DeepCode phase 1: Compilers approach

1st Miquel Puig i Mena
Student at Universitat Politècnica de Catalunya (UPC)
Lunds Tekniska Högskola (LTH)
Lund, Sweden
miquel.puig.mena@lu.lth.se

***Abstract*—Yet being immersed in the globalisation era, we can still perceive isolation and rivalry between software developing teams that share ambitions. Nonetheless, DeepCode proposes a cooperation between parties in order to achieve greater results. DeepCode is a tool that, by processing a data-set formed by source codes performing a generic *task A*, learns how to implement *task A* and outperform every individual from studied data-set. Unfortunately, no existing tool is capable of homogenizing a source code data-set in a manner that can be post processed by AI tools. DeepCode *phase 1* targets to fill this gap in the field by offering a standardised data structure representing a generic source code. Furthermore, this paper presents a practical application that generates aforementioned representation given a well-formed source code file.**

***Index Terms*—Abstract syntax tree, Tree traversal, Code auto-generation, Deep Learning.**

I. Introduction

Coders, Group of coders, Departments, Companies or Group of companies compete every day to get the most efficient algorithm for a specific problem. Regardless of the nature of the research, a common denominator can be spotted: they all try to get the best existing ideas on the field and apply some new improvement/s that makes their own implementation the best in a specific set of "tangible" aspects (performance, usability, efficiency, etc.). In that sense, enormous amount of time and effort is spent by each researcher analyzing and mastering the field. Additionally, told hypothetical researcher won't have any kind of assurance that a competitive approach has been discarded during the process.

In this paper, DeepCode is presented, a new tool that will try to auto-generate unique and original code. DeepCode system aims to generate code that solves a generic *task A* by learning, from a large set of different codes, data-set, how to solve *task A*. Notice that each element in the data-set tries, as well, to solve *task A*. Therefore, the concept data-set under same distribution, references a group of different code implementations that perform the exact same job such as compressing a file, processing a signal or solving mathematical paradigms. Furthermore, DeepCode's most innovative objective is the ability to overcome best test-performance reached by any individual from the data-set it's been learning from.

Given the complexity and nature of DeepCode, two stages are defined: *Phase 2*, that, with Artificial Intelligence (AI) techniques, learns which ideas in the data-set imply good code performance and *phase 1*, which studies and builds customised code representation in order to provide *phase 2* an optimised and workable data-set.

This paper deeply describes *phase 1* of DeepCode but not *phase 2*. Nevertheless, *phase 2* is relevant for this paper since it defines what kind of transformation has to be applied to the source code data-set in order to accomplish future intelligent decisions (See Figure 1). In that sense, this paper suggests, builds and provides an original code representation to be utilised afterwards by AI processes.

Most generally, source codes, irrespective of it's original language, can be represented in lower abstractions levels. From machine code[1], unintelligible from human perspective, until low level code such as Assembly. For DeepCode's purposes, a middle level representation of source code must be generated. Such intermediate depiction of the code allows any data-set to be homogenized, allowing, this way, to make fair comparisons among code samples under the same distribution.

Arbitrary code analysis and transformation is accomplished by generating an abstraction of the source code called Abstract Syntax Tree [2] (AST). Generally, AST abstraction format is fed to subsequent compiler's layers such as optimization of code or Assembly code generation. This paper is highly focused in AST generation stage.

Installed Python packages include an in-build Python compiler written in C Programming language. Additionally, Python packages include in-build front-end libraries enabling, from Python code, to access compiler's internals. Among other modules, *ast* [3] front-end can be found, which converts a Python source code into an AST. From *ast* object, a rich AST context is accessible, providing, this way, a great entry point to transform the AST.

By DeepCode's means, tree representation of the code (AST) is very profitable. Due to time consuming processes in *phase 2*, every redundant information in the AST must be eliminated. In other words, internal AST generated by the compiler includes neglectable nodes from DeepCode's perspective. [4] proved that such strategy implies better results against close competitors and, in consequence, their work inspired paper's solution. In that sense, AST generated by the compiler must be simplified as much as possible without losing

[1]Machine code, also called machine language, is a computer language that is directly understandable by a computer's Central Processing Unit (CPU), and it is the language into which all programs must be converted before they can be run. Each CPU type has its own machine language, although they are basically fairly similar. [1]

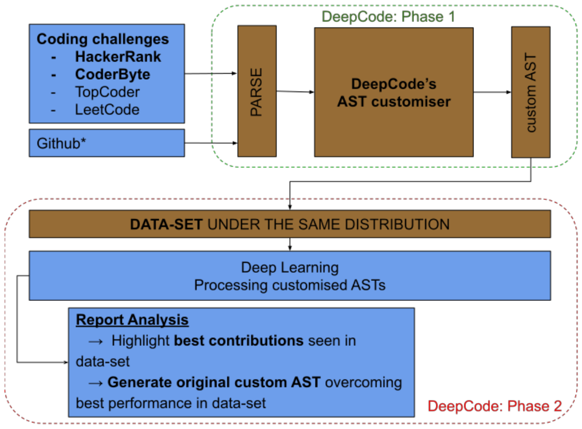
DeepCode: Phase 1
Coding challenges
HackerRank
CoderByte
TopCoder
LeetCode
Github*
PARSE
DeepCode's AST customiser
custom AST
DATA-SET UNDER THE SAME DISTRIBUTION
Deep Learning Processing customised ASTs
Report Analysis
→ Highlight best contributions seen in data-set
→ Generate original custom AST overcoming best performance in data-set
DeepCode: Phase 2

Fig. 1: Architectural schema representing DeepCode's full scope. Notice the contribution provided by each phase of DeepCode to the full project.

valuable information. Hence, above strategy satisfies sought middle level representation of source code.

Summarizing, this paper suggests and provides a compressed and enriched code representation that will be used to feed further post-processes layers within DeepCode scope.

The rest of this paper is organized as follows. Section II introduces similar existing ideas in the field. Section III characterizes AST architecture and suggests a feasible transformation while section IV describes how to achieve customised AST. Section V evaluates results achieved by testing the application against realistic use cases. Suggested extensions to paper's work are found in section VI and, finally, section VII concludes with a summary of report's achievements.

II. Related work

Source code analysis has drastically evolved since AI's reborn. While code processing tools were exclusively based in complex and deterministic techniques, AI provides another dimension to existing tools. By gathering information from user experience, AI enables code processing tools to optimize itself on the fly. Currently, there's a big interest and investment from a well established community. Specifically, a big effort is invested by Integrated Development Environment (IDE) and code analysis applications such as bug finders or providing interesting and personalised hints while coding.

[5] provides a brief text summary of what an inputted code intends to do while [6] provides a tool to learn which non-written conventions and patterns are followed by developers. Both studies have in common that they treat source code as simple text instead of utilising representations such as ASTs or machine code.

Furthermore, finest results were achieved by studies utilising abstracted representations of source code. Code clone detection application [7] overcame state-of-the-art of the moment by extracting AST of it's source code data-set.

An original suggestion is found in [8], where they propose an encoded version of the AST based in [9], that converts tree structures in vectorised information. Great improvement on bug localisation was achieved by [4] by adopting the program representation proposed by [9]. Additionally, [4] included the concept of compressing the AST as much as possible in order to facilitate AI layers to complete their job with optimised results. In consequence, [4] is used as the inspiration to achieve project's objectives.

Commonly, each programming language has it's own AST generator engine and, in many cases, more than one. This paper attempts to corroborate that an intelligent filter in current ASTs can be beneficial for post processing techniques.

Low Level Virtual Machine [10] (LLVM) is a compiler framework that uses a build in intermediate human readable code language with slightly higher level abstractions than Assembly. Although LLVM code can be written from scratch, it's typical to produce it when compiling source code, as intermediate step for code optimization layers and/or optimised machine code generation.

Clang project provides a language front-end and tooling infrastructure for languages in the C language family for the LLVM project [11]. Clang is an open source project well established in the community, having up to 79k commits by the time this report is written. Although Clang is backed by a huge community and provides reliable services, this paper will choose Python compiler [12], a less complex compiler engine system that demands less time to familiarise with.

III. Architecture

Within DeepCode's scope, DeepCode *phase 1* commences once the text data-set is assembled (Figure 2). Proposed tool by this paper handles each element in the data-set individually. In this regard, CLI module provides an entry point to DeepCode *phase 1* where each sample is injected in an iterative way.

Python's inner compiler engine produces an AST that will be trust and used by subsequent modules. Such dependency on Python's compiler core can be assumed given the reliability provided by the existence of a vast community along Python itself.

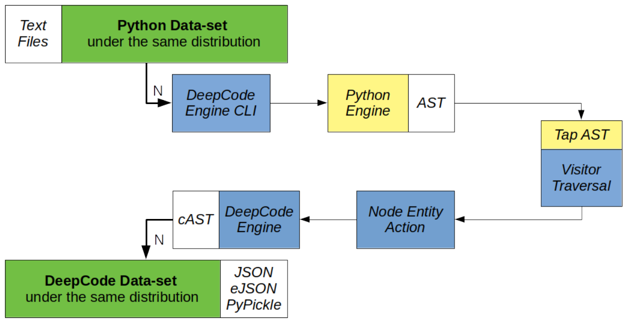

Fig. 2: Architectural model of DeepCode *phase 1*. From gathered python text files[2], DeepCode's command line interface (CLI) triggers transformation cycle for each individual sample in the data-set. Notice that final version of cAST can be outputted in JSON, eJSON, or as PyPickle.

Parallel, Python's engine is bound to report's tool in a way that AST context is shared and accessible. Thereafter, every node in retrieved AST is traversed following a visitor strategy in order to forge pursued custom AST(cAST). While visiting a node, paper's implementation decides which information is relevant from that node, if any at all, and persists it in the cAST. Notice that customised tree carries a one to one connection between cAST nodes and their AST node pair.

Finally, DeepCode *phase 1* application grants to the user the functionality to store or output created cAST in diverse formats. This report contemplates JSON, encoded JSON[3] (eJSON), or pickle format (PyPickle) to be shown at console or saved in a file.

IV. Implementation

This section deeply describes meaningful steps involved in the transformation of the data-set. Notice that told conversion is achieved by applying the same algorithm to each sample inside the data-set, modifying, this way, each individual at a time. By the end of this section, an optimised set of code representations for DeepCode will be accessible.

In order to provide a better depiction of the optimisations suggested by this paper, the reader can find AST's and cAST's final graphical representations extracted from snippet of code (Listing 1) at Figure 3 and Figure 4 respectively.

A. Gathering the Abstract Syntax Tree

AST is a hierarchical representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code [2]. As stated in section II, state-of-the-art tools that process source code utilise AST as their inputted data format.

This project only considers Python AST abstraction, constraining, this way, to only take into account Python source code. Even though it might appear a limitation, it is enough to proof paper's objective.

Python's in-build front-end class *ast* hosts a method *parse()* which takes as input a string and returns an AST tree object. Internally, *ast.parse()* method calls *builtin.compile()* including, as argument, *_ast.PyCF_ONLY_AST*. Named constant implies that only AST representation is calculated while other compiler computations are skipped such as optimization.

Injecting *ast*'s functionalities in report's tool is straightforward by binding them in code. Consequently, a fine AST object and it's context is reachable from DeepCode *phase 1*'s operations.

B. Traversing the AST

Multiple approaches to traverse trees are known. Concretely, this implementation considers recursive visitors technique while other approaches such as AST matchers [13] or cursor traversing [14] are also functional.

By visiting nodes, the reader should think of walking through each node in the AST from north-to-south direction. For each encountered node, the node must be analysed and, afterwards, *visit()* function called for every of it's childs.

Visitor class, inherited from *ast.NodeVisitor*, is defined. Among others, *Visitor* implements *visit_{NODE_ENTITY}()* method for every conceivable node entity in the AST. Moreover, given the case where a node entity couldn't match any pre-defined visit method, *generic_visit()* will match as a last instance. Notice that latter visit method is defined to provide robustness to the implementation and to apply a default logic.

Recursive traversal of nodes enables the *Visitor* class to build up the cAST and provide access to it from other points of the implementation.

C. Filtering nodes

Selecting substantial information from a node is crucial in order to reach report's goals. An intelligent filtering logic is applied in order to maximise compression of an AST without losing important information.

Filtering cycle will be prompt for each traversed node in the original AST. Depending in node's entity and it's near context, different information will be persisted in the cAST. Following, an overview of the filters considered by report's implementation.

1) Disallowing Aliases: Aliases are extremely useful in terms of smoothing the readability of a source code by humans. However, they add a counterproductive link between nodes inside the tree which must be erased.

Visitor class keeps track of aliases assigned in source code and performs a translation to it's original name when persisting information in cAST.

2) Minimizing number of node types in AST: According to [4] implementation, grouping related node types can be very productive for following post-processing techniques. By minimizing the number of conceivable types in the cAST, the natural number of combinations that AI layers have to deal with will decay following equation 1.

$$num\_combinations = N^M \tag{1}$$

Where N is the total number of node types and M is the amount of nodes the tree is composed of.

Followed, a set of the most repeated groups:

* Loop group includes *For*, *AsyncFor*, and *While* node types.
* Name group includes *Name*, and *NameConstant* node types.
* Import group includes *Import*, and *ImportFrom* node types.
* Op group includes *BoolOp*, *BinOp*, and *UnaryOp* node types.

Grouping types strategy entails a relative compression rate of expressions of 0.63 (27 AST expressions and 17 cAST expressions) and a relative compression rate of statements of

[3]Encoded JSON directly depends in final global DeepCode data-set. Depending in the different node types sighted in data-set, the eJSON will cipher accordingly.

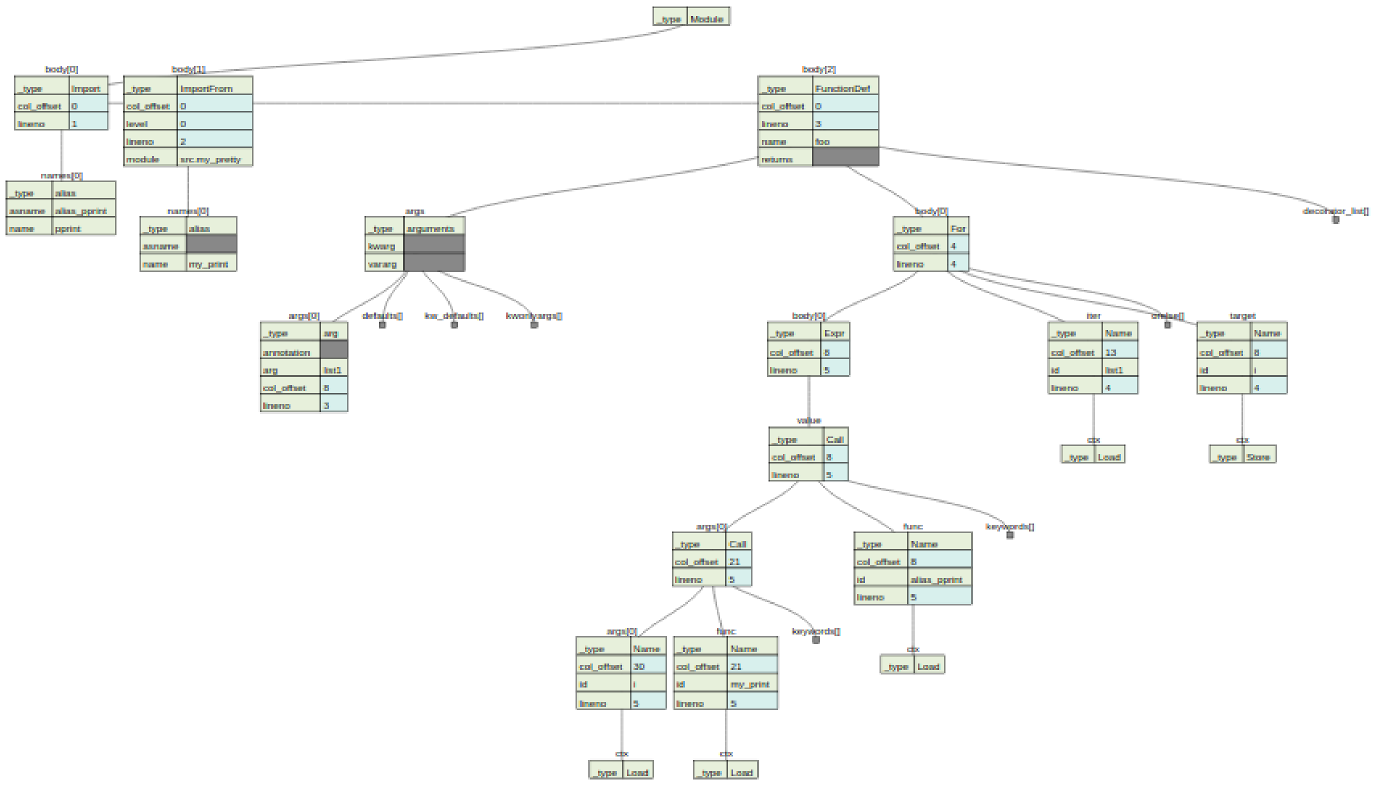
Fig. 3: Representation of test code(1) as AST.

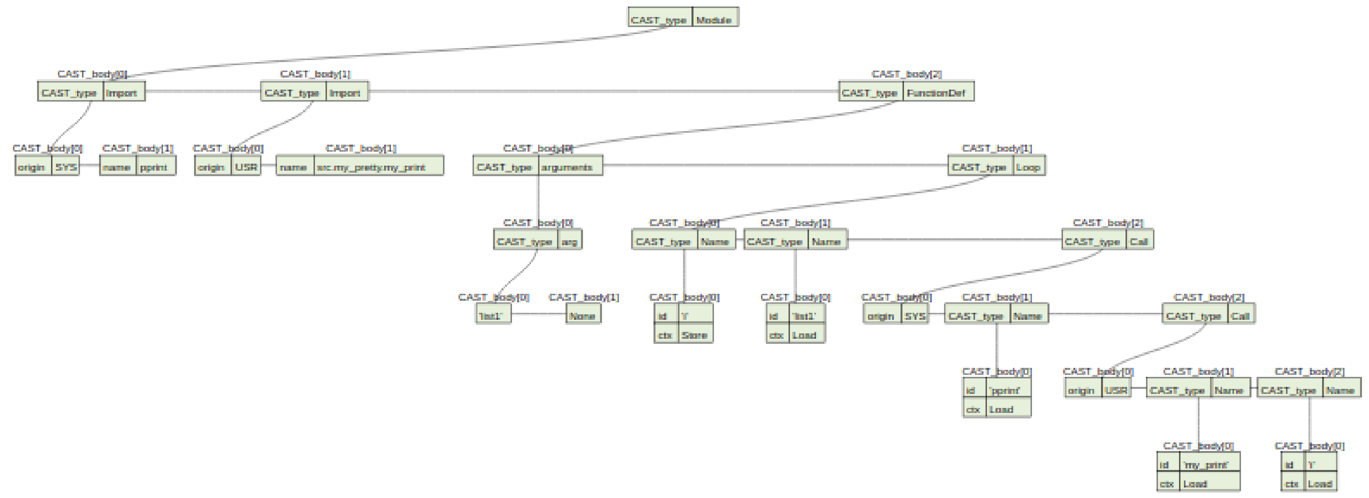
Fig. 4: Representation of test code(1) as cAST.

0.7 (27 AST statements and 19 cAST statements). Notice that it doesn't necessarily lead to an absolute compression rate as the ones stated before since absolute compression rate directly depends on the probability of appearance of each entity. Section V provides absolute compression rates.

3) Minimizing number of nodes in AST: Naturally, compression can be achieved if nodes can be neglected without losing information. Inspired in [4], nodes are neglected following a deterministic strategy.

This paper understands a negligible node as the ones that, once dropped, can be re-injected again by deterministic means. For instance, a function *Call* node entity emerges from an *Expr* node with no other substantial information from post processing layers perspective. In this case, we can conclude that *Expr* node can be neglected from the AST.

```
import pprint as alias_pprint
from src.my_pretty import my_print

def foo(list1):
    for i in list1:
        alias_pprint(my_print(i))
```

Listing 1: Snippet of code chosen to depict an example of *Disallowing Aliases* (IV-C1), *Minimizing number of Types* (IV-C2), *Minimizing number of Nodes* (IV-C3), *Discriminating method Calls* (IV-D1) and *Representing the cAST* (IV-E)

D. Enriching the AST

Yet an enrichment of the AST can contradict the goal of maximising the compression of the AST, this report suggests that some extra information can be exceptionally advantageous for DeepCode *phase 2*.

1) Discriminating method Calls: Knowing the origin of a function call can be convenient in order to decide which nodes should gain more attention or, even, to treat them differently. DeepCode *phase 1* classifies method calls by the ones coming from a method system (SYS), the ones created by the user (USR), and Python buildins methods (NATIVE).

Firstly, the tool has to find potential Python libraries accessible from the system. Such non-trivial task is performed with the help of Python's *modulefinder* package. Once potential SYS and NATIVE method calls are constructed, it's a trivial problem of classifying traversed nodes whose types are directly related to method calls (e.g., *call* or *import* nodes).

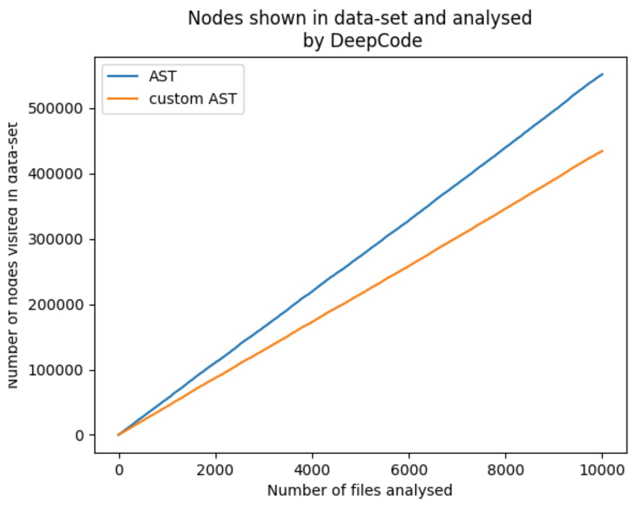

Fig. 5: Analysis of a 10k samples data-set under the same distribution. Depicted, accumulation of number of nodes seen during the analysis of the data-set when formed by ASTs and cASTs.

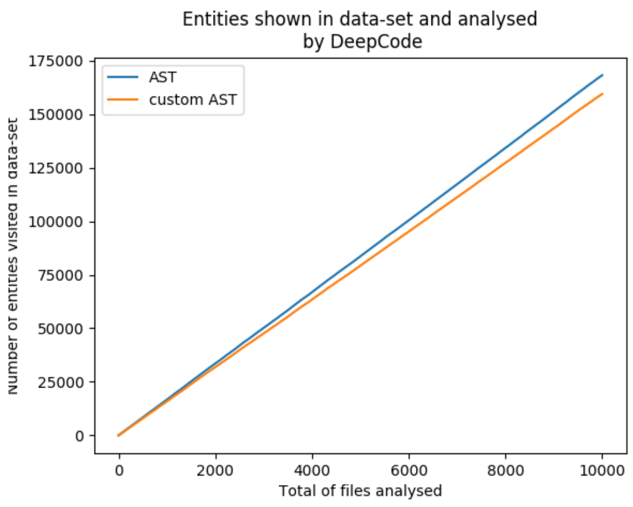

Fig. 6: Analysis of a 10k samples data-set under the same distribution. Depicted, accumulation of number of node types seen during the analysis of the data-set when formed by ASTs and cASTs.

E. Representing the customised AST

Subsequent treatment of cAST by a generic group of consumers can be very heterogeneous. From consumer's point of view, could be interesting to decide which format the tool should output the cAST. Ergo, implemented tool offers three options to specify application's output format.

By default, JSON format is used, which fits the tree representation in to a JSON object. Extending it, an encoded JSON representation can be demanded, which codifies every node representation depending in the importance of such node in the global data-set. In contrast, *PyPickle* option enables a way to output a Python object representation as a *pickle* representation [15].

Lastly, DeepCode *phase 1* application allows the consumer to specify if chosen cAST representation will be outputted in console or written in a file.

V. Experiments

The experiments and evaluations are sequentially divided and sorted by increasing significance to the report.

A. Custom-simple test code

In order to provide an specific example for each of report's implementations, custom source code (Listing 1) is analysed. Figure 3 represents the original AST while Figure 4 illustrates the transformation applied by DeepCode *phase 1*'s core.

Utilising DeepCode *phase 1*'s analysis tool (see results in box below) the report defines that it's been achieved a compression rate of rate_types=0.692, and rate_nodes=0.682.

```
DeepCode - Evaluation - INFO - Analysis AST:
total_types=13, total_nodes=22
DeepCode - Evaluation - INFO - Analysis cAST:
total_types=9, total_nodes=15
```

B. Custom-complex test code

Secondly, this paper tries to adjust difficulty of code by analysing a more elaborated implementation. For that reason, big integer factorising algorithm[4] is studied.

One more time, from DeepCode *phase 1*'s analysis tool (see results in box below), a compression rate of rate_types=0.927, and rate_nodes=0.751 have been reached.

```
DeepCode - Evaluation - INFO - Analysis AST:
total_types=55, total_nodes=1151
DeepCode - Evaluation - INFO - Analysis cAST:
total_types=51, total_nodes=865
```

C. Transforming a 10k files data-set

Finally, this report analyses AST transformation quality in a real data-set under the same distribution. DeepCode includes a web parsing module in order to gather such amount of data.

After studying and comparing every pair of AST and cAST from each individual in the data-set, we can foresee promising results regarding the amount of needed nodes in order to represent the same information (Figure 5). In average, the difference of nodes between AST and cAST extracted from each individual from the data-set is 10.8 nodes. In reference to number of node types, a successful behaviour can be spotted (Figure 6).

The report believes that if a data-set hosting a more complex task was chosen, more nodes and, specially, more node types would be involved. In consequence, more patterns that imply a compression rule would be seen enhancing, this way, higher compression ratios.

VI. Future work

This section suggest two lines of work that can directly benefit paper's implementation.

This report suggests that there's margin of improvement regarding current grouping techniques. Adding new group or fine tuning existing groups might lead to higher compression rate in terms of number of types in cAST.

Moreover, DeepCode *phase 2* tries to discover every *sub task A* that constructs *task A*. When distinguished, each sub task is processed independently with the aim to identify how beneficial, compared to aligned sub tasks in the data-set, the sub task is to accomplish *task A*. From this perspective, custom AST can be enriched in a manner that identifies and divides each *sub task* within *task A*'s code.

VII. Conclusions

This paper has investigated an original manner of assembling an optimised data-set of source codes for future artificial intelligence processing layers. During the study process, a lack of relation between post-processing techniques and the source code text has been spotted. Thus, DeepCode *phase 1* has decided to use Abstract Syntax Tree format as it's source code representation baseline to produce an optimised data-set.

Subsequently, mimicking best results found in the field, report's solution focuses in maximizing the compression rate of the AST whilst keeping intact the information contained in the original tree object. DeepCode *phase 1* tool achieves promising results in terms of absolute number of nodes needed in customised AST against a generic AST extracted from compiler's engine. Additionally, the report introduces a grouping strategy that, with further fine tuning adjustments, can be beneficial for AI related applications.

Lastly, this paper includes the concept of appending auxiliary information to source code representations in order to facilitate and flatten the learning curve of a generic AI system.

In future, new paradigms are worth studying such as subtasking a concrete task utilising deterministic means or statistically studying the impact of entity grouping strategy towards the compression rate and compression quality achieved by the tool. Nonetheless, this paper offers an original bridge for AI systems to work in an optimised manner with source code data-sets.

[4]Find big integer factorising implementation at https://github.com/miquelpuigmena/cryptography/blob/master/factorising/factoring.py.

References

[1] The Linux Information Project. (2006) Machine code definition. [Online]. Available: http://www.linfo.org/machine_code.html

[2] J. Jones, "Abstract syntax tree implementation idioms," *10th Conference on Pattern Languages of Program Design*, p. 26, 2003. [Online]. Available: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf

[3] Python Docs. Abstract syntax trees. [Online]. Available: https://docs.python.org/3/library/ast.html

[4] H. Liang, L. Sun, M. Wang, and Y. Yang, "Deep learning with customized abstract syntax tree for bug localization," *IEEE Access*, vol. PP, pp. 1–1, 08 2019.

[5] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[6] M. Allamanis, E. T. Barr, and C. A. Sutton, "Learning natural coding conventions," *CoRR*, vol. abs/1402.4182, 2014. [Online]. Available: http://arxiv.org/abs/1402.4182

[7] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 87–98.

[8] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[9] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang, "Building program vector representations for deep learning," *CoRR*, vol. abs/1409.3358, 2014. [Online]. Available: http://arxiv.org/abs/1409.3358

[10] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, March 2004, pp. 75–86.

[11] University of Illinois at Urbana-Champaign. Clang: a c language family frontend for llvm. [Online]. Available: https://clang.llvm.org/

[12] Python Docs. Python compiler package. [Online]. Available: https://docs.python.org/2/library/compiler.html

[13] University of Illinois at Urbana-Champaign. How to write recursiveastvisitor based astfrontendactions. [Online]. Available: https://clang.llvm.org/docs/RAVFrontendAction.html

[14] The Clang Team. Traversing the ast with cursors. [Online]. Available: https://clang.llvm.org/doxygen/group__CINDEX__CURSOR__TRAVERSAL.html

[15] Python Docs. Python object serialization. [Online]. Available: https://docs.python.org/3/library/pickle.html