MULTI-PERSPECTIVE EXPLORATORY ANALYSIS OF SOFTWARE DEVELOPMENT DATA

JOSE RICARDO DA SILVA JUNIOR,

ESTEBAN CLUA[*] and LEONARDO MURTA[†]

Instituto de Computação, Universidade Federal Fluminense
Rua Passo da Pátria 156, Niterói, Rio de Janeiro, Brazil
{jricardo, esteban, leomurta}@ic.uff.br

ANITA SARMA

Computer Science and Engineering, University of Nebraska
1400 R St, Lincoln, NE 68588, United States
asarma@cse.unl.edu

In this paper, we present Dominoes, an approach for analyzing software repositories with thousands of artifacts by considering multiple perspectives of the software development data. In order to achieve computational power we model the data and its relationships as matrices, making possible to efficiently process them with a GPUs (Graphics Processing Unit) based architectures. Dominoes can support automated exploration of different relationships among project artifacts, where users have the flexibility to interactively combine and compose them. Our solution organizes data extracted from software repositories into multiple matrices that can be treated as domino pieces (e.g., [commit|method]). The connection of such pieces corresponds to a set of matrices operations, which derive additional domino pieces. These derived domino pieces represent specific project entity relationships (e.g., number of commits in which two methods co-occurred) and can be used for further explorations. As an evaluation of the Dominoes framework we present two exploratory case studies based on Apache Derby. First, we use Dominoes to show how dependencies among artifacts can be derived. Then, we identify expertise of developers by considering the commits that developers make to artifacts. We show that identifying relationships among 34,335 elements along 7,578 commits takes about 0.2 minutes in GPU, while the same processing in CPU takes about 413 minutes. Besides, identifying expertise of developer on a set of 34,335 files and 36 developers takes about 0.1 minute in GPU, whereas in CPU it takes 324 minutes.

Keywords: Exploratory data analysis; software dependencies; developer expertise; GPU computing.

1. Introduction

When working on software projects, developers often need to answer numerous questions, such as: "*which other methods do I need to edit if I make this change?*"; "*who was the person that last edited this method?*"; "*who do I need to coordinate my changes with?*"; "*who is the expert in a specific file?*" and so on [1]. Since software development leaves behind activity logs (i.e., commits recorded in the version control system and tasks recorded in the issue tracking system), it is possible to answer such questions by analyzing these software repositories. However, finding answers to these questions in the software repositories is not trivial, especially when there is an extensive amount of data that is accrued over the project's lifecycle and when this data is (typically) stored across different repositories [2]. This makes creating the right queries a nontrivial task [1].

Several approaches attempt to help in project explorations. For example, Tesseract [2] allows interactive investigation of relationships among files, developers, and issues through a network representation. Information Fragments [1] allows a user to compose information from tasks, change sets, and teams to explore the relationships between these entities. CodeBook [3] builds a graph of all relationship, and then provides specific applications for answering questions (e.g., finding related developers or artifacts). EEL [4] identifies expertise in a team by analyzing past commits of developers.

However, there are several constraints with these current approaches. First, these tools often focus only on a particular development aspect (e.g., EEL [5] primarily helps in expert identification). Second, these tools typically allow explorations of specific relationships that are fixed a priori (e.g., Tesseract preprocesses the sets of dyadic relationships first). Third, these tools often need a complete history of the project (e.g., in order to traverse the relationship graph, Codebook requires the full history). Fourth, all these tools operate at a predefined granularity level (usually high level, such as file), while navigating from fine-grain to coarse-grain and vice-versa is essential for exploratory analysis. Finally, these tools need to restrict the data that can be analyzed, because performing interactive data analytics of software archives through visual explorations of relationships among project elements is infeasible at the scale of operation that is needed.

In this paper, we present Dominoes, a novel approach designed to enable interactive exploratory analysis of relationships of different software entities at varying levels of granularity by utilizing matrices operations processed in GPUs (Graphics Processing Unit) architectures. Our approach organizes data from a software repository into multiple matrices that are treated as domino tiles, such as [developer|commit], [commit|method], [class|method], amongst many other combinations. Just as in the Dominoes game, where joining two congruent squares edge to edge can form a rectangle, our matrices can be combined to create additional (derived) matrices. This derivation process is guided by a set of matrices operations, such as addition, multiplication, and transposition. For example, a computation of logical coupling at the method level can be achieved as [method|method] = $[\text{commit}|\text{method}]^T$ × [commit|method]. With this new (derived) domino tile, we can derive dependency among developers as [developer|developer] = [developer|commit] × [commit|method] × [method|method] × $[\text{commit}|\text{method}]^T$ × $[\text{developer}|\text{commit}]^T$. Many different combinations are possible, with each derived domino tile representing a particular aspect in software engineering.

A primary goal of Dominoes is to enable users to explore the relationships in their project elements across different levels of granularity. Therefore, the granularity aspect is a central architectural element (e.g. [package|class], [file|class], and [class|method]). Connecting any other domino tile with these composition tiles or their transpose allows navigation from coarse-grained to fine-grained analysis or vice versa. However, fine-grained analysis can lead to extremely large data sets to be analyzed. In order to solve the scalability problem, Dominoes implements the exploratory analysis of software project entities as linear algebra operations over matrices, which can be parallelized in GPU [6]. This allows boosts in performance of about three orders of magnitude [7]. Therefore, Dominoes opens a new realm of exploratory software analysis, as endless combinations of domino pieces can be experimented and generated with in an exploratory fashion. It is important to state that we have already used GPU for solving software engineering problems. In a previous work [8] we achieved boosts of two orders of magnitude when running image *diff*, *patch*, and *merge* operations in GPU.

In our previous work [9], we introduced Dominoes and showed its feasibility by applying it in the identification of logical coupling among methods. This paper extends our previous work by (1) providing implementation details including the software architecture and (2) showing how Dominoes can also be applied in a completely different situation: identification of Expertise of Developers (ED) over artifacts in a project. As a first step we consider a file as the unit of analysis. By using our composition tiles, Dominoes could be easily tailored to perform finer-grained analysis at the level of methods or lines of code. This paper also provides more details about how Dominoes contrasts with related work.

The remaining of this paper is organized as follows: Section 2 presents the Dominoes approach, detailing its architecture, basic tiles extracted from software repositories, and operations that allow creation of new tiles. Section 3 presents two scenarios on which we base the experimental evaluation of Dominoes, which comprises the identification of logical coupling among methods and the identification of expertise of developers over files. Section 4 presents the evaluation of Dominoes in the aforementioned scenarios over the Apache Derby project. Section 5 contrasts Dominoes with related work. Finally, Section 6 concludes the paper and discusses future work.

2. Dominoes Approach

Dominoes extracts data from a software repository and converts them into multiple matrices (called "tiles" in the Dominoes nomenclature), correlating the desired attributes in lines and columns. This strategy allows data manipulation and its operations using massively parallel architecture. In our case, this fact allows interactive manipulations even with large datasets, as we are using GPUs to perform matrices operations.

We denominate a matrix as M and its transpose by using a superscript (M^T). Individual elements in a matrix are denoted as M[*i*,*j*]. The operator "×" represents matrix multiplication. It is important to note that when multiplying two matrices the number of columns in the first operand must be equal to the number of rows in the second operand. In our case, the column and rows of the operand over which we are multiplying also needs to be congruent (same project element), similar to the Dominoes game. In other words, we can multiply [developer|commit] × [commit|method], but not [developer|commit] × [method|method]. In the rest of this section, we describe the architecture of Dominoes and then focus on the matrix definitions and how we operate over them.

2.1. *Dominoes Architecture*

Dominoes architecture is designed in such a way that data from a software project repository is extracted and the associated change information is archived. Basically, it is composed of a set of modules responsible to extract and process data, as seen in Fig. 1.

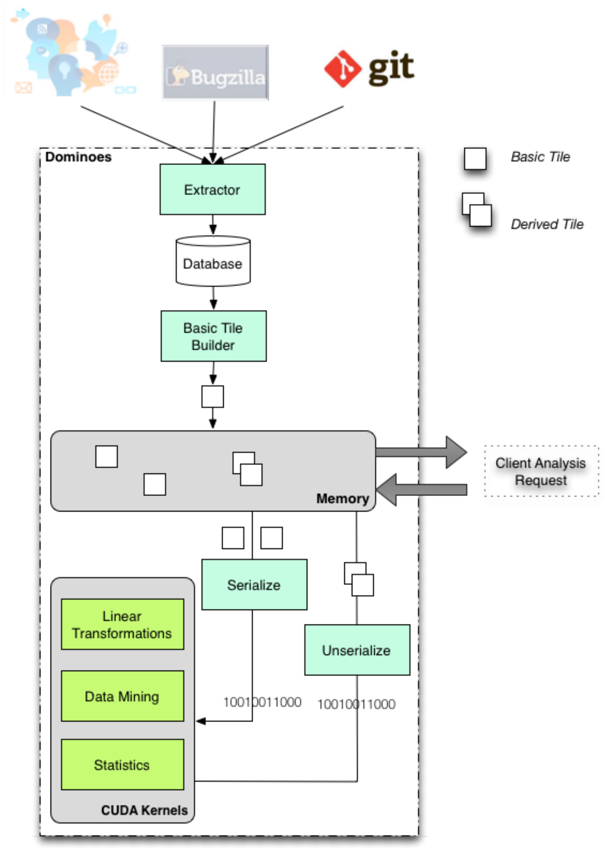

Fig. 1. Dominoes Architecture.

Currently, the **Extractor** module is responsible for mining Git projects (for version management) by cloning and accessing the local repository. The local repository is then preprocessed and it is generated a tree of all modifications performed in all commits by analyzing which files, packages, classes, and methods were modified. It is important to note that the information of each modification is decomposed to get a fine-grained view of the changes by using the Eclipse ASTParser (suitable for Java-based projects). For example, even if we represent changes at the package level (for a coarse-grained analysis), we know exactly which class was modified, as well as which methods were modified. This information is then stored in a relational database. Furthermore, after the initial data collection, information about subsequent changes can be updated incrementally to the database.

After the pre-processing stage, basic Dominoes tiles are constructed on the fly by the **Basic Tile Builder** module, which relies on querying the database in order to perform the desired relationship request, based on the granularity chosen by the user (e.g., File, Method, Package). These tiles then become available to the users, allowing them to manipulate the tiles according to their needs.

There are several additional manipulations of the data that is allowed by Dominoes: manipulating the set of building tiles as well as filtering their values. These manipulations include **Linear Transformations** (e.g., addition, multiplication, and transposition of matrices), **Data Mining** metrics (e.g., calculating confidence and lift in a tile), and **Statistics** operations (e.g., calculating the mean), as presented in Section 2.3. Basic building tiles can be further combined through linear transformation operations to yield derived tiles that allow exploration of derived project relationships. These derived domino tiles can also be saved as new template pieces in case that they will be used in other calculations and compositions. All tiles (basic and derived) are stored in memory, allowing their use as needed for analysis since the data is cached.

Performance becomes an issue when we compute relationships at a fine-grained level. Therefore, in order to allow efficient computation at the level of interactive speeds, we model the aforementioned manipulations as Single Instruction Multiple Thread (SIMT) architecture, making it possible to execute the intensive matrix operation computation at a GPU device. When a matrix manipulation is needed, Dominoes forks its execution by triggering the respective asynchronous GPU code (called *kernel*) according to the desired operation.

A GPU kernel is implemented in CUDA, a proprietary Nvidia programming language based on C, and is targeted to be executed only over GPUs. Basically a CUDA kernel is a function that generates thousands of parallel threads in the GPU device. While all these threads must work over the same code, they operate at different parts of the data. Since modern GPUs have thousands of cores, when the data is correctly distributed, it is possible to achieve speed-ups of two or even three orders of magnitude when compared with traditional multi-core CPUs [10], depending on the nature of the problem. Modeling the data structure as matrices allows optimal parallelization, especially in the case of operations that have only local data dependencies and avoid code divergence, as it is in our case.

Except for the CUDA kernel operations, Dominoes is otherwise developed in JAVA, which includes maintaining the tiles in memory. Performing operations over these tiles therefore, requires communicating the data with a C code, as kernels in CUDA must be programmed using the C language. Therefore, Dominoes implements a Java Native Interface (JNI) that is responsible for serializing and deserializing building tiles to and from C. During **serialization**, matrices are flattened to a vector and converted back to matrices during **deserialization**.

2.2. *Dominoes Tiles*

Dominoes includes basic building tiles, which can be combined to create derived building tiles, which can be further combined with other basic or derived tiles. The basic building tiles are created by extracting data from existing software repositories (version control systems, issue tracking systems, etc.). For example, commits, issues, discussions about a commit, or pull request can be collected from GitHub. The basic building tiles around commits include:

- [class|method] (ClM): relationship between a class and its constituent methods, where cell ClM[*i*,*j*] has a value of 1 when class *i* contains method *j*.
- [commit|method] (CM): relationship between commits and methods, where cell CM[*i*,*j*] has a value of 1 when commit *i* adds or changes method *j*. Note that the index *i* does not necessary express the commit id.
- [developer|commit] (DC): relationship between developers and their commits, where cell DC[*i*,*j*] has a value of 1 when developer *i* is the author of commit *j*.
- [bug|commit] (BC): relationship between commits and bugs, where cell BC[*i*,*j*] has a value of 1 when commit *j* fixed bug *i*.

These *basic building tiles* can then be combined to form a series of *derived building tiles*. In the following we show a small set of *derived building tiles* that can be computed using the multiplication and transposition operations:

- [method|method] ($MM = CM^T \times CM$): represents method dependencies, where MM[*i*,*j*] denotes the strength of the dependency of method *j* on method *i*. The rationale of this matrix is based on logical dependencies, as elements that are co-committed together share some program logic. Note that we can also create an MM matrix through program analysis, in which case it would be termed as a *basic building tile*. Such MM matrices have been explored by Steward [11] in Design Structure Matrices.
- [class|class] ($ClCl = ClM \times MM \times ClM^T$): represents class dependencies, where ClCl[*i*,*j*] denotes the strength of the dependency of class *j* on class *i*. Note that using the composition tile, we can easily provide analysis results at a higher-level of abstraction.
- [bug|method] ($BM = BC \times CM$): represents the methods that were changed to fix each bug. This matrix could be used to identify which methods are "buggy".
- [developer|method] ($DM = DC \times CM$): represents the methods that a developer has changed. This matrix could be used to identify experts on a particular method.

- [developer|class] (DCl = DM × ClM^T): represents classes that a developer has changed. DCl uses the composition operation to provide expertise information at the class level, which is typically used during bug triaging [3].
- [developer|developer] (DD = DM × MM × DM^T): represents the expertise dependency among developers, where developer j depends on some knowledge of developer i, because of underlying technical dependencies in their work. Note that here this *derived building tile* uses other *derived building tiles* (MM and DM) in its definition.

2.3. *Specialized Operations*

Our basic matrices are typically binary, that is, M[i,j] is either 1 or 0 for any i and j, whereas our derived matrices are not. This is largely because commits are atomic transactions and therefore most associated matrices with commits are binary. In the case of derived matrices, cell values have associated semantics. Simple operations such as multiplication and transposition allow us to compose different types of domino tiles to derive more complex matrices and, thereby, different software engineering constructs. However, there are three "specialized" operations that can be applied on derived matrices where individual cells are not binary.

Let us take the example of the MM matrix. The diagonal shows how frequently a method has been changed and each cell (M[i,j]) shows how frequently a method (i) has changed together with another method (j). This semantics is equivalent to the *absolute support*, largely adopted in the data mining community. The support of an item set is defined as the proportion of transactions in the dataset that contains the item set. According to [12], the rule $X \longrightarrow Y$ has support **s** if **s**% of transactions contain $X \cup Y$. As this operation pattern of multiplying a matrix by its transpose is very popular and semantically rich, we treat it as a specialized operation computed according to Eq. 1.

$$\mathrm{M}^{\mathrm{sup}} = \mathrm{M} \times \mathrm{M}^{\mathrm{T}} \tag{1}$$

The semantics of support allows us to answer software engineering related questions regarding the strength of the relationships. For example, if we are interested in predicting which other methods a developer needs to edit because of a change, we can use the concept of logical coupling (files that are committed together have underlying logical dependencies) to identify all those methods that are dependent on the edited method and may also need to be changed. We could use the $\mathrm{MM} = \mathrm{MC}^{\mathrm{sup}}$ matrix to answer this question.

Unfortunately, support is transitive, so $\mathrm{M}^{\mathrm{sup}}[i,j] = \mathrm{M}^{\mathrm{sup}}[j,i]$. Consequently, using support to represent dependencies is not precise, as program dependency is not transitive. In order to obtain a more precise relationship that reflects the direction of the dependency, Zimmermann et al. [13] use *confidence* to represent logical coupling. This metric suggests which artifacts should be modified together, given that a specific artifact is being modified. According to [12] the rule $X \longrightarrow Y$ has confidence **c** if **c**% of transactions that contain *X* also contain *Y*. In the context of our approach, when applied to MM matrix, confidence quantifies the occurrence of an entity (e.g., method) change given that the other entity (e.g., method) has also been changed. The confidence operator is computed according to Eq. 2.

$$\mathrm{M}^{\mathrm{conf}}[i,j] = \frac{\mathrm{M}^{\mathrm{sup}}[i,j]}{\mathrm{M}^{\mathrm{sup}}[i,i]} \tag{2}$$

Confidence does not have a transitive property among elements, so it is possible to define different levels of dependency for each pair. However, confidence suffers form another type of problem. In the context of data mining, confidence is used to quantify relations such as *"those who buy product A also buy product B"*. In this case, if product *B* is presented in almost all orders, purchase of any product will lead to a high confidence in buying *B*. For this reason, analyzing confidence alone tends to be imprecise, and can exhibit false relationships.

To address this problem we can use a third metric, called as *lift* [12]. Lift measures the influence of the antecedent in the frequency of the consequent. Formally, the rule $X \longrightarrow Y$ has lift **L** if the frequency of *Y* increases in **L** times when *X* occurs. According to this definition, we are interested in dependencies with lift greater than 1, as any other value implies irrelevant (coincidental) relationships. The lift operator is defined by Eq. 3, where the scalar multiplication by the number of commits ($\mathrm{M}^{\mathrm{rows}}$) transforms the absolute support ($\mathrm{M}^{\mathrm{sup}}$) into relative support (values ranging from 0 to 1).

$$\mathrm{M}^{\mathrm{lift}}[i,j] = \frac{\mathrm{M}^{\mathrm{conf}}[i,j] \times \mathrm{M}^{\mathrm{rows}}}{\mathrm{M}^{\mathrm{sup}}[j,j]} \tag{3}$$

3. Usage Examples

In this section we introduce a simple scenario to show how data mining concepts such as support, confidence, and lift matter when identifying artifact dependencies (Section 3.1). Additionally, Section 3.2 exhibits how Dominoes can be used to identify expertise across files.

3.1. *Calculating Method Dependencies*

Consider a scenario where three developers (*Alice*, *Bob*, and *Carlos*) work together on a "geometry project", containing four classes (Circle, Cylinder, Cone, and Shape). Circle has a method *circumf()* that calculates its circumference. Shape has a method *draw()* to render a shape. Finally both Cylinder and Cone have methods *area()* to calculate the area of the respective shapes. Table 1 describes five commits and their change descriptions. Table 2 shows which commits modified which methods. Note that this is an intentionally simple example to explain the concepts in the paper.

Fig. 2 represents the support, confidence, and lift values for the MM matrix, where Ci represents *Circle.circumference()*, Cy – *Cylinder.area()*, Co – *Cone.area()*, and S – *Shape.draw()*.

Table 1. Commits made by developers.

Commit #	Developer	Description
C_1	Alice	Change type of function parameter to compute the radius (Circle) and how to render it (in Shape)
C_2	Carlos	Change the side of Cone and how to render it
C_3	Alice	Change how a Shape is rendered
C_4	Alice	Calculation of how circumference and area are calculated using PI. Required modification on how to draw a Shape
C_5	Bob	Modify the height calculation of a cylinder and how it is rendered

Table 2. Methods changed for commit.

Commit #	Circle circumf()	Cylinder area()	Cone area()	Shape draw()
C_1	1	1	0	1
C_2	0	0	1	1
C_3	0	0	0	1
C_4	1	1	1	1
C_5	0	1	0	1

$$\underset{\text{Support}}{\begin{array}{c|cccc} & Ci & Cy & Co & S \\ \hline Ci & 2 & 2 & 1 & 2 \\ Cy & 2 & 3 & 1 & 3 \\ Co & 1 & 1 & 2 & 2 \\ S & 2 & 3 & 2 & 5 \end{array}} \quad \underset{\text{Confidence}}{\begin{array}{c|cccc} & Ci & Cy & Co & S \\ \hline Ci & 1 & 1 & 0.5 & 1 \\ Cy & 0.6 & 1 & 0.3 & 1 \\ Co & 0.5 & 0.5 & 1 & 1 \\ S & 0.4 & 0.6 & 0.4 & 1 \end{array}} \quad \underset{\text{Lift}}{\begin{array}{c|cccc} & Ci & Cy & Co & S \\ \hline Ci & 2.5 & 1.6 & 1.2 & 1 \\ Cy & 1.6 & 1.6 & 0.7 & 1 \\ Co & 1.2 & 0.8 & 2.5 & 1 \\ S & 1 & 1 & 1 & 1 \end{array}}$$

Fig. 2. Support, Confidence, and Lift calculated from previous scenario.

If we consider the confidence matrix, we notice that the dependency from *Cy* to *Ci* (100% confidence – row 1 column 2) is stronger than from *Ci* to *Cy* (60% confidence, row 2, column 1), because whenever *Ci* was changed it also required changes to *Cy* (commits C_1 and C_4) (see Table 2). However, *Cy* was changed once without *Ci* (commit C_5). With such a confidence analysis we can state that *Cy* (always) depends on *Ci*, but *Ci* does not necessarily depend on *Cy*. Therefore, using confidence to derive the DD matrix would identify that Bob should communicate with Alice, but not necessary the opposite way.

The confidence matrix also indicates high dependency from *S* to all other methods. However, this occurs not because *S* really depends on all other methods, but because *S* was independently changed in all commits (see Table 2). The lift matrix eliminates such coincidental dependencies, keeping only dependencies between *Cy* and *Ci*, and *Co* and *Ci*, since all other values are either equal to or below 1.

In summary, support alone is not sufficient to indicate dependencies among project entities, but helps in eliminating dependencies that appear by chance (e.g., *Co* and *Ci*). In a large project, with thousands of commits, thresholding on a predefined support level can help to eliminate accidental dependencies. On the other hand, lift plays a complementary role of identifying dependencies to elements that are very frequent (e.g., *S*) and, therefore, may be a cause of coincidental changes. Finally, confidence is important to identify the direction of the dependency (e.g., from Cy to Ci). With such an analysis, we find that the only real dependency in our scenario is from Cy to Ci, which would lead to a communication requirement from Bob to Alice in the DD matrix.

Our approach, therefore, provides four distinct advantages. First, the confidence measure allows more nuanced investigations (e.g., direction of dependency). Second, the use of lift measure increases the accuracy of finding by filtering out common, but unrelated changes. Third, the fine-grained analysis from the method level increases accuracy, since we can identify dependencies among individual methods. Therefore, if we find that *Cy* depends on *Ci*, we can find that Bob needs to coordinate with Alice, who is working on *Ci*, and not another developer who is working on the same file (Circle, but on a different method). Finally, GPU processing allows these investigations to be performed interactively.

3.2. *Identifying Expertise in a Project*

In this section, we introduce our strategy for identifying Expertise of Developers (ED) over artifacts in a project by considering the entire artifacts as an atomic element. This strategy is vastly adopted in the literature [2], [5], [14] and is based on the frequency of changes to an artifact (e.g., file, project, etc.) made by a given developer. Frequency of edits has long been used as a proxy for identifying the knowledge that a developer has about an artifact, typically a file [15]. The intuition is that the more someone has edited a file, the more working knowledge the developer has about that file. The frequency of edits can therefore help to answer two related questions: (1) which developer is an expert for a given file, and (2) which files is a given developer an expert of.

Table 3 uses the same scenario from previous section but now presenting information of which file each one has worked on ([developer|file] – DF for short). It is important to note that we arrived at the DF matrix by operating over the basic tiles ([developer|commit] x [commit|method] x [file|method]^T). The cells in the derived matrix DF represent the number of times a developer d_i changed methods inside f_j. Besides that, Table 3 also shows the number of commits performed by each developer (note that it is different from summing all columns in a row, as a commit could comprise multiple method changes on more than one file).

Table 3. Developer x File (DF matrix).

Project	*Circle.java*	*Cylinder.java*	*Cone.java*	*Total Commits*
Alice	14	2	20	28
Carlos	10	24	12	25
Bob	25	10	8	40

To answer the first expertise question (who is an expert for a given file f_j), we search for the developers who edited the file the most. This is done by scanning down the column of f_j in the DF matrix. In our simplistic example (see Table 3), if we want to identify an expert for *Cone.java*, we would select Alice. Carlos would be considered as the second most knowledgeable developer of that file.

To answer our second question, about the expertise of a specific developer d_i, we scan the rows in the matrix for the highest values. In our example, we find that Alice has expertise in *Cone.java*, Carlos in *Cylinder.java*, and Bob in *Circle.java*.

4. Case Study: Apache Derby

In order to evaluate Dominoes in a real environment, we perform two case studies by using the history of Apache Derby[a], an open source relational database project. The first study shows how to calculate artifact dependencies. It also demonstrates how solely working with support is error prone. The second study demonstrates how Dominoes can be used for expertise identification in a project. All analyses were performed by considering the repository data from 08/11/2004 to 01/23/2014, which in total comprises **7,578** commits, **36** distinct developers, **34,335** files, and **305,551** methods committed during approximately 10 years. The case study was performed at the file-level for easier interpretation of the results. However, as discussed before, Dominoes can easily navigate from coarse to fine-grained analysis and vice-versa by using the "composition" operation.

4.1. *Calculating dependencies in Derby*

We first create the MM matrix (dependencies between methods based on co-commit information) and then apply the composition operation to calculate the ClCl (class to class) matrix. We found that in Derby a file was associated with only one class, therefore, for our purposes the ClCl matrix is the same as the FF ([file|file]) matrix. We then applied the confidence and lift analysis on this matrix. We found that due to the characteristics of Derby, the lift analysis does not filter out any coincidental dependencies. This is because the Derby file dependencies are highly clustered, causing low support and a very high lift. When we filter the lift values by thresholding it with "1", no data points were eliminated. Therefore, here we restrict our discussion to the support and confidence analysis results.

[a] Derby Repository: https://github.com/apache/derby.

Table 4. Top 5 logical dependencies in terms of high support and biggest confidence difference.

Artifact A	Artifact B	Support	Conf. (A-B)	Conf. (B-A)
DataDictionary.java	DataDictionaryImpl.java	79	0.88	0.37
DD_Version.java	DataDictionaryImpl.java	45	0.78	0.21
LanguageConnectionContext.java	GenericLanguageConnectionContext.java	44	0.86	0.48
DRDAConnThread.java	DRDAStatement.java	37	0.22	0.68
ResultSetNode.java	SelectNode.java	36	0.54	0.45

Table 4 presents the top 5 logical dependencies in terms of support and with the biggest difference in confidence. It is important to remember that confidence is not transitive.

Considering the first case as an example, it is possible to observe that using the common approach that is based on support, artifacts *DataDictionary.java* and *DataDictionary-Impl.java* would be considered as dependent to each other as they have a high support (in fact, 79 is the highest value of absolute support in the whole system). However, when observing the confidence, it is possible to see that only *DataDictionaryImpl.java* has dependency with *DataDictionary.java*, which is reasonable as changing a method implementation normally does not result in a change to its interface. The following two rows are also interface/implementation cases, presenting the same behavior. In the fourth row, we have a composition case, where *DRDAConnThread.java* possesses a *DRDAStatement.java* instance. In this case, modifying the former does not necessary imply a modifications in the latter. However, there is a high likelihood of a related change in the opposite direction, that is, modifications in *DRDAStatement.java* can change method signatures used by *DRDAConnThread.java*, for instance.

Finally, the last case is a class specialization, which normally requires modification to both files, with a slightly higher dependence from the subclass to the superclass. These analyses show the importance of using confidence to identify the direction of the dependencies.

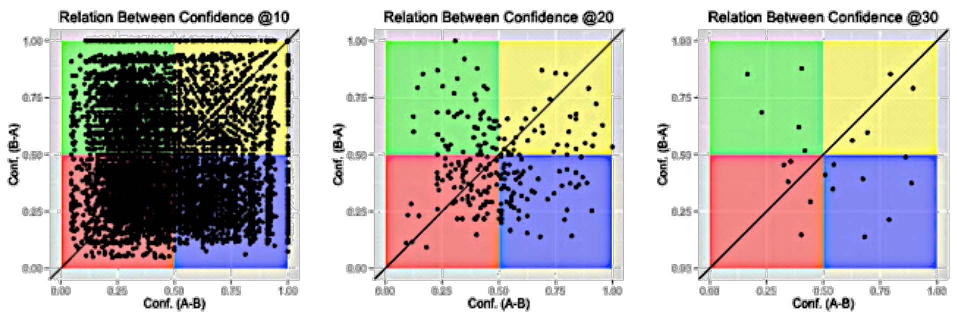

Fig. 3. Relation among confidence for various support threshold. The leftmost chart considers a threshold of 10, while the middle uses 20, and finally the rightmost uses 30.

Besides these five top dependencies, Fig. 3 presents a scatter plot chart with all dependencies at three specific support levels (10, 20, and 30). This chart plots each dependency according to its confidence in both directions (A-B and B-A). This way, dependencies with the same confidence value in both directions are plotted along the diagonal. As we can see, however, there are several cases where points are located far from the diagonal. When we consider the rightmost chart in Fig. 3 (support threshold at 30) for discussion, we can observe some distinct patterns. The red quadrant shows that both **conf(A-B)** and **conf(B-A)** are less than 0.5, thus containing weak bidirectional dependencies. The yellow quadrant, on the other hand, shows dependencies where both **conf(A-B)** and **conf(B-A)** are above 0.5, thus containing strong bidirectional dependencies. Finally, the green and blue quadrants show unidirectional dependencies with highest divergences among confidence. In this case, dependencies from these quadrants can be erroneously classified as bidirectional if we solely use support to analyze dependencies.

Performing an analysis such as the one in Fig. 3 can unveil how inaccurate dependencies extracted from support-based approaches tend to be. As demonstrated for the Derby project, only dependencies in the yellow-quadrant should be classified as bidirectional. Both blue and green quadrants present unidirectional dependencies.

In this evaluation, the CCl (i.e., [commit|class]) matrix was of size 7,578 × 34,335. The generation of CCl^{sup} and CCl^{conf} using GPU (NVidia GeForce GTX 580) took about 0.2 minutes. However, performing the same computation using CPU (Intel Core 2 Quad Q6600) took 413 minutes. This shows that we get a speedup of three orders of magnitude when using GPU – with just the simple calculation that requires one transposition and three multiplication operations. Similarly, when we process MC (i.e., [method|commit]), with size of 305,551 × 7,578, it takes about 0.3 minutes in GPU. This calculation was infeasible to do so in a reasonable amount of time when processed on CPU (after waiting for 720 minutes). This makes CPU processing of data infeasible when analyzing fine-grained project data.

4.2. *Identifying expertise in Derby*

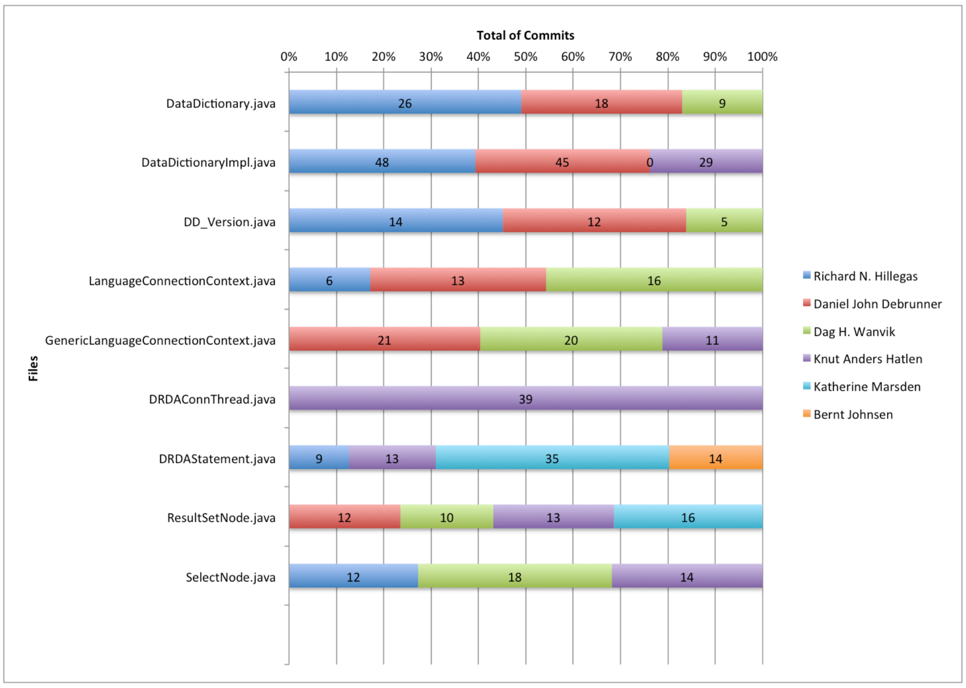

Fig. 4. Expertise of developer for files listed in Table 4. Each bar in the chart represents the total percentage of changes to a file, where each segment displays the number of commits made by a developer (color coded).

Expertise of a developer is processed by analyzing the data about methods that were modified (and committed) by a developer. However, here we present the results at the granularity of files for simplification of presentation, with out any loss of generalization. Further, we discuss results for the files shown in Table 4. The expertise of a developer for these files is presented in Fig. 4.

When we go back to the two expertise questions that we stated earlier in Section 3.2, we can find the answer for the first question (*which developer is an expert for a given file?*) by examining the developer who most committed the file under consideration. Using the set of artifacts presented in Table 4 among its specific relationships (the first three rows represent interface/implementation; the fourth represents a composition; and the last row represents specialization), it is possible to observe from Fig. 4 that Richard has the highest expertise in *DataDictionary.java* and *DataDictionaryImpl.java*. This result suggests that he has the highest expertise in the interface as well as its implementation. The same phenomenon is observed for *DD_Version.java* and *DataDictionaryImpl.java*, where the developer with the higher expertise (also Richard) is the same for the interface and its implementation. For the third interface/implementation case, represented by files *LanguageConnectionContext.java* and

GenericLanguageConnectionContext.java, we find that there are two experts – Dag and Daniel, respectively.

The composition case (represented in the fourth line of Table 4) between *DRDAConnThread.java* and *DRDAStatement.java* is also reflected in Fig. 4, as it is possible to see that Knut is the only expert in *DRDAConnThread.java* while *DRDAStatement.java* has Katherine as the highest expert. An implication of this finding about no other person having expertise over *DRDAConnThread.java* is that if Knut would leave the project then others in the project will have trouble to maintain *DRDAConnThread.java*. An advantage of using Dominoes is that project managers can use such information to being proactive and ensure that there are enough experts for each file in the project (or at least for the crucial files in the project). Finally the specialization case (last row in Table 4) represented by *ResultSetNode.java* and *SelectNode.java* has different expert developers: Katherine for the former, while Dag has the highest expertise for the latter.

Finally, answering the second question (*which files is a given developer an expert of?*) requires locating in the chart which file a developer has the most expertise in the project. We find that Richard and Daniel have the most expertise in *DataDictionaryImpl.java*; Dag in *GenericLanguageConnectionContext.java*; Knut in *DRDAConnThread.java* (and also being the only expert in this file), while Katherine and Bernt have the most expertise in *DRDAStatement.java*.

Identifying expertise of developers on a set of 34,335 files among 36 developers in the project (34,335 × 36 matrix size) takes about 0.1 minute when using GPU in Dominoes. When we performed the same operations using CPU (and keeping everything else the same) it took a total of 324 minutes.

5. Related Work

Determining Dependencies can be done by numerous approaches that focus on identifying structural dependencies (through syntactic analysis) or logical dependencies (through change history) amongst artifacts. Cataldo et al. [14] stands out as they use matrices to process dependencies among developers based on dependencies among artifacts. In their approach, both structural dependencies and logical dependencies become Task Dependency (T_D) matrices, and change requests, associating developers to artifacts, becomes Task Assignment (T_A) matrix. These matrices are used in an equation that indicates coordination requirements $T_A \times T_D \times T_A^T$. Our approach generalizes this idea by allowing different kinds of exploration over matrices. Finally, our identification of relationships is innovative, as it allows combining support, confidence, and lift, to compose the dependency matrix depending on the analysis need.

Exploratory Data Analysis tools provide either predefined questions or are very limited to derive information that was not conceived beforehand. In the case of Tesseract [2], for example, the available relationships are preprocessed and the matrices are fixed at a coarse grain (file-file, file-developer, file-bug, bug-developer). CodeBook [3] follows a similar approach that builds a graph of all relationship, which can then be used by applications for answering specific questions (e.g., identifying related developers or artifacts). Gall et al. [16] built a tool for mining software archives at a fine grain in order to compare source code changes. From these analyses, recommendations such as change type patterns and consistency of changes can also be made. Dominoes allows for a wide open set of questions that can be answered based on how the relationships are composed by the user during exploration. Also, our architecture in modeled in a way that any repository can be plugged into the system, avoiding the necessity to build a new tool.

Recommender Systems are approaches designed to help decision-making. McDonald and Ackerman [4] introduced Expertise Recommender (ER), which is based on two heuristics for recommending developers for specific tasks: tech support and change history. The tech support heuristic uses an issue database to search for similar situations and recommends the people involved in previous situations. The change history heuristic states that the last person that changed an artifact is a good candidate for changing it again. Unfortunately, the latter heuristic places a high weight on the most recent changes and ignores the past, which might affect the quality of the recommendations. Anvik et al. [17] deal with the problem of recommending developers for a specific issue by using machine learning techniques exclusively over an issue database, avoiding dealing with source code artifacts. Kagdi et al. [18] proposed a system for assisting in the tasks of allocating developers for changing a given file. It considers three metrics to compose a ranked list of recommended developers: contribution, activity, and recency of changes. The contribution metric indicates the number of commits each developer has made for a file. The activity metric indicates the number of days the developer has committed at least once in the project. The recency metric indicates the date of the last commit of each developer. This approach works only at coarse grain (i.e., files) and is not designed to support online exploratory analysis. Instead of a recommendation system, Dominoes provides a generic and flexible platform for exploratory analysis of project elements at any granularity level, which is compatible with multiple data types and relationships. Its interactive capabilities are mainly possible due to the adoption of the massively parallel architecture of the GPU.

6. Conclusion

Dominoes is an exploratory data analysis approach that allows users to select information about different project elements and their interrelationships from a repository. Relationships are represented by matrices, defined as *basic building tiles* and *derived building tiles*. Both kinds of building tiles can be combined iteratively to reveal deeper, complex relationships. Through such explorations, relationships that have not been computed or published before can be discovered via operations over these building tiles. As all operations are performed in parallel over GPU, exploratory analysis can occur seamlessly at interactive rates, even when computing relationships in fine-grained data. The current version of Dominoes tool extracts data from a Git repository and operates over the matrices by using GPU kernels implemented in CUDA.

Our evaluation contrasted the use of support alone and the use of support and confidence to distinguish the dependence directions. In the case of Apache Derby, employing confidence leads to a more accurate analysis for finding dependencies among artifacts. Moreover, using confidence for thresholding a relationship is more natural for the user, as it represents a normalized value. Besides that, we demonstrated how we can identify an expert developer for a file in a project. This information can be valuable during task assignment, where a task needs to be assigned to a developer who knows the content (file) best; as well as during development, where a developer may seek help from an expert.

The Dominoes architecture was intentionally designed to easily accommodate the definition of new *basic building tiles*, such as relationships mined from communication channels (e.g., email, chat, discussion forums). The same extensibility feature also applies for operations. Besides the basic matrix operations, such as multiplication and transposition, specialized operations can also be easily created and plugged into Dominoes, as showed in section 2.3 for support, confidence, and lift. This makes Dominoes a key contribution to the scientific community, as empirical studies can be reproduced over different corpora in order to validate an investigation. This has the potential of alleviating the pain of setting up an environment for each trial of an investigation.

Although we currently use matrices and GPU underneath Dominoes, other data representations and execution environments could be adopted in the future. For example, relational algebra is a compelling alternative to link sparse data. Moreover, SMP is the *de facto* architecture of modern personal computers. In this case, some kinds of analysis, such as reachability (used in impact analysis), can heavily benefit by operating over matrices in GPU.

A concept not discussed in this paper, which is currently under development, is the use of three-dimensional (3D) building tiles. These 3D building tiles represent time as the third dimension over the matrices. We posit that using this additional dimension would allow us to observe the evolution of relationships in the project over time. Another ongoing work is on creating visualizations of basic and derived building tiles both in two and three dimensions to support visual explorations of data by end users.

Acknowledgments

This work is partially supported by CNPq, CAPES, FAPERJ, and NSF (awards HCC-1110916, 1314365, and CCF-1253786).

References

[1] T. Fritz and G. C. Murphy, “Using Information Fragments to Answer the Questions Developers Ask,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA, 2010, pp. 175–184.

[2] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, “Tesseract: Interactive visual exploration of socio-technical relationships in software development,” in *Proceedings of the*

31st International Conference on Software Engineering, Washington, DC, USA, 2009, pp. 23–33.

[3] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA, 2010, pp. 125–134.

[4] D. W. McDonald and M. S. Ackerman, "Expertise Recommender: A Flexible Recommendation System and Architecture," in *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, New York, NY, USA, 2000, pp. 231–240.

[5] S. Minto and G. C. Murphy, "Recommending Emergent Teams," in *Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07*, 2007, pp. 5–5.

[6] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *ACM SIGGRAPH 2003 Papers*, New York, NY, USA, 2003, pp. 908–916.

[7] S. Rajasekaran, L. Fiondella, M. Ahmed, and R. A. Ammar, *Multicore Computing: Algorithms, Architectures, and Applications*. New York, NY: CRC Press, 2013.

[8] J. R. da Silva, T. Pacheco, E. Clua, and L. Murta, "A GPU-based Architecture for Parallel Image-aware Version Control," in *2011 15th European Conference on Software Maintenance and Reengineering*, Los Alamitos, CA, USA, 2012, vol. 0, pp. 191–200.

[9] J. R. da Silva Junior, L. Murta, E. Clua, and A. Sarma, "Exploratory Data Analysis of Software Repositories via GPU Processing [Unpublished Manuscript]," in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*, 2014.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General-purpose Applications on Graphics Processors Using CUDA," *J Parallel Distrib Comput*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008.

[11] D. V. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Trans. Eng. Manag.*, vol. EM-28, no. 3, pp. 71–74, 1981.

[12] W.-Y. Lin, M.-C. Tseng, and J.-H. Su, "A Confidence-Lift Support Specification for Interesting Associations Mining," in *Advances in Knowledge Discovery and Data Mining*, M.-S. Chen, P. S. Yu, and B. Liu, Eds. Springer Berlin Heidelberg, 2002, pp. 148–158.

[13] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Softw. Eng. IEEE Trans. On*, vol. 31, no. 6, pp. 429–445, 2005.

[14] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, New York, NY, USA, 2008, pp. 2–11.

[15] J. Anvik and G. C. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions," *ACM Trans Softw Eng Methodol*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.

[16] H. C. Gall, B. Fluri, and M. Pinzger, "Change Analysis with Evolizer and ChangeDistiller," *IEEE Softw*, vol. 26, no. 1, pp. 26–33, Jan. 2009.

[17] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?," in *Proceedings of the 28th International Conference on Software Engineering*, New York, NY, USA, 2006, pp. 361–370.

[18] H. Kagdi and D. Poshyvanyk, "Who can help me with this change request?," in *IEEE 17th International Conference on Program Comprehension, 2009. ICPC '09*, 2009, pp. 273–277.