Automated Evaluation of Programming Assignments

Report

Rishabh Manoj

January 10, 2018

1 Introduction

The assessment of programming assignments places significant demands on the instructor's time and other resources. This demand has led to the creation of automatic evaluation systems. Students are given programming assignments which are assessed using automatic program evaluation systems. These systems provide a possibility for the student to submit a solution, test it in some way and receive feedback. Such systems could also provide feedback for the teachers and evaluators.

One naive method which is commonly used is to manually create a set of test-cases that can be used to assess the solutions. While these are effective in identifying whether the solution is correct or not, this will not give an indication to how similar the student's solution is to the correct solution.

In this paper we come up with a method that will try to answer the above question.

2 Definitions

Symbolic Testing

Symbolic testing is the one of the program analysis technique used for proving program correctness. This technique executes the programs using symbolic values instead of concrete input data and represents the program variables as symbolic expressions. Therefore, we are able to represent the output of the program as a function of symbolic values. The most widely used application of symbolic testing is the automated test case generation for proving program correctness.

Program Dependence Graph:

Program dependence graph (PDG) is a graphical representation of the source code of a program. Basic statements, like variable declarations, assignments, and procedure calls, are represented by program vertices in PDG's. The data and control dependencies between statements are represented by edges between program vertices in PDG's.

Graph Isomorphism

A graph can have the different form which contains the same number of vertices, edges and vertices are connected in the same way are said to be isomorphic graphs.

Formally, the graphs G and G' are isomorphic, then there exists a function $F : V(G) \rightarrow V(G')$ such that

i. f is a bijection.

ii. f preserves adjacency of vertices. i.e. if the $(u, v) \in E(G)$ then $(f(u), f(v)) \in E(G')$

γ Isomorphism

A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G' , and $\|S\| \geq \gamma\|G\|$, $\gamma \in$ (0,1). γ is said to be the mature rate of isomorphic subgraphs. Initial γ is set based on a belief on how much the subgraphs would be isomorphic. Value of γ helps in pruning the search space for isomorphism. In other words, value of γ would limit the number or vertices which would be subgraph isomorphic to the other graph.

3 Proposed Method

3.1 Gold Standard Solutions:

In this module we separate the students' solution into Correct and Incorrect submissions.
We are initially given all the students' solutions and one reference solution. Using the reference solution we automatically generate a set of test-cases that covers all the corner-case of the assignment and test it against the students' solution. The solutions that passes all these test-cases are considered correct and the others are considered incorrect.

We generate test-cases using Symbolic Testing [5] tools like *CREST*, *API Sanity Check*, etc...

3.2 Similarity Measure:

In this module, we calculate the similarity measure of the solutions with each other which is then later used for clustering.

The central idea is to generate *Program Dependence Graphs* [4] for the solutions and check for sub-graph isomorphism between any two PDG's. If it is sub-graph isomorphic then based on the number of vertices mapped we calculate similarity. (No. of mapped vertices of Graph G1 / Total no. of vertices of Graph G2) [G1 sub-graph isomorphic to G2]

PDG's are generated for each function. In this instance we assume that the datset contains only one function(main) and genearte PDG's for main functions.

We use *frama-c* software to generate PDG's of the solution. This creates a DOT file for each of the function. To parse this, we use a DOT parser in python to get a grpah with vertices and nodes.We use γ-isomorphism method to calculate similarity. Thisγ-isomorphism method checks for sub-graph isomorphism between the PDG's and calculates similarity as mentioned above.

Summary:

- Create PDG's using frama-c for each solution.
  - DOT files for each function
  - *frama-c -pdg -pdg-dot filename c-programfile* .
- Extract graph nodes and edges from DOT file using a DOT parser in python
- Calculate similarity measure using γ-isomorphism as mentioned above.

Output of this module is similarity measure of each pairs of solution ranging from 0 to 1 with 1 being high similarity and 0 being no similarity.

3.2.1 Implementation Details:

- For each program, PDG has to be generated, we use *frama-c* to generate a DOT file which contains the PDG.
- A DOTParser for python was used to extract nodes and edges from the DOT file. This was converted to a NetworkX graph in python.
- Sub graph isomorphism algorithm was used to calculate similarity.

3.2.1.1 Program Dependence Graph

Frama-c is a set of interoperable program analyzers for C programs. We use *Frama-c* to generate dependency graph in the form of DOT files for C programs. While it generates for most of the C programs it does not encompass the entire C language.

Frama-c (Flourine version) the stable release for Ubuntu 14.04 does not work with Variable Length array, the latest release of Frama-c (Phosphorous) while it works with single dimensional variable length array it doesnot for multi-dimensional arrays.

The output of Frama-c's pdg generation is a DOT file which contains the nodes and edges of the PDG. DOT files are hard to interpret and contains unnecessary details, so a DOTParser was used to get only the nodes and edges. This was then used to instantiate a NetworkX graph (a python library for graphs) which was selected for it's vast graph related library.

3.2.1.2 γ-isomorphism

We follow the algorithm provided in [7] named γ-isomorphism as it uses a γ-filtering technique to eliminate certain graphs before performing subgraph isomorphism.

Sub graph Isomorphism is an NP-Complete problem which means that there is no polynomial time solution. (Note that Graph Isomorphism is neither P nor NP-complete but it's generalisation(sub-graph isomorphism) is NP-complete). Over the years there have been many heuristic solutions that attempt to solve this for a specific set of graphs like the Glasgow algorithm [8], the VF2 algorithm [3] and recently the LAD filtering algorithm [6].

Initially the *VF2* algorithm was implemented as it was used in the algorithm mentioned in [7], we found that the algorithm is very time-consuming, while for graphs with very less nodes (in the range of 20 to 30) it calculates in seconds, the higher the vertex count (usually in the range of 100's which is the case for our dataset) it sometimes takes days to calculate!!

Due to this the *VF2* algorithm approach was abandoned and the latest development in sub graph isomorphism the LAD filtering approach was chosen. The latest variant, *pathLAD* [6], was used. A time limit was set which can act as a threshold as to when the search can be discontinued.

While the LAD algorithm works much faster than *VF2*, there are still cases where sub-graphs could not be found within the time-limit. In some cases increasing the time-limit by a few seconds leads to solutions where it would not have found in the decreased time-space. This leads to drastic change in the output of the next phase.

3.2.2 Suggestion:

Due to the unreliability of sub graph-isomorphism, find other similarity metrics like LOC, Function count, running time, memory usage, $\cdots$. Use a weighted average of these metrics to calculate a similarity index. This will lead to less dependence on graph isomorphism which will not drastically alter the overall output.

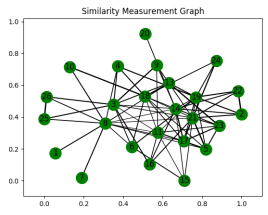

Figure 1: Similarity Measure Ouput

3.3 Clustering:

Graph clustering involves the task of dividing nodes into clusters, so that the edge density is higher within clusters as opposed to across clusters. Graph Clustering is a deeply studied problem as there are many use cases in biochemistry especially in clustering protein complexes.

In this module we cluster similar solutions together to form some distinct types/methods/algorithms of solutions possible for the given problem.

The output from the above Similarity Measure module is considered as an edgelist in this module. The program files become the nodes and the similarity measure becomes the weights of the edge between them.

Using this graph, we start clustering. Idea is that nodes with edges whose similarity values are high will cluster together.

The output of this module are clusters which contains program files.

3.3.1 Implementation Details

We use various clustering algorithms and compare their clusters to some human annotated clusters.

3.3.1.1 Louvain modularity

We use a method called Louvain modularity [1] which is a greedy optimization method that appears to run in time $O(n \log n)$. In this method, first small communities are found by optimizing modularity locally on all nodes, then each small community is grouped into one node and the step is repeated.

Observation: Nodes within Clusters sometimes do not have high edge weights, while one node will be highly connected to everything the other nodes are not so strongly connected to each other node. *Possible explanation:* Since similarity values are calculated based on isomorphism which is NP-complete, the values for some pair of nodes might actually be high but since it cannot be found out in the stipulated time interval, the values are given zero. While clustering, if the neighbours of the nodes are highly similar then it is also highly likely that the node is similar to the rest of the others.

For eg: For example there are 4 nodes 1,2,3 & 4 if 1,2,3 are highly similar and 3,4 are also highly similar but the values for 1,4 and 2,4 are zero, it is highly possible that 1,4 and 2,4 are actually similar but the sub graph isomorphism could not find a solution within the given interval.

Solution: Add other parameters to include in similarity measurement.

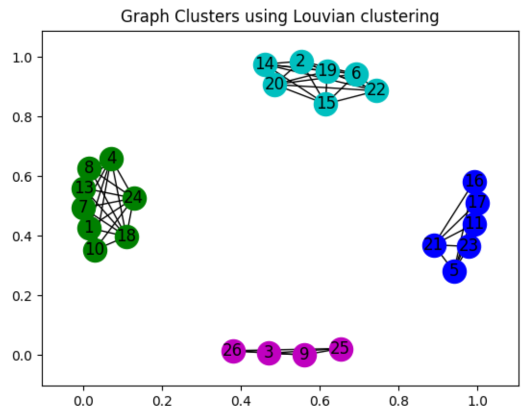

Figure 2: Louvian Clustering

3.3.1.2 IPCA Clustering

Each point's measure property can be transformed as the influence power against its neighbor points.[2] If one point's measure is larger, it would have more influence power to attract its neighbor points, and its neighbors would have a trend to be absorbed by this point.

Observation: Within clusters, the similarity is high but same nodes are found in multiple clusters

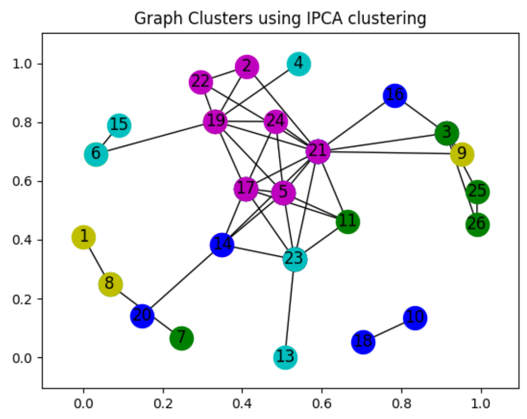

Figure 3: IPCA Clustering

3.4 Marking Scheme:

This module is primarily for Incorrect solutions. The idea is to categorize the solution into the nearest cluster by calculating similarity measure with each of the Gold Standard solutions (Correct Solutions) and clustering into the nearest group.

We get the similarity measure from the above module and find the most similar gold standard program and assign the cluster which the gold standard program is in.

To assign a grade we use many factors like the no. of testcases it passed, the running time, amount of memory used, strength of similarity and intra-cluster distance, etc...

References

[1] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[2] Yudong Chen, Sujay Sanghavi, and Huan Xu. “Improved graph clustering”. In: *IEEE Transactions on Information Theory* 60.10 (2014), pp. 6440–6455.

[3] Luigi P Cordella et al. “A (sub) graph isomorphism algorithm for matching large graphs”. In: *IEEE transactions on pattern analysis and machine intelligence* 26.10 (2004), pp. 1367–1372.

[4] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349.

[5] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[6] Lars Kotthoff, Ciaran Mccreesh, and Christine Solnon. "Portfolios of subgraph isomorphism algorithms". In: *International Conference on Learning and Intelligent Optimization*. Springer. 2016, pp. 107–122.

[7] Chao Liu et al. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: ACM, 2006, pp. 872–881. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150522. URL: http://doi.acm.org/10.1145/1150402.1150522.

[8] Ciaran McCreesh and Patrick Prosser. "A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 295–312.