GSWC 2014

Proceedings of

The 9th Annual Graduate Student Workshop on Computing

October 10th 2014
Santa Barbara, California

http://gswc.cs.ucsb.edu

Department of Computer Science
University of California, Santa Barbara

Organized By

Alexander Pucher, Chair

Vaibhav Arora, Vice-chair

Nevena Golubović, Industry Liaison

Xiaofei Du, Industry Liaison

Cetin Sahin, Financial Coordinator

Hiranya Jayathilaka, Proceedings Coordinator

Stratos Dimopoulos, Website Coordinator

<u>Program Committee</u>

Kevin Borgolte

Victor Fragoso

Yanick Fratantonio

Madhukar Kedlaya

Ana Nika

Divya Sambasivan

Saiph Savage

Paul Schmitt

Morgan Vigil

Platinum Partners

Silver Partners

Bronze Partners

Keynote Speech

Title

Google Cloud Platform, IoT, and Beyond

Abstract

Ok, so you've got a few billion sensors, a few trillion rows, and no matter how you try, it just doesn't seem to fit in your spreadsheet. Google is doing this today. Join us to take a look under the hood at the new tools, methods, and results for folks pushing the state of the art in IoT in the cloud.

About the Speaker

Miles Ward is a three-time technology startup entrepreneur with a decade of experience building cloud infrastructures. Miles is Global Head of Solutions for the Google Cloud Platform; focused on delivering next-generation solutions to challenges in big data and analytics, multi-tiered storage, high-availability, and cost optimization. He worked as a core part of the Obama for America 2012 “tech” team, crashed Twitter a few times, helped NASA stream the Curiosity Mars Rover landing, and plays electric sousaphone in a funk band.

Panel Discussion

Title

Building an Internet of Things for Humans

Abstract

The Internet of Things (IoT) is expected to connect 50 billion devices to the Internet by 2020! Sensors and actuators will be everywhere including streets, farmlands, enterprises, households, cars, and even on our bodies. In this panel discussion we will try to understand what IoT is, how it affects the software industry, and the skillset that new computer science graduates should possess to cope with it. We will also discuss the challenges, opportunities and risks IoT introduces, along with the new areas of research that it opens up.

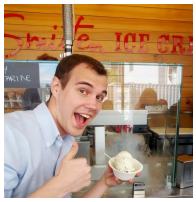
Panelists

Luca Foschini is a co-founder and data scientist at The Activity Exchange, a platform that uses data analytics on people's activities to incentivize healthy behavior. Luca earned a Ph.D. in Computer Science from UC Santa Barbara where he developed efficient algorithms for routing in road networks under heavy traffic conditions. He also holds a Master in Engineering from the University of Pisa, and is an alumnus of the Sant'Anna School of Advanced Studies. In a previous life, Luca accumulated 5 years of industry experience at Ask.com, Google, and the CERN, and was a coach of the Italian national team participating in the International Olympiad in Informatics (IOI).

Hagen Green is a Program Manager Lead at Microsoft on the Windows Shell Experiences team. Previously, he was a Test Lead in Office on SharePoint. Hagen holds several patents, written several technical articles, contributed to a book, and authored a book. In his spare time, Hagen enjoys the great Northwest by cycling, running, skiing, and backpacking. When he's not outside, he's reading, tinkering with technology, or spending time with his wife Jaime and their 15 month old son Nolan. Hagen holds a B.S. in Computer Science from UC Santa Barbara.

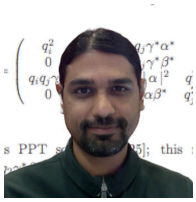
Christopher Kruegel is the co-founder of Lastline, Inc., where he currently serves as the Chief Scientist. He is also a Professor in the Computer Science Department at the UC Santa Barbara. His research interests are computer and communications security, with an emphasis on malware analysis and detection, web security, and intrusion detection. Christopher enjoys to build systems and to make security tools available to the public. He has published more than 100 conference

and journal papers. Christopher is a recent recipient of the NSF CAREER Award, the MIT Technology Review TR35 Award for young innovators, an IBM Faculty Award and several best paper awards. Moreover, he served as an associate editor for several journals and on program committees of leading computer security conferences.



Charles Munger is an alumnus of UCSB, having completed the 5 year BS/MS program in 2013. He is currently a software engineer for Google, working on frameworks and analysis/build tools for first party android apps.

Andrew Mutz is Chief Scientist at Appfolio, a Santa Barbara company focused on creating easy-to-use, web-based software that helps small and mid-sized businesses more effectively market, manage and grow their business. Before joining Appfolio, Andrew completed his PhD in Computer Science at UC Santa Barbara. Prior to his PhD, Andrew worked as a Software Engineer at Expertcity (now Citrix Online).



Ashish Thapliyal is a Principal Research Engineer at Citrix. He is interested in applying Summarization, Text Analytics and Machine learning to the real world. Previously, he was VP of Engineering at Lastline, and he has almost a decade of experience designing security for key Citrix Collaboration products. Before that he was a Postdoctoral Researcher at the Computer Science Department at UC Berkeley, and a Student Researcher at IBM T.J. Watson Research Center. His academic research in Quantum Entanglement and Information Theory produced more than 10 peer-reviewed publications. He has two patents and 10+ applications pending. Ashish has an MS in Computer Science, and a PhD in Physics from UCSB.

Peerapol Tinnakornsrisuphap is the systems engineering lead for Connected Home R&D in Qualcomm Research. His team has addressed many critical issues facing Internet of Things including low power protocol optimization, security and provisioning, multi-hop and mesh networking, proximal services discovery, and smart energy management. He received Ph.D. in Electrical Engineering from University of Maryland and holds 34 US Patents.

Table of Contents

GNSS Positioning Improvement and 3D Mapping Using Crowdsourced Satellite SNR Measurements

Andrew T. Irish, Daniel Iland, Jason T. Isaacs, Elizabeth M. Belding, João P. Hespanha, and Upamanyu Madhow

***Abstract*—Geopositioning using Global Navigation Satellite Systems (GNSS), such as the Global Positioning System (GPS), is inaccurate in urban environments due to frequent non-line-of-sight (NLOS) signal reception. This poses a major problem for mobile services that benefit from accurate urban localization, such as navigation, hyperlocal advertising, and geofencing applications. However, urban NLOS signal reception can be exploited in two ways. First, one can use satellite signal-to-noise ratio (SNR) measurements crowdsourced from mobile devices to create 3D environment maps. Second, in a sort of reverse process called Shadow Matching, SNR measurements provided by a particular device at an instant in time can be compared to 3D maps to provide real-time localization improvement. In this extended abstract we briefly explain how such a system works and describe a scalable, low-cost, and software-only architecture that implements it.**

I. Introduction

While many mobile applications require accurate geolocalization outdoors, it is an unfortunate fact that in dense urban environments positioning accuracy using the Global Positioning System (GPS) degrades significantly, with errors on the order of tens of meters. The main culprit is that in large cities the line-of-sight (LOS) to various satellites becomes occluded by buildings, leading to non-light-of-sight (NLOS) and multipath signal reception. As a result, the only satellites useful for trilateration come from a narrow region in the sky, yielding poor satellite geometries and positioning accuracy. The underlying geometry problem is not solved even as additional constellations of Global Navigation Satellite Systems (GNSS) – such as the Russian GLONASS – become supported by mobile devices.

One promising method to address this satellite *Shadowing Problem* is Shadow Matching (SM). In SM, 3D map databases can be used to compute the shadows of buildings with respect to various satellites. Then, low (or high) satellite signal-to-noise ratio (SNR) measurements can be used to match the device's location to areas inside (or outside) various shadows, thereby reducing uncertainty. Since, for example, any GNSS-capable Android smartphone or tablet can provide via the Location Application Programming Interface (API) its estimated position with uncertainty, as well as satellite coordinates and SNRs, SM can be done entirely in software without any additional infrastructure. A major hurdle to widely deploying SM, though, is that up-to-date urban 3D maps are not always available and can be expensive to obtain. Fortunately, as we elaborate on in [1], [2], large amounts of GNSS data can be used to *create* 3D maps. Intuitively, this is done by assigning many crisscrossing receiver-satellite rays likelihoods of blockage based on measured SNRs, and then stitching these rays together into 3D maps. If the data is crowdsourced from many devices and cloud computing is leveraged, such maps can be built cheaply and scalably, enabling SM-based positioning improvement anywhere GNSS data is regularly collected.

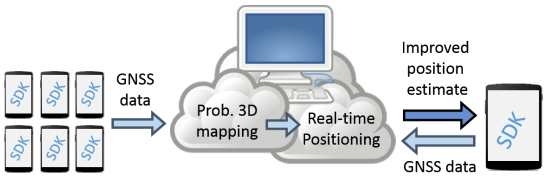


Fig. 1. Proposed system architecture.

II. System Overview

A schematic of a version of the system we proposed in our earlier conference paper [3] is shown in Figure 1. In this system, a Software Development Kit (SDK) is distributed among many mobile devices which allows for crowdsourcing of GNSS data. Cloud-based machine learning routines are then used to process large amounts of this data into probabilistic 3D maps of the environment; these maps are continually updated as additional data becomes available. We give a brief overview of our mapping algorithms in Section III. Once an estimate of the 3D environment is available in a given area, GNSS data from a single mobile device can be streamed via the same SDK to a cloud-based Bayesian filter which performs SM and transmits revised position estimates to the device in real-time; we give a summary of the localization filter in Section IV.

As we recently demonstrated in [4], the above system can be implemented entirely in software at the application level. One slight variation on it would be to provide the same functionality as the SDK via a web API. Another would be an "offline mode" where 3D maps are downloaded to the mobile device, allowing the SM computation to take place there and alleviating the need for a reliable network connection. In any case, efficient crowdsourcing is possible using opportunistic transmission strategies (e.g., only uploading when the device is already transmitting) and by using passive listeners (i.e., only logging data when the GPS receiver is already active).

A.T. Irish, J.T. Isaacs, J.P. Hespanha and U. Madhow are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara ({andrewirish, jtisaacs, hespanha, madhow}@ece.ucsb.edu)
D. Iland and E.M. Belding are with the Department of Computer Science, University of California, Santa Barbara ({iland,ebelding}@cs.ucbs.edu)

Fig. 2. Aerial view of downtown Santa Barbara, with GNSS traces in red and mapped region outlined in yellow.

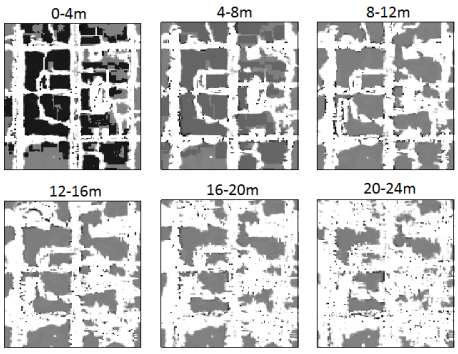


Fig. 3. Horizontal layers of the generated occupancy map of downtown Santa Barbara. White/black corresponds to areas identified as empty/occupied, with shades of grey in between.

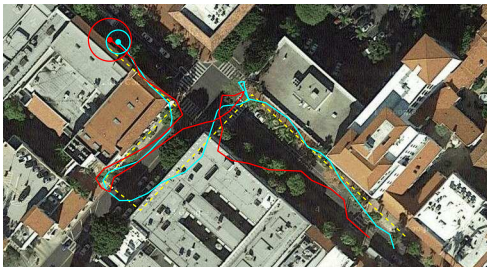


Fig. 4. Positioning improvement in downtown Santa Barbara: true path in yellow, a-GNSS output in red, and corrected (post-SM) path in light blue.

III. Probabilistic 3D Mapping

The considered mapping problem can be described as using noisy GNSS position and SNR measurements, denoted y and z, to estimate a 3D map m. In our previous works [1]–[3] we proposed several different methods to compute a probabilistic estimate of the map. A common thread in these papers is that we employed an *Occupancy Grid* (OG) environment model, where the map is partitioned into a 3D grid of cube-shaped voxels m_i, each of which can either be empty ($m_i = 0$) or occupied ($m_i = 1$). Assuming this representation, a natural question is then the following: Given all of the measurements, what is the likelihood that each voxel is occupied (or empty)? Mathematically, this is equivalent to determining the marginal posterior distributions, $p(m_i|y,z)$, for all i. However, because the *exact* paths of the devices x are unknown, to arrive at solution for the map it turns out one *also* must estimate quantities of the form $p(x_t^j|y,z)$, where x_t^j is the position of a particular device j at time t. In the robotics community, this is referred to as the Simultaneous Localization and Mapping (SLAM) problem.

The major difference between the works [1]–[3] is that each strikes a different balance between computational efficiency and mapping accuracy. In [3] we describe a lightweight algorithm to recursively estimate the map and device path given sequential GNSS measurements. However, in that work (as in virtually all recursive OG mapping techniques), we rely on cell independence assumptions which, although vastly simplifying, are known to be incorrect and lead to overconfident results. In [1] we explicitly model the dependencies of the mapping problem using a Bayesian network, and apply a scalable version of Loopy Belief Propagation, a graphical machine learning algorithm, for inference purposes. In that work, though, we make the simplifying assumption that all GNSS position fixes are error free, i.e., $x = y$. In the third paper [2], we tackle the SLAM problem using a similar but more complex graphical framework. Example experimental results of this last approach, leveraging OpenStreetMap (OSM) data as a-priori information on the first two layers, can be seen Figures 2 and 3, which show the traces and the generated OG map for about 25 hours of input data from 4 Android devices in downtown Santa Barbara.

IV. Real-time Positioning

Making use of notation previously defined, the localization problem can be described as follows: Given a stream of noisy SNR and location data for device j, denoted z^j and y^j, along with a noisy estimate of the map m, what is the best estimate of the device's current location x_t^j, and how confident are we in that estimate? Denoting the occupancy probabilities of the map cells around the device as o^j, and treating these as additional measurements, the quantity we are then interested in is $p(x_t^j|z^j,y^j,o^j)$. In [3] we describe a particle filtering approach which allows one to apply SM against the occupancy map in a recursive, real-time fashion, and purely in software at the mobile application level; in [4] we demonstrated this approach in real-time. An example result of this technique can be seen Figure 4, which shows the positioning improvement in downtown Santa Barbara for a Motorola Moto X smartphone with cellular assisted GPS+GLONASS (a-GNSS).

References

[1] A. T. Irish, J. T. Isaacs, F. Quitin, J. P. Hespanha, and U. Madhow, "Probabilistic 3D mapping based on GNSS SNR measurements," in *Proc. of IEEE International Conf. on Acoustics and Signal Processing*, 2014.

[2] ——, "Belief propagation based localization and mapping using sparsely sampled GNSS SNR measurements," in *Proc. of the International Conf. on Robotics and Automation*, 2014.

[3] J. T. Isaacs, A. T. Irish, F. Quitin, U. Madhow, and J. P. Hespanha, "Bayesian localization and mapping using GNSS SNR measurements," in *Proc. of IEEE Position Location and Navigation Symp.*, 2014.

[4] A. Irish, J. Isaacs, D. Iland, J. Hespanha, E. Belding, and U. Madhow, "Demo: ShadowMaps, the Urban Phone Tracking System," in *Proc. of ACM International Conf. on Mobile Computing and Networking*, 2014.

A Shared Log Storage for Applications Running on the Cloud

Faisal Nawab Vaibhav Arora Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara, Santa Barbara, CA 93106
{nawab,vaibhavarora,agrawal,amr}@cs.ucsb.edu

***Abstract*—Web applications are facing unprecedented demand with users issuing millions of requests per second. The developer needs to scale the application to this demand while providing fault-tolerance and availability. This process is error-prone and need expertise that is not necessarily part of the arsenal of a web developer. The cloud platform attracted many developers for its scalability and availability guarantees. Here, we propose a distributed log store (FLStore) that bridges the gap between the developer and the cloud. This is done by exposing a simple interface (*i.e.*, the log) that is scalable, available, and fault-tolerant. The developer uses a specific API to issue read and append operations which allows building complex applications without worrying about answering the question of scalability. FLStore is designed to be fully distributed to allow the maximum possible scalability.**

***Keywords*—cloud computing, log, availability, fault-tolerance**

I. Introduction

Developing a web application to support millions of requests per second is an arduous task that requires certain expertise that is not necessarily acquired by web developers. This led many developers to adopt the cloud model for scalability and availability. However, a problem still remains that the developer needs to orchestrate these resources and manage them in a way to achieve the desired scalability. This is not an easy process.

Cloud application reside in datacenters that provide the cloud infrastructure. These applications are the ones facing the problem of scalability and trying to achieve it independently. This process is error-prone and lead to solving the same problem repeatedly by all applications. Bridging this gap between the developer and the orchestration of compute and storage resources is the main problem we are tackling in this paper.

We envision a common shared data structure that can serve as a basis for developing web applications. This shared data structure should be built to be distributed and scalable. It also needs to provide a simple API that masks the complexities of achieving a high, scalable performance. A log is an attractive structure because of its simplicity and familiarity to developers. Logs were used in a large number of applications ranging from checkpointing and transaction management to debugging and auditing.

The use of the log as a common shared data structure in the cloud has been tackled recently [1]–[3]. The most notable example is the CORFU shared log [1] that is used in Tango [2]. The main problem that is faced in a shared log is that appending to the log must satisfy uniqueness and order guarantees. Every record must be assigned a unique offset and the existence of a record with a certain offset leads to necessarily existing records with smaller offsets. This is a trivial problem for single-node systems. However, in a shared log, the participating machines must coordinate together to ensure that these guarantees are satisfied. Coordination limits scalability. For a shared log this is specially limiting because all appends contend to access the same resource, *i.e.*, assigning an offset that is larger than the head of the log by one. CORFU proposes the use of a *centralized sequencer* that will assign log offsets to clients to be used later when they append. This takes the sequencer out of the path when the record is appended and reduces the amount of I/O needed by the centralized unit. This, however, is still a bottleneck that limits larger web applications.

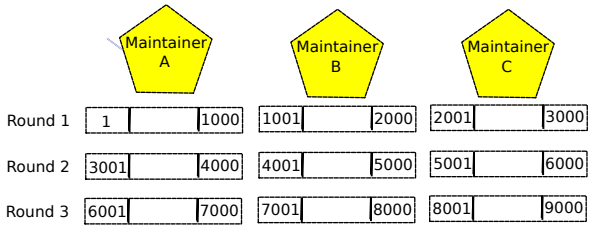


Fig. 1: An example of three deterministic log maintainers with a batch size of 1000 records. Three rounds of records are shown.

We propose FLStore to overcome the need of a centralized entity to append to a shared distributed log. FLStore consists of log maintainers, where each maintainer is assigned to be a champion of a subset of the log. By following a deterministic approach in assigning records, a maintainer can assign an offset to a record without the need of coordination with other maintainers. In the rest of this paper, we will detail the design of FLStore and show how it can scale with added resources.

II. FLStore design, challenges, and evaluation

The distributed shared log, FLStore, consists of a simple interface for adding to and reading from the log. It ensures total order, persistence, and fault-tolerance of records in the log. It is designed to be *fully distributed* to overcome the I/O bandwidth constraints that are exhibited in current shared log protocols.

Design. The FLStore consists of a group of *log maintainers*. The shared log itself is distributed among the participating log maintainers. This means that each machine holds

a partial log and is responsible for its persistence and for answering requests to read its records. This distribution poses two challenges. The first is the way to append to the log while guaranteeing uniqueness and the non-existence of gaps in the log. This includes the access to these records and the way to index the records. The other challenge is maintaining explicit order guarantees requested by the log user. We employ a *deterministic* approach to make each machine responsible for specific ranges of the log. These ranges round-robin across machines where each round consists of a number of records. we will call this number the *batch size*. Figure 1 depicts an example of three log maintainers, *A*, *B*, and *C*. The figure shows the partial logs of the first three rounds if the batch size was set to a 1000 records. If an application wants to read a record it directs the request to the appropriate maintainer.

Adding to the log is done by simply sending a record or group of records to one of the maintainers. The log maintainer batches records together until they reach a threshold and then incorporates them in the set of partial logs. The set of partial logs is ordered to enable identifying which partial log contains a requested record. It is possible that a log maintainer will receive more record appends than others. This creates a load-balancing problem that can be solved by giving the application feedback about the rate of incoming requests at the maintainers.

Log gaps. A maintainer receiving more records also makes it ahead of others. For example, maintainer A can have 10 partial log batches ready when maintainer B is still at its 5^{th} batch. This causes temporary gaps in the logs that can be observed by applications reading the log. To overcome these temporary gaps, simple gossip is propagated between maintainers. The goal of this gossip is to identify the record offset that will guarantee that any record with a smaller offset can be read from the maintainers. We call this offset the *Head of the Log (HL)*. Each maintainer has a vector with a size equal to the number of maintainers. Each element in the vector corresponds to the maximum offset at that maintainer. Initially the vector is initialized to all zeros. Each maintainer updates its value in the local vector. Occasionally, a maintainer propagates its maximum offset to other maintainers. When the gossip message is received by a maintainer it updates the corresponding entry in the vector. A maintainer can decide that the HL value is equal to the vector entry with the smallest value. When an application wants to read or know the HL, it asks one of the maintainers for this value. This technique does not pose a bottleneck for throughput. This is because it is a fixed-sized gossip that is not dependent on the actual throughput of the shared log. It might, however, cause the latency to be higher as the the throughput increases. This is because of the time required to receive gossips and determine whether an offset has no prior gaps.

Explicit order requests. Appends translate to a serial order after they are added by the maintainers. Concurrent appends therefore do not have precedence relative to each other. It is, however, possible to enforce order for concurrent appends if they were requested by the appender. One way is to send the appends to the same maintainer in the order wanted. Maintainers ensure that a latter append will have an offset higher than ones received earlier. Otherwise, it is possible to enforce order for concurrent appends across maintainers. The application waits for the earlier append to be assigned an offset and then attach this offset as a minimum bound. The maintainer that received the record with the minimum bound ensures that the record is buffered until it can be added to a partial log with offsets larger than the minimum bound. This solution however must be pursued with care to avoid a large backlog of partial logs.

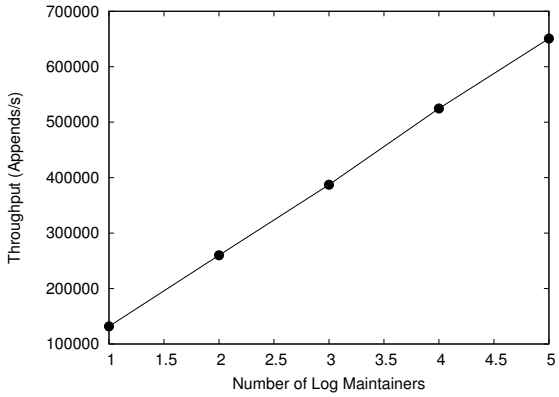


Fig. 2: The append throughput of FLStore while increasing the number of log maintainers.

Evaluation. FLStore was evaluated on a cluster with the following specifications: each machine has Intel Xeon E5620 CPUs that are running 64-bit CentOS Linux with OpenJDK 1.7. The nodes in a single rack are connected by a 10GB switch with an average RTT of 0.15 ms. We show here a single set of experiments to test the scalability of FLStore. The results are shown in Figure 2. In it we increase the number of log maintainers from one to five. A single log maintainer has a throughput of 131747 record appends per second. As we are increasing the number of log maintainers a near-linear scaling is observed. For five log maintainers, the achieved append throughput is 650815 record appends per second. This append throughput is 98.8% of the perfect scaling case. This is to be expected because the shared log design of FLStore removes any dependencies between maintainers.

III. Conclusion

Removing dependencies between maintainers of a shared log store allows scalability and high performance. FLStore removes these dependencies and enables applications to interact with the cloud infrastructure through a simple interface that masks the intricacies of orchestrating compute and storage resources.

References

[1] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14, 2012.

[2] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[3] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

EAGER: API Governance for Modern PaaS Clouds

Hiranya Jayathilaka, Chandra Krintz, Rich Wolski
Department of Computer Science
Univ. of California, Santa Barbara

***Abstract*—Recently, there has been a proliferation of web APIs on the Internet. This makes it difficult to track, control, and compel reuse of web APIs. To address this challenge, we investigate a new approach to API governance – combined policy, implementation, and deployment control of web APIs. Our approach, called EAGER, provides a software architecture that can be easily integrated into cloud platforms to support systemwide, deployment-time enforcement of API governance policies. Specifically, EAGER can check for and prevent backward incompatible API changes from being deployed into production PaaS clouds, enforces service reuse, and facilitates enforcement of other best practices in software maintenance via policies.**

I. Introduction

The growth of the World Wide Web (WWW), web services, and cloud computing have significantly influenced the way developers implement software applications. Instead of implementing all the functionality from the scratch, developers increasingly offload as much application functionality as possible to remote, web-accessible application programming interfaces (web APIs) hosted "in the cloud". As a result, web APIs are rapidly proliferating. At the time of this writing, ProgrammableWeb [1], a popular web API index, lists more than $11,000$ web APIs and a nearly 100% annual growth rate.

This proliferation of web APIs demands new techniques that control and govern the evolution of APIs as a first-class software resource. A lack of API governance can lead to security breaches, denial of service (DoS) attacks, poor code reuse and violation of service-level agreements (SLAs). Unfortunately, most existing cloud platforms within which web APIs are hosted provide only minimal governance support.

Toward this end, we propose EAGER (**E**nforced **A**PI **G**overnance **E**ngine for **R**EST), a model and an architecture that augments existing cloud platforms in order to facilitate API governance as a cloud-native feature. EAGER enforces proper versioning of APIs and supports dependency management and comprehensive policy enforcement at API deployment-time.

Deployment-time enforcement (heretofore unexplored) is attractive for several reasons. First, if run-time only API governance is implemented, policy violations will go undetected until the offending APIs are used. By enforcing governance at deployment-time, EAGER implements "fail fast" in which violations are detected immediately. Further, the overall system is prevented from entering a non-compliant state which aids in the certification of regulatory compliance.

EAGER further enhances software maintainability by guaranteeing that developers reuse existing APIs when possible to create new software artifacts. Concurrently, it tracks changes made by developers to deployed web APIs to prevent any backwards-incompatible API changes from being put into production.

EAGER includes a language for specifying API governance policies. It incorporates a developer-friendly Python programming language syntax for specifying complex policy statements in a simple and intuitive manner. Moreover, we ensure that specifying the required policies is the only additional activity that API providers should perform in order to benefit from EAGER. All other API governance related verification and enforcement work is carried out by the cloud platform automatically.

To evaluate the proposed architecture, we implement EAGER as an extension to AppScale [2], an open source cloud platform that emulates Google App Engine. We show that the EAGER architecture can be easily implemented in extant clouds with minimal changes to the underlying platform technology.

In the sections that follow, we present the design and implementation of EAGER. We then empirically evaluate EAGER using a wide range of experiments, and conclude.

II. EAGER

Figure 1 illustrates the main components of EAGER (in blue) and their interactions. Solid arrows represent the interactions that take place during application deployment-time, before an application has been validated for deployment. Short-dashed arrows indicate the interactions that take place during deployment-time, after an application has been successfully validated. Long-dashed arrows indicate interactions at run-time.

EAGER is invoked whenever a developer attempts to deploy an application, using the developer tools available on his/her workstation. In some cloud implementations these tools could be available as an online service accessed via a web browser. In either case, the application deployment request is intercepted by EAGER API Deployment Coordinator (ADC), which then performs the required governance checks based on the metadata stored in the Metadata Manager. Metadata manager stores application names, versions, dependencies, API specifications, user profiles and API keys. Governance checks are driven by a set of administrator-specified policies that are stored along with the ADC. These policy files are written in Python, and make use of the following assertion functions:

```
assert_true(condition, optional_error_msg)
assert_false(condition, optional_error_msg)
assert_app_dependency(app, d_name, d_version)
assert_not_app_dependency(app, d_name, d_version)
assert_app_dependency_in_range(app, name,\
  lower, upper, exclude_lower, exclude_upper)
```

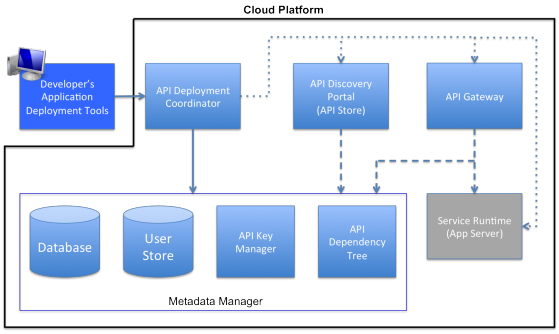


Fig. 1. EAGER Architecture

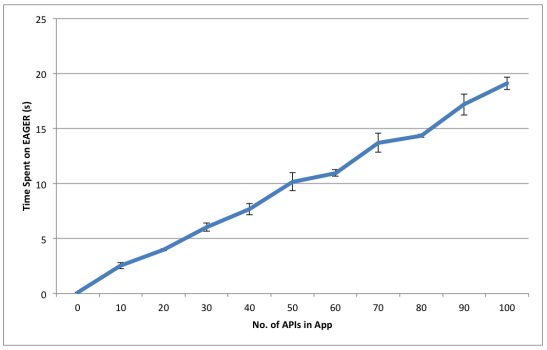


Fig. 2. EAGER Overhead vs. Number of APIs Exported by the Application

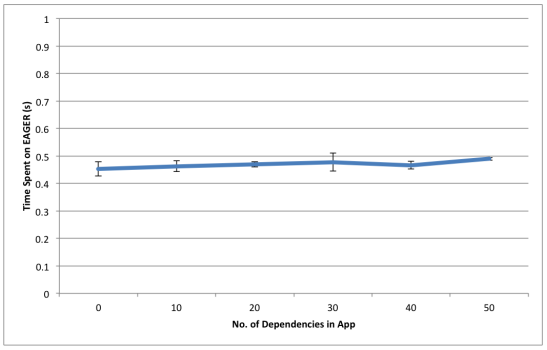


Fig. 3. EAGER Overhead vs. Number of Dependencies Declared in the Application

If a governance check fails (i.e. assertion failure), EAGER will preempt the application deployment, and return an error. Otherwise it proceeds with the application deployment by activating the deployment mechanisms on the developer's or administrator's behalf.

The API Discovery Portal is a web GUI that enables application developers to browse and discover available APIs, and obtain the necessary API keys. The API Gateway intercepts API calls at runtime and performs security and rate-limiting checks.

III. Prototype Implementation

We implemented a prototype of EAGER by extending AppScale [2], an open source PaaS cloud that is functionally equivalent to Google App Engine (GAE). AppScale supports web applications written in Python, Java, Go and PHP. Our prototype implements governance for all applications and APIs hosted in an AppScale cloud.

EAGER Component	Implementation Technology
Metadata Manager	MySQL [3]
API Deployment Coordinator	Native Python implementation
API Discovery Portal	WSO2 API Manager [4]
API Gateway	WSO2 API Manager

TABLE I. Implementation Technologies used to Implement the EAGER Prototype

Table I lists the key technologies that we use to implement various EAGER functionalities described in Section II as services within AppScale itself.

IV. Experimental Results

In this section, we describe our empirical evaluation of the EAGER prototype and evaluate its overhead and scaling characteristics. Figure 2 shows that EAGER overhead grows linearly with the number of APIs exported by an application. This scaling occurs because the current prototype implementation iterates through the APIs in the application sequentially.

Next, we analyze EAGER overhead as the number of dependencies declared in an application grows. For this experiment, we first populate the EAGER Metadata Manager with metadata for 100 randomly generated APIs. Then we deploy an application on EAGER which exports a single API and declares dependencies on the fictitious APIs that are already stored in the Metadata Manager. We vary the number of declared dependencies and observe the EAGER overhead.

Figure 3 shows the results of these experiments. EAGER overhead is not significantly influenced by the number of dependencies. In this case, the EAGER implementation processes all dependency-related information via batch operations. As a result, the number of web service calls and database queries that originate due to varying number of dependencies is fairly constant.

V. Conclusions

In this paper, we describe EAGER, a model and a software architecture that facilitates API governance as a cloud-native feature. EAGER supports comprehensive policy enforcement, dependency management, and a variety of other deployment-time API governance features. It promotes many software development and maintenance best practices including versioning, code reuse and retaining API backwards compatibility.

Our empirical results show that EAGER adds negligibly small overhead to the cloud application deployment process, and the overhead grows linearly with the number of APIs deployed. We also show that EAGER scales well to handle hundreds of dependencies. Our future work considers static and dynamic analysis that automates detection of API specifications and dependencies.

References

[1] "ProgrammableWeb – http://www.programmableweb.com."

[2] C. Krintz, "The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment," *IEEE Internet Computing*, vol. Mar/Apr, 2013.

[3] "MySQL," http://www.mysql.com/, [Online; accessed 25-March-2014].

[4] "WSO2 API Manager – http://wso2.com/products/api-manager/."

Efficient Sparse Matrix-Matrix Multiplication on Multicore Architectures*

Adam Lugowski† John R. Gilbert‡

Abstract

We describe a new parallel sparse matrix-matrix multiplication algorithm in shared memory using a quadtree decomposition. Our implementation is nearly as fast as the best sequential method on one core, and scales quite well to multiple cores.

1 Introduction

Sparse matrix-matrix multiplication (or *SpGEMM*) is a key primitive in some graph algorithms (using various semirings) [5] and numeric problems such as algebraic multigrid [9]. Multicore shared memory systems can solve very large problems [10], or can be part of a hybrid shared/distributed memory high-performance architecture.

Two-dimensional decompositions are broadly used in state-of-the-art methods for both dense [11] and sparse [1] [2] matrices. Quadtree matrix decompositions have a long history [8].

We propose a new sparse matrix data structure and the first highly-parallel sparse matrix-matrix multiplication algorithm designed specifically for shared memory.

2 Quadtree Representation

Our basic data structure is a 2D quadtree matrix decomposition. Unlike previous work that continues the quadtree until elements become leaves, we instead only divide a block if its nonzero count is above a threshold. Elements are stored in column-sorted triples form inside leaf blocks. Quadtree subdivisions occur on powers of 2; hence, position in the quadtree implies the high-order bits of row and column indices. This saves memory in the triples. We do not assume a balanced quadtree.

3 Pair-List Matrix Multiplication Algorithm

The algorithm consists of two phases, a *symbolic phase* that generates an execution strategy, and a *computational phase* that carries out that strategy. Each phase is itself a set of parallel tasks. Our algorithm does not schedule these tasks to threads; rather we use a standard scheduling framework such as TBB, Cilk, or OpenMP.

3.1 Symbolic Phase We wish to divide computation of $C = A \times B$ into efficiently composed tasks with sufficient parallelism. The quadtree structure gives a natural decomposition into tasks, but the resulting tree of sparse matrix additions is inefficient. Instead we form a list of additions for every result block, and build the additions into the multiply step. We let C_{own} represent a leaf block in C, and *pairs* the list of pairs of leaf blocks from A and B whose block inner product is C_{own}.

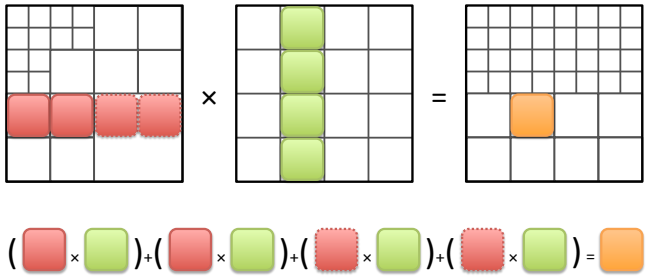


Figure 1: Computation of a result block using a list of pairwise block multiplications.

$$C_{own} = \sum_{i=1}^{|pairs|} A_i \times B_i \tag{3.1}$$

The symbolic phase recursively determines all the C_{own} and corresponding *pairs*.

We begin with $C_{own} \leftarrow C$, and $pairs \leftarrow (A, B)$. If *pairs* only consists of leaf blocks, spawn a compute task with C_{own} and *pairs*. If *pairs* includes both divided blocks and leaf blocks, we temporarily divide the leaves until all blocks in *pairs* are equally divided. This temporary division lets each computational task operate on equal-sized blocks; it persists only until the end of the SpGEMM.

Once the blocks in *pairs* are divided, we divide C_{own} into four children with one quadrant each and recurse, rephrasing divided $C = A \times B$ using (3.1):

$$\begin{aligned} C_1 &= [(A_1, B_1), (A_2, B_3)] \\ C_2 &= [(A_1, B_2), (A_2, B_4)] \\ C_3 &= [(A_3, B_1), (A_4, B_3)] \\ C_4 &= [(A_3, B_2), (A_4, B_4)] \end{aligned} \tag{3.2}$$

For every pair in *pairs*, insert two pairs into each child's *pairs* according to the respective line in (3.2). Each child's *pairs* is twice as long as *pairs*, but totals only 4 sub-blocks to the parent's 8.

3.2 Computational Phase This phase consists of tasks that each compute one block inner product (3.1). Each task is lock-free because it only reads from the blocks in *pairs* and only writes to C_{own}. We extend

*Supported by Contract #618442525-57661 from Intel Corp. and Contract #8-482526701 from the DOE Office of Science.

†CS Dept., UC Santa Barbara, alugowski@cs.ucsb.edu

‡CS Dept., UC Santa Barbara, gilbert@cs.ucsb.edu

Gustavson's sequential algorithm [4] in Algorithm 1.

Our addition to Gustavson is a mechanism that combines columns j from all blocks B_i in *pairs* to present a view of the entire column j from B. We then compute the inner product of column j and all blocks A_i using a "sparse accumulator", or *SPA*. The SPA can be thought of as a dense auxiliary vector, or hash map, that efficiently accumulates sparse updates to a single column of C_{own}.

A and B are accessed differently, so we *organize* their column-sorted triples differently. For constant-time lookup of a particular column i in A, we use a hash map with a $i \rightarrow (\mathit{offset}_i, \mathit{length}_i)$ entry for each non-empty column i. A CSC-like structure is acceptable, but requires O(m) space. We iterate over B's non-empty columns, so generate a list of $(j, \mathit{offset}_j, \mathit{length}_j)$. Both organizers take $O(nnz)$ time to generate. A structure that merges all B_i organizers enables iteration over logical columns that span all B_i.

Algorithm 1 Compute Task's Multi-Leaf Multiply

Require: C_{own} and *pairs*
Ensure: Complete C_{own}
  for all (A_b, B_b) in *pairs* **do**
    organize A_b columns with hash map or CSC
    organize B_b columns into list
  end for
  merge all B organizers into *combined_B_org*
  for all (column j, $PairList_j$) in *combined_B_org* **do**
    $SPA \leftarrow \{\}$
    for all (A_b, B_b) in $PairList_j$ **do**
      for all non-null k in column j in B_b **do**
        accumulate $B_b[k, j] \times A_b[:, k]$ into SPA
      end for
    end for
    copy contents of SPA to $C_{own}[:, j]$
  end for

4 Experiments

We implemented our algorithm in TBB [7] and compared it with the fastest serial and parallel codes available, on a 40-core Intel Nehalem machine. We test by squaring Kronecker product (RMAT) matrices [6] and Erdős-Rényi matrices.

Observe from Table 1 that QuadMat only has a small speed penalty on one core compared to CSparse, but gains with two or more cores.

5 Conclusion

Our algorithm has excellent performance, and has the potential to be extended in several ways. Our next steps include a triple product primitive that does not materialize the entire intermediate product at any one time, and computing $A^T \times B$ with similar complexity to $A \times B$.

Table 1: SpGEMM results on E7-8870 @ 2.40GHz - 40 cores over 4 sockets, 256 GB RAM. Note: CombBLAS is an MPI code that requires a square number of processes.

Squared Matrix		R_{16}	R_{18}	ER_{18}	ER_{20}
Each Input nnz		1.8M	7.6M	8.39M	33.6M
Output nnz		365M	2.96G	268M	1.07G
CSparse [3]	1p	14s	122s	9s	58s
CombBLAS [2]	1p	154s	1597s	64s	248s
	9p	19s	155s	8s	34s
	36p	8s	49s	3s	12s
QuadMat	1p	19s	150s	13s	111s
	2p	10s	87s	8s	66s
	9p	3s	21s	3s	18s
	36p	2s	11s	2s	9s

References

[1] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proc. 21st Symp. on Parallelism in Algorithms and Arch.*, 2009.

[2] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *Intl. J. High Perf. Computing Appl.*, 25(4):496–509, 2011.

[3] T. A Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, Sept 2006.

[4] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[5] J. Kepner and J. R. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[6] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *Proc. 9th Principles and Practice of Knowledge Disc. in Databases*, pages 133–145, 2005.

[7] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[8] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, 1984.

[9] Y. Shapira. *Matrix-based Multigrid: Theory and Applications*. Springer, 2003.

[10] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[11] R. A. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

Fuzz Testing using Constraint Logic Programming

Kyle Dewey Jared Roesch Ben Hardekopf
PL Lab, Department of Computer Science, University of California, Santa Barbara
{kyledewey, jroesch, benh}@cs.ucsb.edu

***Abstract*—Fuzz testing, a technique for automatically generating programs, is useful to build confidence in compilers and interpreters. In this domain, it is desirable for fuzzers to exploit semantic knowledge of the language being tested and to allow for targeted generation of programs that showcase specific language features and behaviors. However, the predominant program generation technique used by most language fuzzers, stochastic context-free grammars, does not have this property. We propose the use of constraint logic programming (CLP) for program generation. Using CLP, testers can write declarative predicates specifying interesting programs, including syntactic features and semantic behaviors. We apply CLP to automatically generate JavaScript and Scala programs that exhibit interesting properties, including arithmetic overflow, prototype-based inheritance, higher-order functions, absence of runtime errors, and well-typedness. We show that the resulting program generation is fast and that our ability to generate interesting programs is much greater than that of stochastic grammars.**

I. Introduction

Language fuzzing, i.e., the automated generation of programs for testing purposes, is a proven strategy for building confidence in the correctness of compilers and interpreters. For example, `jsfunfuzz` [1] and CSmith [2] have found thousands of bugs in interpreters for JavaScript and in compilers for C. However, existing language fuzzing techniques are, in general, ad-hoc and limited in their scope and their effectiveness. Most existing fuzzers are based on the generation of programs belonging to *stochastic grammars* [3], which are Backus-Naur Form (BNF) grammars annotated with probabilities of emitting individual nodes. This approach is problematic for two reasons: 1.) it is decidedly syntax-oriented, making it difficult to express semantic properties, and 2.) it conflates the search space with the strategy used to search this space. We elaborate further on these two points.

The syntax-oriented nature of stochastic grammars makes it hard to express anything beyond simple syntactic validity. For example, consider well-typed programs in a statically typed language. Using only the language grammar, ill-typed programs can result [4], which prevent testing past the type-checking phase. Artificially restricting the grammar to emit well-typed programs (e.g, [3], [5]) is non-trivial, and it prevents whole classes of well-typed programs from being emitted. Generating all well-typed programs requires additional techniques beyond the grammar (e.g., Csallner et al. [6]), which are highly problem-specific. A syntax-oriented approach also inhibits the specification of programs with interesting *behaviors* that span multiple syntactic forms. For example, consider the use of prototype-based inheritance in JavaScript, which requires the coordination of function declarations, object updates, the use of `new` in a specific way. Simply tuning probabilities is not enough to overcome these issues, since these expressibility problems go beyond syntax.

Additionally, stochastic grammars conflate search space (the infinite set of syntactically valid programs) with search strategy (how programs in this set are chosen). Specifically, stochastic grammars force the tester to choose a *random* search strategy, that of choosing programs in a random fashion according to some probability distribution. While a random strategy can be effective, it is not always the most effective strategy [7], and indeed the best strategy can be problem-specific.

In short, stochastic grammars do not give the language implementer adequate tools for specifying what sorts of tests are interesting, making any sort of targeted testing literally a case of luck. To amend this situation, we propose the use of constraint logic programming (CLP) [8] for fuzzing. CLP is strictly more general than traditional stochastic grammars, allowing for **both** syntactic and semantic properties to be specified with relative ease. Additionally, CLP treats the search technique as an orthogonal concern to the search space, allowing testers to freely choose between different search strategies as they see fit. To get a better idea of what CLP is and does, we have provided an example in Section II. We apply CLP to generating real-world JavaScript and Scala programs with specific properties in Section III, and compare these to traditional stochastic grammars in Section IV.

II. Example in CLP

Say we have the following stochastic grammar, where n is a placeholder for arbitrary integers.

$$e \in \mathit{Exp} ::= (n \in \mathbb{Z})^{0.6} \mid (e_1 + e_2)^{0.4}$$

This can be directly translated into CLP like so:

```
exp(num(N)) :- maybe(0.6),
    N #>= INT_MIN, N #=< INT_MAX.
exp(add(E1, E2)) :- exp(E1), exp(E2).
```

The above code can be used as a generator to create valid expressions, as with the following query:

```
:- exp(E), write(E).
```

With respect to the above code, semantically, the CLP engine will first try to generate a number (`num`), which succeeds with probability `0.6`. In the event of failure, the engine will backtrack to the `add` alternative, which conveniently happens 40% of the time (hence no need to specify `maybe(0.4)`).

As of yet, we have not demonstrated anything that traditional stochastic grammars cannot do. Let us take a step beyond this, and start reasoning about expressions that evaluate to particular values:

```
eval(num(N), N) :- maybe(0.6),
    N #>= INT_MIN, N #=< INT_MAX.
eval(add(E1, E2), Res) :-
    eval(E1, N1), eval(E2, N2),
    Res #= N1 + N2.
:- eval(E, 42), write(E).
```

The query above generates arbitrary expressions which evaluate down to `42`. Using this same basic strategy of using the CLP code as a generator, we can encode a variety of interesting properties. For example, to generate well-typed expressions, all one needs is a typechecker implemented in CLP, which can be used in a manner similar to the one above to generate any and all well-typed programs.

III. Application to JavaScript and Scala

JavaScript is a dynamically typed language which has undergone extensive fuzz testing in the past [1], [9]. Using CLP, we have implemented a fuzzer that uses a simple unsound type system for reducing the number of runtime errors (`js-err`), a fuzzer for generating arithmetic expressions that overflow into doubles from integers (`js-overflow`), a fuzzer for generating programs that stress prototype-based inheritance (`js-inher`), and a fuzzer for stressing the use of the `with` construct along with closures (`js-withclo`). These fuzzers employ a mix of complex syntactic forms along with type information and semantic information.

Scala is a feature-rich statically typed language with a fairly sophisticated type system. We are particularly interested in generating well-typed Scala programs. Given the language's complexity, we chose to focus on a subset of the language which exposed many details of the type system, specifically variables, higher-order functions, classes, methods, method calls, and conditionals. We implemented a fuzzer which was simply-typed in this context (`scala-base`), along with a more complex fuzzer which handled additional features like generic types, subtyping, inheritance, and pattern matching (`scala-full`). To the best of our knowledge, this is the first time fuzzing has been applied to generics and higher-order functions, neither of which can be handled by the state of the art on fuzzing well-typed programs (e.g, Csallner et al. [6]).

IV. Evaluation

Our hypothesis is that stochastic grammars cannot possibly express the more complex language properties described in Section III, since grammars and probabilities alone are insufficient to describe these properties. To gather relevant evidence, we first wrote JavaScript and Scala fuzzers using the traditional stochastic grammar approach. We used CLP to implement these traditional fuzzers, which helps demonstrate that CLP is truly more general. Next, the probabilities of the traditional stocahstic fuzzers were painstakingly tuned to make them behave as closely to the specialized fuzzers (described in Section III) as possible. We then compared the rates at which the two kinds of fuzzers generated programs of interest. For example, with `scala-base`, we are interested in seeing how many well-typed programs a traditional stochastic fuzzer can produce within a given unit a time relative to our specialized CLP fuzzers. The results are presented in Table I.

Generator	Stochastic	CLP	CLP / Stochastic Ratio
`js-err`	9,880	37,759	3.8
`js-overflow`	123	958	7.8
`js-inher`	0	126,194	∞
`js-withclo`	0.04	125,901	3,147,525
`scala-base`	56	105,510	1,884
`scala-full`	0	183,187	∞

TABLE I. Number of programs exhibiting properties of interest generated by traditional stochastic fuzzers and our CLP-based fuzzers, in programs generated per second.

As shown, our specialized fuzzers always outperform the traditional stochastic versions, often by a significant margin. In two separate cases, the stochastic versions could not generate even a single program that exhibited the property of interest, demonstrating that even trivial-sounding properties may be effectively out of range for stochastic techniques.

While data has not been provided regarding complexity or the number of lines of code (LOC) required for the implementation of the specialized generators, we have found that these tend to be surprisingly simple. In the vast majority of cases, the specialized generator is within 100 LOC of the stochastic version. In the absolute worst case (`js-err`), the specialized generator is around $2\times$ larger than the stochastic version, which we consider acceptable considering the strong guarantees on the programs generated. This leads us to conclude that our CLP-based technique allows for highly specialized generation relative to traditional stochastic grammars, without excessive additional difficulty.

References

[1] J. Ruderman, "Introducing jsfunfuzz," 2007. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[3] W. M. McKeeman, "Differential testing for software." *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[4] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287651

[5] V. St-Amour and N. Toronto, "Experience report: applying random testing to a base type environment," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '13. New York, NY, USA: ACM, 2013, pp. 351–356. [Online]. Available: http://doi.acm.org/10.1145/2500365.2500616

[6] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for java," *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1002/spe.602

[7] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An evaluation of exhaustive testing for data structures," MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, Tech. Rep., 2003.

[8] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *Journal of Logic Programming*, vol. 19, pp. 503–581, 1994.

[9] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362831

Automated Test Generation from Vulnerability Signatures

Abdulbaki Aydin, Muath Alkhalaf, and Tevfik Bultan
Computer Science Department, University of California, Santa Barbara
Email: {baki,muath,bultan}@cs.ucsb.edu

***Abstract*—Web applications need to validate and sanitize user inputs in order to avoid attacks such as Cross Site Scripting (XSS) and SQL Injection. Writing string manipulation code for input validation and sanitization is an error-prone process leading to many vulnerabilities in real-world web applications. Automata-based static string analysis techniques can be used to automatically compute vulnerability signatures (represented as automata) that characterize all the inputs that can exploit a vulnerability. There are several factors that limit the applicability of static string analysis techniques: 1) undecidability of static string analysis requires the use of approximations leading to false positives, 2) static string analysis tools do not handle all string operations, 3) dynamic nature of the scripting languages makes static analysis difficult. In this paper, we show that vulnerability signatures computed for deliberately insecure web applications (developed for demonstrating different types of vulnerabilities) can be used to generate test cases for other applications. Given a vulnerability signature represented as an automaton, we present algorithms for test case generation based on state, transition, and path coverage. These automatically generated test cases can be used to test applications that are not analyzable statically, and to discover attack strings that demonstrate how the vulnerabilities can be exploited.**

I. Introduction

Correctness of input validation and sanitization operations is a crucial problem for web applications. Unfortunately, web applications are notorious for security vulnerabilities such as SQL injection and XSS that are due to lack of input validation and sanitization, or errors in string manipulation operations used for input validation and sanitization.

We present an automated testing framework that targets testing of input validation and sanitization operations in web applications for discovering vulnerabilities [2]. We combine automated testing techniques with static string analysis techniques for vulnerability analysis. We use static string analysis [1] to obtain an over-approximation of all the input strings that can be used to exploit a certain type of vulnerability. This set of strings is called a vulnerability signature, which could be an infinite set containing arbitrarily long strings.

Using the vulnerability signature automata generated by analyzing the deliberately insecure web applications, we automatically generate test cases based on three coverage criteria: state, transition and path coverage. Each test case corresponds to a string such that, when that string is given as a text field input to a web application, it may exploit the vulnerability that is characterized by the given vulnerability signature. Our automated test generation algorithm minimizes the number of test cases while achieving the given coverage criteria.

This research is supported in part by NSF grant CNS-1116967. Muath Alkhalaf is funded in part by the King Saud University.

II. Automated Testing Framework

The high-level flow of our automated testing framework for input validation and sanitization functions is shown in Figure 1.

A. Automata-based Static String Analysis

Automata-based string analysis is a static program analysis technique. Given a set of input values represented as automata, it symbolically executes the program to compute the set of string values that can reach to each program point. Using a forward-analysis that propagates input values to sinks (i.e., security sensitive functions), we identify the attack strings that can reach to a given sink. Then, propagating the attack strings back to user input using backward analysis results in an automaton that corresponds to the vulnerability signature.

Automata-based static string analysis is challenging due to several reasons. Due to undecidability of string verification, string analysis techniques use conservative approximations that over-approximate the vulnerability signatures. Hence, vulnerability signatures may contain strings that do not correspond to attacks, leading to false positives. Moreover, string analysis tools only model a subset of available string library functions, and when an un-modeled library function is encountered, the function has to be over-approximated to indicate that it can return all string values, which results in further loss of precision. Furthermore, forward and backward symbolic execution using automata can cause exponential blow-up in the size of the automata when complex string manipulation operations such as string-replace are used extensively. Finally, dynamic nature of scripting languages used in web application development makes static analysis very challenging and applicable to a restricted set of programs. Due to all these challenges it is not possible to have a push-button automata-based string analysis that works for all real-world applications.

Combining static vulnerability analysis techniques with automated test generation allows us to compensate for the weaknesses of the static vulnerability analysis techniques. In our approach static vulnerability analysis is applied to a small set of programs and the results from this analysis is used for testing other applications. Hence, programs with features that make static vulnerability analysis infeasible can still be checked using automated testing. Moreover, the approximations that are introduced by static vulnerability analysis that lead to false positives are eliminated during testing.

B. Generating Vulnerability Signatures

Security researchers have developed applications that are deliberately insecure to demonstrate typical vulnerabilities. These applications are sometimes used to teach different pitfalls to avoid in developing secure applications, and sometimes

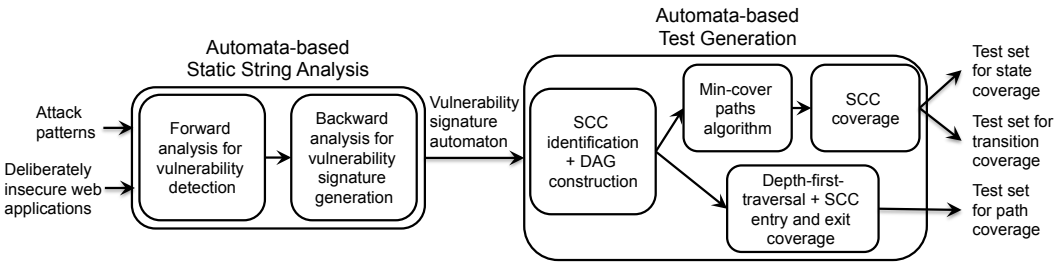


Figure 1. Automated Test Generation from Vulnerability Signatures

they are used as benchmarks for evaluating different vulnerability analysis techniques. In our framework we use static string analysis techniques to analyze deliberately insecure applications and to compute a characterization of inputs that can exploit a given type of vulnerability.

In order to generate the vulnerability signature for an application, we need an attack pattern (specified as a regular expression) that characterizes a particular vulnerability. An attack pattern represents the set of attack strings that can exploit a particular vulnerability if they reach a sink (i.e., a security sensitive function). Attack patterns for different types of vulnerabilities are publicly available and can be used for vulnerability analysis.

Given an attack pattern and a deliberately insecure web application, we use automata-based static string analysis techniques to generate a vulnerability signature automaton that characterizes all the inputs for that application that can result in an exploit for the vulnerability characterized by the given attack pattern.

C. Automated Test Generation from Vulnerability Signatures

Given a vulnerability signature automaton, any string accepted by the automaton can be used as a test case. Hence, any path from the start state of the vulnerability signature automaton to an accepting state characterizes a string which can be used as a test case. However, a vulnerability signature automaton typically accepts an infinite number of strings since, typically, there are an infinite ways one can exploit a vulnerability. In order to use vulnerability signature automata for testing, we need to prune this infinite search space.

The mechanism that allows an automaton to represent an infinite number of strings is the loops in the automaton. In order to minimize the number of test cases, we have to minimize the way the loops are traversed. We do this by identifying all the strongly-connected components (SCCs) in an automaton using Tarjan's strongly connected components algorithm [3] and then collapsing them to construct a directed acyclic graph (DAG) that only contains the transitions of the automaton that are not part of an SCC and represents each SCC as a single node. Using this DAG structure, we generate tests based on three criteria: 1) *state coverage* where the goal is to cover all states of the automaton (including the ones in an SCC), 2) *transition coverage*, where the goal is to cover all transitions of the automaton (including the ones in an SCC), 3) *path coverage*, where the goal is to cover all the paths in the DAG that is constructed from the automaton, while also covering all possible ways to enter and exit from an SCC.

We implement the state and transition coverage using the min-cover paths algorithm [4] on the DAG representation followed by a phase that ensures the coverage of the states and transitions inside the SCC nodes. We implement the path coverage using depth-first-traversal, where, when an SCC node is encountered, we ensure that all entry and exit combinations are covered in the generated test cases.

We use the test strings generated from vulnerability signatures of deliberately insecure web applications to test other applications. If the applications we test contain sanitization errors similar to the errors in deliberately insecure web applications or if they do not use proper sanitization, then the generated test cases can discover their vulnerabilities without analyzing them statically. Note that the test inputs generated from vulnerability signatures can also be used for applications that are statically analyzable in order to eliminate false positives and construct exploits (i.e., to generate concrete inputs that demonstrate how a vulnerability can be exploited).

D. Implementation and Experiments

We implemented this approach and conducted experiments to evaluate its effectiveness [2]. We generated vulnerability signatures from two deliberately insecure applications, and then used the automatically generated tests from these vulnerability signatures to test 5 open source applications. Our experiments show that our approach generates effective tests for detecting vulnerabilities. We also observed that tests generated using path coverage are most effective in detecting vulnerabilities. Transition coverage generates tests that are almost as effective, and the number of test cases generated by transition coverage is smaller (resulting in more efficient testing). Finally, tests generated using the state coverage criteria are not effective in detecting vulnerabilities.

REFERENCES

[1] F. Yu, M. Alkhalaf, and T. Bultan, "Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses," in *ASE*, 2009, pp. 605–609.

[2] A. Aydin, M. Alkhalaf, and T. Bultan, "Automated test generation from vulnerability signatures," in *ICST*, 2014.

[3] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[4] E. Ciurea and L. Ciupal, "Sequential and parallel algorithms for minimum flows," *Journal of Applied Mathematics and Computing*, vol. 15, no. 1-2, pp. 53–75, 2004.

Coexecutability: How To Automatically Verify Loops

Ivan Bocić, Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara, USA
{bo, bultan}@cs.ucsb.edu

***Abstract*—Verification of web applications is a very important problem, and verifying loops is necessary to achieve that goal. However, loop verification is a long studied and very difficult problem. We find that interdependence of iterations is a major cause of this difficulty. We present coexecution - a way to model a loop that avoids the problem of iteration interdependence. We introduce the coexecutability condition that implies that coexecution is a correct model. Through experiments, we demonstrate that coexecution reduces the number of inconclusive verification results by three times, and in 43% of cases increases performance of verification by at least an order of magnitude.**

Keywords*—*Verification, Loops

I. Introduction

Web applications are integral to the functioning of the modern society. As such, their correctness is of fundamental importance. In our recent work [3], we focused on automated verification of data store properties in web applications. For example, in an online forum application involving `User`s and `Post`s, our approach could be used to automatically answer questions such as "Does every `Post` have an associated `User`?". In Ruby on Rails [4], a `delete_user` action could be implemented as shown in Figure 1.

```
class UsersController
  def destroy_user
    user = User.find params[:user_id]
    user.posts.each do |post|
      post.destroy
    end
    user.destroy
  end
end
```

Fig. 1. Rails Code for User Deletion

Our tool analyzes the actions of a given web application using automated theorem proving techniques to deduce whether a given property is preserved by all actions.

Verification of loops is a long studied, difficult and generally undecidable problem [2], [1], often requiring manual intervention in form of loop invariants. We found that the main problem with verification of loops stems from automated reasoning about of how iterations affect each other. In general, the sequence of iterations is of an arbitrary length, with any iteration being potentially affected by all previous iterations. When reasoning about a loop's behavior, an automated theorem prover would enumerate iterations one by one to deduce all the possible results of a loop, producing increasingly more complex formulas, and may never terminate.

In this paper we identify a special class of loops for which modeling iteration interdependence is not necessary. We define a concept called *coexecution*, which models loop iteration executions in a way that avoids iteration inter-dependability. We define a condition under which coexecution is equivalent to sequential execution. Through experiments, we show that coexecution significantly improves the viability and performance of loop verification.

II. Formalization

For brevity and simplicity, we refer to the elements of a data store as *entities* without going into specifics of their nature. Think of them as objects that represent the database, associated to one another as defined by the schema. For similar reasons, we will assume that these entities contain no mutable data, and can only be created or deleted. Modifications could be implemented as deletion followed by creation of a similar entity populated with the updated data.

A data store *state* is a set of entities that exist in some point in time, for example, a set of `User` objects and their associated `Post`s. A statement serves to update the state. We define a statement S as a set of state pairs such that executing the statement S from state s may result in state s' if and only if $(s, s') \in S$. For example, a statement that creates an entity e can be defined as a set of state pairs (s, s') s.t. $e \notin s$, $e \in s'$, and $\forall e' : e \neq e' \rightarrow (e' \in s \leftrightarrow e' \in s')$.

At a high level a loop is defined as a sequential execution of k iterations, where k is the size of the set being iterated on. While all iterations execute the same loop body, in our formal model, we treat each iteration as a unique statement. Hence, a loop is modeled as the sequential execution of iteration statements $S_1, S_2, \ldots S_k$ that migrates state s_0 to s_k if and only if:

$$\exists s_1, \ldots s_{k-1} : (s_0, s_1) \in S_1 \wedge \cdots \wedge (s_{k-1}, s_k) \in S_k$$

Our key problem comes from this definition. To describe possible migrations from s_0 to s_k, the theorem prover needs to deduce all possible s_1s that are reachable from s_0, then s_2s from the s_1s etc. The complexity of each subsequent state increases and, since k can often be any integer, this process may never terminate.

III. Solution

When we manually investigated loops in web applications, we found out that, typically, loop iterations do not affect each other. Therefore, modeling the rules of how iterations affect each other is not necessary, and is in fact not desirable because this dependence is the major problem of verification feasibility. For example, the loop iterations in Figure 1 are not inter-dependent.

We introduce coexecution, a way to model the composition of multiple statements that are not inter-dependent. Coexecution entails identifying the effects of each iteration's execution as if this iteration were executed in isolation from all others, and combining these effects into one major operation that we can use to model the loop.

To formally express this process, we first need to express the effects of a statement's execution. We define a *delta* of states s and s' (denoted as $(s' \ominus s)$) to be a structure (O_c, O_d) where O_c is the set of entities that exist in s' but not in s (created entities), and O_d is the set of entities that exist s but not in s' (deleted entities).

Next, we need a way to combine multiple deltas into one delta that encompasses all of them. We combine deltas $\delta_1 = (O_{c1}, O_{d1})$ and $\delta_2 = (O_{c2}, O_{d2})$ by using the *delta union* ($\cup$) operator, defined as $\delta_1 \cup \delta_2 = (O_{c1} \cup O_{c2}, O_{d1} \cup O_{d2})$. Note that, in general, the same entity may be both in the create and the delete set of the result of a delta union. We call a delta *conflicting* if its delete and create sets are not exclusive.

Finally, we need a way formalize the execution of a delta. Given a state s and a non-conflicting delta $\delta = (O_c, O_d)$, we can *apply* δ to s using the operator $\oplus$: the result of that operation will be a state that contains all elements of $(s \cup O_c) \setminus O_d$. The apply delta operation is undefined for conflicting deltas.

We can finally define coexecution. The coexecution of statements $S_1, \dots S_k$ migrates between states s and s' iff:

$$\exists s_1 \dots s_k : (s, s_1) \in S_1 \wedge \dots \wedge (s, s_k) \in S_k \wedge \\ s' = s \oplus (s_1 \ominus s) \cup \dots \cup (s_k \ominus s)$$

A. The Coexecutability Condition

In general, the result of coexecution of statements is different from the result of sequential execution. For example, in sequential execution, a statement may undo some of the previous statement's work, or may read the modifications done by a previous statement and behave differently because of that, even if never undoing previous work. These possibilities would not be captured by the coexecution approach, since coexecution models the execution of all statements in isolation. Therefore, we need a condition under which sequential execution and coexecution are equivalent - the coexecutability condition.

It is intuitive that these two models of execution should be equivalent if no statement reads what another creates or deletes, and if no statement creates what another has deleted or vice versa. However, we defined statements as arbitrary migrations between states, and so the very notion of *reading* an entity is undefined.

Intuitively, a statement reads an entity if the statement's semantic or intent changes depending on the presence of the read entity. However, this is a problematic intuition since, for example, creating or deleting an entity is possible if and only if that entity does not or does exist, respectfully. That means that any modification of the data implies reading said data, which is a very limiting factor. Consider a loop in which every iteration deletes all data. Coexecution is equivalent to the sequential execution for this loop, yet if all iterations delete all data they also read all data, implying that the coexecutability condition fails. This intuition is too coarse.

We define reading in a more flexible way. Let's introduce the concept of a *delta cover* Δ of a statement S and a set of states α as a set of deltas such that all executions of S from any state in α is expressed by at least one delta from Δ, and that applying any delta from Δ to any state in α is accepted by the statement S. Defined like this, a delta cover exists for a statement S for these states only if S has a certain semantic (which is the set of deltas) that is not different between the given states. In words, it does not differentiate them.

Using the concept of delta covers, we can define what it means for a statement to read an entity e. A statement reads an entity e if and only if there exists a pair of states such that the only difference between them is the existence of e, and there exists no delta cover of S over these two states.

Let us reconsider the loop that deletes all data in every iteration, within this new definition of reading. The delete all statement does not read any entity because there exists a delta cover over any pair of states as defined above: this delta cover contains a single delta that has an empty create set, and all possible entities in the delete set. This makes sense because a delete all statement is not affected by the existence of any entity, it always does the same thing. In this loop, because all iterations delete the same set of entities without undoing or being guided by other iterations, this loop is coexecutable.

Note that coexecution, while seeming similar to parallelization, is fundamentally different. Consider a loop that, on each iteration, deletes all entities and subsequently creates an entity in one atomic step. If iterations were parallelized, the end result of the loop will always have exactly one entity in the state. If these iterations were coexecuted, the end result will have as many entities as there were iterations. This loop is not coexecutable, but it demonstrates the difference between parallelization and coexecution.

IV. Experiments and Conclusion

We measured and compared the performance of verification on four open source Ruby on Rails applications when using coexecution vs sequential models of loops. We run our experiments under different heuristics in order to demonstrate that coexecution is generally easier to automatically reason about.

Our results show that, by using coexecution, the number of inconclusive results involving loop verification is reduced more than threefold - from 62% down to 17%. Furthermore, the performance of verification is largely improved. In 43% of cases, coexecution yields an improvement of at least an order of magnitude. In 22% cases, coexecution provided an improvement of four orders of magnitude.

Our results demonstrate that automated verification of web application actions is feasible in most cases, even when they contain loops.

References

[1] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 260–264, 2001.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] I. Bocic and T. Bultan. Inductive verification of data model invariants for web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, May 2014.

[4] Ruby on Rails, Feb. 2013. http://rubyonrails.org.

Code-Specific, Sensitive, and Configurable Plagiarism Detection

Kyle Dewey Ben Hardekopf
PL Lab, Department of Computer Science, University of California, Santa Barbara
{kyledewey, benh}@cs.ucsb.edu

***Abstract*—There is great incentive to be able to accurately and automatically detect plagiarism in source code, and there exist a variety of techniques and tools for performing this task. That said, many of these are fraught with issues, such as being designed around plain text instead of source code, being lossy and fundamentally prone to false negatives, and lacking the ability to be configured for specific plagiarism-detection tasks. We have developed simple plagiarism-detection technique which addresses all these problems, being specific to source code, lossless, and highly configurable. We apply this technique to developing a plagiarism detector for Scala code, which has already proven itself to be useful.**

I. Introduction

Plagiarism of source code is an all too common phenomenon. In instructional settings, plagiarism undermines the entire learning process, as it allows students to get ahead without any real learning occurring. As such, there is great incentive to be able to detect plagiarism, especially in an automated manner. To this end, a large variety of automated plagiarism detection methods and tools have already been proposed with varying success.

While existing tools abound, these are not without their flaws, especially with respect to source code. Most tools (e.g., MOSS [1]) are designed around plain text, which has dramatically different structure than source code. For example, text-based methods are prone to false negatives with source code, since they may consider inconsequential changes like variable renamings to be significant. Conversely, text-based methods are also prone to false positives with source code, since they do not understand that common elements like particular sequences of keywords (e.g., `using namespace std;` in C++) are relatively unimportant.

Many tools (e.g., MOSS [1]) also systematically discard information about their inputs, making these systems inherently lossy and prone to false negatives. Finally, all tools which we are aware of cannot be specialized for specific tasks. For example, say a given programming assignment is heavily based on the use of conditionals (e.g., `if`), and so many solutions free of plagiarism are likely to have very repetitive patterns of conditionals. An obvious solution is to de-emphasize conditionals during plagiarism detection, but no tool we know of exists which can work with this information.

To address these concerns, we have developed a technique which is specific to source code, lossless in nature, and highly configurable. We allow programmers themselves to decide what is important and by how much via simple abstract syntax tree (AST) traversals and scoring functions. We have successfully used this technique to detect plagiarism in real Scala code, which previously had to be checked manually due to the lack of availability of language-specific plagiarism detectors.

II. Our Technique

A diagram illustrating our overall technique is presented in Figure 1. We first parse input programs into an AST representation. From there, a user-defined function is applied which converts the AST into a stream of features, where the features themselves are user-defined. For example, the user may have prior knowledge that the AST `1 + n` is typically involved in plagiarism whenever the type of `n` is of `double`. With our technique, the user can emit a feature for this pattern and this pattern only, skipping over the rest of the AST. Not only does this remove noise (reducing false positives), it reduces the input size for downstream processing, which can help speed up detection. As another example, say that for a particular assignment a handful of functions are provided, and those provided functions are considered more important than any other functions. With our technique, the user can emit individual features just for these functions, and uniformly use some other feature for all other functions. This allows the downstream detection to focus in on provided routines, potentially improving the precision of detection.

Once ASTs are converted to streams of features, the streams must be compared for similarity. We want this to be a lossless comparison to avoid false negatives, and so we apply the Smith-Waterman algorithm [2]. This algorithm, originally used for comparing biological sequences, is used to align two sequences and provide a score describing how good the alignment is. The algorithm guarantees that the provided alignment is optimal, made possible via dynamic programming. The algorithm can insert interleaving spaces, known as *gaps*, into the sequences, in order to maximize scores. To demonstrate, an optimal alignment of the strings "foobar" and "foobazbar" is shown below, where "-" represents a gap:

```
f o o - - - b a r
f o o b a z b a r
```

The score for aligning a particular feature or gap with some other feature or gap is determined via a user-defined scoring function. Since the scoring function is user-defined, the user can ultimately adjust exactly how similar one AST node is to another. For example, the user may consider `for` and `while` to be very similar, and so a comparison of these two nodes

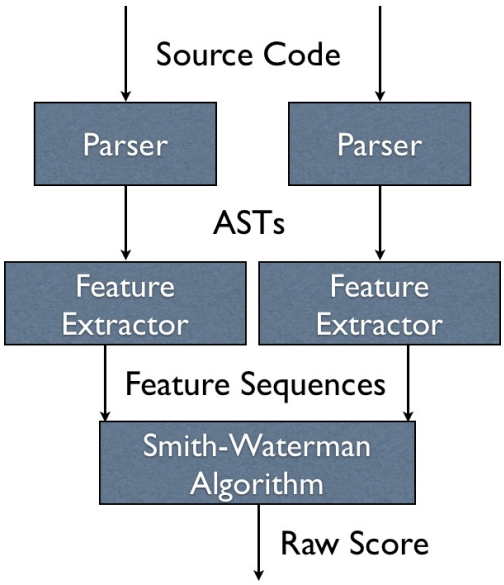


Fig. 1. Our technique, from input programs to output scores. The "Feature Extractor" is provided by the user.

would impart a high score. However, attempting to align very distinct AST nodes like variable declarations and arithmetic additions may be met with a low score. Once the alignments are complete, we use the raw scores as a means of comparing the similarity of any two ASTs. If two scores are high and similar, it is likely that the underlying ASTs are also quite similar, which can alert a human to look at the code.

III. Application and Evaluation

We have observed that in Scala the AST nodes corresponding to conditionals, match expressions, label definitions, value definitions, literals, identifiers, case definitions, and function/method calls tend to be shared between plagiarized code. As such, we developed a feature extractor which emits fairly generic features corresponding to these specific nodes. Additionally, we have observed that reordering functions is a common strategy to circumvent plagiarism detection. As such, before any features are emitted, we first sort all functions in the code by size and then process the ordered results, completely defeating this counter-detection strategy. We process nodes according to a BFS traversal, which we believe to be more difficult to circumvent than a DFS traversal. As for the scoring function, we found that scoring a direct feature match of 2 and a non-match with -1 was sufficient, though much more complex scoring mechanisms are possible.

We applied the resulting plagiarism detector to a large batch of course-related files which had previously been manually checked for plagiarism, complete with several known instances of plagiarism. The plagiarism detector correctly flagged all of the known cases as high-scoring, in addition to marking an additional pair of files. Upon further examination, it was revealed that these additional files were also plagiarized, though it was obfuscated severely due to reordering, superfluous comments, and the injection of no-op code. As such, the plagiarism detector has already proven itself more effective than a visual scan, and so we have adopted it for our own needs in teaching.

IV. Related Work

Winnowing [1] is a lossy, text-based technique employed by the extremely popular MOSS. While the techique is not specific is designed for plain text, the proprietary MOSS system is restricted to plagiarism detection of a few dozen languages, implying that MOSS goes beyond what is published. The inability of MOSS to handle Scala is ultimately what motivated this work.

In Zhang et al. [3], an AST is first converted to a LISP-like S-expression form, one function at a time. The Smith-Waterman algorithm [2] is subsequently used to compare the different S-expressions. Information from running the algorithm regarding common substrings is then gathered, which is, in turn, fed to a clustering algorithm to ultimately determine what is flagged. From Zhang et al. alone, it is unclear exactly how S-expressions are compared, though earlier work shows it to be a naive approach where the whole expression is always considered in plaintext [4]. This strategy largely degrades into a text-based strategy, as the names of different productions actually becomes significant. Moreover, this prevents any sort of customized feature extraction, which is a central novelty to our work. It is also unclear in Zhang et al. as to how functions are treated, as they are first decribed as separate components but later implied that they behave within a single unit. This makes it impossible to compare to our own strategy of sorting functions by size in our application to Scala. The plagiarism flagging in Zhang et al. is also quite dissimilar to our own. We flag results based on raw scores produced by the Smith-Waterman algorithm [2], whereas Zhang et al. flag based on a fairly ad-hoc method that looks at the lengths of common substrings. This makes the flagging method in Zhang et al. sensitive to short interleaving gaps in the source code, which would reduce the size of any common substrings. Our flagging method is far more resistant to the introduction of such short gaps.

V. Conclusions and Future Work

Our technique is fairly simple to implement, and it allows for a high degree of configuration. The application of our technique to Scala has already proven itself to be quite effective, and we currently use it internally for plagiarism detection. As for future work, we have a version that works with Prolog code which we are actively tuning.

References

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 76–85. [Online]. Available: http://doi.acm.org/10.1145/872757.872770

[2] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875

[3] L. ping Zhang and D. sheng Liu, "Ast-based multi-language plagiarism detection method," in *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, May 2013, pp. 738–742.

[4] L. Zhang, D. Liu, Y. Li, and M. Zhong, "Ast-based plagiarism detection method," in *Internet of Things*, ser. Communications in Computer and Information Science, Y. Wang and X. Zhang, Eds. Springer Berlin Heidelberg, 2012, vol. 312, pp. 611–618.

Analyzing Expert Behaviors in Collaborative Networks

Huan Sun*, Mudhakar Srivatsa†, Lance M. Kaplan‡, Shulong Tan*, Yang Li*, Shu Tao†, Xifeng Yan*
*University of California, Santa Barbara
{huansun, shulongtan, yangli, xyan}@cs.ucsb.edu
†IBM T.J. Watson Research Center
{msrivats,shutao}@us.ibm.com
‡U.S. Army Research Lab
{lance.m.kaplan.civ}@mail.mil

***Abstract*—Collaborative networks are composed of experts who cooperate with each other to complete specific tasks, such as resolving problems reported by customers. A task is posted and subsequently routed in the network from an expert to another until being resolved. When an expert cannot solve a task, his routing decision (*i.e.*, where to transfer a task) is critical since it can significantly affect the completion time of a task. In this work, we attempt to deduce the cognitive process of task routing, and model the decision making of experts as a generative process where a routing decision is made based on mixed routing patterns.**

In particular, we observe an interesting phenomenon that an expert tends to transfer a task to someone whose knowledge is neither too similar to nor too different from his own. Based on this observation, an *expertise difference* based routing pattern is developed. We formalize multiple routing patterns by taking into account both rational and random analysis of tasks, and present a generative model to combine them. For a held-out set of tasks, our model not only explains their real routing sequences very well, but also accurately predicts their completion time. Under three different quality measures, our method significantly outperforms all the alternatives with more than 75% accuracy gain. In practice, with the help of our model, hypotheses on how to improve a collaborative network can be tested quickly and reliably, thereby significantly easing performance improvement of collaborative networks.

I. Introduction

Collaborative networks are abundant in real life, where experts collaborate with each other to complete specific tasks. In service businesses, a service provider often maintains an expert network where service agents collaboratively solve problems reported by customers. Bugzilla[1] is a bug tracking system where software developers jointly fix the reported bugs in projects. In a classic collaborative network, upon receiving a task, an expert first tries to solve it; if he fails, the expert will route the task to another expert. The task is completed until it reaches an expert who can provide a solution.

Figure 1 shows a sample collaborative network with task routing examples. Task t_1 starts at expert A and is resolved by expert D, and task t_2 starts at expert D and is resolved by expert F. The sequences $A \rightarrow B \rightarrow C \rightarrow D$ and $D \rightarrow E \rightarrow F$ are called *routing sequences* of task t_1 and t_2 respectively. The number of experts on a routing sequence measures the completion time of a task. The average completion time of tasks signifies the efficiency of a collaborative network in problem solving: the shorter, the more efficient.

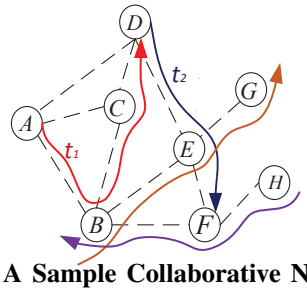


Fig. 1: A Sample Collaborative Network

When the number of experts in a collaborative network becomes large, to whom an expert routes a task significantly affects the completion time of the task. For example, in Figure 1, task t_1 can be directly routed to the resolver D from A. In this case the routing decision made by expert A is critical. Therefore, understanding how an expert makes a certain routing decision and detecting his routing behavioral patterns will help us identify the inefficiency of a collaborative network.

The task resolution problem in collaborative networks has been studied before. Shao *et al.*[2] propose a sequence mining algorithm to improve the efficiency of task resolution in IT service. Miao *et al.*[3] develop generative models and recommend better routing by considering both task routing sequences and task contents. In [4], Zhang *et al.* study the resolution of prediction tasks, which are to obtain probability assessments for a question of interest. All of these studies aim at developing automated algorithms that can effectively speed up a task's resolution process. However, they largely ignore human factors in real task routing. Take Figure 1 as an example. Why does expert A route task t_1 to B instead of D? Is it because he does not understand t_1 well, thus randomly distributing it to B, or he believes B has a better chance to solve it, or B has a better chance to find the right expert to solve it? Does expert A make more rational decisions than random decisions? While it is very hard to infer A's decision logic based on an individual task, it is possible to infer it by analyzing many tasks transferred and solved by A, B and D. In this work, we focus on analyzing real expert networks and try to understand experts' decision logic, *i.e.*, what kind of routing patterns an expert follows when deciding where to route a task. This understanding will help detect the inefficient spots in a collaborative network and give guidance to the management team to provide targeted expert training.

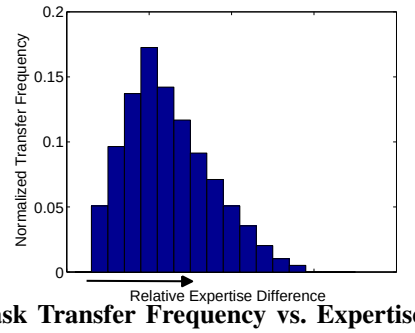


Fig. 2: Task Transfer Frequency vs. Expertise Difference.

After analyzing thousands of tasks in an IBM service department, we recognize that in many cases, an expert might not route a task to the *best* candidates (in terms of the possibility to solve the task), especially when the task is far beyond his expertise. Instead, the task is transferred to an expert whose speciality is between the current expert and the best candidates. This routing pattern is clearly indicated by Figure 2. Figure 2 plots the histogram of expertise difference $\frac{\|E_A - E_B\|}{\|E_A\|}$, which is calculated when expert A transfers a task to B. E_A represents the expertise of A and is automatically learnt based on A's task resolution records. It is observed that an expert tends to transfer a task to some expert whose expertise is *neither* too similar to *nor* too different from his own. This phenomenon can be explained as follows: An expert is less likely to transfer a task to another expert whose expertise is very similar, given that the current expert already fails to resolve the task. On the other hand, if the expertise of two experts are very different, they might actually specialize in quite different domains; therefore, an expert might not be clear about the other's speciality and few tasks would be transferred between them.

Inspired by the above observation, we introduce a routing pattern describing the general trend of expert A transferring a task to B, based on the *expertise difference* between A and B. Apart from this routing pattern, another two are also formalized. Specifically, when an expert finds there are five candidates to dispatch a task to – all of them can solve the task, who is he going to contact? A straightforward approach is to randomly pick one. An alterative is to look at the capacity of these candidates and route more tasks to an expert who can process more tasks. An expert could follow a certain pattern when deciding where to transfer a task. Different experts might adopt each routing pattern to a different degree and demonstrate different routing behavioral characteristics. This study is going to infer the routing patterns as well as experts' preferences over them, from the historical routing data, and finally give insightful analysis of experts' performance in a collaborative network.

The technical contributions of this work are three-fold: First, to the best of our knowledge, we make the first attempt to analyze the routing behaviors of experts in a collaborative network in a large-scale, quantitative manner. We present a general framework to model the decision making and cognitive process of experts, and instantiate the framework with multiple routing patterns potentially followed by an expert. A generative model is then presented to model experts' routing decisions as a result of mixed routing patterns. After trained on a historical task set, the model can uncover experts' underlying decision logic and explain real routing sequences in a held-out testing set very well.

Second, our analytical model can accurately predict the completion time of a task before actually routing it in the network. On the one hand, we verify that our analysis of experts' routing decision making reflects the real one, in the sense that a task navigated according to our model shall be completed in a similar time as in the real situation. On the other hand, estimating task completion time is important itself, because an accurate estimate of completion time can provide early signals on the "difficulty" or "abnormality" of a task, and managers can allocate more resources and take early actions to deal with such tasks and shorten customer waiting time.

Third, it is usually expensive, if not impossible, to alter real-world collaborative networks for hypothesis testing, *e.g.*, providing more training to a few critical but inefficient experts or changing network structures for better performance. Since our analytical model has shown similar characteristics to the real human routing in collaborative networks, it can be used to conduct virtual hypothesis testing. Through case studies, we discuss how to utilize our model to optimize collaborative networks.

II. Conclusion

In this paper, we model the decision making and cognitive process of an expert during task routing in collaborative networks. A routing decision of an expert is formulated as a result of a generative process based on multiple routing patterns. We formalize each routing pattern in a probabilistic framework, and model experts' routing decision making through a generative model. Our analytical model has been verified that it not only explains the real routing sequence of a task very well, but also accurately predicts a task's completion time in the current collaborative network. In comparison with all the alternatives, our method improves the performance by more than 75% under three different quality measures. We also demonstrate that our model can provide guidance on optimizing the performance of collaborative networks.

Acknowledgment

This research was sponsored in part by the Army Research Laboratory under cooperative agreements W911NF-09-2-0053, NSF IIS 0917228, and 0954125. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

References

[1] "Bugzilla: http://www.bugzilla.org/."

[2] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis, "Efficient ticket routing by resolution sequence mining," in *SIGKDD*. ACM, 2008, pp. 605–613.

[3] G. Miao, L. E. Moser, X. Yan, S. Tao, Y. Chen, and N. Anerousis, "Generative models for ticket resolution in expert networks," in *SIGKDD*. ACM, 2010, pp. 733–742.

[4] H. Zhang, E. Horvitz, Y. Chen, and D. C. Parkes, "Task routing for prediction tasks," in *AAMS-Volume 2*. IFAAMS, 2012, pp. 889–896.

SLQ: A User-friendly Graph Querying System

Shengqi Yang Yinghui Wu Huan Sun Xifeng Yan
University of California Santa Barbara
{sqyang, yinghui, huansun, xyan}@cs.ucsb.edu

***Abstract*— Querying complex graph databases such as knowledge graphs is a challenging task for non-professional users. In this work, we present SLQ, a user-friendly graph querying system enabling schemaless and structureless graph querying, where a user need not describe queries precisely as required by most databases. SLQ system combines searching and ranking: it leverages a set of transformation functions, including abbreviation, ontology, synonym, etc., that map keywords and linkages from a query to their matches in a data graph, based on an automatically learned ranking model. To help users better understand search results at different levels of granularity, it supports effective result summarization with "drill-down" and "roll-up" operations. Better still, the architecture of SLQ is elastic for new transformation functions, query logs and user feedback, to interactively refine ranking model and searching. SLQ significantly improves the usability of graph querying. This demonstration highlights (1) SLQ can automatically learn an effective ranking model, *without* assuming manually labeled training examples, (2) it can efficiently return top ranked matches over noisy, large data graphs, (3) it can summarize the query matches to help users easily access, explore and understand query results, and (4) its GUI can interact with users to help them construct queries, explore data graphs and inspect matches in a user-friendly manner.**

Online demo: slq.cs.ucsb.edu

I. Introduction

Graph querying is widely adopted to retrieve information from emerging graph databases, *e.g.,* knowledge graphs, information and social networks. Given a query, it is to find reasonable top answers (*i.e.,* matches for the query) from a data graph. Searching real-life graphs, nevertheless, is not an easy task especially for non-professional users. (1) Either no standard schema is available, or schemas become too complicated for users to completely possess. (2) Graph queries are hard to write and interpret. Structured queries (*e.g.,* XQuery [2] and SPARQL [3]) require the expertise in complex grammars, while keyword queries [7] can be too ambiguous to reflect user search intent. Moreover, most of these methods adopt predefined ranking model [3], which is barely able to bridge the gap between the queries and the true matches. (3) Moreover, it is a daunting task for users to inspect a large number of matches produced from querying large-scale heterogeneous graph data.

Example I.1: Consider a query asking "tell me about the history of Jaguar in America". The query can be presented as either a keyword query "Jaguar history America", or a small graph in Fig. 1. To find answers for such a simple query is, nevertheless, not easy. (1) A keyword *e.g.,* "Jaguar" may not have identical matches, but instead can be matched with entities that are semantically close, *i.e.,* luxury cars or animals. *How to find matches that are semantically related to the query?* (2) A large number of possible answers can be identified by various matching mechanisms. For example, "Panthera Onca" is closely related with "Jaguar" as its scientific name, while "Jaguar XK" is another match obtained simply by string transformations. *Which one is better?* A ranking model should be employed and tuned with or without manual tuning effort. (3) There are a large number of good results, *e.g.,* different species related to Jaguar (result 1 and result 2), or various car prototypes (result 3 to result k). *How to help users better understand results without inspecting them one by one?* This kind of complexity contrasts to the impatience of web users who are only interested in finding good answers in a short time. □

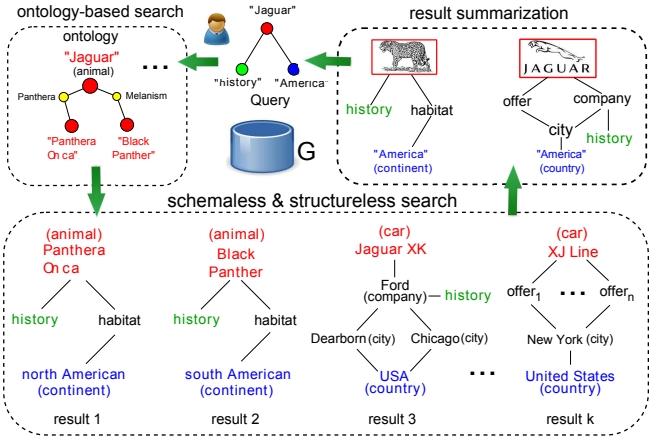


Fig. 1. Searching a knowledge graph

To answer these questions, we propose SLQ, a novel graph querying system for schemaless and structureless querying. (1) To better understand search intent, SLQ interprets queries with external ontologies to find semantically close matches. (2) It automatically supports multiple mapping functions, namely, *transformations*, *e.g.,* synonym, abbreviation, ontology, to identify reasonable answer via learning to rank, and works with both (a) a cold-start strategy that requires no manual effort for system tuning, and (b) a warm-start strategy to adjust the ranking model with available user feedback and query logs. (3) To help users better understand results and refine queries, it supports concise result summarization. Users may inspect small summaries, and then decide to (a) drill-down for detailed matches, or (b) interactively refine queries with interesting summaries. To the best of our knowledge, these features are not seen before in any previous graph querying systems (*e.g.,* [3], [5], [4]).

SLQ system is among the first efforts of developing a unified framework for schemaless and structureless querying. Designed to help users access complex graphs in a much easier and powerful manner, it is capable of finding high-quality

matches when structured query languages do not work. We next introduce SLQ in more details.

II. Schemaless and Structureless Querying

In this section we mainly present two key components in SLQ, offline ranking model learning and online query processing. As remarked earlier, SLQ does not require a user to describe a query precisely as required by most search applications. To render this high flexibility to the users, in the core of the framework is a mechanism of *transformation*: given a query, the query evaluation is conducted by checking if its matches in a graph database can be reasonably "transformed" from the query through a set of transformation functions.

Transformations. SLQ adopts transformation defined on the attributes and values of both nodes and edges of the query. A query node (or edge) has a match if it can be converted to the latter via a transformation. For example, a query node "IBM" shall be matched to a node with label "International Business Machines." The table below summarizes several common transformations supported by SLQ.

Category	Transformations
String	First token, Last token, Abbreviation, Prefix, Acronym, Drop word, Bag of words
Semantic	Ontology, Synonym (WordNet [1])
Numeric	Unit conversion, Date gap, Date conversion, Numeric range
Topology	Distance, Labeled path

Better still, SLQ exposes a generic interface abstracted for any transformation. New transformations complying with the interface can be readily plugged in.

Example II.1: As in Fig. 1, SLQ captures the matches for the keyword "Jaguar" as follows. (1) The matches "Panthera Onca" and "Black Panther" can be identified by ontology transformation. (2) The match "Jaguar XK" can be matched by "First token", a String transformation. (3) The match "XJ Line" can be captured by "Bag of words", indicating "Jaguar" appears in its text description. For edge ("Jaguar", "America"), an topology transformation "Distance" maps it to a path from "Panthera Onca" to "north America" in result 1. □

Matching Quality Measurement. To measure the quality of the matches induced by various transformations, SLQ adopts a ranking function of *weighted transformations*. The function incorporates two types of score functions: node matching and edge matching score function. Each score function aggregates the contribution of all the possible transformations with corresponding weights. More specifically, by harnessing the probabilistic graphical model, we define the ranking function with *Conditional Random Fields* [6], since it satisfies two key considerations: using training data, its *formulation* could optimize the weights of the transformations for good ranking quality; the *inference* on the model provides a mechanism to search top-k matches quickly.

Ranking model learning. A key issue is how to select a ranking function by determining reasonable weights for the transformations. Simple approaches, *e.g.,* assigning equal transformation weights or calibrating the weights by human effort, are clearly not the best strategies. Instead, SLQ automatically figures out the weights from a set of *training instances* with machine learning techniques. Each instance is a pair of a query and one of its relevant (labeled as "good") answers. The learning aims to identify for each transformation a weight, such that if applied, the model ranks the good answers as high as possible for each instance.

SLQ begins with a cold-start stage where no manual effort is required for instances labeling, and can be "self-trained" in the warm-start stage, using user feedback and query logs. Once the transformation weights are determined, SLQ readily use the ranking model to determine good matches.

Query Processing. SLQ efficiently finds top matches as follows. (1) To fast extract matches, it leverages *approximate inferencing* [6] for graphical models. It treats a query as a graphic model and the matches as assignment to its random variables (query nodes). By iteratively propagating messages among the nodes in the graphical model, the inference can identify top assignments (matches) that maximize the joint probability (ranking score) for the model. This technique dramatically reduces the query processing time, with only small loss of match quality (less than 1% in our validation). (2) SLQ further reduces the matches to be inspected. It constructs a small "sketch" of a data graph that can be directly queried. The sketch guarantees that the ranking score of each match it provides estimates an *upper bound* of a set of matches in the data graph. Putting (1) and (2) together, SLQ computes good matches by performing a two-level inferencing. It iteratively extracts "upper-level" matches from the sketch and then drill down the match to find more accurate "lower-level" ones, until the lowest ranked lower-level match is better than a next upper-level match. Following this, SLQ avoids unnecessary computation for the low quality matches.

III. Conclusion

In this work, we developed a novel searching framework SLQ. Surrounding this new query paradigm, there are a few emerging topics worth studying in future, *e.g.,* comparison of different probabilistic ranking models, compact transformation-friendly indices, and distributed implementation.

References

[1] Wordnet. *wordnet.princeton.edu.*

[2] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001.

[3] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *WWW*, 2011.

[4] D. Mottin, M. Lissandrini, V. Yannis, and T. Palpanas. Exemplar queries: Give me an example of what you need. In *VLDB*, 2014.

[5] J. Pound, I. F. Ilyas, and G. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, 2010.

[6] C. Sutton and A. McCallum. An introduction to conditional random fields for relational learning. *Introduction to statistical relational learning*, 93:142–146, 2007.

[7] H. Wang and C. Aggarwal. A survey of algorithms for keyword search on graph data. *Managing and Mining Graph Data*, pages 249–273, 2010.

Advocacy Citizen Journalism and their Participatory Audience

Saiph Savage
4 Eyes Lab, UCSB
Email:saiph@cs.ucsb.edu

Andres Monroy-Hernandez
Microsoft Research
Email:amh@microsoft.com

***Abstract*—This paper presents a descriptive analysis of a popular Facebook page used to circumvent the information blackout in certain regions of Latin America. The page has more than 170,000 fans respectively. This work presents a descriptive analysis of the page and its audience. We examine the full 6,000 posts by the page's administrators, and more than 108,000 comments by their fans. We extract themes, post frequency, and its relationship with offline events and public figures. We argue that the page presents a novel form of participatory propaganda online that helps people make sense of what is taking place in their neighborhoods and country, and, in some cases, foster offline collective action. We conclude by discussing possible applications of these findings for the design of civic engagement technologies.**

I. Introduction

In recent years, journalists across the world have been threaten or kidnapped for covering certain topics.These attacks from organized crime, and even government officials force many journalists to censor themselves. Previous research has shown how this information vacuum motivates people to use social media to report, in real-time, the location and nature of important events taking place in their cities and towns [1]. The work has also documented the emergence of citizen "news correspondents" who use their Twitter visibility to receive and curate news reports of what happens in their communities.

In addition to Twitter, citizen reporters are using other platforms, such as Facebook and YouTube for their news reporting. This paper studies how citizen reporters use the open platform of social media, and how people react and take part in their content. One angle we are particularly interested in analyzing is what content from citizen reporters obtains the most participation from their audience, and what is the profile of the audience members who participate the most in the content. We attempt to address the following research questions using data from a Facebook page related to citizen reporters:

1) What type of content are citizen reporters sharing online, and how do their audience react to such content?
2) What type of content from citizen reporters attracts the most newcomers, i.e., new audience members?
3) What are the traits of the audience members who participate the most in the content from citizen reporters?
4) What are the characteristics of the most popular content from citizen reporters?

We examine the patterns of communication between audiences and the Facebook page in question, i.e., the citizen reporters who control and administer the page. We apply human coding and quantitative analysis to obtain a deeper understanding of how audiences participate in content linked to citizen reporters.

We reveal that citizen reporters are creating an interactive digital space where citizens are explained how the complicity of the political situation in their towns and neighborhoods relates to them personally. Through their page citizen reporters even suggest direct actions audiences can conduct to take part in collective efforts that could transform their towns. The page is also harvesting a space where audiences are helped to maintain safety and are provided social support, where they can discuss and grieve personal family problems.

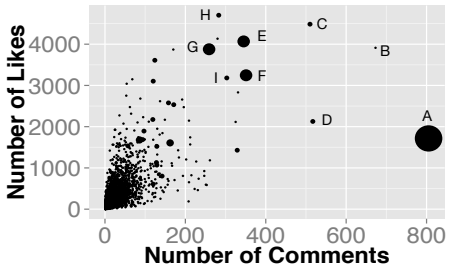


Figure 1. Number of likes and comments for all posts, size of a point is directly correlated with the number of reshares of a post. The most popular content was published during major offline events.

Table I
Data.

Number of Posts	6,901
Number of Fans	158,000
Number of Participants (active audience size)	127,374
Number of Comments	108,967
Number of Post Likes	1,481,008
Number of Reshares	364,660

We find that the most popular content from citizen reporters emerged during major offline events, e.g., street manifestations against the government. It is also during these times the page manages to recruit the most newcomers. We also uncover that these citizen reporters have two main types of highly active audience members: those who use their page to discuss crime; and those who use the page to vent about the government. From a practical outlook, our study is important to the development of technology for political participation.

Through our case study of these citizen reporters, we obtain a glimpse of three possible emerging phenomena: 1. how self-organized citizen reporting looks like online, 2. collaborative commentary and reflection of community issues, 3. scaffold in collective action.

II. Methods

Our study is based on data from a facebook page linked to citizen reporters downloaded via the Facebook API. The downloaded data includes posts, comments, likes, and reshares, and spans a time

period from the page's start date, August 14, 2013, to May 7, 2014. Details of our dataset are shown in Table I.

To respond our research questions, we characterize the page's content and the audience's reactions. More specifically, for R1 we obtain an overview of the posting behavior of the page admins (citizen reporters) and its audience's participatory activity through time. To discover the topics we use a grounded approach with Odesk workers to thematize the 6,901 posts. The purpose of our coding is to obtain a descriptive assessment of the posts; for R2 we study the nature of the pages messages which have the largest number of active newcomers; for R3 we characterize the most active audience members; and finally for R4 we focus on characterizing the pages most popular content.

In the following, we describe some of the attributes we analyze to characterize the pages content and the behavior of its most active audience members:

Profiling Most Active Audience Members

We profile the page's most active audience members to begin to uncover the traits of the people most dedicated to the reports made by citizen journalists. We consider that the most active audience members are the ones who produce the most comments. We focus on comments because they typically take more time to produce than a "like," and can provide more hints about how dedicated a person is in a page or even a political movement. We use comments as a window to study how active audience members are in the page.

We identify the most active participants by finding those individuals whose number of comments deviate by three times the standard deviation (normal procedure to identify outliers.) We then characterize the most active audience members in terms of the type of public figures they mention in their comments (such analysis helps signal what a person cares about most.) We calculate the degree with which a participant mentions a certain type of public figure as follows:

$$P(a, G) = \sum_{w \in \mathcal{W}} P(a)P(w|a)P(G|w), \quad (1)$$

where $P(a, G)$ is the probability audience member a mentions a public figure related to group G (type of public figure) given her comments; $P(a) = n^{-1}$, is the probability of selecting audience member a; n is the total number most active audience members; $P(w|a) = m^{-1}$ is the probability of a particular word w appearing in audience member a's comments; m is the total number of words that audience member a has used; $P(G|w) = 1$, when the word w is relevant to the group G (i.e., the word w references a public figure that belongs to group G,) or zero otherwise.

For each active audience member we calculate for all groups her $P(a, G)$, and form a vector representing how much the person has mentioned different groups of public figures in her comments. The vector's size corresponds to the different groups of public figures, in this case the size is 4 (this corresponds to the number of different public figures that were discovered in our data.)

We use these vectors to cluster audience members who mention the same type of public figures and to a similar degree. For instance, people who only mention public figures related to the government might be grouped together. We use mean shift algorithm to group together similar vectors, and discover clusters of people. We decided to use mean shift algorithm because it is based on nonparametric density estimation, and therefore we do not need to know the number of clusters beforehand (unlike K-means.) We let mean shift algorithm discover the clusters from our data. We use the clusters to study the different behavior of the most active audience members.

Most Popular Content

In Figure 1, we present a scatter plot illustrating the number of likes, comments, and reshares of each Facebook post from the citizen reporters. The Xaxis shows the number of comments the post received; the Yaxis, the number of likes. The size of the circle relates to the number of reshares; the larger the circle, the more the post was reshared. We defined VXM's most popular content as the posts whose total number of likes, comments, and reshares deviated by three times the standard deviation. We labeled the most popular posts alphabetically and examined a few of their characteristics.

III. Discussion

Through our case study of a Facebook page linked to citizen reporters we witnessed how an online space linked to citizen reporters is starting to foster a participatory culture [3] by providing: 1. low barriers for participatory engagement to audience members; 2. informal mentorship; 3. social connection and 4. support for participation.

Over and over we saw how the audience helped contribute a better picture of what was taking place in certain towns. We believe there is value in creating tools that let citizen reporters better collaborate with their online audience. Perhaps it is about designing tools that let citizen reporters visualize [7] and understand [5] the traits of their audience to dispatch journalist tasks for them. For this purpose it might make sense to adopt techniques used to crowdsource other creative tasks with citizens, e.g., a crowdsourced orchestra [6]. Within these types of systems, it might also be relevant to model the spatial temporal constraints of audience members and willing citizen reporters [4] with the purpose of better dispatching who will report about a certain event that is taking place. For instance, a person on a car could provide reports not available to citizens walking and vice-versa.

References

[1] Monroy-Hernandez, Andres, et al. "The new war correspondents: The rise of civic media curation in urban warfare." CSCW'13.

[2] Mark, G. J., Al-Ani, B., and Semaan, B. Resilience through technology adoption: Merging the old and the new in iraq. CHI09.

[3] Jenkins, H. Confronting the challenges of participatory culture: Media education for the 21st century. Mit Press, 2009.

[4] Savage, N.S, Baranski, M.,Chavez,N.E., Hollerer T. I'm feeling LoCo: A Location Based Context Aware Recommendation System. Proc. 8th International Symposium on Location-Based Services, Lecture Notes in Geoinformation and Cartography,Springer.

[5] Savage, Saiph, et al. "Visualizing Targeted Audiences." COOP 2014-Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France). Springer International Publishing, 2014.

[6] Savage, S., Chavez, N. E., Toxtli, C., Medina, S., Alvarez Lopez, D., & Hollerer, T. (2013, February). A social crowd-controlled orchestra. In Proceedings of the 2013 conference on Computer supported cooperative work companion (pp. 267-272). ACM.

[7] Forbes, Angus Graeme, Saiph Savage, and Tobias Hollerer. "Visualizing and verifying directed social queries." IEEE Workshop on Interactive Visual Text Analytics. Seattle, WA. 2012.

Collaborative Interfaces for Designing Optical Fiber Networks

Hugo Leon
Universidad Nacional Autonoma de Mexico

Julio Cruz
Universidad Nacional Autonoma de Mexico

Saiph Savage
4 Eyes Lab, UC Santa Barbara

Norma Elva Chavez
Universidad Nacional Autonoma de Mexico

Tobias Hollerer
4 Eyes Lab, UC Santa Barbara

***Abstract*—The deployment of optical fiber networks is increasing in the world. However, designing these networks can be challenging, especially because there are currently few specialized tools for designing such networks. In this paper we discuss interfaces focused on helping people design optical fiber networks. Our tool is equipped with interaction mechanisms through which people can automate many of the tasks related to designing networks. We also give people different data visualizations that enable them to have a better overview and understanding of the physical space where their network design will be placed. We also leverage the knowledge of other designers to help network designers collaborate and build off the work of others, while still leaving room for their own unique creations. This work contributes to the design of tools for sketching and modeling telecommunication networks.**

I. Introduction

In the last decade we have witnessed a growth in the number of optical fiber networks that exist in people's homes and organizations [2]. This increase has mainly been because people want faster Internet communication to access: video-based multimedia, rapid peer-to-peer file transfer, high definition multimedia online gaining, among other services. Similarly, the increment in personalized services that know the end-user's every need through constant monitoring, has also increased network traffic. Optical fiber networks are now found in a variety of places, such as homes, government organizations, private companies, among other edification [3]. Despite its increase, little attention has been paid to creating specialized tools that facilitate the design of fiber optical networks. Most research and industry tools, including those tailored for traditional copper wires, have focused on the management and documentation of networks, or helping people visualize network vulnerabilities [1,5,6]. Yet, specialized network design tools could streamline design work, reduce network costs, and facilitate troubleshooting in the design stage and not at the stage of installation and implementation of the network.

In this paper we present: FiberKraft, a computer aided design (CAD) tool with a graphic environment that is tailored to assist people in the design of optical fiber networks for homes and businesses. Our tool helps designers to: 1) spatially organize the physical space their network will cover; 2) save time designing the telecommunication and urban infrastructure of their network; 3) quantify the amount of material their network requires; d) and obtain ideas from a specialized crowd on how to design the network of a certain home or organization.

II. System Design

FiberKraft was developed in C#, in conjunction with AutoCAD 2013 which works as an external program execution. We choose this configuration, so that anyone acquainted with AutoCAD could potentially start using FiberKraft and avoid having to learn a new

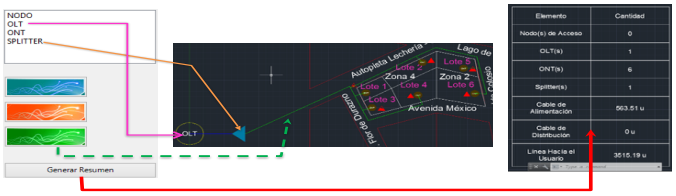


Figure 1. Screenshot of the insertion of elements FTTH and creation of a summary table.

system. The interface of our tool is composed of 2 main parts: Workplace Definition and Sketching Area.

Workplace Definition: This component lets designers work on a new project that contains all the elements needed for designing a network. The end project is loaded in AutoCAD. The designer can set values such as the address of the physical space where the network will be placed, or date. We use the geographical location to help designers potentially find other network designs in the area. We believe designers can benefit by exploring the networks designs of others, especially if the designs are from homes near the one the designer is currently working on, as they might have similar structures and geographical problems.

Sketching Area: This component lets designers draw the physical space surrounding the network they are designing, and explore different network designs. We let people share the physical spaces and networks they have sketched, as well as use the sketches of others. FiberKraft tries to make people leverage the work of the crowd to help in the design of their own the fiber networks. To start their sketches, designers are given the option of either entering a drawing in the sketching area by inserting a DWG file from any computer path, or directly drawing blocks that represent different neighborhoods. Four vertices define the geometry of one block and when the person finishes drawing the block, the information is saved in the internal data base of our application. The block basically helps to define a neighborhood with its houses and roads. The person can also add metadata to the neighborhood she drew. The metadata usually holds details about the neighborhood, such as date it was founded etc. People can use the metadata and stored information to search for neighborhoods that are similar to the ones they are working on. This enables people to see how others have structured the network under similar conditions, and get ideas of how they will start their own design. We also provide the option of splitting neighborhoods into lots. Lots are basically residential buildings that can have (or not) already an optical fiber network. Examples of lots are business buildings, malls, apartment houses, among others. Designers can also draw lots with four vertices. Designers can set values for each lot such as: address, number of telephone lines and lot type (e.g., mall). FiberKraft automatically calculates the area that defines the lot, thus helping the designer

to better calculate the costs of putting a fiber optical network in a particular building. Internally our tool differentiates between the geometry of blocks (neighborhoods) and lots (buildings.) We allow people to zoom in and out to easily view the total number of neighborhoods and buildings they have drawn. This can help people get a better overview of the physical space where they will design the network. The designer is also empowered to insert urban infrastructure into the sketching area such as: pipelines, trees, poles, terminals, among other elements.

We provide a default set of urban structures that people can use in their sketches These elements facilitate network designers tasks, as in most cases the person had to invest time in manually drawing the elements herself. Our tool also empowers designers to insert telecommunication infrastructure, specifically those related to the infrastructure of a Fiber to the X (e.g., home) network with PON (passive optic network) architecture such as: optical splitters, optical line terminals (OLT's), optical network terminations (ONTs) etc. Designers can also select the types of wires that will be drawn from the fiber network such as the feeder cabling, distribution cabling and the users line. People can make this selection by simply setting the insertion point and scale for each item. Additionally, we provide mechanisms to help automate this selection process: the person has to simply draw a polyline of n vertices that represents the path of the cable and the type of cable. Overall, our interface provides mechanisms to save designers time in repetitive tasks. Figure 1 shows a palette that contains telecommunication elements that can be inserted onto the sketching area. The counting for each telecommunication element inserted and the total length of each kind of cable is presented in the summary table shown in the right. The table helps designers have an overview of the costs of their current network design. Aside from the default urban and telecommunication infrastructure provided, designers can also define their own infrastructures and share it with the crowd. People can easily build off the infrastructures others have defined. This type of sharing lets our tool be always up to date with the different infrastructures as it is driven by the people.

FiberKraft also provides designers with different visualizations and cues to obtain better perspectives of the physical space and infrastructure where their network will be placed. As mentioned previously we provide people with a table that lets them have an overview of the number of urban and telecommunication infrastructures. Additionally, we give designers tree views where they can visualize the buildings they have defined in different neighborhoods. We show the data related to the buildings in a hierarchical form to facilitate the visualization of the urban infrastructures and telecommunication elements in the building, and within the neighborhood. Similarly, people can view the network designs of others, letting them better inspect the physical spaces others have worked on, and network designs proposed for those spaces. We believe this can help people obtain ideas of how they will design their own network, and avoid some of the problems others have gone through.

III. Conclusions

The installation of Fiber optical networks is increasing in the world. It is therefore necessary to have tools that facilitate the design of these networks for a variety of different physical locations. In this paper we presented a specialized tool, FiberKraft, for designing fiber optical networks. Our tool strives to help network designers in three main ways: 1) save designers time in trivial tasks, such as drawing network infrastructure or doing summary tables; 2) provide ways to visualize their network design from different perspectives; 3) facilitate integrating the crowd to provide ideas and keep the tool updated. Our research contributes to the design of novel interfaces for designing networks. It also lets us explore how tools for conducting specialized tasks can be enhanced by providing specialized interaction mechanisms and leveraging the crowd's work. In the future, we plan on exploring how different interface designs affect the collaborations between designers. We wish to explore how contextual cues about a neighborhood and its citizens, e.g., type of buildings, roads and citizen mobility [7], can facilitate network design. We would also like to adopt techniques from visualizing online users [8], [10] to help designers visualize the traits of other network designers. This could help them identify people with whom they could collaborate with; or people whose designs they could trust more. Designing optical networks can involve some creativity. It might then make sense to adopt techniques used to crowdsource creative tasks, e.g., a crowdsourced music making [9], [4].

References

[1] Miguel Alberto Planas, Doug Edward Talbott, Network management graphical & user interface, US Patent, 2000

[2] Koonen, Ton. "Fiber to the home/fiber to the premises: what, where, and when?" Proceedings of the IEEE 94.5 (2006): 911-934.

[3] Lin, Chinlon, ed. Broadband optical access networks and fiber-to-thehome: Systems Technologies and Deployment Strategies. John Wiley & Sons, 2006.

[4] C. Jette, K. Thomas, J. Villegas, and A. G. Forbes. Translation as technique: Collaboratively creating an electro-acoustic composition for saxophone and live video projection. Proceedings of the International Computer Music Conference (ICMC), 2014.

[5] Carl T. Madison, Jr., Richard C. Flathers, Configurable graphical user interface useful in managing devices connected to a network, US Patent, 1999

[6] Marshall Strickland, Rob Strickland, Graphical user interface for customer service representatives for subscriber management systems, US Patent, 1999.

[7] Savage, N.S, Baranski, M.,Chavez,N.E., Hollerer T. I'm feeling LoCo: A Location Based Context Aware Recommendation System. Proc. 8th International Symposium on Location-Based Services, Lecture Notes in Geoinformation and Cartography,Springer.

[8] Savage, Saiph, et al. "Visualizing Targeted Audiences." COOP 2014-Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France). Springer International Publishing, 2014.

[9] Savage, S., Chavez, N. E., Toxtli, C., Medina, S., Alvarez Lopez, D., & Hollerer, T. (2013, February). A social crowd-controlled orchestra. In Proceedings of the 2013 conference on Computer supported cooperative work companion (pp. 267-272). ACM.

[10] Forbes, Angus Graeme, Saiph Savage, and Tobias Hollerer. "Visualizing and verifying directed social queries." IEEE Workshop on Interactive Visual Text Analytics. Seattle, WA. 2012.

Comparing Different Cycle Bases for a Laplacian Solver

Erik G. Boman* Kevin Deweese† John R. Gilbert†

1 Kelner et al.'s Randomized Kaczmarz Solver

Solving linear systems on the graph Laplacian of large unstructured networks has emerged as an important computational task in network analysis [7]. Most work on these solvers has been on preconditioned conjugate gradient (PCG) solvers or specialized multigrid methods [6]. Spielman and Teng, showed how to solve these problems in nearly-linear time [8], later improved by Koutis et al. [5] but these algorithms do not have practical implementations. A promising new approach for solving these systems proposed by Kelner et al. [4] involves solving a problem that is dual to the original system.

The inspiration for the algorithm is to treat graphs as electrical networks with resistors on the edges. The graph Laplacian is defined as $L = D - A$ where D is a diagonal matrix containing the sum of incident edge weights and A is the adjacency matrix. For each edge, the weight is the inverse of the resistance. We can think of vertices as having an electrical potential and net current at every vertex, and define vectors of these potentials and currents as $\vec{v}$ and $\vec{\chi}$ respectively. These vectors are related by the linear system $L\vec{v} = \vec{\chi}$. Solving this system is equivalent to finding the set of voltages that satisfy the currents. Kelner et al.'s SimpleSolver algorithm solves this problem with an optimization algorithm in the dual space which finds the optimal currents on all of the edges subject to the constraint of zero net voltage around all cycles. They use Kaczmarz projections [3][9] to adjust currents on one cycle at a time, iterating until convergence. They prove that randomly selecting fundamental cycles from a particular type of spanning tree called a "low-stretch" tree yields convergence with nearly-linear total work.

*Sandia National Laboratories, Sandia is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000. eboman@sandia.gov

†UC Santa Barbara Dept. of Computer Science, Supported by Contract #618442525-57661 from Intel Corp. and Contract #8-48252526701 from the DOE Office of Science. kdeweese@cs.ucsb.edu, gilbert@cs.ucsb.edu

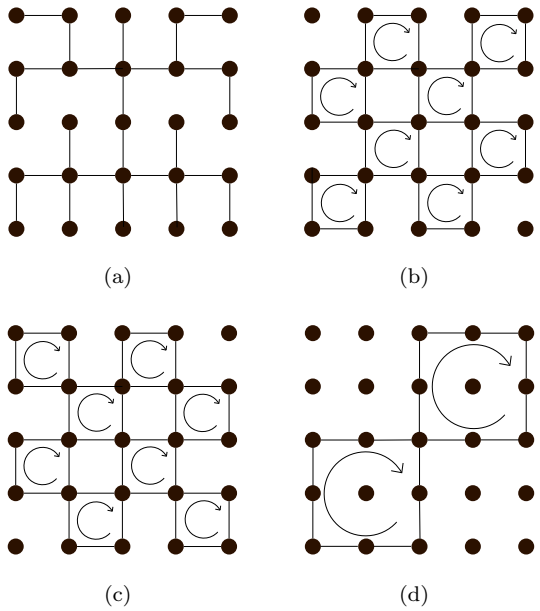


Figure 1: Grid Cycles

2 Choosing the Cycle Basis

We examine different ways to choose the set of cycles and their sequence of updates with the goal of providing more flexibility and potential parallelism. Our ideas include the following.

- Provide parallelism by projecting against multiple edge-disjoint cycles concurrently.
- Provide flexibility by using a non-fundamental cycle basis.
- Provide flexibility by using more (perhaps many more) cycles than just a basis.
- Accelerate convergence by varying the mixture of short and long cycles in the updating schedule.

Sampling fundamental cycles from a tree will require updating several potentially long cycles which will not be edge-disjoint. It would be preferable to update edge-disjoint cycles as these updates could be done in parallel. Instead of selecting a cycle basis from a

spanning tree, we will use several small, edge-disjoint cycles. We expect updating long cycles will be needed for convergence, but we consider mixing in the update of several short cycles as they are cheap to update and have more exploitable parallelism. These cycles can then be added together to form larger cycles to project against in a multigrid like approach.

An example of these cycles can be seen on the 5 by 5 grid graph in Figure 1. Figure 1(a) shows a spanning tree in which each cycle is determined by an edge not in the tree. The smallest cycles of a non-fundamental scheme are shown in Figures 1(b)(c). All the cycles in each of these two figures are edge-disjoint and can be updated in parallel. They can also be summed together as in Figure 1(d).

3 Preliminary Experiments and Results

We performed our initial experiments on grid graphs of various sizes. We used a non-fundamental set of cycles with a hierarchical ordering. The smallest set of cycles are updated. Then the cycles are coarsened and the next level of cycles are updated. This is done until reaching the perimeter cycle before resetting back to updating the smallest cycles. We also implemened the SimpleSolver algorithm in Matlab, except that we used a random spanning tree for sampling instead of a low-stretch tree. We also haven't implemented a clever data structure Kelner et al. use to quckly update edges. We also compared our results to PCG with Jacobi.

The metric we choose for comparison is the total number of edges updated, or matrix elements touched in CG. We can see the total work measured in edges updated in Table 1. Also shown in the table is an estimated potential parallelism using the work-span model [10]. The span, or critical path length, is the maximum number of edges that would have to be updated by a single processor if we can split the work over infinitly many processors.

Grid Size (Vertices)	25	289	4,225
Fundamental Cycles Work	8K	1.4M	296M
Alternative Cycles Work	1K	.08M	4M
Alternative Cycles Span	.5K	8.4K	105.8K
PCG Work	1K	.09M	5M

Table 1: Edges Updated

4 Conclusions and Future Work

Our preliminary experiments show that choosing a non-fundamental set of cycles can save significant work compared to a fundamental cycle basis, and can be at least competitive with PCG.

We are exploring ways to find a non-fundamental cycle basis of more general graphs; one challenge is how best to find large sets of short edge-disjoint cycles for parallelism. Our ideas for cycle finding include shortcuts to the spanning tree cycles and growing small cycles locally around vertices and edges. We also plan to make a rigorous comparison with several other preconditioned CG methods, including incomplete Cholesky and support-graph techniques.

We note that any of these graph Laplacian solvers can be extended to general symmetric diagonally dominant systems via standard reduction techniques. [1] [2].

References

[1] E. G. Boman, D. Chen, B. Hendrickson, and S. Toledo. Maximum-weight-basis preconditioners. *Numerical Linear Algebra Appl.*, 11:695–721, 2004.

[2] K. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996.

[3] S. Kaczmarz. Angenäherte auflösung von systemen linearer gleichungen. *Bulletin International de lAcademie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[4] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Pro 45th ACM Symp. Theory of Comp.*, (STOC '13), pages 911–920, New York, 2013.

[5] I. Koutis, G. L. Miller, and R. Peng. Approaching optimality for solving sdd systems. *CoRR*, abs/1003.2958, 2010.

[6] O. E. Livne and A. Brandt. Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver. *SIAM Scientific Comp*, 34(4):B499–B522, 2012.

[7] D. A. Spielman. Algorithms, graph theory, and linear equations in Laplacian matrices. In *Proceedings of the International Congress of Mathematicians*, volume 4, pages 2698–2722, 2010.

[8] D. A. Spielman and S. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the Thirty-sixth Annual ACM Symp. on Theory of Comp.*, STOC '04, pages 81–90, New York, NY, USA, 2004. ACM.

[9] S. Toledo. An algebraic view of the new randomized Kaczmarz linear solver. Presented at the Simons Institute for the Theory of Computing, 2013.

[10] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2004.

Assailed: A story illustration algorithm to generate a data structure connecting content, art and object

Carrie Segal and Joseph McMahan

Fig. 1. A Sequence of Pages, expressing color and emotion.

Abstract—Assailed is a visual story illustrator that generates a data structure connecting content, art and objects. When the algorithm analyzes the text of a story it scans for configurable keywords. The keywords contain dictionaries of additional information, to be accessed at a later point in the visualization generation. The selection of keywords is customizable. The keywords selected for the initial visualization are common colors that are observed to occur frequently near description of physical objects. A list of phrases based on the keywords is constructed for use in data mining content from the physical web through curated content filters. Assailed builds a representation of a book from which (1) Physical objects could be made (2) An illustrated paper book is generated, and (3) A paper book can be viewed through a device portal and return to the web.

Index Terms—Algorithm, visualization, picture generation, line and curve generation

1 Introduction

The Information Age has given rise to deep changes in many aspects of our lives, not the least of which is the field of "digital media." However, the oldest and longest-lasting common medium for information and entertainment — the written word — has remained unchanged by these new technologies. Though reading now often takes place on a tablet or screen instead of a physical page, the format of the presented information is identical. The encoding of strings consisting of a limited number of glyphs remains the primary form of communication across these devices which, are connected to a vast pool of information resources.

With the high level of connectivity offered by the Internet, the common act of reading has the potential to become an immersive experience. When curious, it is possible to enter a keyword into Google and instantly see thousands of results — our visualization algorithm intends to bring the same high level of exploration and integration of physical content into the reading experience. The story incorporates access points to explore information, art, and things.

Assailed is a fast and responsive cloud-based engine which integrates a variety of web APIs for gathering content and generating an illustrated story whose illustrations connect back to real objects, designs, and ideas. The resulting html page or pdf is reminiscent of illuminated manuscripts — but produced on-demand, from content around the web.

At its first level, Assailed aims to create an automated visual companion to stories, providing a computerized update to the oldest art form. Beyond this, it's about connecting the stories we read to the Network of Ideas and the Network of Things. A user can simply enjoy the new, visual mood set by the collection of images gathered for each paragraph, or can click and navigate at will to explore new art, objects, and ideas, allowing the story to leap from the page to the web.

2 Motivation

Prior to the age of the printing press, information was communicated through handwritten documents. Medieval manuscripts, which were drawn by hand on parchment, were frequently enhanced with artwork, including intricate ornamented decorative borders and narrative scenes. [7] The books are drawn using line drawings and finished with colored inks. Many of the books are decorated with gold, showing the high degree of value attributed to the recorded words.

The arrival of the printing press in 1450 resulted in the first uses of movable type, to replicate a fixed information structure across many different dissemination copies. Of particular note is that this mass produced printed work was also hand finished with detailed illustrations depicting the natural world. The first book printed using Gutenberg's movable type was the Bible. In particular, the first page of the Book of Proverbs is decorated with illustrations. The paintings of the natural world, including monkeys and berries, were added after the book was printed using the press. The surviving copies of the Gutenberg bible are unique, which each copy including individually made art-

work. Assailed is an algorithmic approach to illustrating books. It

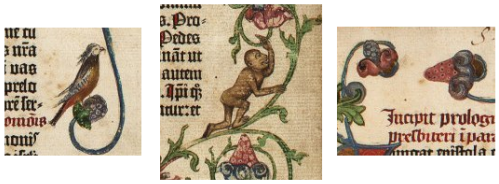


Fig. 2. A close up of the illustrations in the Book of Proverbs show a parrot, a monkey and berries. [2]

implements a variant of procedural generation, similar to evolutionary art which changes based on external variables. [3] The seeds used in the rule based generation of the illustrations are taken from the text of stories. The changes in the images used in the borders are at first decided by maker communities, and as time progresses by the readers of the book.

3 Page Generation

Assailed is intended to evoke familiarity through resemblance to a book, and, at the same time, to present additional information in parallel alongside the text. A single book is created from an original template. Currently there is one template that is used to generate all illustrations. The template is drawn using scalable vector graphics [SVG] which are generated by the text on that page. When a page is created, it is assigned a collection of paragraphs which contain keyword phrases. The keyword phrases are then used to find images corresponding to the phrase, and the keywords have their own features assigned to their data structure.

The overall goal of Assailed is to encapsulate the experience of reading an old illuminated manuscript with connectivity to the web.[4] This is the motivation for the page layout. The interest in the particular template used to generate all of the books is based on the simplicity of the design. All of the rendered circles are placed at locations predetermined by the algorithm. This means additional modifications to the drawn illustrations are possible, by incorporating functions for curves into the data structure containing information on keyword features. [8]

Fig. 3. Illustrations contained within the borders of illuminated manuscripts make frequent use of geometric curves, which are incorporated into the Assailed illustration algorithm. Lines are formed from overlapping circles, and once lines can be rendered, a variety of other shapes can be drawn, such as trefoils, curls, spirals and overlapping combinations of each. Different feature descriptors can incorporate color, shape and shading.

The print copy of the book is built from the same underlying data structure as the web copy. The only difference is the print copy uses higher resolution graphics and replaces the link to the next page with the page number of the book.

The imagery of the page is created using D3 [1], which is a javascript library for the dynamic rendering of webpages using HTML, CSS and SVG web standards. The rendering is performed using scalable vector graphics (svg) and JPEG images. The output and placement of the images and shapes are driven by the properties of the feature descriptors contained within the data models for the keywords. This means the templates are expandable to encompass a variety of display algorithms.

The ID's of the circular images are generated as strings from an integer property stored in the array corresponding to that region. They are then used as string URLs in the background fill of the circles. This is how the data structure is able to render new images on each turn of the page. The printed pdf is created by rendering each page to PDF using an automated tool such as wkpdf. [6]

Overall, the process of page generation consists of accessing information from several sources and using the crowd-sourced images and tags to determine context. Parameters are then set to configure how the borders are displayed. Information from the content sources provides data on type of information, descriptive tags, etc., which can be combined with keyword features extracted from textual phrases (like the sentiment of a passage).

The combination of keyword features and natural language processing analysis determines what should be displayed. Image processing can yield additional data for this purpose, such as dominant shapes, colors, and scene descriptors for each image, adding to the information used in deciding what SVGs to render. Furthermore, the companion colors and layout patterns of the images offer another degree of customization based on what is being displayed and what is happening in the passage. For example, if textual parsing reveals that a paragraph is tense or contains conflict, the geometric curves governing layout can be made higher frequency, the subset of displayed images used could focus more on art and objects that are tagged with conflict-related keywords, and the algorithm could more heavily weight images that have sharp features or stark contrasts. The intention is for the algorithm making these decisions to learn over time, varying output based on story, user, and page.

4 Results

While exploring preliminary results of the story illustration algorithm we used two books. The books were selected for there commonly known use of colors throughout the story. First, the book "Dorothy and the Wizard in Oz", by L. Frank Baum was used for development. A second book, "The Colors Of Space", by Marion Zimmer Bradley, was periodically used to test for differences across works. The Colors Of Space is a longer book, with a distinctly different tone than Dorothy and the Wizard in Oz.

The analysis of results begins with reviewing the keyword phrases. The keywords are a mixture of physical things such as 'the sky balloon', atmospheric descriptions such as 'the grey dawn' and characters i.e. 'little black dog'. The resulting curated search images for these phrases are relatively accurate, with 'little black dog' returning artwork of black dogs. The phrase 'the sky balloon' returns with images relating to how hot air balloons work and how we measure information about the sky using weather balloons. A way to gain an overall feel for the book is to examine the colors corresponding to the keywords found on each page.

Comparing the color spectrum of Dorothy and the Wizard in Oz vs The Colors of Space, the two books clearly have different narrative arcs. Seeing a simple summary of the text via color is a fast method for a user to perceive the mood of a story. [5] These results use only a single feature descriptor as well. Once we increase the number of feature descriptors we expect to be able to quickly present increasingly more meaningful summary images.

References

[1] M. Bostock. D3.js data-driven documents.
[2] British Library C.9.d.4, f.1. *Gutenberg's (42-line) Bible*.
[3] T. Dreher. *History of Computer Art*.
[4] F. T. Marchese. Medieval information visualization. *VIS*, 2013.
[5] S. K. B. . S. J. Palmer, S. E. Visual aesthetics and human preference. *Annual Review of Psychology, 64: 77-107*, 2013.
[6] C. Plessl. wkpdf.
[7] Syracuse University Library. *Medieval Manuscripts of Syracuse University Library*.
[8] E. W. Weisstein. *"Folium." From MathWorld–A Wolfram Web Resource.*

Towards Real-time Spectrum Monitoring

Ana Nika, Zengbin Zhang, Xia Zhou†, Ben Y. Zhao, Haitao Zheng
Department of Computer Science, UC Santa Barbara, †Department of Computer Science, Dartmouth College
{anika, zengbin, ravenben, htzheng}@cs.ucsb.edu, xia@cs.dartmouth.edu

I. Introduction

Radio spectrum is one of the most sought-after resources in the world. But, despite the value placed on wireless spectrum, little attention has been paid to a difficult problem, *spectrum enforcement*, *i.e.* how to detect and locate unauthorized users whose transmissions may interfere and disrupt transmissions from authorized spectrum users. Spectrum enforcement faces two important challenges. First, perhaps the biggest challenge is how to gather detailed spectrum measurements with strong coverage of deployed regions. Second, given the dynamics of wireless performance and mobility of users, these measurements should be "real-time". In contrast, today's spectrum measurements are carried out by government employees driving around with spectrum analyzers and specialized hardware that is usually bulky, expensive, and difficult to operate.

Our solution to the spectrum monitoring problem is to leverage the power of the masses, *i.e.* millions of wireless users, using low-cost, commoditized spectrum monitoring hardware. We envision an ecosystem where crowdsourced smartphone users perform automated and continuous spectrum measurements using their mobile devices and report the results to a monitoring agency in real-time. Our proposed system integrates two components: *a crowdsourcing measurement framework* that gathers spectrum measurement data in wide areas and *a low-cost mobile platform* that allows crowdsourced users to perform spectrum measurements automatically in real-time. Our current prototype leverages commodity mobile devices (Android smartphones and laptops) and portable RTL-SDR devices as spectrum sensors.

We performed initial measurements to compare the efficacy of our low-cost monitoring platform to that of conventional monitoring devices like USRP GNU radios. Based on our results, we believe a real-time spectrum monitoring system via crowdsourcing and commoditized hardware is indeed feasible in the near future. We conclude by identifying a set of open challenges and potential directions for follow-up research. To the best of our knowledge, we are the first to propose a real-time spectrum monitoring system using crowdsourcing and commodity mobile devices.

II. A Real-Time Spectrum Monitoring Platform

Our spectrum monitoring platform has two hardware components: a commodity mobile device, *e.g.* smartphones/laptop and a cheap (<$20) and portable Realtek Software Defined Radio (RTL-SDR) that connects to smartphones/laptops via a USB cable. The RTL-SDR behaves as a "spectrum analyzer" and collects raw spectrum usage signal, while the mobile host behaves as a "data processor" and translates the raw data into a data stream that is more compact and meaningful for the monitoring system. Figure 1 illustrates two prototypes: the RTL-SDR connected to a laptop and a smartphone.

(a) RTL-SDR/Laptop (b) RTL-SDR/smartphone

Fig. 1. RTL-SDR connected to a laptop or a smartphone

More specifically, the RTL-SDR device [1] is a DVB-T dongle that operates on the frequency range of 52-2200MHz, covering a wide range of todays wireless networks, and supports a maximum sample rate of 2.4MHz. The portable device can transfer on the fly raw I/Q samples to the host it is connected to. We also built necessary software to interconnect the two hardware components. For smartphones, we built an Android app on top of the existing RTL-SDR code [1]. For the laptop version, we leveraged the open-source project PyRTLSDR [2]. After obtaining the raw I/Q samples from the RTL-SDR device, the mobile host performs FFT to produce power spectrum density map of the collected signal. These can be used to identify active transmissions or detect useful features related to spectrum misuse detection [3].

III. Initial Feasibility Results

In this section, we evaluate the feasibility of using low-cost mobile platform (RTL-SDR) for spectrum monitoring. Compared to sophisticated hardware like USRP GNU radios, the RTL-SDR device has two key limitations:

- *Limited Sensing Sensitivity:* While USRP outputs 14-bit I/Q samples, RTL-SDR outputs 8-bit I/Q signal samples. Because of this resolution difference, RTL-SDR is less sensitive to weak signals and can fail to detect them.
- *Limited Sensing Bandwidth:* While USRP supports up to 20MHz bandwidth, RTL-SDR can only support up to 2.4MHz. In order to monitor a frequency band wider than 2.4MHz, RTL-SDR needs to sweep the band sequentially. This means that it can fail to detect certain short-term transmissions that occupy a portion of the frequency band.

[1]https://github.com/keesj/librtlsdr-android
[2]https://github.com/roger-/pyrtlsdr

Next, we perform measurement studies to understand the implications of these limitations on spectrum monitoring. Specifically, we evaluate and compare three monitoring platforms: a USRP N210 radio connected to a laptop, a RTL-SDR radio connected to a laptop and a RTL-SDR radio connected to a smartphone. All three platforms use the same antenna model and all three antennas are co-located.

A. Impact of Sensing Sensitivity

We start from quantifying the sensing sensitivity difference between USRP and RTL-SDR using noise and signal measurements.We configure all three monitoring platforms to operate on a 2.4MHz band.

Signal and Noise Measurements. First, we perform noise measurements and we identify that once the sensing duration increases beyond $1ms$, the RTL-SDR based platforms perform similarly to the USRP platform. Next, we perform signal measurements by turning on a transmitter to emit OFDM signals continuously. We vary the transmit power to create signals of different signal-to-noise-ratio (SNR). Due to RTL-SDR's limited sensitivity, the corresponding two monitoring platforms report lower SNR values ($\approx$13dB lower for RTL-SDR/smartphone, and $\approx$8dB lower for RTL-SDR/laptop) compared to the USRP platform.

Impact on Spectrum Monitoring. Our above results show that the limited sensitivity of RTL-SDR leads to 8-13dB loss in SNR reports, which means that it cannot capture weak signals reliably. To further understand this impact, we identify the SNR level that can be detected by each platform. The USRP platform can reliably detect signals with SNR $\geq$ -2dB, which increases to 7dB for RTL-SDR/laptop and 10dB for RTL-SDR/smartphone. Such 12dB difference translates into roughly 50% loss in distance, which means that the coverage requirement for a monitoring system using RTL-SDR/smartphone devices needs to be 50% denser than that using USRP/laptop. This should be easily achievable using our proposed crowdsourcing approach.

Addressing the Sensitivity Limitation. There are two potential methods to address the sensitivity limitation. First, with crowdsourcing, we can deploy many monitoring devices in a given area to reduce the sensitivity requirement on each individual device. Second, we can exploit certain signal features such as pilot tones or cyclostationary features that can potentially relax the per-device sensitivity requirement.

B. Impact of Sensing Bandwidth

Next, we investigate the impact of RTL-SDR's limited sensing bandwidth, *i.e.* 2.4MHz compared to USRP's 20MHz. To monitor a frequency band wider than 2.4MHz, the device must scan the band sequentially in segments of 2.4MHz. Intuitively, the overall scan delay is the product of the number of segments and the sum of the sensing time per segment and the time required to switch between frequency segments.

To examine the scan delay, we configure the two RTL-SDR monitoring devices to monitor a wideband of bandwidth between 24MHz and 240MHz, with a sensing duration of $1ms$ (based on our previous result) per 2.4MHz segment. We configure the USRP device to monitor the same band. We identify that the scan delay of RTL-SDR is two times higher than USRP because its frequency switching delay is higher. We observe that the switching delay of RTL-SDR is upper bounded by $50ms$ with a median of $16ms$ while USRP takes a stable value of $3ms$. Overall, the RTL-SDR radio can finish scanning a band of 240MHz bandwidth within $2s$.

Impact on Spectrum Monitoring. We quantify the impact of RTL-SDR's sensing bandwidth on spectrum monitoring by the amount of signal detection errors it can lead to. We measure the detection error rate for monitoring a 24MHz band and a 120MHz band. For the 24MHz band, the RTL-SDR/smartphone achieves less than 10% detection error even when detecting highly dynamic signal events. As the band becomes wider (120MHz), the error rate can reach 35% if the signal is highly dynamic.

Overcoming the Bandwidth Limitation. There are two potential directions to address the bandwidth limitation. First, we can leverage the power of the crowd, either dividing each wideband into several narrow bands and assigning users to specific narrow bands, or aggregating results from multiple users whose scans are inherently asynchronous. Second, we can leverage existing wideband sensing techniques such as QuickSense [4] or BigBand [2], which apply efficient signal search algorithms to perform wideband sensing using narrowband radios.

IV. Additional Challenges

Finally, we discuss several key challenges and potential research directions to address them.

Achieving Adequate Coverage. With crowdsourcing, a practical challenge is how to ensure adequate coverage. One potential solution is to adopt a combined in-network and out-of-network mechanism. Passive measurements will be collected from each wireless service provider's own user population and measurements from users of other networks can be requested as necessary to augment passive data.

Minimizing Measurement Overhead. A practical design must account for energy consumption and bandwidth cost to attract crowdsourcing participants. One research direction is to schedule measurements based on user context, including location, device placement, user movement speed/direction, statistics of observed signals, and density of nearby transmitters. Furthermore, one can explore novel data compression algorithms that compress spectrum reports on the fly without missing significant events.

References

[1] http://sdr.osmocom.org/trac/wiki/rtl-sdr.

[2] Hassanieh, H., et al. Ghz-wide sensing and decoding using the sparse fourier transform. In *INFOCOM* (2014).

[3] Yang, L., et al. Enforcing dynamic spectrum access with spectrum permits. In *MobiHoc* (2012).

[4] Yoon, S., et al. Quicksense: Fast and energy-efficient channel sensing for dynamic spectrum access networks. In *INFOCOM* (2013).

http://gswc.cs.ucsb.edu

http://cs.ucsb.edu