The purpose of this assignment is to write a Java program to answer queries about software packages that are used to build a very big system. (The data that you will use comes from Seneca’s CDOT Linux Distribution and contains hundreds of thousands of items of data that are linked together.)

The raw data are 4 tables: **goal**, **requires**, **provided** and **created**. They contain data about **sources**, **capabilities**, and **binaries**. They are available, as Excel CSV files, at: <http://england.proximity.on.ca/chris/outbound/lb/> ; the CSV names end with: goal, dep\_cap, dep\_rpm and dep\_srpm.

Examples of sources:

fedora-release-notes-21.06-1.fc21.src.rpm

firefox-32.0.2-1.fc21.src.rpm

autoconf-2.69-15.fc21.src.rpm

Examples of capabilities:

dbus-devel

glib2-doc

intltool

Examples of binaries:

glib2-doc-2.42.0-1.fc21.noarch

gtk3-devel-docs-3.14.2-1.fc21.x86\_64

The **goal** table is a list of sources that comprise a Linux Distribution. It contains about 14000 unique sources. Here are the first 2 rows:

anaconda-21.48.8-1.fc21.src.rpm

fedora-release-notes-21.06-1.fc21.src.rpm

The **requires** table is a list of sources and associated required capabilities. Each row contains a source, S, and a capability, C; and means that building S requires the capability, C. It is a many:many relationship – each source can have different capabilities, and different sources can have the same capability. There are about 90000 rows. Here are some sample rows:

fedora-release-notes-21.06-1.fc21.src.rpm publican

fedora-release-notes-21.06-1.fc21.src.rpm desktop-file-utils

fedora-release-notes-21.06-1.fc21.src.rpm publican-fedora

anaconda-21.48.8-1.fc21.src.rpm dbus-devel

xorg-x11-server-1.16.1-1.fc21.src.rpm dbus-devel

The **provided** table is a list of capabilities and associated binaries. Each row contains a capability C and a binary, B; and means that C is provided by B. It is a 1:1 relationship – each capability is provided by one binary, and each binary provides one capability. There are about 10000 rows. Here are the first 3 rows:

dbus-devel 1:dbus-devel-1.8.6-3.fc21.x86\_64

python-bugzilla python-bugzilla-1.1.0-2.fc21.noarch

libxklavier-devel libxklavier-devel-5.4-4.fc21.x86\_64

The **created** table is a list of binaries that are created by sources. Each row contains a binary B and a source S; and means that B is created by S. It is a many:1 relationship – different binaries can be created by a source. There are about 7700 rows. Here are a few sample rows:

1:dbus-devel-1.8.6-3.fc21.x86\_64 1:dbus-1.8.6-3.fc21.src.rpm

1:dbus-x11-1.8.6-3.fc21.x86\_64 1:dbus-1.8.6-3.fc21.src.rpm

glib2-doc-2.42.0-1.fc21.noarch glib2-2.42.0-1.fc21.src.rpm

When building large systems the concept of software dependency is important. We say that one source S1 **depends** **on** another source S2 if S1 requires some capability C, and the same C is provided by some binary B, and the same B is created by source S2. We will write this as S1 ⇒ S2. This can be summarized as:

means

For example, anaconda-21.48.8-1.fc21.src.rpm depends on 1:dbus-1.8.6-3.fc21.src.rpm, because:

anaconda-21.48.8-1.fc21.src.rpm requires dbus-devel,

dbus-devel provided 1:dbus-devel-1.8.6-3.fc21.x86\_64, and

1:dbus-devel-1.8.6-3.fc21.x86\_64 created 1:dbus-1.8.6-3.fc21.src.rpm.

The collection of all dependencies can be represented by a directed graph (without multiple edges, but can have loops). Each vertex represents a source and each directed edge ( such as S1 ⇒ S2 ) represents a dependency. Finding paths and cycles in this graph is important when attempting to build the system. If only a few sources change from one release to another you don’t want to re-build the entire system. Instead you only need to rebuild parts of it, according to the dependencies. Eg, if S1 depends on S2, and S2 depends on S3 and S3 depends on S4, then if the source S4 changes then you need to rebuild all of S3, S2 and S1. This is summarized by:

S1 ⇒ S2 and S2 ⇒ S3 and S3 ⇒ S4

What happens if S4 also depends on S1? We get a cycle (“circular” path) of dependencies, and manual intervention is required when building.

Here is a path of length 2:

kernel-3.17.0-301.fc21.src.rpm ⇒ binutils-2.24-21.fc21.src.rpm ⇒ glibc-2.20-5.fc21.src.rpm

Here is a loop (a cycle of length 1):

mono-2.10.8-8.fc21.src.rp ⇒ mono-2.10.8-8.fc21.src.rp

Here is a large cycle (path of length 8) of dependencies:

kernel-3.17.0-301.fc21.src.rpm

==> gettext-0.19.2-4.fc21.src.rpm

==> automake-1.14.1-5.fc21.src.rpm

==> 4:perl-5.18.4-304.fc21.src.rpm

==> tcsh-6.18.01-12.fc21.src.rpm

==> autoconf-2.69-15.fc21.src.rpm

==> gcc-4.9.1-11.fc21.src.rpm

==> glibc-2.20-5.fc21.src.rpm

==> kernel-3.17.0-301.fc21.src.rpm

Your program will build a graph of dependencies and answer various queries about chains of dependencies. In detail, your program will:

* load in the tables (in Excel CSV format)
* determine all (direct) dependencies, using only goal sources
* create a directed graph from the dependencies
* answer these type of queries:
  + find the size of the dependency graph: # of edges and vertices
  + list all goal sources that match a given pattern (ie substring)
  + list all dependencies S1 ⇒ S2 where S1 matches a given pattern and S2 matches a given pattern
  + find a path of specified length L between 2 goal sources that are specified by two patterns – find the first goal source, S1, that matches, and then only for S1, find all sources S2 connected to S1 by a path of length L.

You will work in pairs (TBA). One person will program main, the other programs a Graph class. You will use suitable data structures: arrays and Java classes chosen from

* ArrayList
* TreeSet
* TreeMap
* LinkedList

**Main** performs:

* loads data from CSV files into appropriate data structures
* determines all dependencies and uses a Graph method to add the associated edge
* performs queries:
  + find size of graph by using a Graph method
  + print matching goals
  + print matching dependencies by using a Graph method
  + finding paths by using a Graph method

The **Graph** **class** only knows about vertices (that are labelled with a name) and edges – it knows nothing about sources or dependencies. It provides general purpose graph methods to:

* add an edge
* get size of graph
* print matching edges
* find a path of given length with given patterns
  + use a recursive depth-first search with an additional parameter for length

You must justify all data structures used by following this format in your Report:

|  |  |  |  |
| --- | --- | --- | --- |
| data to be represented | data structure | operations | complexity |
|  |  | operation 1  operation 2  : | O( ? )  O( ? )  : |

**Marks** –similar to A1.

**Submission** – consists of:

* Cover page
  + indicates your name & partner’s name.
* Choice of Data Structures
  + for each major data structure explain why you chose it, and include the above table
* Java program
  + include the complete Java program
    - clearly indicate if you wrote main, or Graph class
    - surround your code with a coloured border
* Java output
  + demonstrate a variety of queries
  + highlight with colour the part of the output that you wrote

**Plagiarism** – same as A1.

**Due Date and Time** Monday, December 1st; similar to A1