I, Darius Cepulis, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled: Creating Socio-Technical Patches for Information Foraging: A Requirements Traceability Case Study

Student’s name: Darius Cepulis

This work and its defense approved by:

Committee chair: Nan Niu, Ph.D.
Committee member: Chia Han, Ph.D.
Committee member: Carla Purdy, Ph.D.
Creating Socio-Technical Patches for Information Foraging: 
A Requirements Traceability Case Study

Darius Cepulis

B.S. Computer Engineering, University of Cincinnati, 2018

Dr. Nan Niu, Chair
Dr. Carla Purdy
Dr. Chia Han

Thesis submitted to the Faculty of the
University of Cincinnati College of Engineering and Applied Science
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

July 13th, 2018
Cincinnati, OH
ABSTRACT

Work in information foraging theory presumes that software developers have a predefined patch of information (e.g., a Java class) within which they conduct a search task. However, not all tasks have easily delineated patches. Requirements traceability, where a developer must traverse a combination of technical artifacts and social structures, is one such task. We examine requirements socio-technical graphs to describe the key relationships that a patch should encode to assist in a requirements traceability task. We then present an algorithm, based on spreading activation, which extracts a relevant set of these relationships as a patch. We test this algorithm in requirements repositories of four open-source software projects. Our results show that applying this algorithm creates useful patches with reduced superfluous information.

Keywords: Information foraging theory, Spreading activation, Requirements Traceability
Copyright 2018, Darius Cepulis
Acknowledgments

A big thanks to Kristin, who turned my doodles into figures while I was furiously writing.

A huge thanks to Dr. Niu’s research team, who put together an enormous and useful database and answer set. All I had to do was query.

Finally, an enormous thanks to Dr. Niu, who provided the ideas that formed the backbone of this paper, and the guidance that made sure it got done.
Dedication

To Kristin, Stefan, Mom, Dad, Alex, Ryan, Stack Overflow, Audrey, Sofia, and iOS 12.
## Contents

1. Introduction ........................................ 1

2. Background ........................................ 5
   2.1 Information Foraging .......................... 5
   2.2 Socio-Technical Networks .................... 8

3. Examining Requirements Socio-Technical Graphs 11
   3.1 The Information Environment ................ 12
   3.2 Constructing Requirements Socio-Technical Graphs 15
      3.2.1 Nodes .................................. 15
      3.2.2 Edges .................................. 15
      3.2.3 Scripting Graph Construction .......... 18
   3.3 Properties of Requirements Socio-Technical Graphs in Information Foraging 19
3.3.1 One or Two Degrees of Separation From Question ........................................ 21

3.3.2 Collaborators and Contributors—Three or Four Degrees of Separation From Question ................................. 22

3.3.3 Five or More Degrees of Separation ................................................................. 25

3.3.4 Unconnected Users ......................................................................................... 25

4 Creating Socio-Technical Patches for Information Foraging .................................. 28

  4.1 Spreading Activation over Requirements Socio-Technical Graphs ......................... 29

  4.2 Delineating Patches within Spreading Activation RSTGs .................................... 33

  4.3 Results and Analysis ......................................................................................... 34

    4.3.1 Quantitative Evidence ................................................................................. 34

    4.3.2 Two Practical Examples .............................................................................. 35

5 Future Work ............................................................................................................ 41

  5.1 New Data Types: Text ....................................................................................... 42

  5.2 Systematic Spreading Activation Parameters .................................................... 45

    5.2.1 Weight ......................................................................................................... 45

    5.2.2 Frequency Incentives .................................................................................. 47

  5.3 New Domains ..................................................................................................... 47
6 Discussions

6.1 Implications ................................................................. 50

6.2 Conclusion ................................................................. 52

Bibliography ................................................................. 53

Appendices ................................................................. 62

A Graphs ................................................................. 62

B Code ................................................................. 67
List of Figures

2.1 Example of Spreading Activation in Neuroscience. “Dog” is initially activated; darker shading indicates higher activation. Source: [1] 7

2.2 The Codebook Topology, Source: [2] 10

3.1 A Screenshot of a Common Issue in Jira, in the DROOLS project. From this, we will take Issue, Assignee, Reporter/Created, and questions like the first comment. 13

3.2 Requirements Socio-Technical Graph 16

3.3 Degrees of Separation between Question and Answer in Projects 20

3.4 Referenced (1 Degree) 21

3.5 Creator or Assignee (2 Degrees) 22

3.6 Frequent Collaborator of Asker or Referenced User (3 Degrees) 23

3.7 Frequent Contributor to Issue (3 Degrees) 24
List of Tables

3.1 Jira Projects and Characteristics ........................................ 12
3.2 Select Questions with Identified Answers from DROOLS ............ 14
3.3 Number of Nodes Within N Degrees of Question .......................... 26
4.1 Patch Size With Cutoff (Samples from Figure 4.2) ......................... 35
5.1 Activating Comment and Issue Instead of Weighting Edges ............... 46
Chapter 1

Introduction

If we understand how a user seeks information, then we can optimize an information environment to make that information easier to retrieve. Pirolli and Card worked to understand information-seeking by defining information foraging theory [5]. Information foraging theory describes a user’s information search by equating it to nature’s optimal foraging theory: in the same way that scent carries a predator to a patch where it may find its prey, a user follows cues in their environment to information patches where they might find their information.

Information foraging theory has seen many applications since Pirolli’s seminal work. For example, in web search, foragers follow information scent to their patches, web pages [6, 7]. Understanding how foragers find information in web search has helped developers design the information environment of their web pages [7]. In code navigation and de-
1. Introduction

bugging, information foraging theory describes how developers seek to resolve a bug report by navigating from fragment to fragment of code to define and fix the problem [8]. By understanding the process of finding and fixing bugs, models can be written to assist in this process [9]. In both of these scenarios, the patch is clearly defined: in web search, a forager’s patch is a web page, and in debugging, a developer’s patch might be a fragment of code. What happens, though, when the patch is not clearly defined?

Consider socio-technical systems, where information artifacts are connected to people. Facebook, YouTube, Twitter, GitHub, and Wikipedia all have information artifacts, like posts and code snippets, with a rich context of social interactions tying them together. A forager traverses both the artifacts and the social structures behind them in an information seeking task. Therefore, both artifacts and social structures should be considered when defining a patch, and patches are not necessarily immediately evident. Consider a user wondering how a reposted image became so popular among their friends: how could patches be defined for this forager? Photos, posts, comments, friends’ connections? A GitHub code fragment or Facebook Friend can’t answer these questions alone, so patches must consist of some combination. In this paper, we describe a method for delineating such patches.

Requirements traceability is an ideal field for examining patch creation in a socio-technical environment. Requirements traceability is a socio-technical system used to describe and follow the life of a requirement by examining the trail of artifacts and people behind them, from the requirement’s inception to implementation. Requirements traceability problems,
as studied by Gotel and Finkelstein [10], arise when questions about the production and refinement of requirements cannot be answered. More specifically, with a traceability failure, US Food and Drug Administration might cast doubt in product safety [11]. With a traceability failure, the CEO of a prominent social media company cannot explain to Congress how a decision to withhold information from customers was made [12]. Applying information foraging to these problems could significantly increase efficiency and efficacy of these traceability tasks.

We relate requirements traceability to information foraging theory and its patches by considering requirements traceability questions. We define a requirements traceability question as a query that a project stakeholder issues in situ wanting to know a requirement’s life. A requirements traceability question is where a user’s traceability task becomes a foraging task; the question represents the user’s information need, or foraging prey. If questions represent a traceability forager’s prey, what represents a traceability forager’s patch? Put simply, we aim to answer the research question: where should a user search to understand their requirements traceability question?

This paper makes two contributions by deriving a method for delineating these patches. First, by examining requirements socio-technical graphs constructed from four requirements repositories containing 125 traceability questions, we identify classes of relationships that should be considered in similar requirements traceability tasks. Second, we derive an algorithm, based on spreading activation, which combines these classes with information foraging concepts to create relevant patches where foragers can conduct their
traceability tasks. The patches that our algorithm produces are as small as 5-10 nodes—a manageable quantity for a forager—representing knowledgeable users and useful information artifacts. The method for identifying these classes and deriving this algorithm can be extended to other socio-technical tasks.
Chapter 2

Background

2.1 Information Foraging

Information Foraging Theory (IFT) [13] simplifies, in a principled way, analyzing information-seeking tasks by providing constructs borrowed from its optimal foraging theory roots. Optimal foraging theory describes predators that pursue prey through an environment, following scent from locality to locality. The predator is always trying to optimize their task. In IFT, the information seeker pursues information through an information environment. Scent, in the information environment, is a construct that exists in the forager’s mind, representing their perception of where they might find information; this perception is shaped by proximal cues—hints provided by the information environment. Information foragers follow scent from locality to locality, information patch to information patch,
pursuing their prey.

Consider the following analogy: a forager is seeking berries. They go to a patch of berries, and estimate how many berries might be in that patch. They then pick berries until they decide that they have picked enough from this patch, and should move to the next one. In this model, proximal cues (how full the patch looks, how many berries nearby patches had), or scent, indicate to the forager how many berries might be in that patch. Of course, the forager might not estimate correctly how many berries there are if the indications are misleading, and they might switch to another patch too early (still easy berries to pick) or too late (time wasted looking for berries instead of switching patches). Building a correct estimate, and making berries easy to pick, so to speak, is an important application of IFT.

The constructs provided by IFT were first used to analyze how a web user might search for information online [6], modeling scent as relatedness of a link to the forager’s prey. This work eventually developed into the WUFIS (Web User Flow by Information Scent) algorithm [7]. WUFIS represents network topology as a graph, where nodes are web pages and edges are the links that a user can click to navigate from one to another. IFT’s scent is represented by relatedness of the words in a webpage to the forager’s information need. The algorithm then predicts where the user will navigate by applying spreading activation.

Spreading activation is a concept borrowed from neuroscience: Collins and Loftus [14] posited that all memories and knowledge in our minds are connected in a network, and activating one memory also activates neighboring memories. Spreading activation, as
adapted by the Information Foraging world, works by assigning one (or several) nodes in a graph with an initial activation of 1. Each edge in the graph is assigned with a weight. In the case of WUFIS, the weight is proportional to the similarity of the two nodes that the edge is connecting. In other cases, all edges are assigned the same weight. Each node in the graph is traversed, and the activation is “spread” to its children: typically, the parent node’s activation is multiplied by the weight so that the child node’s activation is lower. Each node, then, has an activation value that represents its similarity to the originally-activated node, as dictated by proximity to the original node and edge-weights. This mechanism can be seen in Figure 2.1. For WUFIS, spreading activation assigns each node with a value which represents the probability that a forager, given their current location and information need together with the scent of links connecting pages, will navigate to a specific page.
2. Background

This spreading activation concept was utilized to model programmer navigation in the development of PFIS [15] and its subsequent revisions [3, 16]. PFIS built upon the spreading activation of WUFIS by applying it to the field of developer navigation [15], inferring the forager’s goal [3], and creating multi-factor models with PFIS [16]. In the programmer navigation domain, WUFIS web page nodes were now code fragments, and its edges were any click-able link that would navigate a developer from one fragment to another. When inferring the forager’s goal, PFIS authors introduced the concept of heterogeneity to their network: in addition to linking code fragments, the PFIS algorithm also linked code fragments to key words (e.g., those extracted from a bug report), creating a more nuanced topology. Inspired by this heterogeneous approach, we take spreading activation to the socio-technical realm.

2.2 Socio-Technical Networks

In order to develop a spreading activation algorithm to answer requirements traceability questions in the socio-technical realm, we first examine work conducted in socio-technical graphs. Both Herbsleb and colleagues’ socio-technical theory of coordination (STTC) [17], and the Codebook project [2] attempt to answer requirements traceability questions with socio-technical models. We begin by examining Herbsleb et al.’s work.

Herbsleb et al. conduct their work examining dependencies among engineering decisions—“Who should work with whom?”. They answer this question by defining a con-
2. Background

constraint satisfaction problem that people must organize to solve. The theory draws from the key observation that the number of people was a powerful predictor of coordination problems (e.g., delay in development [18]), and quantifies people’s working relationships via a dependency network of work items. The solution to this problem provides an answer to the earlier “Who should work with whom?” question, which in turn serves as a unifying basis for solving a class of coordination problems, including expertise finding [19, 20], cross-site communication [21], and pull request acceptance in social coding environments [22]. While this STTC can address some of the traceability questions (e.g., ramifications [10]), other questions that we want to answer, such as “Which programmers are more error prone in their code according to the test results?” [23], need the relationships between people and artifacts. For a foundation that can answer all of our traceability questions, we turn to Codebook.

In the Codebook [2] project, people and work artifacts were “friends” in a social network. A user might be connected to an email they sent, bug they closed, and a commit they pushed; that commit has changes in code containing classes and calls. A sample Codebook topology can be seen in Figure 2.2. By using a single data structure to represent these people, artifacts, and relationships, and a single algorithm (regular language reachability) to analyze this graph, Codebook could handle all the inter-team coordination problems identified in a survey responded by 110 Microsoft employees [24], including requirements traceability problems.

Codebook addresses problems by having a project personnel cast their coordination needs
into regular expressions. For example, the requirements traceability question “Which program manager wrote the specification for that code?” could be addressed in Codebook with the regular expression “CODE MENTIONED BY WORD DOCUMENT AUTHORED BY PERSON”. This is a manual task, requiring a domain expert. In contrast, spreading activation can provide a mechanism for automated querying. We therefore adopt Codebook’s underlying data structure, but instead of regular language reachability, we adapt the people-artifact graph for spreading activation.
Chapter 3

Examining Requirements

Socio-Technical Graphs

In order to successfully create patches in a requirements traceability environment, we must first understand the characteristics of the environment. To do this, we construct graphs of the environment following the Codebook paradigm. We then examine the types of human-human and human-artifact relationships that connect a traceability question to an identified answer; encoding these relationships can build a patch that a requirements traceability forager can explore to understand their prey—their need to understand their traceability question—better.
3.1 The Information Environment

The literature suggests that issue trackers are essential for open-source projects to manage requirements [25–29]. Although the requirements of an open-source project can originate from emails, how-to guides, and other socially lightweight sources [30], the to-be-implemented requirements “eventually end up as feature requests in an issue-tracking system” [27]. We therefore turn to the issue-tracking system Jira, with issues like Figure 3.1 to understand the life of requirements.

From Jira, we select four open-source projects from the Apache software foundation [31] or the JBoss family [32]. The four projects tackle problems in different domains with implementations written in different programming languages, as seen in Table 3.1.

Within the chosen projects, we focus on questions and answers. As discussed, questions represent a traceability forager’s information need. By finding the exact person who answered a forager’s question, we can gain insight into the information environment; we

<table>
<thead>
<tr>
<th>Project</th>
<th>Domain</th>
<th>Written</th>
<th>Initial &amp; Latest Releases Examined</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DASHBUILDER</td>
<td>data reports</td>
<td>Java, HTML</td>
<td>Aug 27, 2014 &amp; Apr 14, 2016</td>
<td>31</td>
</tr>
<tr>
<td>DROOLS</td>
<td>business rules</td>
<td>Java</td>
<td>Nov 13, 2012 &amp; Jul 17, 2017</td>
<td>57</td>
</tr>
<tr>
<td>IMMUTANT</td>
<td>complexity reduction</td>
<td>Clojure</td>
<td>Mar 14, 2012 &amp; Jun 23, 2017</td>
<td>18</td>
</tr>
<tr>
<td>JBTM</td>
<td>business process</td>
<td>Java, C++</td>
<td>Dec 5, 2005 &amp; Jul 14, 2017</td>
<td>5</td>
</tr>
</tbody>
</table>
3. Examining RSTGs

Figure 3.1: A Screenshot of a Common Issue in Jira, in the DROOLS project. From this, we will take Issue, Assignee, Reporter/Creator, and questions like the first comment.
3. Examining RSTGs

Table 3.2: Select Questions with Identified Answers from DROOLS

<table>
<thead>
<tr>
<th>id</th>
<th>issue</th>
<th>body</th>
<th>asker</th>
<th>answered by</th>
<th>role</th>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1206</td>
<td>972</td>
<td>Can you please send a PR adding a new example...? …</td>
<td>Mario Fusco</td>
<td>Mauricio Salatino</td>
<td>Creator</td>
<td>add pull request</td>
</tr>
<tr>
<td>1220</td>
<td>963</td>
<td>This should be fixed for 6.4.0.Final, does it sound possible?</td>
<td>Mauricio Salatino</td>
<td>Petr Siroky</td>
<td>Assignee</td>
<td>change fix version</td>
</tr>
<tr>
<td>1371</td>
<td>907</td>
<td>...Please look at [my page]. What do you think…</td>
<td>Michael Kiefer</td>
<td>Geoffrey De Smet</td>
<td>Creator</td>
<td>marked as done</td>
</tr>
<tr>
<td>1377</td>
<td>905</td>
<td>Thanks...[martenscs]. Is there any other prefix? …</td>
<td>Petr Siroky</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1383</td>
<td>901</td>
<td>any news here [mfusco]?</td>
<td>David naranjo</td>
<td>Geoffrey De Smet</td>
<td></td>
<td>provide opinion</td>
</tr>
<tr>
<td>1627</td>
<td>823</td>
<td>Hi Geoff, I don’t know what should be the expected behaviour? …</td>
<td>Michael Kiefer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

know that the person who answered a traceability question can provide information on the traceability issue. For each project, two researchers identified comments that were questions (using tools from Stanford’s CoreNLP project [33] to identify questions in multiple forms, e.g., with/without a question mark) and identified the respective answer comments, building their answer sets individually. Some examples of the answer sets generated can be seen in Table 3.2. The researchers reached a substantial degree of agreement (average Cohen’s kappa=0.67) on requirements traceability questions over the 4 projects. Discrepancies were resolved in a joint meeting between the researchers.

With our information environment identified, the next step to creating patches is to build a requirements socio-technical graph (RSTG) so that we can inspect the topology of the environment using graph theory concepts. A graph requires nodes and edges.
3.2 Constructing Requirements Socio-Technical Graphs

3.2.1 Nodes

In Jira, issues (typically tasks or feature requests) are our requirements. Comments are provided to the issue, and may contain questions or answers. Users create issues, are assigned to issues, submit comments, and reference other users within their comments. This process can be seen in the screenshot of a typical Jira issue in Figure 3.1. Contemporary approaches to requirements traceability are either artifact-based (e.g., trace retrieval) and would consider only the comments and issues [34], or are driven by social roles (e.g., contribution structures in the requirements specification production) and would consider the relationships of the people [35]. As we constructed our answer set, however, we realized the impact of the people-artifact relationships on the ability to track a requirement’s life. Therefore, we consider Jira’s issues, comments, and people in our information environment. These will serve as our nodes. What, then, serve as our edges?

3.2.2 Edges

By manually inspecting the paths between questions and their eventual answers, we are able to define the edges in our network topology. Consider the following example: Figure 3.2 is a subgraph from the IMMUNANT project, showing two questions and their answers. User 2881 created Issue 73650. User 6655, the forager, commented
3. Examining RSTGs

Figure 3.2: Requirements Socio-Technical Graph
on the issue (Comment 149789), asking User 2881 for clarification by referencing them in the comment. User 2881 commented on the issue (Comment 149790), providing the clarification. In another foraging interaction, User 2881, now the forager, commented on the issue (Comment 149792), asking “This is now available in [incremental build — http://immutant.org/builds/2x/] 591 and newer. Can you give that a try and confirm it works for you?” User 6655 responded (Comment 149793) “I’ve finally found the time to test this and can confirm it works! Thanks!”.

These two foraging interactions demonstrate most of the relationships that we observed between nodes in the datasets. These relationships are treated as the edges within our graph:

- **Creator**: Users create issues
- **Commented, Asked**: Users post comments and questions
- **Comment, Question**: Comments and questions are posted to Issues
- **Referenced**: Comments can reference other users
- **Assignee**: (Not pictured) Users can also be designated assignees on issues

While some relationships could be considered as unidirectional, e.g., a user posting a comment, the comment also serves as a bridge connecting the issue to the user, who is knowledgeable on the issue. We therefore elected to encode the network as an undirected graph.
3.2.3 Scripting Graph Construction

With our desired node and edge types identified, we then automated graph construction in Python, leveraging the graph construction and analysis library, NetworkX [36], heavily. The full code described here can be found in our online repository [37], as well as in Appendix B. Our objective was to automate the construction of RSTGs like Figure 3.2 for each question, representing the state of the project at the time of the question. Representing the network at the time of the question was important to us so that only relationships that had existed prior to asking of the question would be considered.

In order to construct RSTGs, our script queried a database that held tables with the following characteristics. Note that irrelevant tables have been omitted:

- **jira_issue**: All projects’ issues, as well as assignee, creator, and a description

- **jira_issue_comment**: All comments, as well as who posted them on which issue

- **jira_issue_type**: Issue types, e.g., “Bug”, “New Feature”, “Improvement”

- **jira_project**: Keys of projects in database, as well as names and descriptions.

- **jira_user**: All of users’ usernames, real names, and database IDs

We began by pulling all of a project’s issues and associated users and adding them to a graph (dbManager.addProjectIssuesToGraph(project, graph)). This was performed by querying the database (SELECT * FROM JIRA_ISSUE WHERE PROJECT = X) and
adding a node for each result, each result’s creator, and each result’s assignee. The creators and assignees were connected to their issue with a labeled edge.

Next, each comment was added to the graph, and was connected to its issue, author, and any referenced user (dbManager.addProjectCommentsToGraph(project, graph)) with the query `SELECT * FROM JIRA_ISSUE_COMMENT WHERE ISSUE IN (SELECT ID FROM JIRA_ISSUE WHERE PROJECT = X)`. While each issue has its creator and assignee explicitly defined by Jira, and each comment has its author and issue, some additional processing was required to identify references. In practice, a reference can either be the user’s username (e.g., `@geoffrey`) or first name (e.g., “Geoffrey,” or “Geoffrey:”). With regular expressions (`\[˜(.+?)\]`), usernames were identified in the body of a comment and connected the comment to the user. However, a dictionary of first-name references had to be manually built so that an identified reference (regex: `[A-Z][a-z]+?, | [A-Z][a-z]+?:`) could be connected to its user.

### 3.3 Properties of Requirements Socio-Technical Graphs in Information Foraging

We now tie our RSTGs to information foraging theory. When a forager asks a question, like the ones in Figure 3.2, where do they find their answer? The location of the answer might grant us insight on what we should include in a patch of information relevant to a
question. What path might the traceability forager typically follow to the user who will provide the answer, gathering information on their prey along the way?

By analyzing the paths connecting question nodes and answer nodes, we identified recurring patterns connecting traceability questions with answers. The patterns are organized by degrees of socio-technical separation, which we define as the minimum number of edges to be traversed between two nodes. Figure 3.3 displays the frequency of degrees separating questions and answers for our 125 questions.

Figure 3.3: Degrees of Separation between Question and Answer in Projects
3.1 Examining RSTGs

3.3.1 One or Two Degrees of Separation From Question

Figure 3.3 shows that more than three-quarters of answers are within two degrees of the question. These represent simple foraging tasks. By the time a forager posts a question, they may be relatively familiar with an issue. The very fact that they chose a specific issue to comment on demonstrates that they expect their answer to be near the issue. A well-informed forager will often reference the exact user they expect to know the answer. These kinds of relationships fall within this category.

![Figure 3.4: Referenced (1 Degree)](image)

Within one degree of the question, two types of answers were observed. The first is when the forager answers their own query. Because the forager, in the RSTG, is directly connected to their comment, this is one degree of separation. The second is when a user is referenced within a comment. As shown in Figure 3.4, when a user is referenced in a comment, they are directly connected to the comment. Comment 70842, from the DROOLS project, is a good example of this: "What should the URL look like in your opinion ["tirelli"]?" asks User 4079. User tirelli answers. A vast majority of the 31% of answers one degree away from the question came from direct references.

Within two degrees of separation, we observed two more types of answers: when a for-
ager asks a question on an issue, either the issue creator or issue assignee often responds. Figure 3.5 shows this interaction. When a forager posts a question to an issue without a reference, the creator, with their knowledge of what the issue is, or the assignee, with knowledge of what is being done to solve the issue, can answer. In Comment 353748, from the JBTM project, User 133 posts to an issue: “Status update please?” User 3619, assignee, provides an answer.

3.3.2 Collaborators and Contributors—Three or Four Degrees of Separation From Question

More challenging foraging occurs when the user with the answer is three or more degrees of separation away from the question. With each degree, the number of information artifacts and other users a forager must traverse increases exponentially. Identifying patterns and presenting useful patches to the forager in this class of foraging presents far greater potential. We present a few of the most frequent types of interactions that occur within this radius.

Most answers in our projects within three degrees of separation were of the type shown
3. Examining RSTGs

in Figure 3.6: Comment–Person–Issue–Person. The question connects to a person—either the traceability forager or a referenced user—and the answer comes from a Frequent Collaborator. While neither the forager themselves nor a referenced user has an answer, someone they collaborate with does. The Collaborator is connected with the user or the referenced by one or more comments (where the Collaborator is directly referenced). We notice that Collaborators are often highly-central users in a project, whose Frequent Collaborator status arises from their frequent contributions to projects. In other words, they are Frequent Collaborators because they are frequent users in general.

The second type of answer within three degrees of separation is that shown in Figure 3.7: Comment–Issue–Comment–Person. We call this type of answer a Frequent Contributor. This user does not directly know the forager, nor are they creator or assignee of the issue. However, the Contributor has commented and asked on a given issue one or more times. Again, Frequent Contributors tend to be highly-central users. Figure 3.3 shows that Frequent Collaborators and Frequent Contributors represent 11% of answers in our
Within four degrees, we see another variant of The Collaborator: a *Frequent Collaborator of the Creator or Assignee* (Figure 3.8). This type of relationship is signified by the pattern Comment–Issue–Person–Issue–Person. The creator or assignee of the issue has collaborated with the answerer on other issues before, either as creator or assignee on those issues. While the creator or assignee of the primary issue being considered may not have the answer, someone they often work with may. Figure 3.3 shows that these kinds of interactions represent 3% of our dataset.
We hypothesize that, within three or four degrees of separation, several other types of collaborators could be observed. Frequent Collaborators of the Asker could manifest by Comment–Person–Comment–Person, rather than Comment–Person–Issue–Person as we observed. Within four degrees, we observed Frequent Collaborators of Creators or Assignees; a Frequent Collaborator of the Asker could also show up within four degrees (Comment–Person–Issue–Comment–Person, or Comment–Person–Comment–Issue–Person).

3.3.3 Five or More Degrees of Separation

In our sample set of 125 questions, there were no instances of five or more degrees of separation. Within this class, though, we hypothesize that more interesting variations of Frequent Collaborators and Frequent Contributors would arise; if one user does not know the answer, someone they know will. However, not having observed this class, we do not anticipate that it will be a commonplace sight.

3.3.4 Unconnected Users

The final pattern observed was one instance of an answer unconnected from the graph of
the project (Figure 3.9). At the time the question was asked, the user who will eventually answer the forager’s traceability question was not yet connected to the project by the relationships we chose to express as edges. (For example, in Appendix A, Figure A.4, some users are not in the network at the time of that comment). When that user finally did connect to the the network, they were four degrees of separation away (Comment–Person–Comment–Issue–Person), appearing as the assignee to an issue where the asker had commented. With a larger dataset, these not-yet-connected relationships would be interesting to assess. More edge and node types could enable a connection between that user and the question.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Q1</th>
<th>Med</th>
<th>Q3</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Degrees</td>
<td>4</td>
<td>29</td>
<td>192</td>
<td>442</td>
<td>1908</td>
</tr>
<tr>
<td>3 Degrees</td>
<td>5</td>
<td>271</td>
<td>510</td>
<td>1405</td>
<td>4192</td>
</tr>
<tr>
<td>4 Degrees</td>
<td>5</td>
<td>625</td>
<td>2288</td>
<td>3636</td>
<td>7542</td>
</tr>
<tr>
<td>5 Degrees</td>
<td>5</td>
<td>1949</td>
<td>3084</td>
<td>4597</td>
<td>7950</td>
</tr>
</tbody>
</table>

It appears, from these classes, that a substantial amount of traceability foraging takes place in four-or-less degrees of separation. Seeking to apply information foraging to traceability, one could simply present all nodes within three-to-four degrees of the question as a patch where the forager might seek to understand their question. This would satisfy our requirement of encoding frequently-traversed foraging paths into a patch. However, as shown in Table 3.3, these patches are extremely large. If we included all nodes within 4 degrees of separation, most patches would have over 2200 nodes. Appendix A, Figures
A.1, A.2, and A.3 may emphasize this further: thanks to a central user, foraging through these patches seems entirely impractical. With some notion of relatedness, however, these patches could be made smaller without losing relevant information.
In order to create smaller patches which still contain the described relationships, we turn to the foraging concept of scent. Following the example of WUFIS and PFIS, we define the foraging concept *Information Scent* as the “inferred relatedness of a cue to the prey, as measured by amount of activation from a spreading activation algorithm.” In WUFIS and PFIS, relatedness between two nodes was determined by a function of textual similarity and encoded into the weights of the edge connecting those two nodes. The application of spreading activation introduced a notion of proximity; each node’s activation is representative of its textual similarity and proximity to the starting point—the information need.

Our challenge is taking the work of WUFIS and PFIS and defining these concepts in our socio-technical context. Once we have a measure of relatedness in traceability foraging,
4. Creating Patches

we can create patches.

4.1 Spreading Activation over Requirements Socio-Technical Graphs

If a forager posts a question on an issue, the user who will provide the answer will likely require knowledge on that issue. From this postulate, we define our notion of “relatedness” between two nodes: the degree of a user’s relatedness to an artifact is proportional to their knowledge on the artifact. This serves as an analogue to PFIS’s textual similarity. The user’s knowledge on an artifact is encoded as the weight on the edge connecting them. Note that, to support our implementation of spreading activation, a lower weight represents a more powerful connection.

- **Comment to Issue, Question to Issue:** \textbf{weight} = 1. This is the only type of artifact-to-artifact edge we defined in our RSTGs. If a comment or question is posted to an issue, it is directly a part of the traceability history of that issue. Therefore, its connection to the issue should be extremely strong.

- **Creator to Issue, Assignee to Issue:** \textbf{weight} = 2. As discussed, the creator and assignee have a high degree of knowledge on a given issue. In relationships connecting question to answer, the assignee or creator frequently answered questions if the forager did not directly reference a user first (Figure 3.3: 55% of questions). However, an
issue can develop without the supervision of the creator or assignee. Therefore, the connection of this relationship is lower than that of an issue to a comment.

- *Comment to Referenced: weight = 2.* The user who wrote the comment has determined that the referenced user should have strong knowledge on the comment. Figure 3.3 shows that 31% of questions were answered by a referenced user; this relationship is therefore prioritized with a strong connection.

- *Comment to User: weight = 3.* While the connection of a comment to the issue is strong, the user is not guaranteed to have knowledge on the issue to the same degree as Creator or Assignee. Therefore, we set the user’s connection to the issue lower than that of a creator or assignee.

- *Question to User: weight = 4.* Questions, like comments, do not guarantee knowledge on the issue. If anything, a user asking a question is least-qualified of the users discussed to provide an answer. Therefore, we give this relationship the weakest connection.

With weight defining the relatedness between two given nodes, now a given node’s relatedness to the forager’s information need can be determined. The classes discussed in the previous section demonstrated that answers fell within four degrees of separation, and that answers were likely to come from Frequent Collaborators. These concepts can be encoded by adapting spreading activation.

As shown in Algorithm 1, our variant of spreading activation starts at the question-node.
4. Creating Patches

Algorithm 1 Spreading Activation over an RSTG

Spreading Activation

Input: Graph, Source Node
Output: Graph with Activated Nodes
1: \(\text{Decay} \leftrightarrow 0.1\)
2: \(\text{source.activation} \leftrightarrow 1\)
3: for node in Breadth-First Traversal from source do
4:   for preNode in Neighbors of node do
5:     if preNode has been traversed then
6:       weight \(\leftarrow\) edge(preNode, node)
7:       newActivation \(\leftarrow\) preNode.activation \(\ast\) (1 \(\ast\) weight \(\ast\) decay)
8:       node \(\leftrightarrow\) Activate(node, newActivation)
9:   end if
10: end for
11: end for

Co-Routine: Activate

Input: Node, New Activation
Output: Activated Node
1: \(\text{frequencyReward} \leftrightarrow 0.01\)
2: if node has activation then
3:   maxAct \(\leftarrow\) max(node.activation, newAct)
4:   minAct \(\leftarrow\) min(node.activation, newAct)
5:   node.activation \(\leftrightarrow\) maxAct + ((1 \(\ast\) maxAct) \(\ast\) minAct \(\ast\) frequencyReward)
6: else
7:   node.activation \(\leftrightarrow\) newActivation
8: end if
4. Creating Patches

The question’s activation is set to 1. Then, surrounding nodes are traversed (Algorithm 1, Line 3), and activation is spread from their predecessors (Algorithm 1, Lines 4-8). Spreading activation traditionally has each node firing to its successors; our predecessor variant exhibits greater decay while still producing useful networks. Our exact mechanism of spreading activation is shown in Algorithm 1, Line 7, and the Algorithm 1 Co-Routine.

Line 7 shows how exactly weight is factored in. A lower weight implies a higher degree of relatedness. If activation is a measure of relatedness, lower weights should result in higher activations. Therefore, the lower the weight, the smaller the effect of the decay, and the higher the resulting activation.

The co-routine is divided into two branches by a conditional structure (Algorithm 1, Co-Routine, Line 2). If a node has no activation yet, the activation is simply spread to the new node. However, if a node already has activation, which activation is considered? Following the example of PFIS2 [3], the higher activation value is spread. However, in order to incentivize frequency (in order to promote the Frequent Collaborator and Frequent Contributor patterns), if a node already has activation (i.e. the node has an existing relationship to the question), a percentage of that existing relationship is added to the new activation. Currently, this frequency reward is set to 1% (Algorithm 1, Co-Routine, Line 1); otherwise, highly-central figures gained disproportionately high activations, and therefore activations did not suitably diminish throughout the graph. Another method for reinforcing frequent collaboration could be a subject for future work.

Figure 4.1 provides an example of this variant of spreading activation being applied to the
4. Creating Patches

RSTG from Figure 3.2. Each of the relationships is assigned its weight, and the activation of Question 149792 is set to 1. Then, User 2881 and Issue 73650 are visited and activation from the question is spread. Comment 149790 and Question 149789 had activation spread from both User 2881 and Issue 73650; the higher activation was retained.

4.2 Delineating Patches within Spreading Activation RSTGs

With spreading activation encoding relatedness with the traceability forager’s information need, patches can be created. Recall, from WUFIS and PFIS, that activation is a measure of information scent. By grouping together nodes with high activation, a patch with nodes related by the relationships described previously will be created. To do this, though, a threshold of “high” activation must be defined.

Earlier, creating patches by simply enclosing all nodes within four degrees of separation was proposed, because all answers in our dataset fell within four degrees. We now consider those answers’ activations. Examining graphs of the classes discussed earlier, with spreading activation completed, reveals that a forager setting the cutoff at 0.72 would include 84% of results. Setting the cutoff at 0.45 would include all results. Examining earlier classes revealed that many Frequent Contributors and their related information artifacts fall above 0.56; many Frequent Collaborators fall around 0.50.
4. Creating Patches

4.3 Results and Analysis

4.3.1 Quantitative Evidence

Figure 4.2, and samples taken from that figure in Table 4.1, show us how big and how effective patches delineated by activation might be, similarly to Table 3.3. In both, we provide descriptive statistics that tell us how large our patches would be if we only included nodes above a certain activation, as well as how many questions would contain their corresponding answers in that patch. While we don’t see the inclusion of the answerer as necessary for an effective patch (as the patch should provide information other than a user who can answer questions), it does still provide a metric to show how relevant the patch might be.

The descriptive statistics in Figure 4.2 and Table 4.1 suggest that delineating patches by activation creates smaller patches than by degrees of separation (Table 3.3). Statistically comparing 4 Degrees and Activation $\geq 0.45$, we conclude that the two sets are non-identical ($t = 10.901$, p-value < 0.01). A similar test for Activation $\geq 0.45$ and $\geq 0.72$ reaches the same conclusion ($t = 9.6481$ p-value $\geq 0.01$). In other words, each cutoff has significantly smaller patches than the previous.

While patch sizes at cutoff Activation $\geq 0.45$ are still too big for timely foraging, patch sizes at $\geq 0.72$ are substantially more reasonable. That being said, $\geq 0.72$ patches frequently do not include the answer node for three or four degree of separation relation-
4. Creating Patches

Table 4.1: Patch Size With Cutoff (Samples from Figure 4.2)

<table>
<thead>
<tr>
<th>Cutoff</th>
<th>Min</th>
<th>Q1</th>
<th>Med</th>
<th>Q3</th>
<th>Max</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Degrees</td>
<td>5</td>
<td>625</td>
<td>2288</td>
<td>3636</td>
<td>7542</td>
<td>100%</td>
</tr>
<tr>
<td>A ≥ 0.45</td>
<td>5</td>
<td>273</td>
<td>649</td>
<td>1860</td>
<td>4834</td>
<td>100%</td>
</tr>
<tr>
<td>A ≥ 0.50</td>
<td>5</td>
<td>190</td>
<td>406</td>
<td>1399</td>
<td>3961</td>
<td>93%</td>
</tr>
<tr>
<td>A ≥ 0.56</td>
<td>4</td>
<td>44</td>
<td>165</td>
<td>384</td>
<td>3207</td>
<td>85%</td>
</tr>
<tr>
<td>A ≥ 0.72</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>24</td>
<td>84%</td>
</tr>
</tbody>
</table>

ships. However, within these patches, we believe that foragers would still find information relevant to their information need.

4.3.2 Two Practical Examples

Figure 4.1 serves as a practical example of both the mechanism of the algorithm and a tradeoff to consider. Figure 4.1 is a network with a cutoff set to A ≥ 0.56—a cutoff chosen for its inclusion of many Frequent Contributors. Indeed, it was a Frequent Contributor that answered User 2881’s question; User 6655 had commented twice on Issue 73650 already. While the question (‘‘Does it work?’’) was a pointed request, asking for a specific piece of information, the patch generated by the request yields not only the user who will answer the request, but also related traceability information—a history of work conducted towards that issue, leading to ‘‘Does it work?’’.

Figure 4.1 also demonstrates a limitation of the algorithm in its current state. Other implementations of spreading activation begin from one or more nodes; we could have started
the activation from the question and the asking user. We chose to include only the question node, as to avoid superfluous information from the asking user’s connections. In this case, though, had we included User 2881 as an initial node for activation, the algorithm would have assigned higher activations to the direct collaboration between User 6655 and User 2881 (User 2881–Question 149789–User 6655). This collaboration was key to the traceability history of Issue 73650.

Another typical example, seen in Figure 4.3, demonstrates another interesting property of our algorithm. This figure was generated using the Cytoscape [38] Network Analysis and Visualization tool, and was filtered to only show nodes with $A \geq 0.72$. (This method is how we actually conducted our analyses described throughout this work.) The complete network, before filtering, can be seen in Appendix A, Figure A.4. Intensity of color correlates to activation, with most intense color being $A=1$.

In this situation, the question posed was Comment 9396: “Did this make it into PropertiesFactory?”, and Person 700, Gytis Trikleris, answered. While Trikleris did indeed make it into the patch, both he and the author were the lowest-activation nodes in this patch. In other words, thanks to the weighting assigned to commenters and askers, people do not have as high priority as information artifacts do. This could have interesting implications, not all necessarily positive—perhaps the current implementation of the algorithm might not be leveraging social connections as much as it should, and a better weighting scheme should be devised. We explore this further in the next section.

The other interesting characteristic of this example is that the comments and people in
4. Creating Patches

the patch are all of the issue’s commenters and contributors, and nothing from other issues. After examining neighboring nodes, we determined that this was the correct behavior—adding more nodes would have provided irrelevant information. Our algorithm, in many cases, provides only the neighbors to an issue, as those provide the highest degree of relevance to the question.
Figure 4.1: Spreading Activation Applied to an RSTG (Without Frequency Bonus)
Figure 4.2: Patch Size With Cutoff
Figure 4.3: Graph Generated from Comment 9396, $A \geq 0.72$
Chapter 5

Future Work

We believe that the results presented in this paper are very promising. By applying this preliminary algorithm, we are able to create small patches that the developer can traverse. That being said, additions can be made and parameters can be further refined to optimize the performance of the algorithm in this environment. More importantly, we believe that applying the foundation of our design thinking and methods to new environments could create new socio-technical graphs where spreading activation can provide relevant and small patches like ours.
5. Future Work

5.1 New Data Types: Text

Our process can further be extended within the requirements traceability realm by incorporating new node and edge types. In an earlier version of this paper, when talking about unconnected users in Section 3.3.4, we originally said “at the time the question was asked, the user who will eventually answer the forager’s traceability question was not yet connected to the project”. It was only after further reflection that we added “...by the relationships we chose to express as edges” to the sentence. In other words, while a user might not have commented or created an issue or been referenced, and therefore is not connected in our current topology, we believe that that user could still be connected in some way. With the addition of new nodes and relationship types, like code and commits, edges between questions and answers, or semantic similarity, unconnected users might be connected, and new and more-insightful relationships might emerge, creating richer and deeper patches.

We’d like to briefly present our preliminary work in semantic similarity, in hopes that it may prove instructive for future work. We decided to extract keywords from bodies of text and to add them as nodes. We drew heavily on PFIS2’s approach, visible in Figure 5.1, as it had a similar objective of connecting otherwise-unrelated information artifacts with keywords. In order to achieve this scheme, though, we had to extract keywords from comment and issue bodies.

Extracting keywords in our initial explorations was attempted using two keyword extrac-
5. Future Work

Figure 5.1: Image From PFIS2 [3] Demonstrating Semantic Similarity Scheme

tion algorithms: TF-IDF [39] and RAKE [40]. Briefly, TF-IDF (or Term Frequency–Inverse Document Frequency) is a popular information retrieval metric, applied to a body of documents, which reflects how important a term is in a given document. The most important terms can be treated as keywords. RAKE attempts to extract keywords from a single document by extracting not-commonly-used words or phrases and ranking them relative to each other. Results we observed when implementing both algorithms were similar; we will therefore only mention results from RAKE.

Keywords were only ever part of a shortest path between a question and answer when Degree $\geq 3$—perhaps unsurprisingly, the simple “One or Two Degrees” cases remain untouched. In “Three or Four Degrees”, adding hundreds of keywords provided the algorithm more ways to form shortest paths, but rarely created shorter paths. The average path length before keywords was 2.944, and after was 2.928. Average activation before and after were 0.708 and 0.706, respectively. Min, Max, Median, and Quartiles remained
unchanged. Statistically, these differences are insignificant. This suggests that the keywords that we selected added more information, but not necessarily more insightful information. In fact, “more information” resulted in a problem: because we were inserting all keywords, networks at the time of question were around 10 times bigger with RAKE keywords, resulting not only in much slower scripting and visualization performance (we ran out of memory trying to generate graphs for this section), but also larger patches.

To examine these statistics further, we can look at the two cases where distance was actually shorter: Comment 1940 (from DROOLS) and Comment 9732 (from JBTM), as seen in Figure 5.2. Before adding keywords, both were four degrees of socio-technical separation apart. After keywords, they were three. However, in both cases, the shorter path provided through keywords went through generic (to this environment) words: “workaround” in the former case and “issue” in the latter. This suggests to us that these shorter paths existed by chance, rather than by actual similarity. If keywords are to be pursued in future work, they should be implemented with far greater discrimination than inserting all RAKE or TF-IDF keywords, and they should be studied most intensely in distant cases where otherwise they would be four or more degrees of socio-technical separation apart.
5. Future Work

5.2 Systematic Spreading Activation Parameters

Future work could also be conducted on the implementation of the algorithm itself. Our decay, frequency reward, weights, and patch cutoff parameters were set through observation and trial and error. More sophisticated statistical analyses, like machine learning algorithms, could help better set these parameters. Our method only suggested the first patch for foraging; in reality, a forager will go through several patches in search of their prey. To accommodate this pattern, this work could be extended to multiple-patch creation. Already, we have begun exploring ways to remove the need to manually determine weight for each type of relationship, and considered a possible approach to implement frequency incentives.

5.2.1 Weight

When establishing weights for our algorithm, our objective was to have activation flow towards people and information artifacts that presented knowledge on the initially-activated points. We achieved this by allowing activation to flow more freely towards users who likely had authority on a given comment or issue. This approach does require us, however, to come up with a new weight for each new relationship added. It would be beneficial, therefore, to have the same properties emerge from an automated process.

While there could be many approaches for future work in this direction, we would like to present one promising direction that arose in our research. With the objective of spread-
ing activation towards knowledge, and with the postulate that knowledge is typically centered around the issue where the comment is posted, we removed weights, and activated the issue in addition to the comment. We then spread activation (with a higher decay of 0.25 rather than 0.1). In other words, as a replacement to weighting schemes, we started the spreading of activation from comment and issue. Remarkably, this had similar results in some initially-explored metrics. Furthermore, if we only activated the issue or only increased decay, our results weren’t as successful. A brief summary of these results can be seen in Table 5.1. The graphs in Appendix A were actually generated during this test. The activation-cutoff graph was the same both in the Weighting scheme and the Weightless + Activated Issue scheme.

Table 5.1: Activating Comment and Issue Instead of Weighting Edges

<table>
<thead>
<tr>
<th></th>
<th>Median Patch Size A ≥ 0.45</th>
<th>Median Patch Size A ≥ 0.72</th>
<th>% Answers in Patch A ≥ 0.45</th>
<th>% Answers in Patch A ≥ 0.72</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighting Edges</td>
<td>286</td>
<td>5</td>
<td>100</td>
<td>84.8</td>
</tr>
<tr>
<td>Weightless + Activate Issue</td>
<td>204</td>
<td>6</td>
<td>96.8</td>
<td>84.8</td>
</tr>
<tr>
<td>Only Weightless</td>
<td>192</td>
<td>3</td>
<td>84.8</td>
<td>24.0</td>
</tr>
</tbody>
</table>

Table 5.1 shows that both weightless schemes have slightly smaller patches, which makes sense considering the more aggressive decay. Also, it shows that only with the issue activated does performance (as measured by answers in patches) come close to the weighted method explored earlier in this paper.
5. Future Work

5.2.2 Frequency Incentives

We think that incentivizing frequency is an important aspect of our algorithm; a person who comments on an issue multiple times should have higher activation than one with only one comment. However, our approach of giving a bonus to nodes that were frequently visited by our traversal was not extremely effective. It was hard to find a value that would provide that incentive while still allowing activation to decay.

One method we suspect might prove successful is counting number of shortest paths. Consider our three frequency examples, in Figures 3.6, 3.7, and 3.8. The shortest path between the answer node and the question node is three or four degrees, but there were multiple ways that this distance could be achieved. We see this as being the same as the frequency we want to incentivize. Of course, as distance increases, number of shortest paths increases, so any metric developed to incentivize frequency through number of shortest paths should be inversely proportional to distance, as well. Finally, in initial tests using NetworkX, counting number of shortest paths was extraordinarily slow, increasing runtime by a factor of 10 or more.

5.3 New Domains

Most importantly, this method could be extended to other socio-technical tasks. We believe that applying the foundation of our design thinking and methods to new domains
could create socio-technical graphs where spreading activation can provide relevant and small patches like ours. We see these domains as being anything from suggesting movies based on the cast and crew that link them together, to gathering evidence of work on requirements for safety certification.

Figure 5.3: A Composite Network with Two Sets of Nodes $E_1$ and $E_2$ Source: [4]

One particular domain that has caught our attention is that of composite networks. A composite network could be defined as a graph with one set of nodes and two or more distinct sets of edges, like in Figure 5.3. Wei et al. explored composite networks when using epidemic models to analyze how memes might spread over two social networks that share the same set of users (e.g. people who have both a Facebook and Twitter account) [4]. We think that similar network topologies would be very interesting to explore with spreading activation.

Immediately, multi-Jira-project analysis comes to mind. Users who participate in multiple projects, or shared code-bases across projects, would serve as our composite nodes; each project’s issues and comments would provide a distinct set of edges. Connections from one project would provide a set of deeper context to another.
Reaching further, spreading activation in composite networks could be applied to every-day Internet life. In a world where algorithms provide us only with news we want to read, and we follow friends and influencers who say only what we are already thinking, polarization is driving a wedge into modern-day society. We imagine an application of our spreading activation algorithm that builds patches of content relevant to an issue, not in an issue tracker, but in our society, by spreading activation over the people and news stories across multiple composite networks.
Chapter 6

Discussions

6.1 Implications

Piorkowski and his colleagues [8] codified the fundamental challenges faced by software developers when foraging in the information environment. We believe that our socio-technical approach can help directly address the challenge of “prey in pieces” where the foraging paths were too long and disconnected by different topologies. By explicitly integrating humans in the underlying topology, information foragers can exploit a richer set of relationships.

Codebook [24] confirmed that the small-world phenomenon [41] was readily observed in the socio-technical networks built from the software repositories, e.g., any pair of stakeholders would be connected in six-or-less hops. Our findings suggested that the
requirements-centric socio-technical graphs are even smaller with relevant nodes surrounded in four-or-less degrees of separation from the traceability forager’s question. Meanwhile, our results revealed several common relationships and their compositions. In light of the recent work on collecting practitioners’ natural-language requirements queries (e.g., [42–44]), the patterns uncovered by our study could be used to better classify and answer project stakeholders’ traceability needs.

Automated requirements traceability tools have been built predominantly by leveraging text retrieval methods [34]. These tools neglect an important factor—familiarity—which plays a crucial role in tracking the life of a requirement. Our work further points out that familiarity is multi-faceted: it could be the person familiar with the issue, the project, or the asker that is knowledgeable about the traceability information. Our results here are to be contrasted with the empirical work carried out by Dekhtyar et al. [45] showing that experience (e.g., years worked in software industry) had little impact on human analysts’ tracing performance. While a developer’s overall background may be broad, we feel that the specific knowledge about the subject software system and the latent relationships established with project stakeholders do play a role in requirements tracing. Automated ways of inferring a developer’s knowledge degree (e.g., [46]) would be valuable when incorporated in traceability tools.

Requirements traceability serves as a critical case [47] for our investigation into developers’ information foraging in a socio-technical environment. This is because, without the traceability information, many software engineering activities cannot be undertaken,
6. Discussions

such as verification that a design satisfies the requirements, validation that requirements have been implemented, change impact analysis, system level test coverage analysis, and regression test selection [48]. However, requirements traceability is by no means the only case. In fact, due to information foraging theory’s parsimony, its constructs (e.g., patch and scent) have been adapted to support a variety of tasks including debugging, refactoring, and software reuse [49–51]. The socio-technical patch created by our spreading activation algorithm, therefore, could help developers answer their needs beyond requirements traceability.

6.2 Conclusion

By considering common relationships connecting a requirements traceability question to its answer, and encoding these relationships into a spreading activation algorithm, we were able to delineate patches for use in understanding context surrounding requirements traceability questions in a socio-technical environment. In this process, we found that traceability questions were answered by users within four degrees of socio-technical separation; these users were typically Frequent Collaborators of the traceability forager or of the creator/assignee of the issue, or Frequent Contributors to the issue. Encoding these relationships as parameters to a spreading activation algorithm resulted in patches of nodes that a traceability forager could traverse, searching for their answer. While simply creating patches including all nodes within four degrees of socio-technical separa-
tion would include all answers, the addition of spreading activation created significantly smaller patches. We believe that this work can serve as a foundation for future work in creating patches in socio-technical networks
Bibliography


[37] “Github repository,” https://github.uc.edu/cepulide/RSTG-SA.


Appendix A

Graphs
Figure A.1: Patch Delineated by 2 Degrees of Socio-Technical Separation
Figure A.2: Patch Delineated by 3 Degrees of Socio-Technical Separation
Figure A.3: Patch Delineated by 4 Degrees of Socio-Technical Separation
Figure A.4: Example Cytoscape Visualization of Project With Spreading Activation Applied. Darker Nodes Have Higher Activation
Appendix B

Code

# Darius Cepulis
# cepulisde@mail.uc.edu
# dc.cepulis@gmail.com
#
# RSTG-SA.py
#
# Built and Tested with Anaconda for Python 3.6
# https://www.anaconda.com/download/
#
# Might still work with Python 2.7
# from __future__ import print_function
#
# Utility IO Functions
# functions for reading and writing pickles
#
# MySQL Database Functions
# class with functions for connecting to and reading MySQL
#
# Graph Manipulation Functions
# class with functions for creating, reading, and writing networkx graphs
#
# Main
# procedure for reading CSVs with questions and answers
# producing graphs, and analyzing the distance between questions and answers
#
# Significant caching is utilized in this script
# Delete cache to rebuild stuff
# View cache in ./cache
# Operations that require querying the MySQL database are the slowest

import mysql.connector # https://dev.mysql.com/downloads/connector/python/
# conda install --c anaconda mysql-connector-python
import pickle
import os, os.path
import errno
import networkx as nx
import re
import glob
import csv
from datetime import datetime
from collections import defaultdict
from numpy import percentile
import RAKE # pip install python-rake

#-------------------------------------------------------------------------------
# Utility IO Functions #
#-------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------
def saveObject(obj, fname):
    with safeOpenWrite('cache/ ' + fname + '.pkl') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

def loadObject(fname):
    try:
        with open('cache/ ' + fname + '.pkl', 'rb') as f:
            return pickle.load(f)
    except FileNotFoundError:
        raise

def mkdirPath(path):
    try:
        os.makedirs(path)
    except OSError as exc:
        if exc.errno == errno.EEXIST and os.path.isdir(path):
            pass
        else:
            raise

def safeOpenWrite(path):
    mkdirPath(os.path.dirname(path))
    return open(path, 'wb')

def isValidXMLChar(i):
    # XML standard defines a valid char as:
    # Char ::= #x9 | #xA | #xD | [#x20−#xD7FF] | [#xE000−#xFFFD] | [#x10000−#x10FFFF]
    return (0x20 <= i <= 0xD7FF
            or i in (0x9, 0xA, 0xD)
            or 0xE000 <= i <= 0xFFFD
            or 0x10000 <= i <= 0x10FFFF)

def cleanXMLString(s):
    return ''.join((c for c in s if isValidXMLChar(ord(c))))

class dbManager():
    # input: login = { 'username': 'user', 'password': 'pwd', 'host': 'host', ['database': 'db', ...]}
    # output: usersByKey = { username: id }
    # usersByDisplay = { full name: id }
    # usersByID = { id: { 'key': key, 'display': display } }
    # usersByReference = { project #: { Name ': id, 'Name ': id } }
    def __init__(self, login):
        self.rake = RAKE.Rake(RAKE.SmartStopList())
        self.cnx = mysql.connector.connect(**login)
    # input: connection from dbConnection()
    # output: usersByKey = { username: id }
    # usersByDisplay = { full name: id }
    # usersByID = { id: { 'key': key, 'display': display } }
    # usersByReference = { project #: { Name ': id, 'Name ': id } }
    def loadUsersFromCache(self):
        try:
            self.cnx = mysql.connector.connect(**login)
        except mysql.connector.errors.InterfaceError as exc:
            print('Connection with database timed out. Attempting to run with cached data...')
            print(exc)
            self.cnx = None
        self.loadUsersFromCache()
usersByReference = loadObject('usersByReference')

self.usersByKey = usersByKey
self.usersByDisplay = usersByDisplay
self.usersByID = usersByID
self.usersByReference = usersByReference

except FileNotFoundError:
    print('Loading from cache failed. Attempting to rebuild from database...')
    self.loadUsersFromDB()

# input: connection from dbConnection()

# output: usersByKey = {username: id}
# usersByDisplay = {fullName: id}
# usersByID = {id: {key, display}}
# usersByReference = {project #: {'Name,': id, 'Name,': id}}
def loadUsersFromDB(self):
    print('Loading users from database...')

    query = "SELECT * FROM jira_user"
    cursor = self.cnx.cursor()
    cursor.execute(query)

    usersByKey = {}
    usersByDisplay = {}
    usersByID = {}

    for (id, org, jira_key, name, display, email_address, url, timezone, locale) in cursor:
        usersByKey[jira_key] = id
        usersByKey[name] = id
        usersByDisplay[display] = id
        usersByID[id] = {'key': jira_key, 'display': display}

    cursor.close()

    # usersByKey['nmcl']
    # usersByKey['stephen.fikes']
    # usersByKey['christa']
    # usersByKey['halli']
    # usersByKey['anbaker']
    usersByKey['roger.martinez'] = 17
    usersByKey['dzonca'] = 61
    # usersByKey['croatkins']
    # usersByKey['ganandar.Redhat']
    usersByKey['mario.fusco'] = 34
    usersByKey['andrew.collins'] = 355
    usersByKey['psiroky@redhat.com'] = 27

    usersByDisplay['Geoffrey De Smet'] = 48
    usersByDisplay['Youcef Ellemen'] = 299
    usersByDisplay['Daveide Scottara'] = 291
    usersByDisplay['Liza Claytton'] = 7
    usersByDisplay['Roger Martinez'] = 17

    usersByReference = {
        3: {}
        # 'Amos,': 683,
        # 'Ivo,': 1192,
        # 'Mike,': 1634,
        # 'Paul,': 45,
        # 'Tom,': 28,
        # 'Mark,': 133
    }

    2: {}
    # 'Edson,' : 2,
    # 'Yousef': 299,
    # 'Geoffrey': 48,
    # 'Daveide': 4149,
    # 'Chen,' : 4146

    4: {}
    # 'Chas,': 6286,
    # 'Mariano,': 6684,
    # 'Paul,': 6690,
    # 'Jim,': 113

    saveObject(usersByKey, 'usersByKey')
    saveObject(usersByDisplay, 'usersByDisplay')
    saveObject(usersByID, 'usersByID')
    saveObject(usersByReference, 'usersByReference')

self.usersByKey = usersByKey
self.usersByDisplay = usersByDisplay
```python
self.usersByID = usersByID
self.usersByReference = usersByReference

# input and output:
# * getUser(key = <user key>)
#   returns <user ID> as int
# * getUser(display = <user name>)
#   returns <user ID> as int
# * getUser(id = <user id>, ret = key or display)
#   returns <user key> or <user display> as string
# * getUser(reference = <user reference>, e.g. 'Paul, ', project = <project number>)
#   returns <user ID> as int

def getUser(self, **kwargs):
    try:
        if 'key' in kwargs:
            return self.usersByKey[kwargs['key']]  # lấy ID người dùng bằng key
        elif 'display' in kwargs:
            return self.usersByDisplay[kwargs['display']]  # lấy ID người dùng bằng display
        elif 'id' in kwargs:
            try:
                return self.usersByID[kwargs['id']][kwargs['ret']]
            except KeyError:
                raise
        elif 'reference' in kwargs:
            try:
                return self.usersByReference[kwargs['key']][kwargs['reference']]
            except KeyError:
                raise
    except KeyError:
        raise

# input: project = int representing project number in database
# graph = graph to populate with project issues and their related people

def addProjectIssuesToGraph(self, project, graph):
    print('[Populating graph with issues from project...]', project)
    query = 'SELECT jira, issue FROM jira_issue WHERE project=\(\)'.format(project)
    cursor = self.cnx.cursor()
    cursor.execute(query)
    for id, jira_id, jira_key, url, project, issue_type, priority, assignee, description,
        summary, timestamp, creator, watches, votes, has_questions, has_needs in cursor:
        issue = "issue": str(id)

        graph.addNode(issue,  
            type="issue",  
            summary=summary,  
            description=description,  
            timestamp=str(timestamp),  
            key=jira_key
        )
        
        # Connect issue to keywords
        keywords = (word[0] for word in self.rake.run('{}'}  
            format(summary, description)) if word[1]>=1.0
        for word in keywords:
            graph.addNode(word, type="keyword")

        graph.addEdgeOrUpdateWeight(issue, word, 'keyword'

        
        # Connect issue to people
        if (assignee != None):
            assignee_label = "Person": str(assignee)
            try:
                graph.addNode(assignee_label,  
                    type="person",  
                    display=self.getUser(id=assignee, ret='display')
                )
                graph.addEdgeOrUpdateWeight(issue, assignee_label, 'assignee')
            except KeyError:
                if vErr: print("User\(\) not found", format(assignee))

        if (creator != None):
            creator_label = "Person": str(creator)
            try:
                graph.addNode(creator_label,  
                    type="person",  
                    display=self.getUser(id=creator, ret='display')
                )
                graph.addEdgeOrUpdateWeight(issue, creator_label, 'creator')
            except KeyError:
                if vErr: print("User\(\) not found", format(creator))

    cursor.close()

# input: project = int representing project number in database
```
The body of the comment or question

The full name of the commenting user, as a string

The code block contains functions and methods for interacting with a database, specifically for adding project comments and issues to a graph. It includes SQL queries to retrieve data and methods to add nodes and edges to a graph based on the retrieved data. The comments and issues are associated with nodes, and keywords are connected to comments.

Here are some key parts of the code:

- Functions to add comments, issues, and keywords to a graph.
- SQL queries to select and format data from a database table.
- Methods to add nodes to the graph with labels and types.
- Handling of exceptions and key errors.

The code is written in Python and uses database operations to extract and manipulate data.
def findUserComment(self, userDisplay, comment):
    # we begin by replacing spaces with wildcards because
    # some spaces from sample data are actually new lines in the db
    comment = comment[:40].replace('"', '%').replace('\\', '\')
    try:
        userid = self.getUser(display=userDisplay)
    except KeyError:
        if vErr:
            print('User:' + userDisplay + 'not_found.')
        return []
    query = 'SELECT * FROM (SELECT * FROM jira_issue_comment WHERE author=?)
    AS q1 WHERE q1.body LIKE "{"%","" format(userid, comment)
    cursor = self.cnx.cursor()
    comments.append(
        "comment": "Comment" + str(id),
        "issue": "Issue" + str(issue))
    cursor.close()
    return comments

def addEdgeOrUpdateWeight(self, a, b, typ):
    w = self.getWeight(typ)
    if (a, b) in self.G:
        attr = self.G[a][b]['weight']
        new_weight = max(old_weight, w)
        attr.update({'weight': new_weight})
    else:
        self.G.add_edge(a, b, weight=w)
        for attr in:
            self.G.add_edge(a, b, **attr)

# ####################################################################################
### Graph Manipulation Functions ####################################################
# ####################################################################################

class graph():
    # Initialize graph() with a project and database to attempt to load from cache
    # input: project = [optional] project number to load
    # db = [optional] database to load from if project not in cache
    def __init__(self, **kwargs):
        self.weights = {'assignee':2, 'creator':2, 'asked':4, 'posted':3,
        'referenced_in':2, 'question':1, 'comment':1, 'keyword':4}
        self.timeformat = '%Y-%m-%d %H:%M:%S'
        if 'project' in kwargs and 'db' in kwargs:
            self.loadProject(kwargs['project'], kwargs['db'])
        elif 'graph' in kwargs:
            self.G = kwargs['graph'].copy()
        else:
            self.G = nx.Graph()
    # input: project = project number to load from cache or db
    # db = db from which to load project if cache fails
    def loadProject(self, project, db):
        try:
            print('Loading project ' + project + ' from cache... format(project))
            self.G = loadObject('{}', graph='format(project))
            print('Loading Complete!')
        except IOError:
            print('Project ' + project + ' not in cache. Attempting to reload... format(project))
            self.G = nx.Graph()}{db.addProjectToGraph(project, self)
            saveObject(self.G, '{}', graph='format(project))
    # input: typ = type of edge
    # output: weight of edge
def getWeight(self, typ):
        try:
            return self.weights[typ]
        except KeyError:
            raise
    # input: G = graph to search
    # a = node 1
    # b = node 2
    # typ = type of edge to add
    def.addEdgeOrUpdateWeight(self, a, b, typ):
        w = self.getWeight(typ)
        attr = {'type': typ}
        if (a, b) in self.G:
            old_weight = self.G[a][b]['weight']
            new_weight = max(old_weight, w)
            attr.update({'weight': new_weight})
        else:
            attr.update({'weight': w})
        self.G.add_edge(a, b, **attr)
# input: node — unique node name in graph
# attr_dict — dictionary of attributes to give to the node
def addNode(self, node, *attr):
    self.G.add_node(node, **attr)

# input: timestamp — datetime object
# e.g. datetime.strptime(G.node[question][‘timestamp’],
# self.timeformat)
# output: subgraph before timestamp
def newSubgraphBeforeTimestamp(self, timestamp):
    g = self.G.copy()
    sgFilter = []
    for n in g.nodes(data=‘timestamp’):
        if n[1] == None: # if there is no timestamp in the node it’s a person. Add.
            sgFilter.append(n[0])
        else:
            tNode = datetime.strptime(n[1], self.timeformat)
            if tNode <= timestamp:
                sgFilter.append(n[0])
    return g.subgraph(sgFilter)

# output: copy of current graph
def newGraphCopy(self):
    return self.G.copy()

# Traverses nodes, spreads activation from predecessors
# input: questionNode — name of node from which to spread (e.g. ‘Comment 123’)  
# issueNode — name of issue node for question node, also activated to 1
# spaceDecay — rate, from 0–1, at which activation decays relative to space
# 0.1 recommended
# T r a v e r s e s n o d e s , s p r e a d s a c t i v a t i o n f r o m p r e d e c e s s o r s
# i n p u t : q u e s t i o n N o d e — n a m e o f n o d e f r o m w h i c h t o s p r e a d ( e . g . ‘ C o m m e n t 1 2 3 ’ )
# i s s u e N o d e — n a m e o f i s s u e n o d e f o r q u e s t i o n n o d e , a l s o a c t i v a t e d t o 1
# s p a c e D e c a y — r a t e , f r o m 0–1, a t w h i c h a c t i v a t i o n d e c a y s r e l a t i v e t o s p a c e
# 0.1 recommended

def spreadActivationSimple(self, questionNode, issueNode, spaceDecay):
    try:
        self.G.node[questionNode][‘activation’] = 1.0
        self.G.node[issueNode][‘qa’] = ‘question’
        if issueNode:
            self.G.node[issueNode][‘activation’] = 1.0
            self.G.node[issueNode][‘qa’] = ‘issue’
        bft = nx.bfs_edges(self.G, questionNode)
        for edge in bft:
            preNode = edge[0]
            for postNode in self.G.neighbors(preNode):
                if ‘activation’ in self.G.node[preNode]:
                    weight = self.G[preNode][postNode][‘weight’]
                    calcActivation = (self.G.node[preNode][‘activation’] * (1 – (weight * spaceDecay)))
                    if ‘activation’ in self.G.node[postNode]:
                        currentActivation = self.G.node[postNode][‘activation’]
                        newActivation = max(calcActivation, currentActivation)
                        self.G.node[postNode][‘activation’] = newActivation
                    else:
                        self.G.node[postNode][‘activation’] = calcActivation
    except KeyError:
        raise

# Traverses nodes, spreads activation from predecessors
# input: questionNode — name of node from which to spread (e.g. ‘Comment 123’)  
# issueNode — name of issue node for question node, also activated to 1
# spaceDecay — rate, from 0–1, at which activation decays relative to space
# 0.1 recommended

def spreadActivationComplete(self, questionNode, issueNode, spaceDecay):
    try:
        self.G.node[questionNode][‘activation’] = 1.0
        self.G.node[questionNode][‘qa’] = ‘question’
        if issueNode:
            self.G.node[issueNode][‘activation’] = 1.0
            self.G.node[issueNode][‘qa’] = ‘issue’
        bft = nx.bfs_edges(self.G, questionNode)
        i = 0
        for edge in bft:
            preNode = edge[0]
            for postNode in self.G.neighbors(preNode):
                if ‘activation’ in self.G.node[preNode]:
                    weight = self.G[preNode][postNode][‘weight’]
                    calcActivation = (self.G.node[preNode][‘activation’] * (1 – (weight * spaceDecay)))
                    if ‘activation’ in self.G.node[postNode]:
                        currentActivation = self.G.node[postNode][‘activation’]
                        newActivation = max(calcActivation, currentActivation)
                        self.G.node[postNode][‘activation’] = newActivation
                    else:
                        self.G.node[postNode][‘activation’] = calcActivation
self.G.node[postNode]["activation"] = calcActivation
except KeyError:
    raise

# Finds shortest path between two nodes
# with option to analyze by weight
# input: sourceNode - source node
# destinationNode - destination node (duh)
# weight - shortest path, including weight? True/False
def findPath(self, sourceNode, destinationNode, weight=False):
    try:
        if weight:
            return nx.shortest_path(self.G, sourceNode, destinationNode,
                                     weight="weight")
        else:
            return nx.shortest_path(self.G, sourceNode, destinationNode)
    except nx.exception.NetworkXNoPath:
        raise

# Finds shortest path between two nodes
# with option to analyze by weight
# input: sourceNode - source node
# destinationNode - destination node (duh)
# weight - shortest path, including weight? True/False
def findPaths(self, sourceNode, destinationNode, weight=False):
    try:
        if weight:
            return nx.all_shortest_paths(self.G, sourceNode, destinationNode,
                                         weight="weight")
        else:
            return nx.all_shortest_paths(self.G, sourceNode, destinationNode)
    except nx.exception.NetworkXNoPath:
        raise

# input: node - node whose activation to inspect
# output: activation of node
def getNodeActivation(self, node):
    return self.G.node[node]["activation"]

# output: the number of nodes over input activationCutoff
def numNodesAboveActivation(self, activationCutoff):
    return sum([1 for n in self.G.nodes(data="activation")
                if (n[1] != None and n[1] >= activationCutoff)])

# output: the number of nodes within radius of sourceNode
# input: radius - number of hops from sourceNode within which to count
# sourceNode - node from which to count
def numNodesInRadius(self, radius, sourceNode):
    return nx.number_of_nodes(nx.ego_graph(self.G, sourceNode, radius=radius))

# input: node to search for
# output: bool
def nodeInGraph(self, node):
    return node in self.G.nodes

# input: node - node whose attribute to set
# attribute - attribute to set
# value - value to set attribute to
def setAttribute(self, node, attribute, value):
    self.G.node[node][attribute] = value

# input: node before which to create graph
# output: subgraph before node
def newSubgraphBeforeNode(self, node):
    iQuestion = datetime.strptime(self.G.node[node]["timestamp"], self.timeformat)
    return self.newSubgraphBeforeTimestamp(iQuestion)

# input: filename - filename of saved graph
# output: GraphML
def saveAsGraphML(self, filename):
    # clean structure
    # for node in self.G.nodes:
        # for attrib in self.G.nodes[node]:
            # if type(self.G.node[node][attrib]) == str:
                # text = self.G.node[node][attrib]
                # clean = re.sub(u'\[u0020--u0131\]u000A\u000D\uE000--\uffFD\u00000000--\U0000FFFF]+', '', text)
            # self.G.node[node][attrib] = clean
        # if self.G.node[node][attrib] == None:
            # self.G.node[node][attrib] = "None"
        # else:
        #     mkdirPath('/graphs/')
        #     nx.write_graphml(self.G, '/graphs/'+filename+'-graphml')
    mx = findMX(filename)
    self.G = nx.read_graphml(filename)
    iQuestion = datetime.strptime(self.G.node[node]["timestamp"], self.timeformat)
    iQuestion = iQuestion.replace(minute=0, second=0, microsecond=0)
    nx.write_graphml(self.G, filename, 'None')

    # input: node - node whose attribute to set
    # attribute - attribute to set
    # value - value to set attribute to
    def setAttribute(self, node, attribute, value):
        self.G.node[node][attribute] = value

    # input: node - node whose attribute to set
    # attribute - attribute to set
    # value - value to set attribute to
    # with option to analyze by weight
    def findPath(self, sourceNode, destinationNode, weight=False):
658  # clean structure:
659  for node in self.G.node:
660      for attrib in self.G.node[node]:
661          if type(self.G.node[node][attrib]) == str:
662              text = self.G.node[node][attrib]
663              clean = re.sub('"[\u0020=\u00A0\u00D0\u00FF\u0009\u000A\u000D\u0080\u00A0\u00FF\u0009\u000A\u000D\u0080\u0084\u0085\u0086\u0087\u008E\u008F\u0091\u0092\u0093\u0094\u0095\u0096\u0097\u009C\u009D\u009E\u00A0\u00A1\u00A2\u00A3\u00A4\u00A5\u00A6\u00A7\u00A8\u00A9\u00AA\u00AB\u00AC\u00AD\u00AE\u00AF\u00B0\u00B1\u00B2\u00B3\u00B4\u00B5\u00B6\u00B7\u00B8\u00B9\u00BA\u00BB\u00BC\u00BD\u00BE\u00BF\u00C0\u00C1\u00C2\u00C3\u00C4\u00C5\u00C6\u00C7\u00C8\u00C9\u00CA\u00CB\u00CC\u00CD\u00CE\u00CF\u00D0\u00D1\u00D2\u00D3\u00D4\u00D5\u00D6\u00D7\u00D8\u00D9\u00DA\u00DB\u00DC\u00DD\u00DE\u00DF\u00E0\u00E1\u00E2\u00E3\u00E4\u00E5\u00E6\u00E7\u00E8\u00E9\u00EA\u00EB\u00EC\u00ED\u00EE\u00EF\u00F0\u00F1\u00F2\u00F3\u00F4\u00F5\u00F6\u00F7\u00F8\u00F9\u00FA\u00FB\u00FC\u00FD\u00FE\u00FF]+', ' ', text)
664  self.G.node[node][attrib] = clean
665  elif self.G.node[node][attrib] == None:
666      # write file
667      mkdirPath('./graphs/')
668      nx.write_gml(self.G, './graphs/' + filename + '.gml')
669  ...
672
673  # == SETUP ==
674  vErr = True  # Verbose Error Output
675
676  login = {'user': 'root', 'password': 'foraging',
677            'host': '127.0.0.1', 'database': 'network'}
678  db = dbManager(login)
679
680  try:
681      questionDB = loadObject('questionDB')
682  except FileNotFoundError:
683      print('QuestionDB not cached. Rebuilding...')
684  except AttributeError:
685      print('Cannot proceed without server connection...')
686  except IOError:
687      print('Can not proceed without server connection...')
688  g = graph(project = projects[os.path.basename(f)[:-4]],
689             db = dbManager(login),
690             login = login)
691  try:
692      project = projects[os.path.basename(f)]
693      g = graph(project = project, db = db)
694      except AttributeError:
695          exit('Can not proceed without server connection...')
696  g.saveGML(str(project))
697  ...
698  for f in glob.glob(os.getcwd() + '/' + answers + '/' + '*.csv'):
699      try:
700          questionDB = loadObject('questionDB')
701          answerCSV = 'answers/' + os.path.basename(f)
702          print(answerCSV)
703          with open(answerCSV, encoding='utf-8') as f:
704              reader = csv.reader(f)
705              for row in reader:
706                  # Parse Row
707                  body = row[0].strip()
708                  asker = row[1].strip()
709                  answered = row[2].strip()
710                  answerer = row[3].strip()
711                  if not answered:
712                      # if vErr: print('Question not answered.')
713                      continue  # onto the next row in the reader
714                  try:
715                      answer = 'Person' + str(db.getUser(display=answerer))
716                  except KeyError:
717                      if vErr: print('Unknown user ID, format(answerer)')
718                      continue  # onto the next row in the reader
719                  # Filter by Question
720                  try:
721                      question = questionDB[body]['question']
722                      issue = questionDB[body]['issue']
723                  except KeyError:
724                      questions = db.findUserComment(asker, body)
725                      question = None
726                      issue = None
727                      # for all the user's comments that have the same body we pick
728                      # the first one in our graph
729                      if g.nodeInGraph(qi['comment']) and in nodeInGraph(qi['issue']):
```python
question = qu["comment"]
issue = qu["issue"]

if not question:  # and if we didn’t find a question…
    if vErr: print('Question not found in graph.')
    continue  # onto the next row in the reader

questionDB[body] = {
    "comment": question,
    "issue": issue
}

try:
    q = loadObject(question)
except FileNotFoundError:
    # we filter our graph by time
    q = graph(graph = g.newSubgraphBeforeNode(question))

# — Find Answer —

if not q.nodeInGraph(answer):
    if vErr: print('Answerer {} is not in network.'.format(answerer))
    continue

q.setAttribute(answer, 'qa', 'answer')
saveObject(q, question)

# — Spread and Analyze —

q.spreadActivationSimple(question, issue, 0.25)  # Weightless
q.spreadActivationSimple(question, None, 0.1)
q.saveAsGML(question)

try:
    paths = list(q.findPaths(question, answer))
    answerActivation = q.getNodeActivation(answer)
    distance = len(paths[0])
    print("Activation: {}, Paths: {}, Distance: {}, Sample: {}.
format(answerActivation))
    data["activations"].append(answerActivation)
    data["distances"].append(distance)

    for i in range(1, 100):
        iActivation = 0.01 * i
        data["nodesAboveActivation"].append(
            q.numNodesAboveActivation(iActivation))
    data["answerAboveActivation"].append(
        int(answerActivation >= iActivation))

    for i in range(1, 6):
        data["nodesInRadius"].append(q.numNodesInRadius(i, question))
        data["answerInRadius"].append(int(distance <= i))

except nx.exception.NetworkXNoPath:
    print("Path Not Found for {} > {}.
format(question, answer))

q.saveAsGraphML(question)

# — Summarize Statistics —

for k in data["nodesInRadius"]:  # len(data["nodesInRadius"])
    boxGraph = percentile(data["nodesInRadius"], [0, 25, 50, 75, 100])
    plot = defaultdict(list)
    for k in data["nodesAboveActivation"]:  # len(data["nodesAboveActivation"])  
        boxGraph = percentile(data["nodesAboveActivation"], [0, 25, 50, 75, 100])
        plot[0].append(boxGraph[0])
        plot[25].append(boxGraph[1])
        plot[50].append(boxGraph[2])
        plot[75].append(boxGraph[3])
        plot[100].append(boxGraph[4])

    saveObject(questionDB, "questionDB")
```