Hyperparameter Optimization for AST Differencing

Matias Martinez, Jean-Rémy Falleri, Martin Monperrus

Hyperparameter Optimization for AST Differencing

Matias Martinez, Jean-Rémy Falleri, Martin Monperrus

***Abstract*—Computing the differences between two versions of the same program is an essential task for software development and software evolution research. AST differencing is the most advanced way of doing so, and an active research area. Yet, AST differencing algorithms rely on configuration parameters that may have a strong impact on their effectiveness. In this paper, we present a novel approach named `DAT` (*Diff Auto Tuning*) for hyperparameter optimization of AST differencing. We thoroughly state the problem of hyper-configuration for AST differencing. We evaluate our data-driven approach `DAT` to optimize the edit-scripts generated by the state-of-the-art AST differencing algorithm named GumTree in different scenarios. `DAT` is able to find a new configuration for GumTree that improves the edit-scripts in 18.7% of the evaluated cases.**

I. Introduction

The computation of the differences between two versions of the same program is an essential task for software development. It is done on a daily basis when developers share and discuss code changes, through pull-requests in development platforms (eg. Github, Gitlab, etc) or through patches over mailing lists (patches on the Linux Kernel Mailing List). The most common differencing strategy of developer tools is line-based: a diff is a list of *chunks* where each chunk is a set of added and/or removed lines. For example, `git` provides the widely used command 'git-diff' for computing a line-based diff script. To push this state-of-practice further, there is an active line of research on differencing algorithms that work at the level of Abstract Syntax Trees (AST) instead of lines [8], [19], [15], [24], [14], [21]. In this case, the differences between two program versions are expressed in terms of actions over nodes from the trees under comparison (e.g., Insert, Remove, Update and Move nodes). AST differencing has been shown to be significantly better than line differencing for a number of tasks [15], [13].

AST differencing is of great importance not only for practitioners but also for software engineering research. AST differencing algorithms have been extensively used to study software evolution [27], [18], [48]. Among the key usages of AST differencing are pattern inference for automated program repair [32], [30], [31], bug fix analysis [53], [45], [34], code recommendation [36], [37], [40]. Consequently, it is of great importance for both practitioners and researchers to have reliable AST differencing tools.

Virtually all AST differencing algorithms can be configured in some way. That is, they have *hyperparameters* that control and guide the differencing process. For example, the state-of-the-art differencing tool GumTree has one hyperparameter called `BUM_SMT` for setting a minimum similarity threshold between two trees. This parameter has a default value of 0.5. Changing that value impacts on the produced edit-scripts.

Unfortunately, as shown in previous research [16], [12], state-of-the-art tools such as GumTree often generate sub-optimal diffs, that is, edit-scripts that contain *spurious* AST operations. *In this paper, we show that those hyperparameters matter and that one can improve AST differencing performance by finding hyperoptimized values.*

In this paper, we present a novel approach named `DAT` (*Diff Auto Tuning*) for hyperparameter optimization of AST differencing. Given an AST differencing algorithm, `DAT` finds the optimal parameter configuration given a set of file-pairs to be compared. `DAT` requires neither ground-truth nor labeled file pairs, the optimization is guided by a widely accepted quality metric for AST differencing [15], [23], [22], [24], [21], [14], [35]: the length of the edit-script. To our knowledge, we are the first to thoroughly study the problem of suboptimal configuration for AST differencing and to propose a data-driven approach to solve this problem.

We evaluate `DAT` by searching for the best configurations of the AST differencing algorithm GumTree. In particular, we evaluate it in two scenarios. The first scenario consists of searching for the configuration that works best on a set of file pairs. In this case, `DAT` does *global hyperparameter optimization* The configuration found can be used as the new default configuration when a practitioner aims to apply diffs in a new programming language or AST meta-model. Secondly, we evaluate the ability of `DAT` to find the best configuration for a particular case (i.e., a single file-pair). In this case, `DAT` does *local hyperparameter optimization*. Our evaluation consists of executing those optimizations on more than 30,000 pairs of Java files extracted from real-world commits of open-source Java projects.

Our experimental results show that *1)* `DAT` is able to find a hyperoptimized configuration which produces shorter edit-scripts for 18.7% of the cases using the JDT meta-model (which is the meta-model used by default by GumTree). *2)* Local hyperparameter optimization is effective, as it allows finding shorter edit-scripts than the default configuration in up to ≈22% of cases. *3)* TPE, a Bayesian optimization approach, is an effective optimization method which can be used by `DAT` on both global and local optimizations.

To sum up, the contributions of this paper are:

Matias Martinez is with Université Polytechnique Hauts-de-France, France. E-mail: matias.martinez@uphf.fr

Jean-Rémy Falleri is with University of Bordeaux, France and Institut Universitaire de France, France

Martin Monperrus is with KTH Royal Institute of Technology, Sweden. Email: monperrus@kth.se

- The novel problem statement of hyperparameter optimization for AST differencing.
- An original data-driven approach, called `DAT` for hyperparameter optimization in the context of AST differencing.
- An original and sound protocol for studying the performance of AST differencing, with cross-validation and statistical validation. The results of this experiment are available at: https://github.com/martinezmatias/dat-experimental-results.
- A publicly available tool for hyper-optimization of AST differencing. It natively supports the popular GumTree AST differencing engine and provides extension points to integrate other differencing tools: https://github.com/martinezmatias/diff-auto-tuning.

The paper continues as follows. Section II presents three cases in which state-of-the-art AST differencing produces incorrect outputs. Section III presents the terminology used in the paper. Section IV presents `DAT`, our hyperparameter optimization approach for AST differencing. Section V presents the methodology used to evaluate our approach. Section VI presents the results of the evaluation. Section VII presents the threats to validity. Section VIII presents the related work. Section IX concludes the paper.

II. Motivation

Previous work has revealed that state-of-the-art AST differencing algorithms generate inaccurate mappings, which impacts on the quality of the edit-scripts generated. For example, Fan et al. [16] show that GumTree [15] generates inaccurate mappings for 20%-29% of the file pairs analyzed.

In this Section, we present two cases for which GumTree with its default configuration produces incorrect or non-optimal edit-scripts. Later, in Section VI, we will show how autotuning GumTree using our approach `DAT` allows GumTree to produce more understandable edit-scripts for these two cases.

A. Case 1: Spurious Add-Remove

Diff algorithms such as GumTree can produce edit-scripts with spurious edits [12]. A recurrent case of these edits is presented in Figure 1, which shows the `Interpreter.java file` from the Log4J project. It presents the version from commit a04d92 (right part) and its previous version (left part). The figure also shows in a colored box the code affected by the edits obtained by GumTree after computing the diff of those two files, using the default configuration and the JDT metamodel. Each color represents a different edit type (red corresponds to remove, green to insert). The edit-script includes, in total, six edits. Three edits remove tree tokens (`public`, `class` and `Interpreter`), each one represented by an AST node (left part, in red), and the other three edits insert the same three tokes (left part, in green). All these are spurious edits. GumTree should not generate any of these six edits. As we discuss later in this paper, GumTree using the best configuration found by `DAT` does not produce those edits.

B. Case 2: Including Updates in the Edit-script

We now focus on the differences that GumTree detects between the version of the file `PanelWindowContainer.java` from the jEdit project in commit 6867bd and its previous version.

The changes made by the developer were two: updates the type of variables `dockables` and `buttons` from `ArrayList` to `List`. However, the diff produced by GumTree looks like that produced by a line-based diff such as GNU diff: The edit-script removes the fields from the left part, adds the fields from the right part (using the new field's type `List`), and moves the tokens related to the field names. This edit strip does not clearly express the changes done by the developers (i.e., update two nodes), and introduces spurious edits (e.g., the remove and add of tokens 'private').

III. Terminology

In this section, we present the key terminology related to our contributions.

AST Differencing Algorithm: computes the differences between two ASTs (Abstract Syntax Tree). For example, ChangeDistiller [19] and GumTree [15] are two popular AST diff algorithms. Typically, an AST diff algorithm has two input parameters, the two ASTs (AST_{left} and AST_{right}) generated from two source code files (F_{left} and F_{right}, respectively). Those ASTs are modeled using a *meta-model* (defined below). Finally, the AST algorithm outputs the differences between the two ASTs in the form of an *edit-script* (defined below). An AST differencing algorithm is implemented in a tool, for example, GumTree [15] is implemented in GumTreeDiff.[1]

Edit-script: is the output of an AST diff algorithm. It represents the result of the comparison between the two ASTs given as input parameters. The edit-script is a sequence of edit operations applied to the AST nodes of AST_{left}. The operations are typically typed as {insert, remove, update, move}, and are meant to represent the transformation from AST_{left} into AST_{right}.

Matcher: For computing an edit-script between two ASTs, AST_{left} and AST_{right}, it is necessary to try to match (i.e., *link*) some tree nodes of AST_{left} with nodes from AST_{right}. The algorithm that executes this task is known as *matcher*. The criterion for matching two nodes depends on the matching algorithm, and can consider, for instance, the node's type and label, the topology, etc. The list of matched and unmatched nodes is then used for deducing an edit-script. For instance, those nodes from AST_{left} and AST_{right} that could not be matched correspond to removed or inserted nodes, respectively.

AST Meta-model: defines the structure of an AST. In particular, the meta-model describes: *a)* the possible types of AST nodes (e.g, invocations, assignments, methods); *b)* the attributes of each node type (e.g., a label); *c)* the optional and mandatory children of each node type, if any.

For example, a simplified meta-model for the Java language may have 4 node types for *classes*, *methods*, *fields* and

[1] https://github.com/GumTreeDiff/gumtree

Figure 1: (Case 1) Example of spurious add-remove edits found when GumTree computes the diff of the file `Interpreter.java` from Log4J project (commit a04d92) using the default configuration and JDT meta-model. The best configuration found by `DAT` using the global GridSearch strategy does not produce those spurious edits.

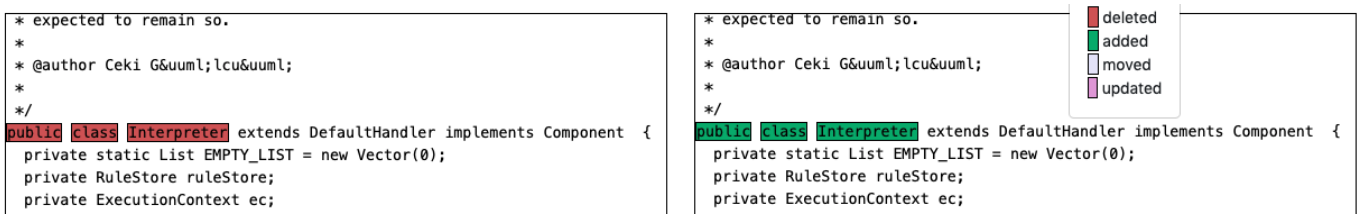
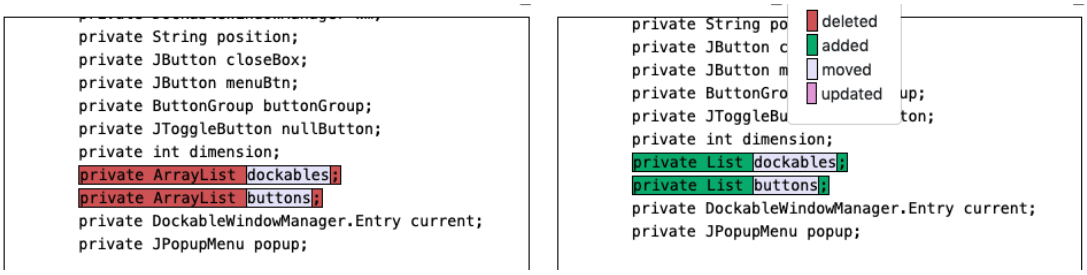


Figure 2: (Case 2) Visualization of the edits computed by Gumtree between the file `PanelWindowContainer.java` (jEdit project) from commit 6867bd (right part) and its previous version (left part) using default configuration. The AST diff is too coarse grain and does not focus on the type change.



statements; the *method* nodes have *a)* an attribute called *name*, *b)* a list of zero or more *statements* nodes as children.

Note that there can be several meta-models for representing ASTs of a same programming language. For example, GumTree can model a Java AST according to four different meta-models: JDT, Spoon, JavaParser and srcML. Since the choice of meta-model has an impact on the topology of the resulting ASTs, it finally also has an impact on the computed edit-scripts.

IV. DAT: An Approach for Hyperparameter Optimization of AST Differencing

We present `DAT`, an approach for optimizing AST differencing algorithms which have configuration parameters such as [15], [19], [22], [14]. The main goal of `DAT` is to find the optimal algorithm configurations of an AST differencing algorithm in a data-driven manner. `DAT` finds the best algorithm configurations with respect to a benchmark of file pairs to diff.

A. AST Differencing Hyperparameters

An AST diff hyperparameter is a parameter whose value is used to control a particular procedure from an AST diff algorithm. A hyperparameter has a domain and a default value. The *default configuration* of a diff algorithm corresponds to the default values of the set of hyperparameters. Hyperparameters can be set using different mechanisms: *1)* at compile time, by writing the values directly in the code; *2)* at startup time, through a command-line option or environment variable; *3)* at run time, through a method call.

To better understand hyperparameters, we take the example of the GumTree algorithm. At some point, GumTree computes a mapping between nodes from the ASTs under comparison that have more than a certain ratio of common children. This threshold is called `BUM_SMT`, its domain goes from zero to one, and its default value is 0.5. This threshold is an hyperparameter of GumTree.

A **diff algorithm configuration** for an AST diff algorithm δ is a set of actual values for all available hyperparameters in δ, The hyperparameters are noted $(h_1, h_2, \ldots, h_k)$ and the hyperparemeter values $(v_1, v_2, \ldots, v_k)$. A value v_i belongs to the **hyperparameter space** HS_i for h_i. The **hyperoptimization space** contains the Cartesian product of all hyperparameter spaces. The hyperoptimization space has n-dimensions, where each corresponds to a hyperparameter. A point in that space corresponds to a particular algorithm configuration of a diff algorithm. For each diff algorithm, there is one single point in its hyperparameter optimization space that corresponds to the *default* configuration.

An important dimension of AST differencing algorithms is the used *AST meta-model*. Two meta-models can produce, for a same source code file, two different ASTs in terms of topology (such as size or height). However, thresholds applied on values computed from the topological properties of the ASTs are very common in AST differencing algorithms. For instance, in GumTree at some point an optional mapping phase is launched depending on the number of children of the node under consideration (the `BUM_SZT` hyperparameter which default value is 1000). For instance, as the Spoon Java parser produces ASTs with more nodes than the JDT Java

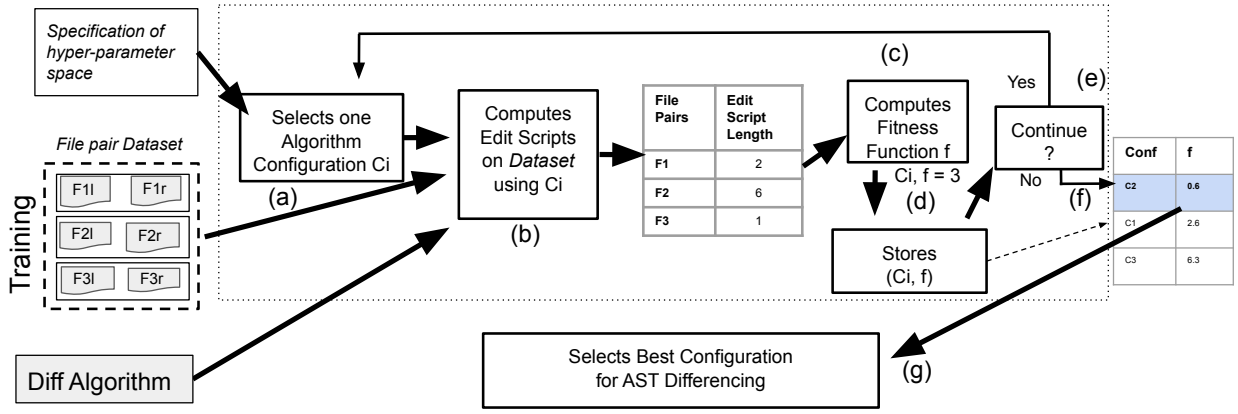


Figure 3: Workflow of DAT. It searches for the best AST differencing configuration in a data-driven manner, according to a set of file-pair.

parser[2], it could affect the behavior of the algorithm. For this reason, we suspect that a first interesting case to apply hyperparameter optimization is at the level of each AST meta-model.

Another important dimension is the source code itself. For instance, a source code with very long methods will also induce a very different behavior of the algorithm, due to the previously mentioned threshold, compared to a source code with very short methods. Therefore, a second interesting case for hyperparameter optimization is at the level of given file pair for a given AST meta-model.

We summarize these two use cases in the next section.

B. Use-cases of DAT

There are two main use cases of DAT: *global* and *local* hyperparameter optimization.

For **global hyperparameter optimization**, the goal is to find a default configuration that is expected to work well for ASTs produced using a given meta-model. The global hyperparameter optimization is designed to assist AST differencing tools maintainers to compute good default values for the hyperparameters for a given AST meta-model. These values will then be automatically applied when using the diff tool to provide an improved end-user experience without suffering additional cost (the optimized values are computed on a training set and then stored in the diff tool).

For this use case DAT does a *set*-based optimization: it searches for the best-performing configuration over a set of training file pairs written in a same language and parsed using a same meta-model.

[2]We applied a statistical test to ensure that ASTs generated from the JDT meta-model (JDT) are different from those generated from the Spoon meta-model. As the distributions of AST' sizes and heights are not normalized, we use a Wilcoxon signed-rank test to reject (at 0.05 level) the two null-hypotheses which state that the size (resp. height) of a AST_{Spoon} is similar to the size (resp. height) of an AST_{JDT}.

For **local hyperparameter optimization**, DAT does a *case*-based optimization: it searches for the best performing configuration on a given file pair, using a given meta-model. It is designed for end-users that want to compute the best possible diff at the expense of computation time (since in this setup the optimization is computed each time an end-user wants to diff a file pair).

C. Workflow

DAT takes as input: *1)* an *unlabelled* dataset of file pairs, *2)* a meta-model, *3)* an AST diff algorithm, *4)* and the specification of the algorithm's hyperparameter space. It outputs a sorted list of algorithm configurations, ordered in ascending order, from the best to the worst performing according to a given metric.

DAT implements the workflow presented in Figure 3. Given a set of file pairs (i.e., a training dataset for data-driven optimization), and a specification of the hyperparameter space, DAT first selects one algorithm configuration, C_i, to evaluate (step a). The exploration and selection of other configurations is done according to a search technique (as described in Section IV-E).

Then, DAT computes the edit-scripts for each file pair from the input dataset using the selected configuration C_i (step b). After having executed all the AST differencing tasks, DAT computes the $fitness$ value C_i, this fitness value then guides the search process. As the fitness value, by default DAT computes the averages of the length of those edit-scripts (step c), and stores that fitness value together with the configuration in a list (step d). Note that DAT can be extended to use another fitness function. Then, it may repeat the aforementioned step by selecting another algorithm configuration (step e), or it stops the search (step f). The stopping criterion depends on the search technique employed by DAT.

Finally, DAT sorts the list with the evaluated configurations according to their fitness values (i.e., the average edit-script lengths) in increasing order: DAT considers that whose algorithm configurations which produce shorter edit-scripts have

better fitness than those algorithm configurations that produce larger edit-scripts on average.

The best algorithm configuration found by DAT in the previous phase (the first element from the returned list) can be used to test the optimization done by DAT on a separate dataset. For example, we can compare the performance on a testing dataset of the best configuration found (step g) with that one from the *default* configuration (step h). This best configuration can be used in production as the new *default* configuration of the AST diff algorithm under optimization.

In Sections IV-D and IV-E, we detail, respectively, the fitness function and the search techniques employed by DAT

D. Fitness Function

The main challenge that AST diff hyperparameter optimization faces is the definition of a good fitness function. There is no ground truth on whether a hyperparameter configuration is good or not. DAT overcomes the lack of ground truth for hyperparameter optimization by using metrics that compare the outputs of two algorithm configurations (i.e., two edit-scripts). Based on the result of that comparison, DAT decides which of those algorithm configuration produces an edit-script of *higher quality*.

In DAT, we use the *length of an edit-script* as the metric to guide hyperparameter optimization. As suggested by [15], the length of the edit-scripts is an indicator of the effort required to understand the changes between two files, because shorter edit-scripts are typically easier to understand. The length of edit-scripts is an accepted proxy of the quality of the edit-scripts: other researchers have used it to measure the improvements introduced by new AST diff algorithms (e.g., [23], [22], [24], [21], [14], [35]). Since the goal of DAT is to find a configuration that performs best on a set of file pairs, DAT finds the algorithm configuration that produces, on *average*, the shortest edit-scripts from those pairs.

Algorithm 1 $fminLengthEditScript$ function

Input: α: a single algorithm configuration (it contains one particular value for each hyperparameter)
Input: FP: set of file pairs
Output: *fitness*: the fitness value of α

1: **for** pf_i in FP **do**
2: ed$_{i\alpha}$ $\leftarrow$ computeDiff(α.algorithm, α.configuration, pf_i.left, pf_i.right)
3: EDs$_i$ $\leftarrow$ EDs$_i$ $\cup$ (ed$_{i\alpha}$, α)
4: **end for**
5: $fitness \leftarrow$ computeAverageEditScriptLength(EDs)

Algorithm 1 presents the fitness function fminLengthEditScript that DAT minimizes. The function fminLengthEditScript receives as input: *1)* single algorithm configuration α (i.e., a particular value for each hyperparameter), and *2)* a set of file-pairs. It returns a fitness value for α. The fitness value is computed as follows. For each pair file received as parameter (line 1), the function computes the edit-script using an AST diff algorithm configured with algorithm configuration (line 2) and stores it (line 3). Finally, it computes and returns the average of the lengths of those edit-scripts (line 5).

Algorithm 2 GridSearch

Input: $fmin$: Objective function to minimize
Input: SC: set of algorithm configurations
Input: FP: set of file pairs
Output: L_{best} : list of sorted algorithm configurations (same size than SC)

1: $L_{best} \leftarrow$ []
2: **for** α in SC **do**
3: fitness$_i$ $\leftarrow$ $fmin$(α_i, FP)
4: $L_{best} \leftarrow L_{best} \cup$ (α, fitness$_i$)
5: **end for**
6: sort(L_{best}) ▷ Sorts each tuple (configuration$_\alpha$, fitness$_\alpha$) according to the fitness value, in increasing order.

E. Search Techniques

DAT includes two powerful search-based techniques, also used in hyperparameter optimization for other software engineering tasks (e.g. [46], [47]). These are grid search and Bayesian optimization.

1) DAT Grid Search: The grid search, refered as GridSearch, consists of exhaustively searching through a specified slice of the hyperparameter optimization space.

Given the specification of an hyperparameter space, DAT first creates algorithm configurations by performing the Cartesian product between all the selected subsets of hyperparameters. Then, DAT invokes the function $GridSearch$, presented in Algorithm 2, passing as parameters: *a)* the fitness function fminLengthEditScript, *b)* the created set of algorithm configurations, and *c)* a set of file-pairs (the training set). The output of this function is the algorithm configuration that performs better on that set. For each algorithm configuration α received as a parameter (line 2), DAT first invokes the fitness function fmin passing as a parameter the configuration α and the set of file pairs FP received as parameter (line 3). Then, it stores the tuple α and its fitness value in a list L_{best} (line 4). Finally, it sorts the list according to the fitness value in increasing order (line 2). The first element from that sorted list corresponds to the algorithm configuration with the best performance on FP. If there are several best-performing configurations, DAT returns arbitrarily one of them.

2) DAT Bayesian Optimization with TPE: DAT provides hyperparameter optimization based on Tree-of-Parzen-Estimators (TPE) algorithm [7]. TPE has been used, for example, by [46], [33] to optimize defect prediction models.

TPE is a Bayesian optimization approach [43], more concrete a *Sequential Model-Based Optimization* [25]). The latter builds a probability model $p(score|hyperparameter)$ (aka the "surrogate" function) of the objective (fitness) function and uses it to select the most promising hyperparameters to evaluate in the objective function. In other words, the model maps hyperparameters to a probability on the objective function, and acts as an estimation of the objective function. This model is built iteratively using the results of the evaluation of previously selected and evaluated hyperparameters.

Algorithm 3 TPE Search

Input: $fmin$: Objective function to minimize
Input: shs: Specification of hyperparameter space
Input: FP: set of file pairs
Input: NR_EV: number of configurations to evaluate
Output: $best$: the evaluated configuration with better fitness

1: model ← init(shs)
2: **for** i from 1 to NR_EV **do**
3: α_i ← model.getConfigurationToEval()
4: $fitness_i \leftarrow fmin(\alpha_i, FP)$
5: model.updateModel(α_i, fitness$_i$)
6: **end for**
7: $best$ ← model.getBest()

The advantage of this algorithm when evaluating expensive fitness functions (such as the function we use presented in Algorithm 1) is its execution cost [7]: It is less expensive than GridSearch. Rather than exhaustively exploring the hyperparameter space (which invokes the objective function per each point of the space), it only evaluates the hyperparameters that are selected according to the probabilistic model. TPE uses a "selection function" which selects the next hyperparameter to be evaluated from the probabilistic model. This selection is based on the Expected Improvement (EI) criterion for a set of hyperparameters (which form a single algorithm configuration). TPE aims at optimizing (maximizing) that criterion bu using a Tree-structured Parzen Estimator. A more detailed description of TPE can be found in [7].

`DAT` implements TPE to find the algorithm configuration α that *minimizes* (or eventually *maximize*) a fitness function (for example, one that minimizes the average *lengths* of the edit-scripts computed using configuration α on a set of file-pairs). Algorithm 3 presents its workflow. TPE has four main input parameters: *1)* specification of the hyperparameters space, *2)* objective function to minimize (fminLengthEditScript), *3)* training data, and *4)* number of evaluation executions (this corresponds to the budget of the search). TPE from `DAT` performs n evaluations of algorithm configurations (line 2). In each, TPE selects one algorithm configuration α included in the hyperparameter space (line 3) using the probabilistic model (initialized in line 1), Then, TPE computes the fitness of α in the training data FP using the function $fmin$. By default, the $fmin$ value passed as parameter by `DAT` is fminLengthEditScript (Algorithm 1). Then, TPE upgrades the model given the latest fitness value (line 5), and continues with the next evaluation. Finally, it returns the algorithm configuration with the best fitness (line 7). More details on how TPE uses and updates the probability model can be found in [7].

V. Experimental Methodology

In this section, we present our methodology for evaluating `DAT` and answerinr the following three research questions.

- RQ1: To what extent does hyperparameter optimization improve the performance of AST differencing?
- RQ2: To what extent can one speed up hyperoptimization with TPE compared to and exhaustive search technique?
- RQ3: To what extent is local hyperparameter optimization effective?

A. Differencing Algorithm under Study

In this paper, we select the state-of-the-art AST differencing algorithm GumTree [15] has been used in hundreds of research works relying on AST differencing, according to the citations of the GumTree paper [15])collected by Google Scholar.

1) Hyperparameters: First, we specify the hyperparameter space of GumTree. This is done by analyzing the source code and discussing it with the lead developers of GumTree, one of which is also co-author of this paper. Table I lists and explains the hyperparameters of GumTree. For each hyperparameter, the column `Default` shows the default value used in the implementation in version 2.1.2.[3]

a) Bottom-up Matcher: The first hyperparameter of GumTree is the *bottom-up matching algorithm*. GumTree sequentially applies two types of matchers [15]:

1) Top-down matchers (also known as subtree matcher): find isomorphic subtrees of decreasing height or size. Mappings are established between the nodes of these isomorphic subtrees. They are called *anchors* mappings.
2) Bottom-up matchers: navigate a tree in post-order (e.g., visit first leaves, then their parents, etc.) in order to match nodes not previously matched in the top-down phase. Two nodes match if their descendants (children of the nodes) include a large number of common anchors. Whenever a new mapping is established during this phase, a *recovery* phase is applied as the last chance to find mappings of the descendants of the nodes.

There is only one stable top-down matcher in GumTree while there are three different stable bottom-up matchers: classic, simple, and hybrid. These three matchers differ only in the way they apply the recovery phase.

b) Priority calculator: During top-down matching, GumTree greedily matches whole isomorphic subtrees. To establish the priority of the chosen subtrees, it uses a metric based upon the topology of the subtree: either its size (number of nodes in the subtree) or its height (length of the longest path from one leave to the root of the subtree). For example, when using size, GumTree will first try to find an isomorphic subtree for the subtrees with the largest number of nodes. This hyperparameter is called `STM_PC`

c) Minimum priority threshold: As explained in the previous paragraph, GumTree uses a topological metric to order the subtrees to be matched by the top-down matcher. However, subtrees that have a too small such metric are not considered by the matcher. The effect of the value depends on the chosen priority calculator. For instance, with size and 3, GumTree will not consider subtrees with two nodes or less during the top-down matching. This hyperparameter is called `STM_MPTH`.

[3]GumTree version considered: https://github.com/GumTreeDiff/gumtree/commit/ed3beeab1e00a31f23ab5e9a8292c3168221a1ca (July 2020).

Table I: Hyperparameter space for the GumTree AST Differencing Algorithm.

Hyperparameter	Description	Default	Values			
			Min	Max	Step	Total
Bottom-up Matcher	Bottom-up matcher used to compute the diff	Classic	{Classic, Simple, Hybrid}			3
STM_PC	Indicates the priority calculator used by the subtree matchers	Height	{Size, Height}			2
STM_MPTH	Threshold on the minimum priority value computed using STM_PC	1	1	5	1	5
BUM_SMT	Threshold on the minimum similarity between two AST nodes	0.5	0.1	1	0.1	10
BUM_SZT	Threshold on the maximum size of AST nodes to match	1000	100	2000	100	20

d) Minimum similarity threshold: A bottom-up matcher matches two AST nodes if *1)* they have the same type, and *2)* have a similarity greater than a threshold. Similarity is computed based on the common number of mapped descendants that both nodes have. Increasing this threshold implies that a bottom-up matcher increases the minimum ratio of common descendants, and consequently, tries to match more similar subtrees. The bottom-up matchers of GumTree obtain the similarity threshold in different ways. The greedy matcher uses the hyperparameter `BUM_SMT` to establish the similarity threshold. The default value is 0.5, which means that a bottom-up matcher only consider nodes that have, at least, a 50% of common descendants. The other two bottom-up matchers (Simple and Hybrid) automatically compute the threshold by using the following formula: $threshold(t_1, t_2) = 1/(1 + log(desc(t_1) + desc(t_2)))$, where t_1, t_2 are subtrees, and $desc(t)$ gives the number of descendants of subtree t.

e) Maximum size threshold: As explained previously, once a bottom-up matcher finds, from a subtree on tree t_1, the most similar subtree from tree t_2, it applies a *recovery* phase (Section V-A1a), which relies on an algorithm that searches for matches among the descendants of both subtrees that are still unmapped. Classic uses an optimal tree-edit distance algorithm which has a cubic complexity and, therefore, is slow on large subtrees. Simple uses an heuristic which is much faster than the optimal algorithm. Hybrid applies the algorithm of classic or simple, depending on the size of the subtree under consideration. Given the fact that classic and hybrid matchers can have a large running time if they try to apply the optimal tree-distance algorithm on large subtrees, they use the maximum size threshold hyperparameter which value sets the maximum size of a subtree for which this algorithm is applied. This hyperparameter is called `BUM_SZT` and its default value is 1000.

2) Defining the Hyperparameter Domain: Table I shows the domain of each hyperparameter space. Some hyperparameters are numeric; in this case, we give the minimum and maximum values. In addition, we give a reasonable step value to explore the input domain for this parameter, this value was suggested and agreed on with the GumTree lead developer. For example, the hyperparameter `BUM_SMT` goes from 0.1 to 1, with steps 0.1, giving as a result the hyperparameters {0.1, 0.2, 0.3, . . . , 0.9, 1}. For the hyperparameters with categorical scale, we give a list of possible values. For example, the bottom-up matcher hyperparameter could receive three values: *Classic, Simple, or Hybrid*. The 'Total' column gives the number of values to explore per hyperparameter.

We recall that the Cartesian product on all hyperparameters creates all possible algorithm configurations that `DAT` evaluates. In total, we obtain 2210 different configurations in GumTree. Note that this total is not equivalent to the scalar multiplication of the values of each hyperparameter (shown in the last column of Table I because there are dependencies between the hyperparameters. For example, the hyperparameter `BUM_SZT` is used by (*GreedyBottomUpMatcher* matcher but not by *SimpleBottomUpMatcher*. Thus, 2000 configurations correspond to ClassicGumTreeMatcher, 200 to HybridGumTreeMatcher and 10 to SimpleGumTreeMatcher,

3) Metamodels: In our experiments, we perform AST differencing on Java programs, as done in the original publication of GumTree [15]. GumTree supports multiple AST metamodels for Java code. The default one is called *JDT*, it is based on the Eclipse JDT Parser. The other meta-model we choose is the one defined using Spoon [42], an open-source library to analyze, rewrite, transform, and transpile Java source code.

4) Evaluation Dataset: The evaluation of `DAT` consists in running the hyperparameter optimization GumTree on a set of file-pairs. We create a dataset of file-pairs used in the evaluation as follows. First, we choose software repositories in order to extract revisions of files done by developers on open-source projects. We choose CVSVintage [38], a dataset composed of 14 CVS repositories of open-source projects written in Java, because GumTree [15], the differencing algorithm that we hyper-optimise in this paper, was initially evaluated on that dataset [15].

To create the set of file pairs, we first convert each CVS repository to a GIT repository using the tool `cvs2git`[4]. We were able to successfully convert 13 out of the 14 repositories. Then, we navigate the history of each GIT repository, commit by commit and for each one, we store the Java files that have been updated according to the command `git diff`. More precisely, for each file f updated by commit C, we create a pair file $(f_p, f,)$, where f_p is the previous version of file f (i.e., the version before commit C). All file pairs are publicly available in our appendix.

We exclusively study on file-pairs that introduce AST changes. File-pairs that only differ on the code format (e.g., indentation) are not considered, since the number of AST changes between the pair is equal to zero. For detecting those file-pairs to discard, we compute the edit-script GumTree (using the default configuration) on each pair and we keep those that have an edit-script longer or equal to one.

[4]cvs2git: https://www.mcs.anl.gov/~jacob/cvs2svn/cvs2git.html

As the number of file-pairs we obtained is larger than 100000, and our computational resources are limited, given the magnitude of this experiment (run different configurations on each of those pairs), we take a subset of them. We randomly select up to 5000 file-pairs per project. Note that 9 out of 13 projects have less than 5000 file-pairs, so we consider all of them. In total, we consider 31,543 file-pairs. Given this amount of data, we perform 69,710,030 unique executions of GumTree (i.e., 31,543 file-pairs $\times$ 2210 different configurations).

B. Protocols

1) Protocol for RQ1: To answer this research question, we execute the *global* hyperparameter optimization from DAT on the evaluation dataset described in Section V-A4, composed of 31,543 file-pairs. We performed this hyper-optimization for the two considered AST meta-models (JDT and Spoon) using the GridSearch technique implemented in DAT.

To minimize the risk of data overfitting, we apply a 10-fold cross-validation. For each fold, we generate two sets (training and testing) for the complete dataset. To create these sets, we split the data into 10 groups. Then, each fold uses one of those groups as testing (10% of the data) and the remaining as training (90% of the data).

For each fold, we hyperoptimize GumTree using the training dataset with the goal of finding the configuration with the best performance C_{Best}. Then, using the testing dataset, we calculate the performance of: *a)* the best configuration (C_{Best}), and *b)* the default configuration ($C_{Default}$). Next, we compute the proportion of file pairs where: *a)* hyper-optimization (C_{Best}) improves the edit-script w.r.t. default configuration ($C_{Default}$) (Metric I), *b)* hyper-optimization produces an equivalent edit-script (Metric E), *c)* hyper-optimization produces a worse edit-script (Metric W). Finally, we report the average of I, E and W over all folds.

We also proceed to a statistical assessment of the results using a Wilcoxon signed rank test against the size of the edit-scripts produced by two different configurations: the default and the best one found using the GridSearch technique. We use this test since we have no assumption about the distribution of the edit-script sizes. Our null and alternative hypotheses that we focus on this RQ are as follows:

- H^1_{null} There is no difference between the median length of the edit-scripts produced using the global and default configuration (alternative H^1_{alt} the edit-scripts produced using global have a median shorter length than the ones produced using default).

We also report the effect size Rosenthal's R, whose value varies from 0 (small effect) to close to 1 (large effect).

2) Protocol for RQ2: In the previous research question, we execute DAT to compute global optimization of GumTree using the GridSearch technique. This technique evaluates *all* possible configurations on the *complete* evaluation dataset. In this research question, we study two potential optimizations on DAT: *1)* the use of another search technique, and *2)* the use of less training data.

We first analyze the impact of using another search technique, TPE described in Section IV-E2, which requires as input a *search budget*. In the context of this research, the budget corresponds to the number of diff executions (each uses a different configuration) that TPE applies.

To study the impact of different budget values, we follow the protocol applied for responding to RQ 1 (Section V-B1) to execute TPE instead of GridSearch. We execute that experiment four times, each time with a different budget B: 10, 25, 50 and 100. A budget of, for instance, 50 means that TPE evaluates 50 different diff configurations in the training data. We recall that the budget B has an upper limit equal to 2210, which corresponds to the Cartesian product on all GumTree hyperparameters (described in SectionV-A2). We report the percentages of improvement given by the best configuration found by TPE using a particular budget.

Secondly, we analyze the impact on the improvement according to the amount of data used during the training. Instead of running DAT on all data (31,543 file-pairs), we run it on samples of different sizes. In this paper, after analyzing the results obtained from dataset different sizes, we report those from two sizes: 100 file pairs and 1000 file pairs. For each of these sizes, we take five samples and for each of them apply the protocol exampled in RQ 1 (i.e., 10-fold cross validation). Finally, we report the mean improvement.

3) Protocol for RQ3: To answer this research question, we extend the protocol used to answer research question 1 (Section V-B1) based on 10-fold cross-validation. In each fold, in addition to compute the best global configuration, we find, using GridSearch and TPE, the best local configuration for each file pair from the testing set.

Then, for each of those training points, we compare the fitness value (i.e., length of the edit-script) given by the local search on a data-point with the value obtained using the default default configuration on the same data-point. Similarly, we compare the fitness value from the local search with the obtained using the best configuration (found using GridSearch on the training set from the fold). We configure TPE with 25 evaluations per each file pair from the testing set, because, as shown in RQ2, that value shows a good trade-off between the number of evaluations and % of improvement. Finally, we report the average of I, E and W over all folds, similarly to the previous research question.

We also proceed to a statistical assessment of the results using a Wilcoxon signed rank test against the size of the edit-scripts produced by the different configurations. Our null and alternative hypotheses are study in this research questions are:

- H^2_{null}: There is no difference between the median length of the edit-scripts produced using the local and default configuration (alternative H^2_{alt} the edit-scripts produced using length have a median shorter length than those produced using default).

VI. Experimental Results

A. RQ1: To what extent does hyperparameter optimization improve the performance of AST differencing?

Table II presents the results of this research question. It displays three columns that present the percentage of cases from the testing set where hyperoptimized GumTree finds:

Table II: RQ1: Comparison between the performance of globally hyperoptimized GumTree and default GumTree.

Meta-model	% cases where GridSearch optimization			p-value
	Improves (I)	Equals (E)	Worse (W)	
JDT	18.7%	78.7%	2.21%	$2.2e-16$
Spoon	13.12%	84.9%	1.68%	$2.2e-16$

a) a shorter edit-script than default GumTree, meaning that the hyperoptimization improves the performance of GumTree (column I), *b)* the same length edit-script (column E) *c)* a larger edit-script, meaning that the hyperoptimization harms the default configuration of the differencing algorithm (column W).

To diff ASTs from JDT, the optimization of GumTree improves the performance of default GumTree for 18.7% of cases. The detriment of applying global optimization is much lower: in only 2.21% of the cases, hyperoptimized GumTree produces larger edit-scripts. For the rest of the cases (78.7%), hyperoptimization has no impact on the length of edit-script produced by GumTree. Table III shows the best configurations found by DAT for the JDT and Spoon meta-models, and the default value used by GumTree. We observe that for JDT the best configuration uses a different matching algorithm (Hybrid) than the default configuration (Classic).

We use a Wilcoxon signed rank test to statistically assess the differences of edit-script lengths produced using the hyperoptimized configuration versus the default configuration. The obtained P-value is $2.2e-16$, therefore, we reject the null hypothesis H^1_{null}. The effect size, calculated using Rosenthal's R, is-0.536, which can be considered between medium and large.

To diff ASTs designed with the Spoon meta-model, hyperoptimization with DAT has less impact on the number of improved cases (13.12%). It means that the default configuration of GumTree works already well in most cases. We observe from Table III that the best configuration for Spoon has the same matching algorithm as the default (Classic). However, there are three parameters that receive different values: STM_ PC, BUM_SMT and BUM_SZT.

Again, we run a Wilcoxon signed rank test on the edit-script length distribution and the obtained P-value is inferior to $2.2e-16$, therefore, we reject the null hypothesis H^1_{null}. The effect size, computed using Rosenthal's R is -0.668, which can be considered between medium and large.

Figure 4 shows the distribution of the percentage of reduction in the size of the edit-scripts calculated with the optimized configuration compared to the edit-script calculated with the default configuration. The figure considers the cases that present improvement due to the optimization process (these have % positive and correspond to 18.8% in JDT and 13.12% in Spoon), cases for which optimization produces worse results (% negative, 2.21% in JDT and 1.68% in Spoon), but ignores those with equal results to facilitate visualization. For JDT, half of the optimization-affected cases have a reduction of at least 20%, and there is a considerable number of cases with an improvement greater than 75%. In contrast, for Spoon, the reduction in the size of the edit-scripts is less than for JDT, and there are fewer cases that are reduced by more than 50%.

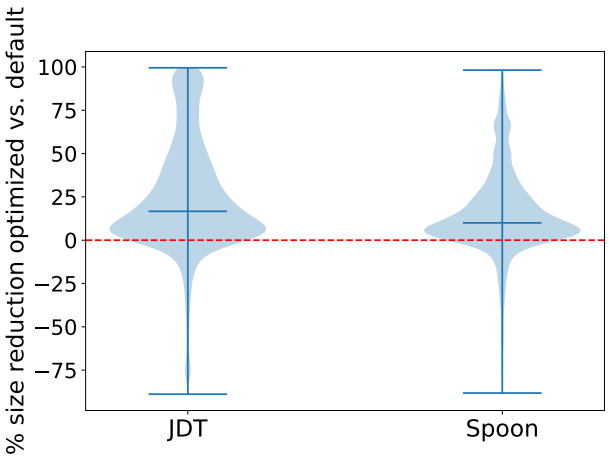


Figure 4: Distribution of the percentage of reduction of the edit-script size using the optimized configuration w.r.t. the default configuration. (Cases with no improvement or detriment are ignored).

Hyperparameter	Default	Best for JDT	Best for Spoon
Matching Algorithm	Classic	Hybrid	Classic
STM_ PC	Height	Size	Size
STM MPTH	1	1	1
BUM_SMT	0.5	-	0.2
BUM_SZT	1000	200	600

Table III: (RQ 1) Best global configurations found by DAT using GridSearch for JDT and Spoon Java meta-models. The symbol '-' means that the configuration does not use the hyperparameter.

This shows that the effect of the optimization is different according to the meta-model used: for JDT, the optimization *a)* affected more cases, and *b)* for those affected cases, the reduction is more significant.

Answer to RQ1: Global hyperparameter optimization improves the performance of GumTree by providing a better configuration than the default configuration. The hyperoptimized configuration produces the shortest edit-scripts for 18.7% of the cases using the JDT meta-model and 13.12% using the Spoon meta-model, in a statistically significant manner.

Implications for practitioners: Maintainers and users of AST differencing tools such as GumTree have a new tool in their toolbox. When they apply AST differencing to a new programming language, they can first perform a global hyperparameter optimization, which would identify a new configuration that is *1)* better than the default, and *2)* tuned to a given AST meta-model.

Dataset	DAT GridSearch	DAT TPE with budget (# evaluations per training sample)			
	2210 evals	10	25	50	100
All	18.7% (Stdev 1%)	14.5% (Stdev 0.5%)	16.3 (Stdev 0.9%)	18.24% (Stdev 0.15%)	18.21% (Stdev 0.16%)
Reduced_{1000}	18.8% (Stdev 3.9%)	13.9% (Stdev 3.2%)	17.03% (Stdev 3.98%)	17.2% (Stdev 4.05%)	17.5% (Stdev 3.9%)
Reduced_{100}	13.6% (Stdev 11%)	12.3% (Stdev 10.6%)	15.5% (Stdev 11.2%)	15.4% (Stdev 11.2%)	15.5% (Stdev 11.2%)

Table IV: (RQ2) Percentage of improvement of GumTree using the best configurations found by DAT GridSearch and DAT TPE vs default configuration under different scenarios: a) less evaluations (10, 25, 50 and 100), and b) less training data (Reduced_{100} and Reduced_{1000}).

B. RQ2: To what extent can one speed up hyperoptimization with TPE compared to and exhaustive search technique?

We now study the impact of using another search technique on the search for the best configuration: the TPE technique.

Table VI-B shows the percentage of improvement of GumTree using the best configuration found by TPE and GridSearch compared to the default configuration. There is one column for GridSeach, which corresponds to the exhaustive evaluation of the space (2210 evaluation per case), and four columns for TPE: each one corresponds to the search budget passed to TPE (10, 25, 50 and 100 evaluations per case).

We focus on the first row (All), in which both techniques are trained on the complete data. We observe that it is enough for TPE to consider 100 evaluations per training sample in order to obtain the same improvement as GridSearch: GumTree using the best configuration found by TPE using 100 evaluations improves the 18% of the testing samples with respect to GumTree using the default configuration. Notably, the number of evaluations performed by TPE to achieve that improvement is much lower: 100 vs. 2210, that is, a reduction of $\approx 95\%$ of the evaluations.

We observe that even by reducing the number of evaluations more, TPE is still able to find a configuration that produces improvements. For example, using 25 configurations (that is, inspecting the 1.13% of the hyperparameter space) TPE produces improvements of 16%, while GridSearch, which inspects the 100% of the hyperparameter space, arrives at 18%.

Now, we focus on the impact of the improvement when we reduce the training dataset. Table shows the results obtained using a dataset composed of 1000 and 100 samples. As we perform cross-validation (Section V-B2) the number of samples for training is 900 and 90, respectively, where those used for testing are 100 and 10, respectively.

We observe that using Reduced_{1000} (that is, 900 samples for training), we obtain improvements similar to those obtained using all data ($\approx$ 29000 samples). Moreover, if we use even fewer samples for training and testing from Reduced_{1000} produces a lower mean improvement (e.g., 12.3% using TPE with 100 evaluations / sample). However, the standard deviation of the improvement is much higher (e.g., 11% for GridSearch) than that obtained using all data (1%). The reason is that the small training samples used in the cross-validation may not represent all the population.

Answer to RQ2: Using TPE, DAT can significantly reduce the number of evaluations executed to find the best configuration of GumTree w.r.t. the chosen fitness function, meaning a faster hyperoptimization.

Implications for practitioners: To perform a global hyperparameter optimization for a new programming language or meta-model, it is enough to collect 1000 file-pairs (diffs) and use TPE with 50 evals/diff.

C. RQ3: To what extent is local hyperparameter optimization effective?

Now, we compare the effectiveness of hyperparameter optimization on the global and local scales. Recall that the local scale means hyperoptimizing a single AST diff. Table V presents the results of this experiment. It contains three sets of columns which present, for both the JDT and Spoon meta-models, the percentages of file-pairs for which hyperoptimization improves the results (I), produces equal results (E), or produces worsened AST diffs (W). We compare all possible setups between default configuration, globally hyperoptimized configuration and locally hyperoptimized configuration.

The row **DAT Local GridSearch vs. Default** presents the results obtained from local optimization. Local optimization positively impacts the performance on GumTree, producing shorter edit-scripts than those from the default configuration for 22.2% of cases with JDT and for 15.6% of cases with Spoon. We note that the improvement in local hyperoptimization is larger than that provided by global hyperoptimization ($22.3\% >> 18.7\%$). Using the Wilcoxon signed rank test, we reject the null hypothesis H^2_{null} for both the JDT and the SPOON meta-models. The effect size computed using Rosenthal's R are -0.533 and -0.664 respectively, which can considered between medium and large.

Notably, local optimization never worsens the performance of AST differencing. This is by construction, as it optimizes a single file-pair, if the technique does not find a configuration that improves the default, then the default is used.

The **DAT Local TPE vs Default** from Table V shows the results of using TPE instead of GridSearch as the optimization approach. We compare the results using the configuration found locally by DAT-TPE and the default configuration. We observe that using TPE the percentage of improved cases is a bit lower than those using GridSearch. In particular, the configurations found by DAT using TPE improve 21.44% of

Comparison	Percentage of cases improved (I)		Percentage of cases equal (E)		Percentage of cases worsened (W)	
	JDT	Spoon	JDT	Spoon	JDT	Spoon
DAT Local GridSearch vs Default	22.2%	15.6%	77.8%	84.4%	0%	0%
DAT Local TPE vs Default	21.4%	14.8%	78.3%	85.17%	0.23%	0.24%

Table V: (RQ3) Percentages of cases improved by applying local optimization with DAT vs. default configuration.

Hyperparameter	**Default**	**JDT**	**Spoon**
Matching Algorithm	Classic	Classic	Classic
STM PC	Height	Size	Size
STM MPTH	1	1	1
BUM_SMT	0.5	0.1	0.1
BUM_SZT	1000	1000	900

Table VI: (RQ 3) Most frequent local Hyperoptimized Configurations found by DAT for the JDT and Spoon Java metamodels.

the cases, while that found using GridSearch improves 22.2%. For JDT, the percentage of improvements follows the same trend: 14.48% using TPE, 15.6% using GridSearch. However, the number of evaluations (i.e., executions of diff) per file-pair is much lower using TPE than GridSearch: 25 for TPE (value chosen just before) vs. 2210 for GridSearch (we recall that it does an exhaustive search on the configuration space). Developers and practitioners can decide the search method used by DAT according to the scenario they apply DAT, for example, a scenario with limited budget or where they need to optimize fast, thus TPE would be more convenient, or another where they need to obtain the best result and do not have budget restrictions, thus GridSearch would be a better option.

Answer to RQ3: Local hyperparameter optimization is effective: it allows practitioners to find shorter edit-scripts than the default configuration in up to 22% of cases.

Implications for practitioners: In sensitive downstream tasks, we strongly advise to use local hyperparameter optimization for AST differencing in order to obtain the best edit-script according to a fitness function specific to the downstream task.

D. Analysis of the cases studies

In this section, we discuss how the best global hyperparameter configuration found by DAT helps GumTree produce a different output from the default configuration in each of the cases presented in Section II.

1) Case 1: Spurious Add-Remove: As described in Section II-A, GumTree produces six spurious edits that must not be produced. GumTree using the best configuration found by DAT does not produce them. The reason for having different edit-scripts when GumTree uses default and the best configuration is the following.

The three AST nodes in Section II-A cannot be mapped during the first phase (top-down matching explained in Section V-A1a). Both Classic (default) and Hybrid (the matcher used by the best configuration found by DAT) matchers apply the same top-down matching strategy: Greedy subtree matching. In the second matching phase (bottom-up), the Hybrid matcher is able to map those nodes during *recovery phase* (a last step done by a matcher which tries to map the unmapped children of two subtrees whose roots are mapped). However, the Classic matcher does not map them because it invokes *recovery phase* only if the size of the trees to match is smaller than the hyperparameter BUM_SZT (by default 1000). The size of the parent tree of these three mentioned nodes (which corresponds to the class Interpreter) is greater than BUM_SZT=1000, so the *recovery phase* is never called and the three nodes remain unmapped.

2) Case 2: Including updates in the edit-script: As described in Section II-B, GumTree generates an edit-script that removes the fields from the left part, adds the fields in the right part (this time, the fields have a new field's type List), and includes a move of the tokens related to the field names. However, the edit-script generated by GumTree using the best configuration found in Section VI is much shorter and understandable: as Figure 6 shows, it only includes one update of each field declaration statement.

The reason for having different edit-scripts when GumTree uses the default and best configurations is the following. During the top-down matching phase using the default configuration, GumTree maps two nodes (one from the left tree, the other from the right tree) if their labels are equal and, following the default values of hyperparameters STM_PC and STM_MPTH, their heights are $>= 1$. Note that using this configuration, all leaf nodes (even those that are not modified) from the left tree are not mapped to any from the right tree because their heights are 0. For example, the node that represents the modifier private in the left tree is not mapped to the corresponding private node in the right tree, even if it is not affected by the changes made by the developer.

Then, during the bottom-up matching match, Gumtree cannot match any of the nodes corresponding to the field declaration statements private ArrayList dockables; and private ArrayList buttons; from the left part with those corresponding to the field declarations from the right (private List dockables; and private List buttons;). This is because the similarity value between the field declarations from the left and right (e.g., private ArrayList dockables; and private List dockables;, respectively) is 0.4, which is lower than the default threshold BUM_SMT (i.e. 0.5), which controls

Figure 5: (Case 3) Visualization of edits computed by Gumtree between the file `PanelWindowContainer.java` (jEdit project) from commit 6867bd (right part) and its previous version (left part) using the best global configuration.

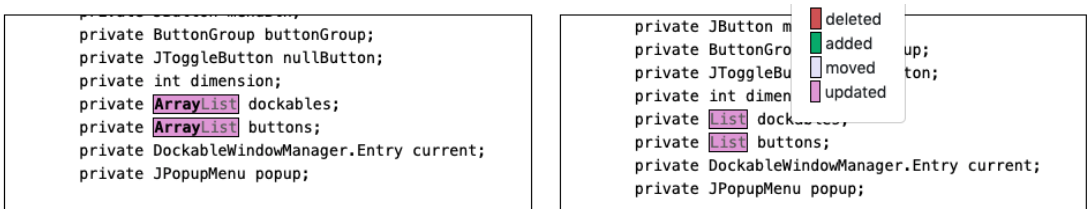


Figure 6: Diff computed using default configuration.

Figure 7: Distribution of execution time of GumTree (in milliseconds). The rightmost bar groups the values larger than 1250 milliseconds.

the mapping of two nodes. This low similarity value is due to the fact that most of the descendants have not been mapped (such as the modifier `private`, the simple type `ArrayList`, the simple name `dockable`). As a result, the edit-script generated using the algorithm of Chawathe et al. [8] and shown in Figure 2, includes remove and add edits that affect these unmapped nodes, including the field declaration.

Gumtree tuned with the best configuration, which uses the Hybrid Bottom-up matcher, finds the expected edit-script. When the tuned version of GumTree uses "size' $= 1$, it arrives to match leaves nodes that are not mapped using the default configuration ('height' $= 1$), such as the node corresponding to modifier `private`. (The map is possible because the size of a leaf node is 1).

These mappings produced by GumTree with the tuned version of GumTree but not with the default, impact the bottom-up matching: Now, the similarity score between the declaration statements (e.g., `private ArrayList dockables;` and `private List dockables;`) is 0.6, greater than the threshold `BUM_SMT` 0.5. For this reason, both statements are mapped, and the edit-script generator does not generate the spurious add and remove edits on these mapped nodes.

E. Execution Time of AST Differencing

Figure 7 shows the distribution of the execution times (in milliseconds) of GumTree, executed on each pair of GumTree configuration and file-pair. We recall that, in total, we execute GumTree 69,710,030 times. As the distribution is right-skewed, in order to facilitate the visualization, the rightmost bar groups all executions with execution time greater than 1250 milliseconds.

The distribution shows that most executions (41,129,161, that is, 59%) take less than 10 milliseconds. This shows the feasibility of performing the hyperparameter optimization of GumTree. Nevertheless, there are still executions that take longer, as the rightmost bar shows. In particular, 3,830,084 diff executions (5.5%) take more than 250 milliseconds. This situation is caused by the high values of the hyperparameter `BUM_SZT`. As explained in Section VI-D1, `BUM_SZT` controls the size of two trees under matching: if the tree's sizes are smaller than the value `BUM_SZT`, it executes a *recovery phase*, which is computationally expensive in large threes. Consequently, when `DAT` tries large values for `BUM_SZT` on large trees, the execution times increase. For example, let us focus on the revision of file `LinkTag.java` in commit `2bc1fb` using `JDT` meta-model. When `DAT` sets to `BUM_SZT` a value less than 1000, the GumTree computes the edit-script very fast, in less than 4 milliseconds. However, as long as the value of `BUM_SZT` increases, so do the execution times. For example, with a value of 1000, GumTree takes $\approx$ 860 milliseconds, and with values greater than 1100 it takes $\approx$ 1900 milliseconds.

`DAT` provides two mechanisms to avoid large executions. First, it provides a timeout on each diff execution, which can be set by the user. Secondly, it provides an option to prune the search: Once `DAT` detects a configuration that produces a timeout on a file-pair, then it does not execute other configurations with similar values on that file-pair. Both mechanisms can be activated by the user.

VII. Threats to Validity

Hyperoptimization techniques. We implemented in `DAT` two hyperparameter optimization techniques (GridSearch and TPE), because all of them were proven to be successful in hyperparameter optimization for software engineering tasks [33], [47]. There may exist other search techniques that produce better results.

Quantization of the hyperparameter space We quantize the hyperparameter space of GumTree using *initial*, *end* and *step* values (all presented Table I). It could be the case that there are values not included in the selected subset that produce better values.

AST metamodeling Our results show the importance of AST metamodeling on differencing. We selected two meta-models for modeling Java ASTs. We are aware that there exist other meta-models (e.g., JavaParser). As shown by the clear performance difference between JDT and Spoon, it may happen that hyperparameter optimization by `DAT` performs differently for other Java meta-models.

Selection of Java. Our results are done on AST differencing for Java programs, as most related work on AST differencing analysis [15], [19], [21], [24]. Future experiments will improve the external validity in other programming languages.

VIII. Related Work

A. Advanced AST Differencing Algorithms

Since the preliminary work by Chawathe et al. [9], two AST differencing algorithms with a large impact recently are GumTree from Falleri et al. [15] and ChangeDistiller from Fluri et al. [19].

Several works have extended ChangeDistiller and GumTree with the goal of improving their performance. For example, Higo et al. propose an extension of GumTree [23] with the goal of making edit-script shorter and closer to developers' actual editing sequences. For that, in addition of the 4 actions proposed by GumTree, they propose a new one: copy-and-paste. They found that 18% of the edit-scripts generated using their approach are shorter than those from GumTree. Dotzler et al. [14] present an extension of GumTree and ChangeDistiller that uses optimizations to shorten the resulting edit-scripts. In particular, they present an algorithm, MTDIFF based on ChangeDistiller, that improves the detection of moved code. As a result, this AST diff algorithm is able to reduce the length of edit-scripts.

Matsumoto et al. [35] present an hybrid AST diff approach that matches the AST nodes with information from the *diff* command (based on the Myers algorithm [39]). Their results show that their approaches generate shorter edit-script than GumTree for the 30-50% of the cases analyzed. Similarly, Yang and Whitehead [49] use textual-differencing to prune the AST. Their results show that this pruning-based technique reduces both the number of nodes and the execution time. The authors also present an extension of ChangeDistiller [50] which allows the diff algorithm to identify fine-grained changes within statements.

Frick et al. [21] present an extension of GumTree, Iterative Java Matcher (IJM), that produces more accurate and compact edit-scripts by improving the quality of the generated move and update actions. Huang et al. present ClDiff [24], an AST differencing algorithm that produces concise edit-scripts by grouping and linking AST nodes affected by changes. Their evaluation of ClDiff shows that this approach generates shorter edit-scripts for 48% file-pairs than GumTree.

There are previous papers that study the quality of the edit-scripts generated by AST differencing algorithms. De la Torre et al. [12] study the quality edit-script generated by GumTree and propose four categories of imprecisions on edit-scripts: redundant, spurious, arbitrary, and ghost changes. They empirically study the presence of such imprecision in a corpus of 107 C# system. Fan et al. [16] define an approach that calculates statements with inaccurate mappings for AST differencing algorithms. The evaluation carried out in that paper shows that GumTree [15], MTDiff [14] and IJM [21] generate inaccurate mappings for 20%-29%, 25%-36% and 21%-30% of the file revisions, respectively. Based on these results, the authors stated that the state-of-the-art AST mapping algorithms still need improvements. In fact, in this paper, we show that `DAT` helps to improve one of those AST differencing algorithms.

We have the same goal as most of those papers: reducing the length of the edit-scripts. Yet, even if some of this related work does manual tweaking of hyperparameters, none of those papers does automated hyperoptimization. Our experiment in this paper shows that the default configuration is indeed suboptimal. We note that our contribution on AST differencing hyperoptimization is applicable to all those past and future AST differencing algorithms to come.

B. Hyperparameter Optimization in Machine Learning and Software Engineering

Hyperparameter optimization has been applied in various areas, notably in machine learning. For example, Kotthoff et al. [29] presented Auto-Weka, a tool that automatically applies hyperparameter optimization to machine learning models from Weka: it searches for the model and its configuration that achieves the best classification performance. A similar work was done by Feurer et al. [17] for the Sklearn framework.

Now, let us focus on hyperparameter optimization in software engineering. Tantithamthavorn et al. [47], [46] studied the impact of automated parameter optimization on defect prediction models. Their study shows that automated parameter optimization can have a large impact on the performance stability and interpretation of defect prediction models. Among four search techniques (including grid search), they find that those optimization techniques yield similar benefits of performance improvement when optimizing defect prediction models.

Similarly, Li et al. [33] studied the impact of automated parameter optimization on cross-project defect prediction techniques. They found that automated parameter optimization substantially improves the defect prediction performance. As we do, they find that the Tree-of-Parzen-Estimators (TPE) algorithm is the most effective search technique.

Arcuri and Fraser [5] applied hyperparameter optimization on EvoSuite [20] a framework for test generation. They showed that this positively impacts the performance of EvoSuite. However, the author stressed that the parameter settings obtained may be worse than arbitrary default values. This finding has also been observed in the replication study by Kotelyanskii et Kapfhammer [28]. This has motivated us to precisely measure those cases where hyperoptimization produces worse results than the default configuration of GumTree (see Table V).

Zamani et al. [52] also focused on tuning search-based test generation. One of the challenges they targeted was to find the right subset of classes (for which tests will be generated) that are worth tuning. For that, they define a measure named

'Tuning Gain' for the cost-effectiveness of tuning in search-based test generation. Also related to software testing, Jia et al. [26] used hyperparameter optimization for Combinatorial Interaction Testing (CIT).

The width of the software engineering domains where hyperoptimization has been used is large. Apel et al. [4] integrated hyperparameter optimization in JDime, a tool for structured-merge. Agrawal et al. [2] presented SMOTUNED, an approach that tunes SMOTE [10], an oversampling technique to fix data imbalance. SMOTUNED was evaluated on the defects prediction task. Then, Agrawal et al. [1] presented Dodge, hyperparameter optimization for machine learning. Dodge detects and ignores redundant *tunings* (i.e. pairs of configurations which lead to indistinguishable results) and this helps Dodge to run orders of magnitude faster without harming the performance of the approaches under tuning. Dogde was initially evaluated on two tasks: Software defect prediction and text mining, and it was then further evaluated on other tasks including bad smell detection, predicting Github issue close time, bug report analysis, and defect prediction [3]. Yedida et Menzies [51] presented GHOST, a method that relies on a combination of hyper-parameter optimization of feedforward neural networks and a novel oversampling technique. Shu et al. [44] applied hyperparameter optimization for improving data preprocessing for software bug report classification. Basios et al. [6] applied optimization to the problem of selecting data structures that share a common interface. Panichella [41] carried out hyper-opertimization of LDA (Latent Dirichlet Allocation) applied to identify duplicate bug reports. Chen et al. presented BOCA [11] Bayesian optimization-based approach for compiler autotuning, and evaluated it on two widely-used C compilers (i.e., GCC and LLVM).

To our knowledge, we are the first to propose and comprehensively study hyperparameter optimization for AST differencing.

IX. Conclusion

In this paper, we have proposed to use hyperoptimization for AST differencing. We have described a novel approach, called `DAT`, consisting of *1)* specifying the hyperparameter space of an AST differencing algorithm, *2)* applying a search technique to hyperoptimize the algorithm in a data-driven way based on a training dataset of AST differencing cases. The approach has been instantiated for the popular AST differencing algorithm GumTree [15]. We have performed a comprehensive quantitative assessment of `DAT`, which shows that hyperoptimization improves the AST edit-scripts in up to 18% of the differencing tasks using global hyperoptimization and up to 22% using local hyperoptimization. Our technique is widely applicable to all AST differencing algorithms: it can benefit both already proposed AST differencing systems and future ones to come. The main direction for future work is the improvement of downstream tasks using hyperoptimized AST edit-scripts (e.g. commit clustering and identification).

References

[1] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies. How to "dodge" complex software analytics. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[2] Amritanshu Agrawal and Tim Menzies. Is "better data" better than "better data miners"? on the benefits of tuning smote for defect prediction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1050–1061, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Amritanshu Agrawal, Xueqi Yang, Rishabh Agrawal, Rahul Yedida, Xipeng Shen, and Tim Menzies. Simpler hyperparameter optimization for software analytics: Why, how, when? *IEEE Transactions on Software Engineering*, 48(8):2939–2954, 2022.

[4] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, page 120–129, New York, NY, USA, 2012. Association for Computing Machinery.

[5] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18, 06 2013.

[6] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 118–128, New York, NY, USA, 2018. Association for Computing Machinery.

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 2546–2554, Red Hook, NY, USA, 2011. Curran Associates Inc.

[8] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[9] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 493–504, New York, NY, USA, 1996. Association for Computing Machinery.

[10] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, jun 2002.

[11] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209, 2021.

[12] Guillermo de la Torre, Romain Robbes, and Alexandre Bergel. Imprecisions diagnostic in source code deltas. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 492–502, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Michael John Decker, Michael L Collard, L Gwenn Volkert, and Jonathan I Maletic. srcDiff: A syntactic differencing approach to improve the understandability of deltas. *Journal of Software: Evolution and Process*, 32(4):e2226, 2020. Publisher: Wiley Online Library.

[14] Georg Dotzler and Michael Philippsen. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–671. IEEE, 2016.

[15] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[16] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, Yuan Wang, and Shanping Li. A differential testing approach for evaluating abstract syntax tree mapping algorithms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1174–1185, 2021.

[17] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015.

[18] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79, 2007.

[19] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007. Publisher: IEEE.

[20] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[21] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. Generating accurate and compact edit scripts using tree differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 264–274. IEEE, 2018.

[22] Masatomo Hashimoto and Akira Mori. Diff/TS: A tool for fine-grained structural change analysis. In *2008 15th working conference on reverse engineering*, pages 279–288. IEEE, 2008.

[23] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. Generating simpler ast edit scripts by considering copy-and-paste. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 532–542. IEEE, 2017.

[24] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. Cldiff: generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 679–690, 2018.

[25] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, page 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 540–550, 2015.

[27] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 151–160, 2011.

[28] Anton Kotelyanskii and Gregory M. Kapfhammer. Parameter tuning for search-based test-data generation revisited: Support for previous results. In *2014 14th International Conference on Quality Software*, pages 79–84, 2014.

[29] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 18(25):1–5, 2017.

[30] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering Journal, Springer Verlag*, 2020.

[31] X. B. D. Le, D. Lo, and C. L. Goues. History Driven Program Repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224, 2016.

[32] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 593–604, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Ke Li, Zilin Xiang, Tao Chen, Shuo Wang, and Kay Chen Tan. Understanding the automated parameter optimization on transfer learning for CPDP: an empirical study. *CoRR*, abs/2002.03148, 2020.

[34] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Softw. Engg.*, 20(1):176–205, February 2015.

[35] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Beyond GumTree: a hybrid approach to generate edit scripts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 550–554. IEEE, 2019.

[36] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511, 2013.

[37] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 329–342, New York, NY, USA, 2011. Association for Computing Machinery.

[38] Martin Monperrus and Matias Martinez. Conservation and Replication with CVS-Vintage: A Dataset of CVS Repositories of Java Software. Technical report, 2012.

[39] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[40] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 511–522, New York, NY, USA, 2016. Association for Computing Machinery.

[41] Annibale Panichella. A systematic comparison of search-based approaches for lda hyperparameter tuning. *Information and Software Technology*, 130:106411, 2021.

[42] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. update for oadoi on Nov 02 2018.

[43] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[44] Rui Shu, Tianpei Xia, Jianfeng Chen, Laurie Williams, and Tim Menzies. How to better distinguish security bug reports (using dual hyperparameter optimization). *Empirical Software Engineering*, 26(3):53, 2021.

[45] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140, 2018.

[46] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 321–332, 2016.

[47] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2019.

[48] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494, 2018.

[49] C. Yang and E. J. Whitehead. Pruning the ast with hunks to speed up tree differencing. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 15–25, 2019.

[50] C. Yang and J. WhiteHead. Identifying the within-statement changes to facilitate change understanding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 191–201, 2019.

[51] Rahul Yedida and Tim Menzies. On the value of oversampling for deep learning in software defect prediction. *IEEE Trans. Softw. Eng.*, 48(8):3103–3116, aug 2022.

[52] Shayan Zamani and Hadi Hemmati. A pragmatic approach for hyperparameter tuning in search-based test case generation. *Empirical Softw. Engg.*, 26(6), nov 2021.

[53] H. Zhong and Z. Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923, 2015.