)$, where C is a large constant integer, then we define $\mathbf{m}$ and $\mathbf{p}$ are similar: $\mathbf{m} \simeq \mathbf{p}$, and $\mathbf{p}$ is probably reducible to $\mathbf{n}$, denoted as $\mathbf{p} \rightsquigarrow_Z \mathbf{n}$ if $\mathbf{m} \rightarrow_D \mathbf{n}$.

6 Conclusion

The main contribution of this paper is: a constant dimensional Euclidean space is constructed to mirror an EP database and the behaviors. This embedding can potentially better enrich statistical machine learning, particularly informed machine learning.

Acknowledgement Prof. A. Blumer, C. Seshadhri, and many other reviewers provided valuable inputs on a previous manuscript.

References

[Agrawal 1989] R. Agrawal, A. Borgida. Efficient management of transitive relationships in large data and knowledge bases. ACM SIGMOD Record, Volume 18, Issue 2.

[Apidianaki 2022] M. Apidianaki. From word types to tokens and back: a Survey of approaches to word meaning representation and interpretation. Computational Linguistics (2023) 49 (2): 465–523.

[Berant 2012] J. Berant, I. Dagan, J. Goldberger (2012). Learning entailment relations by global graph structure optimization. Journal of Computational Linguistics, 38(1):73-111.

[Berant 2014] J. Berant, I.Dagan, M. Adler, J. Goldberger. Efficient tree-based approximation for entailment graph learning. Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics, pages 117-125.

[Bizer 2009] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. International Journal on Semantic Web and Information Systems, 5(3):1–22, 2009.

[Bordes 2011] A. Bordes, J. Weston, R. Collobert, Y. Bengio (2011). Learning structured embeddings of knowledge bases. Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence.

[Brass 2019] S. Brass, M. Wenzel. Performance Analysis and Comparison of Deductive Systems and SQL Databases. CEUR-WS.org/Vol-2368/paper3.pdf.

[Cai 2017] H. Cai, V.W. Zheng, K. Chang. A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications. IEEE Transaction on Knowledge and Data Engineering, Sept. 2017.

[Chamil 2020] I. Chami1, A. Wolf1, D. Juan, F. Sala1, S. Ravi, and C. R´e. Low-Dimensional Hyperbolic Knowledge Graph Embedding. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 6901–6914, July 5- 10, 2020

[Daniel 2025] D. Schulz, C. Bauckhage. Informed Machine Learning (Cognitive Technologies), Springer ISSN 2197-6635 (electronic), 2025.

[Erdős 1965] P. Erdős, F. Harary, W. T. Tutte (1965). "On the dimension of a graph". Mathematika. **12** (2): 118–122.

[Halpin 2010] H. Halpin, P. Hayes, J. McCusker, D. Mcguinness, and H. Thompson. When owl: sameAs isn't the same: An analysis of identity in linked data. The Semantic Web–ISWC 2010, page 305–320, 2010.

[Kavangh 2018] R. Kavangh. "Explorations on the dimensionality of graphs". Retrieved August 16, 2018.l

[Lin 2023] Y. E. Lin, R. R. Coifman, G. Mishne, R. Talmon. Hyperbolic Diffusion Embedding and Distance for Hierarchical Representation Learning. Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[Liu 1999] M. Liu. Deductive Database Languages: Problems and Solutions. ACM Computing Surveys, Vol. 31, No, 1, March 1999.

[Ludwig 2009] S. A. Ludwig, C. Thompson, K. Amundson. Performance Analysis of a Deductive Database with a Semantic Web Reasoning Engine: ConceptBase and Racer. Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009

[Nickel 2012] M. Nickel, V. Tresp, H. P. Kriegel. Factorizing YAGO – Scalable Machine Learning for Linked Data. WWW2012 – Session: Creating and Using Links between Data Objects. April 2012, Lyon, France.

[Nickel 2017] M. Nickel, D. Kiela. Poincaré embeddings for learning hierarchical representations. 2017.

[Schaefer 2013] M. Schaefer (2013). "Realizability of graphs and linkages", in Pach, János (ed.), Thirty Essays on Geometric Graph Theory, Springer, pp. 461–482.

[Schonfinkel 1924] Moses Schönfindel, On the building blocks of mathematical logic, 1924.

[Seshadhri 2020] C. Seshadhri, A. Sharma, A. Stolman, A. Goel. The impossibility of low rank representation for triangle-rich complex network. The Proceedings of National Academy of Sciences, March 2020.

[Soifer 2009] A. Soifer (2009). The Mathematical coloring book: mathematics of coloring and the colorful life of its creators. Springer-Verlag, 2009.

[Xu 2025] K. Xu. Classes of bounded functions that are semantically equivalent to Turing-machine are PAC learnable. The 38th Annual Conference on Learning Theory (COLT 2025) - Theory of AI for Scientific Computing Workshop, June 30–July 4, 2025 in Lyon, France.

[Xu 2024] K. Xu. Outline of a PAC learnable class of bounded functions including graphs, accepted by the 6th International Conference on Machine Learning and Intelligent Systems (MLIS 2024).

[Xu 2017] K. Xu. A class of bounded functions, a database language and an extended lambda calculus. Journal of Theoretical Computer Science, Vol. 691, August 2017, Page 81 - 106, 2017.

[Xu 2010] K. Xu, J. Zhang, S.Gao. An Assessment on the Easiness of Computer Languages, The Journal of Information Technology Review, May, 2010.

Appendix A. An EP database *D* and its isomorphic data structure

Notation 4.1 Given a database D, its data structure, denoted as $\mathbb{D}$, is constructed with the following steps:

1) For an identifier or a constant $m \in D$, we add the identifier or the constant into the structure as a node, i.e., $m \in \mathbb{D}$.
2) For an application $m\ n \in D$, we draw a dot below m, draw a up-down link from m to the dot, and draw a dashed arrow from the dot to n. (Therefore, a dot can be uniquely named by $m\ n$.)
3) For an assignment $p := q \in D$, we draw a solid arrow from p to q.

When a term $m \in D$ and an assignment $p := q \in D$, we equivalently denote: $m \in \mathbb{D}$, $p := q \in \mathbb{D}$.

Theorem 4.2 A database D and its corresponding data structure $\mathbb{D}$ are isomorphic

Proof 1) $D \Rightarrow \mathbb{D}$, it is true from the notation 4.1 itself.
2) $\mathbb{D} \Rightarrow D$, where D is initiated to be empty.

a) For an identifier or a constant $m \in \mathcal{D}$, we add it to D.

b) For a dot, we find the term $m\ n$ that uniquely names it by searching upward for its application m and by searching along the dashed arrow for the argument n. we add $m\ n$ to D.

3) For a solid arrow in $\mathcal{D}$, we find the name p for the assignee, where the solid arrow started with, as we did in the step a) or b), and further we find the name q for the assigner, a constant, an identifier, or a dot that the solid arrow points to, based on the step a) or b) above. then we add $p := q$ into D. □