

Supporting Software Evolution Using Adaptive Change Propagation Heuristics

Haroon Malik and Ahmed E. Hassan

Software Analysis and Intelligence Lab (SAIL)

School of Computing, Queen's University, Kingston, Ontario, Canada.

{malik, ahmed}@cs.queensu.ca

Abstract

When changing a source code entity (e.g., a function), developers must ensure that the change is propagated to related entities to avoid the introduction of bugs. Accurate change propagation is essential for the successful evolution of complex software systems. Techniques and tools are needed to support developers in propagating changes. Several heuristics have been proposed in the past for change propagation. Research shows that heuristics based on the change history of a project outperform heuristics based on the dependency graph. However, these heuristics being static are not the answer to the dynamic nature of software projects. These heuristics need to adapt to the dynamic nature of software projects and must adjust themselves for the peculiarities of each changed entity.

In this paper we propose adaptive change propagation heuristics. These heuristics are meta-heuristics that combine various previously researched heuristics to improve the overall performance (precision and recall) of change propagation heuristics. Through an empirical case study, using four large open source systems; GCC (a compiler), FreeBSD (an operating system), PostgreSQL (a database), and GCluster (a clustering framework), we demonstrate that our adaptive change propagation heuristics show a 57% statistically significant improvement over the top-performing static change propagation heuristics.

1. Introduction

Changing a source code entity, e.g., a function, most likely requires propagating the change to other related entities [6]. If the change is not propagated correctly, the project risks the introduction of new bugs.

Change propagation is the process of ensuring that a change to an entity is propagated to all related entities. Researchers have proposed many change propagation heuristics along different dimensions. Change history, entity dependencies and containment information are three of the most explored dimensions. Change history based heuristics predict that changes should be propagated to all entities which frequently co-changed

with the changed entity in the past. For example, if an entity A was changed then the change should be propagated to all entities that co-changed with A over 80% of the time in the past. Dependency based heuristics predict that changes should be propagated to all entities that have some dependency with the changed entity. Call-graph, program-dependency graphs and UML models [1][11][12] are examples of commonly used sources of dependency information. Containment based heuristics predict that changes should be propagated to all entities that are contained within the same container (e.g., file or class) as the changed entity [7][12]. All previously researched propagation heuristics are static [6][15]. They do not adjust over time nor do they adapt to the particular changed entity. For example, during maintenance periods in a project, a change history based heuristic is likely to outperform well. The same heuristic would perform poorly during periods of new development [7]. Also often some entities may perform well with one type of heuristic and perform poorly with another type of heuristic. Therefore, we believe that adaptive co-change heuristics are needed. These heuristics should adjust over time and should adapt to the peculiarities of each entity in order to improve the overall performance of change propagation heuristics.

In this paper we propose two adaptive propagation heuristics. Through a large empirical study, we demonstrate the benefit of adopting our proposed heuristics. Using the development history of four large open source projects, we compare our adaptive heuristics to the top performing static heuristics that have been previously proposed in literature. Our study shows that adaptive propagation heuristics outperform other top-performing static propagation heuristics by as much as 57% based on our statistical analysis.

Paper Organization

The organization of the paper is as follow. Section 2 presents the change propagation process. Section 3 casts light on how to measure the performance of change propagation heuristics. Section 4 presents various change propagation heuristics. Section 5 presents our adaptive propagation heuristics and

highlights the intuition behind our work. Section 6 presents our case study and details the results of the various experiments in our study. Section 7 presents related work. Finally, section 8 concludes the paper.

2. Change Propagation

Change propagation is the process of propagating code changes to other entities in a software system to ensure the consistency of assumptions in the system after changing an entity [6]. For example, a change to an interface may require the propagation of the change to all the components which use that interface. The propagation ensures that both the interface and entities using it have a consistent set of assumptions.

Change propagation plays a central role in software development. However, current practices for propagating software changes rely heavily on human communication, and the knowledge and experience of senior developers or gurus. Recent work by Hassan and Holt proposes automating the change propagation process using various propagation heuristics [6].

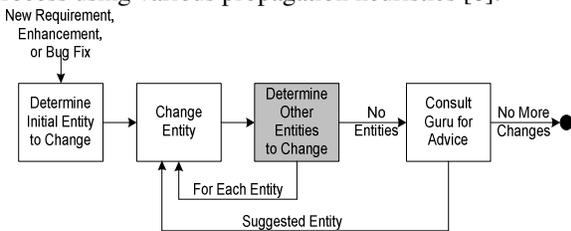


Figure 1: Change propagation process

Figure 1 gives an overview of the change propagation process. Once a developer receives a request to add a new feature or to fix a bug, they determine the initial function to change. Once the function is changed, the developer needs to determine other entities to which the change must be propagated to ensure that this change does not violate the consistency among these related entities. For each changed entity the propagation process is repeated. When a developer’s search for entities to change is exhausted, they consult a Guru such as a senior developer or a test suite. If the Guru points out a missed entity, then the entity is changed and the change propagation process is repeated for the just-changed entity. This process continues until all appropriate entities have been changed. At the end of the process, all entities related to the new-feature or the bug-fix at hand is changed.

3. Measuring the Performance of a Change Propagation Heuristic

In large projects, Gurus rarely exist. Senior developers are usually too busy to check every change

and test suites are seldom complete. Therefore, developers are in dear need for change propagation heuristics that can guide them in identifying entities to which to propagate a change. These heuristics should be accurate (i.e., high precision) and complete (i.e., high recall). Otherwise heuristics with low precision will waste developers’ time and developers will stop using them. If these heuristics have low recall, then developers will miss changing a related entity. Missing to change an entity would result in a bug.

We use an example to explain how we can measure the precision and recall of change propagation heuristic. Dave needs to add a new feature into a large legacy system. He starts off by changing an initial entity A. After entity A is changed, a change heuristic suggests that entities B and X should change as well as. Dave changes B, but then examines X and realizes that it does not need to be changed. So Dave does not need to perform any change propagation for X. He then asks the heuristic to suggest another entity that should change if B were changed. The heuristic suggests Y and W, neither of which need to change—therefore Dave will not perform any change propagation for Y or W. Dave now consults Jenny, the head architect of the project (the Guru). Jenny suggests that Dave should change C as well. Dave changes C and asks the heuristic for a suggestion for an entity to change given that C was changed. The heuristic proposes D. Dave changes D and asks the tool for new suggestions. The heuristic does not return any entities. Dave asks Jenny who suggests no entities as well. Dave is done propagating the change throughout the software system.

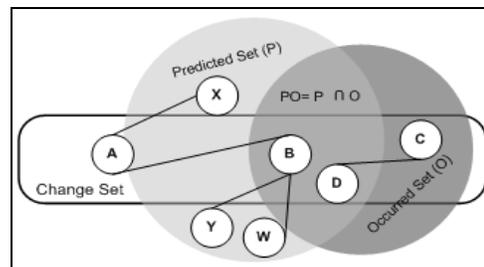


Figure 2: Change propagation flow for a simple example.

All the changed entities in our example represent the *Change Set* for this feature. Figure 2 shows the entities in the example and their interrelationships. Edges are drawn from A to B and from A to X because the heuristic suggested that if A is changed then B and X should change as well. For similar reasons, edges are drawn from B to Y and W, and from C to D. We make the simplifying assumption that a heuristic provides *symmetric suggestions*, meaning that if it suggests entity F when given entity E, it will suggest entity E

when given entity F. We have illustrated this symmetry in Figure 2 by drawing the edges as undirected edges. The total set of suggested entities will be called the Predicted set; Predicted = {B, X, Y, W, D}. The set of entities that should have been predicted will be called the Occurred set; Occurred = {B, C, D}. Note that this does not include the initially selected entity (A), which was selected by the developer (Dave) and thus does not need to be predicted. In other words, *Occurred* = *Change Set* - {*Initial Entity*}.

We define the number of elements in Predicted as P (P = 5), and the number of elements in *Occurred* as O (O = 3). We define the number of elements in the intersection of *Predicted* and *Occurred* (this intersection is {B, D}) as PO (PO = 2). Based on these definitions, we define:

$$\text{Recall} = \frac{PO}{O} \quad (1)$$

$$\text{Precision} = \frac{PO}{P} \quad (2)$$

In our example, Recall= 2/3 =60% and Precision= 2/5=40. The rest of this paper will use these definitions of Recall and Precision. We will make another simplifying assumption, which is that each prediction by a heuristic is based on a single entity known to have changed instead of all the entities that changed up till now. For example, a heuristic may base a prediction on a single element C known to be in the change set, and not on a set of entities such as {A,C} known to be in the change set. A further assumption is that the developer (Dave) will query a heuristic for suggestions before asking the Guru (Jenny). An implication of these two simplifying assumptions is that an approach may not do as well in making predictions as it would without these assumptions. This limitation is not a concern as we are mainly interested in comparing the relative difference between several approaches using the same definitions for precision and recall.

Our simplifying assumptions ensure that *the ordering of queries to a heuristic is immaterial*. For example, Dave might initially select entity B or C or D instead of A. Further, if Dave had a choice of queries to the heuristic (say, to get suggestions based on either entity M or N), either query could be given first. Regardless of the selections, the values determined for *Precision* and *Recall* would not be affected. The change records in a source control system do not record the ordering of selections. Our assumptions simplify our analysis and avoid the need for information that is not available.

We also use the *F-measure* to measure the overall performance of a heuristic. The F-measure combines the precision and recall metrics.

$$F_{\alpha} = \frac{(\alpha + 1) \cdot \text{Precision} \cdot \text{Recall}}{(\alpha \cdot \text{Precision} + \text{Recall})} \quad (3)$$

The value of α ranges between 0 and infinity to give varying weights for recall and precision. For example to indicate that recall is half as important as precision, α would have a value of 0.5. A value of 1.0 indicates equal weight for recall and precision. A value of 2.0 indicates that recall is twice as important as precision. F_0 is the same as precision, F_1 is recall. Due to the dangers of failing to propagate a change to a source code entity, it may be desirable to assign α a value of 2.0 to indicate the higher importance of recall. Alternatively senior developers may prefer not to waste a great deal of time investigating irrelevant entities; therefore they would prefer an approach with high precision. They would use an α value of 0.5. The maximum value for the *F-measure* is 1.0 when both recall and precision are 1.0. The lowest possible value for *F-measure* is 0 when either recall or precision is zero. For the results presented in this paper, we show the precision and recall values as they are more intuitive to reason about their meaning. A high recall would prevent the occurrence of bugs due to missed propagations. A high precision would save developers' time since they do not need to examine irrelevant suggestions.

Measuring performance for multiple change sets:

We have defined precision and recall for a single change set. To measure the performance of a heuristic over time containing (M) change sets using the following definitions:

$$\text{Average } F_{\alpha} = \frac{1}{M} \cdot \sum_{i=1}^M ((F_{\alpha})_i) \quad (4)$$

$$\text{Average Recall} = \frac{1}{M} \cdot \sum_{i=1}^M (\text{Recall}_i) \quad (5)$$

$$\text{Average Precision} = \frac{1}{M} \cdot \sum_{i=1}^M (\text{Precision}_i) \quad (6)$$

4. Heuristics for change propagation

There exists many heuristics to help propagate changes in a large project. We briefly present some of these heuristics and detail the heuristics we used in our case study which is presented in Section 6.

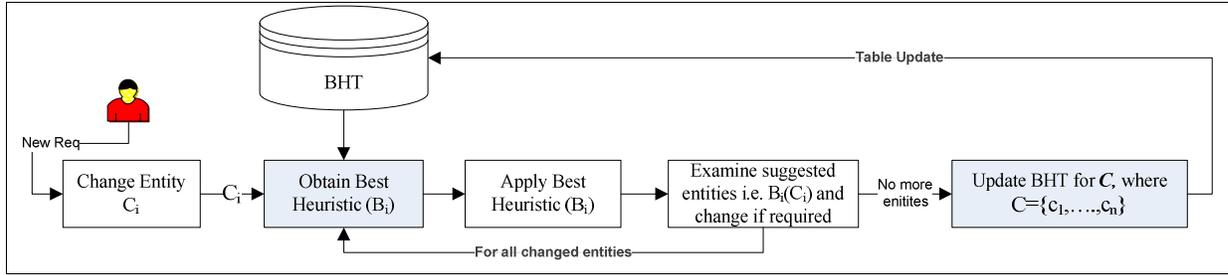


Figure 3: Overview of our Adaptive Co-change Heuristics

History Heuristic (HIST)

Given a changed entity A, a HIST heuristic would suggest all entities that changed often with A in the past. Previous work by Hassan and Holt [6], Zimmermann *et al.* [7] and Sayyad *et al.* [4] have explored the use of this type of heuristic for change propagation. Hassan and Holt have shown that this type of heuristic outperforms other heuristics for change propagation. To avoid returning incorrect entities, a history-based heuristics would prune the list of co-change entities by not returning entities that co-changes less than some threshold. For our case study given an entity A, the HIST heuristics returns the top 7 entities which co-changed the most with A.

Containment Heuristic (FILE)

Research shows that in a software system, entities located in the same file are interrelated [10]. In our case study, we consider a “File” container to be a good predictor of a relation between entities. Given a modified entity A, a file heuristic would return all entities in the same file as A in our case study.

Call Use Depends Heuristic (CUD)

A CUD heuristic [21] exploits the static coupling due to data and control flow between code entities. Given a modified entity A, a CUD heuristic returns all entities that depend on A or that A depends on.

Code ownership Heuristic

Code ownership exploits the code ownership information. Given a modified entity A, a code ownership heuristic would return all entities that are owned by the same developer who changed A. We do not explore the use of this heuristic in our case study. Prior work by Hassan and Holt has shown that this heuristics does not perform well on open source projects since in many open source projects the code ownership is not recorded when performing a code change [6]. This is due to the fact that a limited number of developers are given commit privileges to the code repository. These committers act as proxies to the developers who performed the change.

5. Adaptive Change Propagation Heuristics

Most heuristics presented in literature are based on a single dimension (e.g., historical co-changes or containment) or a combination of several dimensions. However, these heuristics do not adapt to take into account the dynamic nature of the software development process and the variety of entity types in a software system. For example, during maintenance periods a history based heuristic may perform well. The same heuristic would perform badly for predicting co-changes during periods of new development. Using a dependency based heuristic to suggest entities to co-change when a function within a library is changed may not perform as well as a historical co-change heuristic.

We propose the use of adaptive co-change meta-heuristics. Such family of heuristics uses a Best Heuristic Table (BHT) to track for each change entity C_i the best performing heuristic B_i out of several heuristics as shown in Figure 3. Once an entity is changed the best heuristic for that particular entity is used to suggest other entities to change. At the end of the change, the heuristic updates the BHT for each changed entity. Figure 3 gives an overview of our adaptive change propagation meta-heuristics. There are various techniques for updating the entry for an entity in the BHT. In this paper we explore two of such techniques:

1. **Recency Update:** The recency technique updates the BHT by inserting the heuristic that performed best for that entity the last time that particular entity was changed. (**A-Rec**)
2. **Frequency Update:** The frequency technique updates the BHT by inserting the heuristic that performed best for that entity overall for all prior changes of that entity. (**A-Freq**)

By continuously updating the BHT table, we ensure that we are always using the most optimal heuristic for an entity.

Table 1: Performance of the change propagation heuristics for the studied projects.

Project	HIST		CUD		FILE		A-Freq		A-Rec	
	Recall	Precision								
PostgreSQL	0.69	0.14	0.44	0.02	0.73	0.13	0.45	0.25	0.4	0.30
FreeBSD	0.70	0.12	0.40	0.02	0.76	0.11	0.41	0.27	0.41	0.30
GCluster	0.52	0.18	0.38	0.09	0.70	0.14	0.39	0.22	0.35	0.28
GCC	0.78	0.10	0.43	0.02	0.80	0.12	0.51	0.21	0.47	0.25
All	0.67	0.13	0.41	0.04	0.74	0.12	0.44	0.23	0.40	0.28

As described both heuristics require an entity to have a single change before they can start using the different update techniques. For the first change of an entity, we use the file heuristic since it does not require any change history. We conducted a large empirical study to better understand the benefits of adaptive meta-heuristics.

6. Empirical Study

Using our Precision, Recall and F definitions, we measure the performance of our adaptive co-change meta-heuristics. Instead of conducting a user study where developers are asked to perform a limited number of changes using various co-change heuristics, we conducted our study using historical co-change information recovered from the source control repository of several open source projects. The use of historical co-change information permits us to conduct a much larger study where we can explore a large number of changes from several projects. Information about the studied open source project is shown in Table 2.

Table 2: Characteristic of the studied systems

Project	Start	End	Func.	Files
PostgreSQL	July 1996	Feb 2008	31,000	1,493
FreeBSD	June 1993	Aug 2005	27,935	1,629
GCluster	June 2004	Feb 2008	15,539	935
GCC	Aug 1997	Oct 2005	22,460	384

PostgreSQL is one of the leading open source relational database systems. FreeBSD is a successful open source operating project. GCluster is a framework to enable the clustering of commodity hardware to create super-computers. GCC is the leading open source compiler project. The table lists the number of unique functions and files through the lifetime of each project. The average age of a project in our case study is 9 years with a total of 36 years for all projects.

We used the Development Replay framework, presented in [21], to perform our study. The DR framework takes as input the source control repository of a software project and permits the exploration of various co-change heuristics. The DR framework uses the C-REX evolutionary extractor to build a time based dependency graph from the text-based source repository [21]. Whereas a source control repository tracks changes to the source code as changes to text, C-REX converts this information to changes to the dependency graph of the software system. For example, the source control system may indicate that line 4 was removed from a file; the C-REX output would indicate that function A() no longer calls function B. This information is essential to calculate the CUD heuristic over time. Moreover, the C-REX output maps line changes to the particular functions and code entities e.g., structs, that changed. This information is used to determine the studied change sets.

Studied Change Sets

We did not study all the change sets obtained through C-REX. We did not use change sets relating to General Maintenance (GM) modifications. C-REX uses a lexical technique, similar to Mockus and Votta [20], to analyze the change message attached to a change to determine if a change is a GM modification. GM modifications do not reflect the implementation of a particular feature. For example,

1. Modifications to update the copyright notice at the top of each source file are ignored.
2. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty printer are ignored.

We do not analyze GM modifications, since we do not expect any heuristic to effectively predict the propagation of changes of these modifications. Table 3 shows a breakdown of the change sets for the studied projects. We classify the remaining change sets as:

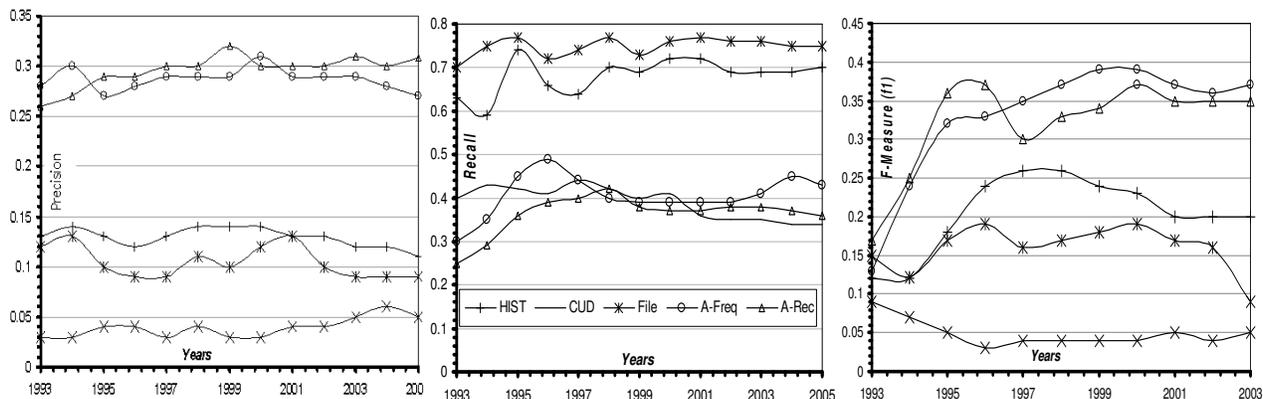


Figure 4: Performance of the studied heuristics over time for the FreeBSD project

1. Sets where entities are added, such as the addition of a function (New Ent. column in Table 3).
2. Sets where no new entities are added. In these sets, entities such as functions are either removed or modified (Studied column in Table 3).

Table 3: Change Sets from Studied Projects

Project	All	GM	New Ent.	Studied
PostgreSQL	12,720	1,562(12%)	1,571(12%)	9,705 (76%)
FreeBSD	27,934	4,065(14%)	3,761(13%)	20,108(73%)
GCluster	2,884	984(34%)	284(10%)	1,890(64%)
GCC	20,415	3,545(17%)	2,290(11%)	13,125(64%)

We chose to measure the performance of propagation heuristics using change sets where no entities were added. This choice enables us to compare different change propagation heuristics fairly, as it is not feasible for any heuristic to predict propagation to or from newly created entities. We note that we still use the sets where entities are added to build the historical co-change information but we do not measure the performance of any heuristic using these sets.

Comparing the Performance of Adaptive Heuristics

Using the change sets from the four projects, we measure the performance of traditional heuristics (*described in* Section 4) and compare their performance with respect to our two adaptive heuristics (*described in* Section 5). Table 1 shows the performance of the heuristics in terms of Precision and Recall. The ALL row shows the weighted average over all projects. The average is weighted using the number of studied change sets for each project as shown in Table 3. Looking at Table 2, we notice that the traditional heuristics have better recall or similar recall than the adaptive heuristics. On the other hand the precision of our adaptive heuristics is much higher than the other heuristics. Looking at Table 4, which shows the performance in terms of F_1 (precision as important

as recall), we note that proposed adaptive heuristics outperform the traditional heuristics. Looking at the ALL row, the F-measure for A-Freq is 0.3. This is 23% better than the performance of HIST, the best performing non-adaptive heuristic. The A-Freq heuristic is more complex to implement since it requires bookkeeping to track the best heuristics for all changes during the lifetime of an entity. However, the A-Rec heuristic only tracks the last change. A closer look at Table 4 shows that both adaptive heuristics show similar overall performance from a practical point of view. There are certain trades-off associated with the use of each adaptive heuristic. The A-Rec is a much easier heuristic to implement. On other hand, the A-Freq has high recall in contrast to A-Rec as evident from Table 1 and Figure 4. A-Rec is of greater interest to developers who do not want to miss propagating a change to any possible entity (since it has high recall).

Table 4: Performance of Heuristics (F-Measure)

Project	HIST	CUD	FILE	A-Freq	A-Rec
PostgreSQL	0.23	0.04	0.22	0.32	0.34
FreeBSD	0.20	0.04	0.19	0.32	0.34
GCluster	0.26	0.15	0.23	0.28	0.31
GCC	0.18	0.04	0.21	0.29	0.32
ALL	0.23	0.06	0.21	0.30	0.33

Performance characteristics of Adaptive Heuristics

To better understand the performance characteristics of the proposed adaptive heuristics, we examine their performance along three aspects:

1. **Performance over time:** The performance of the heuristics over time instead of just using a single aggregate value (e.g., F-measure for the lifetime of a project).
2. **Table composition over time:** The content of the Best Heuristics Table (BHT) to determine how it changes over time as a project evolves.

3. **BHT suggestion vs. optimal suggestions:** The percentage of suggestions by our adaptive heuristics which uses the BHT to give suggestions, versus the optimal suggestions of heuristics to use.

We discuss below these three aspects.

Performance over time

Figures 4 show the performance of all the heuristics over time for the FreeBSD project using the Precision, Recall and F-measures. The Figures for the other project show similar trends and are omitted due to space constraints. In particular, we note that:

1. For precision, the A-Freq and A-Rec heuristics outperform all traditional heuristics. Other than a short period at start of project, the A-Rec heuristic always outperforms the A-Freq heuristic. The precision of the A-Freq heuristic degrades slowly over time indicating that the heuristic is likely suffering from a cache pollution problem [21], where a large number of old irrelevant changes are providing incorrect suggestions for new changes.
2. For Recall, both heuristics do not perform as well as the other heuristics as observed in the Table 1. The A-Freq and A-Rec heuristics show similar trends as precision with A-Rec performing well for a brief period of time at start of the project. Overall A-Rec has lower recall as compared to A-Freq for all projects.
3. Using the F-measure, we note that both adaptive heuristics outperform all heuristics since the start of the project and continue to outperform the other heuristics throughout the lifetime of a project. At the start of the project, the A-Rec heuristic outperforms the A-Freq heuristic while the frequency counts used by the A-Freq heuristic have enough changes to start providing accurate suggestions. This trend is consistent across all studied projects.

BHT composition over time

Examining the composition of the BHT we can better understand the evolution the change propagation process in the studied projects. In Figures 5 and 6 we show the composition of the BHT for FreeBSD for the A-Freq and A-Rec heuristics respectively. We note that in Figure 6, the A-Rec graphs show sharp spikes reflecting the aggressive adaptability nature of this heuristics. Whereas for A-Freq in Figure 5, we notice that the curves are smoother. Both heuristics show a similar composition in the table with the HIST heuristic being the most used heuristic followed by the FILE then CUD heuristic. At the start of the project the HIST is not widely used, however it quickly rises as the performance of the HIST improves with more historical

information available as a project evolves. Again these observations are consistent for all projects.

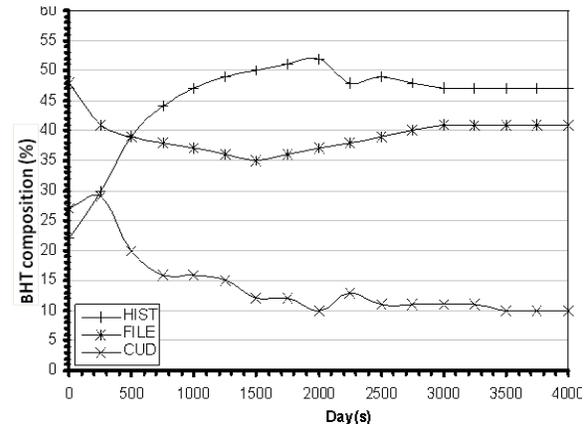


Figure 5: BHT composition using A-Freq

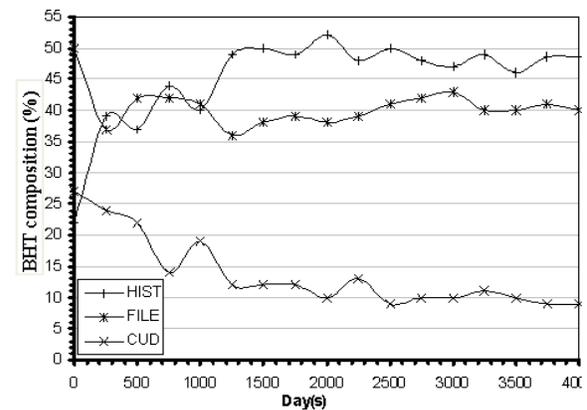


Figure 6: BHT composition using A-Rec

BHT suggestion vs. optimal suggestions

Our proposed adaptive heuristics are meta-heuristics in the sense that they are used to manage a set of heuristics. Given a set of heuristics there exists an optimal use of these heuristics. The optimal (ideal) adaptive heuristic would always propose the best heuristic to use from the set of available heuristics for each changed entity. Since our adaptive heuristics use different techniques to build the BHT, we are interested in measuring the difference between the most optimal (i.e., best theoretical) suggestions of heuristics and the heuristic suggestions proposed by our adaptive heuristics. Table 5 shows the percentage of optimal suggestions by both of our adaptive heuristics. For example, given the three heuristics (HIS, FILE, and CUD), if the A-Rec heuristic were to suggest FILE for a changed entity but the most optimal suggestion is HIST, then the A-Rec heuristic gave a non-optimal suggestion. The Table shows that the adaptive heuristics are within 76-85% of the theoretical optimal suggestions.

Table 5: Percentage of optimal suggestions

Project	Changed entities	% of Optimal Suggestions	
		A-Freq	A-Rec
PostgreSQL	77,650	64,450(83%)	61,344 (79%)
FreeBSD	160,864	130,300(81%)	123,866(77%)
GCluster	15,120	12,852(85%)	11,450(76%)
GCC	105,000	87,150(83%)	82,950(79%)

Table 6: Performance with respect to optimal heuristic

Project	% of Optimal Performance					100%
	HIST	CUD	FILE	A-Freq	A-Rec	Optimal (F)
PostgreSQL	49	9	47	68	72	0.47
FreeBSD	44	9	42	71	76	0.45
GCluster	48	28	43	52	57	0.54
GCC	38	8	44	60	67	0.48
ALL	48	13	44	63	63	0.48

Table 6 shows the F-measure for the optimal adaptive heuristic. The Table also shows the F-measure performance of the various heuristics relative to the optimal heuristics. The actual F-measure values are shown in Table 4. Looking at Table 6, we see that for FreeBSD the Optimal F-measure is 0.45. If were to assign 100% to the optimal performance then HIST, one of the best non-adaptive heuristic, achieved a 44% performance of the optimal heuristic and both adaptive heuristics are within 63% of the optimal heuristic. In future work we plan to explore other more advanced BHT update techniques to further improve the performance of the adaptive heuristics.

Integrating advanced change propagation heuristics

Based on the analysis above we note that the HIST heuristics is the best performing traditional heuristic. Therefore we decide to explore enhancing the history based heuristic and to investigate the effect of the enhanced heuristics on the performance of our adaptive heuristics. We hope to measure the performance improvement of our adaptive heuristics when using more advanced traditional static heuristics. Prior work, by Ahmed [21], shows that these history based heuristics perform well when propagating changes for other large projects:

1. **REC(N)(M):** given a changed entity E, the REC(N)(M) heuristics would suggest all entities that have changed with E in the past M months.
2. **FREQ(A):** given a changed entity E, the FREQ(A) heuristics would suggest all entities that have changed with E at least twice in the past and changed more that A% of the time with E.

We first explore the different values for M and A to determine the best REC(N) and FREQ heuristics using the data for all four projects. Table 7 shows the weighted F-measure across all projects when varying

the value of M and A. The entries marked in bold REC(N)(4) and FREQ(60) are the two best performing heuristics. Therefore we chose to integrate these two heuristics into the pool of heuristics used by our adaptive meta-heuristics.

Table 7: Weighted average F-measure for REC(N)(M) and FREQ(A) for all studied projects

REC(N)(M)	F-Measure	FREQ(A)	F-Measure
REC(N)(2)	0.39	FREQ(50)	0.39
REC(N)(4)	0.40	FREQ(60)	0.44
REC(N)(6)	0.34	FREQ(70)	0.42
REC(N)(8)	0.28	FREQ(80)	0.39

Table 8: Precision and Recall of Adaptive Heuristics

Project	A-Freq		A-Rec	
	Recall	Precision	Recall	Precision
PostgreSQL	0.74	0.67	0.79	0.66
FreeBSD	0.74	0.61	0.70	0.64
GCluster	0.72	0.63	0.77	0.63
GCC	0.73	0.66	0.87	0.59
ALL	0.73	0.64	0.78	0.64

Table 9: Performance of A. heuristics against optimal heuristic

Project	F-measure		Optimal Heuristic		
	A-Freq	A-Rec	Rec	Prec	F ₁
PostgreSQL	0.70 (91%)	0.72 (93%)	0.91	0.62	0.78
FreeBSD	0.66 (78%)	0.67 (80%)	0.90	0.71	0.85
GCluster	0.67 (90%)	0.69 (92%)	0.84	0.67	0.75
GCC	0.69 (90%)	0.70 (92%)	0.83	0.62	0.77
ALL	0.68(87%)	0.69(89%)	0.85	0.62	0.78

Table 8 shows the performance of both our adaptive heuristics (A-Rec and A-Freq) over all projects using the advanced heuristics. The performance of our adaptive heuristics is very high: 73-78% for Recall and 64% for Precision.

Table 9 compares the performance of our adaptive heuristics against the optimal heuristic. Values in brackets indicate the difference between the optimal heuristics and another heuristic. For example, for the GCC project, the performance of the A-Rec heuristic is within 92% of the optimal heuristic and the performance of A-Freq heuristic is within 90% of the optimal heuristic where the performance of the optimal heuristics is scaled to be 100%.

Summary of Limitations

Our analysis uses open source projects from different domains: compilers, operating systems, databases, and cluster computing. Although these projects come from different domains, our results may not hold for all types of software systems.

Instead of performing user studies, we decided to use the historical change information stored in the source control repositories of these projects. However, if our adaptive heuristics are adopted in practice the results may vary since developer will need time to get acquainted to the tools that implement these heuristics. Moreover, developers work practices will likely vary based on their interaction with the tool.

7. Related Work

Change propagation is an essential and important activity in software development practice. Zimmerman *et al.* [7] explored the use of historical project information, such as CVS logs, for detecting change propagation between source-code entities (i.e., files, classes, methods, and variables). Similar work has been done by Shirabad *et al.*[4][5] and by Ying *et al.*[13]. All aforementioned approaches did not explore the use of other non-historical information such as static dependencies. Briand *et al.* [16] study the likelihood of two classes being part of the same change due to an aggregated value that is derived from object oriented coupling measures. Hassan and Holt [6] explore the use of the traditional heuristics presented in this paper to study change propagation. Their work shows that historical heuristics outperform other types of heuristics such as dependency and code ownership heuristics.

Dependency analysis, or understanding how components interact with each other to ensure all necessary changes have been made, has also been widely researched. For instance, Zhifeng and Rajlich [1] present an approach which examines the effect of concealed dependencies on the process of change propagation. They present an algorithm that warns about possible hidden dependencies. In another work, Rajlich [15][12] proposes to model the change process as a sequence of snapshots, where each snapshot represents one particular moment in the process, with some software dependencies being consistent and others being inconsistent. The model uses graph rewriting techniques to express its formalism. Arnold and Bohner [2][3] review several formal models to arrange the propagation pattern of the changes. These methods based on code dependencies and algorithms, such as slicing and transitive closure, have been presented to assist code propagation analysis. Our change propagation model builds on top of the intuition and ideas proposed in these models. In contrast to their work, we propose a simple model for change propagation and the use of various sources for information for propagating changes. We also empirically validate our ideas.

In comparison to all previous work we recognize the dynamic nature of software development and the large variance between different entities in a software system. We propose adaptive heuristics that vary across time and between entities. Our case study shows that these adaptive heuristics can capture the project state and entity characteristics and can outperform other traditional heuristics. For instance, the methods presented in [6][7] show an average precision and recall of around 50-60% and 20-30%, whereas our adaptive heuristics achieve recall and precision of around 73-78% and 64% respectively.

Several researchers have proposed the use of historical data related to a software system to assist developers gain a better understanding of their software system and its evolution. Gall *et al* [8] use CVS logs for uncovering logical coupling and change patterns between source code entities. Cubranic *et al.* [17] present a tool that uses bug reports, news articles, and mailing list posting to suggest pertinent software development artifacts. Our approach focuses on the source code and its evolutionary history as a good source of data for change propagation heuristics. Other types of data sources such as bug reports and mailing list posting can be used as data sources for heuristics as well. Once these sources of data are integrated into heuristics, our adaptive meta-heuristics can make use of them to build more effective change propagation heuristics. Other possible source of data are design rationale graphs such as presented in [18][19]. Yet these later approaches require a substantial amount of human intervention to build the data needed to assist developers in conducting changes.

8. Conclusion and Future work

Developers need tools and approaches to help them in propagating changes in large software systems. Mis-propagating changes will lead to bugs and will increase the cost of developing large software systems.

Much of the work in literature related to change propagation has focused on the use of static heuristics based on a single source of information such as historical co-change information or code dependencies. In this paper, we propose the use of adaptive change propagation meta-heuristics. These heuristics make use of a set of change propagation heuristics. By using various heuristics that are based on several source of information, our proposed adaptive heuristics can adapt to the current state of a software project and to the varying characteristics of the different entities in a software system. Through an empirical study of four large open source projects, we demonstrate that our adaptive heuristics can achieve 73-78% for Recall and

64% for Precision (as shown in Table 8). These results are 57% improvement over the traditional heuristics that are used by our adaptive heuristics. The performance differences are statistically significant based on a paired Wilcoxon signed rank test at a 5% level of significance (i.e. $\alpha=0.05$). Wilcoxon is resilient to strong departures from the assumptions of the t-test. Moreover our case study shows that the A-Rec heuristic outperforms the A-Freq heuristic at the start of a project while the frequency counts used by the A-Freq build up. Therefore, an ideal heuristic would use the recency technique at the start of a project then switch to the frequency technique later in the life of a project.

10. References

- [1] Zhifeng Y, Rajlich V., Hidden dependencies in program comprehension and change propagation, Proceedings 9th (IWPC 2001). IEEE Computer Society Press, pp.293–299, 2001.
- [2] R. Arnold and S. Bohner, Impact analysis toward a framework for comparison, In IEEE ICSM 1997, Montral, Quebec, Canada, 1993, pp.292–301, 1997
- [3] S. Bohner and R. Arnold, Software Change Impact Analysis. IEEE Computer Soc. Press, 1996.
- [4] J. Sayyad Shirabd, T.C. Lethbridge, and S. Matwin, Mining the Maintenance History of a Legacy Software System, ICSM'03, September 22-26, pp.95-104, 2003.
- [5] J. Sayyad Shirabd, T.C. Lethbridge, and S. Matwin, Supporting Software Maintenance by Mining Software Update Records, Proceedings of the 17th IEEE ICSM, pp. 22-31, 2001.
- [6] Ahmed E. Hassan and Richard C. Holt, Predicting Change Propagation in Software Systems, Proceedings of ICSM 2004, September 11-17, 2004, pp.284-293
- [7] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", IEEE Transactions on Software Engineering, vol. 31, no. 6, pp. 429-445, 2005.
- [8] Gall, H., Hajek, K., and Jazayeri, M., Detection of Logical Coupling based on Product Release History, in Proceedings of International Conference on Software Maintenance (CSM'98), 1998, pp. 190-199
- [9] Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. Transactions on Software Engineering and Methodology, 11(3):1-38, July 2002
- [10] D. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053 – 1058, 1972.
- [11] Gallagher, K. and Lyle, J., Using Program Slicing in Software Maintenance, Transactions on Software Engineering, vol. 17, no. 8, August, pp. 751-762. 1991
- [12] Chen, K. and Rajlich, V., RIPPLES: Tool for Change in Legacy Software, in Proceedings of International Conference on Software Maintenance (ICSM'01), Florence, Italy, pp. 230-239, November 07-09 2001.
- [13] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., Predicting Source Code Changes by Mining Change History, IEEE Transactions on Software Engineering, vol. 30, no. 9, September, pp. 574 – 586, 2004.
- [14] Kiczales. G, Lamping. J, Menhdhekar. A, Maeda. C, Lopes. C, Loingtier. J-M, Irwin. J, Aspectoriented programming. In: Akit M, Matsuoka S (eds) Proceedings of the 11th European Conference on Object-oriented Programming, vol. 1241. Springer, Berlin Heidelberg New York, pp 220–242, 1997.
- [15] V. Rajlich. A model for change propagation based on graph rewriting. In IEEE International Conference Software Maintenance (ICSM 1997), pages 84–91, Bari, Italy, 1997.
- [16] L. C. Briand and J. Wust and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 475–482, Oxford, England, UK, Aug. 1999.
- [17] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In Proceedings of the 25th International Conference on Software Engineering (ICSE 2000), pages 408– 419, Portland, Oregon, May 2003. ACM Press.
- [18] E. L. Baniassad, G. C. Murphy, and C. Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In IEEE 25th International Conference on Software Engineering, Portland, Oregon, USA, May 2003.
- [19] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In IEEE 24th International Conference on Software Engineering, Orlando, Florida, USA, May 2002.
- [20] Mockus A, Votta LG, Identifying reasons for software change using historical databases. In: Proceedings of the 16th International Conference on Software Maintenance. San Jose, California, pp 120–130, October 2000.
- [21] Ahmed E. Hassan, Mining Software Repositories to Assist Developers and Support Managers PhD Thesis, School of Computer Science, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 2004
- [22] Xiaotong Zhuang, Hsien-Hsin S. Lee, Reducing Cache Pollution via Dynamic Data Prefetch Filtering, IEEE Transactions on Computers, vol. 56, no. 1, pp. 18-31, January, 2007