Ph.D. Comprehensive Examination

GUANXUAN WU*, University of Texas at Arlington, USA

PAST RESEARCH TOPICS

- Work in defining a novel type of graph with possibly utilizable properties in control and software engineering - Complex-weighted Structurally Balanced Graph; [27]
- Utilize the Complex-weighted Structurally Balanced Graph to create a novel set of Abstract Semantic Graphs and use it as a representation for Alloy predicates; [26]

QUESTIONS BY JEFF LEI

Paper[28]: C. S. Xia, Y. Wei and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1482-1494,
doi: 10.1109/ICSE48619.2023.00129.

Q1: What is the research problem being addressed?

As mentioned in the original paper, the research problem is evaluating the performances of utilizing the Large Pre-trained Language Models (LLMs) on automatic program repair (APR). Specifically, the paper takes nine existing pre-trained LLMs to utilize for automatic bug fixing and compares them to the state-of-the-art traditional APR methods in three different categories of APR: Complete function generation (take the entire faulty function as input and generate the patch), Correct code infilling (remove the faulty block of code and generate the part given the context), and Single line generation (given a faulty line in the code and generate a patched line). The generated patches are ranked with their entropies (the negative log probability of generating each token) and selected randomly, with the lower entropy patches having a higher priority but using a temperature parameter to select; lower temperature means more likely to select the lowest entropy patch.

Q2: What is the main idea of the proposed approach? Explain both how and why the idea works at a high level, i.e., without referencing low-level technical details.

The main idea of the proposed approach is to utilize the LLMs trained or fine-tuned from the dataset consisting of preexisting codes into APR. It works on a high level since those LLMs were designed to generate code given contexts and proper prompts so that they could be used in patch generation following the training data pattern.

Q3: What is your opinion about the strengths/weaknesses of the proposed approach?

Since a set of general-purposed LLMs trained on code as sequential, text-like data exists, it shows a pattern of plausibility of utilizing those larger models on the tasks specifically for codes. There has been a huge sum of works that considered the code as direct texts to be used in the sequence-based models [3, 6, 8, 11, 15, 19, 31], a survey of specifically trained-on-code (or fine-tuned-on-code) LLMs re-asserts the efficiency of those methods. It produces an experimental result

*UTAID 1002046804

Author's address: Guanxuan Wu, gxw6804@mavs.uta.edu, University of Texas at Arlington, P.O. Box 19015, Arlington, Texas, USA, 76019.

that outperforms all the state-of-the-art non-ML methods in a limited dataset scope. However, the greatest weakness of all the previous research is still prevalent here, that is, for logic-blind models that do not really "understand" the logic between the basic blocks of a code, the generated patterns could be prone to multiple syntactic and semantic errors; as per mentioned in the experiment, even the largest model cannot eliminate uncompilable patches.

Another significant downside of the LLMs for APR is their generalizability in less-used languages or unfamiliar program environments. As shown in [12], their performances on the modeling languages, which are more diversified and logic-strict but with fewer data, could be subpar. It could be partially caused by the logic blindness of the LLMs, which could not capture the first-order logic in Alloy. Still, another root cause is a lack of patterns that could be found in the training dataset for the LLMs; that is, if a language has fewer active users contributing to a public code base, the LLMs would perform subpar on the APR of such language.

QUESTIONS BY CHRISTOPH CSALLENER

Paper[10]: Lample, G., Lachaux, M.-A., Lavril, T., Martinet, X., Hayat, A., Ebner, G., … Lacroix, T. (2022). "HyperTree Proof Search for Neural Theorem Proving." arXiv [cs.CL]. http://arxiv.org/abs/2205.11491

Q1: On the topics in the paper provide (in your own words) a brief background that an interested CS-but-not-ML researcher would need to understand your answer of question (2).

The paper provides a deep learning-based method named "HyperTree Proof Search (HTPS)" for the automatic proof search of formal provers such as Metamath, Lean, and the authors' custom-defined Equations; given a goal lemma to be proved, it looks for proof through the proof tree consisting of achievable, expanded subgoals using encoder-decoder transformers [20]. Consider a goal g; the authors defined a pair of models, a policy model P_θ and a critic model c_θ, where the policy model takes the final (root) goal as its input and selects the operations along the way for the minimized cross-entropy loss of predicted tactics as a standard seq2seq model [18]. The critic model decodes the output vocabulary to two tokens of either PROVABLE or UNPROVABLE, so for each subgoal in the tree, we have $c_\theta(g) = P(PROVABLE|g, CRITIC)$.

To begin with, the authors defined a best-first search policy on the HyperTree for selection that is similar to Monte Carlo Tree Search (MCTS) [1]. It expands the nodes down the HyperTree if they are valid proofs, eliminating cycles or invalid nodes. It utilizes the visit counts of each node as an indicator of the policy model's prior / confidence in the estimated value. At each state within the process of the tree exploration, the visit counts and total action values are recorded. The nodes in the graph are either Solved (at least one tactic leads eventually to an empty set), Invalid (all tactics are rejected or lead to other invalid nodes), or Unsolved (the state in-between), with recursive definitions. To assign the recursive definitions to the nodes in the proof tree, a back-propagation that was defined with a value $v_T(g)$ as the probability of solution for each node g in the tree T was computed in each iteration, whereas $v_T(g) = 1$ if it is solved, $v_T(g) = 0$ if it is invalid, otherwise computed with as the product of the values of its children subgoals. In one particular iteration of the back-propagation on a tree T with a root goal g, with a prior on the critic model, each leaf node is firstly assigned the value by either its solvability or the critic model output, then all parent nodes shall be updated with regard to the products of values of children, applying recursively; then, for each (goal, tactic) pair in T, the visit count is incremented and then the total action value $W(g, t) += v_T(g)$. Thus, for a tactic t on a goal g, there would be an estimated value of $Q(g, t) = W(g, t)/N(g, t)$ where N is the visit count.

The training process would be online and distributed, utilizing a reinforcement learning technique inspired by AlphaZero [14]. Once a prover finishes a proof search, two types of training samples would be extracted from its

hypergraph: the Tactic samples consisting of (goal, tactic) pairs of a minimal proof hypertree of the root node as the training samples for the policy model; and Critic samples, where all nodes that are either solved, invalid, or with a significant visit count higher than a threshold. Define $c(g) = 1$ as the training target for solved nodes, $c(g) = W(g, t^*)/N(g, t^*)$ where t^* is the tactic that maximized the search policy at g, and $c(g) = 0$ for invalid nodes. By arranging the samples into two separate, finite-sized queues, the models could select samples for training batches.

Since the policy model uses a seq2seq approach like many popular LLMs, it was pre-trained on the raw LaTeX data on arXiv. Then, the authors put the supervised dataset of theorems in each environment for fine-tuning. After that, the online training process described above was applied to the models, completing the full pipeline.

Q2: Sketch out one or more ways in which one could potentially integrate your Complex Structurally Balanced Abstract Semantic Graph (CSBASG) work with the HyperTree work, especially in the context of Lean. Describe how one would go about evaluating such new work empirically (including which metrics, datasets, and baselines would be appropriate for such an evaluation). Finally, characterize the scenarios in which you would expect such new work to perform better than the state of the art and give your intuition on why this is the case.

Since modeling languages (Alloy, etc.) and formal provers (Lean, Coq, etc.) are built from first-order logic, applying CSBASG on the latter is intuitive. Note that since CSBASG is a semantic representation built on the code segments intended for a comparison between two program segments, theoretically, the ASG has nothing to do with the proof search tree; however, as mentioned in the paper, the proof search tree grows exponentially with the depth, it is possible to consider a nested graph [34] for the HyperTree representation. One possible thought about utilizing CSBASG is considering each string substitution in Metamath or reasoning step in Lean as a mutation from the last goal.

Consider a collection of defined types $\mathcal{C}$ under Lean, predefined variables in the environment $\mathcal{V}$, and facts of pre-proven lemmas bootstrapped up from the axioms plus the assumptions $\mathcal{F}$. Write the goal in processing g as a formula of first-order logic. Then, g could be written in a CSBASG like we considered a predicate in Alloy; it will have a root expression and a set of subgoals. For example, given in Figure 3 of the original paper, the root goal g is

$$(m\ n\ k : \mathbb{N})(h_0 : n \leq m) : n + k \leq m + k$$

which can be written in a standard first-order logic formula

$$\forall m, n, k \in \mathbb{N}, n \leq m \implies n + k \leq m + k$$

where we have the environment with variables m, n, k in the type of $\mathbb{N}$, equivalent to a *QtFormula* in the implement of AlloyASG; we add m, n, k into the scope of variables $\mathcal{V}$, and then it goes to a *BinaryFormula* with a conditional expression of $n \leq m$ and the result expression of $n + k \leq m + k$. We can create a similar graph representation to an unpacked CSBASG, as shown in Figure 1. So similarly, we can expect that every subgoal g could be easily written as CSBASG.

One approach to the situation is online training inspired by the Nested Graph Neural Network (NGNN) [34], where approaches to predict the edge probabilities like GraphGAN [24] or a path-focused graph neural network such as Graph Transformer Network (GTN) [32] could be used for the outer network. Both inner and outer graphs are rooted - the root goal g_0 and the root node of each ASG.

For an example sketch of an implementation using CSBASG in this automatic proof objective, we define the critic model $c(g)$ as a preference ranking of going to node g. An empty ASG that showed a proven ground truth could be

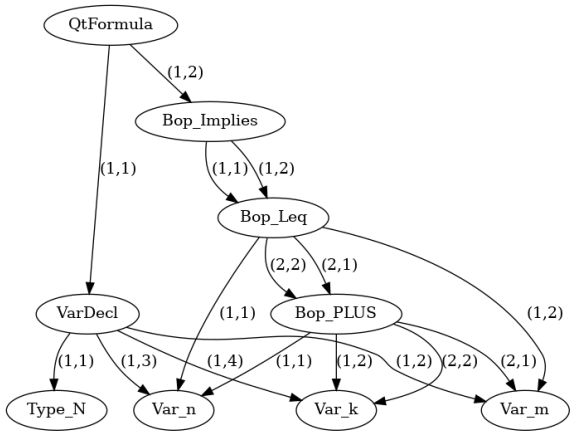

Fig. 1. An illustration of the edge-unpacked ASG for the Lean example as Figure 3 in [10]. (t, j) as the edge labels mean its t-th visit to the ASG node, and the relation between the parent and the child node is at j-th position.

assigned all the edges in the outer graph pointing to it with probability 1 and being a leaf / absorbing node ($c(\emptyset) = 1$). At the same time, invalid ASGs should be leaf nodes with probability 0 for all edges pointed to it.

A set of valid formal proofs $\mathcal{P}$ could be used as the initial dataset. In the pre-training phase, we could assign each node in the proof paths a straight $c_0(g) = 1\ \forall g \in \mathcal{P}$. A simple data augmentation could create invalid formulae by killing mutants g' of the intermediate results within the set of valid formal proofs, $\mathcal{P}'$, with $c_0(g') = 0\ \forall g' \in \mathcal{P}'$. From these two categories, by utilizing a graph regression method like GCN [9], we could have an initial model c_0 for the probability of the existence of a proof for a formula CSBASG. Then, in the online reinforcement learning phase, for each recursively defined goal g, we begin the selection process by finding all its children in the proof tree, remove any that looped back to a previous goal, and rank them with the critic model outputting probability of a valid proof $c(g_i)$.

Since each subgoal could only have three states: solved (leads to an empty set), invalid (leads to an invalid ASG), or unsolved, we could complete the expansion phase to see if the selected approach leads to an empty set or an invalid node. Then, the back-propagation could be done similarly to Section 4.3 of the original paper [10].

An advantage of this approach is removing the dependency of the Seq2seq models and the pre-training process on the text dataset (simply noise for the codes) to reduce overfitting. Since CSBASG was defined as a representation without loss of information and in a formal proof environment, there is usually a reasonable total number of semantic symbols, a logic-aware, graph-based model could perform better than the original paper by reducing the number of tokens. However, a no-loss-of-information data model might take too much memory space for some large-scale arguments that could be put in formal verification.

QUESTIONS BY ISHFAQ AHMAD

Research Plan Involved

(1) **A Laplacian Matrix for ASG**: By definition, the ASG is directed with a loop. Previous work only mentioned the Laplacian matrix of directed graphs without loops or undirected graphs with loops [2]. Since the structural balance requires checking with a Laplacian matrix that grants its properties, it is vital to develop the scheme.

(2) **An Optimized Representation Vector**: In the ASG representation, there is a requirement for the syntactic elements to have their unique signature within $(-\pi, \pi)$. In the current approach, we only did this by a static assignment, which 1) cannot prevent the identical differences between different pairs of nodes and 2) may not intuitively indicate the possibilities of the linkages between each node. Optimizing the representation vector requires a huge dataset of real-life, well-built Alloy models to get a pattern that could be universally applied to Alloy code segments.
(3) **LSM vs LLM for Automated Repair**: We want to evaluate the possibility of utilizing machine learning models to automatically repair Alloy formulas. While there are already some Alloy code interpretation capabilities for the current LLMs, the performance is currently subpar [12]. Another method could be training a language-specific model (LSM) for Alloy and Alloy only, possibly utilizing the ASG definition above for a graph-based model.
(4) **Alloy Code Generation with ASG**: We plan to explore how to predict the most probable node after a given Alloy abstract semantic node written by a programmer.

Q1. For each of the above topics, describe what is the main computational challenge, what optimization schemes you plan to use (cite literature), and identify how you wish to improve that and for what performance measures.

A1.1. A Laplacian Matrix for ASG. Technically, this has nothing to do with computational aspects, but it requires a novel mathematical construct to make the graph networks such as GCN [9], GAT [21], or GTN [32] work. In the classic definition in the Spectral Graph Theory [17], a Laplacian matrix for a directed graph is defined as $L = D - A$, where D is a diagonal matrix with $D_{ii} = \sum_{j=1}^{N} |a_{ij}|$. Even with the more primitive definition, $\mathbf{x}^T L\mathbf{x} = \sum_{(a,b)\in E} w(a,b)(\mathbf{x}(a) - \mathbf{x}(b))$, auto-edges are still irrelevant. Some discussions have been about Laplacians of undirected graphs with self-loops [2, 13, 23]. Still, since a directed graph with complex edge weights is far less mathematically preferable than the symmetric, positive semidefinite adjacency matrices of the undirected graphs, it is challenging to utilize the properties of structural balance [27].

A1.2. An Optimized Representation Vector. In common graph networks like GCN [9], the nodes are vectors, but now we are trying to map it to a complex number. Since by the original working definition of the Complex-weighted Structurally Balanced Graphs [27], the adjacency matrices have the entries $A_{ij} = |w(i,j)|\angle(\theta_i - \theta_j)$, so the difference of the signatures determines the direction of the edge on the complex plane; that could sometimes confuse an interpreter if the types of the nodes are not pre-recorded as in the experiment done in last paper [26]. In the paper, we proposed a baseline of optimization considering the multigraph form:

$$\min \sum_{i,j=1\ldots N} \mathbb{I}((v_i, v_j) \in \mathcal{E})|\theta_i - \theta_j|, \forall i \neq j : \theta_i - \theta_j > 0$$

A more reasonable modification, considering the multiedges, is then

$$\min \sum_{(v_i,v_j,k)\in\mathcal{E}} |\theta_i - \theta_j|, \forall i \neq j : \theta_i - \theta_j > 0$$

where (v_i, v_j, k) is the k-th edge drawn between v_i and v_j. However, it raises a huge concern about a possible trivial solution of near-zero equilibrium. So, a common approach should be another optimization constraint that pushes the

distances away if an edge is not present at each time of evaluation:

$$\max \sum_{i,j=1\ldots N} \mathbb{I}(\forall k : (v_i, v_j, k) \notin \mathcal{E})|\theta_i - \theta_j|$$

After training this with some regularization, we could consider the probability of an edge between two nodes inversely proportional to their angular distances. This creates a non-trivial problem, and one intuitive solution could be dual-annealing [29, 30], which is now, fortunately, a built-in package in SciPy [22].

A1.3-4. LSM vs LLM for Automated Repair and Alloy Code Generation with ASG. The two topics are combined here because Automated Repair is essentially supervised code generation to infill the buggy holes in a code segment like the ARP with LLM paper [28]. A previous review of multiple code generation methods [33] has shown that code generation or APR is now largely confined to two approaches: Language-specific or multi-language models with the NLP-esque sequence-based techniques [3, 6, 8, 11, 15, 19, 31] that ignoring the internal logic of codes, or directly apply LLMs [16, 28]. Our LSM would follow a solution in the last subquestion utilizing the graph algorithms and the LLMs as a benchmark. Since it was shown that LLMs perform poorly on Alloy [12], additional fine-tuning is a solution to improve the baseline of the LLMs, and there have been some state-of-the-art models such as CodeGeeX [35] on general-purpose languages, we could see if a fine-tuning on a set of Alloy models could improve its categorial performance in Alloy and even other specification languages.

Q2. Also, these future plans seem to be disjoint, each presenting its own research threads. What is their interrelationship? Which one is a higher priority and why.

The ultimate goal for these plans is toward a common goal of code generation - either patch generation for APR or directly generating code from scratch (Plan Objectives 3 and 4). We could consider a per-tier approach for a comparison of different machine-learning-based constructions toward this common goal:

- Tier 1. Shallow model by dual-annealing with the minimizer and maximizer representation vector optimizations, as mentioned in A1.2. Directly looking for the syntactic and semantic elements within the circle on the complex plain could be a method to complete the incomplete code segments. Still, it is only likely to work as an unsupervised model to generate random Alloy codes given some temperature that determines the probabilities of the next node.
- Tier 2. Deep, graph-aware models which are expected to be focused. Since almost all graph neural networks are built dependent on the Laplacian matrix, and so is the concept of "mutant as a control operation" that we expect to be shown with the CSBASG. However, the existence of such a construct is not proven yet, and Plan Objective 1 could be completely unachievable. In that case, there is an alternative: creating mirrored twin nodes v, v' to capture the self-loops and create a ping-pong topology for the adjacency matrix, where if there is a self-loop for v, there will be an edge from v to v' and another nested self-loop will result in an edge from v' to v. This is the last resort but still lays the foundation for the Tier 2 models to run. However, with a severely more complicated graph representation and increased graph size, it consumes more resources and becomes slower. And finally, the improved vector-of-nodes representation is still useful in these models.
- Tier 3. LLM Fine-Tuning. This is a relatively small and repeatable work utilizing the mature techniques in the frontiers of industrial fine-tuning of any LLMs. However, it could still serve as a baseline since even the GitHub code fine-tuned models still relatively underperform on specification languages.

(a)

```
sig Process {}
sig Processor { var running: one Process }
fact noConcurrency {
   all disj cpu1, cpu2: Processor |
   always cpu1.running != cpu2.running
}
```

(b)

```
pred changeProcess {
   one cpu: Processor |
   some disj p1, p2: Process |
   cpu.running = p1 &&
   eventually (cpu.running != p1) &&
   eventually (cpu.running = p2)
}
```

(c)

```
pred changeProcessC {
   one cpu: Processor |
   some disj p1, p2: Process |
   cpu.running = p1 &&
   after (cpu.running != p1) &&
   eventually (cpu.running = p2)
}
```

(d)

```
pred changeProcessO {
   one cpu: Processor |
   one p: Process |
   cpu.running != p &&
   after (cpu.running = p)
}
```

Fig. 2. An Alloy model of a simulation of a multi-processor computer, whereas a processor can only hold a single process at a time. The changeProcessO predicate is defined as an oracle solution for modeling that a processor changes the process running within it over time, and changeProcess is another attempt that appears to be faulty; the corrected version is changeProcessC, which is equivalent to changeProcessO.

The higher priority is to prove or disprove that a well-formed Laplacian matrix exists for the CSBASG with self-loops without losing the self-loop weight information, that is, Plan Objective 1. Only could we consider the Tier 2 models deemed to have the most potential after either a working Laplacian matrix for the CSBASG or circumventing this problem.

QUESTIONS BY ALLISON SULLIVAN

Paper[5]: Jorge Cerqueira, Alcino Cunha, and Nuno Macedo. 2022. Timely Specification Repair for Alloy 6. In Software Engineering and Formal Methods, Bernd-Holger Schlingloff and Ming Chai (Eds.). Springer International Publishing, Cham, 288–303

Q1: Explain using an example how TAR works. Include a statement on what you view as the strengths and limitations of the technique's ability to repair models.

Consider an Alloy model written as a program scheduler on an abstract parallel computer. Each processor could execute one program at a time and switch to the next when the program terminates, but cannot be truly idle without a process, as shown in Fig 2. In this particular example, the faulty method changeProcess cannot cover the case where there are two processors and three processes. When at the initial state (time 0), Process 2 is assigned to Processor 0 and Process 1 to Processor 1 (denoted (2@0, 1@1), similar below), then at time 1, (0@0, 1@1), and time 2 (1@0, 0@1). In this instance, let p1 be Process 1, p2 be Process 0, and the variable CPU be Processor 1 in the predicate changeProcess, and let the predicate changeProcessO to be focused the same processor and Process 1, then this condition satisfies changeProcess but not changeProcess0; the oracle solution of Fig. 2(d) requires that the processor change its processing job at the next cycle, but a faulty-written solution Fig.2 (b) could be evaluated True if "eventually" switch its job in the future, even if it is a cycle later than it should be.

TAR could fix the faulty predicate changeProcess to changeProcessC with a generated mutant inside the formula, particularly by searching over the code and replacing the unary temporal logic expression "eventually" at line 5 of the code to another unary expression "after", ensuring that the processor stop running process p1 at the next cycle to make the predicate hold. TAR works relatively well in this situation, since one single replacement mutation could repair the model to correctness. However, there are five unary operators and five binary operators in the small code segment given here, and a near-exhausive search with a record of counterexamples only could already be tens of calls to the SAT

solver. Since a complete mutation testing is undecidable [4] and each SAT solving is NP-complete, the performance is expected to be even lower with larger segments of Alloy predicates.

Q2: What do you think are the strengths and weaknesses of this technique? You can use the evaluation results to help support your observations, as well as your own insights.

The evaluation results is obviously outperforming the preexisting methods of ARepair [25] and BeAFix [7], both are prior state-of-the-art AST mutation based methods. One of the advantages of TAR is recoding of the counterexamples for the future use to test with mutants, which reduces an impressive number of calls to the AST solver, thus improving the performance. However, it is still simply mutation testing based and the undecidable nature of it, any problems of mutation testing still appears in TAR, especially combining with the NP-complete SAT solving. TAR also did not collect a pattern of the successful repairs for a reference or a prior in tactic selection at mutation operations, making a fresh generation on each run, which could also increase the processing time for a larger batch of faulty Alloy predicates.

Q3: One of the main goals of this work is to provide quick hints to the user to help guide them on Alloy4Fun exercises. What limitations does this technique have that a hint generation approach using the AlloyASG representation could solve?

One understandable problem of significance that goes with the conventional mutant-based automatic fixing is the difficulty for training or otherwise summarizing a pattern from prior results for the future, and it also happens to be a concern in hint generation as those mutant patches are not generalizable for a lack of comparison and explainability. AlloyASG would allow a hint generation with an even more mutated code segment: since there is an oracle solution and a pool of the correct answers that could be added into the database by the submission from the public domain, it is possible to find the nearest correct answers and give a detailed list of steps that each could be represented as an addition, removal or replacement of an edge in the ASG toward it. By giving out such a step each time the student request a hint, it could further improve the applicability and quantity of the generated hints in an educational settings.

REFERENCES

[1] Bruce Abramson and Richard Korf. 1987. A Model of Two-Player Evaluation Functions. 90–94.

[2] Behcet Acikmese. 2015. Spectrum of Laplacians for Graphs with Self-Loops. arXiv:1505.08133 [math.OC]

[3] Abdulaziz Alhefdhi, Khanh Hoa Dam, Xuan-Bach Dinh Le, and Aditya K. Ghose. 2020. Adversarial Patch Generation for Automatic Program Repair. *ArXiv* abs/2012.11060 (2020). https://api.semanticscholar.org/CorpusID:260510563

[4] Timothy A. Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta Informatica* 18, 1 (01 Mar 1982), 31–45. https://doi.org/10.1007/BF00625279

[5] Jorge Cerqueira, Alcino Cunha, and Nuno Macedo. 2022. Timely Specification Repair for Alloy 6. In *Software Engineering and Formal Methods*, Bernd-Holger Schlingloff and Ming Chai (Eds.). Springer International Publishing, Cham, 288–303.

[6] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[7] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo Frias. 2021. BeAFix: An Automated Repair Tool for Faulty Alloy Models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1213–1217. https://doi.org/10.1109/ASE51524.2021.9678524

[8] Shan Huang, Xiao Zhou, and Sang Chin. 2021. Application of Seq2Seq Models on Code Correction. *Frontiers in Artificial Intelligence* 4 (2021). https://doi.org/10.3389/frai.2021.590215

[9] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[10] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. 2022. HyperTree Proof Search for Neural Theorem Proving. arXiv:2205.11491 [cs.CL]

[11] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[12] Sanyogita Piya and Allison Sullivan. 2023. LLM4TDD: Best Practices for Test Driven Development Using Large Language Models. arXiv:2312.04687 [cs.SE]

[13] Ugasini Preetha P, M. Suresh, and Ebenezer Bonyah. 2023. On the spectrum, energy and Laplacian energy of graphs with self-loops. *Heliyon* 9, 7 (Jul 2023). https://doi.org/10.1016/j.heliyon.2023.e17001

[14] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815 (2017). arXiv:1712.01815 http://arxiv.org/abs/1712.01815

[15] D. Sobania, M. Briesch, C. Hanna, and J. Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE Computer Society, Los Alamitos, CA, USA, 23–30. https://doi.org/10.1109/APR59189.2023.00012

[16] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. arXiv:2301.08653 [cs.SE]

[17] Daniel A. Spielman. 2012. *Spectral Graph Theory*. https://api.semanticscholar.org/CorpusID:114054711

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. *CoRR* abs/1409.3215 (2014). arXiv:1409.3215 http://arxiv.org/abs/1409.3215

[19] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (sep 2019), 29 pages. https://doi.org/10.1145/3340544

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]

[22] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[23] Deekshitha Vivek Anchan, Sabitha D'Souza, H.J. Gowtham, and Pradeep G. Bhat. 2023. Laplacian Energy of a Graph with Self-Loops. *Match Communications in Mathematical and in Computer Chemistry* 90, 1 (2023), 247–258. https://doi.org/10.46793/match.90-1.247v

[24] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2017. GraphGAN: Graph Representation Learning with Generative Adversarial Nets. *CoRR* abs/1711.08267 (2017). arXiv:1711.08267 http://arxiv.org/abs/1711.08267

[25] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2019. ARepair: A Repair Framework for Alloy. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 103–106. https://doi.org/10.1109/ICSE-Companion.2019.00049

[26] Guanxuan Wu and Allison Sullivan. 2024. AlloyASG: Alloy Predicate Code Representation as a Compact Structurally Balanced Graph. arXiv:2403.00170 [cs.SE]

[27] Honghui Wu, Ahmet Taha Koru, Guanxuan Wu, Frank L. Lewis, and Hai Lin. 2023. Structural Balance of Complex Weighted Graphs and Multi-Partite Consensus. *IEEE Control Systems Letters* 7 (2023), 3801–3806. https://doi.org/10.1109/LCSYS.2023.3341992

[28] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[29] Yang Xiang, Sylvain Gubian, Brian P. Suomela, and Julia Hoeng. 2013. Generalized Simulated Annealing for Global Optimization: The GenSA Package. *R J.* 5 (2013), 13. https://api.semanticscholar.org/CorpusID:10302429

[30] Y. Xiang, D. Y. Sun, W. Fan, and X. G. Gong. 1997. Generalized simulated annealing algorithm and its application to the Thomson model (in Chinese). *Physics Letters A* 233, 3 (Feb. 1997), 216–220. https://doi.org/10.1016/S0375-9601(97)00474-X

[31] Geunseok Yang, Kyeongsic Min, and Byungjeong Lee. 2020. Applying Deep Learning Algorithm to Automatic Bug Localization and Repair. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1634–1641. https://doi.org/10.1145/3341105.3374005

[32] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. 2020. Graph Transformer Networks. arXiv:1911.06455 [cs.LG]

[33] YANG ZeZhou, CHEN SiRong, GAO CuiYun, LI ZhenHao, Li Ge, and Lv RongCong. 2024. Deep Learning Based Code Generation Methods: A Literature Review. *Journal of Software* 35, 2 (02 2024), 604. https://doi.org/10.13328/j.cnki.jos.006981

[34] Muhan Zhang and Pan Li. 2021. Nested Graph Neural Networks. arXiv:2110.13197 [cs.LG]

[35] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv:2303.17568 [cs.LG]