University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Ivan Budiselić

COMPONENT RECOMMENDATION FOR DEVELOPMENT OF COMPOSITE CONSUMER APPLICATIONS

DOCTORAL THESIS

Zagreb, 2014

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Ivan Budiselić

COMPONENT RECOMMENDATION FOR DEVELOPMENT OF COMPOSITE CONSUMER APPLICATIONS

DOCTORAL THESIS

Supervisor: Professor Siniša Srbljić, PhD

Zagreb, 2014

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Ivan Budiselić

PREDLAGANJE KOMPONENATA ZA RAZVOJ KOMPOZITNIH POTROŠAČKIH PRIMJENSKIH PROGRAMA

DOKTORSKI RAD

Mentor: Prof. dr. sc. Siniša Srbljić

Zagreb, 2014.

Doctoral thesis was made at the University of Zagreb,

Faculty of Electrical Engineering and Computing,

Department of Electronics, Microelectronics, Computer and Intelligent Systems

Supervisor:

Professor Siniša Srbljić, PhD

Doctoral thesis contains: 187 pages.

Doctoral thesis number: _____

About the Supervisor

Siniša Srbljić was born in Velika Gorica in 1958. He received B.Sc. degree in EE and M.Sc. and Ph.D. degrees in CS from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), in 1981, 1985, and 1990, respectively. From February 1982, he has been working at the Department ZEMRIS at FER. He worked at Prvomajska, R&D Dep., Croatia (1984-86) and he was a visiting scientist at the University of Toronto, Canada (1993-1995), at the AT&T Labs, USA (1995-99), at the UC, Irvine, USA (2000,08-09), at the GlobalLogic Inc, USA (2011), and at the US Huawei, USA (2011). In March 2007, he was promoted to tenured professor. He coordinated 1 scientific program, led 1 scientific project, 1 technological project, and participated in 9 scientific projects. He led 3 research projects financed by companies from Croatia and the USA, led 2 projects in the USA, and participated in 3 projects in the USA and Canada. He coordinates the scientific program "Distributed Systems, Methods, and Applications" and leads the scientific project "Computing Environments for Ubiquitous Distributed Systems" financed by MZOS RH. He is the author of 2 textbooks, more than 60 papers in journals and conference proceedings, and two patents in the USA in the area of distributed computing systems and consumer computing. Prof. Srbljić is a member of IEEE, ACM, and HATZ. He received the Silver medal "Josip Lončar" from FER for his Ph.D. thesis in 1990 and "Vratislav Bedjanič" Award, Iskra, Ljubljana, for his M.Sc. thesis in 1985.

O mentoru

Siniša Srbljić rođen je u Velikoj Gorici 1958. godine. Diplomirao je u polju elektrotehnike, a magistrirao i doktorirao u polju računarstva na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva (FER), 1981., 1985. odnosno 1990. godine. Od veljače 1982. godine radi u Zavodu ZEMRIS FER-a. Bio je zaposlen u tvornici Prvomajska, odjel Istraživanje i razvoj (1984.-86.), a gostujući znanstvenik bio je na University of Toronto, Kanada (1993.-95.), u AT&T Labs, SAD (1995.-99.), na UC, Irvine, SAD (2000., 2008.-09.), u GlobalLogic Inc (2011.) i u US Huawei, SAD (2011.). U ožujku 2007. godine izabran je u trajno znanstveno-nastavno zvanje redovitog profesora. Koordinirao je 1 znanstveni program, vodio 1 znanstveni projekt, 1 tehnologijski projekt i sudjelovao na 9 znanstvenih projekta MZOS RH. Vodio je 3 istraživačka projekta financirana od kompanija iz Hrvatske i SAD, vodio 2 projekta u SAD i sudjelovao na 3 projekta u SAD i Kanadi. Koordinira znanstveni program "Raspodijeljeni sustavi, metode i primjene" i vodi znanstveni projekt "Računalne okoline za sveprisutne raspodijeljene sustave" koje financira MZOS RH. Autor je 2 udžbenika, više od 60 radova u časopisima i zbornicima konferencija i 2 patenta u SAD u području raspodijeljenih računalnih sustava i potrošačkog računarstva. Prof. Srbljić član je IEEE, ACM i HATZ. Primio je srebrnu plaketu "Josip Lončar" FER-a za doktorsku disertaciju (1990.) i nagradu "Vratislav Bedjanič", Iskra Ljubljana, za magistarski rad (1985.).

Acknowledgments

It has been quite a journey getting here, and this is a good time to reflect and thank some of the people that have helped me along the way—and there sure have been many.

I'd like to start from the beginning and thank Predgrag Brođanac and Tomislav Gracin who taught me the first nontrivial algorithms while I was in high school, and introduced me to the magical world of computer science. There are a few other educators I'd also like to thank for significantly influencing my interests and my life, though I suspect they are not aware of this fact: Dario Mičić for being an amazing physics teacher, and Vladimir Ćepuluć, Darko Žubrinić and Mario Osvin Pavčević for their radiating passion for mathematics.

I want to thank my supervisor Professor Siniša Srbljić for the discussions we had over the years about this research idea that ultimately became my PhD thesis, and for his support along the way. I especially want to thank Professor Srbljić and his friends in the Bay Area, primarily Roman Porenta and Niko Bagoje, for the incredible help they provided me and my mom in settling in in 2008. I'm sure it would have been much easier for Professor Srbljić to let me decide that I wasn't going, and I would have missed what has been one of my most amazing experiences in life.

I'd like to thank my lab colleagues Goran Delač, Ivan Žužak, Klemo Vladimir, Marin Šilić, Zvonimir Pavlić, Miroslav Popović and Dejan Škvorc for their help and support over the last six years—it has been great fun discussing all sorts of things with you (including science!).

I've been incredibly fortunate in this life to have an amazing family and more friends than I dare to name here for fear of leaving someone out—thank you all, from the bottom of my heart. I want to especially thank my grandparents Katica, Marica, Milan and Dušan and aunt Beška for their love, being great people to look up to and learn from, and providing me with some of my fondest memories.

Most of all, I want to thank my parents Duška and Goran, my brother Marko and my aunt Vesna for being the pillars of my life. This is for you.

Abstract

Component Recommendation for Development of Composite Consumer Applications

Consumer computing is a research area that focuses on methodologies and tools that enable consumers—the most general class of users of technology, typically with no special software engineering skills—to create their own applications. Consumers create applications by *composing* existing applications through intuitive actions on their graphical user interfaces. Support for component discovery has been identified as a key challenge in various forms of composite application development, and is especially important in consumer computing. This dissertation introduces a general method for component recommendation based on structural similarity of compositions which dynamically ranks and recommends components as a composite consumer application is being incrementally developed by structurally comparing the partial composition with a database of previously completed compositions. Using this method, four component recommender algorithms are defined: two based on feature vector models and cosine similarity, one based on sequence edit distance and one based on a directed graph model and a probabilistic graph edit distance algorithm. Accuracy, coverage and response time of the presented algorithms are evaluated in detail on a *Yahoo Pipes* dataset and a synthetic dataset that models more complex composite consumer applications. The results show that the presented approach is effective in addressing the component discovery challenge in consumer computing.

Keywords: consumer computing, component-based systems, component discovery, recommender systems, structural similarity.

Sažetak

Predlaganje komponenata za razvoj kompozitnih potrošačkih primjenskih programa

Potrošačko računarstvo je istraživačko područje usmjereno prema metodologijama i alatima koji omogućuju *potrošačima*—najširoj klasi korisnika tehnologije, tipično bez znanja i vještina iz područja programskog inženjerstva—da stvaraju vlastite primjenske programe. Potrošači stvaraju primjenske programe povezujući postojeće primjenske programe koristeći intuitivne akcije na njihovim grafičkim korisničkim sučeljima. Podrška za otkrivanje komponenata je ključan izazov u raznim okruženjima za razvoj kompozitnih primjenskih programa i posebno je važna u potrošačkom računarstvu. Ova disertacija uvodi općenitu metodu za predlaganje komponenata zasnovano na strukturnoj sličnosti kompozicija koja dinamički rangira i predlaže komponente tijekom postupnog razvoja potrošačkog primjenskog programa uspoređujući djelomičnu kompoziciju s bazom prethodno izgrađenih kompozicija. Koristeći navedenu metodu, definirana su četiri algoritma za predlaganje komponenata: dva zasnovana na modelima svojstvenih vektora i kosinusnoj sličnosti, jedan zasnovan na udaljenosti promjene nad slijednim modelom i jedan zasnovan na modelu usmjerenog grafa s označenim vrhovima i vjerojatnosnom algoritmu za udaljenost promjene nad grafovima. Točnost, pokrivenost kataloga i vrijeme odziva predloženih algoritama vrednovani su na skupu *Yahoo Pipes* kompozicija i sintetičkom skupu kompozicija koji modelira složenije potrošačke primjenske programe. Rezultati pokazuju da je predloženi pristup prikladan i učinkovit za rješavanje problema otkrivanja komponenata u potrošačkom računarstvu.

Prvo poglavlje (1 "*Introduction*") predstavlja motivaciju za provedeno istraživanje i uvodi glavnu hipotezu istraživanja da je problem predlaganja komponenata moguće riješiti koristeći strukturnu sličnost kompozitnih primjenskih programa. Poglavlje je zaključeno kratkim pregledom svih preostalih poglavlja rada.

U drugom poglavlju (2 "*Introduction to Consumer Computing*") dan je uvod u istraživačko područje potrošaču usmjerenog računarstva. Ukratko je opisana povijest područja i motivacija za daljnje istraživanje. Poglavlje nastavlja detaljnim opisom metodologije za razvoj aplikacija razvijene unutar područja i zaključuje pregledom prototipa okoline za gradnju potrošačkih kompozitnih aplikacija *Geppeto*, razvijenog u Laboratoriju za potrošaču usmjereno računarstvo. Na primjeru *Geppeta*, prikazane su osnovne značajke takve okoline.

U trećem poglavlju (3 "*Review of Related Research*") prikazan je pregled istraživačkih rezultata iz nekoliko područja povezanih s problemom predlaganja komponenata. Poglavlje započinje analizom sustava za predlaganje, pri čemu su opisani *sustavi zasnovani na analizi sadržaja*, *suradničko filtriranje* i *hibridni* pristupi, a za svaki pristup je navedeno i nekoliko primjera stvarnih sustava. Nadalje, poglavlje opisuje veći broj sustava i pristupa za potpomognuti razvoj usloženih primjenskih programa (engl. *mashup*) s posebnim naglaskom na sustav *Yahoo Pipes*, kao i u području programskog inženjerstva.

Četvrto poglavlje (4 "*Component Recommendation in Consumer Computing*") opisuje proces predlaganja komponenata zasnovanog na strukturnoj sličnosti kompozicija koji se sastoji od četiri ključna koraka: *pretprocesiranju reprezentacije*, *ocjeni sličnosti*, *izračunavanju korisnosti komponenata* i *predlaganju komponenata*. Definirani su osnovni pristupi u sva četiri koraka, u općenitom kontekstu kompozicijskih sustava. U nastavku poglavlja opisan je potrošački pomoćnik za predlaganje komponenata. Pomoćnik je osmišljen u skladu s osnovnim principima potrošaču usmjerenog računarstva i zasniva se na nekoliko koncepata s kojima su potrošači dobro upoznati kroz svakodnevno korištenje Weba i prijenosnih uređaja.

U petom poglavlju (5 "*Modeling Composite Applications*") formalno su opisana četiri modela strukture kompozitnih aplikacija. Najopćenitiji model zasnovan je na usmjerenom grafu s označenim vrhovima. Uz njega, opisan je model poredanog niza komponenata zasnovan na poopćenom topološkom poretku vrhova grafa. Konačno, definirana su i dva modela svojstvenih vektora koji sadrže najnižu razinu opisa strukture kompozicije. Sva četiri modela su prikazana na primjerima kompozicija iz *Geppeta* i iz sustava *Yahoo Pipes*.

Šesto poglavlje (6 "*A Framework for Component Recommendation Based on Composition Structural Similarity*") prikazuje radni okvir za predlaganje komponenata. Definirana je općenita metoda na osnovi koje je moguće definirati svaki algoritam za predlaganje komponenata zasnovan na sličnosti kompozicija. Spomenuta metoda prati proces predlaganja komponenata definiran u poglavlju 4. Nadalje, koristeći navedenu metodu, definirana su četiri algoritma za predlaganje komponenta: algoritam zasnovan na vjerojatnosnoj udaljenosti promjene među grafovima, algoritam temeljen na udaljenosti promjene poredanih nizova komponenata i dva algoritma zasnovana na kosinusnoj sličnosti vektorskih modela strukture kompozicije. Nakon formalne definicije, osnovna svojstva algoritama su sažeta na kraju poglavlja i rad algoritama je prikazan na primjeru.

U semdmom poglavlju (7 "*Evaluation Methodology*") je opisana metodologija vrednovanja

predloženih algoritama korištena u doktorskom radu. Poglavlje počinje analizom dva skupa kompozicija koji su korišteni za vrednovanje (skup pravih kompozicija iz sustava *Yahoo Pipes* i sintetički skup strukturno složenijih kompozicija koji je stvoren u skladu s nekim svojstvima potrošačkih primjenskih sustava koji se grade koristeći *Geppeto*). U nastavku su definirane mjere uspješnosti algoritama. Točnost preporuka ocjenjuje se kroz preciznost, odziv i F_1 vrijednost, koje su uobičajene mjere za točnost iz područja pretraživanja informacija, uz dodatnu intiutivinu mjeru po kojoj se prijedlog smatra točnim ako je barem jedna predložena komponenta korisna. Uz točnost se ocjenjuju pokrivenost kataloga komponenata i vrijeme izvođenja sustava. U nastavku poglavlja opisana su tri jednostavna statistička algoritma za predlaganje komponenata s kojima sa predloženi strukturni algoritmi uspoređuju i koji daju jasniji uvid u relativnu složenost problema predlaganja komponenata na dva skupa kompozicija. Poglavlje je zaključeno detaljima vezanim uz opis vrednovanja vjerojatnosnih algoritama i odabir vrijednosti parametara algoritama.

U osmom poglavlju (8 "*Snapshot Evaluation*") dan je pregled rezultata prvog scenarija vrednovanja koji je zasnovan na *primjerima djelomičnih kompozicija*. Za primjere se odabiru dijelovi postojećih kompozicija, a izlaz sustava za predlaganje se uspoređuje sa sakrivenim dijelom kompozicije koji nije bio dio primjera. Ključni rezultati su prikazani u sažetku na kraju poglavlja.

Deveto poglavlje (9 "*Simulated Composition Evaluation*") prikazuje drugačiji pristup vrednovanju algoritama zasnovan na simulaciji postupka gradnje kompozicije. Kako bi se pokrili različiti načini na koje potrošači dodaju funkcionalnost u svoje kompozicije i kako koriste sustav za predlaganje, definirane su četiri različite strategije gradnje. Osnovni cilj ovog načina vrednovanja je ograničavanje utjecaja popularnih komponenti na točnost sustava koji je uočen u vrednovajnu zasnovanom na primjerima. Poglavlje je završeno pregledom najvažnijih rezultata koji su dodatno uspoređeni s rezultatima vrednovanja iz prethodnog poglavlja. Zaključeno je da je pristup predlaganju komponenata predložen u doktorskom radu prikladan i za skup kompozicija iz sustava *Yahoo Pipes* i za sintetički skup.

Deseto poglavlje (10 "*Conclusion*") zaključuje rad pregledom osnovnih rezultata vrednovanja i ostvarenih izvornih znanstvenih doprinosa.

Ključne riječi: potrošačko računarstvo, sustavi zasnovani na komponentama, otkrivanje komponenata, sustavi za predlaganje, strukturna sličnost.

Contents

Chapter 1

Introduction

Consumer computing is a research area focused on methodologies and tools that enable consumers, which are users of the Web and modern mobile technology like smartphones and tablets with no additional education or experience in application development, to create their own applications. The guiding principle in consumer computing is the *equality of consuming and programming* [1], which states that consumers should create applications in the same way in which they use them in everyday life. In line with this principle, consumer applications are created by *composing* existing applications to create new functionality.

Composite applications are defined in [2] as business applications constructed by connecting disparate software components, thus providing new functionality to an end user, ideally without the requirement to write any new code. Several different types of components are mentioned, including web applications, web services and native widgets. Consumer computing extends this definition beyond business applications to social, scientific, news, entertainment and other kinds of applications[1].

Components have been a vehicle of reuse and interoperability in software engineering for decades, with a large number of component models in use today [5]. The World Wide Web Consortium has recently started work on *Web Components* which promise to bring many benefits in this area to mainstream web development [6].

Composite applications are created in *composition systems* which consist of a *composition workspace* that is the interface of the system for composing applications, and a *composition engine*, which provides support for both the composition process and also executes composite applications. The core ideas of consumer computing are embodied in the prototype con-

[1] Composite applications are closely related to *mashups*, in that mashups are composite *web* applications that are primarily created through programming to APIs [3, 4].

sumer computing composition system *Geppeto*[2] that allows consumers to compose web widgets, which are web applications with a small graphical user interface (GUI) [7].

Another example of a consumer computing composition system is *Yahoo Pipes*[3] which enables feed aggregation and manipulation through point-and-click wiring of feed sources through various processing modules, finally producing an output feed. While *Yahoo Pipes* is restricted to data flow manipulation in a constrained domain, *Geppeto* is geared towards more general applications, allowing consumers to define both data and control flow of the composition over the widgets' GUIs, as well as control the widgets through time, location and other events. Examples of compositions created in both of these systems are given in section 5.1.

A lack of good support for component discovery has been identified by researchers as a common shortcoming of composition systems, both for the web [8–10], and for the enterprise [2], as well as in component-based software engineering [11–13]. If any component discovery support is available, it is typically restricted to textual search based on keywords in component names and descriptions or fairly general tags, or a complex query language. Textual search has two significant limitations. First, searching by keywords and tags has low resolution if a large number of components satisfy a particular query. As the search process is contextually independent of the problem the user is trying to solve, result ranking can only be based on component usage statistics, user ratings or similar measures which are not necessarily good indicators that a particular component will be useful for that problem. Furthermore, when choosing between several components that seem to provide similar functionality, users are faced with a lack of relevant information to make the right choice.

Second, searching for components is a separate activity to composition development. Therefore, before searching for a component, the user has to be aware that a component with a specific functionality might exist.

In consumer computing, the challenge of component discovery is even more significant for three reasons. First, as general-purpose applications are reused as components, the potential number of components is much larger than in other composition system domains. Second, applications that are used as components are mostly not designed particularly for that purpose and are not annotated with any special metadata that could make discovery easier. Third, benefits of consumer computing would be lost if consumers were expected to learn a complex query language just to find useful components. Therefore, textual search approaches in consumer

[2] geppeto.fer.hr
[3] pipes.yahoo.com

computing are limited to simple keyword queries which are not effective, as described above.

The aim of this dissertation is to analyze this component discovery challenge, and describe how it can be alleviated by providing consumers with a component recommender that dynamically suggests potentially useful components during the composition process. The recommender takes as input the partial composition the consumer is working on and compares its structure to finished compositions that have been previously created in the composition system and are stored in a *composition database*. Components found in compositions that are in some sense similar to the input partial composition are then recommended to the consumer. This approach is based on the premise that other consumers might have solved similar problems before and that the structures of the compositions that these consumers have created implicitly encode their knowledge about the used components, e.g. which components are useful together and how they should be connected. This knowledge is sometimes called *composition knowledge* in the research literature [14, 15].

The proposed recommender addresses the mentioned limitations of textual search. First, the problem context as represented by the partial composition is the input to the recommender, and only components that have proved useful in a similar context will be recommended. Second, recommendations can be embedded inside the composition workspace, and new recommendations provided after each step of composition with no additional user action. Therefore, the recommender becomes an integral part of the composition workspace, and using it a part of the composition process.

When recommendations are inadequate, for example because the consumer is trying to solve a completely new problem that shares little similarity with previously solved problems, other discovery mechanisms such as text search will still have to be employed. On the other hand, recommendations are made available even when the consumer has a specific component in mind and would not engage in any form of component discovery otherwise. This provides an opportunity for new ideas to emerge if some of the recommended components provide useful functionality that the consumer was unaware of.

The primary goal of the presented research was to test the hypothesis that structural similarity between compositions can be used to provide useful component recommendations. Furthermore, several different approaches within this framework are compared and evaluated in different scenarios to provide better understanding of the component recommendation problem, especially within consumer computing.

The remainder of the dissertation is organized in the following way. An introduction to consumer computing is given in chapter 2. The motivation for research in consumer computing is discussed first, followed by a description of the consumer computing application development methodology. The generally described programming elements, language and technique of consumer computing are then illustrated on the example of *Geppeto*.

Chapter 3 provides an overview of related research with two major focus points. First, general recommender system technology developed in the last two decades is presented, and examples of both content-based and collaborative filtering recommender systems are discussed. Second, many systems that have been developed to assist people in mashup development in general, *Yahoo Pipes* development and classical software engineering are analyzed.

Chapter 4 describes the component recommendation process based on composition structural similarity in detail. Four key steps in the process are identified: *representation preprocessing*, *similarity evaluation*, *component scores computation* and *component recommendation*. The process is analyzed in a general composition system setting. Then, the chapter focuses on the interaction between consumers and the component recommender system through a proposed machine assistant widget *NextComponent* designed for use in *Geppeto*.

Chapter 5 defines the structural models used for representing composite applications. Four models with different levels of structural information are considered: a directed graph model with labeled vertices, a component sequence model and two feature vector models. These four models are used as composition representations in the four structural recommender algorithms presented later in the dissertation.

Chapter 6 introduces the framework for component recommendation based on structural similarity of compositions. First, a general method for component recommendation that follows from the component recommendation process is described. Four structural recommender algorithms are defined using this method—two based on cosine similarity of feature vector representations, and two based on edit distance over the component sequence and graph representations. Multiple algorithms are defined with the primary goal of analyzing how different levels of structural abstraction perform in the component recommendation problem. After detailed formal definitions, the main properties of the four algorithms are summarized and considerations for choosing their parameter values are discussed. Finally, the chapter is concluded with an example of algorithm operation.

Chapter 7 presents the evaluation methodology used in the dissertation. Evaluation is based

on using parts of existing compositions as queries to the recommender, whose output is then compared to the hidden part of the composition. Two datasets are used—a *Yahoo Pipes* dataset and a synthetic dataset that aims to model some of the properties of compositions that can be created with *Geppeto*, specifically that compositions can be structurally more complex and diverse than in the Pipes dataset. The properties of these datasets that are important for component recommendation like composition size and component frequency distributions are described in detail, and a summary of these properties with a focus on comparing the two datasets is provided. Recommender quality is measured through accuracy, coverage and response time. Three simple baseline algorithms based on statistical analysis of the composition database are introduced with the goal of providing further understanding of the used datasets through comparison with the structural algorithms. The chapter is concluded with an explanation of how probabilistic algorithms were evaluated and which algorithm parameter values were used.

Chapter 8 presents the results of the first evaluation scenario that is based on composition *snapshots* which are example queries to the recommender generated from existing compositions. There are two key focus points in snapshot evaluation. First, it aims to find the optimal fraction of the compositions most similar to the input snapshot that should be considered by the recommender when making recommendations. This process of using only some of the most similar compositions as the basis for component recommendation is called *similarity filtering*. Second, recommender quality is evaluated with different numbers of components recommended per query to provide guidelines for choosing this important recommender parameter. Additionally, effects of different database sizes, the presence of connections in input partial compositions and two different definitions of useful recommendations on recommender quality are also evaluated. A summary of all the main results of snapshot evaluation is given at the end of the chapter.

Chapter 9 analyzes the results of the second evaluation scenario. Instead of taking snapshots of compositions, their creation is simulated using four different simulation strategies that model different user intents and the interaction between the user and the component recommender system. The main goal of simulated composition evaluation is to further constrain the definition of what makes a component a useful recommendation in order to limit the effect of popular components on recommender accuracy results. After a detailed look at accuracy and response times in this evaluation scenario, the chapter is concluded with a summary of the key results that are also compared to results obtained in snapshot evaluation.

Chapter 10 concludes the dissertation with an overview of main results and original scientific contributions.

Chapter 2

Introduction to Consumer Computing

Consumer computing [1] is a research area introduced by the Consumer Computing Laboratory (CCL)[1] at the University of Zagreb, Faculty of Electrical Engineering and Computing (FER-CCL) that focuses on tools and methodologies for including consumers—the most general class of users of the Web and mobile devices, with no specific education in computer science or software engineering—in not only using digital systems, but also helping create them and improve them. In this way, an unprecedented innovation potential of people with very diverse sets of interests and domain knowledge is unlocked. This chapter briefly introduces consumer computing, with the goal of providing better context for the component recommendation problem that is the topic of this dissertation.

The remainder of the chapter is organized as follows. First, section 2.1 further describes the motivation for research in consumer computing. Then, section 2.2 describes the application development methodology that was defined within the consumer computing research area. Finally, the chapter is concluded in section 2.3 with an introduction to the prototype consumer computing composition system *Geppeto*, where all the key concepts from this application development methodology are further illustrated with examples.

2.1 Motivation for Consumer Computing

Research of consumer computing began with end-user languages for service composition—first with *Coopetition Language* [16], which is an extension of WS-BPEL, then with PIEthon [17], a Python-based DSL for service composition, the service coordination language *Simple*

[1] ccl.fer.hr

Service Composition Language [16], and the service composition spreadsheet-like language *HUSKY* [17, 18]. Through this research, it became clear that consumers could create service compositions, and thus their own applications, if they were given appropriate tools and an intuitive representation of services, and furthermore, that this concept could be applied to applications in general, and not only service compositions.

Two closely related factors are the main motivation for research in consumer computing. First, it is clear that all the applications people want can't be created by professional developers only. With the recent rise of mobile *smartphones* and tablets, application development has increased dramatically. For example, *Google Play* and Apple's *App Store*, which are two of the largest application markets in the world, currently offer more than a combined two million applications [19, 20], with an increase of 60% in the last year. This growth of application markets follows an increase of app usage by consumers—a smartphone user in the US had 32 apps installed in 2011 and 42 apps in 2012, on average [21]. To put this increase in context, several striking statistics about the mobile market need to be considered [22].

First, 56% of people in the world own a smartphone, and 50% of mobile phone users use their mobile devices at their primary means of access to the Internet. 80% of the time spent using a mobile phone is spent in apps. Furthermore, mobile phones and tablets outsold desktop and notebook computers by a factor of four with over *two billion* mobile devices shipped globally in 2013. Fueled by this fact, mobile Web adoption is growing eight times faster than Web adoption had in the last two decades.

A clear connection can be drawn between this rise of mobile app markets and interest in computer science and engineering education. In the US, the number of bachelor degrees in these fields had been decreasing rapidly since 2004 up to 2009 [23]. However, a reverse effect can be observed since 2009. Similar trends are visible in master degrees and PhDs as well.

Regardless of this fact, there is still a shortage of workers for computer science related jobs in the United States [24]. With a projected 144500 average annual job openings in IT and computer engineering, only 88161 degrees are earned. This trend is expected to continue for the foreseeable future.

However, an even more significant problem exists. While professional developers are trying their best to create various applications for consumers, it is simply impossible to create personalized situational applications that consumers sometimes need and want. Tim Berners-Lee, the inventor of the World Wide Web, identified the lack of programming ability as the second

digital divide, along with the more well-known divide between those who have access to technology and the Internet and those that don't [25]. The key consequence of this divide is that consumers depend on *"a bunch of companies who would love to be able to lock it down, so you can only run the applications that they allow; the ones you can get from their app-store"*.

One popular attempt to address this divide is *code.org* which is trying to get programming into schools all over the world, and is supported by famous programmers like Mark Zuckerberg and Bill Gates and many music, movie, and sport stars alike. In a recent *Hour of Code* event, almost 20 million US school kids tried computer programming for an hour, writing an astounding 664 million lines of code [26].

A similar democratization already happened once on the Web in the context of content creation with the rise of the so called *Web 2.0* [27]. In the early Web, creating your personal page required you to know some HTML and buy and manage a domain name and page hosting. All of these things require some level of technical proficiency which was a major barrier to entry for a large group of people who didn't specialize in computers. However, this changed when easy to use tools for *blogging* and web-content management became available, and even more so with the rise of social network sites like *MySpace* and *Facebook*.

Consumer computing aims to apply the same approach to the divide in programming ability—rather than teach everyone the complexities of programming in general, the premise of consumer computing is that consumers can develop their applications with what they already know, provided they are given the right tools for the task.

The second motivating factor for consumer computing which enables such tools for application development is the very fact that app numbers and consumption are on the rise. Specifically, consumers know how to use existing applications and can reuse their functionality by *composing* them to create new applications through *automation of consumption knowledge*. These compositions are based on interactions between existing applications that can be carried out by hand by moving data between applications, clicking on buttons and so forth. However, instead of repeating this menial and error prone process many times, the consumer can define it only once after which it becomes *automated* into a new consumer application. This concept is examined in more detail in the next section.

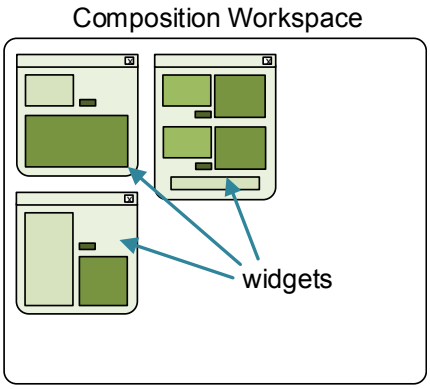


Figure 2.1: Three widgets tiled in a composition workspace.

2.2 Consumer Computing Application Development Methodology

The key guiding principle in the design of the application development methodology for consumer computing is the *principle of equality of consuming and programming* [1] which states that consumers should be given tools to create applications in the same way in which they use them. The methodology is defined through its *programming elements*, *programming language*, and *programming technique* which are described in the following subsections.

2.2.1 Programming Elements in Consumer Computing

In classical software engineering, applications are built using diverse functions, classes and services which are then orchestrated through program code to achieve some desired functionality. This process is inherently complex and typically requires that developers have years of education and training.

In consumer computing, applications are built by composing existing applications, at a much higher level of abstraction. To be usable as a component for a consumer application, an application has to communicate with its users through a graphical user interface (GUI), although more advanced interfaces like voice communication could also be supported in principle.

Ideal examples of consumer computing programming elements include software widgets and mobile applications. These two form factors are specifically suited for consumer computing because their GUIs are mostly small and uniform in size, which makes it easy to *tile* them in a composition workspace, as shown conceptually in figure 2.1. In that way, multiple applications can be used and observed at the same time, without changing views such as browser tabs.

Users are also very familiar with these application formats and special care is taken to design their interfaces to be usable and intuitive. It is important to note that most web applications can be converted to this form factor [28]. Furthermore, services that are typically accessible only through APIs can also be packaged as widgets by creating a suitable graphical interface that exposes their functionality [7]. In fact, most existing widgets and some mobile applications have been created in exactly this way as they execute by communicating with a backing service over the Internet.

Three distinct categories of programming elements are used in consumer computing. *Application-specific components* provide consumers with functionality from different domains that can then be used as the basis of the functionality of the composition. Application-specific components are typically created by professional developers using either web or mobile technologies. However, a once created consumer application can itself become a component in another consumer application. This concept of hierarchical composition is a key enabler of sustainable consumer application development as the set of available functionality continually increases in size.

Generic programmable components allow consumers to define their composition logic through control and data flow, and to specify interactions with other applications and the environment using communication mechanisms as well as processing time-based, location-based and other events. The key challenge in designing these components is defining interfaces and interactions that are intuitive to consumers and not overly technical, while still providing support for creating high quality applications that have satisfactory functional and nonfunctional properties.

Even though application development in consumer computing tries to leverage existing consumer knowledge, several significant challenges arise in practice which stem from the inherent complexity of application development. The challenge that is central to this dissertation is *component discovery*, which is a recurring theme in almost all composition systems, even those designed for professional software developers that are motivated to learn complex query languages [2, 8, 9, 11–13].

With a large and rapidly increasing catalog of components, consumers need help finding the best match for what they need in their application. When choosing components, consumers should be provided with relevant information about the components' functional properties, but also nonfunctional properties such as reliability, security, and privacy that can have a significant effect on the quality of their application. Furthermore, it should be possible to preserve good nonfunctional properties in the entire composition, which is a highly technical problem that

needs to be presented to consumers in a simplified but effective form.

To address these challenges, the third category of programming elements in consumer computing are *consumer assistants* which provide various types of help during application development. Two types of consumer assistants are identified based on the origin of assistance. *Machine assistants* are components that rely on machine learning and recommender system technologies and analyze the database of consumer applications to provide real-time assistance while consumers create applications. On the other hand, *human-based assistants* include fellow consumers in the assistance process and are based on social computing and human computation. Examples of both types of assistants have been developed for *Geppeto* and are briefly described in section 2.3.

2.2.2 Programming Language in Consumer Computing

In traditional software engineering, many different programming languages with vastly different feature sets and properties are employed. For example, systems programming is typically done in low-level compiled languages that excel in memory and time efficiency and provide the programmer with near-direct access to the hardware. On the other hand, in some areas of web development and scientific computing where developer efficiency is much more important than the efficiency of the actual software, scripting languages that provide a higher level of abstraction and typically require less code to do the same task are employed.

In line with the principle of equality of consuming and programming, the programming language used in consumer computing is equated with the "language" consumers use to interact with components anyway—the language of actions on graphical user interfaces. Example commands in this language include actions like *click*, *copy*, *paste*, and *type in* which are all well known to consumers.

2.2.3 Programming Technique in Consumer Computing

Software typically gets created through the process of *writing code*, which entails entering program instructions into a text file using some sort of text editor or an integrated development environment (IDE). Even if consumers were very familiar with the language used to define composite consumer applications, that alone would not be sufficient to enable them to create these applications without significant instruction and education using such a classical programming

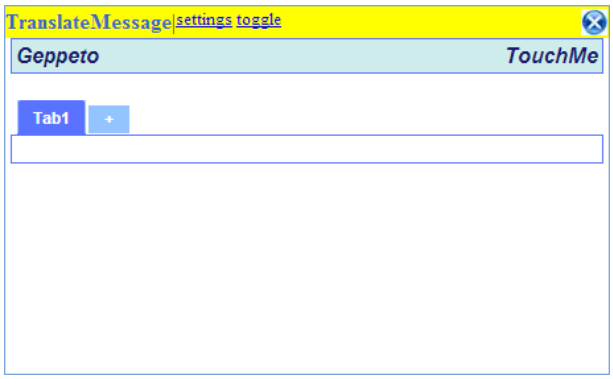


Figure 2.2: The initial state of the *Geppeto TouchMe* programmable widget titled *TranslateMessage*.

technique. For example, the key challenge of identifying various elements of component GUIs in code would not be alleviated.

To overcome this and other challenges, consumer applications are created using *programming by demonstration* [29]. Early examples of programming by demonstration were various macro recorders, and the concept was extended to end-user programming in the 1980s and 1990s [30–33], and has found widespread use in robotics in recent years [34–37].

In its application to consumer computing, instead of writing code into a text file, the consumer *demonstrates* what the composite application should do and the composition system uses that demonstration to create the application. By reusing both the language and technique of application consumption for application development, consumers don't need to be specially trained before they can start composing applications.

2.3 Introduction to *Geppeto*

Geppeto is the prototype implementation of a consumer computing composition system developed at FER-CCL that applies the ideas discussed in the previous section. *Geppeto* uses web widgets as its programming elements, i.e. components, and is built on top of the *Apache Shindig*[2] widget rendering server.

Examples of application-specific components usable in *Geppeto* are the *Google Maps* widget, which provides a powerful mapping service with route planning, the *Deterministic Finite Automaton* widget which allows users to specify and simulate DFAs through point-and-click actions, and a set of widgets for overview of unmanned submarine missions [38, 39].

The central generic programmable component is the *Geppeto TouchMe* widget—an initially

[2] `shindig.apache.org`

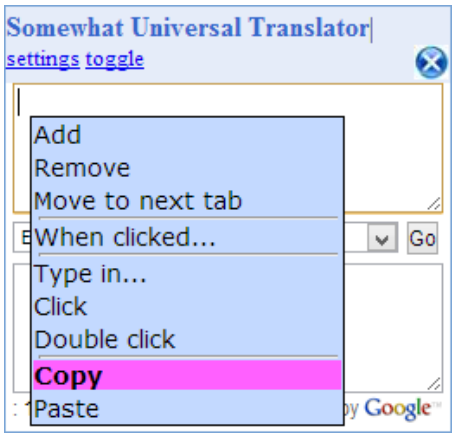


Figure 2.3: An example of the right-click context menu in *Geppeto*.

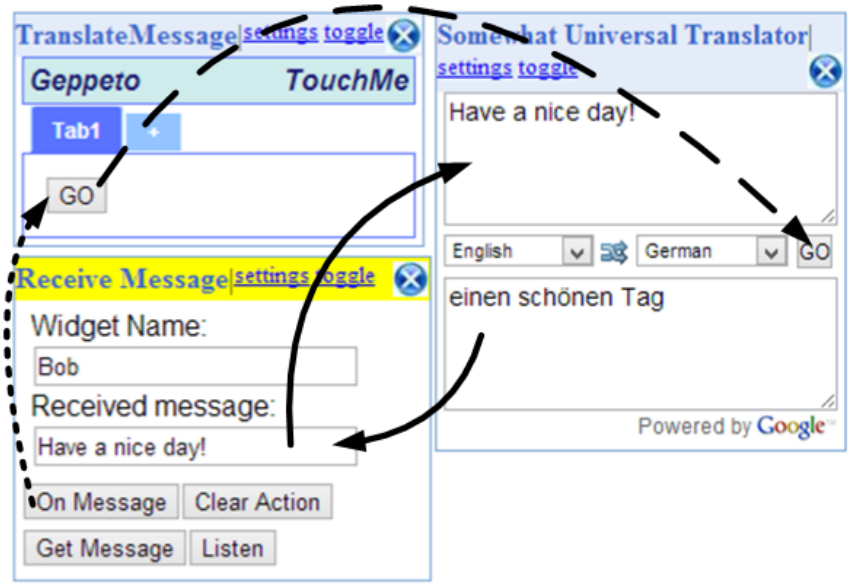


Figure 2.4: A simple *Geppeto* application for translating incoming chat messages from English to German.

GUI-less widget with no functionality as shown in figure 2.2. The consumer sets the title of this widget when it is being added to the composition workspace. In the figure, the *TouchMe* widget is titled *TranslateMessage* indicating the intended functionality of the widget.

Consumers build the GUI of the *Geppeto TouchMe* widget by reusing existing GUI elements of other components that are being composed into the new application using the right-click context menu shown in figure 2.3. Specifically, the *Add* action in the menu is used to add GUI elements, while actions like *Click*, *Copy*, and *Paste* are used to define the logic of the *TouchMe* widget, as well as other generic programmable components in *Geppeto*.

An example application created in *Geppeto* using the *TouchMe* widget is shown in figure 2.4. The goal of this application is to translate received chat messages from English to Ger-

wait for click Go at TranslateMessage	
copy element1 at ReceiveMessage to element2 at SomewhatUniversalTranslator	
click Go at SomewhatUniversalTranslator	
copy element3 at SomewhatUniversalTranslator to element1 at ReceiveMessage	

Figure 2.5: The two dimensional table storing the actions recorded in the *TouchMe* widget.

man[3]. For communication purposes, the *Receive Message* widget is used, which is another generic programmable widget available in *Geppeto*. The *On Message* button on the *Receive Message* widget is programmable and makes it possible to create message-driven applications. Specifically, the consumer can specify which other button is to be clicked whenever a message is received.

The dotted arrow in the figure represents this connection between the *Receive Message* widget and the *Geppeto TouchMe* widget that organizes message translation using the *Google Translate* widget shown on the right. This connection fires whenever a new message is received, and the *GO* button on the *TouchMe* widget is clicked. This button was taken from the GUI of the translation widget via an *Add* action in the context menu. The dashed arrow represents control flow between the *Geppeto TouchMe* widget and the *Google Translate* widget which was defined using a *Click* action from the context menu, and the two full arrows represent data flow of the original and the translated message, both of which were defined using *Copy* and *Paste* actions.

All the actions defined in the *TouchMe* widget are stored in a two dimensional table, as shown in figure 2.5. In this table, time flows from top to bottom and from left to right. This table allows more advanced consumers to reorder independent actions and organize them for parallel execution. While the shown example application is inherently sequential and no meaningful reordering can be done, this can sometimes provide significant improvements in performance as many actions on widgets are executed by communicating with services over the Internet which can be a time consuming process.

Many other programmable widgets can be used in *Geppeto*, some of which are briefly described below. The *TickMe* widget allows consumers to schedule parts of an application to certain moments in time and to repeat actions periodically. The *LocateMe* widget generates events based on the user's GPS location using HTML geolocation APIs found in modern web browsers. A more general concept of events is supported by the *TriggerMe* widget that allows

[3]The presented example application can be reused and extended with message sending logic using the *Send Message* widget also available in *Geppeto* to create a complete chat application that allows two consumers to communicate in different languages using a machine translation service.

consumer applications to interact with applications created by professional software developers by exchanging messages through events. Widgets supporting redirection of control flow based on conditions, synchronized execution of consumer applications [40], and introducing redundancy to increase the application's reliability [41] are also available.

Several consumer assistant widgets have recently been proposed for *Geppeto*. The *ReliabilityOptimizeMe* machine assistant provides consumers with easy to understand information about the reliability of individual widgets [42]. The *ReliabilityAssistant* machine assistant allows consumers to easily identify the weak points in the reliability of their composition and replace them with a more reliable alternative [41]. The *TutorMe* human-based assistant analyzes the procedural knowledge encoded in the partial composition the consumer is working on and in all the completed consumer applications to identify and recommend peer-tutors from the consumer community that can potentially help the consumer solve a particular problem [43]. Finally, the *NextComponent* machine assistant that is proposed in section 4.2 of this dissertation aims to alleviate the component discovery challenge in consumer computing by recommending components during the composition process by comparing the partial composition the consumer is currently working on with previously defined consumer applications.

Chapter 3

Review of Related Research

This chapter provides an overview of related research in several topics of interest for the problem of component recommendation in development of composite consumer applications. An overview of recommender systems technology is given in section 3.1. The chapter concludes with a look at related approaches in assisted application development across mashups and software engineering in section 3.2.

3.1 Recommender Systems

The two key concepts in recommender systems are *users* to which recommendations are made and *items*, which are the object of the recommendation process. There are many reasons to include a recommender system in a product such as a web site [44]. For example, quality recommenders tend to increase the number and diversity of items sold because users are exposed to items they possibly like based on their previous purchases or interactions with the system. Furthermore, a well-designed recommender that provides useful information to users is likely to increase overall user satisfaction and user loyalty as it increases the perceived value of the entire product.

Recommender systems are typically based on *ratings*—a value that represents a user's opinion of an item. Ratings can be either *implicit*, in which case they are generated from the user's interaction with the system (for example, buying an item or adding it to a wish list), or *explicit* in which case users are asked to express their opinion about certain items [44]. Explicit ratings are most commonly collected using a 5-star scale where one star indicates a strong dislike for the item and five stars indicate the user considers the item to be excellent in some sense [45].

In recent years, binary ratings where users just indicate if an item is good or bad (like with *YouTube* videos), and even unary ratings where users simply either *like* an item or assign no rating to it (like in *Facebook*) have risen in popularity.

Users are motivated to provide ratings for several possible reasons [46]. First, by rating items, users update their profile and can expect to get better recommendations in return. Second, some users may simply wish to express their opinion about an item or contribute ratings in the interest of helping others. This particular motivation is most evident when there is a social component to a web site that uses a recommender system. Finally, users might maliciously try to influence the rating of certain items that they want to succeed or fail. This is called a *shilling* attack.

In his seminal paper [46], Herlocker identified six distinct tasks that recommenders can be used for. Out of these six, two tasks are by far the most common. *Annotation in context*, often called the *prediction* task [47–49], aims to help users reason about items in an existing context, for example by adding relevancy annotations to existing links or messages. The *find good items* task, often called *recommendation*, is designed to find items that would be of interest to the user and might normally not be visible. Note that these tasks are closely related as it is often possible to generate recommendations if a system can make accurate predictions of item ratings.

Other tasks, which can be viewed as specializations of prediction and recommendation, include recommending new items, recommending items for a group of users (for example, a movie that a group of friends might like), recommending items in an ordered sequence, finding *all* good items, finding good items in a restricted context (for example, a movie that the user might like within a certain genre), helping users browse with no intent to buy, etc.

Recommender systems rose to prominence in the mid 1990s, closely following the growth of the World Wide Web [50]. Nowadays, all major e-commerce web sites like *Amazon*[1] [51] and *eBay*[2] [52, 53] use recommender systems extensively [54]. Recommender systems are also ubiquitous in other domains, including music (e.g. *Pandora*[3]), movie (e.g. *Netflix*[4]), reading (e.g. *GoodReads*[5]) and video clip (e.g. *YouTube*[6] [55–57]) recommendations. Recommender systems technology has also found important uses in online and mobile advertising [58, 59].

The most common categorization of recommender systems [48, 60–64] identifies *content-*

[1] amazon.com
[2] ebay.com
[3] pandora.com
[4] netflix.com
[5] goodreads.com
[6] youtube.com

based filtering (CBF) and *collaborative filtering* (CF) as the two overarching themes in recommender system techniques, while systems combining these two ideas or several approaches from each are categorized as *hybrid* recommenders. These three approaches are described in the remainder of the section.

3.1.1 Content-Based Filtering Recommender Systems

Content-based filtering is centered around describing items with *metadata* which makes it possible to compare items for similarity. For example, in a movie recommender, the metadata might include the movie's genre, its leading actors, the director, etc.

The content-based approach to recommender systems has roots in the information retrieval research area which predates recommender systems. However, the key difference between CBF and traditional information retrieval is the concept of a *user profile* [65]. User profiles model the user's tastes and interests and are used extensively together with item descriptions when making recommendations.

Balabanović [66] identifies four central concerns of a CBF recommender system. The first two concerns are item description and user profile representations. Third, the designed needs to define a function $predict(i, u)$ which determines the relevance of item i to user u. Fourth, a function $update(i, u, f)$ which updates the user profile u given the user's feedback f on the item i should also be defined.

Due to its roots in information retrieval, CBF is often applied to *documents*, such as e-mail or *Usenet* messages [67–69]. Therefore, the most common representation for both user profiles and item descriptions are *vectors* containing the *term frequency-inverse document frequency* (TF-IDF) weight [70] of the most informative words that describe a user's profile or a document.

The term frequency $TF_{i,j}$ of keyword k_i in document d_j is defined as

$$TF_{i,j} = \frac{f_{i,j}}{\max_x f_{x,j}}, \tag{3.1}$$

where $f_{i,j}$ is the absolute frequency of the keyword in the document, and the maximum in the denominator is taken over all keywords appearing in the document. This normalization ensures that the measure is more resilient to different document sizes.

To measure a keyword's overall discriminative value in a corpus of documents, the inverse

document frequency of keyword k_i IDF_i is computed as

$$IDF_i = \log \frac{N}{n_i}, \tag{3.2}$$

where N is the total number of documents in the system, and n_i is the number of documents that contain keyword k_i.

Finally, the weight in the vector coordinate corresponding to keyword k_i and document d_j is computed as

$$w_{i,j} = TF_{i,j} \times IDF_i. \tag{3.3}$$

User profile vectors can start out as *null-vectors* to which *relevance feedback* is applied as the user starts rating documents [71]. The simplest form of relevance feedback is a linear update rule [66] with[7]

$$update(\boldsymbol{i}, \boldsymbol{u}, f) = \boldsymbol{u} + f\boldsymbol{i}. \tag{3.4}$$

In words, the user profile vector is simply extended by the document vector that the user rated with a rating f, which is scaled so that it can be either positive or negative. Later ratings can be given higher importance in the user profile by continuously decaying old values, for example by multiplying them on a daily basis with a real constant slightly below one.

Finally, the predicted relevance of a document $\boldsymbol{i}$ for a user $\boldsymbol{u}$ is then defined with a vector dot-product as

$$predict(\boldsymbol{i}, \boldsymbol{u}) = \boldsymbol{i} \cdot \boldsymbol{u}. \tag{3.5}$$

The *Fab* recommender of web pages [64] is often cited as an example of early CBF recommender systems, although the authors clearly state that it is a hybrid system, and it does in fact rely heavily on collaborative filtering as well. Its content-based filtering component is based on the framework described above.

InfoFinder [67, 68] is a CBF recommender for *Lotus Notes* that uses a somewhat different approach. Document metadata is generated by heuristic extraction of significant phrases from text, relying on the tendency of users to somehow highlight important parts of the text. User profiles are built by explicitly asking users to select several sample documents that they consider interesting. In this process, a decision tree representing the user's profile is built, and recommendations for new documents are made using a variant of the *ID3* decision tree algorithm [72].

[7] Here, $\boldsymbol{i}$ and $\boldsymbol{u}$ are typeset in bold as they are vectors.

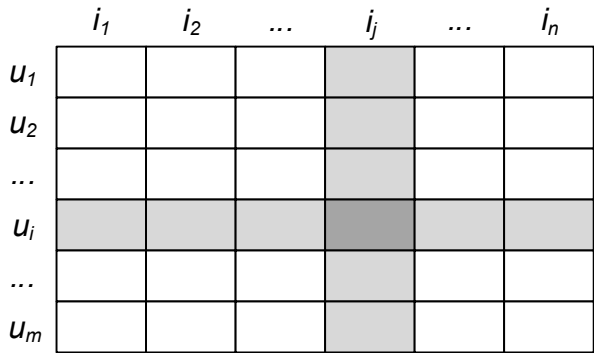


Figure 3.1: The *user-item* rating matrix used in collaborative filtering recommender systems.

Another approach based on TF-IDF can be seen in *NewsWeeder*, which is a *Usenet* CBF recommender system [69]. This system is interesting because it uses the *minimum description length* (MDP) principle [73] to make recommendations, which is a probabilistic framework based on Bayesian inference.

Despite its relative simplicity, pure CBF is not widely used in state-of-the-art recommender systems as it has three well-known limitations [60–62, 64, 65]. First, content analysis is naturally limited in scope. While techniques adopted from information retrieval typically work well for text documents, automatic feature extraction is significantly more difficult for other domains such as music and video. Furthermore, it is challenging to construct an item representation that captures most of the interesting concepts related to the item. Second, CBF systems overspecialize in the sense that an item is never recommended to users who haven't rated a similar item. Due to this fact, the recommendations have low novelty and especially serendipity, which are both becoming increasingly important characteristics of recommender systems. Finally, CBF recommenders suffer from the *new user* problem. For a new user that has rated few items, good recommendations can't be made as the user's profile is very incomplete. This can cause users to give up on using a system before it can actually start performing well.

3.1.2 Collaborative Filtering Recommender Systems

Unlike content-based filtering, collaborative filtering does not analyze items beyond their identifier. Instead, collaborative filtering centers around the *user-item rating matrix* $\mathcal{R}$, shown conceptually in figure 3.1. The rating matrix has m rows corresponding to the users of the system, and n columns corresponding to items. The set of users is denoted by $\mathcal{U}$ and the set of items by $\mathcal{I}$. The value in the jth column of the ith row is denoted by $r_{i,j}$ and represents the rating that the ith user has given to the jth item, or is left blank if this user hasn't yet rated the

item.

Domains in which collaborative filtering can provide good results should have the following sets of properties [47]. First, there should be many ratings per item. Rating matrix sparsity is one of the most significant challenges in collaborative filtering. For the same reason, there should be more users than items to be recommended. As most users tend to rate only a small fraction of all the available items, many users are required to gather a sufficient number of ratings. Furthermore, items should be interesting for a longer period of time so that ratings can accumulate.

Second, groups of users should have similar tastes pertaining to the item set in question. Specifically, users with very unique interests may find collaborative filtering algorithms less useful, which is known as the *grey sheep* problem. Collaborative filtering algorithms assume that user tastes persist. Therefore, users whose taste changes can expect to get bad recommendations, at least for a period of time.

Collaborative filtering recommender systems are further classified as *memory-based* or *model-based* depending on how they use the rating matrix [74]. Memory-based CF systems, sometimes also called *neighborhood-based* systems [62], use the rating matrix to find similar users or similar items and use the ratings that are present in the matrix to predict ratings that are not. When predicting a rating $r_{i,j}$, two distinct approaches are possible. In the *user-based* or *user-user* approach, the recommender computes similarities between user i and all other users that have rated item j, and then combines the ratings that other users have given to item j using these similarities. Conversely, in the *item-based* or *item-item* approach, the recommender computes similarities between item j and all other items that have been rated by user i, and uses these similarities to combine the ratings that user i has given to these other items. These approaches are often combined, for example by taking a linear combination of their predicted rating as the final prediction.

In both approaches, two key decisions need to be made—the definition of similarity, and the method of combining ratings based on computed similarities. Note that similarities are computed solely on the basis of ratings present in the rating matrix, and no additional properties of neither users nor items are considered.

Perhaps the simplest measure of similarity between users or items is the cosine of the angle between their corresponding row or column vectors from the rating matrix. In general, the

cosine between two vectors can be computed easily using the dot product of the vectors

$$cos(\boldsymbol{a}, \boldsymbol{b}) = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|}. \tag{3.6}$$

When applied to item-based collaborative filtering, the similarity of two items x and y is then

$$sim(x, y) = cos(\boldsymbol{r_{*x}}, \boldsymbol{r_{*y}}) = \frac{\sum_{u \in \mathcal{U}} r_{u,x} r_{u,y}}{\sqrt{\sum_{u \in \mathcal{U}} r_{u,x}^2} \sqrt{\sum_{u \in \mathcal{U}} r_{u,y}^2}}, \tag{3.7}$$

where $\boldsymbol{r_{*x}}$ is used to denote the column vector of the rating matrix associated with item x.

A limitation of simple cosine similarity as presented above is that it doesn't take into account that different users use the range of the rating scale differently—for example, some users might never rate items with less than two stars, and some might give five-star ratings more liberally than others. Several modifications exist that eliminate this limitation. When using *adjusted cosine similarity*, the rating vectors are normalized by subtracting the mean rating the user has given to items from each rating of that user. With this change, similarity of items x and y is defined as

$$sim(x, y) = \frac{\sum_{u \in \mathcal{U}} (r_{u,x} - \bar{r_u})(r_{u,y} - \bar{r_u})}{\sqrt{\sum_{u \in \mathcal{U}} (r_{u,x} - \bar{r_u})^2} \sqrt{\sum_{u \in \mathcal{U}} (r_{u,y} - \bar{r_u})^2}}, \tag{3.8}$$

where $\bar{r_u}$ is the mean rating user u has given to his rated items.

Alternatively, the *Pearson correlation coefficient* (PCC) can be used instead of the cosine, using the formula

$$sim(x, y) = \frac{\sum_{u \in \mathcal{U}} (r_{u,x} - \bar{r_{*x}})(r_{u,y} - \bar{r_{*y}})}{\sqrt{\sum_{u \in \mathcal{U}} (r_{u,x} - \bar{r_{*x}})^2} \sqrt{\sum_{u \in \mathcal{U}} (r_{u,y} - \bar{r_{*y}})^2}}, \tag{3.9}$$

where $\bar{r_{*x}}$ denotes the mean of the vector $\boldsymbol{r_{*x}}$, i.e. the mean rating of item x. PCC measures the linear correlation between two vectors and its value is between negative one, which indicates that the vectors are facing in exactly opposite directions, and positive one which indicates that the vectors are parallel. When the vectors are orthogonal, PCC is zero.

When applied to user-based CF, adjusted cosine similarity and PCC are the same measure. Many other definitions of similarity between feature vectors exist [75], but are rarely used in recommender systems. In a recent comparison of six different similarity measures on a use-case in the social network *Orkut*[8] [76], adjusted cosine similarity performed the best.

Having determined the required similarities, a prediction for the rating user a might assign

[8] orkut.com

to item i can be computed using the formula

$$\hat{p_{a,i}} = \bar{r_a} + \frac{\sum_{u \in \mathcal{U}} (r_{u,i} - \bar{r_u}) \cdot sim(a,u)}{\sum_{u \in \mathcal{U}} sim(a,u)}. \tag{3.10}$$

It is assumed that all similarities are nonnegative, i.e. that neighbors with negative correlations are removed from the computation.

Several challenges in memory-based CF are described in the literature [49, 74]. Primarily, this approach is particularly sensitive to rating matrix sparsity. Along with the fact that sparsity may make it impossible to make recommendations for some users or to recommend some items, sparsity can also cause unreliable recommendations, for example when the user has only a few rated items in common with all neighbor users. The second significant challenge is *scalability*—the ability of the recommender to provide recommendations with many users and items in the system. Model-based collaborative filtering algorithms aim to address these challenges.

Instead of directly using the whole rating matrix for each query, model-based CF recommender systems *learn* predictive models from the matrix, and then use these models to respond to queries more quickly. Some of the most frequently used approaches in model-based CF are Bayesian inference [77], which is a probabilistic framework whose parameters are estimated from the rating matrix, and clustering [51, 78], where users or items are clustered into groups by similarity so that neighbor identification can be done efficiently. More advanced approaches include *Markov decision processes* and dimensionality reduction techniques like singular value decomposition [79].

Tapestry [80], developed at *Xerox PARC*, is often cited as the first collaborative filtering recommender system, although it did not actually use any of the techniques that are considered collaborative filtering in the modern sense. The system was designed to process streams of electronic documents and allowed users to annotate these documents. Based on these annotations, users could then filter documents by typing in queries in a specially designed query language. What made *Tapestry* "collaborative filtering" is the fact that annotations were shared between users within the system. Therefore, a user could filter for documents that his friend considered worthwhile reads by naming that friend in the query.

One of the first true CF recommender systems was *GroupLens* [81, 82] which predicted the level of interest each user might have for a *Usenet* newsgroup article. Contemporary systems of *GroupLens* include *Ringo* [83] in the music domain and *Video Recommender* [84] which was an e-mail based movie recommender system.

In 2006, *Netflix*, which was at the time just a web-based DVD-rental company, started an open competition for improving their movie recommender algorithm *Cinematch* [85]. *Cinematch* was a PCC-based collaborative filtering recommender that was used to predict user ratings of movies which were then recommended for rental. When insufficient data was available to make a prediction, the average rating of a move was used instead. *Cinematch* was able to achieve around 10% lower root mean squared error (RMSE) [46] in rating prediction than an algorithm that simply used average ratings for all predictions would have. The goal of the competition was to reduce the RMSE by a further 10%.

The key initial contribution of this competition to the general science of recommender systems was the provided dataset. *Netflix* gave researchers 100 million timestamped ratings on a 5-star scale given by 480 thousand users over 18 thousand movies that were collected between 1998 and 2005. This dataset was orders of magnitude larger than anything researchers previously had access to. Furthermore, the prize for achieving the final goal was one million dollars, with 50 thousand dollar prizes awarded yearly to the leading system until the goal was achieved. Both factors created a lot of interest in the competition.

Three years later, in 2009, the team that also won both yearly progress awards just crossed the 10% benchmark to win the competition. The algorithm they created was called *Bell-Kor's Pragmatic Chaos*, and was, in fact, based on contributions of several teams that joined forces during the competition. The stunning fact about this algorithm is that it uses over a hundred individual results that are then *blended* together to produce the final prediction [86]. To get these individual results, many different approaches to the prediction problem are used, including neighborhood-based models, regression models, matrix factorization and many more. It is this blending of multiple predictors that allows the algorithm to achieve great predictive accuracy. However, the authors state that many fewer predictors might actually be necessary, and with just the blend of 11 different results, an 8% improvement over *Cinematch* can be achieved.

Interestingly, *Netflix* never used this winning algorithm in production [87]. The practical benefit of the algorithm was not sufficiently high to warrant the engineering cost to deploy it. The largest contributor to that fact was that both *Netflix* and its recommenders had evolved significantly in the three years of the competition and moved towards video streaming, which is significantly different than DVD-rentals. The most important take-away from this is that recommender systems need to evolve with their use-case. Even spending three years of immense research and engineering effort on solving the "wrong problem" might not provide any

economic benefit. However, although *Netflix* might not have directly profited from the winning algorithm, by releasing a large industry dataset and organizing this competition, they have helped significantly advance the state-of-the-art of recommender systems.

3.1.3 Hybrid Recommender Systems

Hybrid recommender systems combine collaborative and content-based methods with the goal of avoiding some of the limitations of either of those approaches individually [64, 88–93]. The simplest type of hybrid recommender system uses several separate approaches whose results are then combined. For example, the *Daily Learner* system [94], which provides personalized news access, selects predictions with the highest confidence from several employed approaches.

A more common approach is to add content-based characteristics to a CF recommender. A classical example of such a system is *Fab* [64, 66], which is a web page recommender. Recommending web pages clearly doesn't satisfy the domain property requirement for collaborative filtering that there are more users than items, and is thus significantly impacted by rating sparsity problems. To overcome these challenges, *Fab* combines the standard CF approach with a content-based approach that has been described in subsection 3.1.1.

3.2 Related Approaches to Assisted Application Development

The aim of this section is to introduce and discuss several tools and approaches to component recommendation and related problems in mashup development, *Yahoo Pipes* and software engineering described in the literature. Specifically, approaches in assisted mashup development are described in subsection 3.2.1. Then, subsection 3.2.2 discusses several tools proposed by researchers to ease development of *Yahoo Pipes*. Finally, similar approaches in the larger field of assisted software engineering are presented in subsection 3.2.3.

3.2.1 Assisted Mashup Development

Over the past decade, as mashups rose in popularity, a significant number of mashup tools have been developed [95–98]. In addition to *Yahoo Pipes*, some of the most significant mashup tools that have been discussed in the literature are *Google Mashup Editor*, IBM's three related products *QEDWiki*, *Damia* [99, 100] and *Mashup Center*, *Microsoft Popfly*, Intel's *MashMaker*

[101], *Apache Rave*[9] [102], *Apatar*[10], *Exhibit* [103], *Deri Pipes*[11] [104–107], *Vegemite* [4], *Marmite* [108], *Karma* [109], *d.mix* [110], and *C3W* [111].

With the notable exception of *Yahoo Pipes*, all of the mashup tools created by large IT companies and some of the smaller tools have since been discontinued. While most of these tools were graphical and often based on *programming by demonstration*[12] [29], more recently, mashups are mostly discussed in the context of *Yahoo Pipes* (which are graphical) and *ProgrammableWeb*[13] which promotes creating mashups by programming to APIs. However, several interesting systems were developed to further simplify the usage of some of the mentioned mashup tools, and some of them are described in the remainder of this subsection.

An interesting approach to assisted mashup development which shares several features with the work described in this dissertation is presented in [112]. The authors extracted a dataset of 2786 mashups with 821 distinct components from the *ProgrammableWeb* mashup repository. Several key properties of this dataset are important for proper understanding of the presented results. First, *ProgrammableWeb* mashups are small on average, with only 3.3 components. Second, the distribution of component frequency is extremely skewed with the most popular 1.5% of components (i.e. 12 components) accounting for about 50% of the total component frequency.

Four recommender algorithms were compared on this dataset. The first was an unadaptive algorithm called *Top Popular* which recommends the most popular components for each query. This algorithm is completely equivalent to the baseline algorithm *MostPopular* described in section 7.4 of this dissertation.

Next, two collaborative filtering neighborhood algorithms [62] were also employed. In order to use collaborative filtering methods for this problem, compositions were identified with *users* in the framework, while components were naturally identified with *items*. More specifically, the training dataset of compositions was represented by a binary matrix $\mathcal{R}$ where each row corresponds to one composition and each column to one component. The value $r_{i,j}$ in the ith row and jth column of that matrix is then set to 1 if and only if composition i contains component j.

In the first used collaborative neighborhood algorithm, called *Cosine Neighborhood*, the

[9] rave.apache.org
[10] www.apatar.com
[11] pipes.deri.org
[12] Sometimes also referred to as *programming by example*.
[13] programmableweb.com

similarity between two components x and y was defined as

$$s_{x,y} = \frac{\text{\# compositions using both components}}{\sqrt{\text{\# compositions using } x}\sqrt{\text{\# compositions using } y}}. \tag{3.11}$$

As matrix values are binary, it can be easily seen that equation 3.11 actually represents the cosine between the column vectors of the matrix corresponding to the two components. However, while this value typically measures similarity of items in a collaborative filtering algorithm, it is important to consider the semantics of this similarity in this particular application of the approach. As the authors explain themselves, and as is evident from the definition, this value in fact measures how likely two components are to appear in the same composition, and does not imply they are similar in some other way, for example in their functionality.

The second collaborative neighborhood algorithm that was analyzed, called *Direct Relations*, uses a simpler definition of component similarity defining

$$s_{x,y} = \text{\# compositions using both components.} \tag{3.12}$$

Given a new input composition u, for each candidate component x, both algorithms then compute the relative rating of the component i as

$$\hat{r}_{u,x} = \sum_{y} s_{x,y} r_{u,y}. \tag{3.13}$$

Each term in the sum is the contribution of a component y to the total relative rating—if this component y is frequently used with the candidate component x in the same composition, as measured by $s_{x,y}$, and is also used in the input composition u, in which case $r_{u,y}$ is 1, it will contribute to making x a more likely recommendation. Components with the highest relative rating are then recommended to the user.

Finally, the fourth algorithm, called *PureSVD* [113], was based on latent factor models [114] which are closely related to singular value decomposition. In this algorithm, the "rating matrix" $\mathcal{R}$ is approximated by the factorization

$$\hat{\mathcal{R}} = \mathcal{U} \cdot \Sigma \cdot \mathcal{Q}^T. \tag{3.14}$$

Given n users, m items and f latent factors, $\mathcal{U}$ and $\mathcal{Q}$ are orthonormal $n \times f$ and $m \times f$ matrices

representing the left and right singular vectors associated with the f singular values of $\mathcal{R}$ with the highest magnitude, and Σ is a diagonal $f \times f$ matrix where these singular values are stored. Using this decomposition, relative ratings of candidate compositions are computed as

$$\hat{r}_{u,x} = \boldsymbol{r}_u \cdot \mathcal{Q} \cdot \boldsymbol{q}_x^T, \tag{3.15}$$

where $\boldsymbol{r}_u$ is the component usage vector of the input composition, and $\boldsymbol{q}_x$ is the xth column of Q. Ten latent factors were chosen for evaluation using cross validation.

The algorithms were only evaluated on recall, which can be interpreted as the probability that a relevant component is recommended, but how exactly relevant components were chosen is not completely clear. The authors state that all rated components are considered relevant, but don't describe how this changes for different queries. While the average number of relevant components for each query is not reported, it can be concluded that it is not more than 2 as the best performing algorithm achieves almost 50% recall when recommending only one component per query. Furthermore, it is briefly mentioned that one component from a composition is removed when it is supplied as input to the recommender, and that recommendations are then somehow analyzed against that removed component, but this point requires further elaboration.

Regardless, evaluation results show several interesting properties relevant to the research presented in this dissertation. First, the simplest *Top Popular* algorithm is very competitive with the seemingly more powerful algorithms. It achieves over 40% recall and outperforms the *PureSVD* algorithm by up to 10%, regardless of the number of components recommended per query. On the other hand, the *Cosine Neighborhood* and *Direct Relations* algorithms outperform *Top Popular* by up to 10%, but the difference decreases when a larger number of components is recommended[14].

The obvious cause of this behavior is the skewed component frequency distribution in the dataset—several very popular components are ubiquitous in the dataset. The authors attempted to isolate this effect by removing the 12 most popular components from contention and rerunning the experiments. With this change, the recall of all algorithms decreased by up to 20%, and the *Top Popular* algorithm was affected the most. However, it still remained competitive and within 15% of the other algorithms, and especially so when more components are recommended per query. In this setting, the *Direct Relations* algorithm performed significantly better than the other algorithms, by up to 10% with fewer than 6 recommendations per query.

[14] Up to 20 components are recommended per query.

While a mapping of the component recommendation problem into a collaborative filtering setting provides a rich toolbox for attacking the problem, the semantics of this mapping are questionable. First, while it is common that rating matrices are sparse, with only 3.3 components in a composition on average, it can be expected that the $\mathcal{R}$ matrix is extremely sparse. Second, the inclusion of a component in a composition is inherently different from assigning a rating to a movie or a song. However, this approach shows promise and potential for future research.

OMLETTE is a mashup system built on top of *Apache Rave*. In [115], two extensions to that system that aim to simplify mashup creation for end-users are presented. First, *ACE* (*Automatic Composition Engine*) attempts to find out the user's intentions through a series of questions which the authors state are generated based on previous answers and in the context of the availability and functionality of components. The gathered answers are then used to create a SPARQL [116, 117] search query which is submitted to a *semantic widget registry* [118]. *ACE* then selects appropriate widgets to get the desired functionality and creates a mashup that models the user's goal. The actual effectiveness of *ACE* in creating compositions that closely match what the user wants was not evaluated, and it seems likely that good results will be constrained to some domains where widgets are semantically annotated, though the authors indicate that the current implementation bases widget understanding on textual attributes such as titles, descriptions, and tags. In the user study where 44 users were instructed to create a simple mashup, using *ACE* actually led to an increase in completion time, but this is attributed to usability issues and the learning curve of the tool.

In addition to *ACE*, the paper presents *PR* (*Pattern Recommender*) that is aimed at helping users finish mashups created by *ACE* or build mashups incrementally by recommending appropriate building blocks and how they should be connected. Preexisting mashups are analyzed to extract so called *composition patterns* which are categorized into *widget co-occurrence* and *multi-widget* types. The most applicable patterns are recommended to the user, but the process of identifying them is not precisely defined. Unlike *ACE*, the *PR* tool did produce a significant positive effect on completion time in the user study, but more extensive evaluation is required to show its applicability to various tasks in different problem domains.

The *MatchUp* [119,120] system attempts to introduce the concept of autocompletion, which has long been ubiquitous in browser location bars [121, 122], textual search [123, 124], integrated development environments (IDEs) [125, 126], and is even being used in UML mod-

eling [127], into mashup development. Specifically, given a partial composition as input, *MatchUp* recommends components and connections between components that can be useful for completing the mashup. Higher level connections that potentially connect a larger number of components are referred to as *glue patterns*. It is assumed that the sets of components and glue patterns can be arranged in a graph where arcs represent what is called *syntactic inheritance*, which is defined in terms of more specific interfaces, i.e. one component inherits from another if it can be used in its place. Compositions are represented as points in the vector space spanned by these components and glue patterns. *MatchUp* then finds the k closest glue patterns to the composition and recommends them to the user. Users of *MatchUp* still need to consider semantics when accepting a glue pattern recommendation, as the system does not try to reason about semantics when generating recommendations, but instead, similar to the work presented in this dissertation, takes advantage of previous experiences and knowledge of other users who've already invested the time and effort to understand certain components and how they can be connected.

While *MatchUp* is based on similar ideas about extracting and reusing knowledge from previously created compositions as the work presented in this dissertation, the two approaches cannot be directly compared. The assumed inheritance relations are only informally defined and it is unclear if they can be automatically extracted from compositions in different composition systems, which is a premise to the usefulness of this approach. Furthermore, there seems to be an exponential number of possible glue patterns so defining the graph of components and glue patterns might prove challenging in practice.

The prototype *MatchUp* implementation on top of IBM *Mashup Center* was evaluated through a user study with only ten users working on a single problem, but the results show promise that the system could be useful in practice.

MashupAdvisor [9] decomposes compositions into *concepts*, which, while not precisely defined, are more finely grained than components, representing individual interface elements. The repository of previously completed compositions is preprocessed to estimate probabilities that a concept appears as an input or as an output, and several conditional probabilities of concept co-occurrence as inputs or outputs within the same composition. These probabilities are then used at query time to score every possible concept as a potential output for the given partial composition. The concepts are then ranked and the top ranking concepts are recommended to the user.

When the user selects a recommendation, *MashupAdvisor* creates a plan for including the new concept and possibly other supporting concepts into the composition. The authors indicate that a *semantic matcher* is used to compute levels of semantic similarity between concepts which are then used both when making recommendations and during planning. This semantic matching is based on textual comparison of concept descriptions called *tags* which are assumed to be assigned to each concept. The similarity score of two concepts A and B is defined as

$$score(A, B) = \frac{synNum}{\max\{len(A), len(B)\}},$$

where $synNum$ is the number of synonymous terms in the tags of A and B, while the function len computes the number of terms in the tag.

While both the initial generation of concepts and their descriptions as well as the planning process are underspecified, evaluation was performed on a synthetic dataset of compositions that matches composition size and component frequency distributions of *ProgrammableWeb*. While it is not specified how exactly the composition process was simulated, evaluation results show that the proposed approach outperforms a random concept recommender, though it is difficult to infer the statistical significance of that difference. Furthermore, response times of up to several minutes were observed in experiments where several thousand compositions were stored in the composition database, which the authors identify as a major target for future work.

3.2.2 Tools for *Yahoo Pipes* Development

This subsection focuses on several tools that aid users in *Yahoo Pipes* development.

In [98], the authors introduce *refactoring* [128, 129] to *Yahoo Pipes*. Refactoring is a process in which a software artifact is changed without changing its functionality with the aim to improve its quality by reducing complexity, increasing maintainability or testability, etc. Specifically, the developed system automatically detects suspect constructs in a pipe, often called *code smells* in software engineering in general and also in the paper, and produces the refactored pipe as output.

Pipes are modeled with directed acyclic graphs in the natural way, with vertices representing modules and directed edges representing wires of the pipe. Ten code smells are described and defined using a small formal language over the graph representation for testing various properties of the pipe and categorized into three classes of smells. First, *laziness smells* include

those deficiencies that can easily appear if insufficient care is taken when building a pipe, and include the *unnecessary module* smell, which is a module that doesn't affect the output of the pipe, and the *noisy module* smell, which occurs when a module has extra fields that are not used or duplicate fields that can be removed. The second class of deficiencies is called *redundancy smells* and tries to mimic code duplication defects often found in software. Examples of redundancy smells are *duplicate strings* where the same string constant is used in multiple places and *duplicate modules* where several modules of the same type are used when a single module can accomplish the same task. Finally, the third class of deficiencies is called *population-base smells* and includes pipe paths that do not conform to idioms commonly found in the Pipes community, perhaps performing some functionality using a different order of operations than is the norm in the most used pipes.

A user experiment was conducted to assess the effect of these code smells in pipes on their perceived quality [130]. 14 out of the 50 initial subjects were identified as *end-users* with limited education in computer science through a qualification process, and these users were then used in the remainder of the experiment. Given a deficient pipe and its refactored version, 63% of users answered they prefer the refactored version, while 24% preferred the version with code smells. When asked to predict the output of a pipe, 80% of the answers were correct for refactored pipes and 67% for unchanged deficient versions. Additionally, it took users an average of 68% longer to analyze the deficient version of a pipe.

Refactorings that fix the identified deficiencies were defined in terms of graph transformations. For example, the *Pull Up Module* refactoring extracts duplicated strings into a new module that then provides the value via wires. Of the 8051 pipes scraped from the *Yahoo Pipes* website, 6503 or nearly 81% had at least one smell. After the refactorings were applied to the dataset, only 16% of pipes still contained defects. The remaining defects could not be removed automatically.

These results are interesting because they form the basis for real-time refactoring support in *Yahoo Pipes* which would certainly be beneficial to users as suggested by the recent comprehensive study of the Pipes community [131].

MARIO [132] is a *Yahoo Pipes* development tool that allows users to specify goals in terms of *tags* in a query text box and outputs one or several complete pipes that try to implement that goal, additionally giving users the option to modify recommendations. While *MARIO* uses custom services to implement the functionality of *Yahoo Pipes* modules and does not directly

interact with *Yahoo Pipes*, this seems like an implementation detail, so Pipes terminology is used here for simplicity. The recommended pipes are compositions of what are called *flows*, which are basically patterns of connected Pipes modules. Flows need to be defined and semantically annotated before the system can be used. Furthermore, every service or feed that can be used as a data source to the pipe has to be semantically annotated as well. *MARIO* tries to fulfill query goals using a planner called *SSPL* designed specifically for stream processing [133]. While the problem of finding optimal plans with *SSPL* is PSPACE-complete, the planner can find plans within a few seconds in practice.

While the approach seems promising, requiring even light-weight semantic annotation with tags and simple taxonomies limits its scope to use cases where the numbers of available components and connection patterns is limited. Furthermore, *MARIO* was not evaluated for successfulness in any way. Specifically, the authors only demonstrate that the system responds to short queries with up to 200 annotated feeds known to the system in up to one second. However, the key question of whether the recommended plan actually successfully solves the query is not addressed.

Baya [15] is a plug-in for *Yahoo Pipes* developed by the same group of authors as *OMLETTE* that recommends what the authors refer to as *reusable composition knowledge* in form of patterns from a predefined list of several pattern types, including *parameter value pattern* which fills input parameters of a module and *connector pattern* which represents a connection between a pair of modules. Actual patterns are extracted from a repository of existing pipes, as well as dynamically from incoming queries, although this process is not described in detail. Recommendations are then made using pattern matching algorithms which are only mentioned, presumably due to lack of space. This approach is interesting as it tries to address nearly all aspects of Pipes development, from selecting modules and connecting them, to specifying all module parameters. However, the description lacks significant details and only response time has been evaluated on a very small set of 303 pipes.

3.2.3 Assisted Software Engineering

Assisted software engineering, also often referred to as *Computer-Aided Software Engineering* (*CASE*) has been a research topic for decades [134–137]. The many advances in the state of the art of software engineering like better tools and development processes have been followed with significant increase in software complexity. Specifically, developers are today faced

with large code bases that can depend on many libraries built with different technologies [138]. Therefore, assisting software developers in their endeavors is as important today as it ever was, and assisted software engineering still remains an active area of research [11–13, 139–146]. This subsection gives a brief overview of some of the tools proposed in this area that are most relevant to the work presented in this dissertation.

To help users navigate complex APIs, *Strathcona* applies heuristic matching of developer-defined code fragments with previously written code to produce examples that help the developer understand an API and compete his task. The system was developed for Java, and implemented as an Eclipse plug-in. The *protoexamples*, as the authors call these snippets that get recommended to users, are extracted automatically from an arbitrary code base. Specifically, the extraction is done at the method level—every method is turned into a *protoexample*.

To use the system, the developer is expected to highlight a fragment of relevant code and invoke the plug-in through the context menu of Eclipse. This code fragment is then used to formulate a query, which is then sent to the *Stratahcona* server which replies with 10 recommended snippets for the user to consider. To facilitate structural matching of code fragments, both the input fragment and the *protoexamples* are represented with a *structural context*, which is a set of facts about the piece of code. These facts mostly represent the types used in the fragment in various ways, either through method signatures declared in the fragment, field access, method invocation, etc., and supertypes of so called *declaring types* which host the declared methods.

Four relatively simple heuristics for matching structural contexts are described, including the *CALLS* heuristic, which matches method calls, and the *USES* heuristic, which matches types used in method invocation, field references, etc. Given a query structural context, candidate *protoexamples* are subjected to these four heuristics individually, each of which produces a ranked list of 100 best matches. Then, the four lists are merged and the 10 best *protoexamples* as judged by all four heuristics are recommended to the user.

The recommended examples are presented to the user in several different ways, including a UML-like diagram of classes used in the example. Furthermore, *Strathcona* offers a high degree of explainability as each example is accompanied with a generated description of why it was recommended. For example, the explanation might say that the recommended fragment used the same type and called the same method as the query fragment.

A thorough evaluation of the system is provided. The database used in the experiments contained approximately 3.7 million lines of example code with a total of around three million

facts. With a database of this size, response times ranged between 0.3 and 3 seconds, which is sufficiently fast for real-time interaction. Fast response times are achieved primarily through the use of a heavily indexed *PostgreSQL* database, which then allows finding relevant examples with one complex query per heuristic, which the database management system can optimize well.

The experiments were designed with the aim to test if the system provides useful examples for completing tasks, how it compares to straightforward alternatives like using *grep* and Eclipse's built-in search functionality, and if developers with little knowledge about an API can generate sufficiently good query fragments for the system to provide quality examples of the usage of that API.

While the design of the experiments seems excellent, they all suffer from very small sample sizes as only one or two developers were used in each experiment. However, there is clear indication that the system might be useful in practice.

SPARS-J [11] is a Java class retrieval system based on class usage relations. A piece of software is modeled as a composition with a weighted directed graph which is called a *Component Graph*. Vertices of the graph represent software components—specifically, Java classes in the particular implementation of *SPARS-J*—while arcs represent that one component uses another in some way, e.g. through inheritance, interface implementation, field access or method invocation. Arc weights are computed based on the particular type of usage relation and its frequency in the software database. Based on the arc structure of the graph, node weights are defined as weighted sums of neighboring nodes' weights, giving a system of linear equations. The computed node weight is then identified with the rating of a component. These ratings are used to rank components when users search the component database using textual queries, such that the recommended components are both relevant to the query and also important in that component collection.

The system was evaluated through several experiments. In one experiment, the source code of the Java Software Development Kit (JDK) 1.4.2 was used as the code base for ranking components. The JDK uses a total of about 6100 classes in total, and it took *SPARS-J* 20 minutes to compute the ranked component archive. With a different dataset created by combining many open source projects found on *SourceForge*[15] that contained around 180000 classes, computing the ranked component archive took two days, and the generated database required 5.5GB of disk

[15] sourceforge.net

space. However, this is a preprocessing step that has to be performed only once to set up the system for use in a certain domain, and responses to queries were instant for both datasets. Unsurprisingly, `java.lang.String` and `java.lang.Object` were the two highest ranked components on both datasets.

The effectiveness of *SPARS-J* was compared to that of *Google*[16] and *Namazu*[17]. *Namazu* is a full-text search engine that can be used over custom datasets, and was given the same dataset of source code as *SPARS-J*. Queries submitted to *SPARS-J* were augmented with the words "java" and "source" when they were submitted to *Google* and *Namzu*. A total of 10 queries were submitted, including "quicksort", "binaryseaerch" and "zip deflate". For each query, only the top 10 results returned by the search engines were considered, thus matching the number of components recommended by *SPARS-J*. Then, the precision of the results was evaluated as the ratio of useful results for the given query. Somewhat unsurprisingly, a statistically significant advantage in precision was observed for *SPARS-J*. The authors state that other code search systems were not publicly available for testing, so that is why the system was compared to general purpose search tools.

Interestingly, the system was also tested in two real-world companies[18]. After having used the system for some time, developers of both companies were asked to complete a survey with several questions pertaining to the usefulness of *SPARS-J*. The results indicate that the system was useful, and one of the companies adopted it for further use.

CodeBroker [12] tries to facilitate reuse by delivering task-relevant and personalized recommendations to a developer. It was implemented in Emacs and supports Java. Component granularity is at the method level. Specifically, components are represented by their Javadoc and method signature. When a developer starts entering a new Javadoc or a method signature, *CodeBroker* searches for good recommendation candidates using *Latent Semantic Analysis* (LSA) [147]. The 20 best matches are recommended to the developer, and the results are continually updated as more information is available.

The most interesting aspect of this research is the way in which recommendations are made more task-relevant and personalized. During a development session, *CodeBroker* assembles a *discourse model* which defines components that should not be recommended during that session, based on the course of the session. Furthermore, *CodeBroker* maintains a *user model* for

[16] `www.google.com`
[17] `www.namazu.org/index.html.en`
[18] Daiwa Computer and Suntory Limited.

each developer. A user model is used to represent a particular developer's preferences and is updated through the developer's interaction with the recommender.

Finally, to conclude the subsection, *RASCAL* [13] and *Javawock* [145] are briefly described in parallel because they share several important properties. Both tools are aimed towards Java development, and specify component granularity on the method and class level, respectively. Recommendations of methods and classes are based on a form of collaborative filtering where "users" represent pieces of code, while items naturally map to the objects of recommendation.

The authors of *RASCAL* introduce an interesting concept of a *Knowledge-intensive Integrated Development Environment* (*KIDE*). They identify the lack of tool support as the key factor for low levels of reuse in the industry, and propose to address this issue by integrating a method recommender system inside the IDE, which is, in itself, a fairly common approach. This extended development environment aims to address the *no-attempt-to-reuse* problem where developers often miss reuse opportunities because they are unaware that a certain problem has already been solved.

The evaluation methodology for *RASCAL* and *Javawock* is similar to the one presented in this dissertation—a fragment of a piece of software is used as the input to the recommender, while the recommender's output is compared to the hidden part of the code, and precision, recall and the F_1 score are reported.

Chapter 4

Component Recommendation in Consumer Computing

The aim of this chapter is to further specify the problem of component recommendation, and specifically as it pertains to consumer computing. The chapter is divided into two sections. First, the process of component recommendation based on composition similarity as it applies to a wide array of composition systems is analyzed and described in section 4.1 Then, in section 4.2, this component recommendation process is explored further in the context of consumer computing and *Geppeto*.

4.1 Component Recommendation Process

The process of component recommendation based on composition similarity takes place both inside the composition workspace where the interaction with the user happens, and inside the composition engine that actually analyzes compositions and chooses components to be recommended. The focus of discussion in this section is on the part of the component recommendation process inside the composition engine with only a simple model of a composition workspace.

The component recommendation process is shown in figure 4.1. The end goal of the recommendation process is to assign scores to *candidate components* based on the similarity between the input partial composition and previously completed compositions so that the best candidates can be recommended to the user. The process begins inside the composition workspace when the partial composition the user is working on is encoded into some structured representation

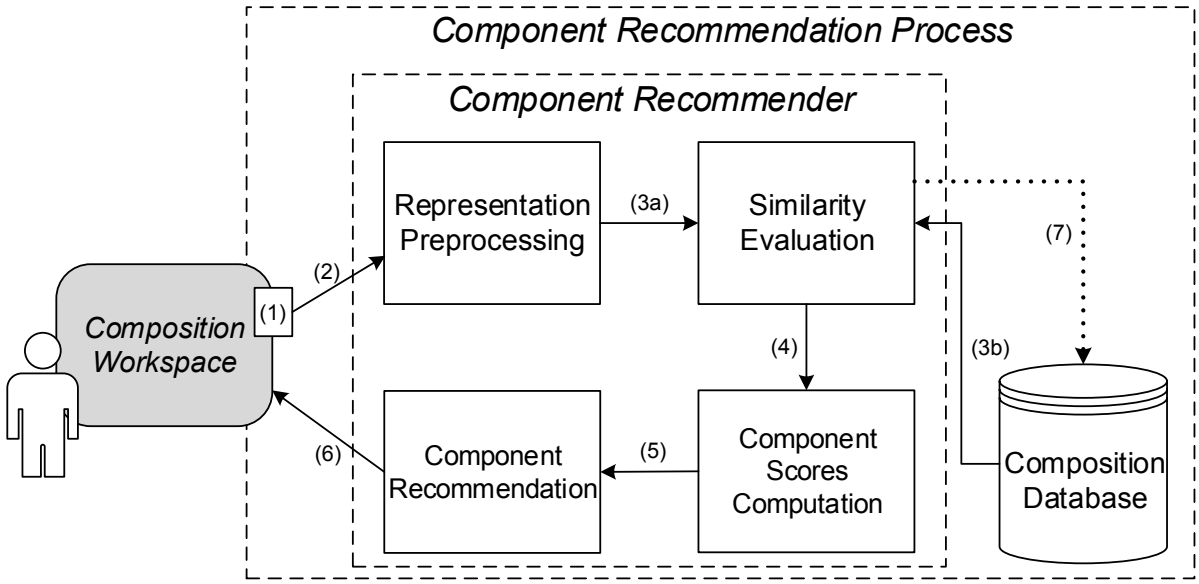


Figure 4.1: The general process of component recommendation based on composition comparison.

(1). This structured representation is then given to the component recommender system that is part of the composition engine. The component recommender is based on four key steps: *representation preprocessing*, *similarity evaluation*, *component scores computation* and *component recommendation*. These steps are shown as rectangles in the figure.

First, the structured representation of the input partial composition (2) is preprocessed into some recommender-specific representation. Note that in most practical implementations of a component recommender, all representation preprocessing could be done inside the composition workspace, before a query is sent to the recommender, i.e. steps (1) and (2) could actually be merged into a single step. However, these steps are logically distinct. A structured representation of a partial composition is certainly required, no matter how the component recommender is actually implemented. On the other hand, it can be beneficial to represent various partial compositions differently. For example, a complex representation that requires a lot of processing could be used for small compositions, while a simpler representation could be used for larger compositions.

After representation preprocessing, the recommender compares the partial composition (3a) and previously completed compositions stored in the *composition database* (3b). It is assumed that all required representations of a composition are available in the composition database. The recommender represents the similarity between the partial composition and a database composition using two pieces of information. First, a real valued *similarity score* is computed. Second, for many definitions of similarity, the recommender can additionally determine how different

components in the partial and database composition affect the similarity of these compositions. This additional information can be useful for computing component scores.

When evaluating similarity, the recommender can either compare the partial composition to all the compositions in the composition database or only to some of them by preprocessing the database in some way to eliminate dissimilar compositions quickly. In this dissertation, the former model is used. The primary reason for this choice is that most composition systems in use today, including both *Geppeto* and *Yahoo Pipes*, have composition databases of several thousand compositions at most, which can be easily processed for each query. Possible strategies for transitioning to filtering the composition database for similarity evaluation are discussed in section 8.4.3.

Once the partial composition has been compared with the compositions from the database, the recommender computes component scores (4). Again, as with similarity evaluation, component scores can be computed based on similarities with all the compositions in the database or only some fraction of the most similar compositions. This process is referred to as *similarity filtering in component scores computation* or simply as *similarity filtering* in the remainder of the dissertation. The effects of similarity filtering on recommender quality are evaluated extensively and discussed in section 8.4.4.

Finally, several components with the highest score are selected (5) and displayed to the user in the composition workspace (6). The appropriate number of components to recommend per query depends on the composition system in question and UI design considerations. The effects of this choice are discussed further in section 8.4.5.

After responding to a query, the recommender can optionally store information about the partial composition and computed similarities in the composition database (7). This information can then be used to speed up subsequent queries that result from future changes to that same partial composition.

4.2 Application to Consumer Computing and *Geppeto*

This section focuses on the interaction between consumers and the component recommender system, specifically within *Geppeto*. The remaining parts of the recommendation process are the topics of the next two chapters.

Component recommendations are provided to consumers through a machine assistant wid-

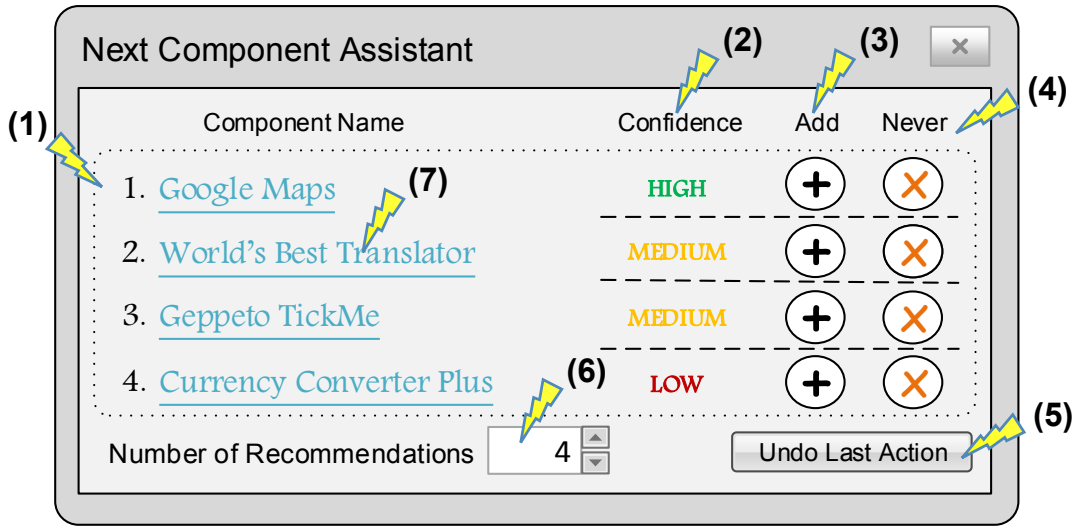


Figure 4.2: The *Geppeto NextComponent* widget.

get called *NextComponent*. After the *NextComponent* widget is added to the composition workspace, it monitors the consumer's composition actions and automatically queries the recommender system and displays recommendations after each composition action made by the consumer.

The design of the *NextComponent* widget is guided by four main goals. First, like all widgets for consumer computing, *NextComponent* should be intuitive to consumers and not require any special training or additional explanations to be used effectively. Second, due to the fact that recommendations will change often, the widget should not distract consumers with dramatic graphical changes in its appearance. In other words, the basic graphical user interface of the widget needs to be simple. Third, the most basic information about recommendations should be available at a glance so that users can notice interesting recommendations even when they are not actively looking for a recommendation and perhaps already have an idea which component to add, or maybe are not even considering augmenting their current application, but are instead only using it. Fourth, a more thorough consideration of recommended components should be easy for consumers and require little action. Specifically, once consumers decide to actively explore the given recommendations, they should be given as much information about the component and its recommendation as possible.

The proposed *NextComponent* widget is shown in figure 4.2. As is usually the case for recommender systems [138], the recommended components are presented to the consumer as a list of items (1) where each item is the name or title of the widget. The list is numbered to indicate an order to the consumer—more highly recommended components are displayed at the

top of the list, while less likely useful components are near the bottom of the list. Furthermore, the recommender's confidence in each particular recommendation is shown to the right of its name (2).

Three levels of confidence are possible, ranging from *high*, through *medium*, and down to *low* confidence. The level of confidence is further emphasized by the use of green, red and yellow colors which carry well known semantics from the physical world to make it easier to scan recommendations at a glance.

Consumers can immediately choose to *add* the component to their workspace (3). This is expected to be a common action for two distinct reasons. First, the consumer might be familiar with a particular recommended component and recognize it as useful by its name alone. Second, the consumer could just add a component whose name seems promising and try it out. The cost of a false positive in this case is very low, since it is easy to remove the component if it is not what the consumer was hoping for. This *trial and error* approach is used by people in many different contexts [148–151].

On the other hand, consumers might also identify widgets that they consider bad for various functional or nonfunctional properties. In this case, this widget can be removed from all further recommendations by clicking the button in the *Never* column of the assistant's interface (4). This button is also color coded to indicate to the consumer that this operation has long-term consequences. The list of ignored widgets can be edited in the settings of the *NextComponent* widget, which is not shown in the figure.

Furthermore, to allow consumers to easily undo an action that they might have done by accident or quickly reconsidered, an *undo* button is available at the bottom of the assistant (5). The undo concept is ubiquitous in many applications consumer use on a daily basis and is thus intuitive to most people [152, 153]. In addition to undoing a *Never* action on a widget, the last widget addition can also be undone with this button.

The number of components that are recommended per query can be regulated either by typing in the wanted number of recommendations or by using the arrow buttons next to the text box (6). The *NextComponent* widget automatically resizes to accommodate the desired number of recommendations. The default number of recommended components is a topic for further research, but evaluation results indicate that three components per query provide the best balance between precision and recall (see section 8.4.5).

Finally, recommendations are displayed as *hyperlinks* (7), as indicated by their bluish color

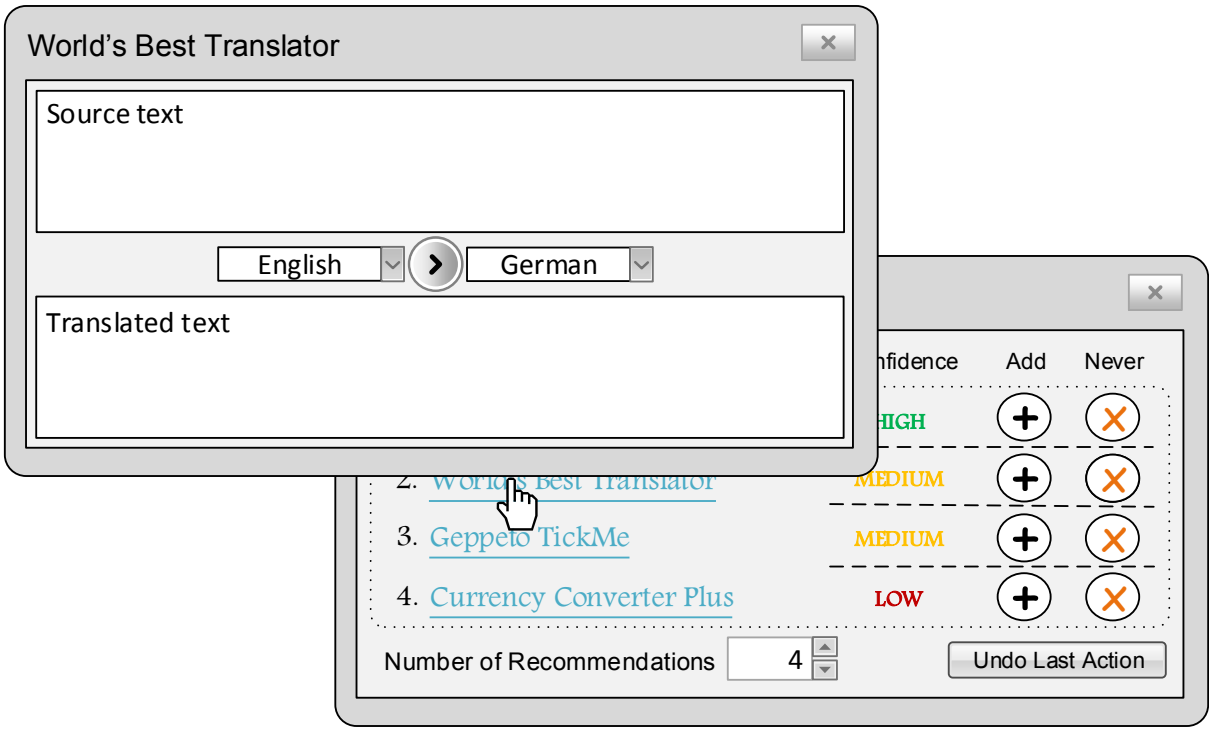


Figure 4.3: The mouseover interaction with the *Geppeto NextComponent* widget.

and underlined text, which are also ubiquitous on the Web and very familiar to consumers. A hyperlink indicates that a piece of text can be interacted with to get more or related information. Two forms of interaction are supported by the *NextComponent* assistant. First, if the consumer hovers the mouse pointer over a recommended widget's name, a preview of the widget's interface is shown above the mouse pointer. As widgets usually have simple and intuitive user interfaces, a glimpse at the GUI can provide a lot of additional information about what the widget can do and allow the consumer to make a more informed decision whether to add the widget or not. An example of this interaction is shown in figure 4.3

The second interaction with recommendation hyperlinks is activated through a click action. When a hyperlink is clicked, the interface of the *NextComponent* assistant is replaced with a more detailed description of the recommended widget. An example of the click interaction is shown in figure 4.4. A preview of the clicked widget is shown at the top, similar to the mouseover interaction. Under the recommended widget's interface, additional information about the widget is displayed, including the text description of the widget provided by the author, the author identifier and the number of times that widget has been used in compositions. Several other pieces of information could be provided in this area, including various nonfunctional properties of the widget like its reliability.

Under this basic information, the consumer can find the explanation why this particular

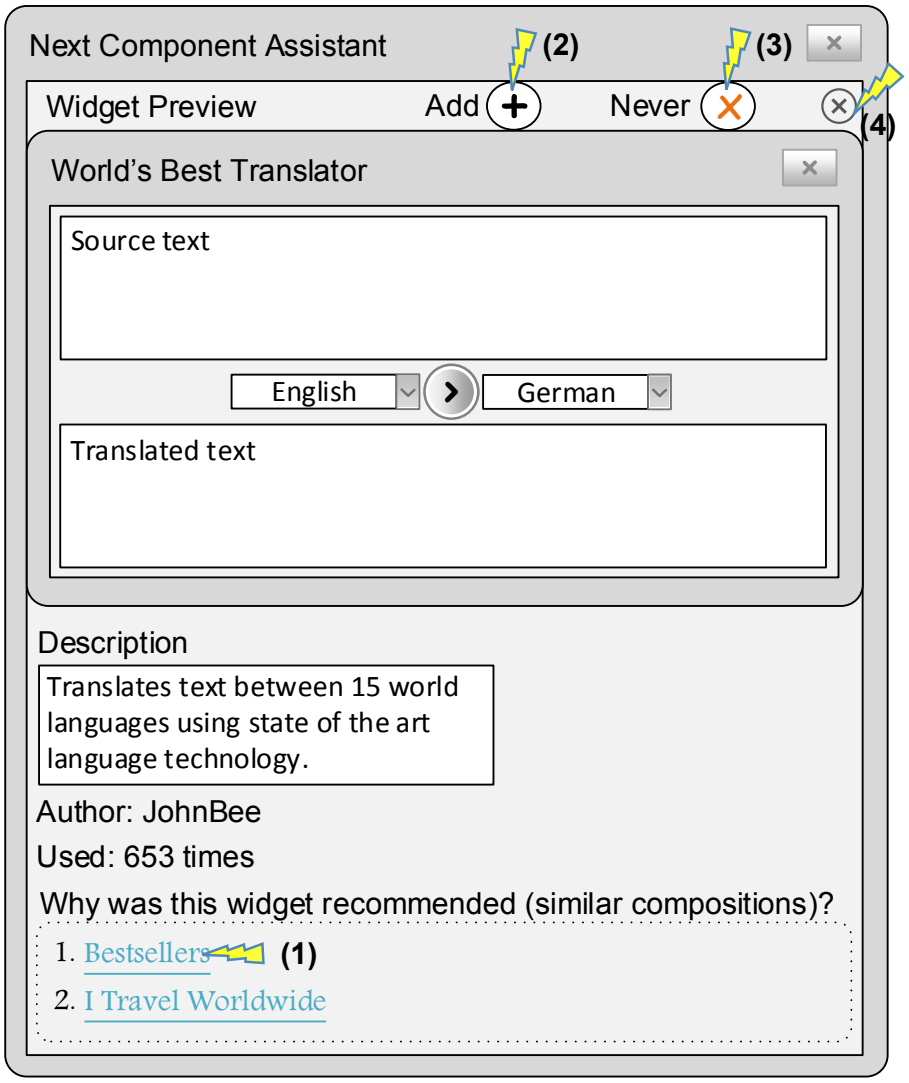


Figure 4.4: The click interaction with the *Geppeto NextComponent* widget.

widget was recommended. Here the *NextComponent* assistant lists the compositions that use the recommended widget and are also the most structurally similar to the consumer's partial composition that was used to query the recommender. By clicking on a provided composition hyperlink (1), the consumer can examine the similar composition in a new window and potentially gain further insight into how the recommended component can be used in a real composition. On occasion, consumers might find that the composition they are trying to build already exists, though this is not a direct goal of the *NextComponent* assistant.

Finally, having considered the recommendation in detail, the consumer can either add it to the workspace (2), add the widget to the ignore list so that it never gets recommended again (3), or close the details view and return to the default view of the *NextComponent* assistant widget (4).

Chapter 5

Modeling Composite Applications

This chapter describes the base representation of composite applications and several simpler variants used in component recommenders presented in this dissertation.

Modeling composite applications using some variant of a formal graph is supported by at least two major factors. First, a graph-based model is easily applicable to various kinds of composite applications in different domains as both components and interactions between them naturally correspond to graph concepts. Specifically, components in the application can be modeled with vertices, while edges provide a way to model relationships between components. Furthermore, it is easy to generate the graph model of a composite application if a structured representation of the composition is available, which is frequently the case.

Second, graph-based models have been used successfully in several other research areas where the structure of an object is of interest, such as in the analysis of the World Wide Web [154–157] and social networks [158–160], computer vision [161, 162] and computational chemistry [163, 164]. While the algorithms developed in these areas are mostly not directly applicable to composite applications due to the differences in graph semantics and graph database properties, approaches and experiences used in developing those algorithms provide valuable guidelines. Graphs are also prominent in case-based reasoning (CBR) [165–168]. Many problems involving composite applications, including component recommendation, can be cast as CBR problems.

To associate component identifiers with vertices, the chosen graph formalism should include vertex labels. This is essential because components carry the functionality of the composition, especially in consumer computing where components are general-purpose applications with high-level functionality. Edge labels could be used to differentiate between, for example, data

flow and control flow connections between components. However, it is likely that a specific pair of components will always have the same types of connections between them, so differentiating these types is less important. For example, a component like the *ReceiveMessage* widget in *Geppeto* will almost invariably have a control flow connection to a *TouchMe* widget or some other control widget that executes a part of the composition for each new message, and only data flow connections to all other widgets that process the received message in some way.

Undeniably, having this information be represented in the model provides useful information for component recommendation. For example, if the recommender can recognize this regularity for the *ReceiveMessage* widget and the input partial composition contains a *ReceiveMessage* widget with one control flow connection, a recommendation of widgets that can be used to process messages will likely be more useful than a recommendation of widgets that control this processing when a message is received.

However, including edge labels in the model is challenging for two main reasons. First, in the general case, a pair of components can be connected multiple times with different connection types. To properly model this with labeled edges, the graph model would have to either allow parallel edges or label edges with a set of connection types, both of which significantly complicate the model and increase the complexity of computing most similarity measures. Second, evaluating the usefulness of edge labels requires a dataset in which connection types can be easily deduced, and such a dataset of sufficient size is not publicly available. For these two reasons, edge labels are not considered in this dissertation.

While a graph can represent many useful structural properties of a composite application, and these properties can typically be explored in time linear in the size of the graph, exact similarity measures for many interesting definitions of similarity over graphs are computationally expensive or intractable. For example, both subgraph isomorphism [169] and graph edit-distance [170] on unlabeled graphs are well know to be NP-hard. With labeled graphs, these computations become tractable but are still challenging to do quickly in the presence of many duplicate labels, which turns out to be common in composite applications. Therefore, a graph-based model is used in this dissertation as a basis for representing composite applications in a computer, but three other models on a lower level of abstraction that retain less structural information are also considered. These models can be generated from the graph representation of a composition, so in the remainder of the dissertation, a composition is identified with its graph representation where no confusion can arise. All four of the explored models of composite

applications are defined formally in the sections that follow.

5.1 Graph Model

Let $\mathcal{T}$ be the finite set of identifiers of all the components available in a certain composition system. A composition of these components is then modeled with an unweighted directed graph with labeled vertices

$$
\begin{aligned}
G =& (V, A, \lambda), \text{ where} \\
& V = \{1, 2, \ldots, |V|\} \text{ is the finite set of vertices,} \\
& A \subseteq V \times V \setminus \{(u, u) : u \in V\} \text{ is the set of arcs, and} \\
& \lambda : V \to \mathcal{T} \text{ is a labeling function.}
\end{aligned}
$$

The number of vertices and the number of arcs[1] in a graph will be denoted with n and m, respectively. Subscripts will be used when discussing multiple graphs in the same context, so that, for example, $n_G = |V_G|$ is the number of vertices of a graph G.

Vertices are used to model components of the composition, while arcs represent connections between the components. Note that by mapping components to vertices, the interface of the component is abstracted away. For example, if components are graphical widgets, an action on any interface element is mapped to an arc incident to the vertex representing the whole component.

Parallel arcs and self-loops are not allowed in the model. Parallel arcs would arise when one component is connected to another component in more than one way, for example, if one component takes several inputs from another. However, as noted earlier, local interaction within a fixed pair of components is often syntactically identical, and modeling the connection with a single arc captures the interaction. Therefore, when converting a composition to the graph representation, parallel connections are replaced with a single arc or possibly with one arc in each direction if the connections between components are bidirectional. Self-loops are simply ignored as they provide no useful information for the purposes of component recommendation.

[1] Arcs are *directed edges*, which are also sometimes called *arrows*.

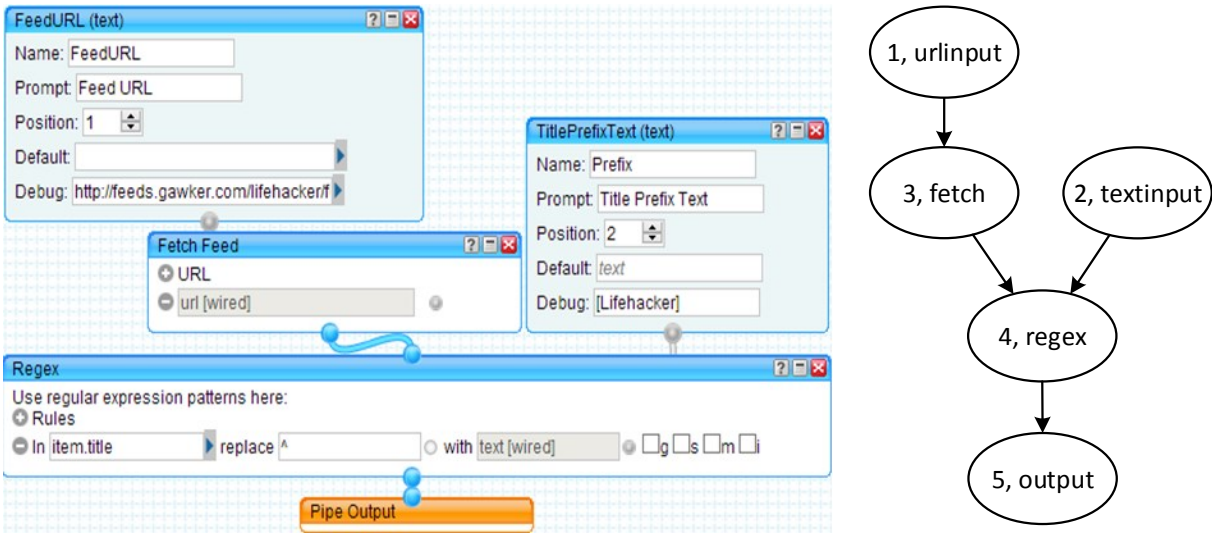


Figure 5.1: The Feed-Item Title Prefixer *Yahoo Pipes* composition and its graph model.

For each vertex $u \in V$, the sets of incoming and outgoing arcs are defined as

$$in(u) = \{(u, v) : (u, v) \in A\}, \text{ and}$$
$$out(u) = \{(v, u) : (v, u) \in A\}$$

respectively. Using these sets the *indegree* and *outdegree* of a vertex u are defined as

$$indegree(u) = |in(u)|, \text{ and}$$
$$outdegree(u) = |out(u)|.$$

The labeling function λ assigns component identifiers to the vertices of the graph model as labels. In the remainder of the dissertation, a vertex will sometimes be identified with its label when that is clear from context.

As most compositions tend to produce sparse graphs, adjacency lists are the most suitable graph representation.

A straightforward example of converting a *Yahoo Pipes* pipe to its graph model is shown in figure 5.1. In *Yahoo Pipes*, components are called *modules*. The five Pipes modules are modeled with the five vertices of the graph. The vertex labels are module names, and are shown to the right of the vertex index on the vertex itself. As is universally the case for *Yahoo Pipes*, the graph model is basically the data flow graph of the pipe.

A slightly more complex example is shown in figure 5.2. The *Geppeto* composition that

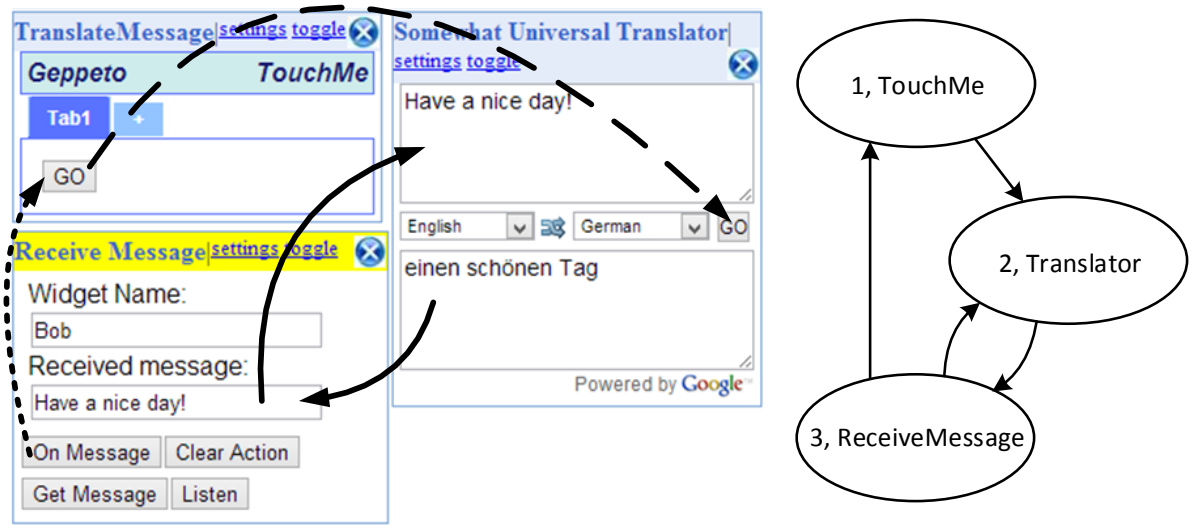


Figure 5.2: The Message Translator *Geppeto* application and its graph model.

translates received messages using the *Google Translate* widget that was introduces in section 2.3 is shown on the left. The three-vertex graph on the right is the graph model representation of this composition. Note that while it is important for the correctness of this composition that these operations are ordered in time, the model does not capture this order in any way.

5.2 Component Sequence Model

Since the goal of a component recommender is to recommend *components* to users of a composition system, a component-centric simplification of the described graph model is defined and used in one of the recommender algorithms. The basic information about the functionality of a composition can be retrieved from just the list of components used in the composition. Such a list is easily generated from the graph model. As this is a sequence model, the order in which the components are listed is significant. One order that provides useful information about the composition is the order in which the components were added to the composition. However, this information is typically not available in composition databases, which is also the case in both datasets used in this dissertation.

Another useful order that preserves some structural information about the composition is a simple generalization of a *topological order*. A topological order of the vertices of a directed graph can be though of as a permutation of vertex indices such that, for each vertex, all of its ancestors in the graph come before it in the permutation. Put another way, if the vertices of a directed graph are aligned along a straight horizontal line in topological order, all the arcs are

```
1:  function GeneralizedTopologicalOrder(G)
2:    componentList ← []                          ▷ Initialize to empty list.
3:    while G is not empty do
4:      candidates ← set of all vertices of G with minimal indegree
5:      u ← an arbitrary vertex from candidates with maximal outdegree
6:      componentList.append(λ(u))
7:      G.remove(u)
8:    end while
9:    return componentList
10: end function
```

Algorithm 5.1: GeneralizedTopologicalOrder algorithm: Returns a list of components of the graph G in generalized topological order.

directed from left to right.

It is obvious that vertices of a directed graph can be topologically ordered if and only if the graph is acyclic. Composite applications can have cyclic relationships between components, so graphs representing compositions may contain cycles. Therefore, the concept of a topological order is generalized for graphs with cycles as described with high-level pseudocode in Algorithm 5.1. For an acyclic graph, the set *candidates* in line 4 will always contain the vertices with indegree zero, i.e. with no incoming arcs. Otherwise, the minimal indegree might be nonzero, but the set is always well defined. From these vertices with minimal indegree, a vertex with maximal outdegree in the original graph is picked. This criterion is largely arbitrary, but is chosen as vertices of higher outdegree might be more central to the functionality of the composition. Remaining ties, if any, are broken arbitrarily, for example by picking the lowest index vertex. In line 6, the label of the picked vertex which is the identifier of the component that the vertex represents is appended to the output component list. Finally, line 7 removes the picked vertex and all its incident arcs from G.

This algorithm can be implemented using a priority queue in $O(m \log n)$ time [171].

The presented example graph in figure 5.1 is acyclic and the component sequence generated by the GeneralizedTopologicalOrder algorithm is

$$< urlinput, textinput, fetch, regex, output > .$$

On the other hand, the graph in figure 5.2 contains two cycles. Initially, the minimal indegree is one. As vertex 3 has a higher outdegree than vertex 1, it is selected as the first component in

the list. The final component sequence representation in generalized topological order is

$$< ReceiveMessage, TouchMe, Translator > .$$

5.3 Feature Vector Models

Santos et al. [172] reported that a feature vector representation produced very similar clusters to a more complex graph representation when clustering workflows. The feature vectors contained the task labels of a particular workflow. A similar idea is applied here to the component recommendation problem.

To define a feature vector model, it is necessary to define the vector space in which objects will be represented. For composite applications, a simple vector space that is analogous to the mentioned task labels vector space is one where each vector dimension counts the occurrences of a particular component in a composition.

Formally, let $T_i \in \mathcal{T}$ be the identifier of the ith component available to the users of a composition system in some fixed order over $\mathcal{T}$. A composition modeled with a graph $G = (V, A, \lambda)$ can then be represented in a $|\mathcal{T}|$-dimensional vector space with the vector $\boldsymbol{x}_{comp}$, where

$$\boldsymbol{x}_{comp}^{(i)} = \#\{u : u \in V, \lambda(u) = T_i\}$$

is the value of the ith coordinate of the vector, where the hash symbol is used to represent set cardinality. This vector is referred to as the *component vector* of the composition.

Note that this model retains no structural information about the composition beyond the used components. One way to reintroduce basic structural information is to include arc information in the vectors. Toward that goal, the vector space is extended to $|\mathcal{T}| + |\mathcal{T}|\,(|\mathcal{T}| - 1) = |\mathcal{T}|^2$ dimensions. The first $|\mathcal{T}|$ coordinates still specify component frequency, while the remaining $|\mathcal{T}|\,(|\mathcal{T}| - 1)$ coordinates represent arc counts with a fixed order over all possible arcs. A vector in this space is referred to as the *structure vector* of the composition.

When there are no repeated components in a composition, the structure vector contains the same structural information as the graph model, i.e. the graph model can be reconstructed from the structure vector. However, since all coordinates are treated in the same way, this structural information is implicit rather than explicit as in the graph model.

Even though these vector spaces can be large, both the component vector and the structure

vector of a composition are sparse, i.e. most of their coordinate values are zero. They can, therefore, be represented with a sorted list of key-value pairs where the keys are the nonzero coordinate identifiers. This makes the size of the representation $O(n)$ and $O(n+m)$, respectively, both of which are independent of $|\mathcal{T}|$.

For example, the component vector for the graph shown in figure 5.2 could be represented with the list

$$[ReceiveMessage : 1, TouchMe : 1, Translator : 1],$$

where an alphabetical order of component identifiers is assumed.

On the other hand, the structure vector for the same graph could be represented with the list

$$[ReceiveMessage : 1, TouchMe : 1, Translator : 1, ReceiveMessage \rightarrow TouchMe : 1,$$
$$ReceiveMessage \rightarrow Translator : 1, TouchMe \rightarrow Translator : 1,$$
$$Translator \rightarrow ReceiveMessage : 1],$$

where arrows between component names are used to represent arcs.

To make the vector representations easier to read and save on horizontal space, the ones in both component vectors and structure vectors are omitted in the remainder of the dissertation, i.e. the frequency of a component or an arc is only explicitly shown if it is greater than one.

Chapter 6

A Framework for Component Recommendation Based on Composition Structural Similarity

This chapter describes how the models presented in chapter 5 can be used to dynamically recommend components during composite application development. First, a general component recommendation method that can be the basis for any algorithm that solves this problem by comparing composition representations is presented in section 6.1 Then, in subsequent sections, four recommender algorithms that are based on this general method are defined, in increasing order of complexity. These algorithms are referred to as *structural algorithms* in the remainder of the dissertation as opposed to the simple statistical algorithms defined in section 7.4 that are not based on this general component recommendation method. A summary of the basic properties of the four structural recommender algorithms is presented in section 6.5. Afterwards, the challenge of choosing algorithm parameters is discussed in section 6.6. Finally, to conclude the chapter, the key steps of all four algorithms are presented on a simple example in section 6.7.

6.1 Component Recommendation Method

The component recommendation method presented in this section is based on the component recommendation process that has been described in section 4.1, and provides a framework within which every component recommendation algorithm based on composition similarity can be defined. The base structured representation used in the recommendation method is the graph

```
1: function MAKECOMPONENTRECOMMENDER(graphDB, PREPROCESS, EVALUATESIMILARITY, COMPUTESCORES, RECOMMEND)
2:   reprList ← []
3:   for all Q in graphDB do                          ▷ representation preprocessing
4:     reprList.append(PREPROCESS(Q))
5:   end for

6:   function COMPONENTRECOMMENDER(P)
7:     P_r ← PREPROCESS(P)                           ▷ representation preprocessing
8:     simList ← []
9:     for all Q_r in reprList do
10:      sim ← EVALUATESIMILARITY(P_r, Q_r)          ▷ similarity evaluation
11:      simList.append((sim, Q_r))
12:    end for
13:    scores ← COMPUTESCORES(simList, P_r)          ▷ component scores computation
14:    return RECOMMEND(scores, P_r)                 ▷ component recommendation
15:  end function

16:  return COMPONENTRECOMMENDER
17: end function
```

Algorithm 6.1: MAKECOMPONENTRECOMMENDER function: Returns a component recommender over the composition database based on the parameter functions PREPROCESS, EVALUATESIMILARITY, COMPUTESCORES, and RECOMMEND.

model from section 5.1. Therefore, it is assumed that all compositions in the composition database are already represented using the graph model.

The goal of the recommender is to, given a graph model of a partial composition that the user is working on, compute real valued scores for each component in $\mathcal{T}$ and then recommend several of the highest scoring components as the most likely to be useful for completing the application. In this recommendation process, four key steps were identified: *representation preprocessing*, *similarity evaluation*, *component scores computation*, and *component recommendation*. These steps are organized in a component recommendation method as described with pseudocode in Algorithm 6.1.

The instantiation of a recommender algorithm from the component recommendation method is described using a higher order function MAKECOMPONENTRECOMMENDER that takes as input four functions that specify the four key steps in the recommendation process and returns the function COMPONENTRECOMMENDER, defined in lines 6–15, which defines the recommender algorithm.

The PREPROCESS function takes a graph model of a composition as input and returns a representation that is suitable for a particular algorithm, along with the original graph represen-

tation in a pair data structure. The input graph representation is preserved as it can be useful when computing component scores. The PREPROCESS function is applied to the entire composition database once, during recommender initialization. Lines 2–5 in the pseudocode initialize *reprList* with the representations of all preexisting compositions. Note that in a practical implementation, compositions are added to the database in an on-line fashion, but this doesn't significantly complicate the problem.

When discussing composition similarity and computing component scores, the letter P is used to denote the input partial composition and the letter Q to denote the specific database composition P is being compared to. The subscript r is used to denote that the composition is represented in some way that might differ from the base graph representation.

The COMPONENTRECOMMENDER function takes as input the graph representation of P and closes over *reprList* and the four parameter functions. Using the PREPROCESS function, the required representation P_r is extracted from the graph in line 7.

The recommender then goes through the preprocessed composition database and evaluates the similarity between P_r and Q_r using the EVALUATESIMILARITY function in line 10. The EVALUATESIMILARITY function should return a *similarity object* representing the similarity of the two compositions with a real number which is called the *similarity score*, along with an algorithm specific description of how that similarity score was computed. This additional information can be used to compute component scores based on their role in achieving the computed similarity score. The computed similarity object is stored in the list *simList* along with the corresponding composition Q_r in line 11.

The completed *simList* is then passed to the COMPUTESCORES function in line 13, along with P_r. COMPUTESCORES should return a *scores* data structure that maps component identifiers to their scores. In its simplest variant, the COMPUTESCORES function would initialize all component scores to zero, go through the entire *simList* and update component scores based on every computed similarity. However, for definitions of similarity that do not discriminate between similar and dissimilar compositions with a large absolute difference in similarity scores, it can be beneficial to compute component scores based only on some of the most similar database compositions, as the large number of dissimilar compositions might otherwise significantly affect which components are recommended. Therefore, for all four algorithms presented in this dissertation, a parameter BP, which stands for *best percent*, is introduced into the COMPUTESCORES function. The COMPUTESCORES function then selects the most similar

BP percent of components from *simList* and computes component scores based only on these similarities. This process is called *similarity filtering*. Conceptually, similarity filtering could be done by sorting *simList* by similarity score, but an asymptotically faster way is to find the $BP\%$ order statistic of *simList* and partition the list around it, both of which can be done in linear time [173]. The effect of similarity filtering on recommendation quality is evaluated in sections 8.2.1 and 8.3.1. When describing COMPUTESCORES for the recommenders, this common structure is omitted and only what COMPUTESCORES does for each composition Q_r that gets considered is defined.

Finally, in line 14, components are recommended based on the computed component scores and the partial composition using the RECOMMEND function.

As all four recommender algorithms defined in this chapter use the same RECOMMEND function, it is defined here instead of in subsequent sections. As mentioned at the start of this section, the recommender should suggest one or more components based on their computed score. The number of recommended components per query is denoted by R and keep constant throughout the algorithms' operation. The challenge of selecting the appropriate number of component to recommend per query is discussed in sections 6.6 and 8.4.5.

If a user has already used a particular component in the composition, recommending the same component seems less useful than recommending a different component that is possibly unknown to the user, even when multiple instances of the same component might be required to complete the composition. Therefore, before selecting the R components with the highest score, all components already present in the partial composition P are removed from contention. The remaining components are partitioned around the Rth largest score, and the top R components are sorted in descending order by score and returned. This can be done in $O(|\mathcal{T}| + R \log R)$ time [173], which is typically $O(|\mathcal{T}|)$ as R is a small constant.

The computational complexity of any recommender algorithm that follows this method will depend on the complexities of the four functions PREPROCESS, EVALUATESIMILARITY, COMPUTESCORES and RECOMMEND. However, it is clear that response time will be dominated by evaluating the similarity to every composition in the database and possibly computing component scores. Based on this observation, to make the recommender usable as a real-time assistant in application development, a key requirement for similarity evaluation and component scores computation is that both procedures should be efficient, and ideally only take time linear in the size of the compositions being compared. This issue is discussed further in section 8.2.3.

6.2 Cosine Similarity for Feature Vectors

The simplest algorithm defined in this chapter is based on the component vector model described in section 5.3. The similarity score for this algorithm is the cosine of the angle between component vectors, so the algorithm is called *ComponentVectorCos*.

As component vector coordinates represent component frequency, they are nonnegative. Therefore, all component vectors lie in the same orthant, i.e. the same multidimensional generalization of a quadrant, and the angle between them lies between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$ so its cosine is nonnegative. If this angle is close to 0, its cosine will be close to 1 which indicates a higher degree of similarity. If the angle is close to $\frac{\pi}{2}$ in absolute value, its cosine will be close to 0 which indicates low or no similarity.

In general, the cosine of the angle between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ can be computed using their dot product as

$$\cos \phi_{\boldsymbol{x},\boldsymbol{y}} = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}.$$

As mentioned in section 5.3, component vectors are represented with a sorted list of key-value pairs where each pair corresponds to a nonzero coordinate value. If there are l_P and l_Q nonzero coordinate values of the component vectors of P and Q, then the dot product in the numerator can be computed in $O(l_P + l_Q)$ time by traversing both lists in parallel using two pointers. Computing the norms in the denominator takes $\Theta(l_P)+\Theta(l_Q)$ time. Consequently, the similarity score can be computed in $\Theta(l_P + l_Q)$ time.

Note that computing the cosine of the angle between component vectors doesn't provide any additional similarity information beyond the computed similarity score. Thus, the EVALUATESIMILARITY function for the *ComponentVectorCos* algorithm only returns the similarity score.

As the goal of the recommender is to suggest components that might be useful for completing the application, the COMPUTESCORES function adds the computed similarity score to the score of every component that is more frequent in Q than in P, multiplied by that difference in frequency.

An analogous approach is applied to the structure vector representation in the *StructureVectorCos* algorithm. The only difference in similarity computation is that structure vectors have more nonzero coordinate values if there is at least one arc in the composition. On the other hand, note that COMPUTESCORES still only deals with components, and arc coordinates of the

structure vectors are simply ignored.

6.3 Component Sequence Edit Distance

The *ComponentSeqEditDistance* algorithm is based on computing an edit distance between component sequences of P and Q. The EVALUATESIMILARITY function for *ComponentSeqEditDistance* finds an optimal way to convert the component sequence of P into the component sequence of Q by matching identical components or adding and removing components from P, subject to nonnegative real costs C_{add} and C_{rem}. Specifically, if k components in the component sequence of P are matched to components in Q, then the edit distance D is defined as

$$D = (n_P - k)C_{rem} + (n_Q - k)C_{add}. \tag{6.1}$$

Equation 6.1 arises because if k components are matched, then all the remaining $n_P - k$ components in P must be removed and all the $n_Q - k$ components in Q that haven't been matched must be added to P to change the component sequence of P into the sequence of Q.

The similarity score of P and Q is then defined as

$$\text{similarity score} = \frac{1}{1 + D_{opt}}, \tag{6.2}$$

where D_{opt} is the minimal possible edit distance.

The similarity score defined by equation 6.2 is a real number between 0 and 1, with values near 0 denoting very low similarity and values near 1 denoting high similarity. The change of similarity score with the minimal edit distance D_{opt} is shown graphically in figure 6.1. When the component sequences of P and Q are identical, then $D_{opt} = 0$ and the similarity score is equal to 1. The similarity score decreases quickly for edit distances between 0 and 5, and very slowly afterwards.

Since the lengths of the component sequences of P and Q are equal to the number of vertices in their graph representation, they are denoted by n_P and n_Q. The edit distance can be computed using dynamic programming in $\Theta(n_P n_Q)$ time by filling in a matrix of edit distances for all pairs of component sequence prefixes in a systematic way.

The PREPROCESS function returns the component sequence in generalized topological order for database compositions. For the input partial composition P, two sequence orders are

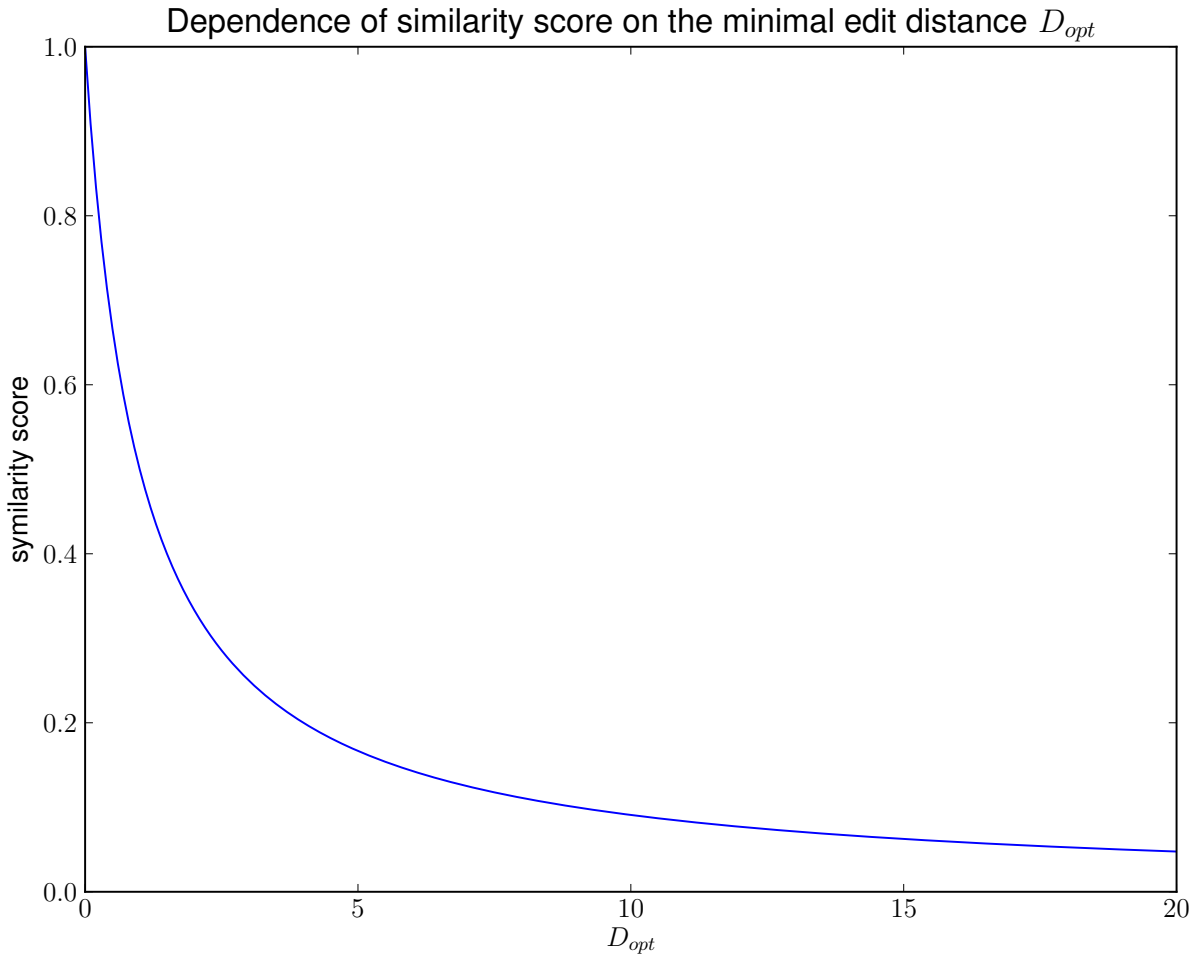


Figure 6.1: The change of similarity score with minimal edit distance for the COMPONENTSEQEDITDISTANCE algorithm.

possible. First, in a practical setting, if the user adds a new component to his partial composition, the recommender can reuse the edit distance matrix computed for the previous query when the input partial composition had one less component if the component sequence order matches component insertion order. Specifically, when computing the edit distance matrix for the input partial composition P, EVALUATESIMILARITY reuses the $n_P - 1$ rows of the edit distance matrix computed for the previous query and updates the last row in just $\Theta(n_Q)$ time. When users remove components from a partial composition, a possibly large part of the edit distance matrix must be recomputed. Additionally, this approach requires the recommender system to store all the computed edit distance matrices, perhaps for a limited time frame, which incurs a possibly significant memory overhead.

Second, the component sequence of the input partial composition can be ordered in generalized topological order as well. For this ordering, storing computed edit distance matrices provides no benefit and the EVALUATESIMILARITY function must compute the whole edit dis-

tance matrix for each query, which takes $\Theta(n_P n_Q)$ time. On the other hand, the generalized topological order provides a stronger foundation for the computed similarity score. Both possible orders are evaluated in this dissertation.

In both cases, EVALUATESIMILARITY returns the edit distance matrix along with the similarity score so that the optimal set of edit operations can be reconstructed. This reconstruction provides additional information about the roles of particular components in the computed edit distance, and this information is useful for computing component scores. In particular, at least two distinct strategies for computing component scores are possible. First, when updating component scores based on a particular database composition Q, the scores of those components that were added to P in the editing process could be increased by the computed similarity score between P and Q[1].

Second, the component sequence editing process also provides the algorithm with a set of matched components. This set of components is a subset of all components that appear both in P and Q, and some of these components might not be matched due to the sequential nature of the representation[2]. This set of matched components can be used to update component scores in a more focused way. For each component c in Q that was matched to a component in P, COMPUTESCORES increases the scores of components that are connected to c in the graph representation of composition Q via an incoming or outgoing arc by the computed similarity score. These components are good candidates to be added to the composition next as they are on the frontier between the similar and the dissimilar parts of the compositions.

With both approaches to computing component scores, it takes $\Theta(n_P + n_Q)$ time to reconstruct the optimal edit operations from the edit distance matrix and $O(m_Q)$ time to update component scores. In this dissertation, only the second approach is evaluated. Additionally, the score of the remaining components in Q is increased marginally by a small fraction of the similarity score so that they might get recommended when too few connected components are found or when most of them have already been used in P. An example of this approach to computing component scores is presented in the next section.

It is important to note that the *ComponentSeqEditDistance* algorithm assigns a nonzero similarity score to every composition in the composition database even if there are zero matching

[1]Note that this is, in fact, identical to increasing the scores of all components in Q and to the component scores computation process used in both *ComponentVectorCos* and *StructureVectorCos* because all the components in Q that are not already components of P must necessarily be added to P in the edit process. When recommending components, all components that were originally in P are removed from consideration, and therefore, changes to the scores of the components in Q that are also components in P are irrelevant.

[2]See section 6.7 for an example.

components which is in contrast to both feature vector algorithms and the graph based algorithm described in the next section. However, the described approach to component scores computation ensures that those compositions that contain no matches still don't contribute to the final component scores used for making recommendations, even in the absence of similarity filtering.

6.4 Probabilistic Graph Edit Distance

Matching of labeled graphs is identified as the best case for matching of general graphs in [168]. This is because if only vertices with equal labels can be matched and there are no repeated labels, the optimal matching can be found in time linear in the sum of the graph sizes. This idea is the starting point for defining a reasonably efficient similarity measure for the composition graph model.

The *GraphEditDistance* algorithm, which is the fourth structural algorithm defined in this dissertation, finds a set of edit operations that convert the vertex set of P into the vertex set of Q. Assume that the algorithm has found a matching $\mathcal{M}$ that maps vertices from P to vertices of Q, respecting labels, i.e. component identifiers. Then the vertex set of P can be converted into the vertex set of Q by removing all the $n_P - |\mathcal{M}|$ vertices of P that are not matched by $\mathcal{M}$ and adding to P all the $n_Q - |\mathcal{M}|$ vertices of Q that are not matched by $\mathcal{M}$. Note that this is similar to the *ComponentSeqEditDistance* algorithm except that the matching $\mathcal{M}$ is not restricted by any order over the vertices. While arcs are not deleted or added, the arc structure of the graphs is used to define a similarity measure for the sets of matched vertices in P and Q.

Let p_1 and p_2 be vertices of P, and q_1 and q_2 be the vertices of Q corresponding to p_1 and p_2 under the matching $\mathcal{M}$. Furthermore, let S be the number of quadruples p_1, p_2, q_1, q_2, such that there is a path from p_1 to p_2 in P if and only if there is a path from q_1 to q_2 in Q. Then the similarity score associated with the matching $\mathcal{M}$ is defined as

$$\text{similarity score} = \frac{|\mathcal{M}| + S \cdot C_{conn}}{1 + (n_P - |\mathcal{M}|)C_{rem} + (n_Q - |\mathcal{M}|)C_{add}}, \tag{6.3}$$

where C_{conn}, C_{rem} and C_{add} are nonnegative real numbers and are parameters of the algorithm. The numerator ensures that larger matches with similar connectivity patterns give larger similarity scores. The denominator is the edit distance increased by 1 to make the result defined when a perfect matching is possible, i.e. when all vertices can be matched.

In the presence of multiple vertices with the same label, many different matchings may be

possible. When the number of duplicates is small, it might be possible to enumerate and evaluate each possible matching. However, the number of possible matchings grows very quickly with the number of duplicates. For example, if there are s instances of a particular component in P and t instances of that same label in Q, the number of ways to match the s vertices of P is

$$(t)_s = t(t-1)\dots(t-s+1),$$

assuming $s \leq t$.

While the average number of a particular component in the *Yahoo Pipes* dataset used in this dissertation is around 1.57, about 20% of the pipes have a component repeated at least five times, and one pipe uses the *fetch* module 95 times. Therefore, and also in the interest of generality, enumerating all possible matchings is infeasible. This problem is addressed by limiting the number of matching choices the algorithm makes and introducing randomness so that all matchings have a chance to be explored. This approach is illustrated with pseudocode in Algorithm 6.2.

The EVALUATESIMILARITY function defines the recursive procedure EXPLOREMATCHING that is the core of this algorithm in lines 5–28. EXPLOREMATCHING recursively tries to match every vertex u from P to an unused vertex in Q that represents the same component. The list of candidate vertices can easily be precomputed in PREPROCESS and is denoted by $Qcands$ in the pseudocode. Before calling EXPLOREMATCHING in line 29, these match candidates are randomly shuffled in line 4, which is also the only place where randomness is introduced into the algorithm.

The number of invocations of EXPLOREMATCHING is limited in line 6 with a number L that is a parameter of the algorithm. If L is larger, the algorithm is potentially slower but more candidate matchings can be explored, possibly yielding better results. Lines 9–14 deal with the case when a matching $\mathcal{M}$ is finalized, i.e. all the vertices of P have been processed. The loop in lines 16–25 tries to match vertex u with the candidate vertices v from Q and recursively complete the matching. The set $Qused$ can be implemented with a bitset, and the matching $\mathcal{M}$ with a simple array allowing updates in lines 18–19 and 22–23 in constant time. In line 20, the new value of S is computed by traversing the matching and comparing connectivity of u in P to that of v in Q, as described earlier. Finally, the possibility of not matching u is explored in line 26. This might be optimal when the number of matching candidates is less than the number of u's duplicates in P, and is necessary when there are no matching candidates at all.

```
1: function EVALUATESIMILARITY(P_r, Q_r)
2:   bestScore ← 0.0
3:   bestMatching ← {}
4:   randomly shuffle every list in Qcands   ▷ Qcands maps components to a list of vertices
     that represent that component in Q, and is a part of Q_r.

5:   procedure EXPLOREMATCHING(u, Qused, M, S)
6:     if this procedure has been called more than L times then
7:       return
8:     end if
9:     if u > n_P then                                   ▷ Evaluate the matching.
10:      candScore ← the similarity score computed based on M and S
11:      if candScore > bestScore then
12:        bestScore ← candScore                        ▷ Update with better matching.
13:        bestMatching ← M
14:      end if
15:    else
16:      for all v in Qcands[u] do                      ▷ Try to match u.
17:        if v ∉ Qused then
18:          Qused.add(v)                               ▷ Update Qused and M.
19:          M.add(u, v)
20:          newS ← updated S value
21:          EXPLOREMATCHING(u + 1, Qused, M, newS)     ▷ Recurse.
22:          Qused.remove(v)                            ▷ Undo the changes.
23:          M.remove(u, v)
24:        end if
25:      end for
26:      EXPLOREMATHCHING(u + 1, Qused, M, S)           ▷ Don't match u.
27:    end if
28:  end procedure

29:  EXPLOREMATCHING(1, {}, {}, 0.0)   ▷ Start from the first vertex of P with no vertices
     from Q used and the empty matching.
30:  return (bestScore, bestMatching)
31: end function
```

Algorithm 6.2: EVALUATESIMILARITY function for the *GraphEditDistance* algorithm: Returns the largest found similarity score and the matching that produced it.

When all matchings have been explored or the invocation limit has been reached, EVALUATESIMILARITY returns the largest found similarity score and the matching that achieved that score in line 30.

It is nontrivial to describe the *a priori* computational complexity of this algorithm in that it largely depends on component frequencies and component duplication in P and Q as well as the size of the maximal matching. Note that L bounds the number of invocations of EXPLOREMATCHING, and not the total number of operations. This allows more operations to be

performed when there are many matching possibilities. For intuition about the computational complexity, assume that the average number of candidates in the loop in line 16 is $\Theta(\frac{n_Q}{n_P})$, i.e. that the candidates are distributed uniformly over all vertices of P. Then EVALUATESIMILARITY takes $O(L(1+\frac{n_Q}{n_P})n_P) = O(L(n_P+n_Q))$ time. The rightmost n_P factor comes from updating the value of S in line 20. This is based on revisiting all the previously matched vertices from P and comparing the connectivity of this pair to their matches in Q, as described earlier. If the connectivity of every pair of vertices can be checked in constant time, then S can be updated in time linear in the size of the matching which is loosely bounded by $O(\min\{n_P, n_Q\})$. Therefore, to allow efficient connectivity checking, the PREPROCESS function of *GraphEditDistance* compute the transitive closure of the input graph and include that in the returned representation.

Unlike vector angle cosine used in the feature vector algorithms *ComponentVectorCos* and *StructureVectorCos* and similar to the edit distance computed in the *ComponentSeqEditDistance* algorithm, a matching between vertices of the composition graphs provides structural information about how or why exactly two compositions are similar. The identical process for computing component scores as in the *ComponentSeqEditDistance* algorithm is used. Specifically, the computed matching is used in COMPUTESCORES to significantly increase only the scores of those components in Q whose corresponding vertex is directly connected to a matched vertex, either by an incoming or outgoing arc. The score of the remaining components in Q is only increased by a small fraction of the similarity score so that these components get considered for recommendation if too few connected components are found in the whole composition database or when most of the candidate components are already in P and, therefore, aren't useful recommendations.

A simple example that illustrates this process is shown in figure 6.2. Of the vertices in P, only A and D appear in Q. The matching computed by EVALUATESIMILARITY is indicated with matching fill colors. There are two instances of D in Q, and the left one is chosen in $\mathcal{M}$. Note that in this case both choices of D lead to the same similarity score with $S = 0$ as neither D is reachable from the vertex A in Q as it is in P. Arcs that influence how component scores are updated based on this P-Q pair are marked with asterisk and tilde symbols. Assume that EVALUATESIMILARITY computed a similarity score s for this matching. Because D was matched, the scores of components X, D, A and Y would be increased by s because they label vertices that are directly connected to the matched D. The connecting arcs are marked with asterisk symbols. Similarly, because A was matched, the scores of D and Y would be further

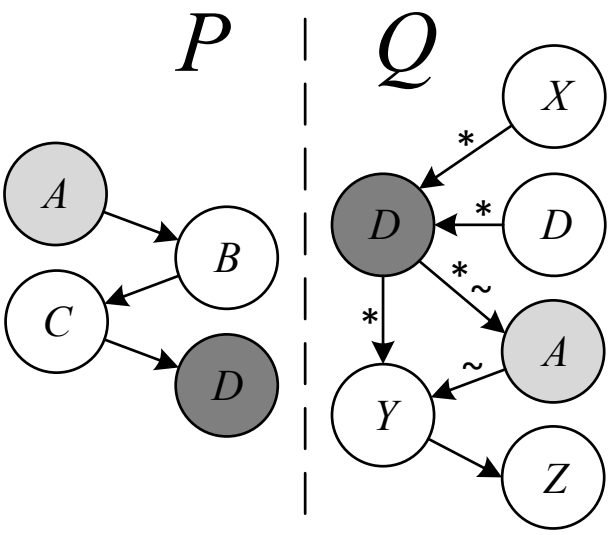


Figure 6.2: A simple P-Q pair illustrating COMPUTESCORES for the *GraphEditDistance* algorithm. The computed matching is indicated by matching fill colors of A and D vertices. The arcs that influence which component scores are increased are marked with asterisk and tilde symbols.

increased by s based on arcs labeled with a tilde. Finally, the score of component Z would be increased by $\varepsilon \cdot s$, where ε is a small real constant much smaller than the inverse of the number of compositions in the composition database. Because of this marginal score increase, the algorithm would prefer component Z to components that were never used in any similar compositions, but would only recommend Z if all connected components have already been used in P. In all evaluations in subsequent chapters, ε was set to 10^{-8}.

Note that the RECOMMEND function eliminates both A and D from recommendation candidates because they are already used in the partial composition P, but, conceptually, their scores are increased in COMPUTESCORES.

6.5 Summary of the Defined Structural Recommender Algorithms

Four structural recommender algorithms based on the general component recommendation method described in section 6.1 have been defined in this chapter. An overview of the basic properties of these algorithms is presented in table I.

The simplest considered algorithm is *ComponentVectorCos* defined in section 6.2. *ComponentVectorCos* uses a feature vector as its base representation for compositions, where features are the frequencies of components used in the composition. This vector is called the *component*

Table I: An overview of the four recommender algorithms.

	ComponentVectorCos	*StructureVectorCos*	*ComponentSeqEditDistance*	*GraphEditDistance*
Base Representation	component vector	structure vector	component sequence	vertex-labeled digraph
Parameters	BP, R	BP, R	BP, R, C_{add}, C_{rem}	BP, R, L, C_{add}, C_{rem}, C_{conn}
Similarity Measure	[a] $\cos\phi$	[a] $\cos\phi$	[b] $\frac{1}{1+D_{opt}}$	Equation 6.3
Affected Component Scores	comps more frequent in Q	comps more frequent in Q	comps connected to matched comps in Q	comps connected to matched comps in Q
Complexity [c]	[d] $\Theta(l_P + l_Q)$	[d] $\Theta(l_P + l_Q)$	[e] $\Theta(n_P n_Q)$	[f] $O(L(n_P + n_Q))$

[a] ϕ is the angle between the vectors
[b] D_{opt} is the minimal edit distance between component sequences of P and Q
[c] per database composition; includes only the dominating time complexity of EVALUATESIMILARITY and COMPUTESCORES
[d] l is the number of nonzero coordinate values of the vector
[e] can be improved to $\Theta(n_P + n_Q)$ when the component sequence of P is ordered in component insertion order instead of the generalized topological order and a new component has just been added
[f] assuming matching candidates are evenly distributed as explained in the text

vector of the composition. The only parameters of *ComponentVectorCos* are BP, which regulates similarity filtering in component scores computation, and R, which determines how many components are recommended for each query. As is common for feature vector algorithms, *ComponentVectorCos* uses the cosine between the component vectors of compositions to measure composition similarity. When a database composition Q is considered during component scores computation, the scores of those components in Q that are more frequent in Q than in the input partial composition P are increased by that difference in frequency multiplied by the computed similarity score between P and Q. Evaluating similarity and updating component scores for each database composition Q takes time linear in the length of the component vectors, i.e. in the number of nonzero vector coordinates which is bounded above by the number of components in P and Q.

The *StructureVectorCos* algorithm, also defined in section 6.2, introduces arc information into the feature vector which is now called *structure vector*. Specifically, in addition to component frequencies, structure vectors contain arc frequencies as well. In all other respects, *StructureVectorCos* and *ComponentVectorCos* are identical. The effect of component connectivity information on the quality of a recommender in this feature vector model is analyzed based on the comparison of these two algorithms.

The *ComponentSeqEditDistance* algorithm defined in section 6.3 is based on a sequential

representation of compositions. Specifically, the components of a composition are ordered into a list that is called the *component sequence* of the composition. Two orders of the component sequence are considered in this dissertation—a generalized topological order that preserves additional structural information about the composition and component insertion order that allows the algorithm to be significantly faster by reusing computation between queries. In the *ComponentSeqEditDistance* algorithm, composition similarity is based on the minimal edit distance D_{opt} between the component sequences of compositions. The allowed edit distance operations are adding a component to the component sequence of the input partial composition P, removing a component from the component sequence of P and matching a component in P to a component in Q. When all the edit operations are performed, the component sequences of P and Q must be identical. Those components in P that are neither added nor removed during the process are considered *matched*. In addition to the BP and R parameters, *ComponentSeqEditDistance* uses two edit distance parameters C_{add}, which specifies the cost of adding a component to P, and C_{rem}, which specifies the cost of removing a component from P.

While one additional strategy to computing component scores with *ComponentSeqEditDistance* is outlined in section 6.3, in this dissertation, *ComponentSeqEditDistance* significantly increases the component scores of only those components in Q that are connected by an incoming or outgoing arc to a component that has been matched in the optimal edit process.

When components of the input partial composition P are ordered in the component sequence in the generalized topological order, the whole edit distance matrix must be recomputed by the recommender for each query. The time complexity of solving this problem with dynamic programming is $\Theta(n_P n_Q)$. If components of P are ordered in insertion order, then parts of the edit distance matrix can be persisted in the recommender system and used for later queries with subsequent versions of P which significantly reduces time complexity, but increases memory requirements.

Finally, the algorithm *GraphEditDistance* defined in section 6.4 uses the vertex-labeled directed graph model defined in section 5.1 as the base representation of compositions. To evaluate similarity between two compositions P and Q, *GraphEditDistance* searches for a matching $\mathcal{M}$ that maps components from P onto the same components in Q, i.e. matches vertices of the graph representations of P and Q respecting vertex labels. Given a matching $\mathcal{M}$, the similarity score of P and Q is defined in a similar way as in the *ComponentSeqEditDistance* algorithm. In particular, an edit distance between the vertex set of P and the vertex set of Q is computed for a

Table II: A summary of algorithm parameters.

	Description
BP	Stands for *best percent*. Regulates similarity filtering in component scores computation—only the most similar BP percent database compositions are used for score computation.
R	The number of components that are recommended per query.
C_{add}	The edit distance cost incurred for adding a component to P.
C_{rem}	The edit distance cost incurred for removing a component to Q.
C_{conn}	The factor that increases the similarity score of a P-Q pair when matched components are similarly connected.
L	Limits the number of matchings considered in *GraphEditDistance*.

particular matching $\mathcal{M}$, along with a similarity factor that is based on the size of the matching $\mathcal{M}$ and the connectivity similarity between the matched components in P and Q. The similarity score is then computed based on the quotient of that similarity factor and the edit distance.

In the presence of duplicate components in P and Q, many different matchings are possible. As enumerating all these possibilities in infeasible, randomness is introduced so that the *GraphEditDistance* algorithm randomly explores this matching space searching for a matching with a large similarity score. The search process is limited by the parameter L which can be used to regulate the tradeoff between faster execution and exploring more possible matchings.

In addition to the parameters BP, R and L, *GraphEditDistance* is parameterized with edit distance costs C_{add} and C_{rem} which serve the same purpose as in the *ComponentSeqEditDistance* algorithm. Finally, the parameter C_{conn} is a measure of significance of similar connectivity between the matched components in P and Q when the algorithm evaluates composition similarity.

6.6 Choosing Algorithm Parameter Values

All the parameters used in any of the four structural algorithms defined in this chapter are summarized in table II. All four algorithms share two parameters—the parameter BP which regulates the degree of similarity filtering in component scores computation and the parameter R which specifies how many components are recommended for each query to the recommender.

As explained in section 6.1, with a higher level of similarity filtering, the recommender bases its recommendations only on compositions from the composition database that are the

most similar to the input partial composition. There are two possible benefits of this procedure. First, in the absence of similarity filtering in component scores computation, components that are very frequent in the composition database can become strong candidates for recommendation even when they are never used in composition that are very similar to the input partial composition and thus are probably not useful recommendations. This problem is most likely to occur if a component recommender algorithm doesn't significantly discriminate between very similar and slightly similar compositions in terms of similarity score. The cumulative effect of the many less similar compositions that contain some frequent component can make the component score of that frequent component higher than the component score of rare components that only appear in a few very similar compositions, but might actually be the best recommendations. Second and less important, while the dominating factor in recommender response time is similarity evaluation, significantly reducing the number of compositions that are used for computing component scores can potentially decrease recommender response times.

On the other hand, aggressive similarity filtering can cause sparsity problems when there aren't any very similar compositions in the composition database to begin with, and the cumulative effect of many compositions is exactly what is being sought.

Choosing the number of recommended components per query R is largely a UI design issue. Recommending more components might expose the user to new ideas, but also reduces the amount of information that can be displayed for each recommendation if UI space is restricted.

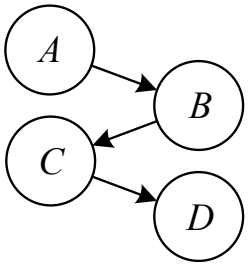
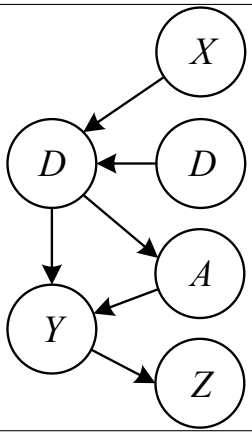
The effects of both parameters BP and R on recommender quality are evaluated in chapter 8.

The edit distance parameters C_{add} and C_{rem} in both *ComponentSeqEditDistance* and *GraphEditDistance* and C_{conn} in *GraphEditDistance* are candidates for model selection techniques. While choosing these parameters manually can provide a way to model the difficulty or frequency of adding and removing components to and from the partial composition, anticipating the true effect of a particular choice is challenging. Therefore, the choice of the values for the edit distance parameters should be based on extensive evaluation on a particular dataset.

For both datasets used for recommender evaluation in this dissertation, the effect of changing these parameters on recommender accuracy was minimal. This issue is discussed further in section 7.6.

Finally, the L parameter of *GraphEditDistance* can be used to control the tradeoff between its execution time and possibly discovering better matches. Note that L also bounds the max-

Table III: Composition base representations used in the four recommender algorithms.

	P	Q
GraphEditDistance		
ComponentSeqEditDistance	$< A, B, C, D >$	$< X, D, D, A, Y, Z >$
StructureVectorCos	$[A, B, C, D,$ $A \to B,$ $B \to C, C \to D]$	$[A, D : 2, X, Y, Z,$ $A \to Y, D \to A, D \to D,$ $D \to Y, X \to D, Y \to Z]$
ComponentVectorCos	$[A, B, C, D]$	$[A, D : 2, X, Y, Z]$

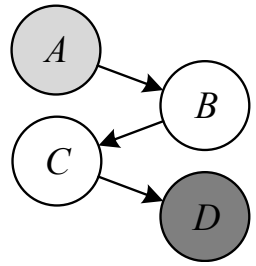
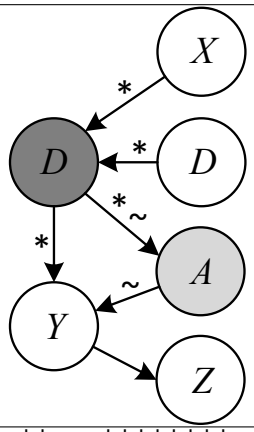
imum size of a composition for which meaningful recommendations can be given as compositions with more than L components won't have a single matching explored.

6.7 An Example of Algorithm Operation

In this section, the key steps of similarity evaluation and computing component scores of all four structural recommender algorithms defined in this chapter are described on the example P-Q pair from figure 6.2.

The component sequence, component vector and structure vector representations shown in rows 2–4 of table III are generated from the graph representations in the top row. The composition P is a simple chain of components, while Q is considerably more complex, but still acyclic. Intuitively, these two compositions are not very similar—they do share the components A and D, but these components seem to perform quite different roles in their respective composition and there are several additional components in each composition that are not present in the other composition. In P, component A is the starting point of the composition in some sense—it is either the source of data or control flow, or something similar. On the other hand, in Q, component A appears to be more central to the composition. The opposite is true for component D—it is the last component in the chain of P, and topologically near the beginning

Table IV: Similarity evaluation for the P-Q pair in the four recommender algorithms. The similarity scores are computed assuming $C_{add} = C_{rem} = 1$.

	P	Q	Computed	Similarity Score
GraphEditDistance			$\lvert\mathcal{M}\rvert = 2, S = 0$	$\frac{2+0}{1+(4-2)+(6-2)} = \frac{2}{7}$
ComponentSeqED	<~~A, B, C~~, D>	<~~++~~ X, D, ~~+++++++++~~ D, A, Y, Z>	$D_{opt} = 3 + 5$	$\frac{1}{1+8} = \frac{1}{9}$
StructureVectorCos	$[A, B, C, D,$ $A \to B,$ $B \to C, C \to D]$	$[A, D : 2, X, Y, Z,$ $A \to Y, D \to A, D \to D,$ $D \to Y, X \to D, Y \to Z]$	$\cos\phi = \frac{1*1+1*2}{\sqrt{7}\sqrt{14}} \approx 0.30$	
ComponentVectorCos	$[A, B, C, D]$	$[A, D : 2, X, Y, Z]$	$\cos\phi = \frac{1*1+1*2}{\sqrt{4}\sqrt{8}} \approx 0.53$	

of Q. Note that because of this reversal of roles of components A and D, their occurrences in the topologically ordered component sequences used by the *ComponentSeqEditDistance* algorithm are interleaved, i.e. A comes before D in the component sequence of P, but after D in the component sequence of Q.

The four structural recommender algorithms evaluate the similarity between this example P-Q pair as illustrated in table IV. *GraphEditDistance* discovers a matching $\mathcal{M}$ between A and D components as indicated by matching gray fills of the components. The size of this matching is 2, and $S = 0$ because there is a directed path from A to D in composition P, while there is no such path in Q. To compute numeric similarity scores for this example, both edit distance costs C_{add} and C_{rem} for both *GraphEditDistance* and *ComponentSeqEditDistance* are set to 1. Based on equation 6.3, *GraphEditDistance* evaluates the similarity of this P-Q pair with a similarity score of $2/7$.

Because of the inversion of roles of components A and D in compositions P and Q discussed at the start of this section, only one of these components can be matched when computing an optimal edit distance between the component sequences of P and Q in the *ComponentSeqEditDistance* algorithm. In fact, matching either A or D in this example leads to the same edit distance. The edit process shown in the table is preferred if the algorithm chooses to add

Table V: Updating component scores based on the computed similarities in the four recommender algorithms. The computed similarity scores are denoted with the symbol s.

	A	D	X	Y	Z
GraphEditDistance	$s = 2/7$	$2s = 4/7$	$s = 2/7$	$2s = 4/7$	$\varepsilon \cdot s = 2\varepsilon/7$
ComponentSeqEditDistance	$\varepsilon \cdot s = \varepsilon/9$	$s = 1/9$	$\varepsilon \cdot s = \varepsilon/9$	$\varepsilon \cdot s = \varepsilon/9$	$\varepsilon \cdot s = \varepsilon/9$
StructureVectorCos	–	$s \approx 0.30$	$s \approx 0.30$	$s \approx 0.30$	$s \approx 0.30$
ComponentVectorCos	–	$s \approx 0.53$	$s \approx 0.53$	$s \approx 0.53$	$s \approx 0.53$

a component to P instead of removing a component from P whenever both choices lead to the same edit distance. The matched D components are underlined in the component sequences. The remaining components of P are removed in the edit operation, which is illustrated with a strikethrough. On the other hand, the additional components in Q must all be added to P to make the component sequences equal, which is denoted by plus symbols above these components in the table. As three components need to be removed and five components need to be added, *ComponentSeqEditDistance* computes the edit distance of $D_{opt} = 8$ and a similarity score of $1/9$ by equation 6.2.

For both feature vector representations in the algorithms *StructureVectorCos* and *ComponentVectorCos*, the vector dot product in the numerator of the cosine equals 3. This is because in all coordinates of the vector spaces except those associated with the frequencies of components A and D, at least one of the vectors has the value 0. In the coordinate corresponding to component A, both vectors have the value 1 and in the coordinate corresponding to component D, the vector of composition P has the value 1, while the vector of composition Q has the value 2. These coordinates are underlined in the table. However, the computed similarity scores significantly differ in the denominator of the cosine where the lengths of these vectors are multiplied. As structure vectors are significantly longer than component vectors, the similarity score computed by the *ComponentVectorCos* algorithm is almost twice as large as the similarity score computed by the *StructureVectorCos* algorithm.

The effect of this P-Q pair on component scores is shown in table V. This example has already been explained for the *GraphEditDistance* algorithm in section 6.4—based on the arcs marked with asterisk and tilde symbols, the scores of components X and A would be increased by $2/7$, and the scores of components D and Y by $4/7$. Finally, the score of component Z would be increased by $2\varepsilon/7$, where ε is a very small constant, e.g. 10^{-8}.

In the *ComponentSeqEditDistance* algorithm, the same approach to computing component

scores is employed. The only component that is matched in the optimal edit process corresponds to the unshaded component D in the graph in table IV. Therefore, because of the $D \rightarrow D$ arc, the score of component D is increased by the similarity score. The scores of the remaining components of Q are marginally increased by $\varepsilon/9$.

In the feature vector algorithms, the scores of all components that are more frequent in Q than in P are increased by that difference in frequency multiplied with the similarity score. For the example discussed in this section, all the components in Q except for component A would get their score increased by the computed similarity score. Note that because components that are already part of the input partial composition P are removed from contention when the top R components are picked to be recommended so the changes in score to components A and D are inconsequential.

Chapter 7

Evaluation Methodology

The approach to component recommendation based on structural similarity of compositions is evaluated through the four structural component recommender algorithms defined in chapter 6 using two distinct evaluation scenarios—evaluation based on composition *snapshots* which are recommender query examples, presented in chapter 8, and evaluation based on simulated composition development where a composition is created incrementally, one component at a time, and the recommender is queried after each step, presented in chapter 9. Both evaluation scenarios are based on the idea that previously finished compositions can be used to create queries for the recommender for which additional information is available about which components would be useful recommendations for that query.

This chapter defines the general evaluation methodology used in both scenarios. First, the two datasets used for evaluation—a dataset of *Yahoo Pipes* compositions, and a synthetic dataset of more complex compositions—are described in section 7.1. Second, measures of recommender quality used in the evaluations are defined in section 7.3. Three simple statistical component recommender algorithms that are used mainly to characterize the datasets and as baselines when evaluating the structural algorithms are defined in section 7.4. Then, section 7.5 explains the methodology used for evaluating probabilistic recommender algorithms. Finally, the chapter is concluded with a discussion of algorithm parameter values that were chosen for evaluation in section 7.6.

7.1 Evaluation Datasets

The presented recommender algorithms are evaluated on two distinct datasets, i.e. two different collections of composite applications. The first dataset is a set of 7600 most popular *Yahoo Pipes* that were extracted from the Pipes website. The second dataset is synthetic and models more complex compositions with some of the properties of composite applications that can be created with *Geppeto*.

In the remainder of this section, these two datasets are analyzed in terms of the number of distinct components used in the compositions, the number of components and connections in compositions, neighbor diversity of components, component repetition in compositions, and structural complexity. The analyses are summarized and the dataset properties are compared from the aspect of component recommendation in section 7.2.

7.1.1 The *Yahoo Pipes* Dataset

In *Yahoo Pipes* terminology, components are called *modules* and connections between modules are called *wires*. Modules are used to construct feed URLs, fetch feed data, process it, merge it with other feed data, and present it as a new feed. When developing a pipe, there are a total of 55 different modules to choose from. The histogram of module frequencies in the dataset of 7600 pipes is shown in figure 7.1.

Three outliers have been remove to make the histogram more readable. First, the most frequent module is the *fetch* module that appears 12685 times in the 7600 pipes in the dataset. This module is used to fetch feed data and is therefore used in every pipe at least once. The second most frequently used module is the *output* module which is used exactly once in every pipe and represents the output feed of the pipe. Finally, the third most used module is *urlbuilder* which enables construction of parametrized URLs from parts, some of which can optionally be entered by the pipe's user. *Urlbuilder* occurs 5889 times in the dataset. These outlier modules have a significant impact on component recommendation on this dataset as they are ubiquitous in *Yahoo Pipes*.

Out of the remaining 52 modules, 10 modules are used between 1800 and 4000 times, and the other 42 modules are used at most 900 times with 22 modules being used at most 200 times. The 13 most frequent modules seem to be essential in Pipes development while the remaining 42 modules are only occasionally useful.

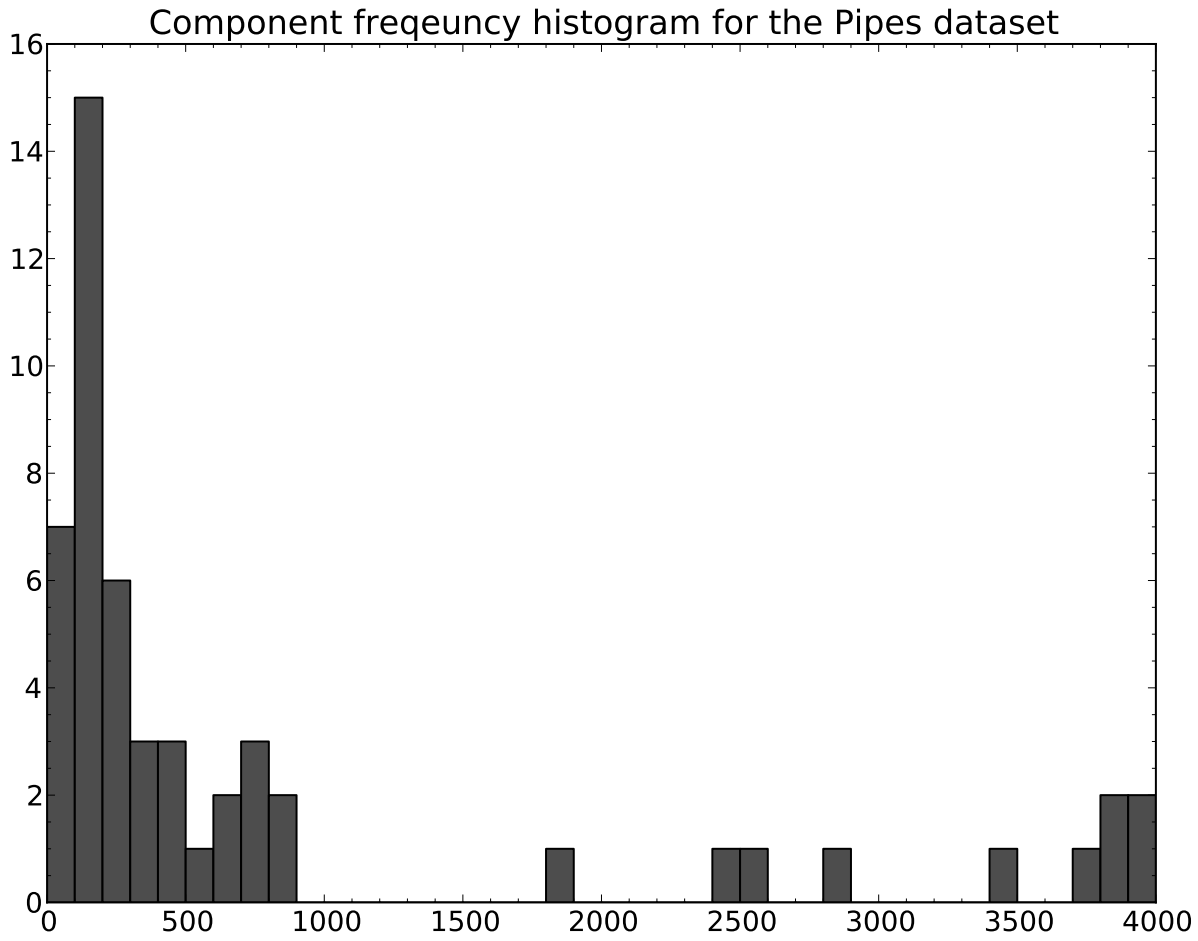


Figure 7.1: The histogram of component frequencies in the *Yahoo Pipes* dataset with a bin size of 100. Three outliers have been removed to make the histogram more readable: the *fetch* module with a frequency of 12685, the *output* module with a frequency of 7600 and the *urlbuilder* module with a frequency of 5889.

The histogram of the number of modules used in a pipe in the Pipes dataset is shown in figure 7.2. The average number of modules per pipe is around 9.3, the median is 7, and the maximum is 177. As is obvious from the histogram, the distribution of composition size for *Yahoo Pipes* is very skewed towards smaller compositions, and pipes with more than 20 modules are rare. This is because most pipes perform simple feed transformations, e.g. feed filtering or geotagging, that can be achieved with only several modules.

The distribution of the number of wires per pipe is shown in figure 7.3. The average number of wires per pipe is around 8.9, the median is 6 wires, and the maximum is 266. Almost all pipes in the dataset have fewer than 20 wires which is to be expected due to the distribution of module counts per pipe discussed above.

The distribution of *neighbor diversity* in the Pipes dataset is shown in figure 7.4. The neighbor diversity of a component is defined as the number of distinct components it is connected

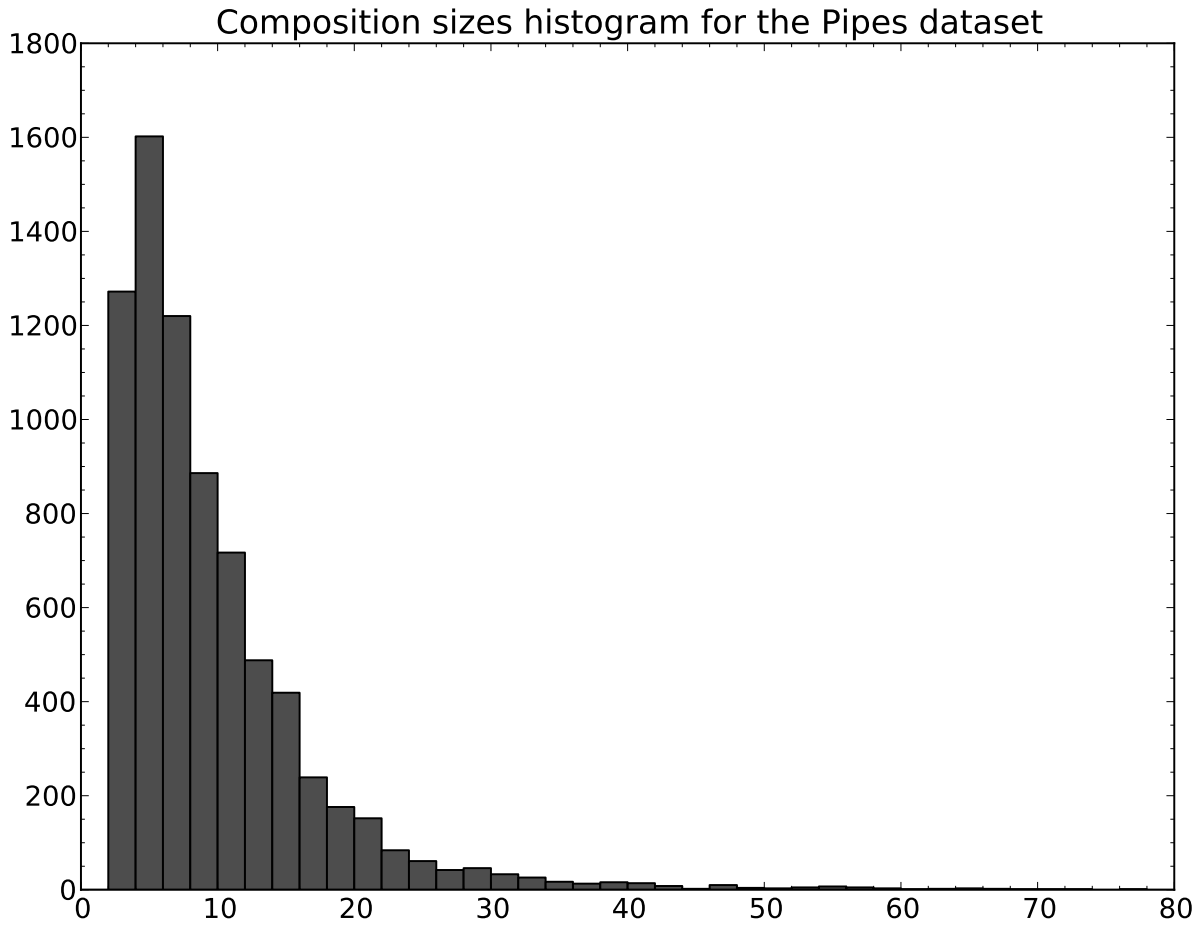


Figure 7.2: The histogram of the number of modules used in a pipe in the *Yahoo Pipes* dataset with a bin size of 2. There are several pipes in the dataset above size 80 up to a maximum of 177 used modules.

to via an arc at least once in the whole dataset. Two significant groups of modules in terms of neighbor diversity can be identified in the histogram—one with modules with neighbor diversity between 8 and 14 and one with modules having a neighbor diversity in the 18–22 range.

In general, components with large neighbor diversity are less useful for component recommendation as they provide less specific information about which components could be useful to finish the composition.

Several modules frequently appear multiple times in a single pipe, most of all *fetch* and *urlbuilder*. As noted in section 6.4, a module is repeated in a pipe an average of 1.57 times, and the *fetch* module appears 95 times in a single pipe. Component repetition is significant for the *GraphEditDistance* algorithm as repeated matching components increase the number of possible vertex matchings.

Yahoo Pipes are structurally very simple compositions. When represented with the graph model from section 5.1, all pipes are acyclic because the graph model represents the data flow

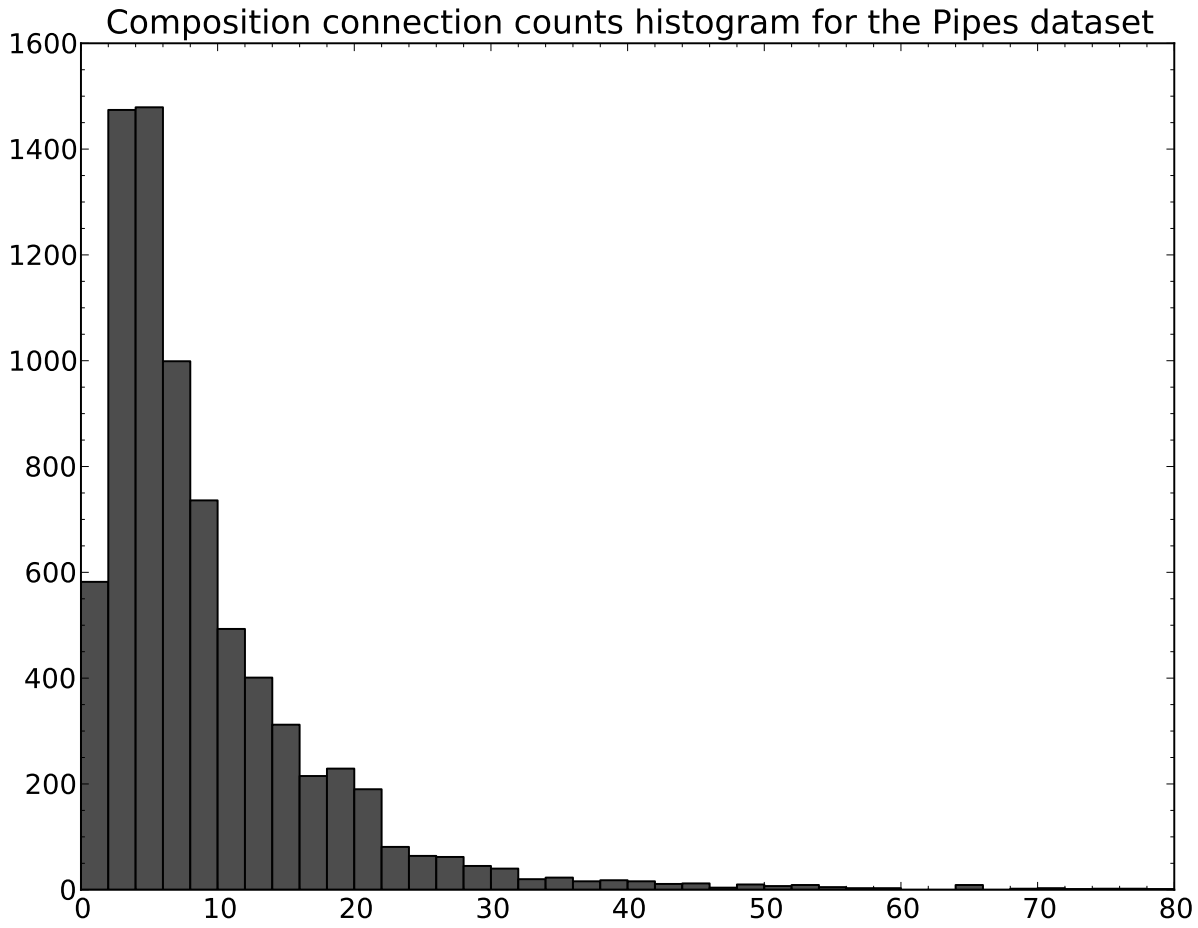


Figure 7.3: The histogram of the number of wires per pipe with bin size 2. Similar to the module counts histogram in figure 7.2, there are several pipes in the dataset with more than 100 wires which are not shown in the histogram. The actual maximum number of wires in a pipe is 266.

of the pipe. Furthermore, out of the 7600 pipes in the dataset, 6302 are directed trees[1].

A recent study [131] analyzed the *Yahoo Pipes* dataset and community and discovered several properties of interest for the research presented in this dissertation. Slightly over 60% of all the analyzed pipes had the same bag of modules as another pipe in the dataset. In fact, under a somewhat loosely defined structural model that is on a similar level of abstraction as the graph model from section 5.1, the study found that around 81% of pipes have a minimal distance of only 2 to another pipe, where the distance metric is the number of modules or wires that have to be changed. The study authors suggest that this is because pipes can be cloned and the duplicate can be used as a starting point for a new pipe. One of the conclusions of the survey is that pipe authors would benefit from better development support which also might reduce the amount of duplication of functionality.

[1] A directed tree is a directed acyclic graph (DAG) that remains acyclic even when arcs are replaced with undirected edges, i.e. becomes an undirected tree when arcs are replaced with edges.

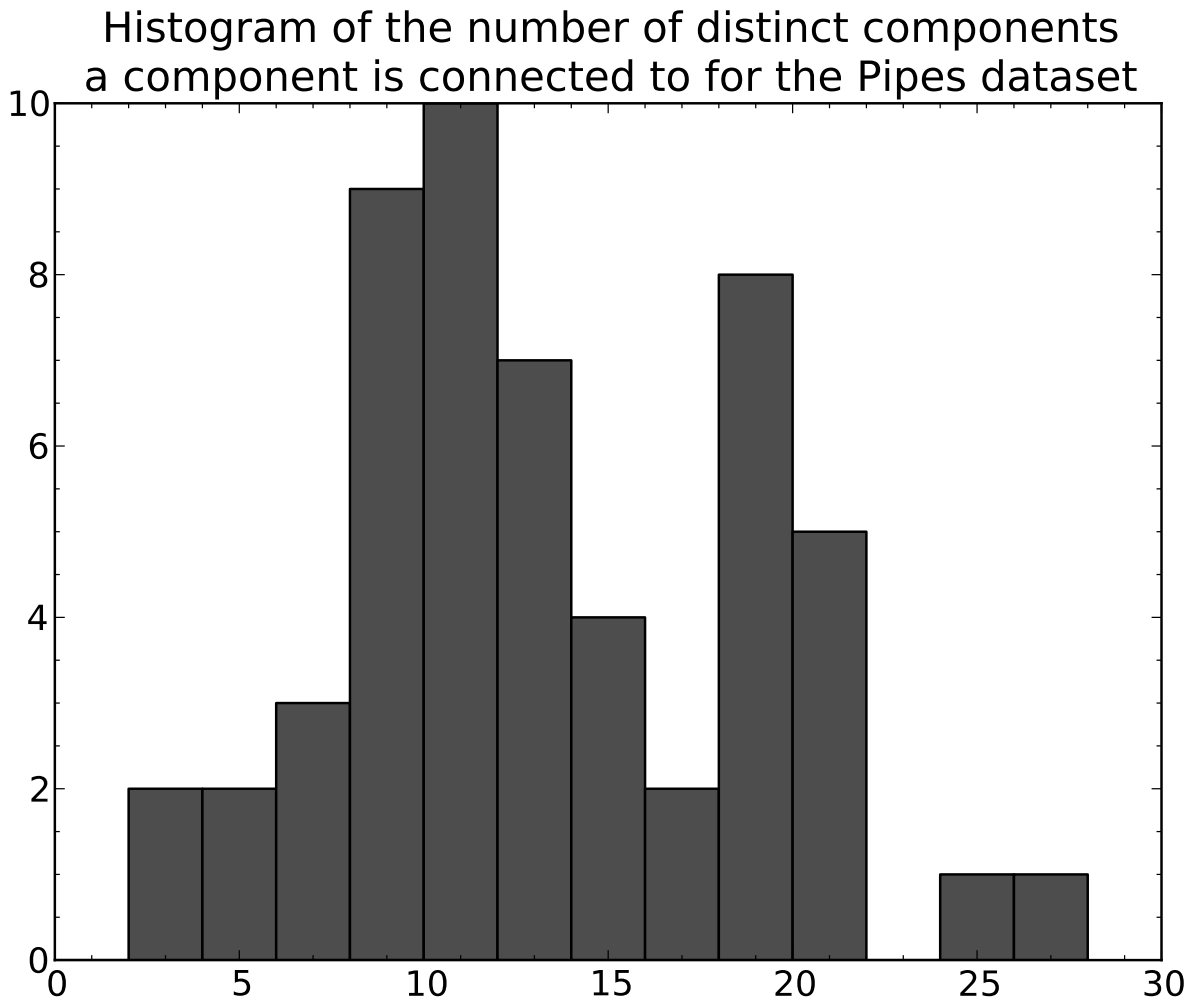


Figure 7.4: The histogram of the number of distinct components a component is connected to at least once in the whole datasets with a bin size of 2.

7.1.2 The Synthetic Dataset

The primary motivation for evaluating the presented algorithms on a synthetic dataset is to explore how the algorithms perform for structurally more complex and diverse compositions than *Yahoo Pipes*. Size, arc density and structural complexity of the generated compositions are based on experiences with *Geppeto*.

To match the Pipes dataset, 7600 compositions were generated, represented in the graph model from section 5.1. The components used in the generated compositions were drawn from a set of 1000 available components. Note that this is more than an order of magnitude more components than there are available in *Yahoo Pipes*. To model the fact that some components are used more often than others, the *base relative frequency* of the kth component was defined as as 0.99^k. For intuition, note that this means that the tenth most frequent component is used about 9% less often than the most frequent component, and the hundredth most frequent component

is used about 73% less often.

The challenging aspect of generating a dataset for this problem is that the premise of a component recommender is that there is composition knowledge stored in the previously created compositions. This composition knowledge is then used towards making recommendations when faced with new queries. To model this property, *component affinity* $\mathcal{A}_{C,D}$ was defined as a measure of how often components C and D appear together in the same composition and also a measure of how likely there is a connection from C to D. Component affinity is not defined to be symmetric. This is because, for example, a particular route planning widget might often feed data to a map widget, but the map widget might be much more popular and rarely connected to that particular route planning widget. Furthermore, connections are directed and that map widget might not even send any data back to the route planning widget.

For every component C of the 1000 available components, a number α between 0.01 and 0.99 was picked uniformly at random. This number is a measure of how many components C has a high affinity to and controls neighbor diversity of the component. The kth component in a random order specific to C was then assigned the affinity α^k. Note that affinity is relative. In other words, a high α characterizes components that can be connected to many different components, i.e. have high neighbor diversity, because α^k decreases fairly slowly. On the other hand, a low α characterizes components that are typically connected to only a small number of other components as α^k decreases very quickly.

The size of every graph was drawn from a Gaussian with $\mu = 9$ and $\sigma = 3.5$, truncated on the left at 2. The first component was chosen according to component base relative frequencies. Every subsequent component D was chosen based on the unnormalized probability obtained by multiplying its base relative frequency with $\mathcal{A}_{C,D}$ for some component C already in the composition, chosen uniformly at random.

When all n components were chosen, each component was assigned an outdegree o drawn from a Gaussian distribution of integers in the interval $[0, n)$, with $\mu = 1$ and $\sigma = 2$. Then, out of all the $n - 1$ potential outgoing arcs, o were chosen using component affinities as relative frequencies.

Finally, if the composition graph wasn't weakly connected[2], arcs with the highest affinity were added until the graph became weakly connected, in a manner analogous to Kruskal's minimum spanning tree algorithm [174].

[2]A graph is *weakly connected* if its undirected analog is connected. In other words, if arcs are replaced by undirected edges, there is a path between every pair of vertices.

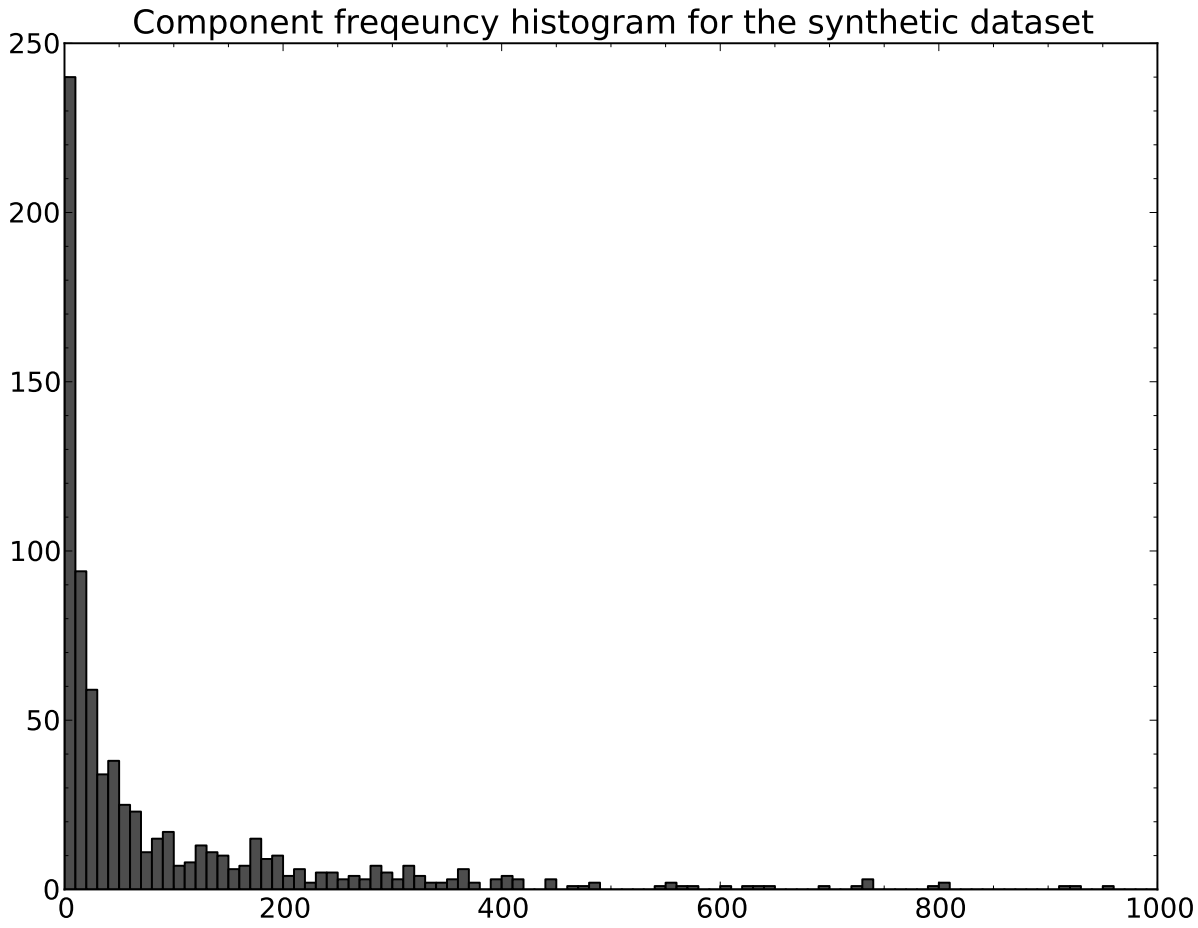


Figure 7.5: The histogram of component frequencies in the synthetic dataset with a bin size of 10.

The described process produced a dataset of 7600 compositions where 762 different components of the possible 1000 appear at least once. The distribution of component frequencies in the dataset is shown in figure 7.5. A small bin size of 10 was chosen to highlight that almost 250 components appear very sporadically in the dataset. The most frequent component appears 950 times which is significantly less than the dataset size. Therefore, no components are ubiquitous in the synthetic dataset.

The histogram of composition sizes is shown in figure 7.6. The average and median size of compositions is 9, and the largest composition has 21 components. The distribution of the number of arcs per composition is shown in figure 7.7. The average number of arcs in a composition is 16.4, with a median of 16 and a maximum of 52.

The distribution of neighbor diversity in the synthetic dataset is shown in figure 7.8. The shown distribution is a result of the interplay of base relative frequencies of components and the parameters α. Even though α parameters are chosen uniformly, the distribution is decidedly nonuniform. This can be explained through a simple model that focuses only on the effect of

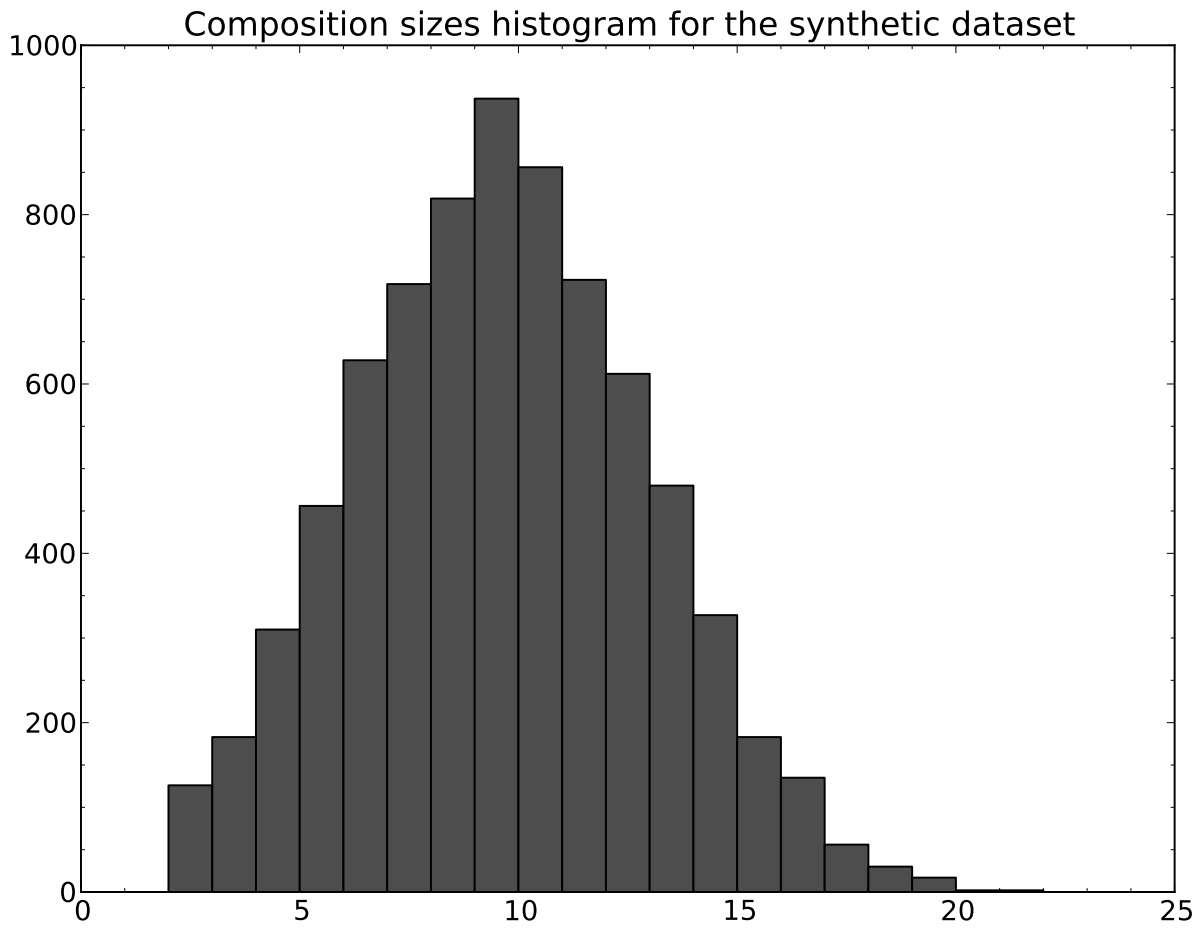


Figure 7.6: The histogram of composition sizes in the synthetic dataset with a bin size of 1.

the α parameter on neighbor diversity.

Given a component C with the neighbor diversity parameter α, the affinity of C to the kth component in some fixed order of components specific to C is defined as α^k. Assume that an arc (C, D) will appear in the dataset if and only if the component affinity $\mathcal{A}_{C,D}$ is at least some constant $p < 1$. This would, in other words, mean then the neighbor diversity of component C would equal the number of such components D that have affinity $\mathcal{A}_{C,D} \geq p$. Denote neighbor diversity with nd. Then

$$\alpha^{nd} < p, \tag{7.1}$$

because k starts at 0. Taking logs reveals the relationship between neighbor diversity and α in this simple model:

$$nd \sim \frac{1}{\log \alpha}. \tag{7.2}$$

In words, neighbor diversity is inversely proportional to the logarithm of α. Therefore, the range of α values that produce a certain value of neighbor diversity decreases quickly as neighbor

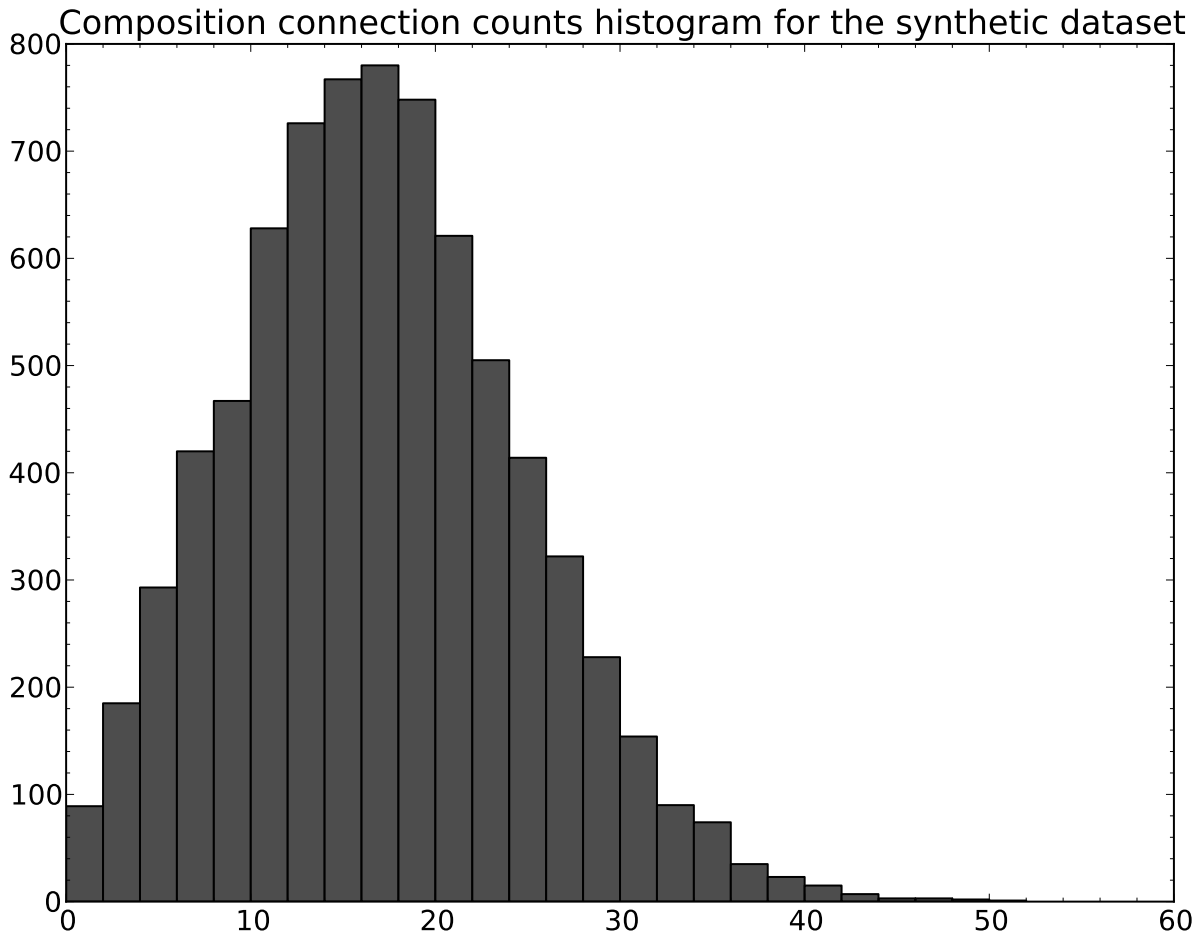


Figure 7.7: The histogram of arc counts in compositions in the synthetic dataset with a bin size of 2.

diversity increases, and this property is clearly reflected in the histogram—most components have low neighbor diversity, there is a large number of components with neighbor diversity between 30 and 150, and only about 50 components have neighbor diversity over 150.

This property of the dataset models the fact that most components usable with *Geppeto* are specific and can't be meaningfully connected to a large number of other components. On the other hand, a small number of generic programmable widgets can be connected to nearly every other component.

Components are, on average, repeated 1.43 times in a single composition in the synthetic dataset, and a single component is, at maximum, used 16 times in one composition. In terms of structural complexity, 6515 of the generated synthetic compositions contain at least one cycle, and 483 of the remaining compositions are directed trees.

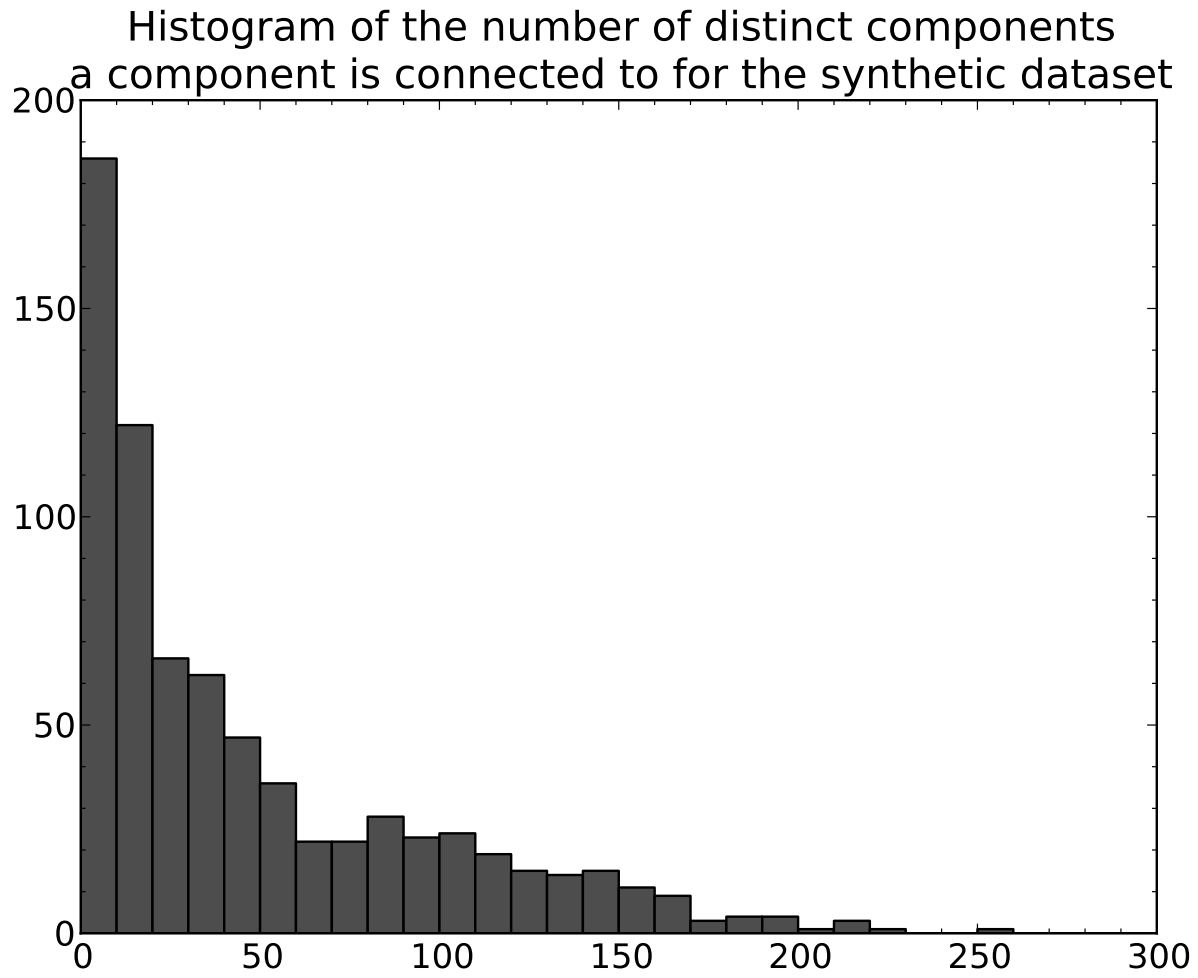


Figure 7.8: The histogram of the number of distinct components a component is connected to at least once in the whole datasets with a bin size of 10.

7.2 Summary and Comparison of Evaluation Dataset Properties

The number of components in *Yahoo Pipes* development is only 55, while 762 components appear in the synthetic dataset. With all else being equal, this would make the problem of component recommendation on the synthetic dataset significantly more challenging. However, a high quality component recommender would also be more useful in a composition system where many components can be used.

There are no ubiquitous components in the synthetic dataset, while in the Pipes dataset, the *fetch* and *output* modules appear in every pipe as their functionality is essential. On the other hand, the distribution of component frequencies is very similar in both datasets. In the synthetic dataset, the 20% of the most frequent components account for about 71% of the total number

of components, while 50% of the most frequent components cover 95% of the total number of components. In the Pipes dataset, these two percentages are 76% and 94%. Therefore, in both datasets, about one half of the available components are used rarely. A similar distribution was found in the use of APIs in mashups listed on the *Programmable Web* website [175].

While the average size of compositions in both the Pipes dataset and the synthetic dataset is close to 9 components per composition, the distribution of composition size is significantly skewed towards smaller compositions in the Pipes dataset, while it is normal in the synthetic dataset. The relatively small number of components per composition is typical for composition systems, especially those where compositions are created through direct manipulation of components in a graphical setting. More complex functionality is then achieved through hierarchical organization where an entire composition is used as a component in another composition.

Compositions in the synthetic dataset are structurally much more complex than *Yahoo Pipes*. Specifically, pipes process feeds in fairly simple patterns and their graph representation is an acyclic data flow graph. On the other hand, the synthetic dataset aims to model more complex compositions where connections arise not only from data flow, but also from control flow, temporal or location-based events, etc.

These structural complexity properties of the datasets are also reflected in the number of connections per composition. For both datasets, the shape of the distribution of the number of connections per composition closely follows that of the composition size distribution. However, while pipes are mostly directed trees, i.e. a pipe with n modules is likely to have $n - 1$ wires which is reflected in the nearly identical distributions of pipe module and wire counts, compositions in the synthetic dataset on average have twice as many connections as components.

In section 7.1.1, the *neighbor diversity* of a composition was defined as the number of distinct components that component is connected to at least once in the entire dataset. In the synthetic dataset, most components have low neighbor diversity relative to the total number of available components and the distribution of neighbor diversity is unimodal. Conversely, in the Pipes dataset, neighbor diversity is more evenly distributed, with two large groups of modules in the 8–14 and the 18–22 ranges.

The average repetition of a component in one composition is only slightly lower in the synthetic dataset than in the Pipes dataset at 1.43 versus 1.57. However, component repetition is qualitatively significantly different in that the most often repeated components in *Yahoo Pipes* are the ubiquitous *fetch* and *urlbuilder* modules, while there are no ubiquitous modules in the

synthetic dataset. Consequently, when evaluating composition similarity in the *GraphEditDistance* algorithm, it is less likely that the same component is repeated many times in both the input partial composition and a database composition in the synthetic dataset than in the Pipes dataset. Due to this qualitative difference, the *GraphEditDistance* algorithm is expected to achieve lower response times on the synthetic dataset than on the Pipes dataset.

7.3 Measures of Recommender Quality

For each query, the recommender produces an output list of R recommended components, where R is a parameter of the used recommender algorithm. The algorithms are evaluated using four measures of recommendation accuracy by comparing the list of recommended components with the set of components called the *useful recommendations* for that query. The set of useful recommendations is chosen in different ways for different experiments and is explained in chapter 8.

The first measure of recommender accuracy is based on the intuition that given a small enough R that all recommendations can be considered by the user at a glance, the recommendation will be successful if at least one recommended component is useful. This measure is called *intersection accuracy* throughout the dissertation. For a particular recommendation of R components, the intersection accuracy of the recommender is set to 1 if at least one of the R components is in the set of useful recommendations for that query. Otherwise, intersection accuracy is 0.

The remaining three accuracy measures are *precision*, *recall* and the F_1 *score*, all of which are commonly used in information retrieval [70, 176, 177]. These measures are based on binary *relevancy* of components for a particular query to the recommender. In the evaluation results in the following chapters, a component is considered relevant for a query if and only if it is in the set of useful components for that query. Let the set of useful components for a query be $\mathcal{U}$ and the set of recommended components be $\mathcal{R}$. Then

$$precision = \frac{|\mathcal{U} \cap \mathcal{R}|}{|\mathcal{R}|}, \text{ and}$$
$$recall = \frac{|\mathcal{U} \cap \mathcal{R}|}{|\mathcal{U}|}.$$

Note that using previously introduced notation, $|\mathcal{R}| = R$, and that precision is equivalent to

intersection accuracy if $R = 1$.

In a probabilistic interpretation, precision is the probability that a recommended component is useful. Conversely, recall is the probability that a useful component is recommended. It is well known that precision and recall are inversely related [46]. Specifically, precision can often be increased by decreasing the number of recommendations R, while recall can be increased by increasing R. For example, perfect recall can always be achieved by recommending every component in in the set of all components $\mathcal{T}$. Therefore, precision and recall are always reported together.

The relative importance of precision and recall depends on the specific use case of the recommender. A low precision algorithm might often lead the user astray by recommending useless components. On the other hand, a low recall algorithm will often miss opportunities to recommend some very useful component.

In this dissertation, this tradeoff between precision and recall is evaluated using the F_1 score, which is the harmonic mean of precision and recall

$$F_1 = \frac{precision \cdot recall}{precision + recall}, \tag{7.3}$$

and is often used to represent both precision and recall with a single number when both are equally important.

In every experiment, the averages of these four measures over all queries are graphed next to each other with equal value ranges on the graph axes.

There are several desirable properties for a recommender except accuracy. In this dissertation, special attention is given to recommendation *coverage*. In general, in the evaluation of recommender systems, coverage is a measure of the size of the set of items for which a recommender can make rating predictions or can recommend to users [46]. A recommender with low coverage is likely to be less useful to users as it is unable to assist them in many choices they might have to make. Conversely, a recommender with higher coverage has a larger chance to recommend components that the user is not aware of.

Herlocker [46] defines *catalog coverage* as the percentage of items in the item set that are actually recommended, usually measured in a set of recommendations in a single point in time. In the evaluation results in the following chapters, catalog coverage is defined to be the percentage of components from the component set $\mathcal{T}$ that get recommended at least once by an algorithm during an experiment.

Furthermore, the recommender algorithms are evaluated for *successful coverage* where only recommendations of useful components are counted towards coverage. Successful coverage, by definition, is at most as high as catalog coverage, and the difference between catalog coverage and successful coverage is the percentage of components that were recommended at least once, but were never actually useful recommendations. Catalog coverage and successful coverage are graphed next to each other with equal value ranges on the graph axes so that they can be compared easily.

Both catalog coverage and successful coverage are analyzed qualitatively using *coverage curves*. A coverage curve is a graph in which the y-axis represents the frequency of recommendations of a particular component, and the x-axis represents the rank of that component when components are sorted in nonincreasing order by that frequency. For example, the most frequently recommended component has rank 0 and its frequency is graphed directly on the y-axis of the coverage curve.

For every algorithm, the area under the coverage curve for catalog coverage is equal to the number of queries multiplied by the number of recommendations per query R. However, the distribution of that area reveals how the algorithm achieves its catalog coverage. For example, an algorithm might achieve high catalog coverage by recommending components randomly. Then the coverage curve for its catalog coverage would be close to a horizontal line. On the other hand, an algorithm that only recommends some of the most popular components will have a very steep coverage curve that quickly touches the x-axis.

The same idea is also applied to successful coverage where the total recommendation frequency of components on the y-axis is replaced with the frequency of successful recommendations.

Finally, the recommenders are evaluated on response time which is defined as the length of the interval between the moment when the recommender receives a query and the moment when it produces a list of recommended components. All the algorithms and the testing framework were implemented in Python 3 and the experiments were run on a workstation with an Intel Core2 Q9400 processor and 4GB of RAM, under Windows XP.

7.4 Baseline Recommender Algorithms

Along with the four structural recommender algorithms defined in chapter 6, three simple recommender algorithms that provide useful information about the datasets and serve as baselines are also evaluated. The algorithm *WeightedRandom* recommends R components picked from the set of all components $\mathcal{T}$ at random. The probability that a particular component is chosen is proportional to its frequency in the composition database for a particular experiment. When $R = 1$, the probability that a component is recommended is equal to its relative frequency in the composition database, and goes up with R.

The algorithm *MostPopular* deterministically recommends the R components that are most frequent in the composition database.

Finally, the algorithm *MostFreqConn* ranks components by how often they are connected to components in the input partial composition P in the composition database and then recommends the R most frequent components. Specifically, during initialization, for each pair of components $u, v \in \mathcal{T}$, the algorithm counts how often the arcs (u, v) and (v, u) appear in the graphs in the composition database. Then when making recommendations for a partial composition P, for every component u of P and every component $v \in \mathcal{T}$ this frequency is added to the score of component v. Finally, the components with the highest R scores are selected, sorted by score and recommended. Therefore, the time complexity of this algorithm is $O(n_P |\mathcal{T}| + R \log R)$.

Note that these baseline algorithms do not conform to the recommender method described in section 6.1 in that they do not search for similar compositions in the database for each query. Rather, they extract statistics from the composition database at initialization time, before any query has been processed. In a practical setting, these statistics would be updated dynamically as new compositions are added to the composition database.

As the structural recommender algorithms, these baseline algorithms also never recommend components that have already been used in the input partial composition P, i.e. these compositions are filtered out before components are recommended.

A summary of the definition of these three baseline algorithms is shown in table I.

Table I: A summary of the three baseline algorithms.

Algorithm	Recommended components
WeightedRandom	R components not already in P chosen at random with probability proportional to their frequency in the composition database.
MostPopular	R components not already in P that are the most frequent in the composition database.
MostFreqConn	R components not already in P that most frequently connected to components in P in the composition database.

7.5 Evaluating Probabilistic Algorithms

Out of the evaluated algorithms, *GraphEditDistance* and *WeightedRandom* are probabilistic algorithms, i.e. they employ a random process in their operation. Furthermore, two of the strategies used for simulating composition development in chapter 9 are also randomized and all the recommender algorithms employed under those strategies are treated as probabilistic because their inputs vary depending on the random process used in this simulation strategy.

Throughout the presented evaluation results, average recommendation accuracy and average recommender response time are reported. When evaluating a probabilistic algorithm, the evaluation was repeated 30 times. The resulting distribution of average accuracies and response times can be approximated with a normal distribution by the central limit theorem. In the remainder of this section, only accuracy is discussed, but the exact same procedure was applied when computing average recommender response times.

The reported accuracy estimate is the mean of the average accuracies over the 30 runs of the experiment. Since the true mean is unknown, the unbiased estimator of the standard deviation is

$$\hat{s} = 1.0086\sqrt{\frac{\sum_{i=1}^{30}(x_i - \bar{x})^2}{29}} \tag{7.4}$$

where x_i is the average accuracy for the ith run of the experiment and $\bar{x}$ is the mean of all x_i.

Since the true standard deviation of the distribution is unknown, the *t*-distribution was used to compute confidence intervals. For 29 degrees of freedom and $\alpha = 0.05$, the half-width of the confidence interval is computed as

$$CI = 2.045\frac{\hat{s}}{\sqrt{30}}. \tag{7.5}$$

When a result was computed in this manner, the legend of a graph shows error bars. How-

ever, the results were very precise in most experiments so the computed confidence intervals are nearly invisible on most graphs.

In addition to average recommender response times, maximum response times were also analyzed. For a probabilistic algorithm, the reported maximum response time is the median of the maximum response times in the 30 runs of the experiment.

Finally, the average coverage in the 30 runs is reported, with a confidence interval $\sqrt{30}$ times larger than in 7.5 as it is not a mean of sample means, but a simple mean of 30 experiments. For the purposes of qualitative coverage analysis via coverage curves, results of the first of the 30 runs are reported for the probabilistic algorithms.

7.6 Chosen Algorithm Parameter Values

Examining the effects of similarity filtering for component scores computation and of the number of recommended components per query on recommendation quality is one of the focus points for the evaluation presented in this dissertation. These effects are evaluated by varying the parameter BP which regulates similarity filtering as the recommender only considers the BP percent most similar compositions in the composition database when computing component scores, and the parameter R which defines the number of components that get recommended for each query. For experiments that focus on different aspects of the recommender algorithms, these parameters are fixed to $BP = 10\%$ and $R = 3$. These parameter values are supported by evaluation results, and these choices are discussed in detail in chapter 8.

The edit distance parameters of the *ComponentSeqEditDistance* and *GraphEditDistance* algorithms are good candidates for model selection techniques [178, 179]. Alternatively, the parameters can be set based on domain knowledge about the particular composition system. During preliminary research, the edit distance parameters of both *ComponentSeqEditDistance* and *GraphEditDistance* were optimized using a simple hill climber over several choices of BP and R with 200 random starting parameter values and up to a hundred iterations with both the Pipes dataset and the synthetic dataset. Interestingly, the F_1 score achieved by the algorithms stayed within several percent for all runs of the algorithms, i.e. with a wide range of different parameters. There are two explanations for this behavior. First, a lot of the edit operations for a specific P-Q pair are fixed and can't be changed regardless of the parameter values. Specifically, all the components in P that don't appear in Q must necessarily be removed, and all the

components that appear only in Q must be added. Second, the absolute value of the computed edit distances affects the final recommendations only slightly—recommendations are made primarily based on the relative differences in the computed similarity scores, and changing edit distance parameters typically only scales the computed edit distances up or down.

Due to the large number of different experiments that were conducted during evaluation of the recommender algorithms, the relatively high duration of each experiment, and the probabilistic nature of the *GraphEditDistance* algorithm, optimizing these parameters for every experiment was infeasible. Based on this fact and the indication that varying edit distance parameters has a relatively small effect on recommendation accuracy, the parameter values $C_{add} = C_{rem} = 1.0$ and $C_{conn} = 0.5$ were used consistently in all experiments, and this issue was left for further research.

Finally, in all the presented results, the parameter L of the *GraphEditDistance* algorithm, which regulates the maximum number of matching choices considered by the algorithm, was set to $\max\{(n_P + 2)^2, 100\}$. This allows progressively more choices to be explored when the partial input composition P gets larger, while also establishing a constant upper limit which then provides a guarantee for worst case performance. This constant upper limit is especially important for unusually large input partial compositions with a lot of repeated components, for example the pipe with 95 *fetch* modules, which would otherwise require an impractically large amount of time to process. Note that this means that for such input partial compositions, the *GraphEditDistance* algorithm might not be able to produce high quality recommendations, but in many cases, producing high quality recommendations for such compositions might be impossible as they are so different from other compositions.

Chapter 8

Snapshot Evaluation

In this chapter, the first evaluation scenario for evaluating component recommendation algorithms based on *composition snapshots* is presented. A composition snapshot (or simply, a *snapshot*) represents a particular stage of development of a composition. In particular, a snapshot is a subgraph of a previously completed composition. A set of components that can be useful for completing the snapshot into that previously completed composition is assigned to every snapshot—these components are then called *useful components* or *useful recommendations* in the sense that the user might benefit from their recommendation in the stage of development of the composition that is represented by the snapshot. The snapshot subgraph is then provided as the input partial composition to the recommender and its output is compared to the set of useful recommendations.

An example of a composition and its snapshot is shown in figure 8.1. The completed composition from the dataset is created by combining components A through E, with two instances of component B. A possible snapshot of this composition consists of components B, C and E which are marked in the figure with a gray fill. Additionally, the snapshot subgraph has the single arc (B, C) which is marked with an asterisk.

There are several possible ways to define the set of useful recommendations for a snapshot, and two of these choices are discussed in this chapter. The simplest definition of the set of useful components of a snapshot is the set difference of the component set of the completed composition from the dataset and the component set of the snapshot. For the example in figure 8.1, components A and D would be considered useful in this definition. Note that component B is not considered a useful recommendation because it is already a part of the snapshot, even though another instance of it is required to complete the snapshot, as discussed in section

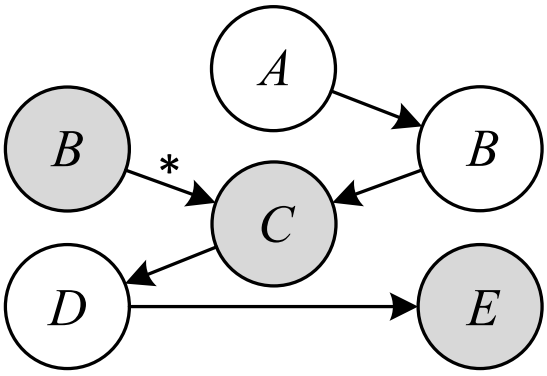


Figure 8.1: A *snapshot* of a composition. The vertices in the snapshot have a gray fill and the only arc in the snapshot is marked with an asterisk.

6.1. This definition of useful recommendations for a snapshot is used throughout most of this chapter.

The goals of the experiments presented in this chapter are as follows. First, the effect of similarity filtering is evaluated because some algorithms, depending on how they evaluate similarity, might experience degradation of recommendation quality due to the influence of the long tail of the composition similarity distribution on the computed component scores. In other words, when computing component scores in the absence of similarity filtering, the cumulative effect of all database compositions that are not very similar to the input partial composition can outweigh the effect of a few very similar compositions, which, in turn, can make the recommender choose poor recommendations.

Choosing the number of components to recommend per query R is a central issue in the design of a component recommender system. If too many components are recommended, users are likely to ignore most recommendations most of the time. On the other hand, if too few components are recommended, opportunities for useful recommendations can be missed. In this respect, choosing R is largely a matter of user interface design. While these properties can't be directly evaluated with an off-line experiment, information about the effect of different values of R on recommender quality can be used as a guideline for choosing an appropriate value.

The effect of the size of the composition database N is evaluated primarily to explore how response times of the various recommender algorithms increase for larger composition databases. The changes of other recommender quality measures are then useful for analyzing the tradeoff between, for example, response time and accuracy as the composition database size changes. Specifically, for some component recommendation use cases, it might be beneficial to

only use a part of the whole composition database when making recommendations. Additional considerations regarding composition database size are discussed in section 8.4 at the end of this chapter.

The effect of arcs in the input partial composition is evaluated because users might create applications in various ways. On one extreme, a user might make all the required connections between a set of components before adding another component to the composition workspace. This situation is explored throughout most of the chapter. However, on the other extreme, a user might not make any connections before all the components required for making a composite application are on the composition workspace. It is important that a component recommender is able to provide useful recommendations in both these situations, and this is a major reason why all the presented algorithms are component-centric in that they base their similarity evaluation mostly on components included in compositions. Through the results in these extreme cases, the goal is to characterize the general effect of arcs in the input partial composition on recommender quality.

Finally, as noted above, different definitions of useful recommendations can be meaningful. The simplest definition of usefulness assumes that all the unused components of a completed composition can be useful for completing a snapshot, i.e. a partial composition. However, in a large composition, the user might not be able to identify a component that is in some sense distant from the partial composition as a useful recommendation, even though that component will eventually be used. For example, a component might be a leaf of the composition graph and the path that leads to it from the nearest snapshot vertex could be long. This issue is further analyzed in the next chapter, but in this chapter, a definition of useful components that is restricted only to the frontier of the snapshot is also considered. Specifically, in this definition, a component is considered useful only if it is directly connected to the snapshot in the completed composition. Such a component is then called *adjacent-useful*.

The rest of this chapter is organized in the following way. First, in section 8.1, the process by which snapshots were generated is explained. Then in sections 8.2 and 8.3, the results of snapshot evaluation on the Pipes dataset and on the synthetic dataset are presented with an emphasis on the changes in accuracy, coverage and response time with different algorithm parameter settings and input and environment properties. In both sections, the generated snapshot set is briefly analyzed, and the following algorithm properties are evaluated:

- the effect of similarity filtering for component score computation, governed by the BP

parameter

- the effect of changing the number of components recommended per query R
- the effect of the number of compositions in the composition database, denoted by N
- the effect of arcs in the input partial composition
- the effect of the *adjacent-useful* definition of useful recommendations

Finally, the main results of snapshot evaluation for both datasets are summarized and discussed in section 8.4. This section includes both an analysis of recommender quality in the absolute sense, and a summary of the effects discussed in sections 8.2 and 8.3.

8.1 Snapshot Generation

Composition snapshots were generated from both composition datasets in the following way. First, 500 distinct compositions were selected from the dataset, uniformly at random. Then, 2000 snapshots were generated from the 500 chosen compositions. When generating each snapshot, one of the 500 compositions was chosen uniformly at random, i.e. an average of four snapshots were generated from each of the 500 compositions.

For a complete composition of size n, the size of the snapshot was chosen from the Gaussian distribution of integers in $[1, n-1]$ with $\mu = n/2$ and $\sigma = n/4$. This distribution of size was chosen over a uniform distribution because the uniform distribution significantly favors small snapshots as they are available in every composition. For a given snapshot size n_s, n_s vertices were chosen from the complete composition, uniformly at random. The snapshot subgraph was then defined as the subgraph induced by the chosen vertices, i.e. all the arcs between the chosen vertices were included in the snapshot.

Finally, the set of useful recommendations for that snapshot was computed as explained above. If this set turned out to be empty, thus making the snapshot useless for evaluating the recommender, the snapshot was discarded and a new snapshot was generated instead.

The same set of snapshots was used for all the conducted experiments. The remainder of the dataset, i.e. the dataset excluding the 500 compositions selected for snapshot generation, was used to generate databases of previously completed compositions that the algorithms use to make recommendations. Databases of size 100, 750, 1500, 2500, 4500 and 7000 were used in the experiments, and this size is denoted by the capital letter N. The databases were generated by initially randomly shuffling the remainder of the dataset after the 500 compositions used for

snapshots were removed. Then the database of size N is simply the first N compositions in the list. In this way, a database of a smaller size is a subset of any database of a larger size, and, for any fixed size N, the composition database is always the same.

Except when evaluating the effect of different composition database sizes on recommender quality, a database size of $N = 2500$ was used in all experiments presented in this chapter.

8.2 Snapshot Evaluation Results on *Yahoo Pipes*

In the set of 2000 snapshots generated from the *Yahoo Pipes* dataset, 54 out of the available 55 modules appear at least once in some snapshot. The average size of a snapshot is around 4.3 modules with 2 wires. This sparsity is to be expected as pipes are very sparse to begin with, and whenever nonadjacent vertices are selected for a snapshot, a possible path between them in the completed compositions will not be present in the snapshot in any way, i.e. arcs of the original snapshot will be lost.

Importantly, the three most popular modules in *Yahoo Pipes*, namely the *fetch*, *output*, and *urlbuilder* modules, are ubiquitous in the snapshots as well. The *fetch* module appears a total of 1564 times, the *output* module 835 times, and the *urlbuilder* module 615 times.

In general, the module frequency distribution in the snapshots remained very similar to that of the whole dataset with 72% and 94% of the total module frequency being covered by the most frequent 20% and 50% of the modules, respectively.

The evaluation results presented in this section are organized in the following way. First, the effects of similarity filtering for component score computation on recommender quality are evaluated in subsection 8.2.1. Then, subsection 8.2.2 focuses on the issue of selecting the number of components to recommend per query. In subsection 8.2.3, the effect of composition database size is evaluated, primarily in terms of recommender response time. The effect of arcs in the input partial composition is explored in subsection 8.2.4. Finally, recommender accuracy under the *adjacent-useful* definition of useful recommendations is examined in subsection 8.2.5.

8.2.1 Effects of Similarity Filtering in Computing Component Scores

Similarity filtering in computing component scores is controlled by the BP parameter which all four structural component recommender algorithms presented in this dissertation share. The BP parameter determines the percentage of best matching compositions from the composition

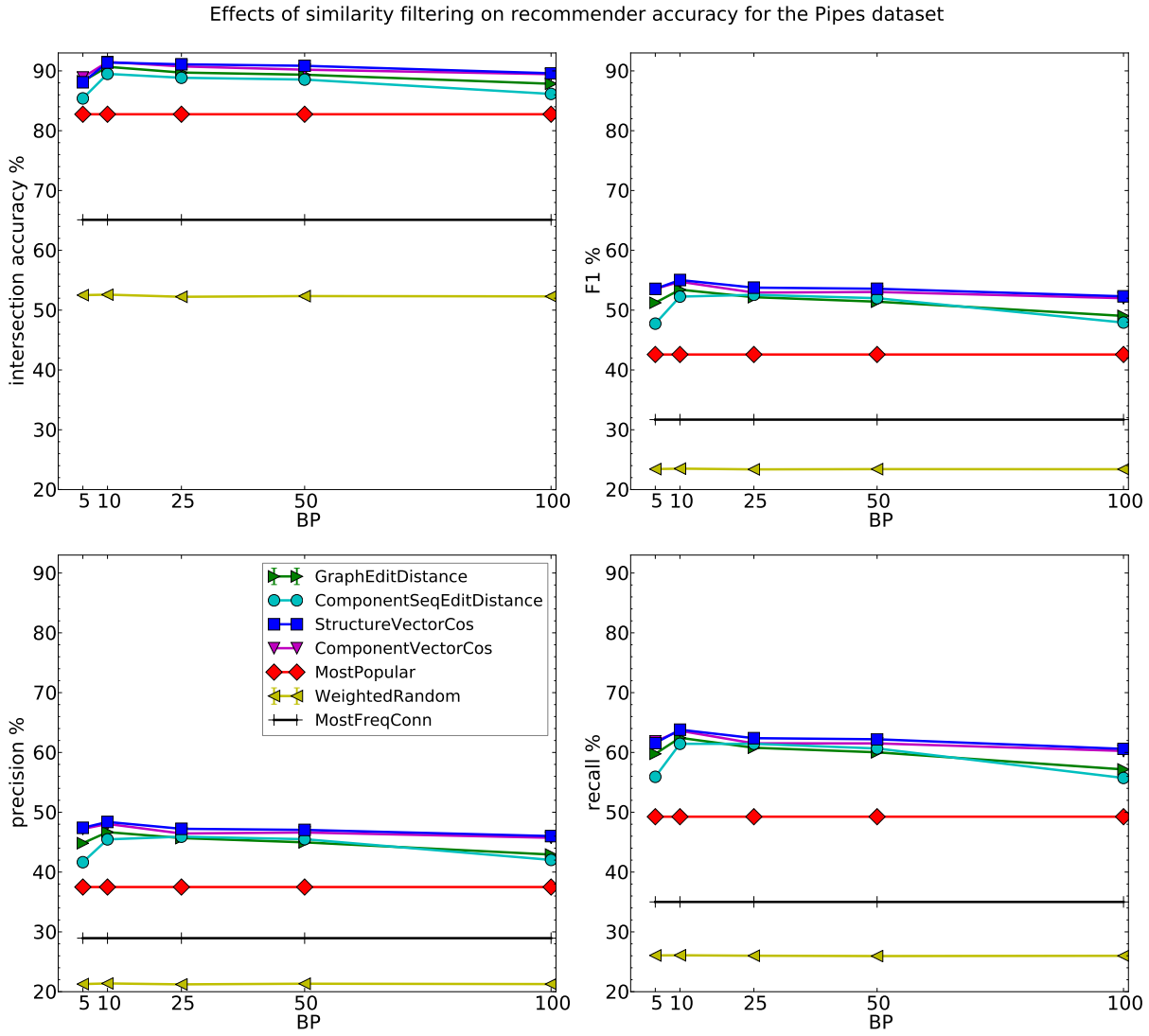


Figure 8.2: Changes in recommender accuracy for BP between 5 and 100 percent with $R = 3$ and $N = 2500$ for the Pipes dataset.

database that are considered in component scores computation. Specifically, a lower value of BP eliminates database compositions on the tail end of the similarity distribution.

The changes in recommendation accuracy for all four structural recommender algorithms and the three baseline algorithms for values of BP between 5% and 100% with $R = 3$ and $N = 2500$ are shown in figure 8.2. In the first row, intersection accuracy is graphed on the left and the F_1 score on the right, while the second row shows precision and recall graphs. For this experiment, all four accuracy measures are qualitatively the same, i.e. are effected in the same way by similarity filtering.

The results for the baseline algorithms form horizontal lines because these algorithms do not depend on the BP parameter in any way as they are not based on evaluating similarity between compositions and thus don't do similarity filtering.

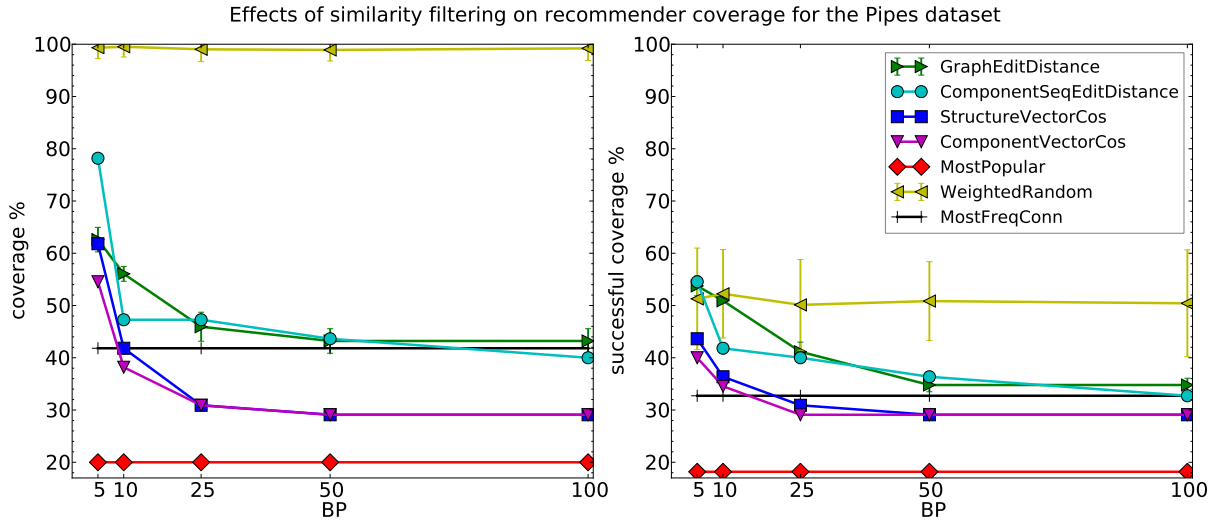


Figure 8.3: Changes in recommender coverage for BP between 5 and 100 percent with $R = 3$ and $N = 2500$ for the Pipes dataset.

The feature vector algorithms *ComponentVectorCos* and *StructureVectorCos* are almost unaffected by similarity filtering for BP values above 10%. Similarly, the accuracy of both *ComponentSeqEditDistance* and *GraphEditDistance* decreases slightly as BP is increased, by close to 3% as BP is increased from 10% to 100%. As both *ComponentSeqEditDistance* and *GraphEditDistance* are affected in almost the same way, it seems that the general shape of the similarity score based on edit distances, which is similar for both algorithms, makes these algorithms more sensitive to similarity filtering.

The accuracy for all four structural algorithms decreases by several percent when BP is reduced from 10% to only 5%. This is because, for some queries, the algorithms fail to score at least R new components using only 125 of the most similar completed pipes, and are therefore unable to recommend R components with a nonzero score. This occurs for queries where the snapshot represents a large subgraph of a particularly common pattern found in *Yahoo Pipes* and most of the 125 most similar pipes are in fact nearly identical and mostly contain only those modules already found in the snapshot.

Similarity filtering also has a significant positive effect on coverage of all four structural algorithms, as shown in figure 8.3. Catalog coverage is shown in the graph on the left, while the right graph shows successful coverage where only useful recommendations are counted. Again, the baseline algorithms are independent of BP as they do not process the composition database for each query so their coverage remains constant, with fluctuations in *WeightedRandom* due to randomness.

Over 2000 recommendations, the *WeightedRandom* algorithm recommends nearly all of the

55 available modules at least once. However, only around 50% of these modules are ever useful. The same level of successful coverage is achieved by *GraphEditDistance* for BP in the 5–10% range. Furthermore, *ComponentSeqEditDistance* and *GraphEditDistance* benefit more from more aggressive similarity filtering than the feature vector algorithms. The *ComponentSeqEditDistance* algorithm achieves more than 30% higher coverage as BP is reduced from 10% to 5%, but only a 13% increase in successful coverage in that BP range. The *GraphEditDistance* algorithm gains nearly 20% of both catalog coverage and successful coverage as BP is reduced from 50% to 5%. Catalog coverage of both feature vector algorithms doubles in this BP range, but about one half of this increase is achieved through inaccurate recommendations.

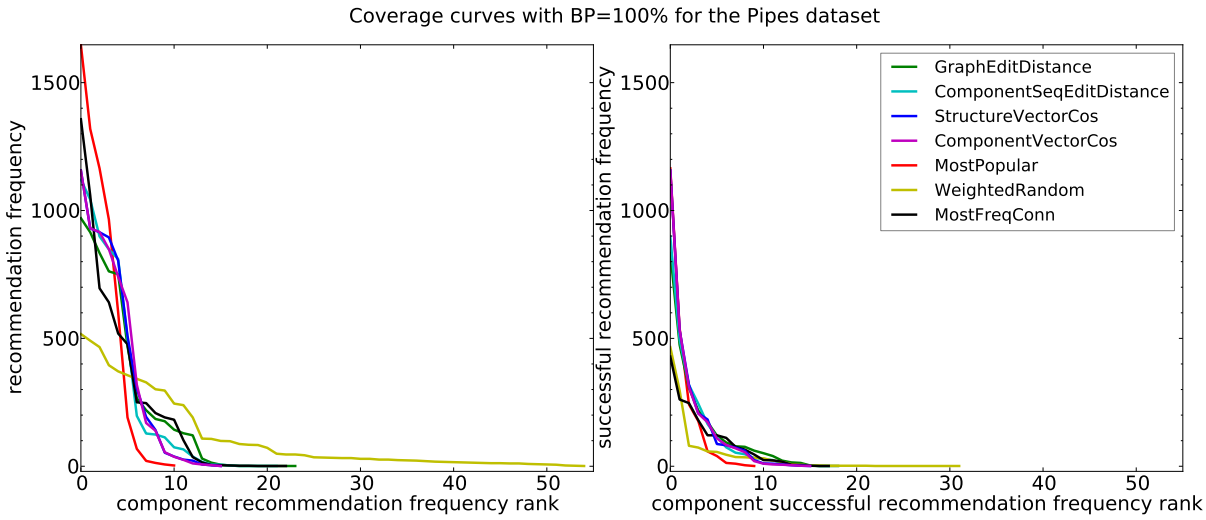


Figure 8.4: Coverage curves for $BP = 100\%$ for the Pipes dataset with $R = 3$ and $N = 2500$.

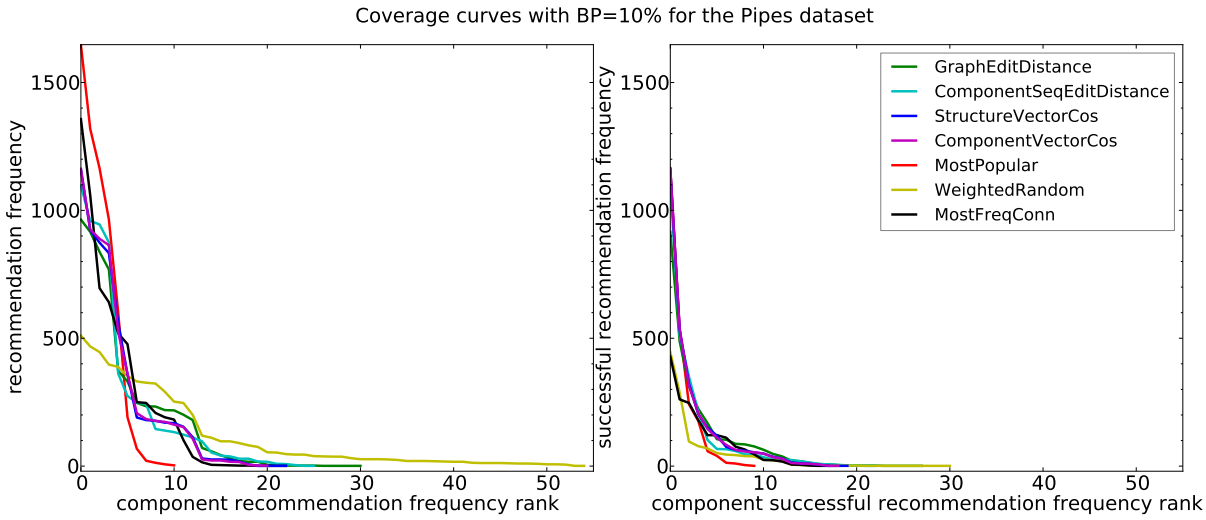


Figure 8.5: Coverage curves for $BP = 10\%$ for the Pipes dataset with $R = 3$ and $N = 2500$.

Both catalog coverage and successful coverage are next analyzed qualitatively using cov-

erage curves. The coverage curve for catalog coverage of an algorithm is generated by sorting components that have been recommended at least once in nonincreasing order by recommendation frequency, and then graphing recommendation frequencies in that order. The coverage curve for successful coverage is generated in an analogous way, except that only successful recommendations are counted.

The coverage curves for the *Yahoo Pipes* dataset with no similarity filtering, i.e. $BP = 100\%$ are shown in figure 8.4, with catalog coverage on the left and successful coverage on the right. As these coverage curves are based on 2000 recommender queries with $R = 3$, the area below the catalog coverage curve for each algorithm is exactly 6000. As expected, the *MostPopular* algorithm has the steepest coverage curve, recommending only 11 of the 55 modules. On the other hand, *WeightedRandom* has the flattest coverage curve.

The coverage curves for all four structural algorithms are similar—they all recommend the most recommended module for around one half of all queries, and there are seven modules in total that are recommended very frequently. The remaining modules are recommended in fewer than 300 queries, and all the modules after the 16th most recommended module are only recommended once or twice.

The coverage curves for successful coverage are similar in shape, but contracted towards the y-axis. In general, the area under the coverage curve for successful coverage equals the number of successfully recommended components. For the structural algorithms, only 9 Pipes modules are recommended successfully more than a hundred times. The *GraphEditDistance* algorithms separates itself slightly from the other algorithms around the 10th rank, and with a slightly longer tail, shorter only than that of the *WeightedRandom* algorithm.

The qualitative effect of similarity filtering on coverage can be seen from comparing the previously discussed figure 8.4 and figure 8.5, which shows coverage curves with aggressive similarity filtering with $BP = 10\%$. Two significant differences can be seen in the coverage curves for catalog coverage. First, when similarity filtering is applied, there is a plateau in recommendation frequency between the 7th and the 13th ranked modules for all four structural component recommenders. Second, the tail of the coverage curve is longer for all four algorithms, especially for *ComponentSeqEditDistance* and *GraphEditDistance*.

A similar effect is seen in successful coverage in that the coverage curves are slightly raised around the 10th rank and have a longer tail.

Similarity filtering has a negligible effect on recommender response time, as shown in figure

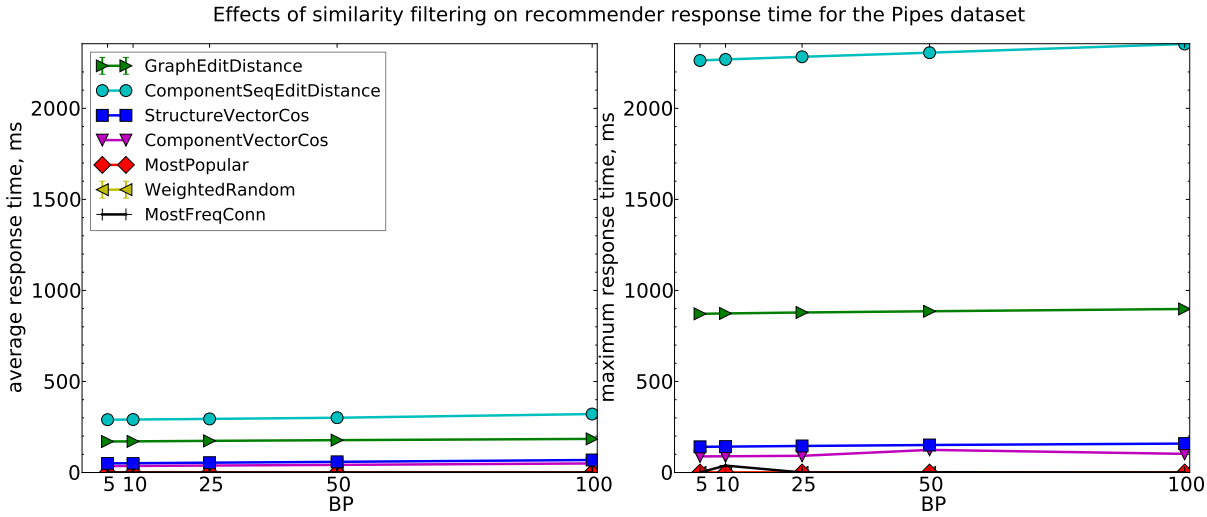


Figure 8.6: Changes in recommender response time for BP between 5 and 100 percent with $R = 3$ and $N = 2500$ for the Pipes dataset.

8.6. The average response times are shown in the graph on the left, while maximum response time is shown on the right. The baseline algorithms respond practically instantly for every query, so their graphs are on the x-axis. Both values decrease very slightly with aggressive similarity filtering for all four structural algorithms, but response time should not be taken into account when choosing the BP parameter. These graphs show that, although both similarity evaluation and component scores computation process the whole composition database, similarity evaluation completely dominates in its effect on response time for all four algorithms.

Based on the results presented in this subsection, a BP value of 10% was used for the remainder of the experiments. While more aggressive similarity filtering provides a further increase in coverage, accuracy decreases slightly and would decrease considerably for smaller database sizes.

8.2.2 Effects of the Number of Recommended Components Per Query R

The effect of changing the number of recommended components R from 1 to 10 on recommendation accuracy is shown in figure 8.7. As expected, increasing R increases intersection accuracy and recall, but decreases precision for all algorithms. Consequently, the F_1 score increases for small values of R and then decreases for larger R. Specifically, for all four structural recommender algorithms, the maximum F_1 score is achieved for $R = 2$, with $R = 3$ being a close second.

ComponentVectorCos and *StructureVectorCos* perform virtually identically at all data points.

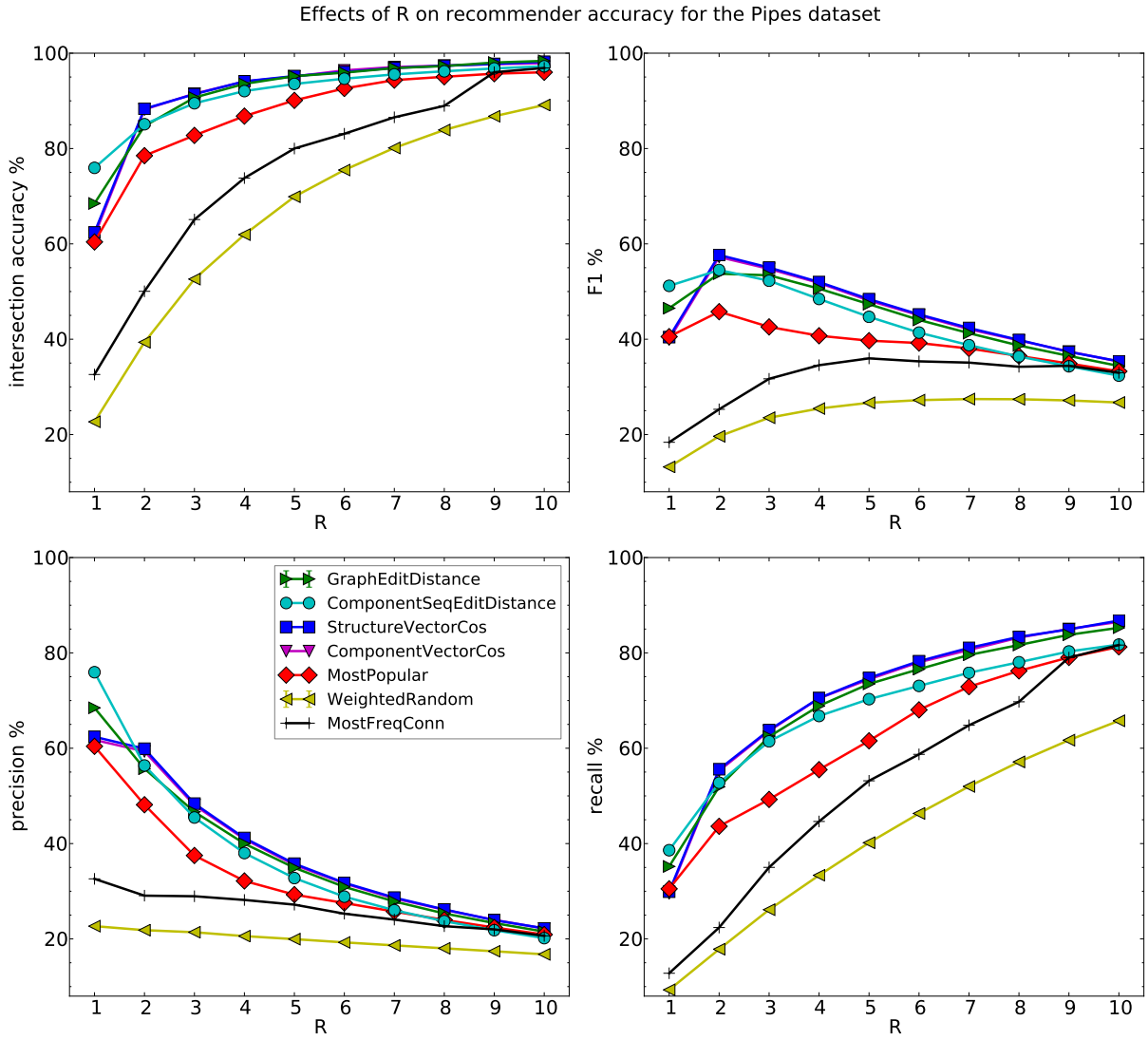


Figure 8.7: Changes in recommender accuracy for R between 1 and 10 with $BP = 10\%$ and $N = 2500$ for the Pipes dataset.

For all R greater than 1, they either match or marginally outperform *GraphEditDistance*, which consistently outperforms the *ComponentSeqEditDistance* algorithm by between 2 and 5 percent.

Of the baseline algorithms, *MostPopular* performs most closely to the structural algorithms, with similar qualitative dependence on R. On the other hand, *WeightedRandom* and *MostFreqConn* increase their intersection accuracy and recall the most as R is increased, but they also start with very low accuracy for small values of R. Furthermore, their precision decreases very slowly with R, staying mostly between 20 and 30 percent. These two effects produce an almost constant F_1 score beyond $R = 5$.

The effect of changing the number of recommended components per query on recommender coverage is shown in figure 8.8. As can be expected by its definition, coverage increases close to linearly with R. Excluding the *WeightedRandom* algorithm which is expected to achieve high

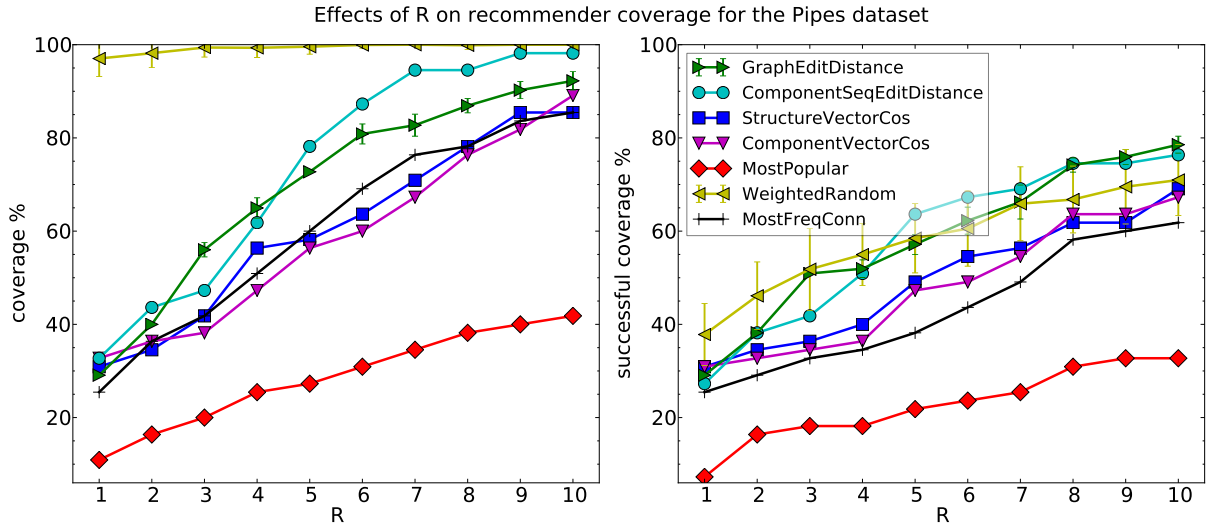


Figure 8.8: Changes in recommender coverage for R between 1 and 10 with $BP = 10\%$ and $N = 2500$ for the Pipes dataset.

coverage, *GraphEditDistance* and *ComponentSeqEditDistance* recommend the largest number of different modules across all values of R, both in catalog coverage and successful coverage.

While their accuracy is nearly identical, *StructureVectorCos* does consistently outperform *ComponentVectorCos* in coverage, indicating that the inclusion of arc information does allow for slightly more focused recommendations.

For the Pipes dataset, special attention needs to be given to the $R = 1$ case as it is especially affected by the three ubiquitous modules *fetch*, *output* and *urlbuilder*. For example, since every pipe contains one instance of the *output* module, an *output* recommendation will be successful for any query where *output* is not already part of the snapshot. This happens in 1165 snapshots. The *fetch* module, which is the most frequent module in *Yahoo Pipes*, appears 1564 times in the 2000 snapshots so it is very often not a valid candidate for recommendation. The *MostPopular* algorithm will recommend the *fetch* or *output* module every time it is not a part of the snapshot. Furthermore, for such snapshots, both feature vector algorithms will also always recommend either *fetch* or *output* because they increase scores of *all* components in Q that are not in P in their COMPUTESCORES function. Particularly, the score of the *output* module will simply be the sum of all similarity scores over all the considered database compositions.

This is good intuition about how these feature vector algorithms work. They recommend the most popular components in the composition database, but weighted by similarity scores. In other words, they recommend the most popular components in the compositions that are similar to the input partial composition.

On the other hand, *ComponentSeqEditDistance* and *GraphEditDistance* sometimes recommend other modules even for snapshots that don't contain *fetch* or *output* as they increase component scores only for those components that are directly connected to matched components.

When both *fetch* and *output* are a part of the query snapshot, all algorithms are forced to recommend some other module. As can be seen from the coverage graphs shown in figure 8.8, the recommendations of *MostPopular* and the feature vector algorithms diverge in this case with the feature vector algorithms recommending about five times as many different modules as *MostPopular*, but their accuracy still turns out to be almost identical. When R is increased beyond 1, the feature vector algorithms consistently outperform *MostPopular* across all measures.

In a practical setting, special components essential for each composition, such as the *fetch*, *output*, and *urlbuilder* modules for Pipes, would need to be handled separately as recommending them is likely not very useful as all users except complete novices will be aware of them and their role in a composition.

The effect of the number of recommended components on recommender response time is negligible as response time is clearly dominated by similarity evaluation, so it is not considered here.

8.2.3 Effects of Composition Database Size N

All four accuracy measures are qualitatively affected by changes in the composition database size in exactly the same way, as shown in figure 8.9 where the results for several values of N from 100 up to 7000 are graphed.

Accuracy remains nearly constant for N beyond 750 for all four structural algorithms. This can be explained by the fact that the *Yahoo Pipes* dataset contains many similar pipes so a fairly small sample of the composition set provides sufficient information to make useful recommendations for most queries. The baseline algorithms perform slightly better as the composition database size increases as the statistics they collect from the database become more accurate predictors of their true values.

Interestingly, all three baseline algorithms have increased accuracy for the smallest database size tested ($N = 100$), while all three structural algorithms lose accuracy for that case. The results for a database size of 100 will depend largely on which 100 compositions are selected, i.e. a large variance of accuracy can be expected for different samples. Obviously, the particular 100 compositions selected for these experiments provided useful statistics to the baseline

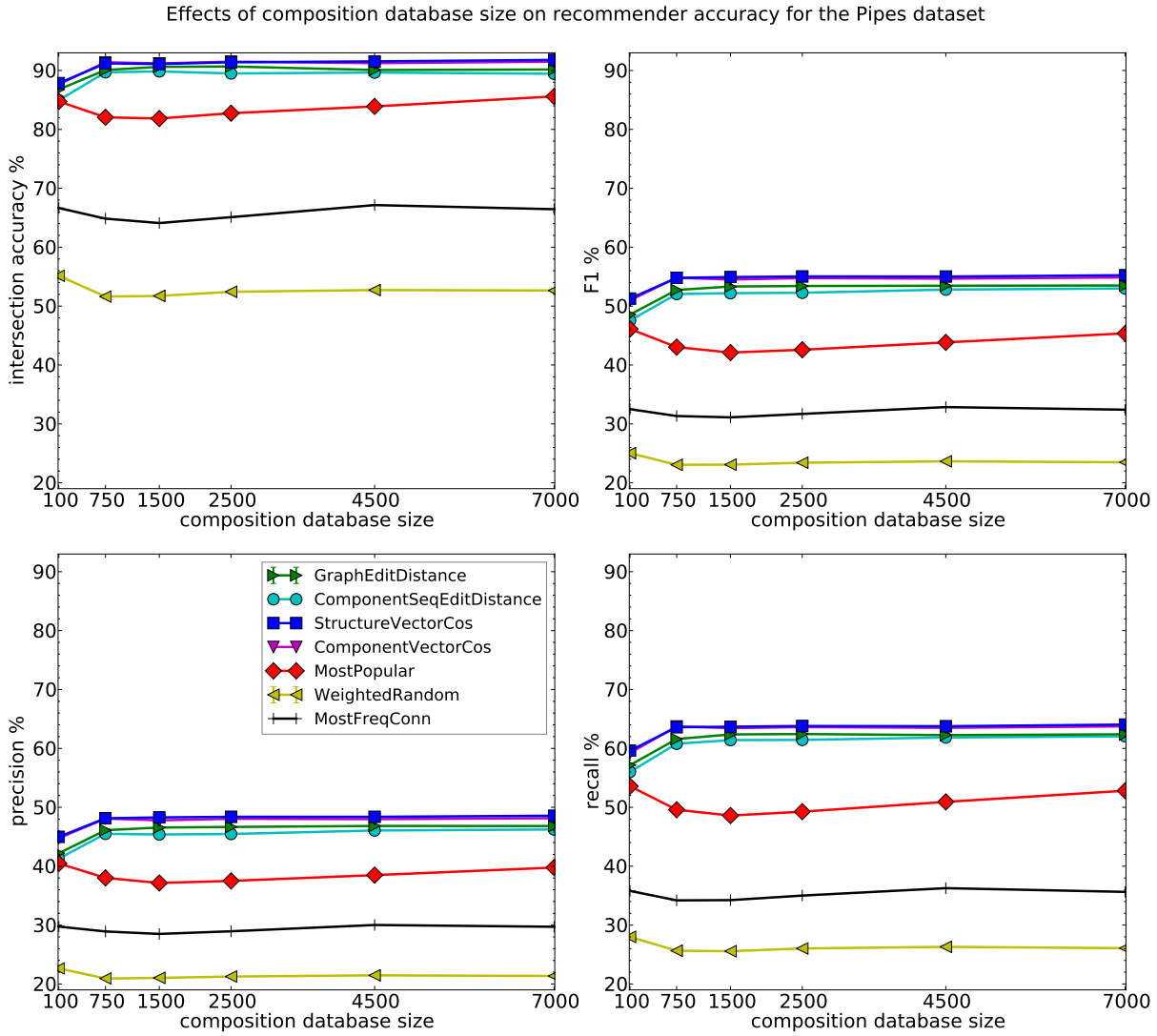


Figure 8.9: Changes in recommender accuracy for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the Pipes dataset.

algorithms which allowed them to perform slightly better than for larger databases where this variance is expected to decrease significantly. On the other hand, it is likely that, for some queries, such a small database contained few or no compositions similar to the input partial composition. Furthermore, with a BP of 10% and $N = 100$, the structural algorithms only consider the 10 most similar compositions when computing component scores. Therefore, they are unable to recommend 3 components for some queries or they recommend some components with very low absolute scores, which decreases their accuracy.

Coverage results for the same range of composition database sizes are shown in figure 8.10. For composition databases of 100 and 750 pipes, total catalog coverage of all four structural algorithms increases by between 15 and 25 percent from the coverage at $N = 2500$, but most of these recommendations are unsuccessful as successful coverage increases only marginally

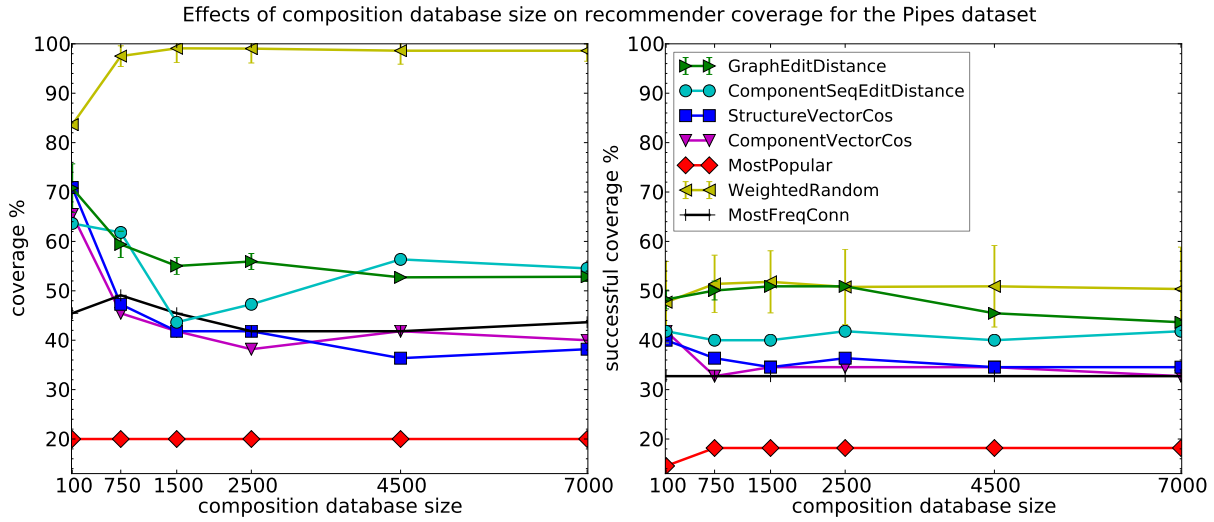


Figure 8.10: Changes in recommender coverage for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the Pipes dataset.

in this range of database sizes. Both the increase in catalog coverage and the relative stability of successful coverage can be explained through the connection of similarity filtering based on a relative, percentage-based BP parameter with the size of the composition database. As the database size decreases with a constant rate of similarity filtering, fewer and fewer of the most similar compositions are involved in component scores computation. Therefore, as seen in subsection 8.2.1, recommendations become more focused and catalog coverage increases. However, component scores computed based on so few compositions chosen from an already small sample of the whole composition database are less reliable and lead to bad recommendations for some queries.

The feature vector algorithms lose around 2% and *GraphEditDistance* loses 6% of successful coverage as N is increased from 2500 to 7000. Both these effects are minor, corresponding to one and three fewer recommended modules, respectively, but can also be explained by the effect of similarity filtering. In other words, increasing N with a constant BP is similar to increasing BP with constant N, which has already been shown to reduce coverage.

It can be concluded that, for the Pipes dataset, the size of the composition database in itself does not significantly affect recommender coverage.

Recommender response time is defined as the time difference between receiving a partial composition as input and producing a list of recommended components. Average response times for different values of N are shown on the left in figure 8.11, while maximum response times are shown on the right.

All four structural algorithms show linear growth of response time with N caused by similar-

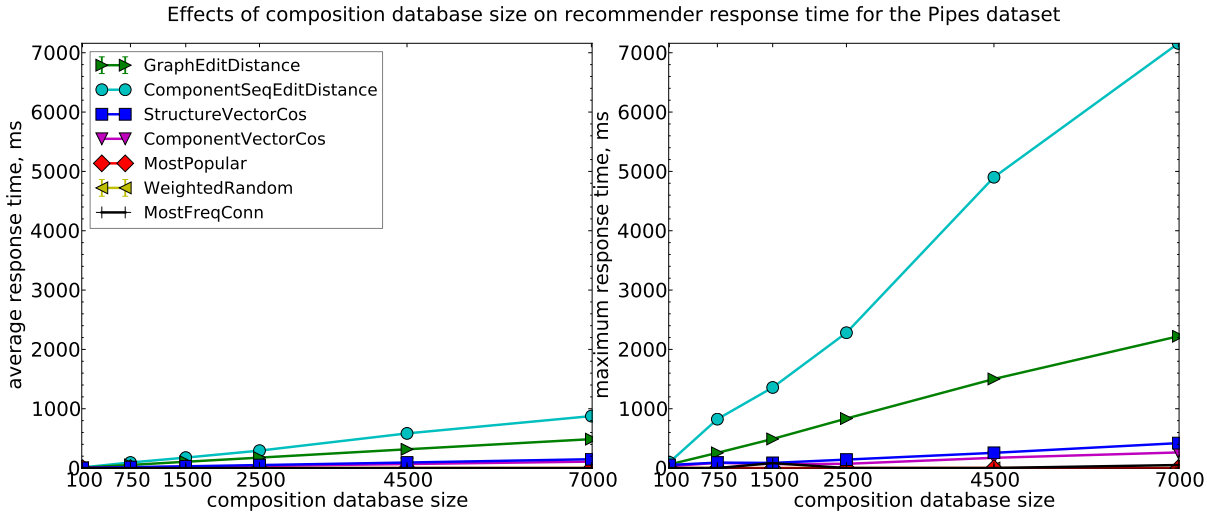


Figure 8.11: Average and maximum response time in milliseconds as a function of N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the Pipes dataset.

ity evaluation over the whole composition database. The slope of the graph is determined by the computational complexity of each algorithm in evaluating similarity of each P-Q pair. The response time grows most quickly with N for the *ComponentSeqEditDistance* algorithm, which, on average, requires almost one second to respond to a query for the largest tested database size of $N = 7000$. The average response time of the *GraphEditDistance* algorithm increases with about half the slope of *ComponentSeqEditDistance*, reaching almost half a second for the largest composition database. Both feature vector algorithms achieve a worst case average response time in the 200ms range. Response times of the baseline algorithms are independent of the composition database size and all three baseline recommenders respond practically instantly.

While average response time is important for predicting hardware requirements for hosting a recommender system, maximum response time is more important in terms of user interaction. While the maximum response time of both feature vector algorithms (263ms for *ComponentVectorCos* and 420ms for *StructureVectorCos*) increases only by about a factor of two compared to average response time, the difference between average and maximum response time is significantly larger for the more complex edit distance algorithms. The *GraphEditDistance* algorithm evaluated the worst case query for 2.2 seconds, while *ComponentSeqEditDistance* required as much as 7 seconds.

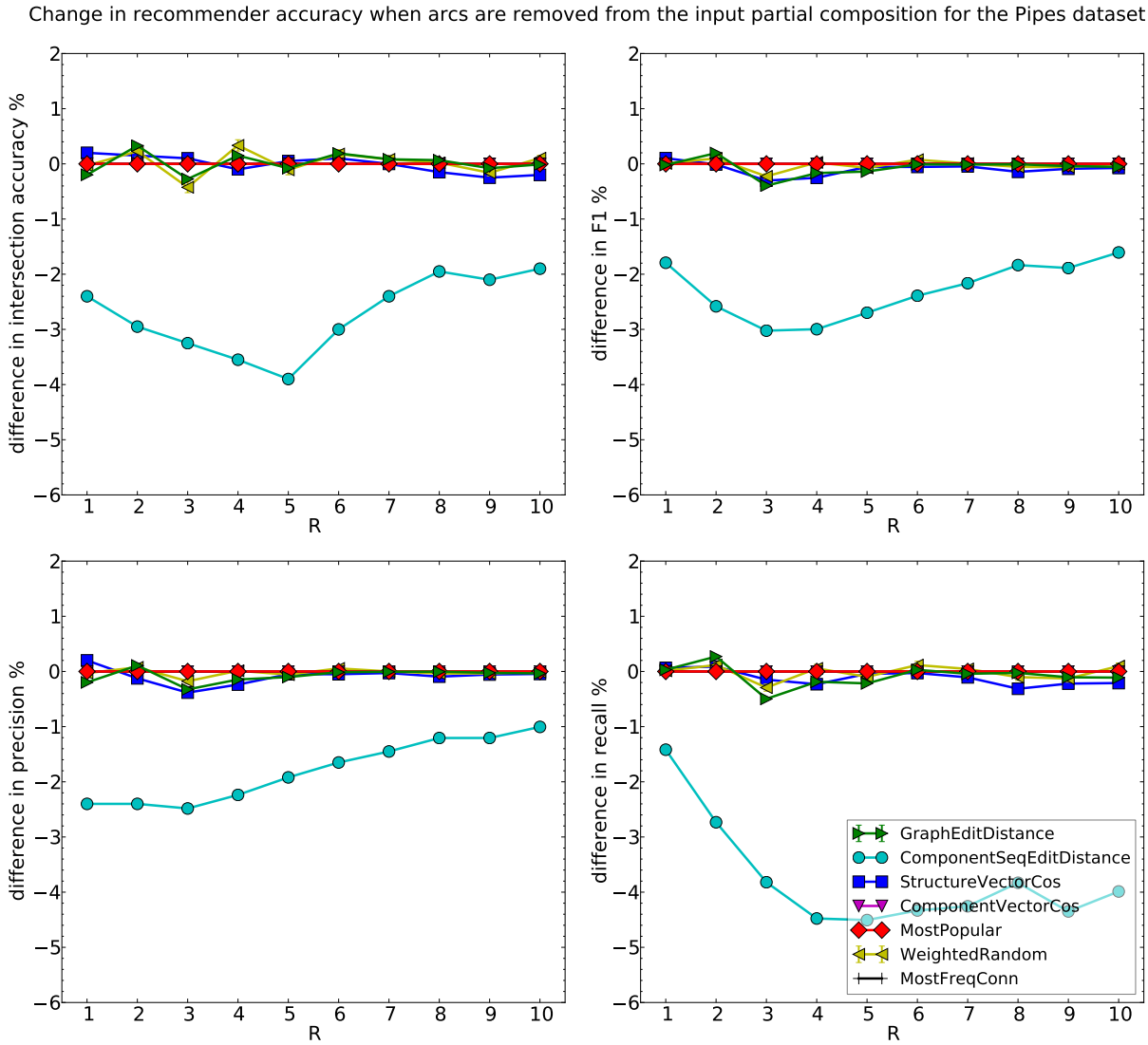


Figure 8.12: Change in recommender accuracy when arcs are removed from the input partial composition P for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the Pipes dataset.

8.2.4 Effects of Arcs in the Input Partial Composition

Based on the nearly identical results of the two feature vector algorithms discussed in the previous subsections, it can be concluded that arcs impact recommender quality much less than components. Specifically, the *StructureVectorCos* algorithm, which includes arc information in the feature vector representation of compositions, outperforms the *ComponentVectorCos* algorithm, which works with feature vectors containing only component frequencies, only slightly in accuracy and coverage—in most experiments, the difference is within one percent and not significant.

In this subsection, the goal is to examine this issue in more detail. While the difference between the two feature vector algorithms is universal in that they are based on different rep-

resentations one of which uses arcs and one of which does not, an important issue in component recommendation is the impact of arcs in input partial compositions. This issue arises because different users compose applications in different ways—some users might add components when all the connections between previously added components have been defined, while others might first get all the required components to the composition workspace and only then start connecting the components. Both these options are only the ends of a spectrum of possibilities in composite application development.

Due to this fact, one of the design requirements for a component recommender is that the recommender should provide useful recommendations even when no connections between components have been established. All of the previously discussed results were based on experiments where a snapshot was the subgraph induced by a set of vertices in a composition graph, i.e. all the possible arcs in the completed composition were present in the snapshot. In this subsection, the algorithms are evaluated in the opposite extreme, where no arcs are present in the input partial composition. Note that arcs were still present in database compositions and all of the recommender algorithms operate exactly as described in chapter 6, i.e. they don't adapt to the lack of arcs specifically.

For all of these experiments, the exact same set of 2000 snapshots and their corresponding sets of useful recommendations were used, except that all arcs were removed from the snapshots. As this change has a negligible effect on recommender response time, only accuracy and coverage are analyzed.

The effect of arcs in the input partial composition on recommender accuracy is shown in figure 8.12. The graphs show the change of intersection accuracy, the F_1 score, precision, and recall, i.e. the difference between a particular measure of accuracy when arcs are included in the input partial composition and when they are removed. This change is graphed against the number of recommendations per query R, but the same qualitative effect can be observed in BP- and N-graphs as well.

Accuracy of *ComponentSeqEditDistance* decreases by between 1 and 5 percent, depending on R. This decrease can be explained by the fact that when no arcs are present in the input partial composition, the component sequence representation used in the *ComponentSeqEditDistance* algorithm essentially contains the components of the composition in random order as any order becomes equally valid for a generalized topological order—in fact, every order of components is a topological order in the strict sense, without any generalization. On the other

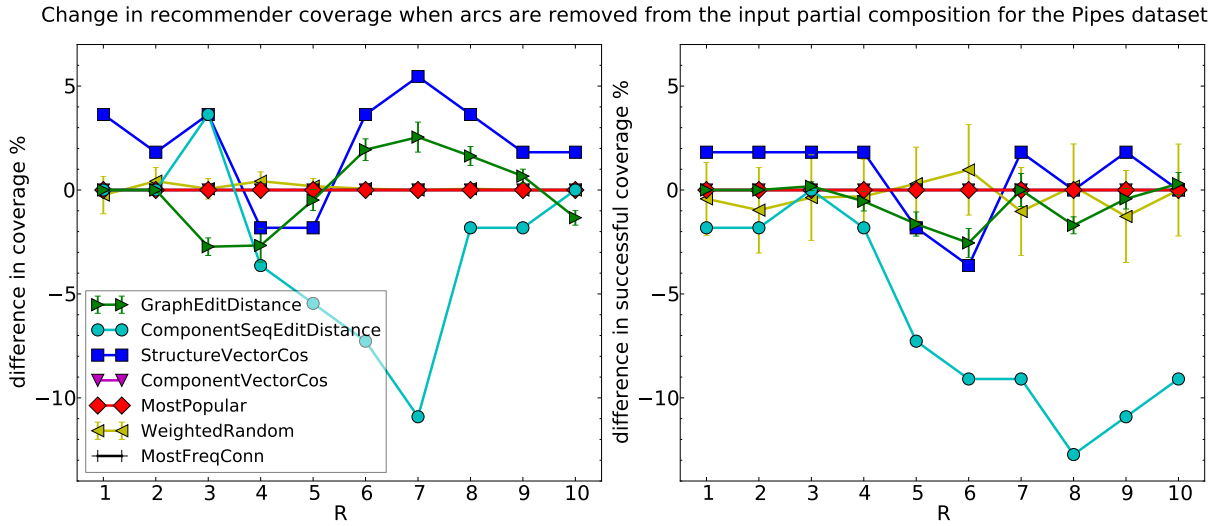


Figure 8.13: Change in recommender coverage when arcs are removed from the input partial composition P for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the Pipes dataset.

hand, the compositions in the composition database still contain arcs so their component sequences list components in generalized topological order. This mismatch causes the algorithm to occasionally miss useful recommendations.

All algorithms except *ComponentSeqEditDistance* achieve almost exactly the same accuracy with and without arcs in the input partial composition as their graphs are grouped around the 0% change value. Specifically, the *ComponentVectorCos* algorithm and the baseline algorithms *MostPopular* and *MostFreqConn*, which are completely independent of arcs in the input partial composition, have completely horizontal graphs indicating no change in accuracy as they produce exactly the same output in both experiments. The accuracy of *StructureVectorCos*, *GraphEditDistance* and *WeightedRandom* fluctuates marginally, in the order of a half of a percent over all measures.

While the more complex feature vector algorithm *StructureVectorCos* does include arc information, their absence in the input partial composition mostly scales the similarity scores of compositions in the composition database, so a lack of significant effect is not surprising.

On the other hand, the results for *GraphEditDistance* are somewhat unexpected in that it too achieves the same accuracy regardless of arcs in the input partial composition. This implies that, for the Pipes dataset, exploring multiple matching options provides no real benefit.

Removing arcs from the input partial composition has a similarly minor effect on coverage, as shown in figure 8.13. Only the *ComponentSeqEditDistance* algorithm shows a significant decrease in coverage and successful coverage of up to 13%, but only when more than four

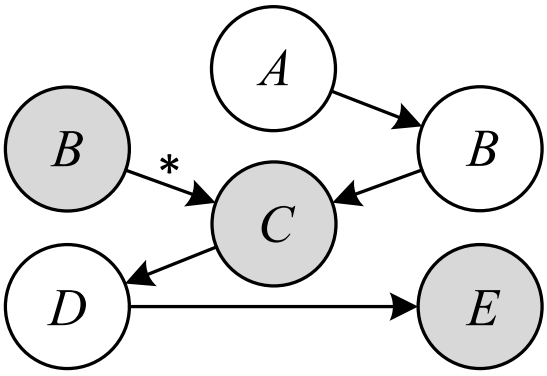


Figure 8.14: A snapshot of a composition. The vertices in the snapshot have a gray fill and the only arc in the snapshot is marked with an asterisk. This figure is a repeat of figure 8.1 to illustrate the *adjacent-useful* definition of useful recommendations.

modules are recommended per query.

The *StructureVectorCos* and *GraphEditDistance* algorithms show slight variation in coverage, corresponding to a difference of between 1 and 4 modules, which is not significant.

8.2.5 Evaluation Results Under the *Adjacent-Useful* Definition of Useful Recommendations

In all the previously discussed experiments, recommendation quality was evaluated based on the broadest definition of usefulness—a component was considered a useful recommendation for a snapshot given to the recommender as the input partial composition if and only if it was used in the original completed composition that the snapshot was generated from. In this subsection, a stricter definition of usefulness is considered. Specifically, a component is considered a useful recommendation for a snapshot if and only if it is directly connected to a component in the snapshot in the completed composition the snapshot was generated from.

For convenience, figure 8.1 is repeated here as figure 8.14. As explained previously in section 8.1, under the usual definition of useful components, components A and D would be considered useful[1]. In this subsection, under the *adjacent-useful* definition of useful recommendations, component A would not be considered a useful recommendation as it is not directly connected to any component in the snapshot.

The rationale for this definition is that recommender users might not be able to identify that a recommended component is useful if it could only be useful after several intermediate components are added to the composition. On the other hand, by this alternate definition of

[1]Component B would not be considered useful because it is already used in the snapshot itself, i.e. the user is aware of it and knows how to use it.

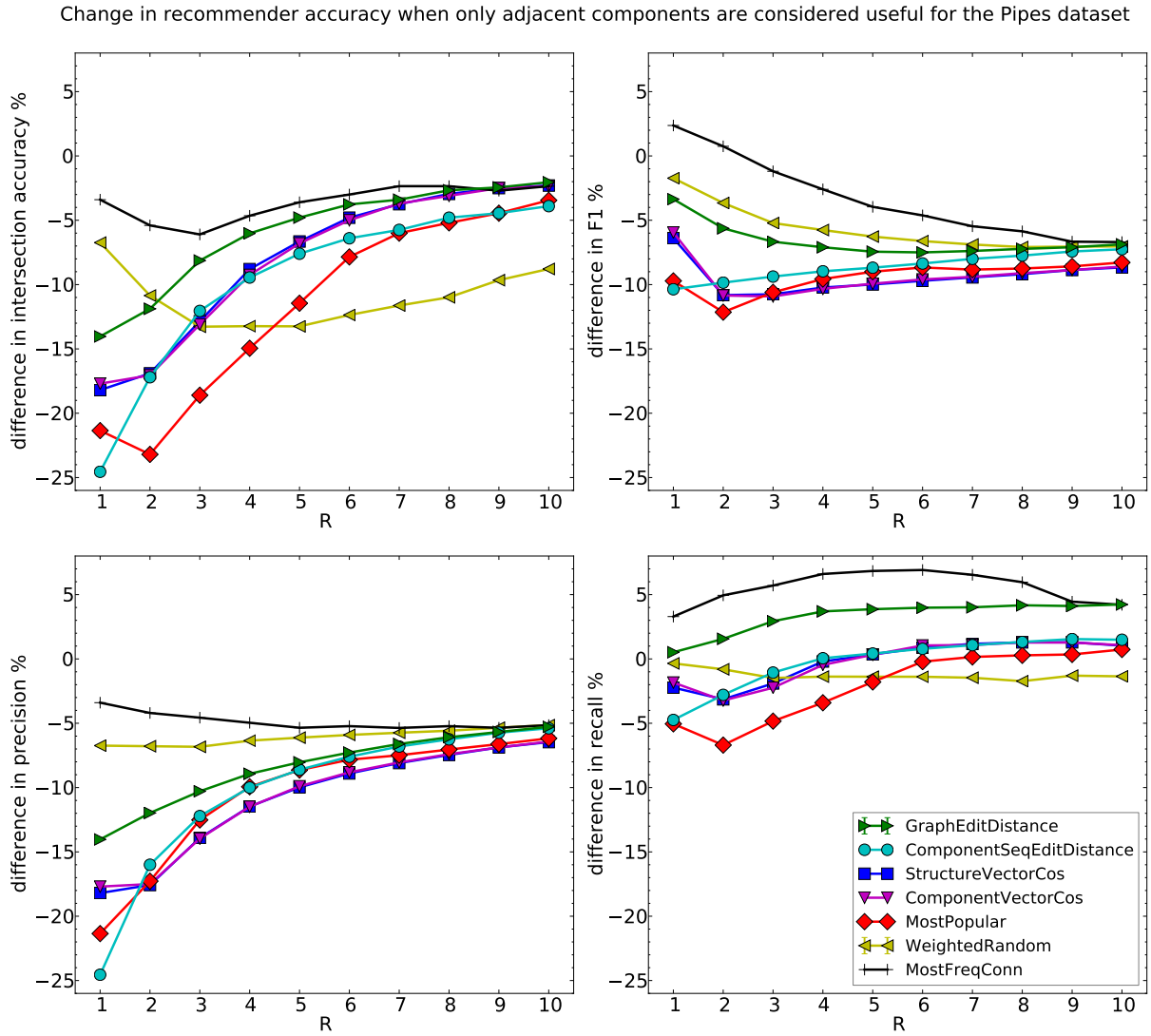


Figure 8.15: Change in recommender accuracy when only adjacent components are considered useful recommendations for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the Pipes dataset.

useful components, the user should identify only those components that are on the frontier between the partial composition and the eventual goal of the composition process.

As before, the exact same set of snapshots was used for these experiments, but the set of useful recommendations was changed as described above. With this different definition of usefulness, the size of the union of the useful recommendation sets for all 2000 snapshots only decreases from 54 to 53 modules, i.e. only one module was never directly connected to a snapshot. Furthermore, the average size of the set of useful recommendations for a snapshot decreases from 2.76 to 1.89 components, and the sets are different for 884 of the 2000 queries.

The effect of this change on recommender accuracy is shown in figure 8.15. Analogously as in the previous section, the graphs show the difference between accuracy with the *adjacent-*

useful definition of useful components and accuracy results discussed earlier, which were obtained with the broader definition of usefulness.

Note that when using the *adjacent-useful* definition of useful recommendations, the set of useful recommendations for every snapshot is at most as big as with the broader definition, but sometimes smaller. This causes a general decrease of intersection accuracy and precision across all structural and baseline algorithms, especially for small values of R. The most extreme decrease of precision is observed for the *ComponentSeqEditDistance* algorithm which loses 25% of precision when only one module is recommended. Of the structural algorithms, *GraphEditDistance* is least affected in precision by this change.

MostFreqConn is the least effected baseline algorithm, decreasing in precision by only 3% even when $R = 1$. However, its precision is low to begin with. The *MostPopular* algorithm behaves similarly to the structural algorithms, losing slightly more precision at almost all data points. Finally, the precision of *WeightedRandom* decreases consistently by around 7% because there are fewer modules to guess, on average.

The more pronounced decrease in precision when only a few components are recommended is caused by the fact that the ubiquitous modules in *Yahoo Pipes* development are more often considered bad recommendations. For example, when a snapshot does not contain and is also not directly connected to the *output* module, a recommendation of that module will be inaccurate.

Interestingly, the recall of both feature vector algorithms and *ComponentSeqEditDistance* also decreases slightly when a small number of components is recommended per query, but levels off and even increases slightly when at least four components are recommended. On the other hand, the recall of the *GraphEditDistance* algorithm increases over all values of R, by up to 4%. A similar and even more pronounced increase is seen for *MostFreqConn*, while the recall of both remaining baseline algorithms decreases by up to 5%.

The cumulative effect of both the changes to precision and recall can be observed through the F_1 score in the upper right graph in the figure. The F_1 score of all algorithms decreases with the restricted definition of useful recommendations. The least affected algorithms are *GraphEditDistance* and the baseline *MostFreqConn* and *WeightedRandom* algorithms. This is expected as *GraphEditDistance* recommends those components that are often directly connected to components in the snapshot, and *MostFreqConn* bases its recommendations exactly on the frequency of such connections. The *ComponentSeqEditDistance* algorithm employs the

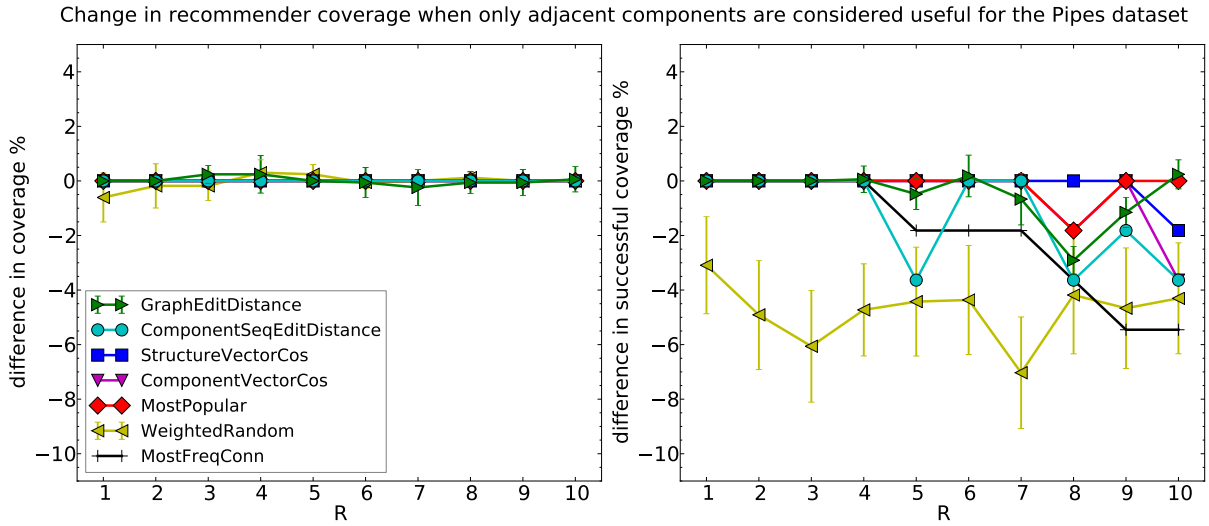


Figure 8.16: Change in recommender coverage when only adjacent components are considered useful recommendations for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the Pipes dataset.

same method of scoring components as *GraphEditDistance*, but it matches fewer components due to a fixed order in the underlying sequential representation.

The effect of this change in the definition of useful recommendations on coverage is shown in figure 8.16. Obviously, there is no effect on catalog coverage as the input to the recommenders is exactly the same under both definitions of useful recommendations. Therefore, the differences in successful coverage are caused by recommendations of components that are actually used to finish an input partial composition P, but they are not directly connected to components of P.

The successful coverage of several algorithms decreases by up to 6 percent when more than four modules are recommended per query, which corresponds to 3 modules and is not significant. Additionally, *WeightedRandom* loses several modules in successful coverage at all data points, with high variance.

8.3 Snapshot Evaluation Results on the Synthetic Dataset

In the snapshot set, only 490 of the total 762 components are used at least once. Similarly to the Pipes snapshot set, the average snapshot has around 4.3 components. The snapshots are denser than the Pipes snapshots, but still sparse with 3.9 arcs per snapshot, on average. In spite of this sparsity, 630 of the 2000 snapshots contain at least one cycle, while the remaining

snapshots are DAGs[2].

The most frequent component appears only 148 times in the snapshot set, which is more than five times less than the frequency of the *fetch* module in the Pipes snapshot set. Overall, the component frequency distribution is significantly flatter than that of the whole dataset, with 60% and 88% of the total component frequency covered by the 20% and the 50% of the most popular components, respectively. For comparison, these percentages are 71% and 95% in the entire synthetic dataset of 7600 compositions.

The remainder of the section is organized in an analogous way to section 8.2. In subsection 8.3.1, the effects of similarity filtering for component score computation are analyzed. Then, in subsection 8.3.2, the effect of the number of recommended components per query is evaluated. The change of recommender quality with different composition database sizes is presented in subsection 8.3.3. Subsection 8.3.4 analyzes how recommender quality is affected by the potential lack of connections in the input partial composition, through repeating the experiments with arcs removed from the snapshots. Finally, the section is concluded with the evaluation of the *adjacent-useful* definition of useful recommendations in subsection 8.3.5.

8.3.1 Effects of Similarity Filtering in Computing Component Scores

In terms of accuracy, similarity filtering only marginally affects the *ComponentSeqEditDistance* algorithm, while all the other algorithms are practically unaffected, as shown in figure 8.17. The accuracy of *ComponentSeqEditDistance* decreases by up to 5% on both sides of the $BP = 25\%$ value across all four accuracy measures, but the only significant decrease happens between BP values of 10 and 5 percent. As explained in subsection 8.2.1 of the Pipes snapshot evaluation results, this decrease is likely caused by a fraction of the snapshots for which fewer than R components achieve a nonzero score. Unlike the Pipes dataset, this effect is not observed in the *GraphEditDistance* algorithm.

The same dichotomy of algorithms is observed in the effect of similarity filtering on coverage, as shown in figure 8.18. While the catalog coverage of the *ComponentSeqEditDistance* algorithm increases significantly by more than 10% from no similarity filtering to BP of 5%, the large majority of this increase is in unsuccessful recommendations, as successful coverage only increases by slightly more than 3% in that range. The coverage of all the other algorithms is unaffected.

[2] A DAG is a directed acyclic graph.

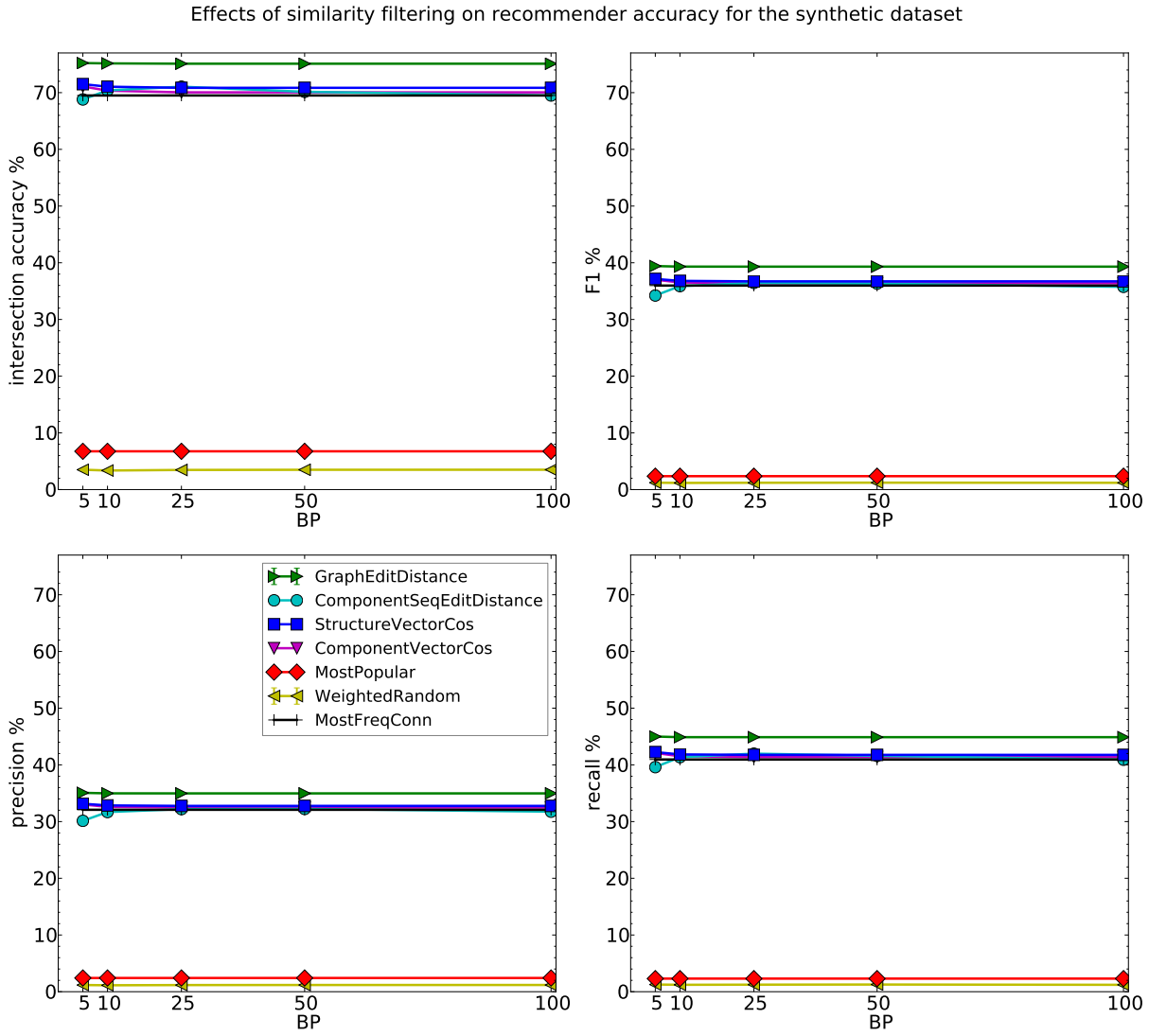


Figure 8.17: Changes in recommender accuracy for BP between 5 and 100 percent with $R = 3$ and $N = 2500$ for the synthetic dataset.

The coverage curves with no similarity filtering are shown in figure 8.19. Two interesting properties that are in contrast with the Pipes dataset can be observed. First, the coverage curves of all algorithms except *MostPopular* are flatter, i.e. many components get recommended frequently, and the most recommended component gets recommended around 150 times. This is the effect of a generally flatter distribution of component frequency in the dataset and the lack of ubiquitous components. As can be expected, the *WeightedRandom* algorithm achieves the flattest coverage curve, followed closely by *GraphEditDistance*.

The second interesting distinction from the Pipes coverage curves is the *MostPopular* algorithm. Throughout the 2000 queries, the *MostPopular* algorithm recommends only five distinct components, and the three most frequent components are recommended for 1900 or more queries. The coverage curve of *MostPopular* is therefore practically a vertical line, and the

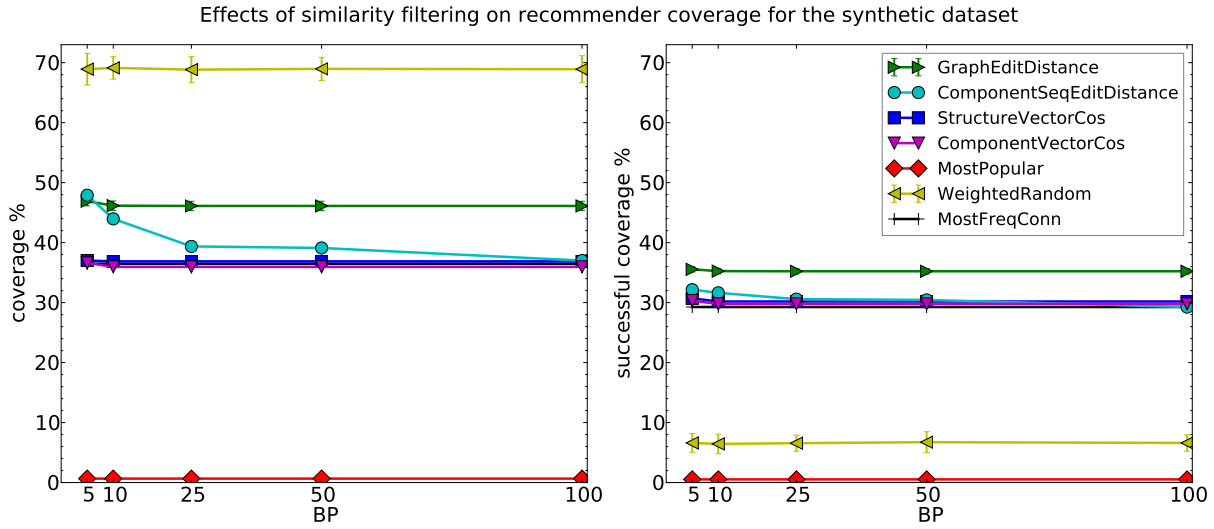


Figure 8.18: Changes in recommender coverage for BP between 5 and 100 percent with $R = 3$ and $N = 2500$ for the synthetic dataset.

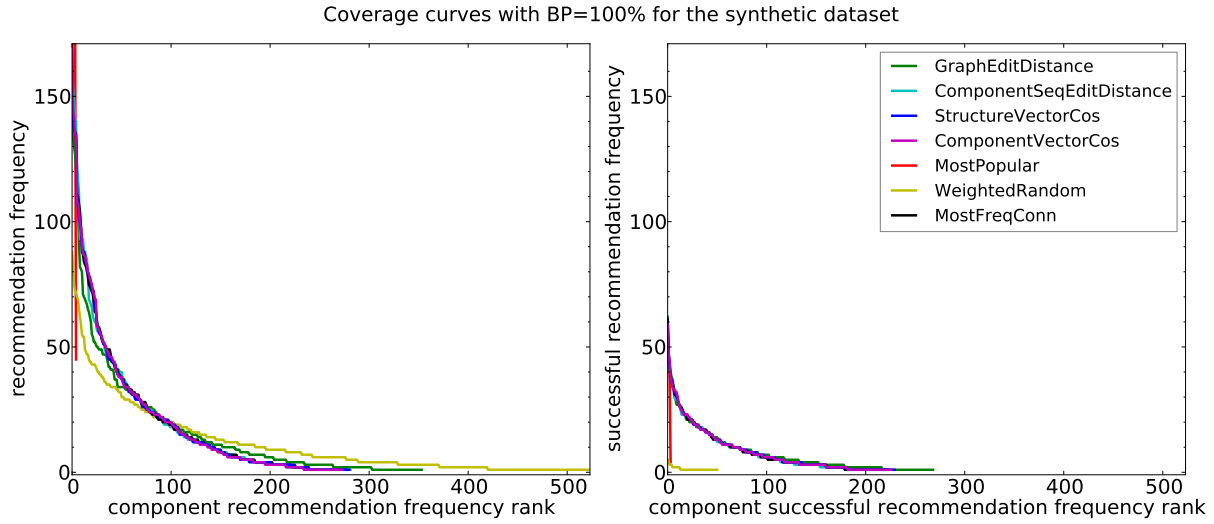


Figure 8.19: Coverage curves for $BP = 100\%$ for the synthetic dataset with $R = 3$ and $N = 2500$.

curve is cut in figure 8.19 around the frequency of 170 as all the other coverage curves would otherwise be melded on the x-axis.

The coverage curves for successful coverage meet around the value 60 on the y-axis, and are very similar for all four structural algorithms and *MostFreqConn*, with *GraphEditDistance* showing the longest tail by around 40 components and the most recommended components at every frequency beyond 80. The area under the curves for *MostPopular* and *WeightedRandom* is vanishingly small as their accuracy on the synthetic algorithm is below 3%.

For completeness, the coverage curves with aggressive similarity filtering at $BP = 10\%$ are shown in figure 8.20. This figure shows that coverage is practically unaffected both quantitatively and qualitatively by similarity filtering, with marginal effects on the *ComponentSeqEdit-*

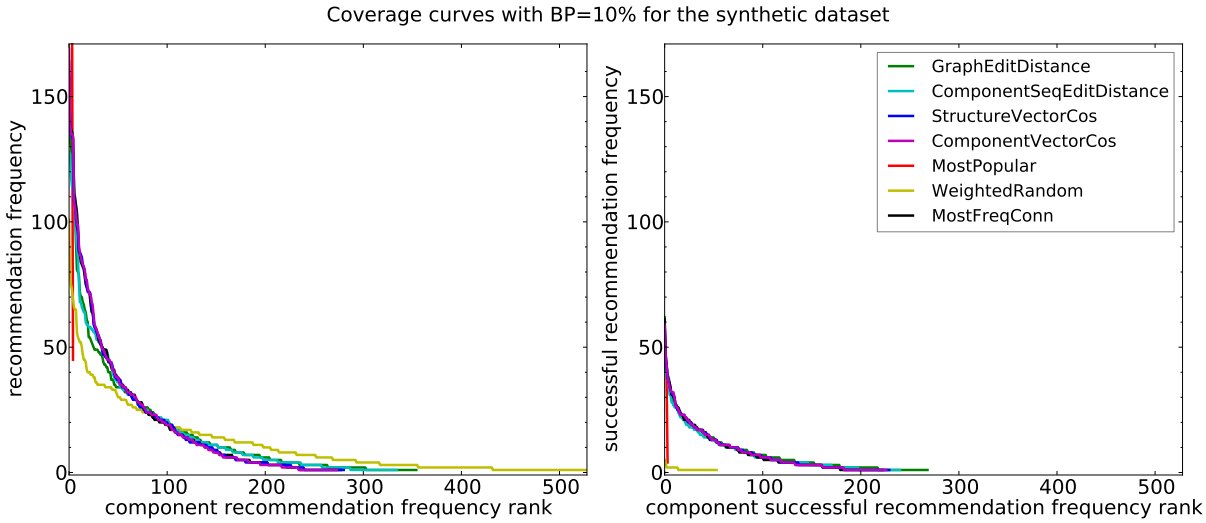


Figure 8.20: Coverage curves for $BP = 10\%$ for the synthetic dataset with $R = 3$ and $N = 2500$.

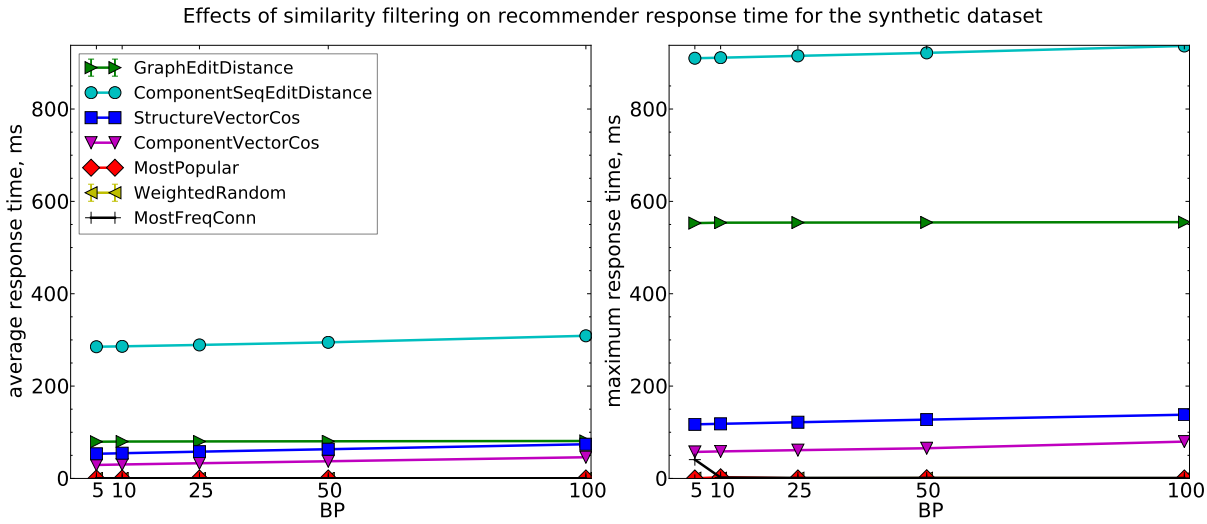


Figure 8.21: Changes in recommender response time for BP between 5 and 100 percent with $R = 3$ and $N = 2500$ for the synthetic dataset.

Distance algorithm, as discussed above.

The effect of similarity filtering on recommender response time is shown in figure 8.21. Analogous to the Pipes dataset, it can be observed that the response time of all recommender algorithms is dominated by the process of similarity evaluation, and that computing component scores accounts for only a small fraction of the time.

While the general effect of similarity filtering for the synthetic dataset is minor, the remaining evaluations in the following subsections were performed with BP set to 10%, which provides some small benefit to the *ComponentSeqEditDistance* algorithm in coverage, and is also consistent with the Pipes evaluation.

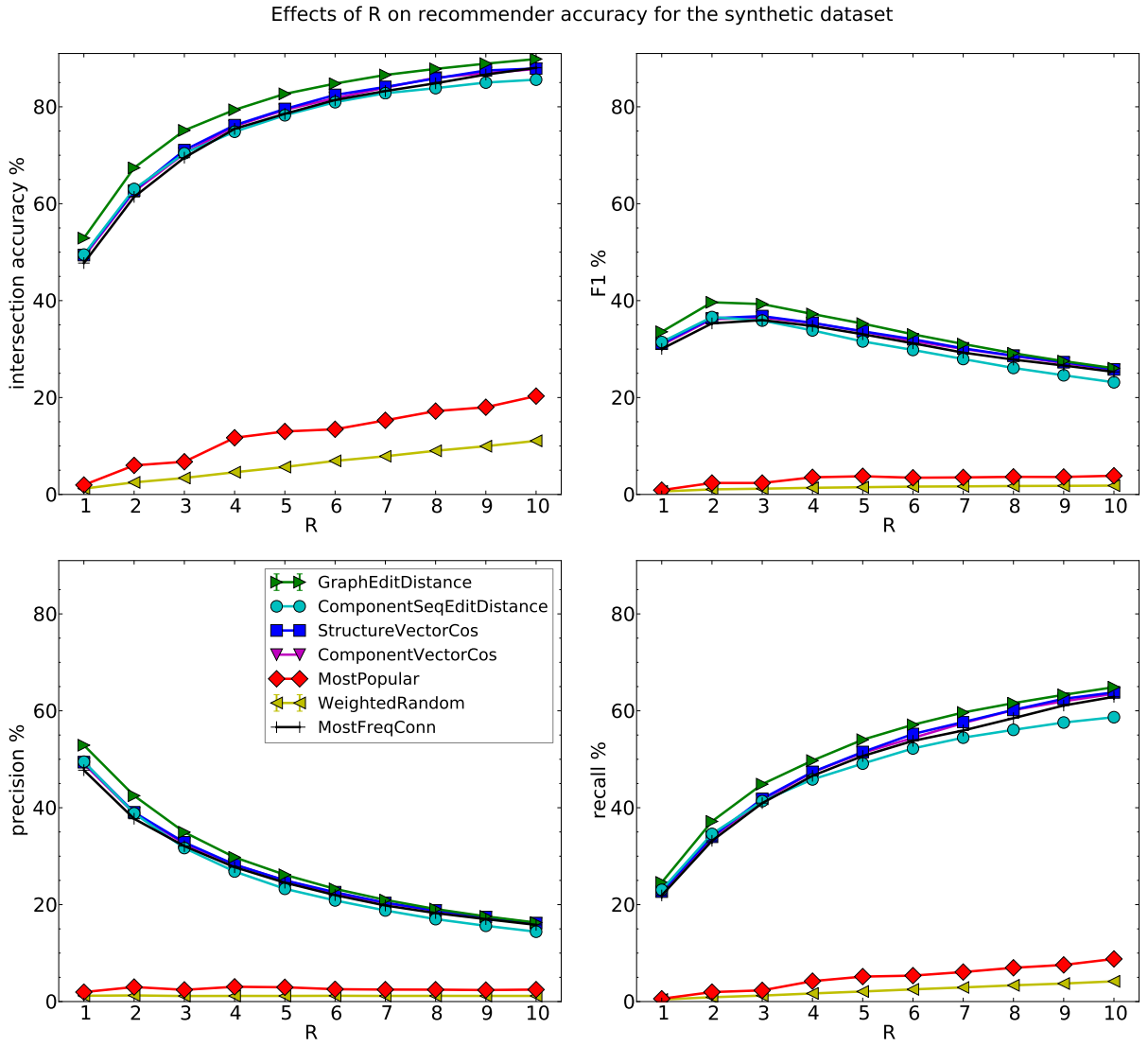


Figure 8.22: Changes in recommender accuracy for R between 1 and 10 with $BP = 10\%$ and $N = 2500$ for the synthetic dataset.

8.3.2 Effects of the Number of Recommended Components Per Query R

Changing the number of recommended components per query has a similar qualitative effect as in the Pipes dataset, as shown in figure 8.22. As more components are recommended, intersection accuracy and recall of all four structural algorithms increase, while precision decreases. The baseline algorithm *MostFreqConn* behaves very similarly to the structural algorithms. On the other hand, due to the larger component diversity, the *MostPopular* and *WeightedRandom* algorithms show practically constant precision below 3%, and their recall and F_1 score increase with R, but remain well under 10% even when 10 components are recommended per query. For all three structural algorithms, the maximum F_1 score is again achieved with R values of 2 and 3.

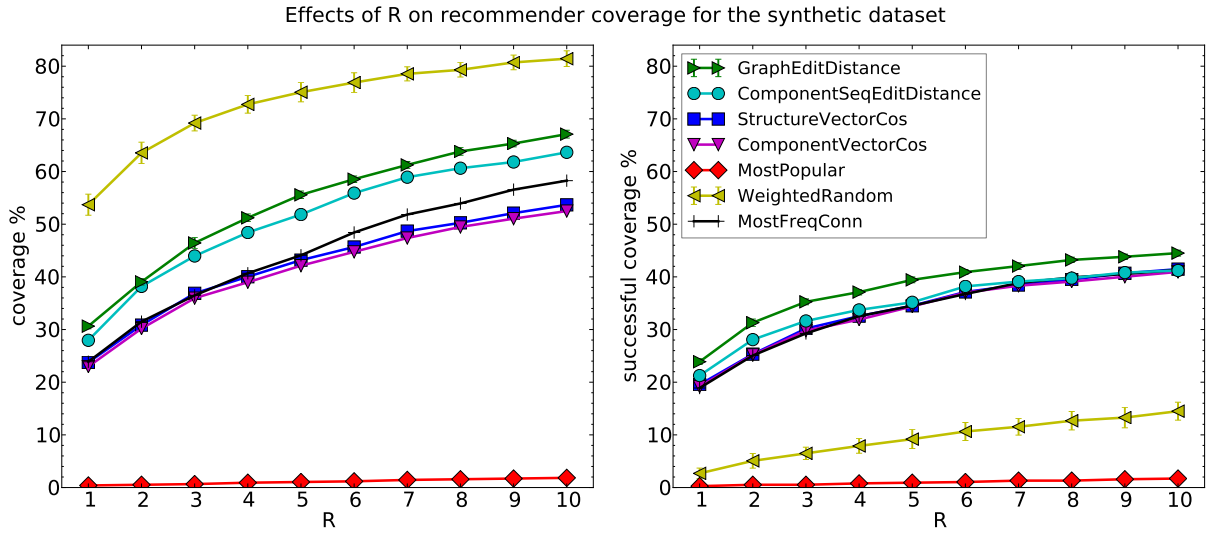


Figure 8.23: Changes in recommender coverage for R between 1 and 10 with $BP = 10\%$ and $N = 2500$ for the synthetic dataset.

Coverage results are shown in figure 8.23. Increasing R increases successful coverage only marginally beyond $R = 5$, and no algorithm is able to reach successful coverage of 45%. This is in contrast with the Pipes dataset where increasing R continually increases both total and successful coverage. It should be noted that coverage is always reported relative to the total number of components in the dataset, which is 762 in the synthetic dataset. However, a significantly smaller number of only 484 components are ever considered useful recommendations for the snapshot set used in the evaluations. In other words, an ideal recommender with 100% precision could only achieve up to 64% successful coverage.

GraphEditDistance achieves the highest coverage except for *WeightedRandom*, and highest successful coverage for all values of R. The *ComponentSeqEditDistance* algorithm recommends between 1 and 5 percent fewer components than *GraphEditDistance*, and both algorithms perform significantly better than the feature vector algorithms and *MostFreqConn*, especially for larger values of R. However, this advantage in catalog coverage appears to result mostly from inaccurate recommendations as the successful coverage of *ComponentSeqEditDistance* is aligned almost perfectly with that of the feature vector algorithms.

8.3.3 Effects of Composition Database Size N

The four accuracy measures for different sizes of the composition database are shown in figure 8.24. Qualitatively, all four accuracy measures are affected by variations in database size in the same way. The feature vector algorithms and the baseline *MostFreqConn* algorithm

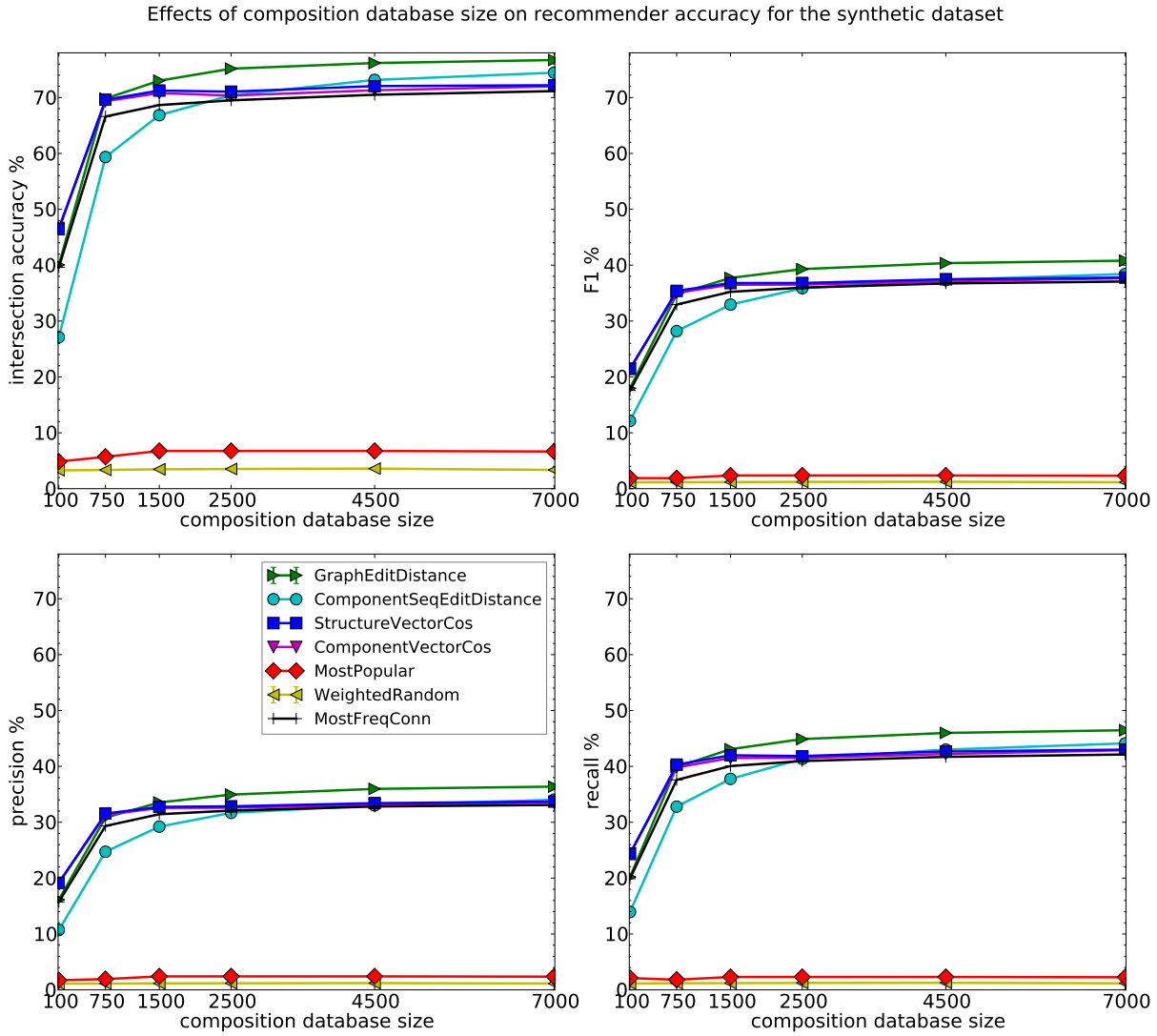


Figure 8.24: Changes in recommender accuracy for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the synthetic dataset.

achieve only a slight increase in accuracy for N larger than 750. On the other hand, the accuracy of *GraphEditDistance* and *ComponentSeqEditDistance* increases significantly by up to 10% as the number of compositions in the database is increased from 750 to 7000. Both *WeightedRandom* and *MostPopular* perform poorly for all database sizes and achieve their optimal accuracy with a database of 1500 compositions.

Changes in recommender coverage for different composition database sizes are shown in figure 8.25. On the Pipes dataset, a decrease in coverage with larger databases was observed for all algorithms except *ComponentSeqEditDistance*. This decrease was explained by the indirect effect of similarity filtering, i.e. through the fact that more and more compositions are considered when component scores are computed as the size of the database increases with a constant BP value. However, on the synthetic dataset, coverage is practically unaffected by similarity

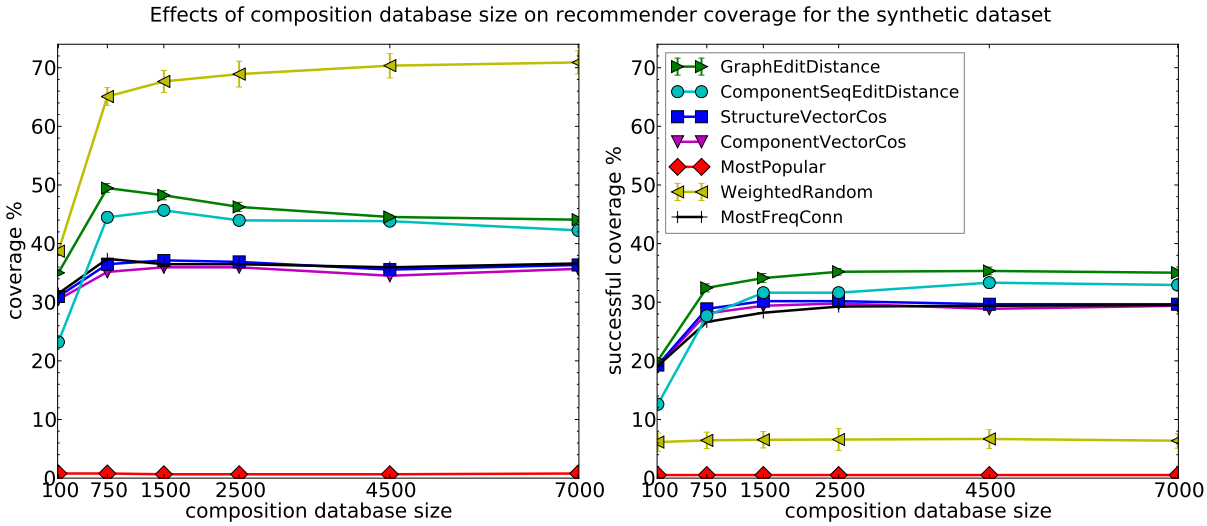


Figure 8.25: Changes in recommender coverage for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the synthetic dataset.

filtering and this effect is not present when changing database sizes.

All the algorithms achieved the worst catalog and successful coverage for the smallest database of only 100 compositions. This is to be expected on a dataset with many components as such a small database doesn't even contain many of the components that are required to complete some of the snapshots.

For databases of 750 compositions and larger, catalog coverage of the edit distance algorithms decreases by up to 5% as the database size is increased to the maximum of 7000 compositions, while it stays nearly constant for the feature vector algorithms and all baseline algorithms except *WeightedRandom*. Looking at the successful coverage graph, it is clear that the edit distance algorithms achieve a higher catalog coverage for smaller databases through inaccurate recommendations. This behavior is closely related to changes in accuracy discussed above. Specifically, the edit distance algorithms require a slightly larger composition database than the feature vector algorithms to reach their optimal accuracy. When fewer compositions are available for analysis, but the composition database is large enough to contain most components in some compositions, many inaccurate recommendations are made, but they cover a larger set of components.

Recommender average and maximum response times for the synthetic dataset are shown in figure 8.26. *ComponentVectorCos* and *StructureVectorCos* perform similarly in both average and maximum response time as for the Pipes dataset, as can be expected due to similar composition sizes between the two datasets. On the other hand, *ComponentSeqEditDistance* and

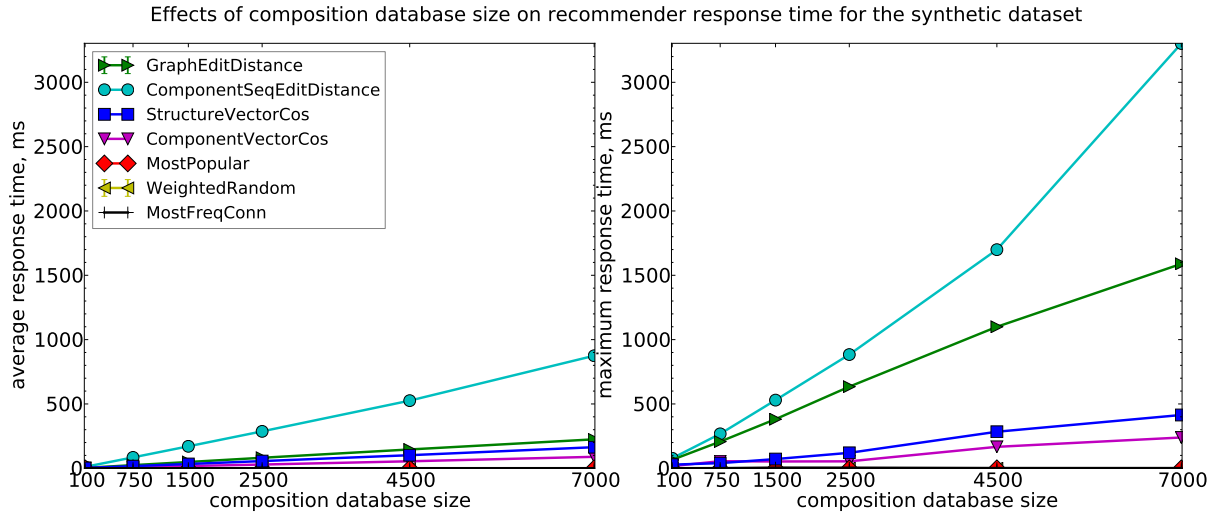


Figure 8.26: Average and maximum response time in milliseconds as a function of N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the synthetic dataset.

GraphEditDistance behave significantly better for the synthetic dataset. In the case of the *ComponentSeqEditDistance*, average response time is very close to the Pipes dataset at just under one second. However, maximum response time is almost halved at 3302 milliseconds as there are some pipes with an especially large number of modules.

The *GraphEditDistance* algorithm processed queries faster, both on average and in the worst case. The average response time on the synthetic dataset is below 350ms for all database sizes and nearly matches the simpler feature vector algorithms. The worst response time recorded in the experiment was slightly below 1.6 seconds, which is still almost four times slower than the *StructureVectorCos* algorithm. This increase in performance of *GraphEditDistance* is due to the number of repeated matching components within compositions of the two datasets. While component repetition within a particular composition is only slightly less common in the synthetic dataset, the case where the same component is repeated in many compositions is much rarer than in the Pipes dataset. Therefore, for most queries, few possible vertex matchings exist to be examined by the algorithm and the limit L is reached rarely.

8.3.4 Effects of Arcs in the Input Partial Composition

The change in accuracy when arcs are removed from the snapshots is shown in figure 8.27. No significant effect can be observed as all changes across all values of R and all four accuracy measures are within one percent, i.e. all tested algorithms achieve nearly identical accuracy results with or without arcs in the input partial composition. This behavior, with one exception,

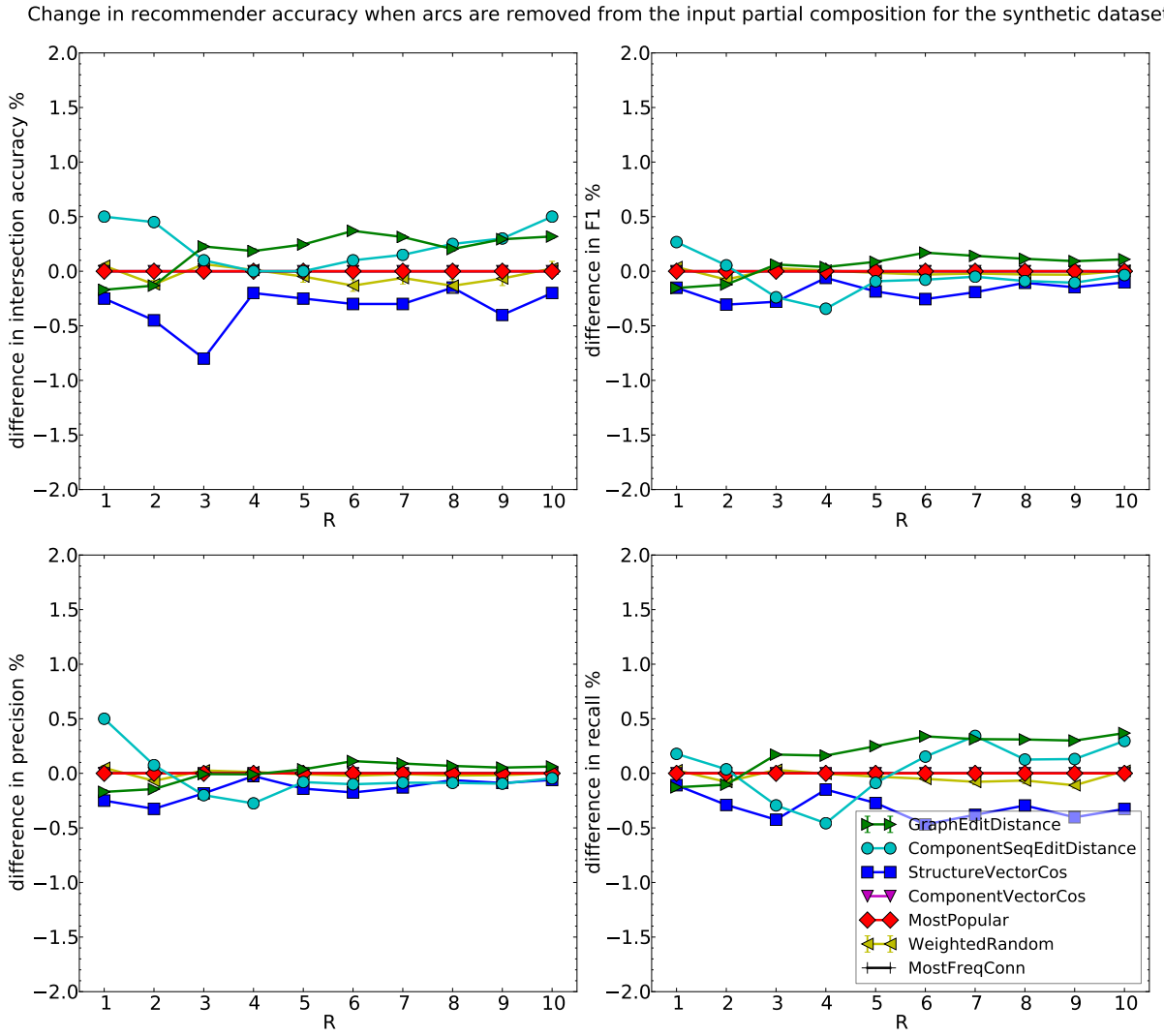


Figure 8.27: Change in recommender accuracy when arcs are removed from the input partial composition P for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the synthetic dataset.

was also seen on the Pipes dataset, and is explained in subsection 8.2.4.

The mentioned exception is the *ComponentSeqEditDistance* algorithm whose accuracy did decrease when arcs were removed on the Pipes dataset, and stays practically constant on the synthetic dataset. This implies that the generalized topological order isn't very representative of the structure of synthetic compositions, which is to be expected based on the generation process used to create the synthetic dataset. Specifically, when synthetic components were generated, no special rules were in place to enforce a similar order of components in a composition. Regularity was introduced into the dataset through component affinities which resulted in similar groups of components appearing together with similar local connectivity. Furthermore, most structurally complex graphs have a large number of possible (generalized) topological orderings, so even if two compositions are exactly the same, their component sequences might not be the same.

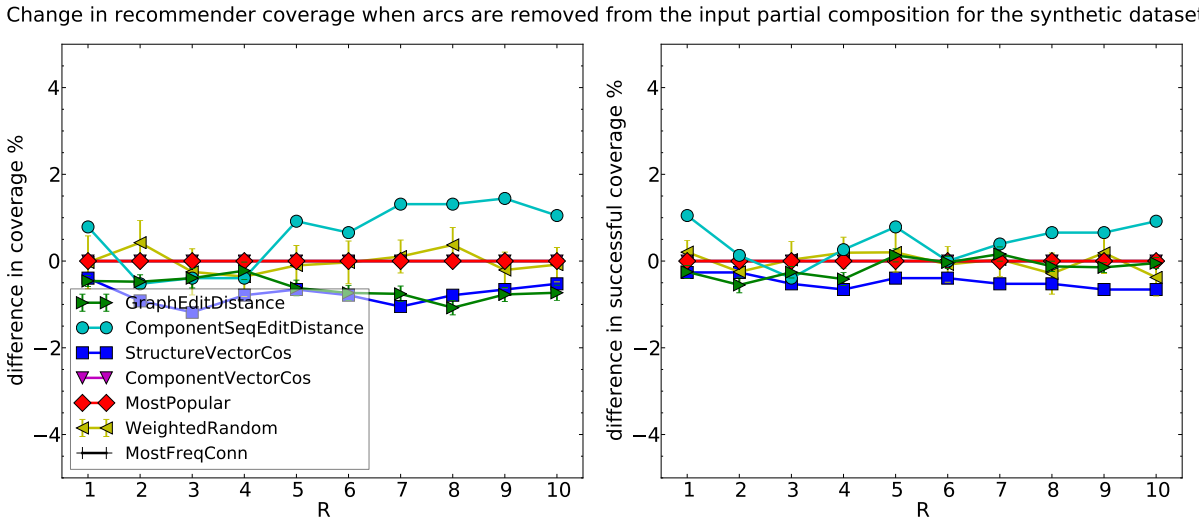


Figure 8.28: Change in recommender coverage when arcs are removed from the input partial composition P for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the synthetic dataset.

In contrast, *Yahoo Pipes* are by their nature very linear and a large majority of them can be represented with directed trees which typically allow fewer valid component sequences so the topological order does actually represent a useful structural property of the pipe.

An analogously small effect can be seen in coverage shown in figure 8.28. The results are largely unchanged, with insignificant fluctuations in several algorithms.

Based on these results, it can be concluded that arcs in the input partial composition have close to no effect on recommender quality for the synthetic dataset. This issue is discussed further in subsection 8.4.7.

8.3.5 Evaluation Results Under the *Adjacent-Useful* Definition of Useful Recommendations

When only those components directly connected to the snapshot are considered useful recommendations, the size of the union of the sets of useful recommendations for all snapshots decreases from 484 to 467 components and the average size of individual sets decreases from 2.89 to 2.33 components. In total, 630 of the 2000 queries are affected.

The change in all four measures of accuracy with this change in definition of useful compositions is shown in figure 8.29. Similar qualitative effects to the Pipes dataset can be observed, with three significant differences. First, the baseline algorithms *MostPopular* and *WeightedRandom* are almost unaffected in the absolute sense by the changed definition of usefulness as their precision is already below 3% on the synthetic dataset. Second, both precision and intersection

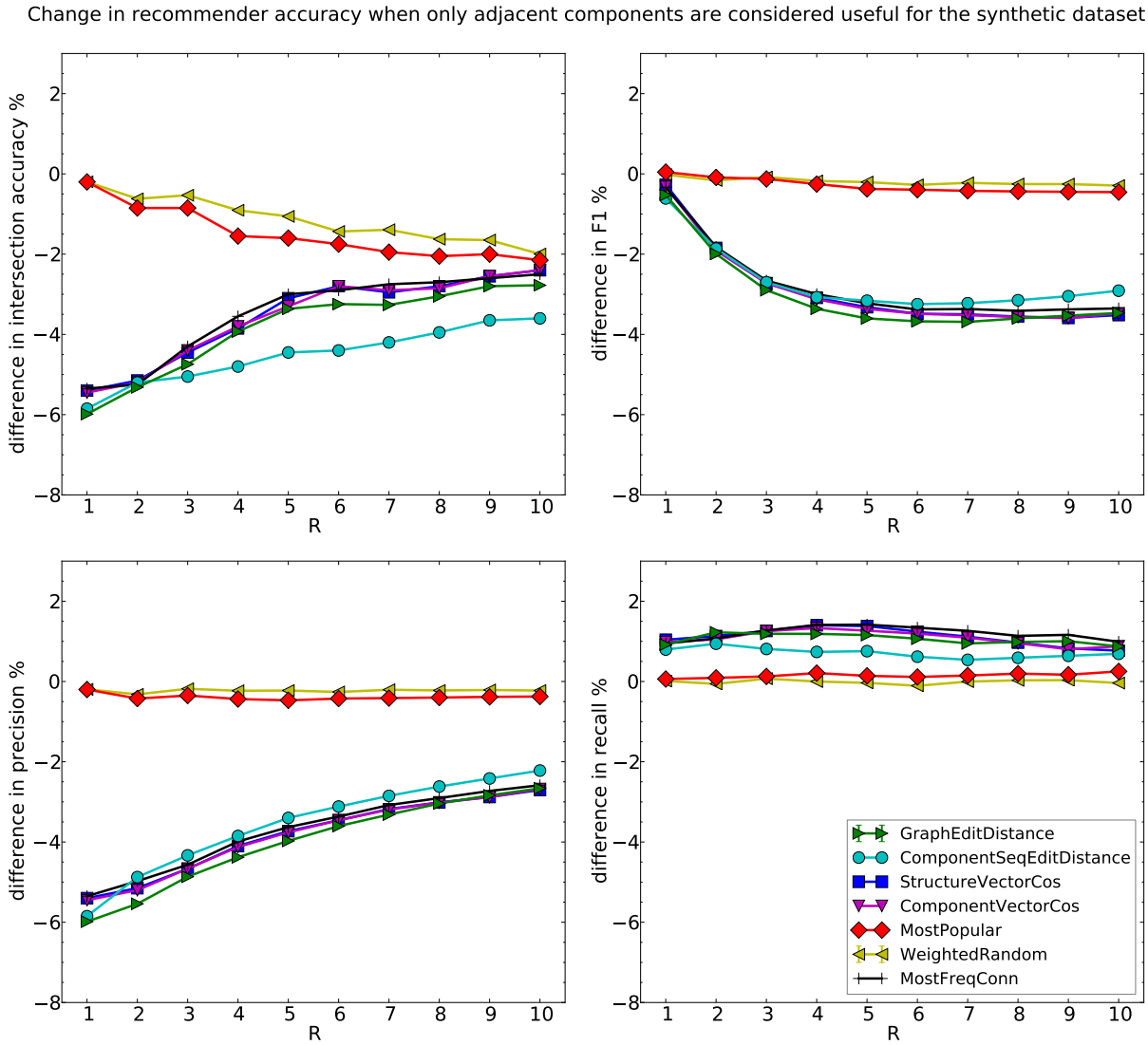


Figure 8.29: Change in recommender accuracy when only adjacent components are considered useful recommendations, for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the synthetic dataset.

accuracy decrease as on the Pipes dataset, but the decrease is more even across different values of R. Precision and intersection accuracy of all algorithms decreases by up to 6%, with larger decreases occurring when a small number of components is recommended per query. Furthermore, this decrease is significantly smaller than on the Pipes dataset, where all algorithms lost between 14 and 25 percent in precision for $R = 1$. The observed difference is the result of the existence of ubiquitous components in *Yahoo Pipes*, whose removal from the set of useful recommendations negatively impacts precision.

Finally, the third qualitative difference is that recall consistently increases by under 2% for all algorithms at all data points. The net effect of both changes to precision and recall is a decrease of the F_1 score by up to 4% for all four structural algorithms and the baseline

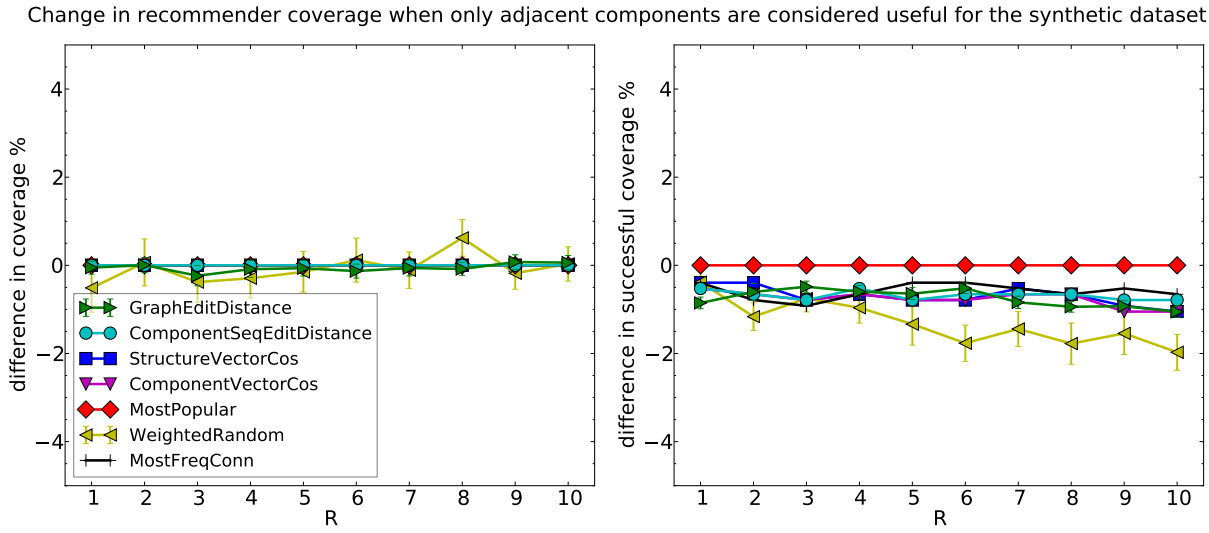


Figure 8.30: Change in recommender coverage when only adjacent components are considered useful recommendations, for values of R between 1 and 10, with $BP = 10\%$ and $N = 2500$ for the synthetic dataset.

MostFreqConn algorithm.

No significant change in coverage is observed, as shown in figure 8.30. Successful coverage of all algorithms except *MostPopular* decreases by up to 2%, which is a direct consequence of decreased precision that was observed above. This change doesn't occur on the Pipes dataset because of the total number of modules which is very low when compared to the number of components in the synthetic dataset.

Overall, it can be concluded that the distinction between the most general definition of usefulness where all components required to complete the partial composition are considered good recommendations and the *adjacent-useful* definition is minor, both in accuracy and successful coverage. This stability of results can be explained through two factors. First, due to the small size of compositions, on average, less than 20% of useful components are not actually connected to the snapshot, as evident from the decrease of average useful component set size from 2.89 to 2.33 components. Second, as the regularity in the synthetic dataset stems from component affinity which models the fact that certain components often appear together and are connected, components that are often directly connected to components of the input partial composition are more likely to get recommended than other components as they will appear more often in similar compositions.

8.4 Snapshot Evaluation Summary

In this section, the results presented in the previous two sections are summarized and discussed. First, in subsections 8.4.1 through 8.4.3, accuracy, coverage, and response times of all algorithms are analyzed in terms of their absolute value with standard parameter values $BP = 10\%$ and $R = 3$, and with a database of $N = 2500$ compositions. Then, the remainder of the section is organized in an analogous way to the previous two sections, with subsections 8.4.4 through 8.4.6 discussing the effects of similarity filtering in component scores computation, the number of component recommendations per query, and the size of the composition database. The section is concluded with subsections 8.4.7 and 8.4.8 that summarize the effects of arcs in the input partial composition and the *adjacent-useful* definition of useful recommendations on recommender quality, respectively.

8.4.1 Accuracy

All four structural recommender algorithms achieve similar accuracy over both datasets. On the Pipes dataset, intersection accuracy of around 92% is achieved by the feature vector algorithms, while *GraphEditDistance* and *ComponentSeqEditDistance* trail the feature vector algorithms by 1 and 2 percent, respectively. Such a high level of intersection accuracy can be attributed to the high impact of ubiquitous Pipes modules like *fetch*, *output*, and *urlbuilder* that are often considered correct recommendations as they are indeed required to complete a partial composition. This claim is supported by the results of the *MostPopular* baseline algorithm, which always recommends these ubiquitous modules and achieves almost 83% in intersection accuracy. The remaining baseline algorithms *MostFreqConn* and *WeightedRandom* perform significantly worse, achieving 65% and 53% in intersection accuracy, respectively.

The precision of all four structural algorithms ranges between 45 and 49 percent. This means that, on average, in a recommendation list of three components, one and a half components are useful for completing the composition. When considering precision, it is important to note that accuracy is defined for each query by observing a single example composition from which the query snapshot is extracted. Specifically, that one composition provides a use case where some components would be useful to complete the partial composition represented by the snapshot. However, it is possible that other components could also legitimately be considered useful recommendations. With this in mind, it is obvious that the reported precision values are, in a sense,

lower bounds on the actual usefulness of the recommender—some of the recommended components that are not useful for completing a partial composition into that particular direction could actually be useful to the user in practice.

All four structural recommenders achieve just over 60% recall, meaning that almost two thirds of useful components actually get recommended. These precision and recall values result in F_1 scores between 52 and 55 percent.

The baseline algorithms perform significantly worse in precision and recall. The maximum F_1 score is achieved by *MostPopular* at 43%, while *MostFreqConn* and *WeightedRandom* achieve 32% and 23%, respectively.

Three important qualitative differences in the results of various recommender algorithms on the Pipes dataset and on the synthetic dataset can be observed. First, the baseline algorithms *MostPopular* and *WeightedRandom* perform very poorly on the synthetic dataset across all four measures of accuracy, both in the absolute sense and relative to the structural algorithms. Particularly, the *MostPopular* algorithm achieves 7% in intersection accuracy, and under 3% in precision, recall and F_1 score. *WeightedRandom* performs even worse, scoring about half that. It can be concluded that, while these algorithms perform somewhat competitively in *Yahoo Pipes*, this is only due to the small number of components used in Pipes and the prevalence of several ubiquitous components. On the other hand, with a significantly larger component set and more diversity in the dataset, these algorithms are useless for any practical implementation.

The second significant difference between performance on the two datasets is is observed in the third baseline algorithm *MostFreqConn*. *MostFreqConn* reaches accuracy levels within one percent of both feature vector algorithms and *ComponentSeqEditDistance* for all four measures. This algorithm was specifically crafted to exploit the high level of regularity in connections between components that is induced into the dataset through component affinities, so this level of performance is not unexpected.

Third, while both feature vector algorithms and *ComponentSeqEditDistance* perform almost identically, the *GraphEditDistance* algorithm outperforms them by 5% in intersection accuracy (75% vs 70%) and between 3 and 5 percent in precision, recall, and the F_1 score.

Precision and recall of the structural algorithms is slightly more than 10% lower than in the Pipes dataset, but the achieved levels show that all four structural algorithms and *MostFreqConn* provide useful recommendations on this dataset.

8.4.2 Coverage

Recommender coverage was evaluated quantitatively in terms of total catalog coverage and successful coverage. Over both datasets, the best performing structural algorithm in terms of coverage is *GraphEditDistance*. On the Pipes dataset, it reaches 57% catalog coverage with 52% successful coverage. This means that more than half of the 55 modules available in *Yahoo Pipes* development got successfully recommended at least once in the 2000 queries.

On the other hand, coverage is somewhat lower on the synthetic dataset. *GraphEditDistance* achieves 46% catalog coverage and 36% successful coverage. While these numbers are lower than on the Pipes dataset, there are 762 components available in the synthetic dataset versus only 55 modules in Pipes. Furthermore, coverage is always reported against that total number of components, but, in fact, only 484 distinct components are actually considered useful in at least one query. As mentioned in subsection 8.3.2, this means that an ideal recommender with 100% accuracy could only hope to achieve up to 64% successful coverage. Additionally, while the considered values of recommended components per query are in fact a significant fraction of the total number of Pipes modules, they are vanishingly small compared to the number of components in the synthetic dataset.

This is illustrated in the difference in coverage of the *WeightedRandom* algorithm for the two datasets, as this algorithm can be expected to achieve the highest coverage of all the considered algorithms. For the Pipes dataset, *WeightedRandom* achieves almost 100% coverage even when only one component is recommended per query. However, for the synthetic dataset, it never recommends around 20% of the available components, i.e. it reaches 80% catalog coverage even when R is 10.

Of the remaining structural algorithms, *StructureVectorCos* outperforms *ComponentVectorCos* in both coverage and successful coverage, but only marginally. The *ComponentSeqEditDistance* algorithm performs more like *GraphEditDistance*, even outperforming it in total coverage for $R \geq 5$ on the Pipes dataset.

As mentioned above, the *WeightedRandom* algorithm recommends the most distinct components and dominates in total catalog coverage, but suffers low accuracy so its successful coverage is significantly lower. Specifically, it matches *GraphEditDistance* in successful coverage on the Pipes dataset, but successfully recommends about seven times fewer components on the synthetic dataset with 5% successful coverage versus 36% of the *GraphEditDistance* algorithm. *MostPopular* is expectedly the worst algorithm in terms of coverage, especially in the synthetic

dataset where it recommends less than 1% of all components. On the other hand, the *MostFreqConn* algorithm is competitive in coverage with the feature vector algorithms, especially on the synthetic dataset where it matches them nearly perfectly.

8.4.3 Response Time

The three baseline algorithms don't process the composition database and respond to queries nearly instantly, within several milliseconds. On the other hand, evaluation results show that both average and maximum algorithm response time grow linearly with the size of the composition database for all four structural algorithms, and that response time is dominated by the process of similarity evaluation, while the contribution of component scores computation is marginal.

As expected from the complexity analysis presented in section 6.5, the *ComponentSeqEditDistance* algorithm is by far the slowest. With a database of 7000 compositions, it requires close to a second to respond to queries on average, both on the Pipes and the synthetic dataset. Furthermore, the worst queries with the largest composition database on the two datasets took 7 and 3.3 seconds, respectively.

GraphEditDistance responds in around half a second for an average partial pipe, and requires up to 2 seconds. However, it performs significantly batter on the synthetic dataset with 250ms on average and up to 1.6 seconds. *Yahoo Pipes* require more computation because they contain many more matching duplicates in compositions, i.e. the same module appears many times in both the input partial composition and in many database compositions, and many matchings are possible. The *fetch* and *urlbuilder* modules are duplicated particularly often.

Finally, the *StructureVectorCos* feature vector algorithm requires less than 200ms on average on both datasets, and up to 450ms for the worst case query on both datasets. The simpler *ComponentVectorCos* algorithm responds about 30% faster in all cases.

Much research has been done on tolerable delays when using a computer [180–184], especially for the web [185–187]. Delays beyond a few seconds decrease user performance and change their intentions, e.g. making them less likely to continue using a site. A study conducted with 30 web users found that most users considered latency up to 5 seconds as *high quality of service* (QoS), but there was generally a lot of variance in these classifications such that, for example, 14 users classified an 8 second delay as high QoS, and 4 users classified it as low QoS which was used for delays over 11 seconds on average [186].

The maximal times listed above do not equate with user-perceived response time since they do not include networking costs, but they are the dominant component in recommender responsiveness. While the achieved response times might be sufficiently low in most cases, there are numerous approaches for increasing recommender responsiveness, and a few are mentioned in the remainder of the subsection. Primarily, evaluating similarity and updating component scores for different database compositions can be parallelized in a straightforward way. Furthermore, the recommender could report preliminary results at any point during computation and potentially refine them when the whole computation is done. The user could be kept aware of this process with a progress bar, which has been shown to increase perceived responsiveness [184].

Based on the results shown in figure 8.9, the easiest way to reduce response times for *Yahoo Pipes* would be to simply sample the database instead of computing similarity scores over the whole database, as larger database sizes provide little to no benefit for recommendation quality. This approach can be applied to other datasets as well, and provides a tradeoff between recommendation quality and response time.

With minimal preprocessing, database compositions that are completely dissimilar to the input partial composition, e.g. those having none or very few components in common, could be filtered out and ignored by the recommender. Implementing the inner loop of the algorithms in a compiled language such as C++ would also provide a significant speedup over the evaluated Python implementations.

Finally, to avoid high infrastructure costs that would be required to support a large number of simultaneous users, the recommender can be easily offloaded to the client side machine as the representation of compositions is very compact, and a database of a million compositions would require in the order of 100MB of storage, and much less with compression.

8.4.4 Effects of Similarity Filtering in Computing Component Scores

Similarity filtering provides a significant increase in both catalog and successful coverage for all four structural algorithms on the *Yahoo Pipes* dataset. The edit distance algorithms are most positively affected, increasing their successful coverage from 35% when no similarity filtering is applied to 55% with aggressive similarity filtering when BP is set to 5%, i.e. only the most similar 5% of compositions from the database are considered when computing component scores.

The qualitative effect of similarity filtering on coverage is also positive, as the analyzed

coverage curves show that more focused recommendations are made when similarity filtering is applied and the most popular modules get recommended less often, while the other modules get recommended more often.

A much smaller effect is observed in accuracy, but all four structural algorithms achieve the highest accuracy with $BP = 10\%$. For more aggressive similarity filtering, too few compositions are considered when computing component scores, and the algorithms are unable to identify three good components to recommend for some queries which decreases accuracy by up to 5%. A similar decrease is observed when no similarity filtering is applied. This is because good recommendations of unpopular modules get missed as the cumulative effect of many compositions with low similarity scores causes more popular modules to be preferred for recommendation. Eliminating this problem was the key motivating factor for introducing similarity filtering in the first place.

On the synthetic dataset, the influence of similarity filtering on both coverage and accuracy is minor. This fact can be explained by two related factors. First, compositions in the synthetic dataset are much more diverse than *Yahoo Pipes*. Therefore, even without similarity filtering, a significantly smaller number of database compositions with a nonzero similarity score should be expected for many queries. The tail of the similarity distribution of database compositions is not large enough to affect recommendations in a negative way. Second, if a component that would not actually be a useful recommendation appears in some compositions with lower similarity, it does not appear in a large fraction of those compositions as no components are ubiquitous in the dataset. Therefore, the cumulative effect of all these compositions with low similarity scores is not significant to the overall component ranking.

It can be concluded that similarity filtering successfully diminishes the domination of ubiquitous components on recommendations, but is not sufficient to completely eliminate it. In practice, special attention should be given to such components, as recommending them is not particularly useful except for novice users. A possible solution is to add weight factors to computed component scores based on component popularity such that more popular components need even higher component scores to be recommended over less popular components. The ultimate usefulness of such a strategy cannot be objectively tested in an off-line experiment, but instead requires a live experiment with real users.

Finally, evaluation results show that similarity filtering has almost no effect on recommender response time, implying that response times are completely dominated by similarity evaluation.

8.4.5 Effects of the Number of Recommended Components Per Query R

As is usually the case [46], increasing the number of recommendations per query increases recall, but decreases precision. For both datasets, the best balance of these two effects is achieved when two or three components are recommended per query, i.e. the F_1 score of all four structural recommender algorithms achieves its maximum for $R = 2$ and $R = 3$. This number of components is also in line with basic UI design considerations specific to component recommendation in consumer computing, presented in section 4.2.

Obviously, coverage of all algorithms also increases when more components are recommended per query. On the Pipes dataset, the edit distance algorithms, which are the best performers in terms of coverage, shown an increase of successful coverage from 25% to almost 80% when the number of recommended components is increased from 1 to 10. This dramatic increase is mainly the result of the fact that only 55 modules in total are available for Pipes composition so that the recommender chooses a significant portion of all available modules in each query when larger values of R are used.

On the synthetic dataset, where more than an order of magnitude more components are available, the coverage of these two algorithms increases from slightly over 20% to slightly over 40% in the same range of R values.

8.4.6 Effects of Composition Database Size N

Due to the regularity of the Pipes dataset, a database of only 750 Pipes is sufficient for all four structural recommender algorithms to achieve their maximal accuracy, i.e. larger databases of Pipes provide no measurable benefit.

On the synthetic dataset, the edit distance algorithms achieve higher accuracy with each larger composition database, but the increase slows down considerably after $N = 2500$. The feature vector algorithms and the well-performing *MostFreqConn* baseline algorithm reach their accuracy potential with only $N = 1500$ compositions in the database, and their accuracy increases only marginally beyond that. This difference between the two groups of algorithms is due to the different method of updating component scores. The edit distance algorithms use the higher degree of structural information gained through similarity evaluation to choose components in a more focused way. In a diverse dataset such as the synthetic dataset used for evaluation, these algorithms require a larger database of compositions to achieve their optimal accuracy. However, with a sufficiently large database, they outperform the simpler feature

vector algorithms by between 3 and 7 percent.

The effect of different database sizes on coverage is small and mostly related to the effect of similarity filtering, as increasing the database size with a constant level of similarity filtering is similar to decreasing similarity filtering with a constant database size. Specifically, due to this effect, the coverage of all structural algorithms decreases from the smallest to the largest database size on *Yahoo Pipes*. On the other hand, on the synthetic dataset where the effect of similarity filtering is minimal, successful coverage remains nearly constant for $N \geq 1500$.

Finally, both average and maximum response time of all algorithms grow linearly with the size of the composition database. This linear growth is caused by similarity evaluation of the input partial composition and every composition in the database.

8.4.7 Effects of Arcs in the Input Partial Composition

One of the requirements for component recommenders in consumer computing is that they are able to provide useful recommendations even when no or few connections between components are established in the input partial composition. This is because a valid and frequently used way of creating a composition is to first get all the required components onto the composition workspace and only then connect them. This workflow results naturally from the knowledge automation process.

All four algorithms presented in the dissertation were designed with this requirement in mind. To test how they perform when no connections are present in the input partial composition, all arcs from the snapshots were removed. On the Pipes dataset, only the *ComponentSeqEditDistance* algorithm is affected negatively by the removal of arcs, both in accuracy, where it loses between 3 and 5 percent across all measures, and in successful coverage, which decreases by up to 12%, but only when more than four components are recommended per query.

Practically no effect is observed on the synthetic dataset. While, as discussed above, it is a requirement for all component recommender algorithms that they provide useful recommendations when there are no connections in the input partial composition, the ideal component recommender algorithm should be able to provide better recommendations when more information is available. However, in both the Pipes and the synthetic dataset, connections are, in some sense, implied by the components. For example, in the Pipes dataset, if the *urlbuilder* and the *fetch* module are used together in a pipe, they are almost always connected, except when there are other instances of those same modules in the pipe. In the synthetic dataset, the strong

relationships between modules and connections is introduced through component affinities. The extent of this relationship can be expected to vary across different composition systems, but it is always present to some degree.

8.4.8 Evaluation Results Under the *Adjacent-Useful* Definition of Useful Recommendations

Under the *adjacent-useful* definition of useful recommendations, only those components that are directly connected to the snapshot in the completed composition the snapshot is generated from are considered accurate recommendations.

With this change, the intersection accuracy and precision of all algorithms decrease on both datasets. The largest effect is observed on the Pipes dataset, for small values of R when recommendations of the ubiquitous Pipes modules are most significant. Specifically, while recommendations of some of these ubiquitous modules are almost always accurate when any component in the completed composition is considered a useful recommendation, under the *adjacent-useful* definition of usefulness, this is true only for a smaller number of queries. Therefore, while precision of the structural algorithms decreases by between 14 and 25 percent when only one module is recommended, it decreases consistently by between 10 and 15 percent when three or more modules are recommended. A significantly smaller effect is observed on the synthetic dataset, where precision decreases by at most 6%.

A smaller change occurs in recall, which increases by up to 2% on the synthetic dataset, and changes between -5 and 4 percent on the Pipes dataset, depending on the algorithm and the number of components recommended per query. Combining the changes in precision and recall, the F_1 score decreases by up to 11% on the Pipes dataset, and up to 4% on the synthetic dataset.

Unlike accuracy, successful coverage is practically unaffected, while catalog coverage understandably stays exactly the same, with slight fluctuations in probabilistic algorithms.

Overall, it can be concluded that the presented algorithms inherently prefer components that are directly useful to augment the input partial composition, especially on the synthetic dataset. This is in line with expectations as it is likely that more evidence for the usefulness of those components directly connected to an input partial composition is present in the composition database than for other, more distant components.

Chapter 9

Simulated Composition Evaluation

In this chapter, a significantly different approach to evaluating recommender quality than the snapshot-based one used in chapter 8 is employed. Instead of presenting the recommender with example inputs generated from existing compositions, the whole process of creating the composition with the help of the component recommender is simulated. Specifically, starting with an empty composition workspace, components of an existing composition are added one by one and connected to recreate the composition. The main goal of this series of experiments is to further constrain the definition of what makes a component a useful recommendation in order to limit the effect of popular components on recommender accuracy results.

As the results of the previous chapter indicate, connections in the input partial composition have a minor effect on recommender quality. Therefore, the simulation process is focused only on components and all possible connections are made whenever a new component is added to the composition. In that sense, the composition at various stages of development resembles a snapshot used in the previous chapter as all the connections between the used components are established. The recommender is queried after each component is added and connected to the other components in the partial composition.

The key process in composition simulation is choosing which component to add at each step. The way users choose components and in which order they are added to a composition inevitably varies with the specific composition system and user preferences. Furthermore, this process is further complicated by the presence of the component recommender. The component recommender introduces several questions into composition simulation such as *is the user able to consider all the recommended components at every step?*, and *to what degree do recommendations change the user's intention, such that, for example, they add a component with some*

functionality that they didn't consider before?

Instead of trying to accurately model this complicated process, four distinct *simulation strategies* are proposed and the recommender algorithms are evaluated using these strategies. While these four strategies do not attempt to model every reasonable way a composition might be created, they are defined with the goal of spanning extremes in two areas. First, they aim to address the order in which the user adds functionality to a composition. On one extreme, the user wants to add functionality that directly extends the current partial composition. Specifically, the user is interested in a component that will directly interact with the components already in the partial composition. This order follows the data and control flow of a composition, and is therefore loosely equivalent to developing an application from its input, through its core logic, to its output. For example, in *Yahoo Pipes*, this corresponds to starting a pipe with a *urlbuilder* module, connecting it to a *fetch* module, and using a *filter* to create the *output* of the pipe, in that order. The described order of components is approximated with the generalized topological order defined in section 5.2.

On the other extreme, the user might want to add a component with some functionality that will eventually be useful in completing the composition, but will not necessarily be directly connected to any of the previously added components. In this model, the user might build the previously described pipe starting with the *output* module, then adding the *fetch* and *filter* modules, and finishing the pipe with a *urlbuilder* module. This possibility is modeled by a random order among the missing components that still need to be added to a partial composition to complete it.

The second variability that is addressed by the four proposed simulation strategies is the level at which the recommender might change the user's intentions. On one extreme, the user considers the recommended components and adds one of the recommended components only if it is perfectly in line with what they were looking for. For example, if the user wants to somehow get a data feed and the *fetch* module is one of the components recommended by the component recommender, the user will identify that this module has the desired functionality and add it to the partial composition. However, if none of the recommended components offers the desired functionality, the user will find such a component through other means, e.g. by employing textual search or asking a friend.

On the other extreme, the user is able to identify any piece of functionality that will eventually be useful to complete the composition. For example, when building the previously de-

scribed composition, the user might want to get a data feed into the pipe at some stage of composition, but will also recognize a recommended *filter* module as useful and add it first. Again, if none of the recommended components are useful for completing the composition, the user then proceeds to find a suitable component in some other way.

The four simulation strategies that try to model these described situations are defined and applied to a simple example in section 9.1. Then, sections 9.2 and 9.3 present the simulated composition evaluation results for *Yahoo Pipes* and the synthetic dataset, respectively. Accuracy and recommender response time are considered with different numbers of components recommended per query and different database sizes. Note that the default database size used in these experiments was 1500 to keep the total evaluation time manageable, and a fixed level of similarity filtering with $BP = 10\%$ is maintained.

In this evaluation scenario, accuracy is binary—a recommendation is considered accurate if and only if one of the recommended components is actually added as the next component of the composition. This definition is equivalent to both intersection accuracy and recall if only the added component is considered useful, while precision is then a constant fraction of recall depending on the number of components recommended per query. Therefore, only the single measure of accuracy defined above is reported. A possible interpretation of this measure is user-perceived recommender accuracy.

No recommendations are made for the empty composition. In practice, it is reasonable to initially recommend generally useful and frequently used components, if such components exist in a particular composition system. However, since this recommendation is not based on any partial composition, every algorithm would score the same on this recommendation which would not contribute to their comparison. Additionally, as a technical detail, since none of the algorithms ever recommends a component that is already a part of the partial composition, the unsuccessful recommendations that result when a previously used component is added to the composition are not recorded.

All the presented results are based on simulated composition of the exact set of 500 compositions that were also used to generate snapshots for snapshot evaluation. A significant change is made to the *ComponentSeqEditDistance* algorithm in that the component sequence of the input partial composition is ordered by insertion time, i.e. components are simply appended to the sequence as the composition is created. As explained in section 6.3, this order then allows the algorithm to reuse previously computed edit distances of earlier versions of the same partial

Table I: Four strategies for simulating how components are added to a partial composition.

	Topological	Random
Unguided	*AlwaysTopological*	*AlwaysRandom*
Guided	*RecThenTopological*	*RecThenRandom*

composition to compute similarity scores much more efficiently, with time complexity $\Theta(n_Q)$ per database composition Q instead of $\Theta(n_P n_Q)$. The component sequences of the database compositions are still kept in generalized topological order as their development is not simulated so the component insertion order is not known.

To conclude the chapter, the results over both datasets are summarized and discussed in section 9.4, and compared to the results obtained through snapshot evaluation.

9.1 Simulation Strategies

Under the first proposed simulation strategy, components are added to the partial composition in generalized topological order, regardless of recommendations made by the recommender system. This strategy corresponds to the situation described in the chapter introduction in which the user adds functionality in a linear order and only identifies a recommended component as useful if it has that particular functionality that is required next. Based on the mechanics of the strategy, it is called *AlwaysTopological*.

In the second strategy, called *AlwaysRandom*, a random missing component is added to the partial composition, again regardless of recommendations. Both these strategies share the property that they model a composition process in which the user has a strong preference for what functionality to add next and only uses the recommender to pick an appropriate component that has that functionality if such a component is recommended. Otherwise, it is assumed that the user employs some other component discovery mechanism to find a component with the desired functionality.

The remaining two strategies follow the opposite idea and model composition processes in which the user is ready to change the preferred order of adding functionality to the composition provided that useful recommendations are provided. Therefore, both of these strategies are called *guided* simulation strategies, whereas *AlwaysTopological* and *AlwaysRandom* are *unguided*.

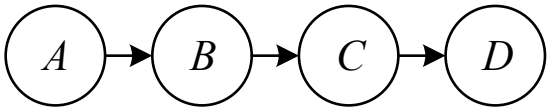


Figure 9.1: A simple composition of four components used to illustrate the four simulation strategies.

The two guided simulation strategies differ only in situations in which no useful component is recommended by the recommender system. When any useful component is recommended, it is added to the partial composition. When several useful components are recommended, the one higher in the list of recommendations is given priority. On the other hand, when no useful components are recommended, simulation proceeds in a way similar to the unguided strategies.

Specifically, under the *RecThenTopological*[1] simulation strategy, the next component in topological order is added when no useful components are recommended. Analogously, under the *RecThenRandom* simulation strategy, a random useful component is added to the partial composition if no useful components are recommended.

The names of the four described simulation strategies are shown in table I. The strategy names are arranged in a 2×2 matrix such that strategies in the same row of the matrix model the same type of interaction between the user and the recommender system, and those in the same column model the same preferred way of adding functionality to the composition.

To further illustrate these four simulation strategies, they are applied to one step of development of the simple composition shown in figure 9.1. Such a simple composition was chosen to make the topological order unique and simplify the discussion. Structurally, this composition corresponds perfectly to the pipe desribed in the chapter introduction, but symbolic names are used to make the figures more readable.

The four strategies are applied to the example composition when the partial composition contains only component A in figure 9.2[2]. The unguided simulation strategies are illustrated in the first two rows. Completely independently of recommendations, under the *AlwaysTopological* simulation strategy, component B is added to the partial composition and connected to A because it is the next component in the topological order of the completed composition. Similarly, in the *AlwaysRandom* strategy, a random useful component is added irrespective of recommendations. In the figure, it is assumed that component C is chosen at random from the set of components B, C and D.

[1] *Rec* stands for *recommended*.

[2] Note that under the two random simulation strategies, the first component is chosen at random. However, to more easily focus on the differences between the simulation strategies, it is assumed that component A was chosen so that all four strategies start from the same partial composition.

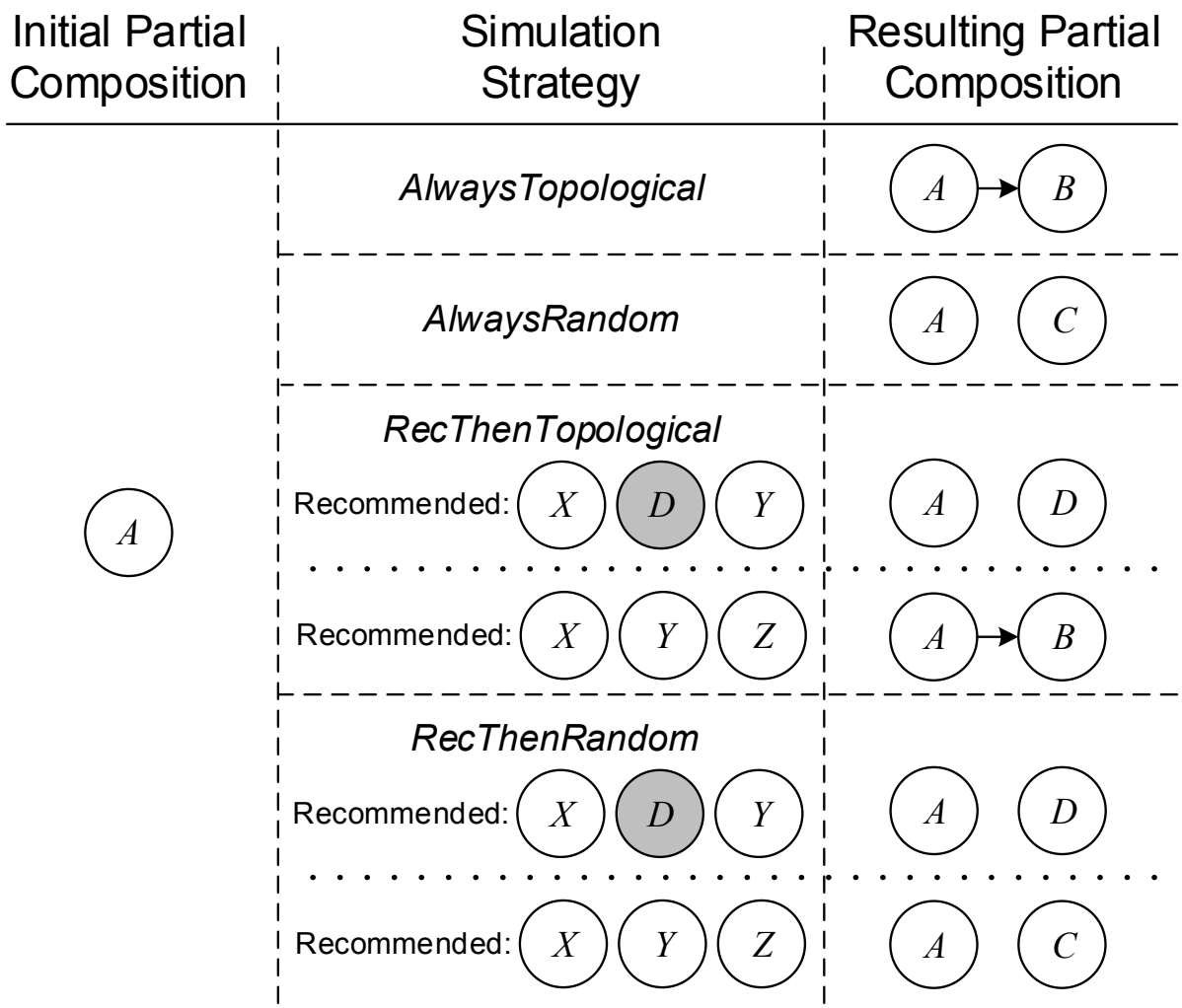


Figure 9.2: An example application of the four simulation strategies. It is assumed that component C is chosen when a random component is needed.

The operation of guided strategies is more complex in that the result depends on the output of the recommender. Two examples of component recommendations for each of the two strategies are shown in the figure. In the first case, components X, D and Y are recommended. In this case, both the *RecThenTopological* and *RecThenRandom* strategies add the component D to the partial composition because it is useful for finishing the composition and it has been recommended.

The two strategies differ in the second example, when three components that are not in the completed composition (X, Y and Z) are recommended. In this case, the *RecThenTopological* strategy proceeds like the *AlwaysTopological* strategy by adding component B, while the *RecThenRandom* strategy proceeds like the *AlwaysRandom* strategy and adds component C to the composition.

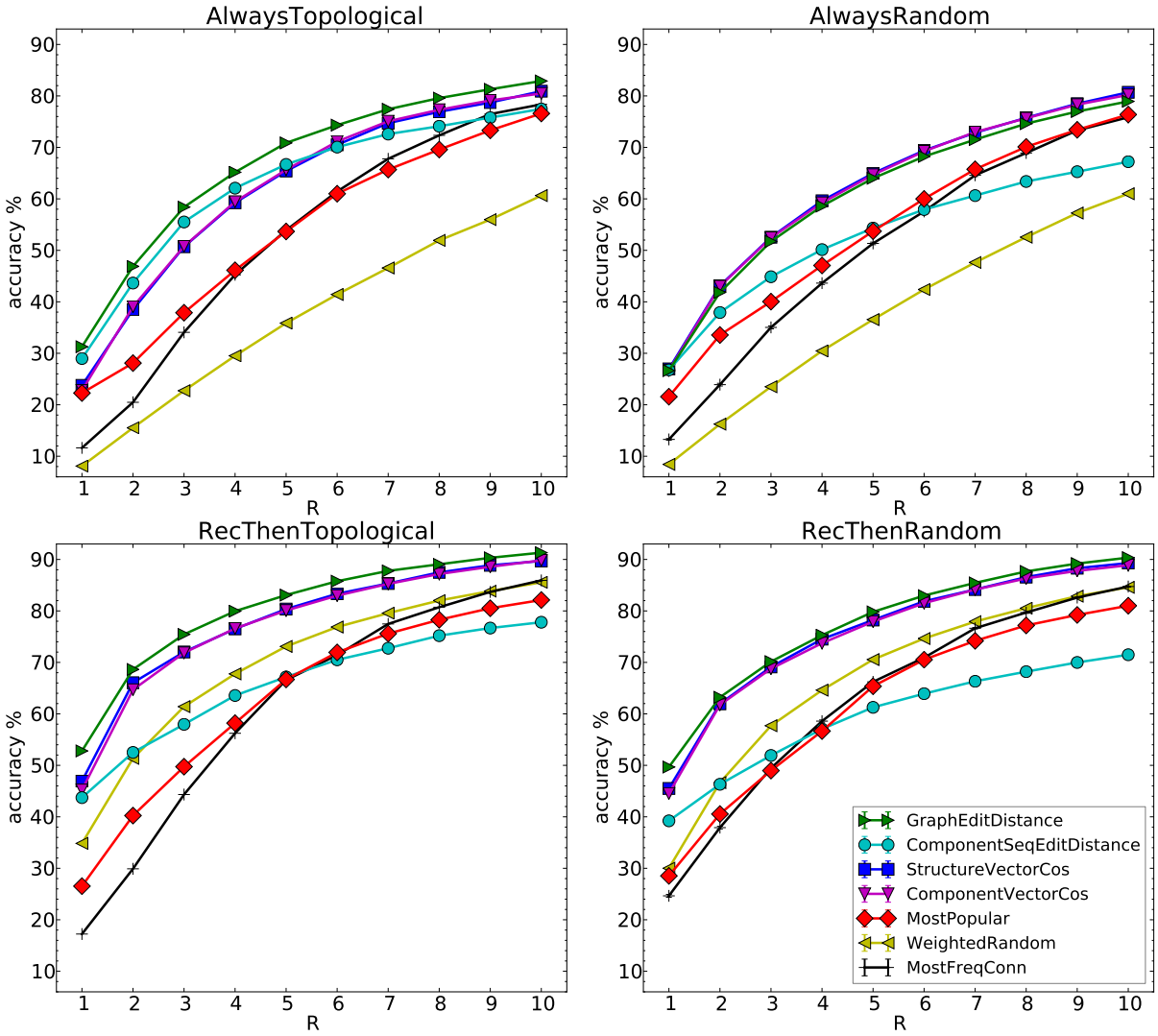


Figure 9.3: Changes in recommender accuracy under different simulation strategies for R between 1 and 10 with $BP = 10\%$ and $N = 1500$ for the Pipes dataset.

9.2 Simulated Composition Evaluation Results on *Yahoo Pipes*

On the Pipes dataset, a total of 3834 queries are submitted to the component recommender system for each experiment, as the 500 test pipes are created component by component. This section is divided into two subsections. In subsection 9.3.1, the accuracy results under different simulation strategies and with different numbers of recommended components per query are presented. Following that, subsection 9.3.2 analyzes accuracy and response time in this simulation model with different composition database sizes.

9.2.1 Effects of Simulation Strategy and the Number of Recommended Components per Query R

Average accuracy over the 3834 recommendations under all four simulation strategies and different numbers of recommended components per query is shown in figure 9.3. The graphs for the four simulation strategies are arranged in the same way as in table I, with unguided strategies in the top row, and guided strategies in the bottom row.

Several interesting effects can be observed in the presented graphs. As already established in subsection 8.2.4, the *ComponentSeqEditDistance* algorithm is sensitive to component sequence order on the Pipes dataset. When compositions are created under a topological simulation strategy (i.e. either *AlwaysTopological* or *RecThenTopological*), the order in which components are added to the partial composition closely matches the generalized topological order that is used by *ComponentSeqEditDistance* to represent database compositions. On the other hand, under the two randomized simulation strategies where components are added in a more irregular manner, the differences in the semantics of the component sequence decrease the algorithm's accuracy by up to 12%.

With the same change from a topological strategy to its corresponding random strategy, the accuracy of the *GraphEditDistance* algorithm decreases consistently by between 2 and 6 percent, while the feature vector algorithms are nearly unaffected, and in fact, achieve slightly higher accuracy under randomized strategies when a small number of components are recommended per query. This difference in behavior is easily explained by the way component scores are computed in *GraphEditDistance* and the feature vector algorithms. The *GraphEditDistance* algorithm prefers components that are more likely to be directly connected to the partial composition and those components are more likely to be added with a topological strategy than with a random strategy. The feature vector algorithms are unaffected by the order in which components are added to the composition and also don't prefer directly connected components when computing component scores.

A significant difference in accuracy is also apparent with strategies in the same column, i.e. those that share the same model of the preferred way of adding functionality to a composition, especially when a smaller number of modules is recommended per query. For example, the accuracy of all algorithms increases by up to 22% when the *AlwaysTopological* strategy is replaced by the *RecThenTopological* strategy. This happens because, under the two guided strategies, a recommendation is accurate if any component that was actually used to create the

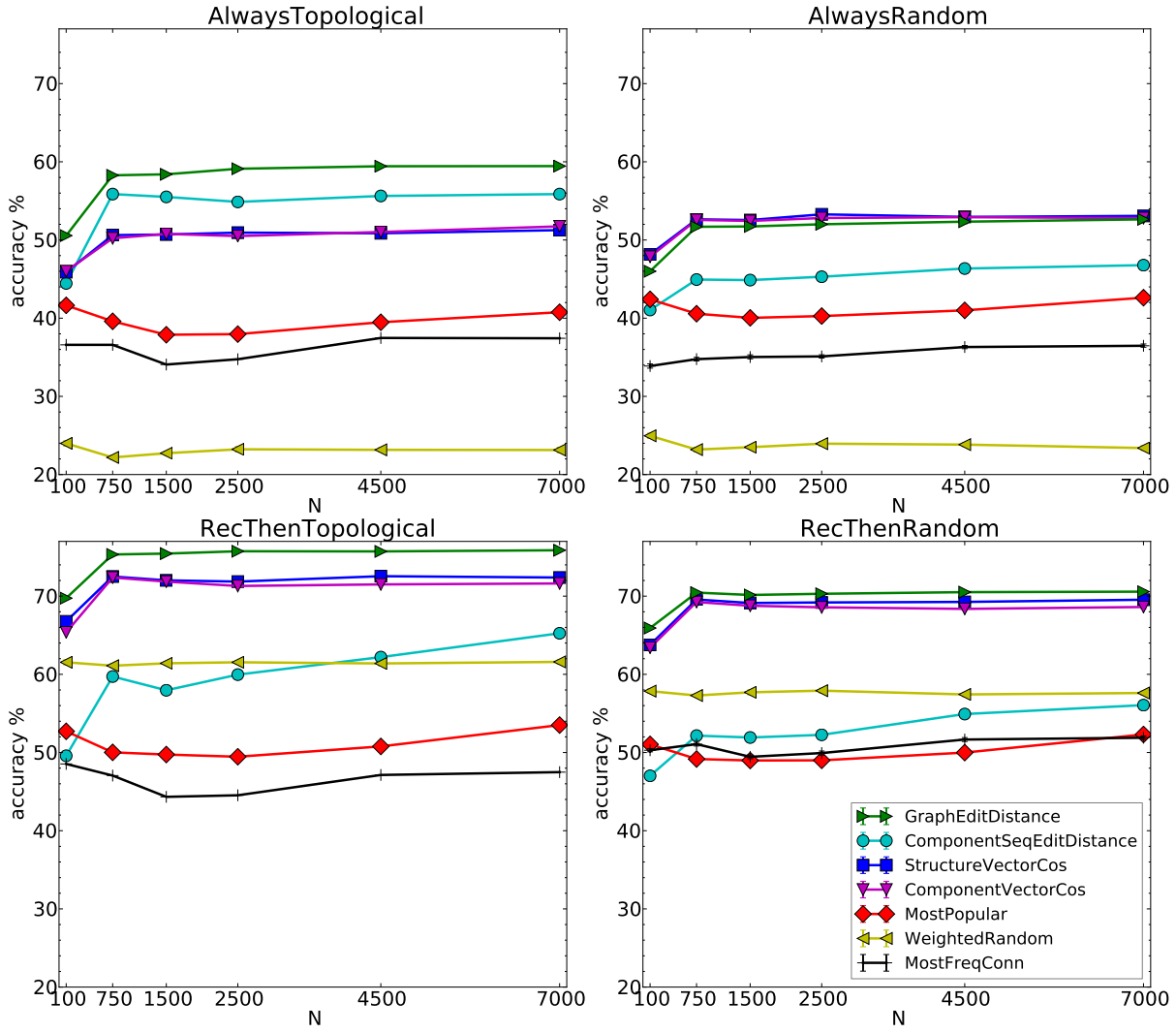


Figure 9.4: Changes in recommender accuracy under different simulation strategies for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the Pipes dataset.

previously completed composition whose construction is being simulated gets recommended.

9.2.2 Effects of the Composition Database Size N

A different view of the same effects on accuracy is presented in figure 9.4, where accuracy is graphed for different composition database sizes with a constant three modules recommended per query. As was already observed in snapshot evaluation, on the *Yahoo Pipes* dataset, a very small database is sufficient for all algorithms to achieve their maximum accuracy, and this remains true under all four simulation strategies.

These graphs show more clearly how the accuracies of the structural algorithms disperse compared to in snapshot evaluation where all four algorithms achieve nearly identical accuracy. Under both topological strategies, the *GraphEditDistance* algorithm performs significantly bet-

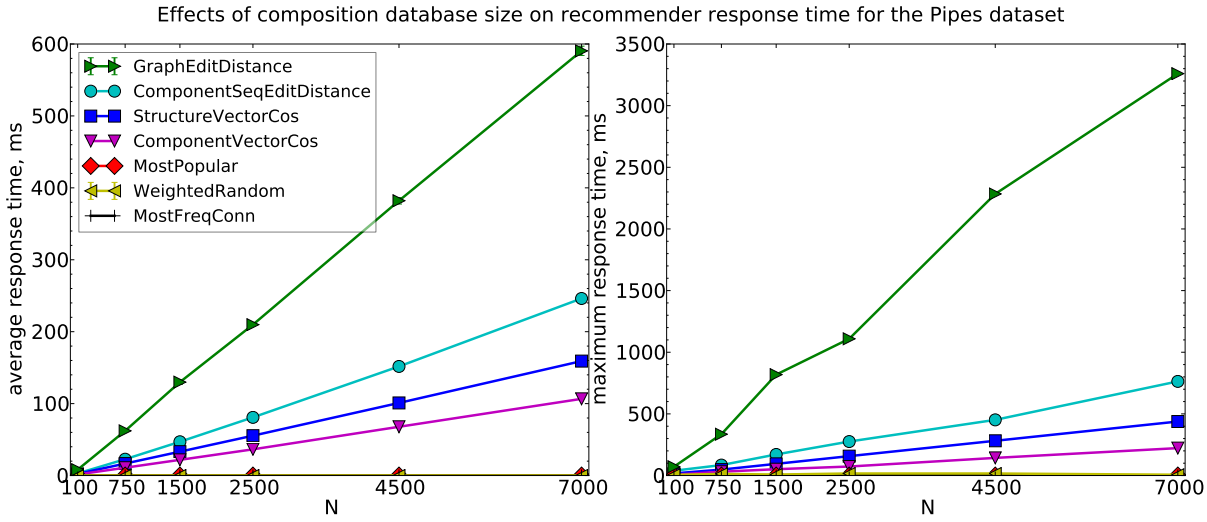


Figure 9.5: Changes in recommender average and maximum response time under the *AlwaysTopological* simulation strategy for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the Pipes dataset.

ter than the other three algorithms. Specifically, it reaches between 4 and 9 percent higher accuracy than the feature vector algorithms and between 4 and 17 percent higher accuracy than *ComponentSeqEditDistance*. This effect is discussed in detail in section 9.4.

A significant difference from snapshot evaluation results can also be seen in algorithm response times shown in figure 9.5. As response times are not affected by the applied simulation strategy, the graphs only show the gathered data for the *AlwaysTopological* strategy.

When components of the input partial composition are put into the component sequence representation in the same order in which they were added to the composition, the *ComponentSeqEditDistance* algorithm is able to reuse already computed edit distances for previous versions of the same partial composition which considerably speeds up its execution. The average response time decreases from 1s observed in snapshot evaluation to 223ms. Similarly, the maximum response time decreases from over 7s to just 752ms.

The response time of the feature vector algorithms increases slightly. Furthermore, average response time of *GraphEditDistance* increases by almost 100ms, while its maximum response time is over 3s—an increase of more than a second from snapshot evaluation. These increases are expected as there are several outliers in the dataset in terms of pipe size and snapshots extracted from them are expected to only use half the modules used in the whole pipe.

9.3 Simulated Composition Evaluation Results on the Synthetic Dataset

On the synthetic dataset, every experiment consisted of 4100 queries that were processed by the recommender while the development of the 500 compositions also used for snapshot generation was simulated. The results of these experiments are presented in the following subsections in an analogous way to the previous section—the effects of different simulation strategies and the number of components recommended per query are analyzed in subsection 9.3.1, and accuracy and response time with different composition database sizes are presented in subsection 9.3.2.

9.3.1 Effects of Simulation Strategy and the Number of Recommended Components per Query R

The accuracy of the recommender algorithms under the four simulation strategies for different values of R is shown in figure 9.6. No significant difference in accuracy can be observed between the simulation strategies in the same row, as all changes are bounded by 2%, regardless of the number of recommended components. This is in contrast to the Pipes dataset where the accuracy of *GraphEditDistance* and especially *ComponentSeqEditDistance* decreases when functionality is added in a random order instead of the generalized topological order. One of the conclusions of snapshot evaluation is that the generalized topological order isn't representative of the structure of synthetic compositions, which was observed by the lack of effect of arcs in the input partial composition on the accuracy of the *ComponentSeqEditDistance* algorithm in subsection 8.3.4. The results presented in the figure further confirm this conclusion as all the algorithms, including *ComponentSeqEditDistance*, perform nearly identically under corresponding topological and random simulation strategies.

On the other hand, accuracy of all structural algorithms increases consistently by about 10% when moving from the top to the bottom row, i.e. when a guided strategy is employed over its corresponding unguided counterpart. The increase in accuracy is achieved because more components are candidates for accurate recommendations under guided simulation strategies. This effect is about doubled on the Pipes dataset due to the significantly smaller set of components. Specifically, the same absolute increase in the number of candidate components for accurate recommendations makes the recommendation problem easier if there is a smaller total number

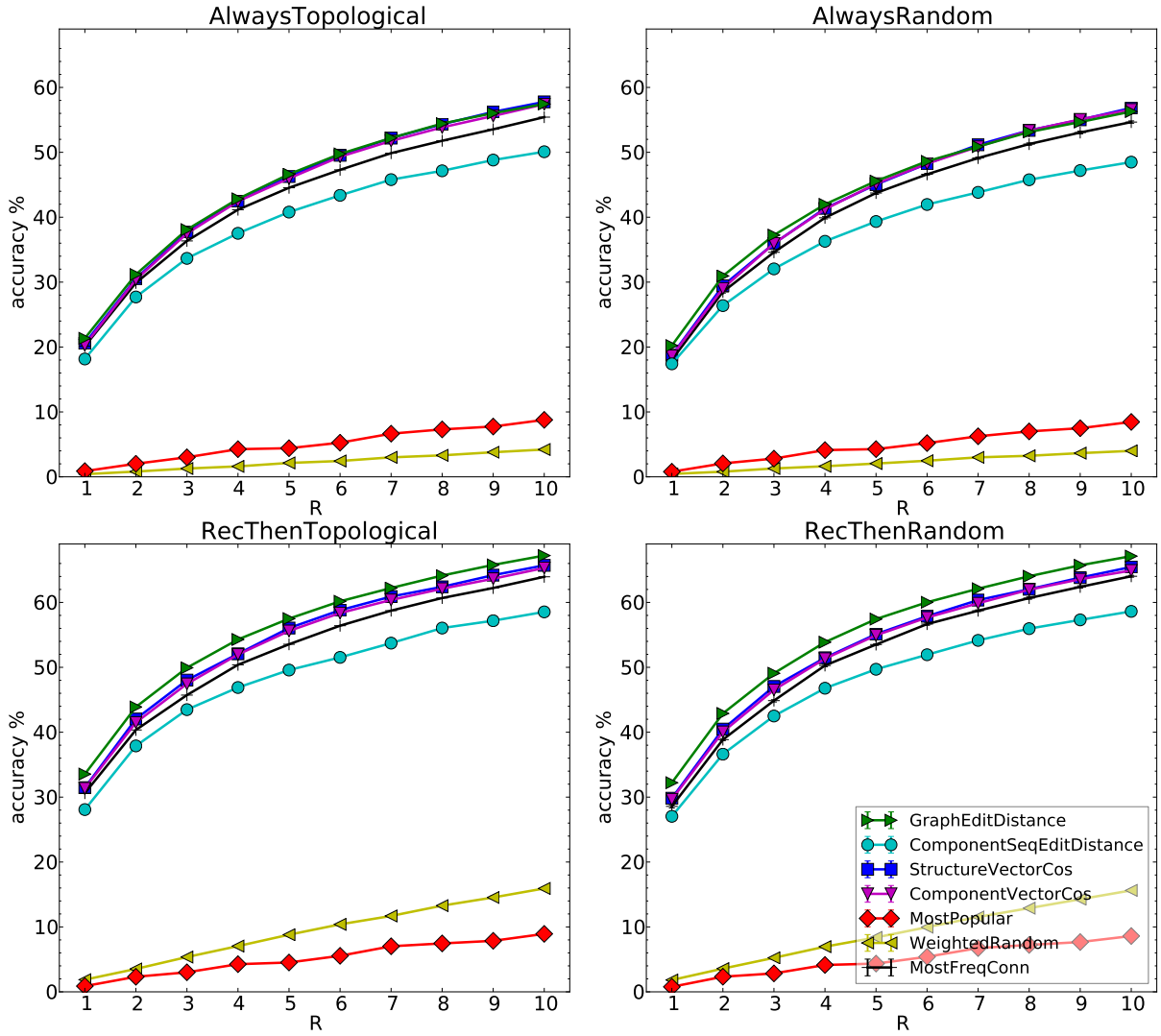


Figure 9.6: Changes in recommender accuracy under different simulation strategies for R between 1 and 10 with $BP = 10\%$ and $N = 1500$ for the synthetic dataset.

of components to choose from. This difference can best be understood by examining the behavior of the *WeightedRandom* algorithm on the Pipes and synthetic datasets as it is basically guessing useful components. While its accuracy increases by up to 35% when a guided strategy is used instead of its corresponding unguided strategy on the Pipes dataset, only an increase of up to 12% is observed on the synthetic dataset, and much smaller gains are achieved with lower values of R.

9.3.2 Effects of the Composition Database Size N

The lack of change between a topological and its corresponding random simulation strategy discussed in the previous subsection can be seen more clearly for the $R = 3$ case in figure 9.7 where accuracy is graphed against the composition database size. In fact, it can be observed that

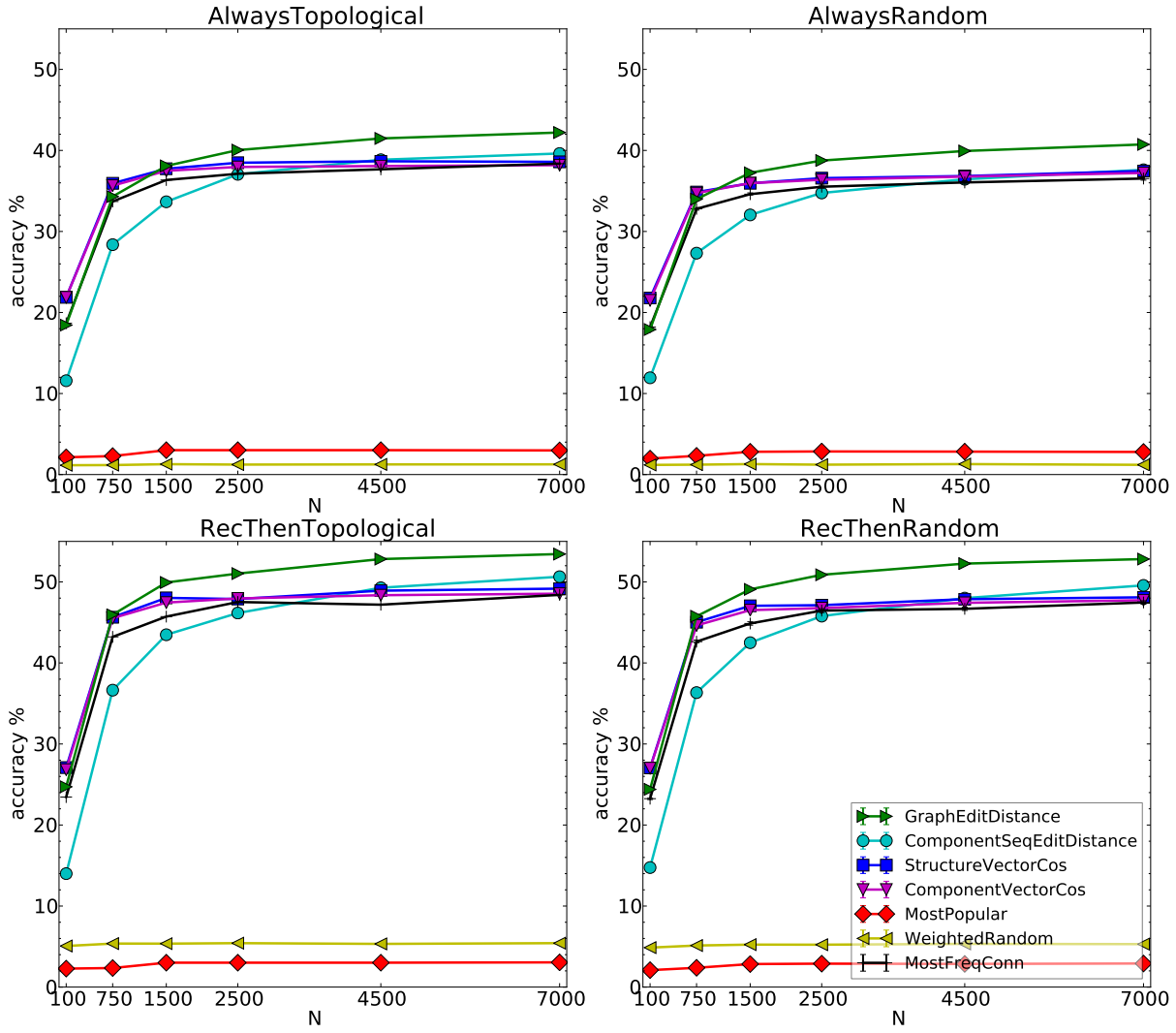


Figure 9.7: Changes in recommender accuracy under different simulation strategies for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the synthetic dataset.

the accuracy graphs for all four simulation strategies are qualitatively identical to the intersection accuracy graph produced in snapshot evaluation. As the size of database composition increases, *GraphEditDistance* and *ComponentSeqEditDistance* become more accurate while both feature vector algorithms achieve their maximum accuracy potential with 1500 compositions in the database.

Finally, response times under the *AlwaysTopological* simulation strategy are shown in figure 9.8. A similar change compared to snapshot evaluation is observed as on the Pipes dataset—the response times of *GraphEditDistance* and both feature vector algorithms slightly increase as more large input partial compositions are processed, while the *ComponentSeqEditDistance* algorithm performs much better than in snapshot evaluation due to the change in the order of its underlying component sequence model which allows edit distance computations to be continued

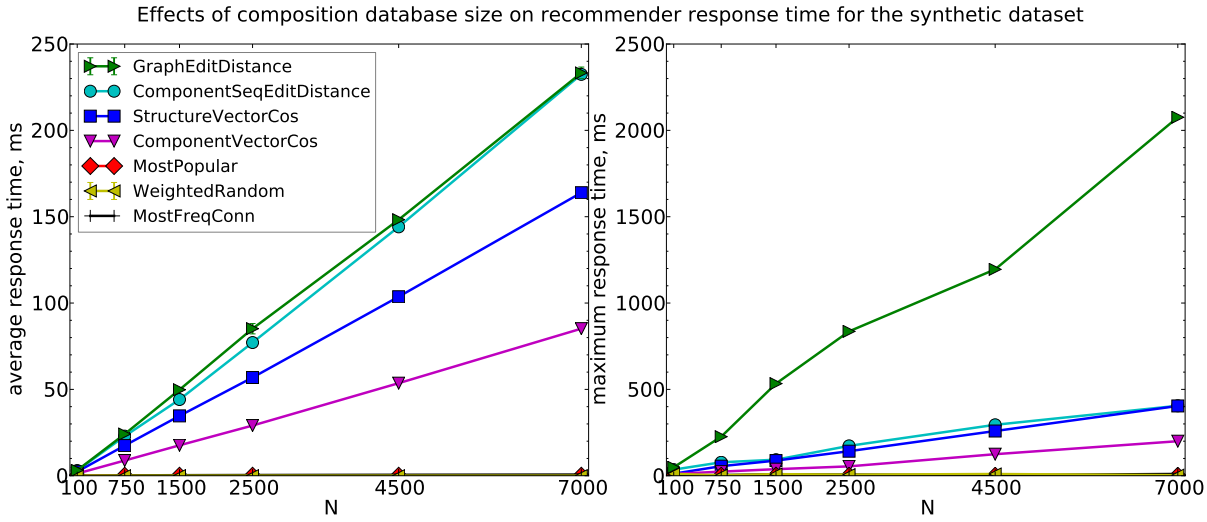


Figure 9.8: Changes in recommender average and maximum response time under the *AlwaysTopological* simulation strategy for N between 100 and 7000 with $BP = 10\%$ and $R = 3$ for the synthetic dataset.

from previous versions of the same partial composition.

9.4 Simulated Composition Evaluation Summary

Several interesting properties of the recommender algorithms are highlighted by the simulated composition evaluation results presented in this chapter.

First, there is a systematic decrease in accuracy compared to previously discussed results from snapshot evaluation, especially when the *AlwaysTopological* or *AlwaysRandom* simulation strategies are used, up to 40% when three modules are recommended per query on the Pipes dataset and up to 30% on the synthetic dataset. The most significant factor in this effect is the reduced number of components that are considered accurate recommendations in simulated composition evaluation. In particular, under both the *AlwaysTopological* and *AlwaysRandom* strategies, only one component is considered an accurate recommendation for each query while almost three components are considered accurate recommendations in snapshot evaluation.

Furthermore, a significant contribution to this decrease on the Pipes dataset comes from the ubiquitous modules—in snapshot evaluation, these modules are often considered useful recommendations, while they can be successfully recommended at most once per composition in the simulated composition evaluation scenario. This is easily confirmed by observing the accuracy of the *MostPopular* algorithm. It achieves almost 80% of intersection accuracy in snapshot evaluation when only two modules are recommended per query. On the other hand, under the *AlwaysTopological* simulation strategy, it reaches less than 30% accuracy with $R = 2$,

and only comes close to 80% accuracy when 10 modules are recommended per query.

For intuition, it is easiest to consider recommendations of the *output* module. While it is often considered an accurate recommendation in snapshot evaluation, it is actually always added last under the *AlwaysTopological* simulation strategy. Therefore, while *MostPopular* keeps recommending it at every step if $R \geq 2$, only the last recommendation is considered accurate.

This insight provides a useful tool to further analyze to what degree different algorithms prefer these very popular modules over more niche recommendations. Of the structural algorithms, the accuracy of the feature vector algorithms decreases the most, especially for smaller values of R. This is expected as these algorithms are closely related to the *MostPopular* algorithm due to their way of computing component scores, as explained in subsection 8.2.2. On the Pipes dataset, the *GraphEditDistance* algorithm achieves almost 10% higher accuracy than the feature vector algorithms under the *AlwaysTopological* strategy. Similarly, the *ComponentSeqEditDistance* algorithm outperforms the feature vector algorithms by 5%.

These differences between the way in which accuracy of different algorithms is affected by the change in evaluation scenario from snapshot evaluation to simulated development is due to two related factors. First, the edit distance algorithms use the matching information obtained from computing edit distances to compute component scores in a more focused way, with the aim to provide more relevant recommendations that are less affected by component popularity. As seen in the results of snapshot evaluation, this allows the algorithms to achieve higher coverage than the feature vector algorithms. Second, modules that are added by the *AlwaysTopological* simulation strategy are directly connected to the input partial composition and are therefore also more likely to be recommended by the edit distance algorithms.

However, both these factors are also present on the synthetic dataset, but the edit distance algorithms do not perform any better in the simulated composition evaluation scenario. There are several likely explanations for this fact. First, synthetic compositions are much denser, with almost twice as many connections than found in pipes, on average. Therefore, the benefit of more focused computation of component scores is diminished. Second, the higher structural complexity of synthetic compositions significantly increases the number of possible generalized topological orderings of components, thus reducing the structural significance of any single ordering. As observed in section 9.3.1, there is no significant difference between algorithm accuracies under a topological and its corresponding random simulation strategy on the synthetic

dataset, while these changes in strategy reduce the accuracy of edit distance algorithms by up to 12% on the *Yahoo Pipes* dataset.

On both datasets, the accuracy of all structural algorithms increases when an unguided strategy is replaced by its guided counterpart by up to 22%. This increase is more pronounced on the Pipes dataset where there are fewer components to choose from and this change of strategy increases the number of components that are considered useful recommendations for most queries. In that sense, both the effect itself and the differences between the datasets are closely related to the effect of the *adjacent-useful* definition of useful recommendations which was analyzed in snapshot evaluation.

In terms of response time, the most significant change from what was seen in snapshot evaluation is observed for the *ComponentSeqEditDistance* algorithm. Its average response time decreases from one second in snapshot evaluation to 223 milliseconds, and its maximum response time decreases from over 7 seconds to only 752 milliseconds. Similar decreases are seen on the synthetic dataset. In simulated composition evaluation, the *ComponentSeqEditDistance* algorithm is changed so that it stores components of the input partial composition in the component sequence representation in the order in which components were added. With this change, it is possible to reuse edit distance information computed for previous versions of the same partial composition to significantly speed up the algorithm, at the cost of increased storage requirements, as discussed in section 6.3.

Chapter 10

Conclusion

The goal of this dissertation was to address the component discovery challenge in consumer computing through a component recommender system based on structural similarity of compositions. The key scientific contributions of this thesis are: (1) a set of formal models of consumer application structure, (2) a method for component recommendation based on analysis of structural similarity of consumer applications, (3) a set of component recommender algorithms for real-time assistance in composite consumer application development and (4) an extensive evaluation of the proposed set of algorithms.

The hypothesis of the presented research was that composition knowledge stored in completed consumer applications can be used to make useful component recommendations by analyzing the structural similarity of these compositions and a partial composition that needs to be completed. Four formal models of consumer application structure were defined in chapter 5 to explore how different levels of structural information impact the quality of the recommender (1). The most general model which preserves most of the structural information about a composition is based on a directed graph with vertex labels. As components carry most of the functionality of composite applications in general and especially consumer applications, a simpler model where a composition is represented by a sequence of its components is also considered. Finally, as is common in many recommender systems, feature vector models were also considered as they are simple and easy to compare for similarity. Two feature vector models were defined, one representing only components of a composition and one representing both components and connections between them. Both representations were employed to explore the impact of connection information on recommender quality.

A general method for component recommendation based on structural similarity of compo-

sitions (2) is defined in chapter 6. Through this method, the basic functional requirements for a component recommender algorithm are defined. When defining a component recommender based on this method, three decisions are most important. First, composition structure needs to be represented in some way. Second, the algorithms should define how two compositions represented in the chosen model are compared for similarity. Third, having computed the similarity of two compositions, the algorithm needs to define how this similarity affects which components are recommended. Specifically, the algorithms assign *scores* to components, and the components with the highest scores are recommended to the consumer.

Four component recommender algorithms are defined in this way (3) in chapter 6: a probabilistic algorithm based on graph edit distance over the digraph model (*GraphEditDistance*), a component sequence edit distance algorithm (*ComponentSeqEditDistance*) and two algorithms based on cosine similarity of feature vectors, one including only components (*ComponentVectorCos*) and one including both components and connections (*StructureVectorCos*) in the vectors. Besides the different representations and similarity measures, the edit distance algorithms and vector algorithms are significantly different in the way they compute component scores. The edit distance algorithms use the additional structural information obtained from the edit distance process to focus recommendations to those components that are often connected to components in the consumer's input composition in similar compositions in the database. On the other hand, the feature vector algorithms recommend components that are simply used frequently in compositions similar to the input partial composition. By comparing these two classes of algorithms, it is possible to explore if this additional structural information provides any benefit in recommender quality.

The defined algorithms are evaluated through a comprehensive set of experiments (4). As commonly seen in evaluation of recommender systems, every recommender should be directly compared to simple intuitive approaches that qualify the problem domain. It can often be the case that one of these simple approaches outperforms a complex algorithm. To that end, three baseline algorithms are defined: an algorithm recommending the most popular components (*MostPopular*), a random recommender (*WeightedRandom*) and an algorithm that recommends components that are most often directly connected to components of the input partial composition (*MostFreqConn*).

Evaluation was conducted on two sets of compositions: one extracted from *Yahoo Pipes* and a synthetic set of compositions. The synthetic set was defined with the goal of exploring how

the algorithms perform for structurally more complex and diverse compositions as *Yahoo Pipes* compositions are very linear in nature as they have no concept of control flow.

The five key evaluation results are as follows. First, results of all experiments clearly indicate that the proposed approach to component recommendation is effective for both *Yahoo Pipes* and the synthetic dataset. On the Pipes dataset, all structural algorithms perform similarly in terms of accuracy, achieving slightly over 90% intersection accuracy, 45% precision and 60% recall when three components are recommended per query. Comparatively, the baseline algorithms perform between 10 and 40 percent worse in all measures.

In coverage, the edit distance algorithms significantly outperform feature vector algorithms, by up to 15% in both total and successful coverage. *GraphEditDistance*, which is the best performer in coverage in general, achieves 57% catalog coverage and 52% successful coverage. This means that more than half of the available Pipes modules are recommended successfully at least once in 2000 recommendations made during the experiment. Furthermore, an analysis of coverage curves shows that the coverage of *GraphEditDistance* is also the best qualitatively in the sense that it recommends less popular components more frequently than other algorithms.

This disparity in coverage results between the edit distance and vector algorithms is attributed to the difference in the way component scores are computed by the algorithms. The more focused component selection process based on additional structural information allows the algorithms to recommend less popular components more often.

On the synthetic dataset, similar qualitative behavior is observed, with two significant differences. First, due to the diversity of the dataset and a much larger component set (762 versus 55 components in the Pipes dataset), the baseline algorithms *MostPopular* and *WeightedRandom* perform very poorly on the synthetic dataset, with accuracy results well below 10% in all experiments. On the other hand, the structural algorithms are grouped around 70% intersection accuracy, with *GraphEditDistance* 5% above. These results are lower than on the Pipes dataset, but that is to be expected as the component recommendation problem is significantly more difficult on the synthetic dataset.

The second significant difference is the performance of the baseline *MostFreqConn* algorithm which matches feature vector algorithms in both accuracy and coverage, but is outperformed by *GraphEditDistance*. *MostFreqConn* performs better on the synthetic dataset than on the Pipes dataset because is it specifically crafted to exploit the regularity in component connectivity that is present in the synthetic dataset.

GraphEditDistance achieves 48% total coverage and 36% successful coverage on the synthetic dataset, outperforming the other structural algorithms by between 5 and 10 percent. Coverage results are lower than on the Pipes dataset due to the fact that the number of available components is more than an order of magnitude greater. Furthermore, to put these coverage results in proper context, due to the way the experiment was set up, an ideal recommender that had 100% precision would only achieve 64% successful coverage.

Overall, the results on the synthetic dataset suggest that the proposed approach generalizes well to more complex composition systems where there is regularity in the way components are connected.

The second significant evaluation result pertains to the concept of *similarity filtering* which is regulated by the BP algorithm parameter. Extensive evaluation shows that optimal accuracy and coverage are achieved when only 10% of all database compositions that are most similar to the input partial composition are used for making recommendations.

Third, the optimal balance between precision and recall for all structural algorithms is obtained when two or three components are recommended per query, which is regulated by the R parameter. For these two values, the F_1 score achieves a global maximum. As the cost of false positive component recommendations in consumer computing can be expected to be low, it is suggested that three or four components be recommended per query by default, although further experimentation on this issue is necessary.

Fourth, all four proposed algorithms are suitable for design-time interactive use in current consumer computing composition systems as they respond to queries in up to a few seconds. Furthermore, several approaches for further improving performance are discussed in section 8.4.3.

Fifth, all four structural algorithms perform nearly identically when no connections are present in the input partial composition. This is especially important for consumer computing as one of the frequently used ways to create applications is to first add all the necessary components to the composition workspace, and only then define the application logic. This effect can be observed througout all experiments on the feature vector algorithms which perform nearly identically in every experiment, with the more complex representation containing connections only occasionally providing marginal benefits in accuracy and coverage. A component recommender algorithm should ideally be able to use additional connectivity information to obtain better results, but none of the proposed algorithms achieve this goal.

To conclude, evaluation results confirm the hypothesis of the presented research. However, the described approach to component recommendation has several limitations. First, in this dissertation, a simple view on component identity was adopted, where two components can be matched only if they are instances of the same exact component. While extending this definition to include components that are not identical but only similar in functionality provides further challenges in designing algorithms to address this task and evaluating them, such a broader definition would likely provide better results in some composition systems in practice. This issue is a major area for further research.

Second, scaling the approach to composition databases beyond several tens of thousands compositions also requires further research. While current compositions systems have an order of magnitude smaller databases, this can be expected to change in this decade.

Third, the usefulness of a component recommender in *Yahoo Pipes* can be questioned. With only 55 modules to choose from, Pipes users will quickly understand what each module does which certainly diminishes the usefulness of a module recommender. Therefore, such a recommender would primarily be useful to novice users. However, there are two benefits to more experienced Pipes users. First, an accurate recommender provides a faster and more user friendly way of getting required modules onto the workspace than the currently used module menus. Second, the structural approach to component recommendation provides a high degree of explainability in that users can be given an opportunity to browse through the most similar database compositions that the recommendations were based on. If users were able to find pipes that already completely solve their problem, duplication of functionality, which has been identified as a problem in a study of Pipes [131], could be significantly reduced.

Yahoo Pipes were used as the main evaluation dataset as no other similar dataset of sufficient size is publicly available.

As stated above, the results on the synthetic dataset show promise that the approach will also be effective in other composition systems. While this dissertation focuses on examples from composition systems where components are composed through their GUIs, the presented approach can be applied to any composition system where it is possible to automatically identify the basic structure of the composition as it is being developed and to component-based design environments.

Bibliography

[1] Skvorc, D., “Consumer Programming”, Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2010.

[2] Ngu, A. H. H., Carlson, M., Sheng, Q., young Paik, H., “Semantic-Based Mashup of Composite Applications”, Services Computing, IEEE Transactions on, Vol. 3, No. 1, 2010, pp. 2-15.

[3] Zang, N., Rosson, M. B., Nasser, V., “Mashups: who? what? why?”, in CHI ’08 Extended Abstracts on Human Factors in Computing Systems, 2008, pp. 3171–3176.

[4] Lin, J., Wong, J., Nichols, J., Cypher, A., Lau, T. A., “End-user programming of mashups with Vegemite”, in Proceedings of the 14th international conference on Intelligent user interfaces, ser. IUI ’09, 2009, pp. 97–106.

[5] Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M. R., “A classification framework for software component models”, Software Engineering, IEEE Transactions on, Vol. 37, No. 5, 2011, pp. 593–615.

[6] Cooney, D., Glazkov, D., “Introduction to Web Components”, W3C working draft, Jun. 2013, http://www.w3.org/TR/components-intro/.

[7] Srbljic, S., Skvorc, D., Skrobo, D., “Widget-Oriented Consumer Programming”, AUTOMATIKA - Journal for Control, Measurement, Electronics, Computing and Communications, Vol. 50, No. 3–4, 2009, pp. 252–264.

[8] Angeli, A., Battocchi, A., Roy Chowdhury, S., Rodriguez, C., Daniel, F., Casati, F., “End-User Requirements for Wisdom-Aware EUD”, in End-User Development, ser. Lecture Notes in Computer Science, 2011, Vol. 6654, pp. 245-250.

[9] Elmeleegy, H., Ivan, A., Akkiraju, R., Goodwin, R., “Mashup Advisor: A Recommendation Tool for Mashup Development”, in Web Services, 2008. ICWS ’08. IEEE International Conference on, Sept., pp. 337-344.

[10] Daniel, F., Matera, M., Weiss, M., “Next in Mashup Development: User-Created Apps on the Web”, IT Professional, Vol. 13, No. 5, 2011, pp. 22-29.

[11] Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., Kusumoto, S., “Ranking significance of software components based on use relations”, Software Engineering, IEEE Transactions on, Vol. 31, No. 3, 2005, pp. 213–225.

[12] Ye, Y., Fischer, G., “Supporting reuse by delivering task-relevant and personalized information”, in Proceedings of the 24th international conference on Software engineering, 2002, pp. 513–523.

[13] McCarey, F., Ó Cinnéide, M., Kushmerick, N., “Knowledge reuse for software reuse”, Web Intelligence and Agent Systems, Vol. 6, No. 1, 2008, pp. 59–81.

[14] Radeck, C., Lorz, A., Blichmann, G., Meißner, K., “Hybrid recommendation of composition knowledge for end user development of mashups”, in ICIW 2012, The Seventh International Conference on Internet and Web Applications and Services, 2012, pp. 30–33.

[15] Roy Chowdhury, S., Rodríguez, C., Daniel, F., Casati, F., “Baya: assisted mashup development as a service”, in Proceedings of the 21st international conference companion on World Wide Web, ser. WWW ’12 Companion, 2012, pp. 409–412.

[16] Srbljic, S., Skvorc, D., Skrobo, D., “Programming language design for event-driven service composition”, AUTOMATIKA – Journal for Control, Measurement, Electronics, Computing and Communications, Vol. 51, No. 4, 2011.

[17] Srbljic, S., Skvorc, D., Skrobo, D., “Programming Languages for End-User Personalization of Cyber-Physical Systems”, AUTOMATIKA – Journal for Control, Measurement, Electronics, Computing and Communications, Vol. 53, No. 3, 2012, pp. 294–310.

[18] Skrobo, D., “Service Composition Based on Tabular Programming”, Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2011.

[19] Mashable, “Google Play hits 1 million apps”, created in 2013, available at: http://mashable.com/2013/07/24/google-play-1-million/, accessed on 2013-12-21.

[20] The Verge, “Apple announces 1 million apps in the App Store, more than 1 billion songs played on iTunes radio”, created in 2013, available at: http://www.theverge.com/2013/10/22/4866302/apple-announces-1-million-apps-in-the-app-store, accessed on 2013-12-21.

[21] Nielsen, “A year of change and growth in US smartphones”, created in 2013, available at: http://www.nielsen.com/us/en/newswire/2012/state-of-the-appnation-%C3%A2%C2%80%C2%93-a-year-of-change-and-growth-in-u-s-smartphones.html, accessed on 2013-12-21.

[22] DigitalBuzz, “Infographic: 2013 mobile growth statistics”, created in 2013, available at: http://www.digitalbuzzblog.com/infographic-2013-mobile-growth-statistics/, accessed on 2013-12-21.

[23] Computing Research Association, “CRA 2012 Taulbee Survey”, created in 2013, available at: http://cra.org/uploads/documents/resources/crndocs/2012_taulbee_survey.pdf, accessed on 2013-12-21.

[24] Exploring Compute Science, “Computer science education statistics”, created in 2013, available at: http://www.exploringcs.org/resources/cs-statistics, accessed on 2013-12-21.

[25] Computerworld, “Programming ability is the new digital divide: Berners-Lee”, created in 2013, available at: http://www.computerworld.co.nz/article/452521/programming_ability_new_digital_divide_berners-lee/, accessed on 2013-12-21.

[26] Code.org, “Hour of Code”, created in 2013, available at: http://code.org/educate/hoc, accessed on 2013-12-21.

[27] O’Reilly, T., What is Web 2.0. O’Reilly Media, 2009, [Online]. Available: http://books.google.hr/books?id=NpEk_WFCMdIC

[28] in Web Engineering, ser. Lecture Notes in Computer Science, Gaedke, M., Grossniklaus, M., Díaz, O., Eds., 2009, Vol. 5648.

[29] Cypher, E., Halbert, D. C., Watch what I do: programming by demonstration. The MIT Press, 1993.

[30] Halbert, D. C., "Programming by example", Ph.D. dissertation, University of California, Berkeley, 1984.

[31] Lieberman, H., "Tinker: A programming by demonstration system for beginning programmers", Watch what I do: programming by demonstration, 1993, pp. 49–66.

[32] McDaniel, R. G., Myers, B. A., "Getting more out of programming-by-demonstration", in Proceedings of the SIGCHI conference on Human Factors in Computing Systems. ACM, 1999, pp. 442–449.

[33] Maulsby, D., Witten, I. H., "Watch What I Do", Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., Turransky, A., Eds., 1993, ch. Metamouse: An Instructible Agent for Programming by Demonstration, pp. 155–181.

[34] Calinon, S., Billard, A., "Active Teaching in Robot Programming by Demonstration", in Robot and Human interactive Communication, 2007. RO-MAN 2007. The 16th IEEE International Symposium on, 2007, pp. 702-707.

[35] Calinon, S., "Robot programming by demonstration", in Springer Handbook of Robotics. Springer, 2008, pp. 1371–1394.

[36] Friedrich, H., Münch, S., Dillmann, R., Bocionek, S., Sassin, M., "Robot programming by demonstration (RPD): Supporting the induction by human interaction", Machine Learning, Vol. 23, No. 2-3, 1996, pp. 163–189.

[37] Zollner, R., Asfour, T., Dillmann, R., "Programming by demonstration: Dual-arm manipulation tasks for humanoid robots", in Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, Vol. 1. IEEE, 2004, pp. 479–484.

[38] Lugarić, T., Barišić, M., Pavlić, Z., "Application of Consumer Programming and Internet-Based Technologies in Underwater Exploration", in 4th Conference on Marine Technology in memory of Academician Zlatko Winkler, 2011.

[39] Pavlić, Z., Lugarić, T., Srbljić, S., Vukić, Z., "Application of widget-based consumer programming techniques in autonomous marine vehicle control system design", in 9th IFAC Conference on Manouvering and Control of Marine Craft, 2012.

[40] Popovic, M., "Consumer program synchronization", Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2011.

[41] Delac, G., "Reliability Management of Composite Consumer Applications", Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2013.

[42] Silic, M., "Reliability Prediction of Consumer Computing Applications", Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2013.

[43] Vladimir, K., "Peer Tutoring in Consumer Computing", Ph.D. dissertation, University of Zagreb, Faculty of electrical engineering and computing, 2013.

[44] in Recommender Systems Handbook, Ricci, F., Rokach, L., Shapira, B., Kantor, P. B., Eds., 2011.

[45] Cosley, D., Lam, S. K., Albert, I., Konstan, J. A., Riedl, J., "Is Seeing Believing?: How Recommender System Interfaces Affect Users' Opinions", in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '03, 2003, pp. 585–592.

[46] Herlocker, J. L., Konstan, J. A., Terveen, L. G., Riedl, J. T., "Evaluating collaborative filtering recommender systems", ACM Transactions on Information Systems (TOIS), Vol. 22, No. 1, 2004, pp. 5–53.

[47] in The Adaptive Web, ser. Lecture Notes in Computer Science, Brusilovsky, P., Kobsa, A., Nejdl, W., Eds., 2007, Vol. 4321.

[48] Meyer, F., "Recommender systems in industrial contexts", Ph.D. dissertation, Universite de Grenoble, 2012.

[49] Sarwar, B., Karypis, G., Konstan, J., Riedl, J., "Item-based Collaborative Filtering Recommendation Algorithms", in Proceedings of the 10th International Conference on World Wide Web, ser. WWW '01, 2001, pp. 285–295.

[50] Resnick, P., Varian, H. R., "Recommender Systems", Commun. ACM, Vol. 40, No. 3, Mar. 1997, pp. 56–58.

[51] Linden, G., Smith, B., York, J., “Amazon.com recommendations: item-to-item collaborative filtering”, Internet Computing, IEEE, Vol. 7, No. 1, 2003, pp. 76-80.

[52] Parikh, N., Sundaresan, N., “Buzz-based Recommender System”, in Proceedings of the 18th International Conference on World Wide Web, ser. WWW ’09, 2009, pp. 1231–1232.

[53] Sundaresan, N., “Recommender Systems at the Long Tail”, in Proceedings of the Fifth ACM Conference on Recommender Systems, ser. RecSys ’11, 2011, pp. 1–6.

[54] Wei, K., Huang, J., Fu, S., “A Survey of E-Commerce Recommender Systems”, in Service Systems and Service Management, 2007 International Conference on, 2007, pp. 1-5.

[55] Qin, S., Menezes, R., Silaghi, M., “A Recommender System for Youtube Based on its Network of Reviewers”, in Social Computing (SocialCom), 2010 IEEE Second International Conference on, 2010, pp. 323-328.

[56] Davidson, J., Liebald, B., Liu, J., Nandy, P., Van Vleet, T., Gargi, U., Gupta, S., He, Y., Lambert, M., Livingston, B., Sampath, D., “The YouTube Video Recommendation System”, in Proceedings of the Fourth ACM Conference on Recommender Systems, ser. RecSys ’10, 2010, pp. 293–296.

[57] Zhou, R., Khemmarat, S., Gao, L., “The Impact of YouTube Recommendation System on Video Views”, in Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, ser. IMC ’10, 2010, pp. 404–410.

[58] Ahn, H., Kim, K.-j., Han, I., “Mobile advertisement recommender system using collaborative filtering: Mar-cf”, in Proceedings of the 2006 conference of the Korea society of management information systems, 2006.

[59] Broder, A. Z., “Computational Advertising and Recommender Systems”, in Proceedings of the 2008 ACM Conference on Recommender Systems, ser. RecSys ’08, 2008, pp. 1–2.

[60] Shapira, B., Recommender systems handbook. Springer, 2011.

[61] Adomavicius, G., Tuzhilin, A., “Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions”, Knowledge and Data Engineering, IEEE Transactions on, Vol. 17, No. 6, 2005, pp. 734-749.

[62] in Recommender Systems Handbook, Ricci, F., Rokach, L., Shapira, B., Kantor, P. B., Eds., 2011.

[63] Artificial Intelligence Review, Vol. 19, No. 4, 2003.

[64] Balabanović, M., Shoham, Y., "Fab: Content-based, Collaborative Recommendation", Commun. ACM, Vol. 40, No. 3, Mar. 1997, pp. 66–72.

[65] in The Adaptive Web, ser. Lecture Notes in Computer Science, Brusilovsky, P., Kobsa, A., Nejdl, W., Eds., 2007, Vol. 4321.

[66] Balabanović, M., "An Adaptive Web Page Recommendation Service", in Proceedings of the First International Conference on Autonomous Agents, ser. AGENTS '97, 1997, pp. 378–385.

[67] Krulwich, B., Burkey, C., "Learning user information interests through extraction of semantically significant phrases", in Proceedings of the AAAI spring symposium on machine learning in information access, 1996, pp. 100–112.

[68] Krulwich, B., Burkey, C., "The InfoFinder agent: learning user interests through heuristic phrase extraction", IEEE Expert, Vol. 12, No. 5, 1997, pp. 22-27.

[69] Lang, K., "Newsweeder: Learning to filter netnews", in In Proceedings of the Twelfth International Conference on Machine Learning. Morgan Kaufmann, 1995, pp. 331–339.

[70] Croft, W. B., Metzler, D., Strohman, T., Search engines: Information retrieval in practice. Addison-Wesley Reading, 2010.

[71] Rocchio, J. J., "Relevance feedback in information retrieval", 1971.

[72] Machine Learning, Vol. 1, No. 1, 1986.

[73] Grünwald, P. D., The minimum description length principle. MIT press, 2007.

[74] Su, X., Khoshgoftaar, T. M., "A Survey of Collaborative Filtering Techniques", Adv. in Artif. Intell., Vol. 2009, Jan. 2009, pp. 4:2–4:2.

[75] Journal of Computer-Aided Molecular Design, Vol. 16, No. 1, 2002.

[76] Spertus, E., Sahami, M., Buyukkokten, O., “Evaluating Similarity Measures: A Large-scale Study in the Orkut Social Network”, in Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, ser. KDD ’05, 2005, pp. 678–684.

[77] Pearl, J., Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann, 1988.

[78] Xue, G.-R., Lin, C., Yang, Q., Xi, W., Zeng, H.-J., Yu, Y., Chen, Z., “Scalable Collaborative Filtering Using Cluster-based Smoothing”, in Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ser. SIGIR ’05, 2005, pp. 114–121.

[79] Billsus, D., Pazzani, M. J., “Learning Collaborative Information Filters.”, in ICML, Vol. 98, 1998, pp. 46–54.

[80] Goldberg, D., Nichols, D., Oki, B. M., Terry, D., “Using Collaborative Filtering to Weave an Information Tapestry”, Commun. ACM, Vol. 35, No. 12, Dec. 1992, pp. 61–70.

[81] Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., Riedl, J., “GroupLens: An Open Architecture for Collaborative Filtering of Netnews”, in Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, ser. CSCW ’94, 1994, pp. 175–186.

[82] Konstan, J. A., Miller, B. N., Maltz, D., Herlocker, J. L., Gordon, L. R., Riedl, J., “GroupLens: Applying Collaborative Filtering to Usenet News”, Commun. ACM, Vol. 40, No. 3, Mar. 1997, pp. 77–87.

[83] Shardanand, U., Maes, P., “Social Information Filtering: Algorithms for Automating &Ldquo;Word of Mouth&Rdquo;”, in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI ’95, 1995, pp. 210–217.

[84] Hill, W., Stead, L., Rosenstein, M., Furnas, G., “Recommending and Evaluating Choices in a Virtual Community of Use”, in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI ’95, 1995, pp. 194–201.

[85] Bennett, J., Lanning, S., “The Netflix prize”, 2007.

[86] Koren, Y., "The BellKor solution to the Netflix Grand Prize", Netflix prize documentation, 2009.

[87] Sawers, P., "Remember Netflix's million dollar algorithm contest? Well, here's why it didn't use the winning entry", created in 2012, available at: http://tnw.co/1buxY1Z, accessed on 2013-12-27.

[88] Basu, C., Hirsh, H., Cohen, W. *et al.*, "Recommendation as classification: Using social and content-based information in recommendation", in AAAI/IAAI, 1998, pp. 714–720.

[89] Schein, A. I., Popescul, A., Ungar, L. H., Pennock, D. M., "Methods and Metrics for Cold-start Recommendations", in Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ser. SIGIR '02, 2002, pp. 253–260.

[90] Soboroff, I., Nicholas, C., "Combining content and collaboration in text filtering", in Proceedings of the IJCAI, Vol. 99, 1999, pp. 86–91.

[91] Basilico, J., Hofmann, T., "Unifying Collaborative and Content-based Filtering", in Proceedings of the Twenty-first International Conference on Machine Learning, ser. ICML '04, 2004, pp. 9–.

[92] Salter, J., Antonopoulos, N., "CinemaScreen recommender agent: combining collaborative and content-based filtering", Intelligent Systems, IEEE, Vol. 21, No. 1, 2006, pp. 35-41.

[93] Claypool, M., Gokhale, A., Miranda, T., Murnikov, P., Netes, D., Sartin, M., "Combining content-based and collaborative filters in an online newspaper", in Proceedings of ACM SIGIR workshop on recommender systems, Vol. 60. Citeseer, 1999.

[94] Billsus, D., Pazzani, M. J., "User modeling for adaptive news access", User modeling and user-adapted interaction, Vol. 10, No. 2-3, 2000, pp. 147–180.

[95] Di Lorenzo, G., Hacid, H., Paik, H.-y., Benatallah, B., "Data integration in mashups", ACM Sigmod Record, Vol. 38, No. 1, 2009, pp. 59–66.

[96] in The Smart Internet, ser. Lecture Notes in Computer Science, Chignell, M., Cordy, J., Ng, J., Yesha, Y., Eds., 2010, Vol. 6400.

[97] Grammel, L., Storey, M.-A., “An end user perspective on mashup makers”, University of Victoria Technical Report DCS-324-IR, 2008.

[98] Stolee, K. T., Elbaum, S., “Refactoring pipe-like mashups for end-user programmers”, in Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011, pp. 81–90.

[99] Simmen, D. E., Altinel, M., Markl, V., Padmanabhan, S., Singh, A., “Damia: data mashups for intranet applications”, in Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008, pp. 1171–1182.

[100] Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., Mau, L., Ng, Y.-H., Simmen, D., Singh, A., “Damia: a data mashup fabric for intranet applications”, in Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 2007, pp. 1370–1373.

[101] Ennals, R. J., Garofalakis, M. N., “MashMaker: Mashups for the Masses”, in Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’07, 2007, pp. 1116–1118.

[102] Pierce, M. E., Singh, R., Guo, Z., Marru, S., Rattadilok, P., Goyal, A., “Open Community Development for Science Gateways with Apache Rave”, in Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, ser. GCE ’11, 2011, pp. 29–36.

[103] Huynh, D. F., Karger, D. R., Miller, R. C., “Exhibit: Lightweight Structured Data Publishing”, in Proceedings of the 16th International Conference on World Wide Web, ser. WWW ’07, 2007, pp. 737–746.

[104] Morbidoni, C., Polleres, A., Tummarello, G., “Who the FOAF knows Alice? A Needed Step Toward Semantic Web Pipes.”. Citeseer.

[105] Morbidoni, C., Polleres, A., Tummarello, G., Le Phuoc, D., “Semantic web pipes”, Rapport technique, DERI, Vol. 71, 2007, pp. 108–112.

[106] in The Semantic Web: Research and Applications, ser. Lecture Notes in Computer Science, Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M., Eds., 2008, Vol. 5021.

[107] Le-Phuoc, D., Polleres, A., Hauswirth, M., Tummarello, G., Morbidoni, C., "Rapid Prototyping of Semantic Mash-ups Through Semantic Web Pipes", in Proceedings of the 18th International Conference on World Wide Web, ser. WWW '09, 2009, pp. 581–590.

[108] Wong, J., Hong, J. I., "Making Mashups with Marmite: Towards End-user Programming for the Web", in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '07, 2007, pp. 1435–1444.

[109] Tuchinda, R., Szekely, P., Knoblock, C. A., "Building Mashups by Example", in Proceedings of the 13th International Conference on Intelligent User Interfaces, ser. IUI '08, 2008, pp. 139–148.

[110] Hartmann, B., Wu, L., Collins, K., Klemmer, S. R., "Programming by a sample: rapidly creating web applications with d.mix", in Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, ser. UIST '07, 2007, pp. 241–250.

[111] Fujima, J., Lunzer, A., Hornbæk, K., Tanaka, Y., "Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access", in Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology, ser. UIST '04, 2004, pp. 175–184.

[112] Cremonesi, P., Picozzi, M., Matera, M., "A comparison of recommender systems for mashup composition", in Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on, 2012, pp. 54-58.

[113] Cremonesi, P., Koren, Y., Turrin, R., "Performance of Recommender Algorithms on Top-n Recommendation Tasks", in Proceedings of the Fourth ACM Conference on Recommender Systems, ser. RecSys '10, 2010, pp. 39–46.

[114] Agarwal, D., Chen, B.-C., "Regression-based Latent Factor Models", in Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD '09, 2009, pp. 19–28.

[115] Roy Chowdhury, S., Chudnovskyy, O., Niederhausen, M., Pietschmann, S., Sharples, P., Daniel, F., Gaedke, M., "Complementary Assistance Mechanisms for End User Mashup Composition", in Proceedings of the 22Nd International Conference on World Wide Web Companion, ser. WWW '13 Companion, 2013, pp. 269–272.

[116] The W3C SPARQL Working Group, “SPARQL 1.1 Overview - W3C Recommendation 21 March 2013”, created in 2013, available at: http://support.mozilla.org/en-US/kb/awesome-bar-find-your-bookmarks-history-and-tabs, accessed on 2013-12-19.

[117] in The Semantic Web - ISWC 2006, ser. Lecture Notes in Computer Science, Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L., Eds., 2006, Vol. 4273.

[118] Chudnovskyy, O., Gaedke, M., “Development of Web 2.0 Applications using WebComposition/Data Grid Service”, in SERVICE COMPUTATION 2010, The Second International Conferences on Advanced Service Computing, 2010, pp. 55–61.

[119] Greenshpan, O., Milo, T., Polyzotis, N., “Autocompletion for mashups”, Proceedings of the VLDB Endowment, Vol. 2, No. 1, 2009, pp. 538–549.

[120] Abiteboul, S., Greenshpan, O., Milo, T., “Modeling the mashup space”, in Proceedings of the 10th ACM workshop on Web information and data management. ACM, 2008, pp. 87–94.

[121] Mozilla, “Awesome Bar - Find your bookmarks, history and tabs when you type in the address bar”, created in 2013, available at: http://support.mozilla.org/en-US/kb/awesome-bar-find-your-bookmarks-history-and-tabs, accessed on 2013-12-16.

[122] The Chromium Project, “OmniBox”, created in 2009, available at: http://www.chromium.org/user-experience/omnibox, accessed on 2013-12-16.

[123] Ortega, R. E., Avery, J. W., Frederick, R., “Search query autocompletion”, US Patent 6,564,213. May 13 2003.

[124] Bast, H., Weber, I., “Type less, find more: fast autocompletion search with a succinct index”, in Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2006, pp. 364–371.

[125] Foster, S. R., Griswold, W. G., Lerner, S., “WitchDoctor: IDE support for real-time autocompletion of refactorings”, in Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012, pp. 222–232.

[126] Khoussainova, N., Kwon, Y., Balazinska, M., Suciu, D., “SnipSuggest: context-aware autocompletion for SQL”, Proceedings of the VLDB Endowment, Vol. 4, No. 1, 2010, pp. 22–33.

[127] Kuschke, T., Mäder, P., Rempel, P., “Recommending auto-completions for software modeling activities”, in Model-Driven Engineering Languages and Systems. Springer, 2013, pp. 170–186.

[128] Mens, T., Tourwe, T., “A survey of software refactoring”, Software Engineering, IEEE Transactions on, Vol. 30, No. 2, 2004, pp. 126-139.

[129] Fowler, M., Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.

[130] Stolee, K. T., “Analysis and transformation of pipe-like web mashups for end user programmers”, Master’s thesis, University of Nebraska at Lincoln.

[131] Stolee, K. T., Elbaum, S., Sarma, A., “Discovering how end-user programmers and their communities use public repositories: A study on Yahoo! Pipes”, Information and Software Technology, No. 0, 2012, pp. -.

[132] Riabov, A. V., Boillet, E., Feblowitz, M. D., Liu, Z., Ranganathan, A., “Wishful search: interactive composition of data mashups”, in Proceedings of the 17th international conference on World Wide Web, 2008, pp. 775–784.

[133] Riabov, A., Liu, Z., “Scalable Planning for Distributed Stream Processing Systems.”, in ICAPS, 2006, pp. 31–41.

[134] Case, A. F., “Computer-aided Software Engineering (CASE): Technology for Improving Software Development Productivity”, SIGMIS Database, Vol. 17, No. 1, Sep. 1985, pp. 35–43.

[135] “Towards assisted software engineering environments”, Information and Software Technology, Vol. 33, No. 8, 1991, pp. 581 - 593.

[136] Chikofsky, E. J., Computer-Aided Software Engineering (Case), 2nd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994.

[137] Valetto, G., Kaiser, G., “Enveloping sophisticated tools into computer-aided software engineering environments”, in Computer-Aided Software Engineering, 1995. Proceedings., Seventh International Workshop on, 1995, pp. 40-48.

[138] Robillard, M., Walker, R., Zimmermann, T., “Recommendation Systems for Software Engineering”, Software, IEEE, Vol. 27, No. 4, 2010, pp. 80-86.

[139] Singer, J., Lethbridge, T., Vinson, N., Anquetil, N., “An Examination of Software Engineering Work Practices”, in CASCON First Decade High Impact Papers, ser. CASCON ’10, 2010, pp. 174–188.

[140] Mockus, A., Herbsleb, J. D., “Expertise Browser: A Quantitative Approach to Identifying Expertise”, in Proceedings of the 24th International Conference on Software Engineering, ser. ICSE ’02, 2002, pp. 503–512.

[141] Holmes, R., Walker, R., Murphy, G., “Approximate Structural Context Matching: An Approach to Recommend Relevant Examples”, Software Engineering, IEEE Transactions on, Vol. 32, No. 12, 2006, pp. 952-970.

[142] Thummalapenta, S., Xie, T., “Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web”, in Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ser. ASE ’07, 2007, pp. 204–213.

[143] Robillard, M. P., “Topology Analysis of Software Dependencies”, ACM Trans. Softw. Eng. Methodol., Vol. 17, No. 4, Aug. 2008, pp. 18:1–18:36.

[144] Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S., “Mining version histories to guide software changes”, Software Engineering, IEEE Transactions on, Vol. 31, No. 6, 2005, pp. 429-445.

[145] Tsunoda, M., Kakimoto, T., Ohsugi, N., Monden, A., Matsumoto, K.-i., “Javawock: A Java Class Recommender System Based on Collaborative Filtering”, in Proc. 17th Intl. Conf. Softw. Eng. and Knowledge Eng, 2005, pp. 491–497.

[146] Mehandjiev, N., Lecue, F., Wajid, U., Namoun, A., “Assisted Service Composition for End Users”, in Web Services (ECOWS), 2010 IEEE 8th European Conference on, 2010, pp. 131-138.

[147] Landauer, T. K., Foltz, P. W., Laham, D., “An introduction to latent semantic analysis”, Discourse processes, Vol. 25, No. 2-3, 1998, pp. 259–284.

[148] Hull, C. L., “Simple trial and error learning: A study in psychological theory.”, Psychological Review, Vol. 37, No. 3, 1930, p. 241.

[149] Starch, D., “A demonstration of the trial and error method of learning.”, Psychological Bulletin, Vol. 7, No. 1, 1910, p. 20.

[150] Simon, H. A., Simon, P. A., “Trial and error search in solving difficult problems: Evidence from the game of chess”, Behavioral Science, Vol. 7, No. 4, 1962, pp. 425–429.

[151] Durkin, H. E., “Trial and error, gradual analysis, and sudden reorganization. An experimental study of problem solving.”, Archives of Psychology (Columbia University), 1937.

[152] Cass, A. G., Fernandes, C. S. T., Polidore, A., “An Empirical Evaluation of Undo Mechanisms”, in Proceedings of the 4th Nordic Conference on Human-computer Interaction: Changing Roles, ser. NordiCHI ’06, 2006, pp. 19–27.

[153] Brown, A. B., Patterson, D. A., “Undo for Operators: Building an Undoable E-mail Store.”, in USENIX Annual Technical Conference, General Track, 2003, pp. 1–14.

[154] Kleinberg, J. M., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A. S., “The web as a graph: Measurements, models, and methods”, in Computing and combinatorics. Springer, 1999, pp. 1–17.

[155] Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tomkins, A., Upfal, E., “Stochastic models for the web graph”, in Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. IEEE, 2000, pp. 57–65.

[156] Drinea, E., Enachescu, M., Mitzenmacher, M., “Variations on random graph models for the web”, Preprint, 2001.

[157] Page, L., Brin, S., Motwani, R., Winograd, T., “The PageRank citation ranking: bringing order to the web.”, 1999.

[158] Newman, M. E., Watts, D. J., Strogatz, S. H., “Random graph models of social networks”, Proceedings of the National Academy of Sciences of the United States of America, Vol. 99, No. Suppl 1, 2002, pp. 2566–2572.

[159] Carrington, P. J., Scott, J., Wasserman, S., Models and methods in social network analysis. Cambridge university press, 2005.

[160] Goodreau, S. M., Kitts, J. A., Morris, M., "Birds of a feather, or friend of a friend? using exponential random graph models to investigate adolescent social networks*", Demography, Vol. 46, No. 1, 2009, pp. 103–125.

[161] Liu, J., Lee, Y. T., "Graph-based method for face identification from a single 2D line drawing", Pattern Analysis and Machine Intelligence, IEEE Transactions on, Vol. 23, No. 10, 2001, pp. 1106–1119.

[162] Shapiro, L. G., Haralick, R. M., "Structural descriptions and inexact matching", Pattern Analysis and Machine Intelligence, IEEE Transactions on, No. 5, 1981, pp. 504–519.

[163] Huan, J., Wang, W., Prins, J., "Efficient mining of frequent subgraphs in the presence of isomorphism", in Data Mining, 2003. ICDM 2003. Third IEEE International Conference on, Nov., pp. 549-552.

[164] Yan, X., Yu, P. S., Han, J., "Substructure similarity search in graph databases", in Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ser. SIGMOD '05, 2005, pp. 766–777.

[165] Cunningham, P., "A Taxonomy of Similarity Mechanisms for Case-Based Reasoning", Knowledge and Data Engineering, IEEE Transactions on, Vol. 21, No. 11, Nov., pp. 1532-1543.

[166] Petrovic, S., Kendall, G., Yang, Y., "A Tabu Search Approach for Graph-Structured Case Retrieval", in Proceedings of the Starting Artificial Intelligence Researchers Symposium, 55-64, IOS. Press, 2002, pp. 55–64.

[167] Champin, P.-A., Solnon, C., "Measuring the Similarity of Labeled Graphs", in Case-Based Reasoning Research and Development, ser. Lecture Notes in Computer Science, Ashley, K., Bridge, D., Eds., 2003, Vol. 2689, pp. 80-95.

[168] Sanders, K., Kettler, B., Hendler, J., "The case for graph-structured representations", in Case-Based Reasoning Research and Development, ser. Lecture Notes in Computer Science, Leake, D., Plaza, E., Eds. Springer Berlin Heidelberg, 1997, Vol. 1266, pp. 245-254.

[169] Garey, M. R., Johnson, D. S., Computers and Intractability: A Guide to the Theory of NP-Completeness, 1979.

[170] Zeng, Z., Tung, A. K. H., Wang, J., Feng, J., Zhou, L., "Comparing stars: on approximating graph edit distance", Proc. VLDB Endow., Vol. 2, No. 1, Aug. 2009, pp. 25–36.

[171] Leiserson, C. E., Rivest, R. L., Stein, C., Cormen, T. H., Introduction to algorithms. The MIT press, 2001.

[172] Santos, E., Lins, L., Ahrens, J., Freire, J., Silva, C., "A First Study on Clustering Collections of Workflow Graphs", in Provenance and Annotation of Data and Processes, ser. Lecture Notes in Computer Science, Freire, J., Koop, D., Moreau, L., Eds., 2008, Vol. 5272, pp. 160-173.

[173] Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., Tarjan, R. E., "Time bounds for selection", Journal of Computer and System Sciences, Vol. 7, No. 4, 1973, pp. 448 - 461.

[174] Kruskal, J. B., "On the shortest spanning subtree of a graph and the traveling salesman problem", Proceedings of the American Mathematical society, Vol. 7, No. 1, 1956, pp. 48–50.

[175] Yu, S., Woodard, C. J., "Innovation in the Programmable Web: Characterizing the Mashup Ecosystem", in Service-Oriented Computing—ICSOC 2008 Workshops. Springer, 2009, pp. 136–147.

[176] Goutte, C., Gaussier, E., "A probabilistic interpretation of precision, recall and F-score, with implication for evaluation", in Advances in Information Retrieval. Springer, 2005, pp. 345–359.

[177] Raghavan, V., Bollmann, P., Jung, G. S., "A critical investigation of recall and precision as measures of retrieval system performance", ACM Transactions on Information Systems (TOIS), Vol. 7, No. 3, 1989, pp. 205–229.

[178] Neuhaus, M., Bunke, H., "A probabilistic approach to learning costs for graph edit distance", in Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on, Vol. 3. IEEE, 2004, pp. 389–393.

[179] Neuhaus, M., Bunke, H., “Automatic learning of cost functions for graph edit distance”, Information Sciences, Vol. 177, No. 1, 2007, pp. 239–247.

[180] Nah, F. F.-H., “A study on tolerable waiting time: how long are Web users willing to wait?”, Behaviour & Information Technology, Vol. 23, No. 3, 2004, pp. 153-163.

[181] Shneiderman, B., “Response time and display rate in human performance with computers”, ACM Comput. Surv., Vol. 16, No. 3, Sep. 1984, pp. 265–285.

[182] Nielsen, J., Usability Engineering. Morgan Kaufmann, 1994.

[183] Johnson, C., Dunlop, M. D., “Subjectivity and notions of time and value in interactive information retrieval”, Interacting with computers, Vol. 10, No. 1, 1998, pp. 67–75.

[184] Myers, B. A., “The importance of percent-done progress indicators for computer-human interfaces”, in ACM SIGCHI Bulletin, Vol. 16, No. 4. ACM, 1985, pp. 11–17.

[185] Galletta, D. F., Henry, R., McCoy, S., Polak, P., “Web Site Delays: How Tolerant are Users?”, Journal of the Association for Information Systems, Vol. 5, No. 1, 2004, pp. 1–28.

[186] Bouch, A., Kuchinsky, A., Bhatti, N., “Quality is in the eye of the beholder: meeting users’ requirements for Internet quality of service”, in Proceedings of the SIGCHI conference on Human factors in computing systems. ACM, 2000, pp. 297–304.

[187] Lederer, A. L., Maupin, D. J., Sena, M. P., Zhuang, Y., “The role of ease of use, usefulness and attitude in the prediction of World Wide Web usage”, in Proceedings of the 1998 ACM SIGCPR conference on Computer personnel research. ACM, 1998, pp. 195–204.

List of Figures

List of Tables

List of Algorithms

Biography

Ivan Budiselić was born in 1985 in Zagreb. He graduated from the University of Zagreb, Faculty of Electrical Engineering and Computing (UniZg FER) in March of 2008 under the mentorship of Professor Siniša Srbljić, PhD, with the thesis "*Accessing Sensor Network Services with the Session Initiation Protocol*". For his overall student success, he was awarded the "Josip Lončar" bronze medal as one of the two best students in the *Computer Science* class of 2008 with a 5.0 GPA.

In the summer of 2008, he was employed by the Department of Electronics, Microelectronics, Intelligent and Computer Systems at UniZg FER as a research and teaching assistant. His research has been focused on the areas of consumer computing, protocol integration and the World Wide Web, and he has published the results of this research at international conferences and in A and B category journals.

List of Publications

Journal Papers

1. Delač, G., Budiselić, I., Žužak, I., Skuliber, I., Štefanec, T. "A Methodology for SIP and SOAP Integration Using Application-Specific Protocol Conversion", ACM Transactions on the Web, Vol. 6, No. 4, November 2012., pp. 15-28.

2. Žužak, I., Budiselić, I., Delač, G., "A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems", Journal of web engineering, Vol. 10, No. 4, 2011., pp. 353-390.

3. Žužak, I., Budiselić, I., Delač, G., "Formal Modeling of RESTful Systems Using Finite-State Machines", Lecture notes in computer science (Web Engineering, 11th International Conference, ICWE 2011), Vol. 6757, 2011., pp. 346-360.

Conference Papers

1. Budiselić, I., Žužak, I., Benc, I., “Application Middleware for convergence of IP Multimedia system and Web Services”, Proceedings of the 33rd international convention on information and communication technology, electronics and microelectronics (MIPRO 2010), 2010., pp. 275-280.

2. Budiselić, I., Srbljić, S., Popović, M., “RegExpert: A Tool for Visualization of Regular Expressions”, Proceedings of the International Conference on “Computer as a Tool”, EUROCON 2007., pp. 1-4.

3. Žužak, I., Ivanković, M., Budiselić, I., “Cross-context web browser communication with unified communication models and context types”, Proceedings of the 34th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2012., pp. 690-695.

Other Papers

1. Žužak, I., Ivanković, M., Budiselić, I., “A Classification Framework for Web Browser Cross- Context Communication”, scientific paper published on *arXiv.org*, 2011.

Životopis

Ivan Budiselić rođen je 1985. godine u Zagrebu. Diplomirao je na Fakultetu elektrotehnike i računarstva u ožujku 2008. godine pod vodstvom prof. dr. sc. Siniše Srbljića na temi "*Sustav pristupa uslugama mreže osjetila protokolom uspostave sjednice*". Za cjelokupni uspjeh studija, nagrađen je brončanom plaketom "Josip Lončar" kao jedan od dvojice najboljih studenata smjera Računarstvo s prosjekom ocjena 5,0.

U ljeto 2008. godine zaposlio se na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave FER-a kao znanstveni novak. Provodio je istraživanja u području potrošačkog računarstva, integracije protokola i World Wide Weba, a rezultate istraživanja objavio je na međunarodnim konferencijama i u znanstvenim časopisima A i B kategorije.