ADAGIO — Automated Data Augmentation of Knowledge Graphs Using Multi-expression Learning

KEVIN DRESSLER, MOHAMED AHMED SHERIF, and AXEL-CYRILLE NGONGA NGOMO, The Data Science (DICE) group, Paderborn University, Germany, Germany

The creation of an RDF knowledge graph for a particular application commonly involves a pipeline of tools that transform a set of input data sources into an RDF knowledge graph in a process called dataset augmentation. The components of such augmentation pipelines often require extensive configuration to lead to satisfactory results. Thus, non-experts are often unable to use them. We present an efficient supervised algorithm based on genetic programming for learning knowledge graph augmentation pipelines of arbitrary length. Our approach uses multi-expression learning to learn augmentation pipelines able to achieve a high F-measure on the training data. Our evaluation suggests that our approach can efficiently learn a larger class of RDF dataset augmentation tasks than the state of the art while using only a single training example. Even on the most complex augmentation problem we posed, our approach consistently achieves an average F_1-measure of 99% in under 500 iterations with an average runtime of 16 seconds.

CCS Concepts: • **Information systems** → *Entity resolution*; *Data cleaning*; **Extraction, transformation and loading**; Data exchange; Mediators and data integration; Wrappers (data mining); Deduplication; Geographic information systems; • **Computing methodologies** → **Ontology engineering**; **Semantic networks**.

Additional Key Words and Phrases: Knowledge graphs, Data augmentation, Data integration, Machine learning, Genetic programming, Multi expression programming

ACM Reference Format:
Kevin Dreßler, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo. 2022. ADAGIO — Automated Data Augmentation of Knowledge Graphs Using Multi-expression Learning. In *HT'22: 33rd ACM Conference on Hypertext and Social Media, June 28 – July 1, 2022, Barcelona, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3511095.3531287

1 INTRODUCTION

Knowledge graph augmentation encompasses the steps from an initial set of data sources to the creation of a useful knowledge graph for a particular application [26]. For example, the generation of *DBpedia*[1] demands the mappings of *infoboxes* to Resource Description Framework (RDF) resources and properties. Large bio-medical datasets such as the *Linked Cancer Genome Atlas* [24] are often the results of the transformation of relational and textual data into RDF including the disambiguation of the terms found in the original dataset. While a plethora of solutions exist for addressing single steps of the knowledge graph augmentation problem (e.g., knowledge extraction [14], link discovery [18], data fusion [15]), there is still a need to combine these solutions with each other as well as with custom dataset transformations to generate the intended data set. Despite a substantial amount of efforts to develop dataset augmentation tasks for specific use cases [1, 5, 10], there are only two frameworks dedicated to facilitating the generation of augmentation tasks, namely DEER [26] and Linked Data Integration Framework (LDIF) [25]. Both frameworks can be configured

[1] https://www.dbpedia.org/

manually, which requires expert knowledge. They also provide supervised machine learning algorithms, i.e., they can also be operated by non-experts. However, DEER can only operate on a single dataset at a time while LDIF only provides a learning algorithm [3] for its data fusion sub-module SIEVE.

We address these shortcomings by developing an automatic configuration approach (dubbed ADAGIO) for augmentation tasks in the shape of Directed Acyclic Graphs (DAGs) (i.e., with multiple inputs). We implement our approach on top of our own fork[2] of the DEER framework that supports a plugin system. Moreover, we treat the plugins as black boxes to show the adaptability of our approach in contrast to the state of the art (e.g. LDIF), which has a static pipeline architecture.

The contributions of this paper are summarized as follows:

- We present a theoretical framework of RDF dataset augmentation and a classification of DAG-shaped augmentation tasks.
- We derive an efficient genotype representation of DAG-shaped augmentation tasks for Multi Expression Programming (MEP).
- We develop various optimizations for our learning algorithm, including a set of semantic genetic operators.
- Finally, we evaluate ADAGIO to answer three research questions. In particular, we study the set of optimal hyperparameters of our approach, its performance in comparison to the state of the art and its performance characteristics. Our results show that ADAGIO outperforms the state of the art while being able to learn complex real-world DAG-shaped augmentation tasks with an average F_1-measure of 99% in at most 500 generations.

2 RELATED WORK

The objective of this work is strongly related to the fields of AutoML [7] and On-The-Fly (OTF) Computing [11]. Another closely related notion is that of knowledge graph *refinement* [22], which is concerned with predictive modelling of certain knowledge graph features.

Genetic Programming (GP) is a subfield of Genetic Algorithm (GA) [17] which originated from the application of a GA in order to evolve computer programs by John Koza[12]. In GP, programs are usually encoded as trees of operations and terminals, but other encodings have also been proposed. For example, Graff et al.[9] use DAGs to encode python programs. Multi Expression Programming (MEP) [6] is a special kind of Genetic Programming [12] that has gained traction in recent years. In particular, MEP has been successfully applied to TSP [19], data prediction [29], software effort estimation [2], on-the-fly hyperparameter optimization for Evolutionary Algorithms [20] and digital circuit design [21]. Semantic genetic operators operate on the semantics of the solution space in contrast to traditional genetic operators, which operate in the encoding space, unaware of the semantics behind traditional bit-vector representations. Such semantic genetic operators also have had a lot of attention in recent literature. In particular, [23] uses semantic genetic operators in an NLP context to match sentences. [4, 8] applied semantic genetic operators to symbolic regression problems while other works focus more on advantageous mathematical properties in traditional numerical optimization settings for GP [28].

RDF data augmentation is a problem that greatly differs from use case to use case. For example, [1] considers the automatic extraction of interests from *Twitter* posts in order to enrich user profiles on the Social Web. On another use case, [5] applies a ranking method to the *Youtube* tag space in order to enrich datasets on the Linked Open Data (LOD) with links to *Youtube* videos. To our best knowledge, the only effort directed towards a general framework of RDF

[2] https://github.com/dice-group/deer

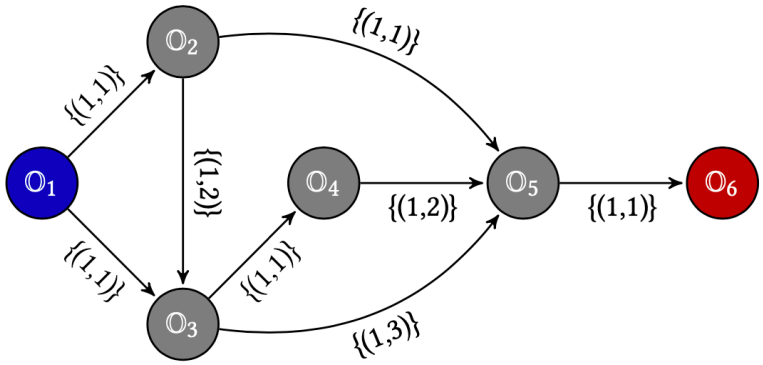

Fig. 1. Running example augmentation graph.

dataset augmentation together with an automatic learning approach can be found in [26], where the authors propose a framework and learning algorithm called DEER. We chose to build our approach based on the existing open source software of DEER, where we replaced the simple sequential pipelining approach with a more sophisticated one, based on arranging the enrichment functions in a DAG. In order to better distinguish our approach from the former, we will call atomic enrichment functions which allow for multiple inputs and multiple outputs *enrichment operators*.

3 APPROACH

We begin by providing a formal definition of augmentation operators and augmentation graphs. We then show how augmentation tables can be used to evaluate such graphs efficiently. Our learning algorithm then exploits the representation of graphs as multi-expressive augmentation tables to learn augmentation pipelines based on minimal examples.

3.1 Formal Model

RDF Dataset. An *RDF dataset* D is a set of triples $\{(s, p, o) \in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{R} \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})\}$, where $\mathcal{R}$ is the set of all RDF IRI resources, $\mathcal{B}$ is the set of all RDF blank nodes and $\mathcal{L}$ is the set of all RDF literals. We denote the set of all RDF datasets as $\mathcal{D}$.

Dataset Operators. A function $\mathbb{O}_{(n,m)} : \mathcal{D}^{n+1} \rightarrow \mathcal{D}^m$ is called a *dataset operator*. Intuitively, a dataset operator $\mathbb{O}_{(n,m)}$ processes n input datasets using another dataset C as configuration to produce m output datasets. We call n the in-degree and m the out-degree of $\mathbb{O}_{(n,m)}$ and will resort to writing just $\mathbb{O}$ when the lack of their specification will incur no loss of generality. Given integers $i \in [1, n]$, $j \in [1, m]$, we call the ith argument of $\mathbb{O}_{(n,m)}$ and the jth component in the output of $\mathbb{O}_{(n,m)}$ the in-port i and out-port j, respectively. The set of all dataset operators is denoted as $\mathbb{O}$. We specify the following naming scheme for dataset operators: $\mathbb{O}_{(0,1)}$ is called a *dataset emitter*, $\mathbb{O}_{(1,0)}$ is called a *dataset acceptor*, $\mathbb{O}_{(n>0,m>0)}$ is called an *augmentation operator* and $\mathbb{O}_{(n,1)}$ is called a *confluent augmentation operator*.

Augmentation Graphs. An augmentation graph $G = (\mathbf{O}, \mathbf{E}, \mathbf{L}, \mathbf{M})$ is a directed acyclic labeled multigraph where $\mathbf{O} \subseteq \mathbb{O}$ is a set of dataset operators, which act as vertices; $\mathbf{E} \subseteq \mathbf{O}^2$ is the set of edges, which represent flow of data; $\mathbf{L} : \mathbf{E} \rightarrow 2^{\mathbb{N} \times \mathbb{N}}$

is the edge labeling function, which defines mappings between dataset operator out-ports and in-ports for a given edge; and $\mathbf{M} : \mathbf{O} \rightarrow \mathcal{D}$ is a mapping from vertices to configuration datasets.

We call the subsets of vertices $\mathbf{O}_r \coloneqq \left\{\mathbb{O}_{(n,m)} \in \mathbf{O} \mid \nexists \left(\mathbb{O}', \mathbb{O}_{(n,m)}\right) \in \mathbf{E} \wedge n = 0 \wedge m = 1\right\}$ *root vertices*, $\mathbf{O}_l \coloneqq \left\{\mathbb{O}_{(n,m)} \in \mathbf{O} \mid \nexists \left(\mathbb{O}_{(n,m)}, \mathbb{O}'\right) \in \mathbf{E} \wedge n = 1 \wedge m = 0\right\}$ *leaf vertices* and $\mathbf{O}_i \coloneqq \mathbf{O} \backslash (\mathbf{O}_r \cup \mathbf{O}_l)$ *inner vertices*.

Note that, per definition all root vertices of an augmentation graph must be dataset emitters, all leaf vertices must be dataset acceptors and all inner vertices must be augmentation operators. In our running example augmentation graph in Figure 1, we coloured all root vertices blue and all leaf vertices red.

The intuition behind $\mathbf{L}$ is that, given $e = (\mathbb{O}_1, \mathbb{O}_2)$, we need to define which of $\mathbb{O}_1$'s out-ports map to which of $\mathbb{O}_2$'s in-ports. In other words, an entry of the label multiset $l \in \mathbf{L}(e) = (i, j)$ establishes a flow of data from $\mathbb{O}_1$'s ith out-port to $\mathbb{O}_2$'s jth in-port. The label function must not allow multiple mappings to the same in-port of the same dataset operator, which is formally expressed as:

$$\forall e_1 = (\mathbb{O}_1, \mathbb{O}_2), e_2 = (\mathbb{O}_3, \mathbb{O}_4) \in \mathbf{E} \colon \mathbb{O}_2 = \mathbb{O}_4$$
$$\rightarrow \forall l_1 = (i_1, j_1) \in \mathbf{L}(e_1) \nexists l_2 = (i_2, j_2) \in \mathbf{L}(e_2) \colon l_1 \neq l_2 \wedge j_1 = j_2$$

For instance, in our running example in Figure 1, the label set on the edge between $\mathbb{O}_4$ and $\mathbb{O}_5$ indicates that $\mathbb{O}_4$'s first output dataset is the second argument to $\mathbb{O}_5$.

To evaluate an augmentation graph, we first obtain the RDF datasets as output of the root vertices in $\mathbf{O}_r$. These datasets then flow through the graph as specified by the semantics we associated with the edge set $\mathbf{E}$ and the label multiset $\mathbf{L}$. Whenever a dataset operator $\mathbb{O}_{(n,m)} \in \mathbf{O}_i$ has received all its n input datasets, it is evaluated using $\mathbf{M}(\mathbb{O}_{(n,m)})$ as its last argument. The flow through the graph continues until eventually all vertices have been evaluated.

Categorization of Augmentation Graphs. We call an augmentation graph $G = (\mathbf{O}, \mathbf{E}, \mathbf{L}, \mathbf{M})$ **linear** iff $|\mathbf{O}_r| = |\mathbf{O}_l| = 1$ and $\forall \mathbb{O}_1, \mathbb{O}_2 \in \mathbf{O} : \mathbb{O}_1 \neq \mathbb{O}_2$ there exists at most a single path between $\mathbb{O}_1$ and $\mathbb{O}_2$; **semi-linear** iff $|\mathbf{O}_r| = |\mathbf{O}_l| = 1$ and there is a pair of vertices $u, v \in \mathbf{O}, u \neq v$ for which there exist multiple paths from u to v; **confluent** iff $|\mathbf{O}_r| > 1 \wedge |\mathbf{O}_l| = 1$; **inherently confluent** iff it is confluent and it only contains confluent augmentation operators, i.e., $\forall \mathbb{O}_{(n,m)} \in \mathbf{O} \colon m = 1$; and **general** otherwise. Note that inherently confluent augmentation graphs correspond to general augmentation graphs without loss of generality[3].

Augmentation Tables. An augmentation table $\mathbb{T}$ is a condensed linear representation for inherently confluent augmentation graphs based on column tables [13]. The idea behind this representation is that each row represents one dataset operator. We can go through this table from top to bottom and evaluate the dataset operators which correspond to a row i using only the results of rows 1 to $i - 1$. Since dataset acceptors produce no output, they are omitted in this representation for the sake of simplicity.

Let $G = (\mathbf{O}, \mathbf{E}, \mathbf{L}, \mathbf{M})$ be an inherently confluent augmentation graph. Moreover, let $N(\mathbf{O}) \coloneqq \max\left\{n \mid \mathbb{O}_{(n,m)} \in O\right\}$ denote *the maximum in-degree* in $\mathbf{O}$. An augmentation table is a table with $3 + N(\mathbf{O})$ columns and $|\mathbf{O}|$ rows, where the first column contains dataset operators, the second column contains configuration datasets and the third column contains the in-degrees of the dataset operators in the first column. The last $N(\mathbf{O})$ columns contain the indices of the rows used as input to the corresponding dataset operator. Given an augmentation table $\mathbb{T}$, we write $\mathbb{T}_i$ and $\mathbb{T}_{i,j}$ to refer to the ith row and the jth column in the ith row of $\mathbb{T}$, respectively. Applying this representation to our running example augmentation graph in Figure 1 gives the augmentation table depicted in Table 1.

[3] Any n-ary operator on a set is a finite composition of binary operators [27]

Table 1. Running example augmentation table.

$\mathbb{T}_1$:	$\mathbb{O}_1$	C_1	0	0	0	0
$\mathbb{T}_2$:	$\mathbb{O}_2$	C_2	1	1	0	0
$\mathbb{T}_3$:	$\mathbb{O}_3$	C_3	2	1	2	0
$\mathbb{T}_4$:	$\mathbb{O}_4$	C_4	1	3	0	0
$\mathbb{T}_5$:	$\mathbb{O}_5$	C_5	3	2	4	3

The algorithm for the computation of an augmentation table from a given inherently confluent augmentation graph is given in [13]. In essence, an augmentation table can be easily obtained from the adjacency matrix of a given Directed Acyclic Graph (DAG) $G = (\mathbf{V}, \mathbf{E})$ when a fixed indexing bijection $\phi\colon V \to \{1, \ldots, |\mathbf{O}|\}$ of its vertices is provided such that $\forall (v, v') \in \mathbf{E}\colon \phi(v) > \phi(v')$.

To obtain the results of a given augmentation table $\mathbb{T}$, it has to be evaluated from top to bottom. The last result is considered to be the result of the whole table. The evaluation result of a row $\mathbb{T}_i$ is denoted as $\mathbf{T}_i$. For example, the augmentation graph depicted in Table 1 is evaluated as follows:

$$\mathbf{T}_1 = \mathbb{O}_1\,(C_1)$$
$$\mathbf{T}_2 = \mathbb{O}_2\,(\mathbf{T}_1, C_2)$$
$$\mathbf{T}_3 = \mathbb{O}_3\,(\mathbf{T}_1, \mathbf{T}_2, C_3)$$
$$\mathbf{T}_4 = \mathbb{O}_4\,(\mathbf{T}_3, C_4)$$
$$\mathbf{T}_5 = \mathbb{O}_5\,(\mathbf{T}_2, \mathbf{T}_4, \mathbf{T}_3, C_5)$$

We call a row within an augmentation table an *output row*, if it is not used as input to a subsequent row. Note that output rows always correspond to dataset acceptors and that our previous definition of augmentation tables allows for only a single output row, as our augmentation tables must be isomorphic to inherently confluent augmentation graphs.

Multi-Expressive Augmentation Tables. A *multi-expressive augmentation table* is a generalized augmentation table that has more than one *output row*. Note that any row $\omega = \mathbb{T}_i$ in a multi-expressive augmentation table can be seen as an output row by just disregarding all rows below ω. Given such a *reference output row* ω in a multi-expressive augmentation table $\mathbb{T}$ we can derive a normal augmentation table $\mathbb{T}'$ in the following recursive fashion:

(1) We initialize $\mathbb{T}'$ as an empty augmentation table, $\mathcal{L}$ as an empty list and $\zeta\colon \mathbb{N} \to \mathbb{N}$ as a bijective mapping of index numbers that initially only contains the mapping $1 \mapsto 1$.
(2) Given $\omega = \mathbb{T}_i$, we add (ω, i) to the list $\mathcal{L}$
(3) Whenever adding a row $\mathbb{T}_j$ to $\mathcal{L}$, we recursively add all the row-index pairs $\left(\left(\mathbb{T}_{\mathbb{T}_{j,4}}, \mathbb{T}_{j,4}\right), \ldots, \left(\mathbb{T}_{\mathbb{T}_{j,\mathbb{T}_{j,3}}}, \mathbb{T}_{j,3+\mathbb{T}_{j,3}}\right)\right)$ to $\mathcal{L}$.
(4) We iterate through $\mathcal{L}$'s entries $(\mathbb{T}_k, j_k)\ \forall k \in [1, |\mathcal{L}|]$ in ascending order according to the index j_k.
(5) For each entry $(\mathbb{T}_k, j_k) \in \mathcal{L}$, we add $\mathbb{T}_k$ to $\mathbb{T}'$ and keep track of the mapping from the old row number to the new row number by adding the mapping $j_k \mapsto k$ to ζ.
(6) We then apply our index mapping ζ to the input definition column entries as $\mathbb{T}'_{x,y} \coloneqq \zeta\left(\mathbb{T}'_{x,y}\right) \forall 1 \leq x \in \leq |\mathbb{T}'|, y \in \left[4, 3 + \mathbb{T}'_{k,3}\right]$

Consider the multi-expressive augmentation table given in Table 2. When we select the 8th row as output reference row, we can reconstruct the normal augmentation table in Table 1 by following the algorithm defined above.

Table 2. Example of a multi-expressive augmentation table. Output rows appear with a gray background.

$\mathbb{T}_1$:	$\mathbb{O}_1$	C_1	0	0	0	0
$\mathbb{T}_2$:	$\mathbb{O}_2$	C_2	1	1	0	0
$\mathbb{T}_3$:	$\mathbb{O}_3$	C_3	2	1	2	0
$\mathbb{T}_4$:	$\mathbb{O}_4$	C_4	1	3	0	0
$\mathbb{T}_5$:	$\mathbb{O}_6$	C_6	2	3	1	0
$\mathbb{T}_6$:	$\mathbb{O}_7$	C_7	1	5	0	0
$\mathbb{T}_7$:	$\mathbb{O}_8$	C_8	2	5	3	0
$\mathbb{T}_8$:	$\mathbb{O}_5$	C_5	3	2	4	3
$\mathbb{T}_9$:	$\mathbb{O}_9$	C_9	1	7	0	0

3.2 Learning Algorithm

The problem under study to find an adequate enrichment graph for a given training example. Since for this work we do not wish to consider multi-objective learning, we restrict ourselves to learning the subclass of inherently confluent enrichment graphs. We will furthermore restrict our study to enrichment graphs where the maximum in-degree of the involved enrichment operators and the number of involved dataset emitters are at most two.

The core of our learning approach is a population-based $(\mu + \lambda)$ Multi Expression Programming (MEP) algorithm[4] that is able to learn the subclass of inherently confluent augmentation graphs. Our population consists of a fixed number $\mu + \lambda$ of multi-expressive augmentation tables that we also call *genotypes*. All genotypes have a fixed number r of rows. Tournament selection [16] with a tournament size of 3 and a selection probability of 0.75 is applied for determining the mating pool and for selecting the survivors. Additionally, we use 1-elitist selection [17] to avoid a decrease in fitness. Both the offspring and the survivors are subject to mutation. The offspring fraction $\frac{\mu}{\lambda}$, mutation probability σ and mutation rate ρ are hyperparameters that need to be determined experimentally. The algorithm will stop when either a perfect solution is found, a maximum number g of generations is exceeded or our convergence detection terminates it.

As the results of RDF dataset augmentation are commonly expected to have a regular structure, we can expect the output dataset to be decomposable into a number of subgraphs that are isomorphic up to a certain error w.r.t. some structural graph similarity measure. We therefore regard the training examples as a list of source Concise Bounded Descriptions (CBDs) $(D_{s_1}, \ldots, D_{s_n}) \in \mathcal{D}^n, n \in \{1, 2\}$ and a single target CBD $D_t \in \mathcal{D}$ of sufficient depth to representatively capture the desired augmentation. This is in accordance with the observation that a single pair of CBDs often suffices for the training of augmentation pipelines [26]. Note our choice to restrict the number of involved dataset emitters to at most two. This restriction does induce loss of generality as a larger number of dataset emitters can be supported by extending some of the formalism and definitions that follow.

In the following, we outline the details of our learning algorithm such as how we compute the initial population, the fitness function we employ and how we use it to evaluate genotypes, as well as a method to speed up learning called *genotype compaction* and the semantic genetic operators that we use for mutation and recombination.

Population Initialization. The population initialization procedure (see Listing 1) receives a set of prespecified dataset emitters $\mathcal{E}^{\perp}$ and a corresponding set of configuration datasets $C^{\perp}$ as well as the number of rows r it should generate. Initially, the configuration dataset for each of the augmentation operators is the empty dataset. We determine the actual configuration dataset of a given genotype row at evaluation time using *heuristic self-configuration*.

[4] μ is the *population size* and λ is the *recombination pool size*.

Algorithm 1: Random generation of multi-expressive augmentation tables.

```
Data: E^⊥, ;                                              // set of dataset emitters
Data: C^⊥, ;                                              // set of configurations
Data: r, ;                                                // number of rows to be generated
T ← ∅;
i ← 1;
while i ≤ r do
    if i < |E^⊥| then
        T_i ←(E_i^⊥, C_i^⊥, 0, 0, 0);
    else
        d ← rnd(1,max(2, i-1)) ;                          // rnd(a,b) returns a random integer in [a,b]
        O_(n,m) ← rndop(d) ;                              // rndop(d) returns a random enrichment operator with a maximum in-degree of d
        d ← n;
        if d = 1 then
            T_i ←(O, ∅, d, rnd(1,i-1));
        else
            T_i ←(O, ∅, d, rnd(1,i-1), rnd(1,i-1));
    i ← i + 1;
return T;
```

Heuristic Self-Configuration of Augmentation Operators. Due to the generality of our approach, we will not predefine any augmentation operators. Rather, augmentation operators are black boxes to the learning algorithm, where each augmentation operator expose a generic interfaces for heuristic self-configuration, i.e., a function that can heuristically compute the configuration dataset for the augmentation operator. Formally, let $\mathbb{T}$ be a genotype. $\mathbf{T}_i$ denotes the dataset resulting from the evaluation of a given genotype row $\mathbb{T}_i$. The input to the self-configuration procedure consists of the result of all input rows $\left(\mathbb{T}_{i,4}, \ldots, \mathbb{T}_{i,\mathbb{T}_{i,3}}\right)$ of $\mathbb{T}_i$ and the target training dataset D_t.

Fitness Function. Let $\mathcal{S}(D), \mathcal{P}(D), \mathcal{O}(D)$ denote the sets of subjects, predicates and objects[5] of a given dataset D, respectively. We define the fitness function of our learning algorithm as

$$\begin{aligned}\mathcal{F}\colon \mathcal{D}^2 &\to [0,1]\\ (D_r, D_t) &\mapsto \frac{1}{4}\left(F_1\left(D_r, D_t\right) + F_1\left(\mathcal{S}(D_r), \mathcal{S}(D_t)\right)\right.\\ &\quad \left. + F_1\left(\mathcal{P}(D_r), \mathcal{P}(D_t)\right) + F_1\left(\mathcal{O}(D_r), \mathcal{O}(D_t)\right)\right),\end{aligned}$$

where D_r is the resulted dataset from our algorithm, D_t is the target dataset from the training data and F_1 is the F_1-measure, $\mathcal{F}$ measures similarity in terms of the overlapping IRI resources in subjects, overlapping IRI resources in predicates and overlapping IRI resources and literals in objects as well as overlapping triples. The design of our fitness function is motivated by the two simple observations that (1) augmentation operators seldom modify all components of a given triple at once and (2) generally only the combination of multiple augmentation operators will lead to the generation of a triple as present in the target training dataset. Therefore, we designed our fitness function to detect any incremental improvements towards the end result. In contrast, DEER uses just the F_1-measure over the set of triples as fitness function, which completely fails to measure incremental improvements in partial solution quality.

[5]Note that we exclude blank nodes from $\mathcal{S}(D)$ and $\mathcal{O}(D)$.

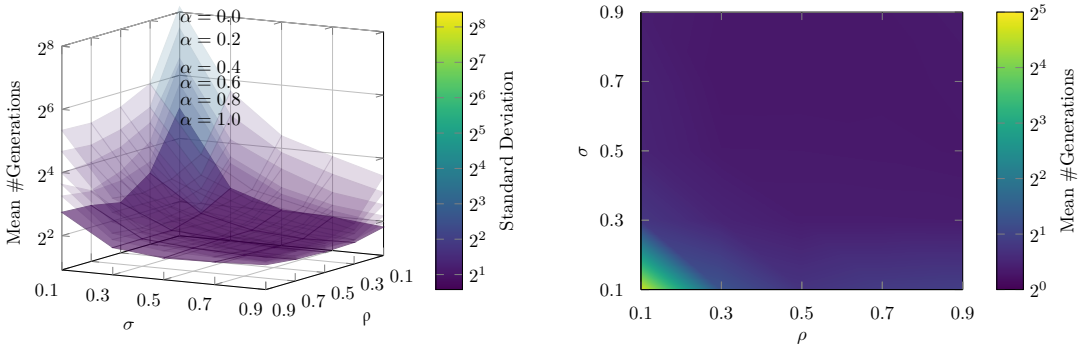

Fig. 2. **Hyperparameter Optimization Results.** The experiment was repeated one thousand times. We measured the average generations to termination and the number of perfect runs. On the right hand side we see a more detailed heat map of the best performing plane on the left.

MEP-based Evaluation of Genotypes. As our genotypes are represented as multi-expressive augmentation table $\mathbb{T}$, they correspond to multiple phenotypes. i.e., normal augmentation tables. Therefore, we regard each row in $\mathbb{T}$ as a potential output row. The evaluation of $\mathbb{T}$ is done row-wise and in a similar way to the evaluation of a normal augmentation table with the notable exception that we apply self-configuration to obtain the configuration dataset.

We compute the fitness for the result of each row using $\mathcal{F}$. The *effective phenotype* of a given genotype is the normal augmentation table obtained by reconstruction where the fittest row is taken as reference output row. Likewise, we define a genotypes *effective fitness* as the fitness of its effective phenotype.

Genotype Compaction. The idea of genotype compaction is that we can speedup learning by restructuring genotypes $\mathbb{T}$ in a way such that we move the effective phenotype $\mathbb{T}_p$ of $\mathbb{T}$ to the front of the compacted genotype $\mathbb{T}'$.

Let $|\mathbb{T}|$ denote the number of rows in a given genotype $\mathbb{T}$. Furthermore, let $\bot(\mathbb{T})$ denote the first one or two rows corresponding to dataset emitters in a given genotype $\mathbb{T}$.

First, we derive the effective genotype $\mathbb{T}_p$ of $\mathbb{T}$. We define $\top(\mathbb{T}) = \mathbb{T} \backslash \bot(\mathbb{T})$. The compacted genotype $\mathbb{T}'$ is then defined row-wise by Equation 1 where `rndrow()` means that in this case we generate a new row randomly in a way that is reflected by lines 12 through 19 of Listing 1.

$$\begin{aligned}
\mathbb{T}'_i &:= \bot(\mathbb{T})_i && \iff 1 \leq i \leq |\bot(\mathbb{T})| \\
\mathbb{T}'_i &:= \top(\mathbb{T}_p)_{i-|\bot(\mathbb{T})|} && \iff |\bot(\mathbb{T})| < i \leq |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| \\
\mathbb{T}'_i &:= \texttt{rndrow()} && \iff |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| < i \leq r
\end{aligned} \tag{1}$$

We apply genotype compaction randomly with a probability of 0.5 to the whole population in the beginning of each generation.

To further minimize the size of the compacted genotype, we detect and remove **no-op rows**, i.e., rows for which the output is identical to one of the inputs. Formally, given a row $\mathbb{T}_i$, its input datasets $I = \left(\mathbf{T}_{\mathbb{T}_{i,4}}, \ldots, \mathbf{T}_{\mathbb{T}_{i,3+\mathbb{T}_{i,3}}}\right)$ and its output dataset $\mathbf{T}_i$, we call $\mathbb{T}_i$ a no-op row iff $\exists D \in I : \mathbf{T}_i = D$.

Subgraph Merging Crossover. The idea behind our subgraph merging crossover is to combine two genotypes $\mathbb{T}, \mathbb{T}'$ in a way that increases the probability of learning a graph where both $\mathbb{T}$ and $\mathbb{T}'$ are part of its effective phenotype. This is motivated by the observation that two well-performing and sufficiently distinct inherently confluent augmentation graphs could be merged into one by redirecting the vertices that have edges to their dataset acceptors to a new vertex with in-degree two.

Let $\mathbb{T}_p, \mathbb{T}'_p$ be the effective phenotypes of $\mathbb{T}, \mathbb{T}'$, respectively. We apply this operator only if $|\bot(\mathbb{T})|+|\top(\mathbb{T}_p)|+|\top(\mathbb{T}'_p)| \leq r$ holds, otherwise we default to the single-point crossover.

The subgraph merging crossover needs to return two child genotypes to guarantee that the population size stays constant. To this end, we first compute both children equally, but then we fill the $r - |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)|$ remaining rows randomly for each child genotype.

If $r - |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)| \geq 1$, we will also insert a row that merges the two subgraphs in $\mathbb{T}_p$ and $\mathbb{T}'_p$ with a probability of 0.25. To this end, we will randomly aquire an augmentation operator with an in-degree of two and set the input columns to the positions of the output rows of $\mathbb{T}_p$ and $\mathbb{T}'_p$ in the newly created child genotype.

Let $\mathbb{O}^{\text{merge}}_{(2,1)}$ be a randomly chosen augmentation operator with in-degree two. Formally, a child genotype $\mathbb{T}^c$ is defined row-wise as

$$
\begin{aligned}
\mathbb{T}^c_i &:= \bot(\mathbb{T})_i && \iff && 1 \leq i \leq |\bot(\mathbb{T})| \\
\mathbb{T}^c_i &:= \top(\mathbb{T}_p)_{i-|\bot(\mathbb{T})|} && \iff && |\bot(\mathbb{T})| < i \leq |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| \\
\mathbb{T}^c_i &:= \top(\mathbb{T}'_p)_{i-|\bot(\mathbb{T})|-|\top(\mathbb{T}_p)|} && \iff && |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| < i \leq |\bot(\mathbb{T})| \\
& && && + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)| \\
\mathbb{T}^c_i &:= \left(\mathbb{O}^{\text{merge}}_{(2,1)}, \emptyset, 2, |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)|, |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)|\right) \\
& && \iff && i = |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)| \\
& && && + 1 \leq r \wedge \text{rnd}(1,4) = 1 \\
\mathbb{T}^c_i &:= \texttt{rndrow()} && \iff && i = |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)| \\
& && && + 1 \leq r \wedge \text{rnd}(1,4) > 1 \\
\mathbb{T}^c_i &:= \texttt{rndrow()} && \iff && |\bot(\mathbb{T})| + |\top(\mathbb{T}_p)| + |\top(\mathbb{T}'_p)| \\
& && && + 1 < i \leq r
\end{aligned}
$$

Precondition Broadcasting Mutation. The idea of the precondition broadcasting mutation is to define a context-sensitive measure of applicability for augmentation operators. To this end, the augmentation operators need to provide an additional method $\eth$, which accepts the same kind of arguments as the self-configuration method. It then returns a real value between 0 and 1, which expresses the applicability of the augmentation operator, given the input and target datasets.

The precondition broadcasting mutation works as follows. When a row $\mathbb{T}_i$ has been chosen for mutation in a given genotype $\mathbb{T}$, we evaluate $\mathbb{T}$ up to $\mathbb{T}_i$ and obtain the evaluation results of its input dataset(s). Subsequently we *broadcast* the input dataset(s) together with the target training dataset to the applicability methods $\eth$ of all available augmentation operators. After the operators have returned their result, we initialize a roulette wheel selection where the selection probabilities of the roulette wheel are proportional to the relative applicabilities of the operators. Finally, we set $\mathbb{T}_{i,1}$ to the winner of the roulette wheel.

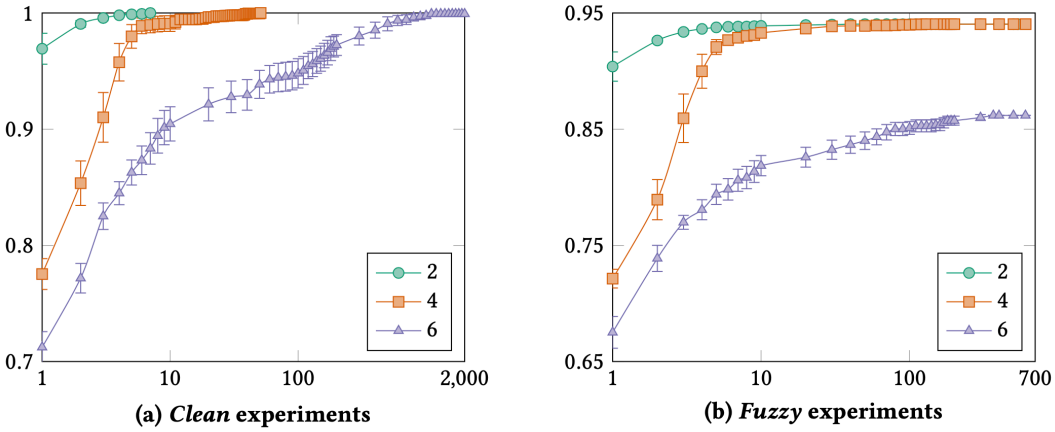

Fig. 3. **Performance Evaluation Results.** The graphs show the achieved mean fitness over 50 repetitions (y axis) vs. the number of generations on a logarithmic scale (x axis).

We chose the name *precondition broadcasting* for this method, as the input datasets to an augmentation operator can be regarded as its preconditions w.r.t. applicability.

Convergence Detection Mechanism. Our convergence detection mechanism is used to (1) detect convergence of our population and (2) prohibit convergence into a local optima by temporarily adjusting σ and ρ. We keep track of the best fitness value and the standard deviation of fitness values in our generations to detect convergence. We assert convergence if the best fitness value has not changed and the standard deviation of fitness values has been under a certain threshold s_{min} for the last ten generations.

When convergence is detected, we attempt to escape the potential local optima by temporarily setting $\sigma = \rho = 1$. In the following generations, we lower them again by multiplication with a constant factor until they reach their initial values. The algorithm finally terminates if, after a fixed number of convergence detections and escape attempts, no perfect solution is found.

4 EVALUATION

Research Questions. **(Q_1)** What is the set of optimal hyperparameters, if any? **(Q_2)** How does ADAGIO perform compared to the only comparable approach DEER? **(Q_3)** What are the performance characteristics of ADAGIO when used for DAG-shaped augmentation tasks?

Hardware. All experiments were carried out on a 64-core 2.3 GHz server running *OpenJDK* 64-Bit Server 1.8.0_151 on *Ubuntu* 16.04.3 LTS. Each experiment was assigned 128 GB RAM.

Augmentation Operator Setup. We reimplemented the set of augmentation operators used in DEER [26] for the purpose of this evaluation with the notable difference that the linking operator is a binary operator in our implementation[6]. Additionally, we implemented caching for augmentation operators that rely on external webservices, such as

[6]This means it is possible to apply instance matching to two input datsets, whereas the linking operator in DEER was unary, requiring another external configuration.

Table 3. **DEER Comparison Results**. Reproduced experiments from DEER. t_l is the learning time of one execution in seconds, F_1 is the F_1-measure and $\max g$ is the maximum generation after which a perfect solution was found for ADAGIO.

Dataset	ADAGIO			DEER	
	$\max g$	t_l	F_1	t_l	F_1
$M^1_{DBpedia}$	1	**0.32**	1.00	1.3	1.00
$M^2_{DBpedia}$	1	**0.32**	**1.00**	0.2	0.99
$M^3_{DBpedia}$	3	**0.33**	**1.00**	6.1	0.99
$M^4_{DBpedia}$	1	**0.32**	**1.00**	0.7	0.99
$M^5_{DBpedia}$	2	**0.32**	1.00	0.7	1.00
$M^1_{DrugBank}$	1	**0.01**	**1.00**	0.1	0.99
$M^2_{DrugBank}$	2	**0.02**	**1.00**	0.1	0.99
$M^1_{Jamendo}$	1	0.33	1.00	**0.1**	1.00

dereferencing or named entity recognition, so that we do not have to unnecessarily query these services a lot of times within our learning loop[7]. Finally, we introduced a binary merge operator that accepts two datasets and returns their union.

4.1 Hyperparameter Optimization

To determine the best set of values for our hyperparameters *offspring fraction* $\alpha = \frac{\mu}{\lambda}$, *mutation probability* σ and *mutation rate* ρ, we used an incrementally explorative method. That is, we ran a series of grid searches on augmentation tasks with increasing difficulty and used our insights from previous runs to fine-tune the next. Due to lack of space, we only report the final grid search results in Figure 2. These results suggest that the best set of hyperparameters are $\alpha = 1$, $\sigma = 0.5$ and $\rho = 0.5$. It is notable that for these hyperparameters we achieve 100% perfect results despite enabling our convergence detection mechanism. Another interpretation of the data worth mentioning is that the average number of generations to termination is lower for some other settings, but these settings did not achieve a similar number of perfect results.

4.2 Comparison with State of the Art

For this series of experiments, we emulated the 8 original experiments undertaken in [26], using the three original datasets, i.e., *DBpedia*[8], *DrugBank*[9] and *Jamendo*[10]. We then faithfully recreated the augmentation pipelines in our system. All experiments were repeated 50 times. The results of our comparison are shown in Table 3. ADAGIO constructed a perfect linear augmentation graph in under 4 iterations for all experiments. Moreover, ADAGIO outperformed DEER in terms of learning time in 7 out of 8 experiments with a mean speedup of 6.99. Note that we cannot explain the missing

[7] Please note that this does not introduce bias into our comparison with DEER, since DEER also evaluate each operator only once in every iteration due to its deterministic design based on refinement operators.
[8] *DBpedia* is a subset of `AdministrativeRegion` class instances from the multi-domain knowledge graph DBpedia (https://dbpedia.org).
[9] *DrugBank* is the Linked Data version of the DrugBank database, which is a repository of almost 5000 FDA-approved small molecule and biotech drugs, available at http://datahub.io/dataset/fu-berlin-drugbank.
[10] *Jamendo* contains a large collection of music related information about artists and recordings

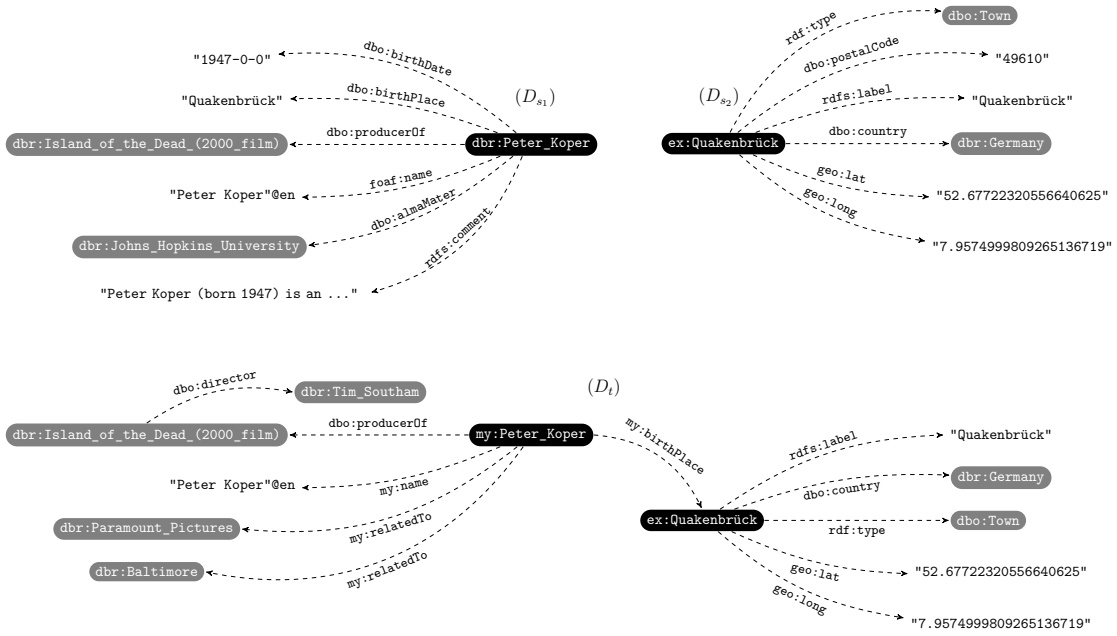

Fig. 4. **Performance Evaluation Training Data.** (D_{s_1}) shows the CBD used as the first source training dataset, (D_{s_2}) shows the CBD used as the second source training dataset and (D_t) shows the target training dataset, which we generated from (D_{s_1}) and (D_{s_2}) using an augmentation graph with 6 augmentation operators.

1% in some of the F_1-measure reported in the original DEER paper and that, given the authors methodology, it is most likely that this is due to floating point errors in their F_1-measure calculations. ADAGIO was able to parallelize the target augmentation graph for the task $M^3_{DBpedia}$, thereby achieving a speedup of 23% compared to the equivalent linear augmentation graph when applying them to the whole dataset. Altogether, these results suggest that our approach outperforms the current state of the art for learning augmentation pipelines.

4.3 Performance Evaluation

We ran two sets of experiments (*clean* and *fuzzy*) with two real-world datasets extracted from DBpedia[11] to measure performance on DAG-shaped augmentation tasks. We obtained one dataset D_{s_1} about 2,230 persons born in Germany and another dataset D_{s_2} about 1,660 towns in Germany. The training data for the experiments was generated using three manually designed augmentation tasks with 2, 4 and 6 augmentation operators for each series of experiments, respectively. For the *clean* experiments, the training data was left untouched, resulting in tasks that ADAGIO is theoretically able to solve perfectly, i.e., with $\mathcal{F} = 1$. For the *fuzzy* experiments, we randomly altered the training data in a way that the augmentation operators in ADAGIO are not able to solve the tasks perfectly.

For each task, we selected a maximal training example as corresponding CBDs from D_{s_1}, D_{s_2} and D_t from the training data, for which all components of the augmentation graph contributed to the augmentation in D_t. ADAGIO was then trained on this maximal example using a genotype size of $r = 10$ and a population size of 30. Figure 4 shows the training

[11] https://dbpedia.org

data for the the hardest *clean* task. We measured the performance of ADAGIO by averaging the best fitness found in each generation over a series of 50 repetitions for each task, terminating each task after at most 2, 000 generations. Moreover, we measured run times of our algorithm. We computed 95%-confidence intervals using Student's t-distribution to account for variance.

The *clean* results in Figure 3a suggest that our approach solved this task for the two simpler tasks in less than 52 and 7 generations, respectively. Note that this means ADAGIO *always* found the perfect solution for the simpler tasks. The hardest tasks fitness also grows rapidly over the first 10 generations, reaching an overall average fitness of *over* 0.99 *within a* 95%*-confidence interval of* ±0.6% *in under 500 generations*. The average runtime reported was 16 seconds *for the hardest task*.

The *fuzzy* results in Figure 3b show that our convergence detection is indeed working, as all tasks terminate before the maximum number of 2, 000 generations is reached. Moreover, we see that the solutions of all executions tend to reach the global maximum before termination as the 95%-confidence interval stays at 0% for all generations starting at the 40th, 130th and 366th generation for the 2-, 4- and 6-augmentation operator tasks, respectively.

5 CONCLUSION AND FUTURE WORK

In this paper, we present ADAGIO, an efficient algorithm based on genetic programming for learning knowledge graph augmentation graphs. We experimentally identified the set of optimal hyperparameters for our algorithm: *offspring fraction* $\alpha = 1.0$, *mutation probability* $\sigma = 0.5$ and *mutation rate* $\rho = 0.5$, which answers our first research question **(Q_1)**. To answer **(Q_2)**, our experiments show that our approach performs at least as well as the previous state of the art. On real-world datasets, ADAGIO reported an average execution time of 16 seconds and a mean solution quality of 99% within a 95%-confidence interval of ±0.6% after 500 generations, thus giving an answer for **(Q_3)**.

In future work, we will investigate research relating to the automated augmentation of RDF datasets in three directions. (I) We will develop a sound theory of self-configuration heuristics, where we intend to use graph embedding techniques and variations of approximate solutions to the subgraph isomorphism problem. (II) We also intend to extend the class of augmentation graphs for which we investigate learning methods, eventually developing multi-objective algorithms. Such algorithms could be used to automate the provision of RDF knowledge graphs for different types of consumers given the same input datasets, thereby exploiting parallelism. (III) Finally, we will research methods for improving the scalability of our approach in a big data setting.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (GZ: SFB 901/3) under the project number 160364472. This work was also supported by the German Federal Ministry of Economics and Climate Protection (BMWK) project RAKI (GA no. 01MD19012D) and the EU H2020 project KnowGraphs (GA no. 860801).

REFERENCES

[1] Fabian Abel, Qi Gao, Geert-Jan Houben, and Ke Tao. 2011. Semantic Enrichment of Twitter Posts for User Profile Construction on the Social Web. In *The Semanic Web: Research and Applications*, Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Pan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 375–389. https://doi.org/10.1007/978-3-642-21064-8_26

[2] Najla Akram, Al-Saati, and Taghreed Riyadh Alreffaee. 2018. Using Multi Expression Programming in Software Effort Estimation. *CoRR* abs/1805.00090 (2018). https://dblp.org/rec/journals/corr/abs-1805-00090

[3] Volha Bryl and Christian Bizer. 2014. Learning Conflict Resolution Strategies for Cross-Language Wikipedia Data Fusion. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 1129–1134. https://doi.org/10.1145/2567948.2578999

[4] Qi Chen, Bing Xue, Yi Mei, and Mengjie Zhang. 2017. Geometric Semantic Crossover with an Angle-Aware Mating Scheme in Genetic Programming for Symbolic Regression. In *EuroGP (Lecture Notes in Computer Science, Vol. 10196)*. Springer, 229–245. https://doi.org/10.1007/978-3-319-55696-3_15

[5] Smitashree Choudhury, John G. Breslin, and Alexandre Passant. 2009. Enrichment and Ranking of the YouTube Tag Space and Integration with the Linked Data Cloud. In *The Semantic Web - ISWC 2009*, Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 747–762. https://doi.org/10.1007/978-3-642-04930-9_47

[6] Cândida Ferreira. 2001. Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. *Complex Systems* 13, 2 (2001). https://dblp.org/rec/journals/compsys/Ferreira01

[7] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. *Auto-sklearn: Efficient and Robust Automated Machine Learning*. Springer International Publishing, Cham, 113–134. https://doi.org/10.1007/978-3-030-05318-5_6

[8] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. Semantics-Based Crossover for Program Synthesis in Genetic Programming. In *Artificial Evolution (Lecture Notes in Computer Science, Vol. 10764)*. Springer, 58–71. https://doi.org/10.1007/978-3-319-78133-4_5

[9] M. Graff, E. S. Tellez, S. Miranda-Jiménez, and H. J. Escalante. 2016. EvoDAG: A semantic Genetic Programming Python library. In *2016 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*. 1–6. https://doi.org/10.1109/ROPEC.2016.7830633

[10] Souleiman Hasan, Edward Curry, Mauricio Banduk, and Sean O'Riain. 2011. Toward situation awareness for the semantic sensor web: Complex event processing with dynamic linked data enrichment, In Proceedings of the 4th International Conference on Semantic Sensor Networks-Volume 839. *Semantic Sensor Networks*, 69–82.

[11] Holger Karl, Dennis Kundisch, Friedhelm Meyer auf der Heide, and Heike Wehrheim. 2019. A Case for a New IT Ecosystem: On-The-Fly Computing. *Business & Information Systems Engineering* 62, 6 (Dec 2019), 467–481. https://doi.org/10.1007/s12599-019-00627-x

[12] John R Koza. 1990. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Vol. 34. Stanford University, Department of Computer Science Stanford, CA.

[13] Vladimír Kvasnièka and Jiøí Pospíchal. 1998. Simple Implementation of Genetic Programming by Column Tables. In *Soft Computing in Engineering Design and Manufacturing*, P. K. Chawdhry, R. Roy, and R. K. Pant (Eds.). Springer London, London, 48–56. https://doi.org/10.1007/978-1-4471-0427-8_6

[14] Jose Lazaro Martinez-Rodriguez, Aidan Hogan, and Ivan López-Arévalo. 2020. Information extraction meets the Semantic Web: A survey. *Semantic Web* 11 (2020), 255–335. https://doi.org/10.3233/sw-180333

[15] Pablo N. Mendes, Hannes Mühleisen, and Christian Bizer. 2012. Sieve: Linked Data Quality Assessment and Fusion. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops* (Berlin, Germany) *(EDBT-ICDT '12)*. Association for Computing Machinery, New York, NY, USA, 116–123. https://doi.org/10.1145/2320765.2320803

[16] Brad L. Miller and David E. Goldberg. 1995. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems* 9, 3 (1995). https://dblp.org/rec/journals/compsys/MillerG95

[17] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT Press. https://dblp.org/rec/books/daglib/0019083

[18] Markus Nentwig, Michael Hartung, Axel-Cyrille Ngonga Ngomo, and Erhard Rahm. 2017. A survey of current Link Discovery frameworks. *Semantic Web* 8, 3 (2017), 419–436. https://doi.org/10.3233/SW-150210

[19] Mihai Oltean and Dumitru Dumitrescu. 2015. Evolving TSP heuristics using Multi Expression Programming. *CoRR* abs/1509.02459 (2015). https://dblp.org/rec/journals/corr/OlteanD15

[20] Mihai Oltean and Crina Groşan. 2003. Evolving Evolutionary Algorithms Using Multi Expression Programming. In *Advances in Artificial Life*, Wolfgang Banzhaf, Jens Ziegler, Thomas Christaller, Peter Dittrich, and Jan T. Kim (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 651–658.

[21] Mihai Oltean and Crina Grosan. 2004. Evolving Digital Circuits using Multi Expression Programming. In *Evolvable Hardware*. IEEE, IEEE Computer Society, 87–90. https://doi.org/10.1109/EH.2004.1310814

[22] Heiko Paulheim. 2017. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web* 8, 3 (2017), 489–508. https://doi.org/10.3233/SW-160218

[23] Dunlu Peng, Shaohong Wu, and Cong Liu. 2019. MPSC: A Multiple-Perspective Semantics-Crossover Model for Matching Sentences. *IEEE Access* 7 (2019), 61320–61330. https://doi.org/10.1109/ACCESS.2019.2915937

[24] Muhammad Saleem, Maulik R Kamdar, Aftab Iqbal, Shanmukha Sampath, Helena F Deus, and Axel-Cyrille Ngonga Ngomo. 2014. Big linked cancer data: Integrating linked tcga and pubmed. *Web Semantics: Science, Services and Agents on the World Wide Web* 27 (2014), 34–41. https://doi.org/10.1016/j.websem.2014.07.004

[25] Andreas Schultz, Andrea Matteini, Robert Isele, Christian Bizer, and Christian Becker. 2011. LDIF -Linked Data Integration Framework. In *Proceedings of the Second International Conference on Consuming Linked Data - Volume 782* (Bonn, Germany) *(COLD'11)*. CEUR-WS.org, Aachen, DEU, 125–130.

[26] Mohamed Ahmed Sherif, Axel-Cyrille Ngonga Ngomo, and Jens Lehmann. 2015. Automating RDF Dataset Transformation and Enrichment. In *The Semantic Web. Latest Advances and New Domains*, Fabien Gandon, Marta Sabou, Harald Sack, Claudia d'Amato, Philippe Cudré-Mauroux, and Antoine Zimmermann (Eds.). Springer International Publishing, Cham, 371–387.

[27] Wacław Sierpiński. 1945. Sur les fonctions de plusieurs variables. *Fundamenta Mathematicae* 33, 1 (1945), 169–173. http://eudml.org/doc/213088

[28] Ranyart Rodrigo Suárez, Mario Graff, and Juan J. Flores. 2015. Semantic Crossover Operator for GP based on the Second Partial Derivative of the Error Function. *Research in Computing Science* 94 (2015), 87–96. https://dblp.org/rec/journals/rcs/SuarezGF15

[29] Qingke Zhang, Bo Yang, Lin Wang, and Jianzhang Jiang. 2013. An improved multi-expression programming algorithm applied in function discovery and data prediction. *IJICT* 5, 3/4 (2013), 218–233. https://doi.org/10.1504/IJICT.2013.054952