Theoretical Models and Applications: Final

Jordan Ell V00660306
University of Victoria, Victoria, British Columbia, Canada
jell@uvic.ca

I. Question 1

Note: For this question, I chose the theoretical content that I used in my project. This is where I did most of my learning in the second half of the course as my project did not pertain to the examples listed in this question. I did study procedural generation but not to the extent that I studied my project material.

A. Identifying the Model

In the second half of the course most of my study went into my course project. This being said, the theoretical material that I covered / learned during this time all pertained to graph isomorphism. Specifically I learned about edit distances in trees and how it is a somewhat solved problem in terms of being runnable in polynomial time. I am aware that graph isomorphism was mentioned in the first half of the class, however, as a group it went unstudied and I only began to really research the subject in the second half of the semester for my project material.

B. My Study

My study really started when I attended the Mining Software Repositories 2013 conference earlier in the semester. Here I was listening to a talk that mentioned a project where types of source code changed could be identified using tree structures. I did some quick searches on the web and landed at a project called Change Distiller. (http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.html). This project was the real beginning point of what would later becomes my ideas for my course project for this class. Change Distiller was all about building source code tree structures known as abstract syntax trees (one of the first steps in compilation of code) and then finding some sort of edit distance between them. Change Distiller has three main publications associated with it [1] [2] [3] published between the years of 2006 and 2007, so I knew right away that is was fairly recent research. The first paper dedicated straight to the tool was my first read. Here is where I found all the basic information I needed in order to fully understand my domain. From here I found several other background papers that I later investigated to get a better understanding of how this theoretical model pertains to other domains and how it came to be. In the references I found a survey paper on tree edit distance and related problems [4]. This paper gave me basically all that I needed to see where the problems I was looking at were standing about 6 years ago (a good indication of where they are now). In all, my study consisted of real several papers listed here as well as some quick internet references such a Wikipedia for graph isomorphism in order to get the understanding that I have today. I will outline exactly what I did with the knowledge in the next section.

C. Learning Examples

My two examples of what I learned through my own means in the second half of this course both come from my course project. My course project was given the name API Evolution or APIE. What it did was fully encompass my learning objectives while pushing the boundary slightly in order to add domain knowledge to the world of software engineering. My first example from the project was the ability to extend the source code of Change Distiller (previously mentioned). Change Distiller is an open source project so I was able to obtain the base code for doing abstract syntax tree differencing right away. However, I knew this would not be sufficient for course objectives. This being the case I set out to make Change Distiller a bit better. Change Distiller came with a taxonomy of source code change types that it could detect, however, it was missing the ability to detect changes to in-line source code comments. I thought it would be an interesting challenge to put my new knowledge to work by adding this feature. I was able to successfully detect additions, deletions and modifications to source code comments by the end of my task. This was my first example of my own design (being able to implement tree edit distance algorithms that I had learned).

My second "own design" aspect of the course involved creating new problems to solve in the domain of software engineering based on my newly acquired theoretical knowledge. I did this by creating the goals / research problems of the APIE project. I now knew how to see what types of source code changed happen in Java projects and I also knew from some papers I had read that software evolution is a largely studied area [5], so I decided to pose the question of "What types of changes frequently occur around release dates in a project and how does that give an indication of project stability?". As far as my knowledge goes, this is an unstudied problem in the field of software engineering and is a novel concept. I believe putting this question forward allowed me to preform some actual research in the field and put my theoretical knowledge to the test. In order to answer this question, I used Change Distiller (my improved version) in order to find out what types of changes (from out taxonomy) occur before and after a release and what types of trends exist across a wide variety of projects. The best part about this learning objective was that I found an answer (see my final report / project). I was

able to determine that what makes a project stable during a release is the downward trend in source code changes to public bodies such as classes and methods. I did all this without having an intimate knowledge of the projects or their software development styles, I only used the theoretical model / process knowledge I had obtained during this class.

II. Question 2

Note: I broke this question up into two sections. In each section I answer the given questions for one out of the two papers I selected. Each paper is references by its title in the subsection header.

A. Partial Parsing via Finite-State Cascades

As with any parsing or understanding algorithm of free text, there is always the speed vs. accuracy debate. Where voice recognition systems may prefer speed for getting results to the user, data miners would prefer accuracy of their results over speed. In this paper, this debate takes the foreground in the scientific process outlined by the authors. The authors took the first 3 pages to explain their new algorithm initially before pinning it against previous attempts in experimental trials. I believe the scientific process in this paper can be used as a good outline for any project where a new approach is added to an already large field of attempts / algorithms / ideas. Here the authors not only address both sides of the coin with comparisons to speed and accuracy, but they also address the concerns with comparing and contrasting their new approach with existing ones.

The authors setup an experimental hardware simulation so that all trials are done according to the same means. Next the authors don't worry about their own work but instead test the works of other to establish baselines, maximums, and minimums. Once that is done, then they feel comfortable testing their own algorithm as they have points of reference to compare against. An interesting thing to note, it that they tested algorithms from a wide spectrum of language parsers in order to show which type of speed and accuracy classes their results fall in. Aside from speed, the authors also tested accuracy. Since the accuracy of partial parsing via finite state cascades involves some human interaction, they tested it using two qualified language judges. These judges were responsible for determining the rules of the language to be parsed. Honestly, I thought the limitation of two judge was quite small for this paper, but two is still better than one. I would have liked to have seen around 5 or so judges so that the reader can get a better understanding of the true mean and median of the results offered by the paper. The paper ended up showing that not only was the new algorithm among the fastest of all test means at 1600 words/second, but it was also fairly accurate at 92% and 93% correct.

In terms of limitations, like I said this paper was limited to only two judges of language rules. This provides a somewhat small basis on what can be learned from the accuracy of the parser. The real limitation itself it however based on the limited number of judge, in order to get this new algorithm to work correctly, the implemented must have a robust knowledge of the language being parser. They must be able to know what makes up prop phrase, or what w transitive verb is and how they can used in sentences. This is in contrast to the other mentioned algorithms which look for simple keywords and could be implemented and used by anyone. This is the major concern I have with this paper and unfortunately it is not listed anywhere in the paper as a limitation. To me this is a severely limiting factor as to how much of a deep impact this paper could have on the rest of the domain.

Finally, in terms of merit, this paper does a good job showing the implications of the proposed algorithm. This is done through their experimental trials discussed earlier. The authors clearly give an understanding of the domain and where other ideas stand in comparison to their own. If the above limitations were not such a concern for me, I would say this paper would be considered high impact.

On a side note, this paper showed exactly what I thought was missing from a lot of student presentations late in the semester. There was a serious lack of comparison to existing ideas in the field when new ideas were being discussed. I think we as scientists should make a better effort in comparing ourselves to existing systems. This is where the true knowledge lies and not necessarily in the new idea itself.

B. When Do Changes Induce Fixes?

The main focus of this paper was a novel concept that would (eventually) open up software engineering to the study of what causes a bug or a system error and how can we prevent it. The basic idea of the paper translates back to the idea of a disease and a cure (which popped up all over the place in this course). There are two schools of though, one if the present situation where a disease is found and a cure is determined, the second being a retrospective approach where we know a cure was used and we work backwards to find the disease. This paper took advantage of the second approach. It is interesting to see how this methodology could be applied to the firefighting problem, disease control, or ever graph diffusion. These three algorithm typically involve having the disease then applying a cure, but we could use the opposite approach (the one used in this paper) to determine retrospectively if those decisions made (the cures) were optimal or correct.

Since the actual algorithm is not being brought too much into question here, I will skip to their experimental setup and their critical thinking skills in this paper. In order to test the algorithm, the authors used two large open source projects (Eclipse and Mozilla) in order to find out what changes were fix inducing. Specifically, they broke those changed down by transaction size and days of the week in order to show the power of the algorithm and what types of results it could find. What they found was that bugs are more likely to occur in large transactions and on Fridays. These were some interesting initial results of the algorithm. Before I get into a bit more of a discussion about the paper, I want to note that this idea was truly novel (see the related work) so comparing against existing ideas was near impossible which is in direct contrast

to the previous paper discussed. I think this is fine as long as a long discussion of the strengths and weaknesses of the algorithm is present, however, in this paper this is not the case.

The main issue I have with this paper is the limitations / lack of improvements / discussion of merit. First off, the results of the experiment were not validated in anyways. To me, the only way to validate a truly new automated process is to provide manual inspection of the data in order to validate results or at least a percentage of results. The authors should have manually inspected some of the transactions in order to determine if their finding were accurate. This was a huge disappointment in the paper. We should always be comparing our results to something else in order to validate! Always! Next, there is no discussion in this paper about limitations or suggested improvements. Just by reading the paper I can come up with some serious flaws in their algorithm. For example, if transaction A edits lines 1-10 in file A, transaction B adds new line 1-15 in file A pushing 1-10 down to 15-25 then transaction C fixes a bug in lines 20-23 this will be marked as transaction B having the bug unless the version control system accounts for the moving code (which I know CVS does not do). The authors should have provided some indication as to where false positives or false negatives would lie in their algorithm as there are more than I care to write. Lastly is the discussion of merit. While this idea is novel, extremely high impact, and opened the door to hundreds of paper, the lack of limitations puts a serious stain on the merit of this paper. There are no accuracy measurements ever given. This causes papers that are based of this to not be able to pinpoint incorrect data or to provide context to possible wrong results because of any errors here. This paper has setup a lot of future work, but I am sure (and from my own work) that is has also caused many issues with researchers in software engineering because of the loose guidelines provided in terms of accuracy and limitations.

III. Question 3

Note: I broke this question up into two sections. In each section I answer the given questions for one out of the two papers I selected. Each paper is references by its title in the subsection header.

A. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction

For this paper, the word optimum can be thought of as the word minimal. The authors of the paper were attempting to find an optimum (minimal) edit distance between two abstract syntax trees of source code. Here the optimum edit distance would be the exact changes preformed by the developer in order to transform one tree to the next (this is referred to as the minimal edit distance in the paper). An example of this would be that in a commit a developers edits the signature of a method (say he/she changes the name of the method) and at the same time changes 50% of the internal method body. Now, an optimum edit distance would be exactly as just described, identifying the change to the method name and the changes to the internal method body. A non-optimum edit distance would be, for example, that the user deleted an old method and inserted a new method. While it could be argued that in effect these two edits have the same end result, only one of them is optimum in the sense that is mimics exactly what the user did is process and not just result.

The results of this optimum searching for this paper were quite good in respect to other graph isomorphism problems that are present today. The authors found that they did not find the optimal (minimum) edit distances in only 34% of all cases studied. This is a significant result because 66% of the time the algorithm is as accurate as possible and 34% of the time the algorithm is only slightly sub optimal. If your research question does not care about slight mistakes in achieving an optimal solution, then this paper is extremely high impact.

In my opinion, the optimization of the tree edit distance is absolutely critical to what types of research questions are being pursued because of the results of this paper. To demonstrate a few [6] [7] [8], most papers referencing this original paper deal with locating some type of pattern or software evolution change analysis based on exact changes that are being made to the software. This being the case, it is crucial that the accuracy of this paper, the optimum result, be as high as possible to show exactly what developers are doing to facilitate these other research objectives. Since I use this algorithm in my own research it is also crucial for me that the optimum be achieved as must as possible so that I can validate my own results in a positive manor. What is being optimized (the actual developer behavior) is the most important aspect of this model and is what gives the result and spin off research so much robustness and a good future moving forward.

B. Modularity and Community Structure in Networks

For this paper, the word optimum is applied to the level of modularity inside of a graph. The author of this paper was developing an algorithm based of graph node modularity in order to create a community detection algorithm which could take a “unordered” graph and create some type of order from the chaos and dissect the graph into modular components. The optimum trying to be achieved here is the modularity in the graph. This being said, the optimum solution was to have tightly bounded communities that were detected in the graph leaving distinct and disjoint groups of nodes (based on a modularity similarity measure) apart from one another. In the case of this paper, an optimum solution would be to have every node inside of a community and to have to communities themselves be distinct and disjoint. A sub-optimal solution would be where nodes are left over after the algorithm, outside of any particular community, or to have communities which are only loosely joined or to have overlapping communities.

In my opinion, for the purposes of this paper and the author’s specific research goal / question the optimization being preformed here is valid. However, I believe that this type of optimization actually closes the door on some of the more interesting research questions that could be proposed. For example, it would be interesting to see, say in an open

source project communication structure, which nodes (people) fall outside of any one particular community but instead act as brokers between two or more communities. Again, sticking with the open source project communication structure, it would be interesting to see if communities overlap and which members (nodes) of these communities are responsible for the overlap and how that person(s) role in the project affects this outcome. I think if the author of this paper, instead of focusing on the optimal solution, focused on a more general and perhaps sub-optimal result, the resulting algorithm, and resulting future research questions, would have been a lot more interesting. This all being said, there are still lots of research questions that can be answered through this optimal approach such as using this algorithm for image segmentation as we saw in GrabCut [9] and in Garreth's presentation / course project. My other suggestion for this type of optimal solution is where community overlap and nodes left outside communities are unacceptable. An example of could be the extraction of components of a software project. An example of this is when you want to reuse a piece of code but don't want to copy over unnecessary files. Here you want a very high modularity in the code being copied and nothing else. An example of this being done with video games can be found here [10]. It would be interesting to see graph community detection used for the same purpose.

REFERENCES

[1] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70731

[2] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, ser. WCRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 70–79. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2007.21

[3] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 35–45. [Online]. Available: http://dx.doi.org/10.1109/ICPC.2006.16

[4] P. Bille, "A survey on tree edit distance and related problems," *Theoretical Computer Science*, vol. 337, no. 13, pp. 217 – 239, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397505000174

[5] A. E. Hassan and R. C. Holt, "Studying the evolution of software systems using evolutionary code extractors," *Principles of Software Evolution, International Workshop on*, vol. 0, pp. 76–81, 2004.

[6] I. I. Brudaru and A. Zeller, "What is the long-term impact of changes?" in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 30–32. [Online]. Available: http://doi.acm.org/10.1145/1454247.1454257

[7] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 35–38. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370759

[8] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 463–466. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.74

[9] C. Rother, V. Kolmogorov, and A. Blake, ""grabcut": interactive foreground extraction using iterated graph cuts," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 309–314, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015720

[10] A. Ampatzoglou, I. Stamelos, A. Gkortzis, and I. Deligiannis, "A methodology on extracting reusable software candidate components from open source games," in *Proceeding of the 16th International Academic MindTrek Conference*, ser. MindTrek '12. New York, NY, USA: ACM, 2012, pp. 93–100. [Online]. Available: http://doi.acm.org/10.1145/2393132.2393152