University of Glasgow | School of Computing Science

Package Recommendation Engine

Keir Alexander Smith

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 15, 2015

Abstract

This paper covers the construction of a basic recommendation engine for operating system packages, specifically for use with the DNF package manager. The tool should allow users to discover useful packages through an easy to use command line interface.

Contents

Chapter 1

Introduction

1.1 Problem overview

In a modern operating system there exist many packages for end users to install and this collection grows every day[1]. Finding useful packages to install can be a laborious task, often involving the use of online search to track down the package the user has been looking for, if it exists. In a series of informal interviews, one user quoted: *"Sometimes, I'll use Google as it has a more intelligent understanding of what I might be looking for and can recommend packages that aren't exact searches"*.
Furthermore, there exists little support for installing packages commonly installed concurrently. Debian offers a 'recommended' field in a package's meta data which the author can set manually, but is often under used or quickly becomes out of date.
For example when a user who has vim[1] installed then installs JDK[2], the user may not be aware of the existence of a Java plugin for vim which could be extremely useful.

1.2 Motivation

Package management is an interesting and very useful tool for many users, however the basic implementation has been static for many years. With the addition of a package recommendation system, we could see a decrease in users having to use search engines to look for packages they should be able to discover easily on command line or local GUI interaction.
Furthermore if this project can be used as a base to look into dynamic package grouping and weighting of relationships between packages. Hopefully this could lead to some very interesting use cases and provide a lot of data for analysis.

1.3 Aims

This project aims to attempt to address the issues discussed above. Primarily the problem of finding new packages by offering a powerful recommendation system for users to discover packages.
Consider a user on a Linux machine, looking for any new useful developer tools to help their work flow. Using an internet search engine returns a massive set of results with various levels of relevance, this project aims to

[1] A well used and long standing text editor for Linux. See http://www.vim.org/about.php
[2] Java Development Kit. See http://www.oracle.com/technetwork/java/javase/overview/index.html

supply that user with a command line interface where they can ask for a recommendation based on package of their choice. In an example scenario our user asks for a recommendation based off of the Java Development Kit package and is returned with a list of useful debugging tools and plugins which they weren't aware of previously.

1.4 Report outline

This report will be structured as the following:

- Background - A look into the research undertaken for this project in order to design the system
- Design - A collection of design documents and details of how the system will be implemented and operated by the end user
- Evaluation - Discussion of how useful the system is as well as how well designed it may be
- Conclusion - A final chapter to wrap up the report as a whole and discuss future potential work

Chapter 2

Background

A package manager[1] allows users to search for, install and update packages containing useful programs. For many years Unix has relied on package managers to allow easy management of tools and underlying applications. However in recent times, as package numbers increase and the ease of search engines becomes more prominent, searching using a command line tool has become less prevalent. Unless a user knows exactly what they want, often times they will resort to a internet search engine to find new packages. One user in an informal interview quoted the following: *"I generally know the name of the program I want, so 'pacman -Ss name' to find the package name"*

2.1 Modern Graphical Package Managers

A more modern solution to this problem is the use of Graphical User Interfaces (GUIs)[2] to abstract the annoyance of searching on command line away from the user. However this requires that the user is running a system with graphical output, a luxury which is often not found when running Virtual Machines (VMs) or using Secure Shell (SSH).

2.2 NuGet Concierge

NuGet[4] is a package manager for Microsoft's .net framework. It has seen large growth since its introduction and has expanded greatly to cover many tools. In 2013 NuGet introduced a new service called Concierge[3] which allowed users to upload their project metadata and get a list of recommended packages back from the service.
Concierge works by giving each package a popularity score and then giving each package pair a bi-directional pairing weighting, shown in Figure 2.1. It then expects a project.conf file from the user, which it parses and uses the list of required packages to recommend a set of packages from the graph.
Concierge was build by a group of Microsoft interns in order to see what they could do with the growing user and package base. However it seems that, as of writing, the project has been abandoned, with the last commit to their GitHub[3] repository in 2013.

[1] Examples include dpkg for Debian (http://linux.die.net/man/1/dpkg), YUM for Red Hat Linux (http://yum.baseurl.org/) and pacman for Arch Linux (https://www.archlinux.org/pacman/)

[2] For example the Ubuntu Software Center shown in Figure 2.1.

[3] https://github.com/NuGet/Concierge

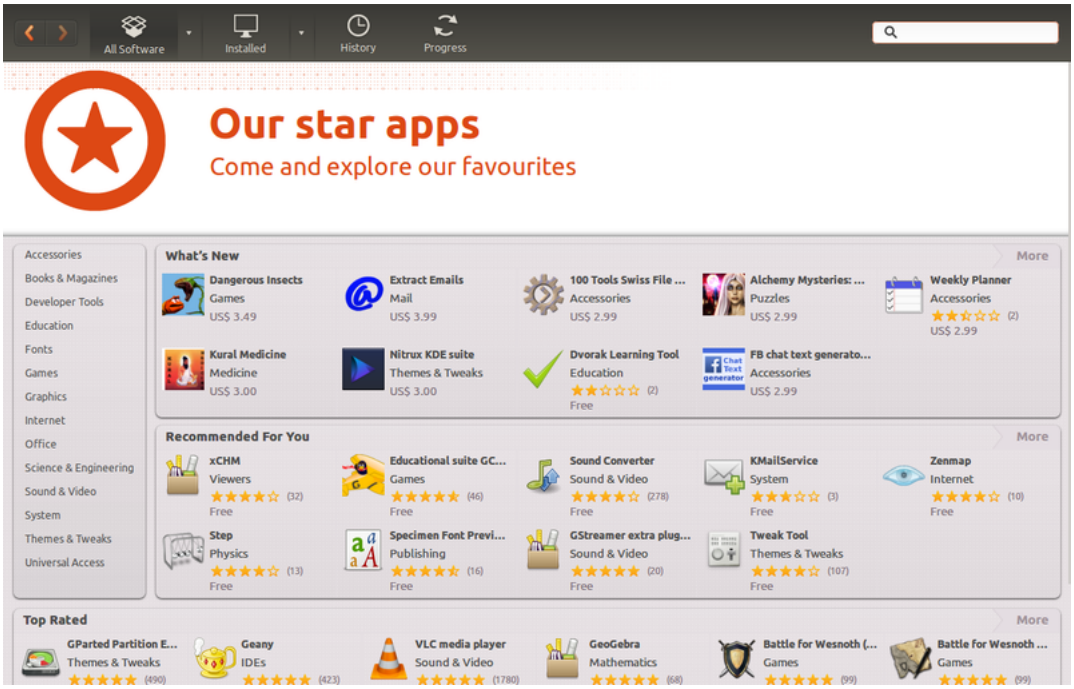

Figure 2.1: Ubuntu Software Center

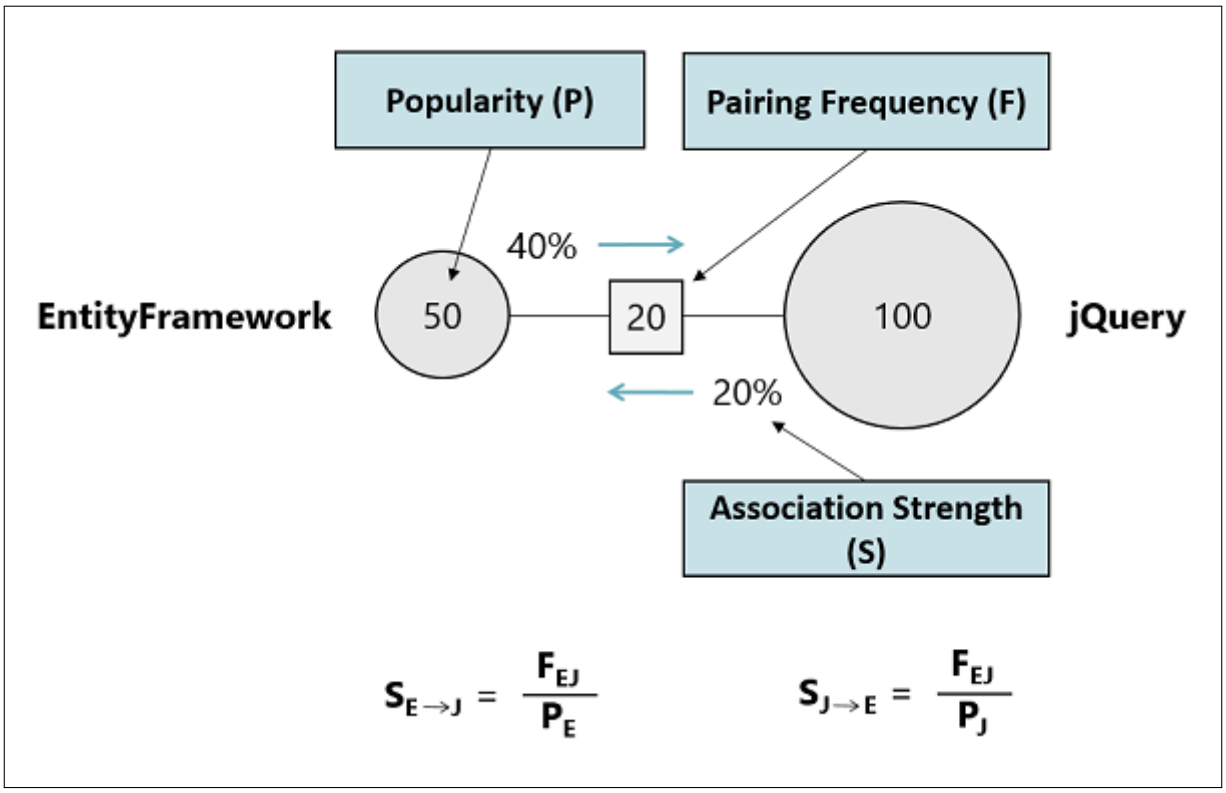

Figure 2.2: NuGet Concierge's internal structure

2.3 DNF

My project aims to supply similar recommendation functionality to users of DNF[4], Fedora 21's new package manager. DNF offers better plugin support over dpkg and it allows plugins to be easily added by simply dropping a Python file into a directory. This allows DNF to be extended easily with little hardship from the user. Building in functionality into DNF is exactly the behaviour this project aims to provide to the end user.

[4] http://fedoraproject.org/wiki/Features/DNF

Chapter 3

Design

This system comes in two parts, a client side plugin for DNF which the user installs by dropping a single Python script into the correct directory. Also a server side database to store user's installed packages, anonymously, and provide data for recommendations. See Figure 3.1
Each of these components will be discussed in their own sections.

3.1 Recommend Plugin

The Recommend plugin has two key features:

- Request recommendation from server
- Upload user's installed packages anonymously

Both functions require a connection to the back end database, a connection over the internet is assumed.
The plugin needed to be easy to use, otherwise the user will resort to a simple web search. With this in mind it was designed with two clear commands. See Figure 3.2

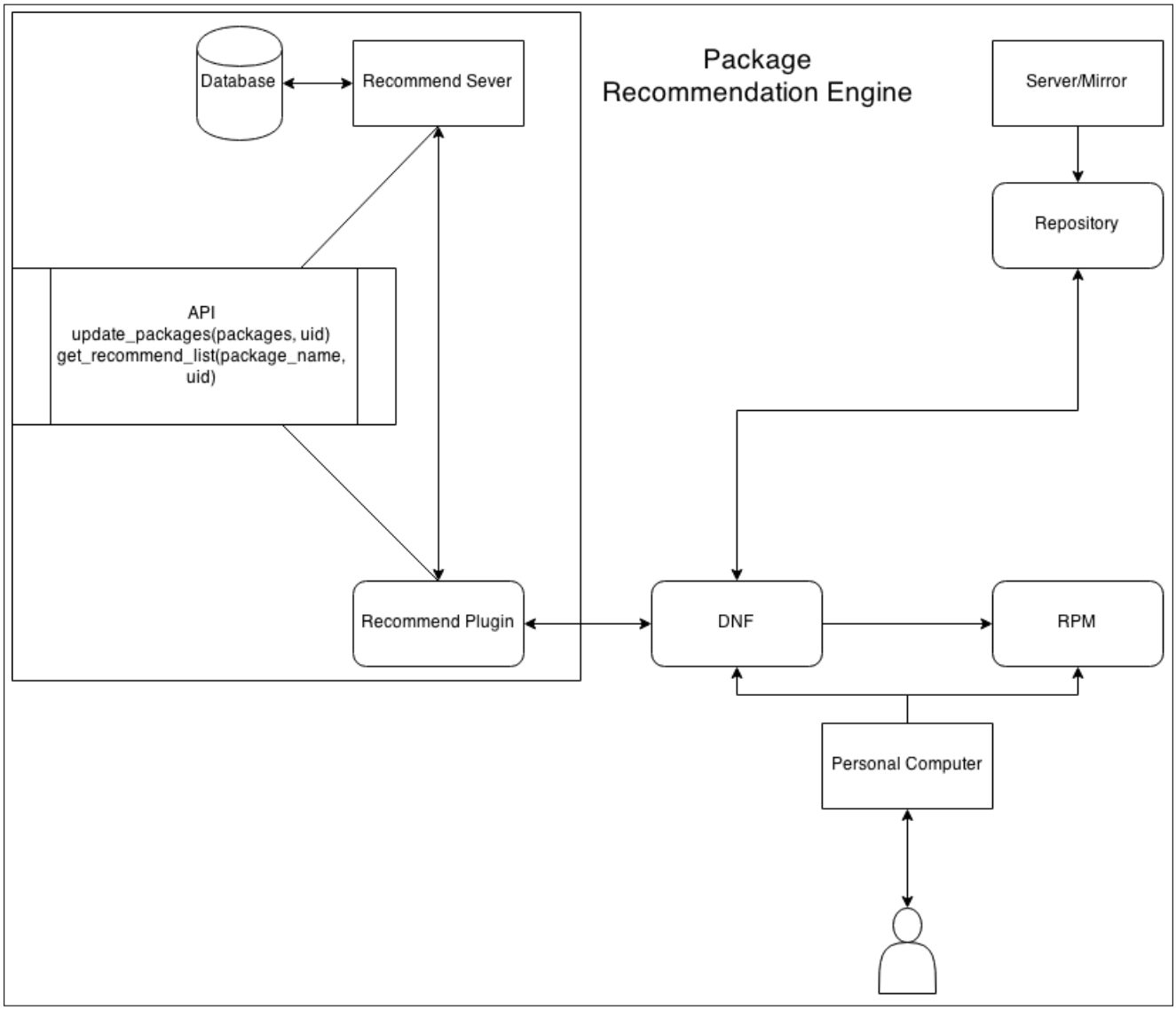

Figure 3.1: System Diagram

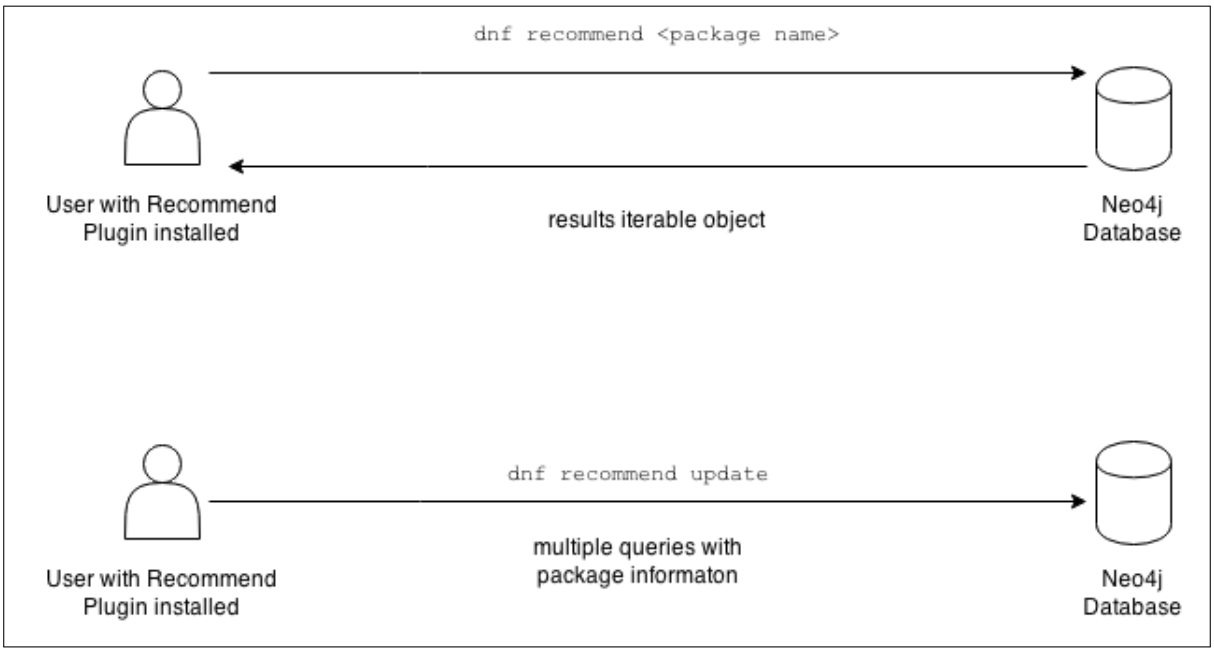

Figure 3.2: Diagram showing the exchanges between the server and client

3.2 Database

The project requires a back end data storage module. Immediately a database comes to mind for a long term, concurrent access method of storing and returning large volumes of data. Below the various potential choices of database are discussed.

3.2.1 Selection

Neo4j was selected as the database of choice for this project. There are several key reasons for this:

mySQL	**Neo4j**
Path finding is difficult	Path finding optimised
Many to Many relations expensive	Many to Many relations light weight
Expensive joins	Rapid pattern matching
Designed to work on one machine	Designed to scale across machines

With these point in mind, it seemed the logical choice to make use of a graph database like Neo4j for the purposes of the system.

3.2.2 Internal Structure

Internally the structure of the graph can be seen in Figure 3.3 Users are stored as unique nodes within the graph, each with a unique user ID attached to them. This allows us to generate their user id when they first upload their package information, the ID generated is entirely random so the user remained anonymous.
Packages are all uniquely stored with their name as their unique identifier, this means package versions are not considered when making recommendations.

Figure 3.3: Diagram showing the internal structure of the database

3.2.3 Keeping Up to Date

As users install and un-install packages the user must update the database with their new state. If the database is not kept up to date it may run into problems making recommendations of deprecated packages or not recommending new packages.
Users may not willing go out their way to update regularly, therefore the following methods were considered to allow users to keep up to date:

- Every time the user installs/un-installs a package, automatically update
- On DNF initialisation prompt the user to update
- Update automatically in set periods

3.2.4 Security

Security was a serious consideration during design, since malicious users could easily tamper with our data. The resulting decision was to ignore security for the scope of this project, however points to be considered will be listed below.

- Users could create many fake users with potentially malicious implications
- Users could run arbitrary cypher scripts on the database server
- Users could upload users with fake package information
- Users could potentially overload server capacity via a Denial of Service[1] attack.

[1] See http://searchsoftwarequality.techtarget.com/definition/denial-of-service

Chapter 4

Implementation

In regards to development environment, all work took place using a GitHub repository with all the Python scripts kept up to date there. Locally the Recommend plugin was kept in the Git repository folder which was then sym linked to the DNF plugin folder. This enabled work and change tracking to take place without disturbing work flow to test the plugin.
With the design settled, work began with a 'Wizard of Oz'[2] style mock up, where the user could use the command line interface as if the system were complete, however anything they got back was simply place holder.

4.1 First Pass Mock UI

For this initial development two features needed to be implemented. A command to push a list of installed packages to server and another to request a recommendation from the server.
DNF allows a developer to hook into its core functionality by extending classes, which allows new commands to be written and functionality added in a single Python script. Figure 4.1 and 4.2 shows two classes extending DNF plugin and command classes respectively.

4.2 Back End Implementation

Now that some form of client side had been written, the back end could be implemented. This turned into a simple case of installing and running a Neo4j instance on a local machine to test scripts.
The initial script written, shown in listing 4.3, read a list of packages from a text file in the format which DNF dumps them and then created each on the database and tied them to a user with a fake ID.
With that was in place, scripts, shown in listing 4.5, to find sets of recommendations between packages can be written and tested.

```
1 class Recommend(dnf.Plugin):
2
3     #Init Plugin
4         #Attempt to load user's ID, if they don't have one generate a new one
5         #Register Recommend Comamnd
6
7     #Hook for grabbing installed packages
```

Listing 4.1: Recommend Class

```
class RecommendCommand(dnf.cli.Command):
    #Init Recommend Command

    #Define a function to grab a recommend list
        #Connect to the graph
        #Set up search string using ID and packge the user wants
        #Contact the server and run the query
        #Package the returned data into a counter
        #Print the 10 most occuring packages

    #Define a function to build this user in the graph
        #Connect to the graph
        #Grab user ID
        #Make this user on the graph if not already present, and store the node locally
        #For each package this user has installed A
        #Add that package to the graph if not already there
            #Add a INSTALLED relationship between user and package

    #Define the run function
        #If we get more than one argument, tell the user
        #else if we get update, update the graph
        #else if we get a package name, query the graph
        #else catch all to get any invalid input
```

Listing 4.2: Recommend Command Class

See Figures 4.1 through 4.3 for graphic examples of how the Neo4j stores user and package information.

4.3 Bringing it together

With the client side UI already written and tested, it was a simple case of changing a constant defining the server address to the operational database which was running on local host.

4.4 Testing

In order to test this system, two scripts were written in Python to fill the database with both real and mock data. Several small Cypher scripts were also written to test that the database was working internally.
Listings 4.3 and 4.4 show the Python scripts for entering data.
Using a database prepared with data from either of these scripts, a series of cypher queries were run to ensure relationships, users and packages were being correctly translated. See Listings 4.5 for these test queries.
The system was only tested updating users once due to time constraints, and is therefore unreliable.

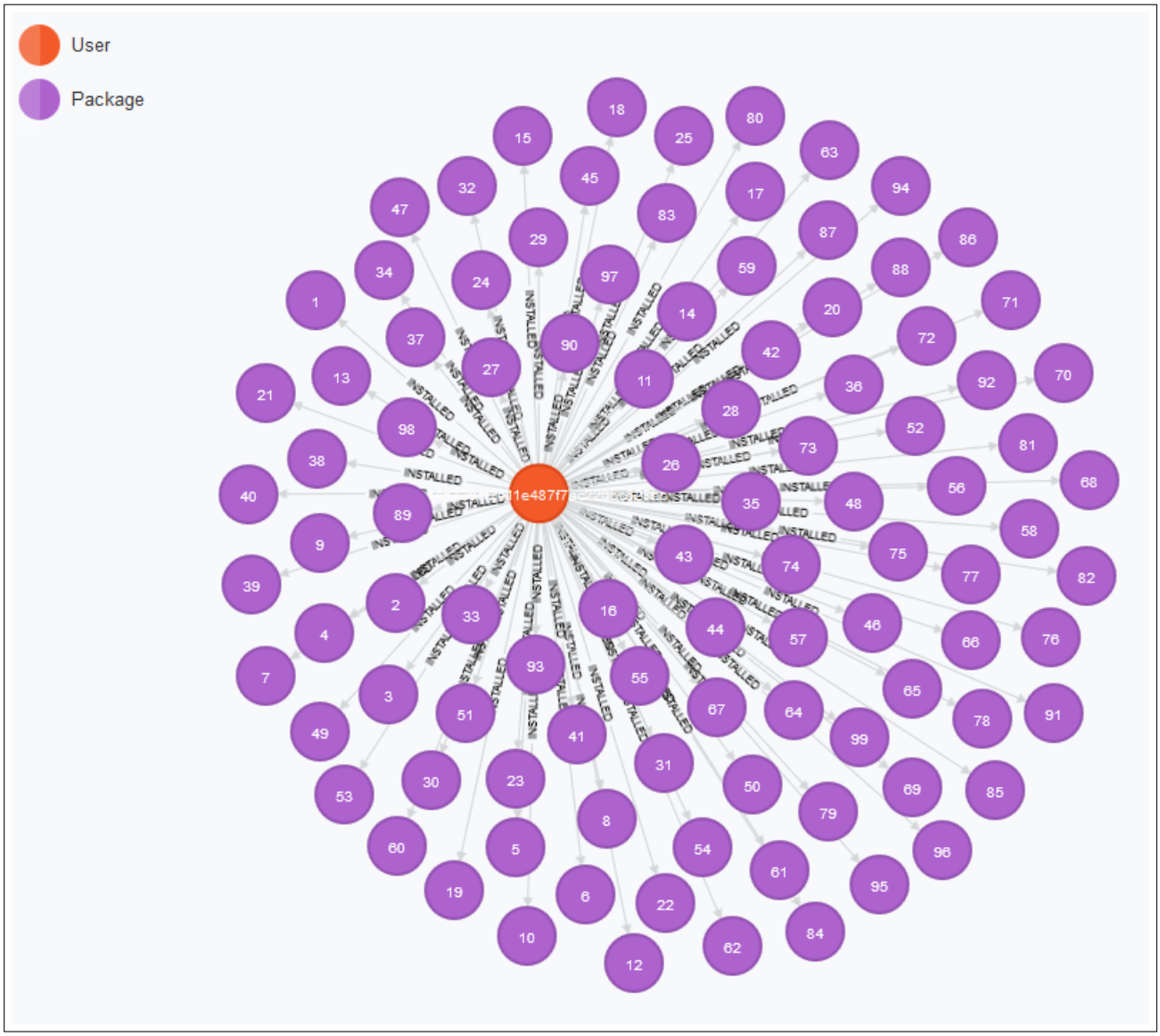

Figure 4.1: Example of a single real user

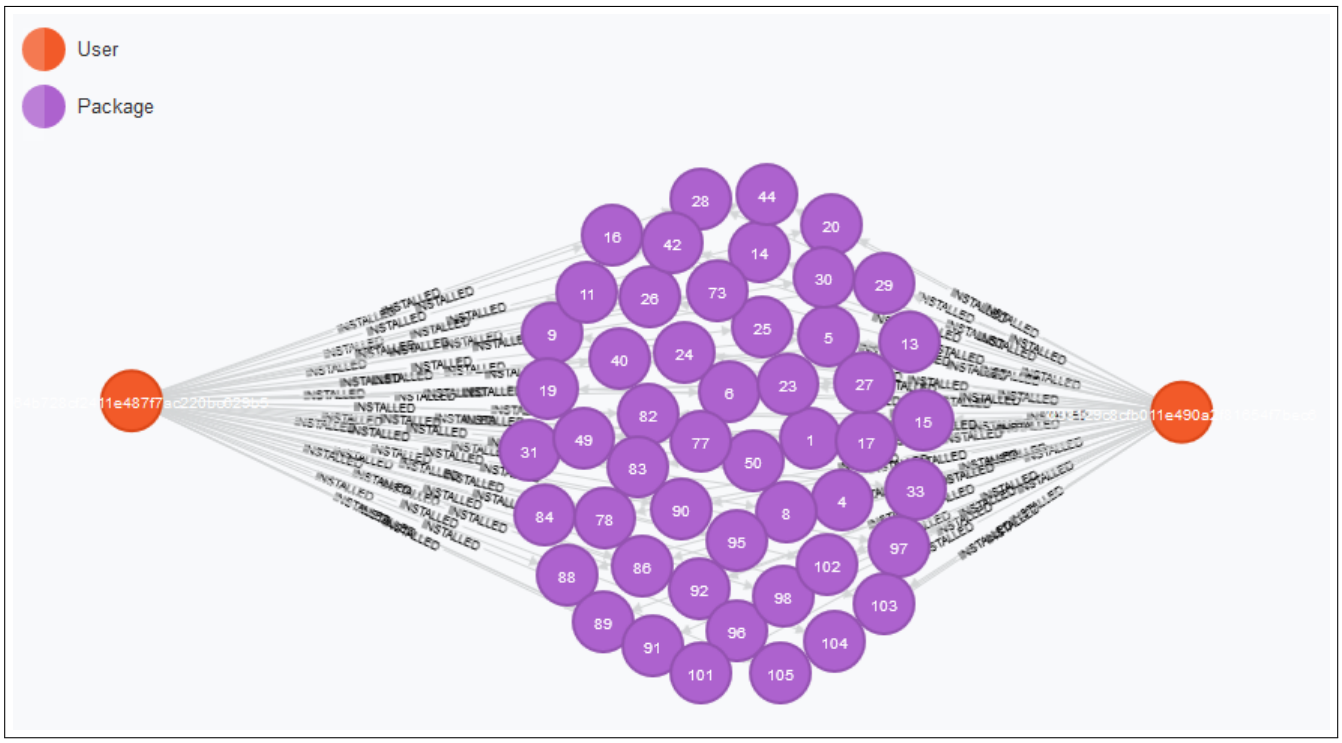

Figure 4.2: Example of two real users sharing packages

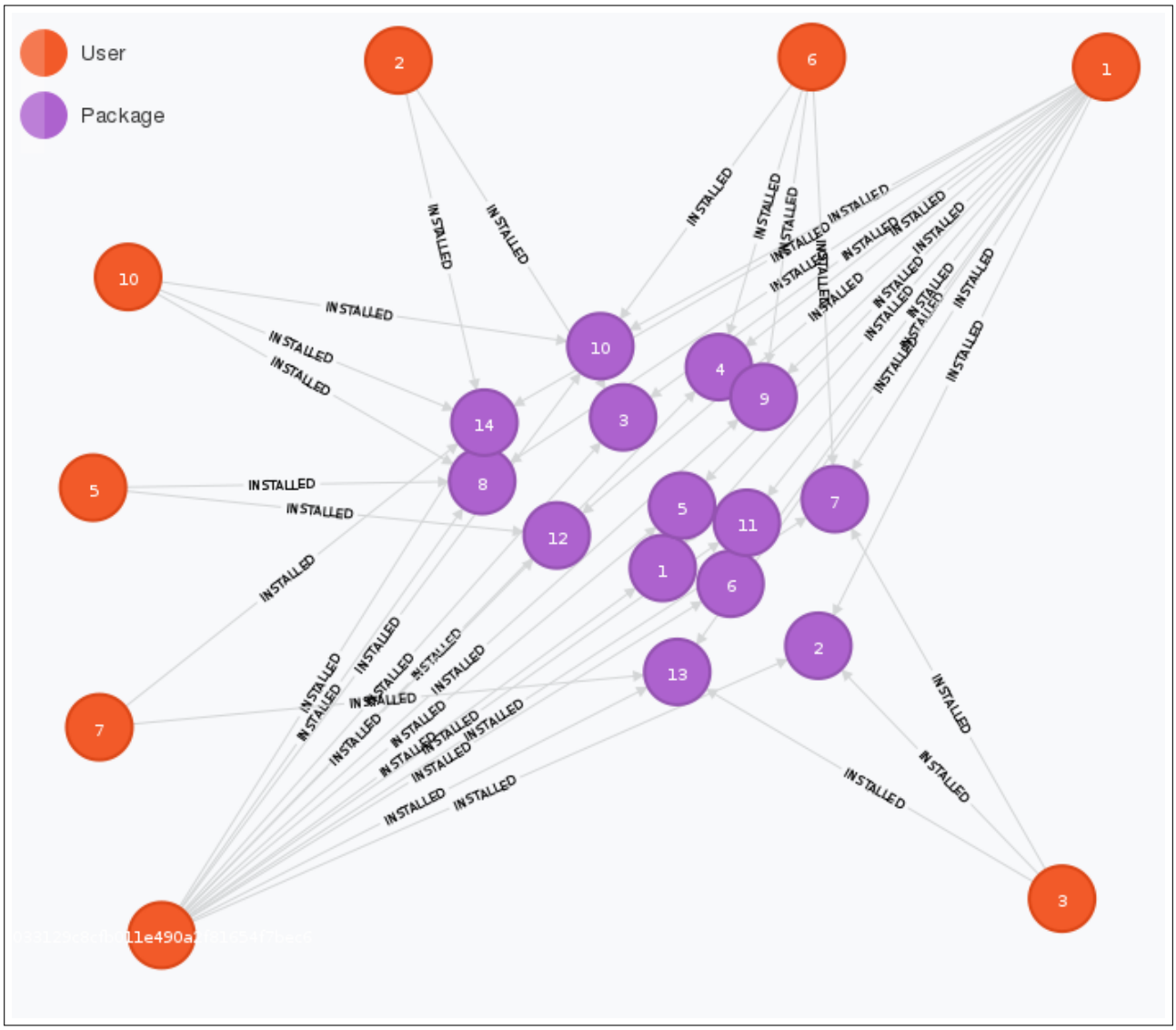

Figure 4.3: Example of 8 mock users sharing various packages

```
#installed.txt is a straight dump from dnf list installed piped to a text file
list_file = open("installed.txt", "r")

packagelist = []

for line in list_file:
    packagelist = packagelist + [line.split('.')[0]]

list_file.close()

graph = Graph()

user_list = graph.merge("User", "id", 1)

for user in user_list:
 this_pc = user

for package in packagelist:
    nodes = graph.merge("Package", "name", package)
    for node in nodes:
        relationship = Relationship(this_pc, "INSTALLED", node)
        graph.create_unique(relationship)
```

Listing 4.3: Fills the graph with real data

```
#packages.txt is a straight dump of dnf list  piped to a file
list_file = open("packages.txt", "r")

packagelist = []

for line in list_file:
    packagelist = packagelist + [line.split('.')[0]]

list_file.close()

#Cut the first three lines since they are junk from stdout
packagelist = packagelist[3:]

graph = Graph()

cycle = int(raw_input("Number of users: "))

for i in range(1, cycle + 1):
    print "Commiting packages for user " + str(i)
    user_list = graph.merge("User", "id", i)

    for user in user_list:
        this_pc = user

    for package in packagelist:
        if r.randint(0, 10) < 1:
            nodes = graph.merge("Package", "name", package)
            for node in nodes:
                relationship = Relationship(this_pc, "INSTALLED", node)
                graph.create_unique(relationship)
```

Listing 4.4: Fills the graph with mock data

```
###This query will return the first user added to the system and the first 99 packages he installed###
MATCH n RETURN n LIMIT 100

###This query will find the first ~33 sets of packages with more than one user having installed it###
MATCH (u:User)--(p:Package)--(u2:User) RETURN u, p, u2 LIMIT 100

###This query will return 2500 packages recommened for this user 1 (a mock user) for the Yum package
    ###
MATCH n--u WHERE u.name = "yum" AND n.id = "1"  MATCH u--n1 WHERE n1.id <> n.id MATCH n1--u1 WHERE u1.
    name <> "yum" AND NOT u1--n RETURN u1 LIMIT 2500

###This query will return 100 packages attached to user X where X is their user id###
MATCH (u:User {id=x})--(p:Package) RETURN u, p LIMIT 100
```

Listing 4.5: A collection of test Cypher queries

Chapter 5

Evaluation

All evaluation was based on a series of short, informal interviews with 5 users.
Majority of users found the concept appealing and were interested to use the system. One user in particular could relate well to the problem.
In using the system, no users had trouble with the user interface, but majority commented on the lack of help and the formatting of the output. One user said *"It's concise, but it just needs the format tidied up so it is easier to read"*.
All users asked how reliable the recommendations were, as they seemed unsatisfied with the results they were given. Two users commented specifically on how their input to the recommend command didn't seem to make any difference to the output of the recommendations.
Four of the five users said they wouldn't use it themselves, with the last user saying they would use the system, but only if it included more specific filter options.
Overall the evaluation has shown the system will need work before being used in a live environment, however it does show promise.

Chapter 6

Conclusion

6.1 Summary

To conclude, the system as a whole was written using a series of small Python scripts, with the main script being the DNF plugin. A Neo4j instance is running on a remote server allowing users to use the system live.
The final system met the minimum requirements. The user is able to upload their installed packages to the database and then ask for a recommendation.
The idea is well received, with users overall agreeing it is an interesting avenue to explore. However many who tried the application were weary of the limited scope of getting recommendations and most users stated they would only use the system if it had finer grain recommendation control.

6.2 Future Work

There exists a lot of scope for this project to expand. This section will discuss two interesting areas for potential future work.

6.2.1 Dynamic Package Groups

As a potential future addition, commonly installed package groups could be identified on the back end, which would be pushed forward to end users, allowing them to quickly install a complete set of tools.
An example of this would be web developers, who commonly install at least the three main browsers[1] and any debugging/developer tools associated with them.[2]
This kind of powerful inferencing can be done using a graph database made to look for groups like this. If it's possible to then analyse the group and identify what it is, this would give users a lot of power when installing packages clusters. Currently Debian implements a static set of group meta packages, but these are user defined and not extensive.

[1] Google Chrome, Mozilla Firefox and Safari
[2] For example Firebug

6.2.2 Weighted Relationships

Weighted relationships is something which NUGET Concierge used to make its recommendations. It's a one directional value assigned between two package nodes to determine how likely it is that one is installed on the same system with the other.
At its core, it's very simple, however we could go lengths to make this weighting more meaningful, perhaps through the use of labels or special cases we can identify when best to include packages in a recommendation. For example it may be the case that package X is usually only installed when package Y is installed and Z isn't. A weighted relationship would not identify this as it would only recommend the most common. By tagging the relationships we can ensure the correct recommendation is made to the user.

6.3 Lessons Learned

A recommendation system is a powerful tool, however it suffers from Metcalfe's law, where in a network is proportionally valuable to the square of the number of nodes. With a small number of users the recommendations have very low value, more than not being complete gibberish. However as you scale the system up, these recommendations become more valuable.

Bibliography

[1] Debian. Debain popularity contest. http://popcon.debian.org/.

[2] J.F.Kelley. Wizard of oz user evaluation method. http://www.usabilitybok.org/wizard-of-oz.

[3] Microsoft. Nuget concierge. http://blog.nuget.org/20130816/introducung-nuget-concierge.html.

[4] Microsoft. Nuget package manager. https://www.nuget.org/.