Learning to Walk over Relational Graphs of Source Code

Pardis Pashakhanloo[1], Aaditya Naik[1], Hanjun Dai[2], Petros Maniatis[2], Mayur Naik[1]

[1]University of Pennsylvania, [2]Google Brain

Summary

- Information-rich relational graphs have great potential in designing source code representations. Existing models, however, struggle to handle the wealth of information due to their limited context size. Sampling random walks over program graphs is useful in addressing this challenge.
- We propose a deep learning technique to capture relevant context over large program graphs. This technique improves upon random walks by **learning task-specific walk policies** that guide the traversal of the graph towards the most relevant context.
- Models that employ learned policies for guiding walks are **6-36%** points more accurate than models that employ uniform random walks, and **0.2-3.5%** points more accurate than models that employ expert knowledge for guiding the walks.

Designing a Walk Policy

There are different strategies for sampling walks over relational graphs, for example:

	Uniform Random Walks	Expert-guided Walks
Pros:	Simple calculation; No dependency on expert	Fewer unrelated walks; Interpretable results
Cons:	Combinatorially large space of possible walks	Overlooking unintuitive choices; Task-dependent

Proposed solution: learn a policy to guide the random walks.

- We can take advantage of *properties of relational graphs* to design such a policy.
- The key components of relational databases have counterparts in relational graphs.

Tuples	⟷	Node values
Relation names	⟷	Node types
Key-Foreign Key relationships	⟷	Edge types

Walk Policy: we define a walk policy as a mapping from each node type to a score, proportionate to its relevance to a specific task. These scores specify the next step in a random walk.

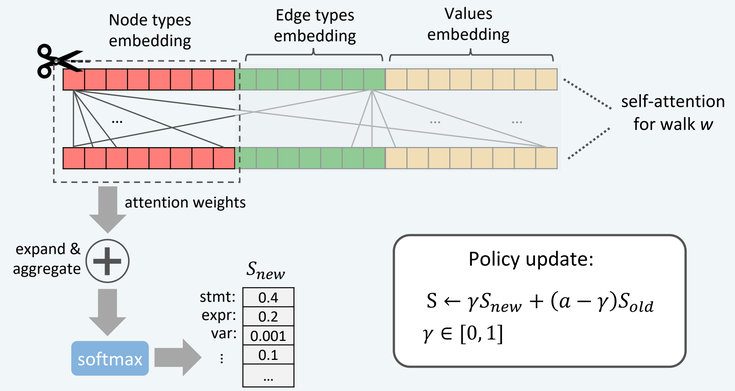
Learning a Walk Policy

- *Walks are latent*. They are unobserved and lack supervision.
- A learned policy improves the *overall predictive performance* of the model.

Walk learning can be viewed as an expectation-maximization (E-M) algorithm.

E step estimates the walk policy based on the current state of the model.

M step improves the current model based on the estimated walk policy.



Policy update:

$$S \leftarrow \gamma S_{new} + (a - \gamma) S_{old}$$

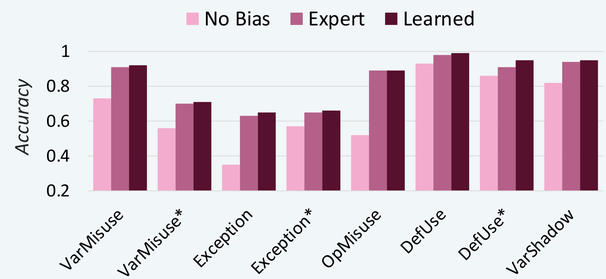
$$\gamma \in [0, 1]$$

This figure illustrates the **E step** where the policy is updated based on the aggregated relevance of relations in previous iterations of training.

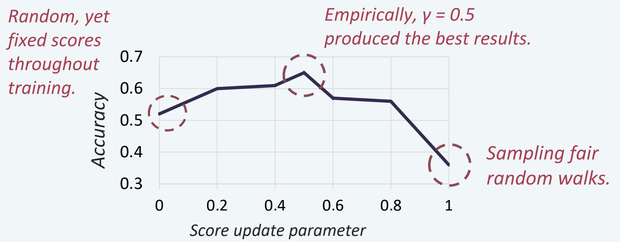
Results

Learning walk policies increases the accuracy.

Learned policies are consistently better than policies set by domain experts in bug classification tasks.



The value of the score update parameter (γ) should not be too close to 0 or 1.



Discussion and Future Directions

Memory

The current mechanism for learning walk policies lacks memory which might hinder its ability to capture complex patterns.

Anchors

The current mechanism requires the domain expert to specify the anchor nodes. One could instead walk from the root (i.e., the top-level node in the program graph which corresponds to "module" or "function") and learn better anchor points from there.

Open questions:

- How many nodes should be selected as anchors?
- Should all anchor nodes for a particular task have the same relation names (i.e., node types)?

Scope of the Policy

- There are cases when a walk policy should learn edge scores rather than node scores. For example, a "func" node may have different relevance/importance depending on whether it is a caller or a callee.
- One could overcome this shortcoming by integrating edge values, names, and walk lengths into the walk policy.

Interpreting Learned Policies

- VarShadow and DefUse can be solved via CodeQL queries.
- Interestingly, the highest-ranking relations are matched with tables used in corresponding CodeQL queries.
- Using learned random walks to represent programs is a promising direction toward more interpretable models for code-understanding tasks.