CS61B, Fall 2014 **Project #3: Graphs** **P. N. Hilfinger**

Due: Monday, 8 December 2014

1 Introduction

For this final project, you will be writing a library package to provide facilities for manipulating graphs, plus two clients that use the package. The package must be general: no specifics of either client may find their way into the code for the graph package. We will check that by running our own clients against your package. That is, in contrast to past projects where we didn't care how you arranged the internals of your program as long as it functioned according to the specification, in this case, part of the project code—the API for the graph package—will be set in stone.

2 Client: Make

We've been using the Make program (specifically, GNU make) in most programming assignments this semester. For this project, we are going to implement a much stripped-down version. An input file ("makefile") will consist of *rules* that indicate that one set of objects (called *targets*) depends on another set (called *prerequisites*). Any line that starts with '`#`' or contains only tabs and blanks is ignored. For other lines, the syntax is

T: P_1 $P_2 \cdots P_m$
command set

where $m \geq 0$, and T starts in the first column. We'll call the first line of the rule its *header.* Each T or P_j is a string of non-whitespace characters other than '`:`', '`#`', or '`\`'. The command set consists of zero of more lines that begin with a blank or tab. Command sets with no lines in them are called *empty.* The commands for one rule end at the next rule (or the end of file). There may be one or more rules that name any particular target, but no more than one of them may be non-empty.

2.1 Interpreting a Makefile

In general, the program is given a "makefile" in the format just described, plus a set of targets that it is to "rebuild," and information about which objects currently exist and when they were last built. (In real life, Make finds out the latter for itself, by querying the file system. We will stop short of this level of realism.)

The program reads the Makefile sequentially. After reading all the definitions, the program considers all the targets it has been asked to rebuild, and figures out which commands (if any) must be executed to rebuild each target, and in what order. To rebuild a target T:

- If the makefile does not have a rule with T as the target,

  - If T does not currently exist, there is an error.
  - If T does exist, it is considered "built."
- Otherwise, if the makefile does have one or more rules for T, then
  - Make considers all the headers in the makefile whose target is T and collects the set of all prerequisites listed in those headers.
  - It then rebuilds all of these prerequisites, recursively applying the procedure being described here.
  - If T does not exist, or if T is older than at least one of its prerequisites,
    * If all the rules with T as their target are empty, there is an error.
    * Otherwise, Make executes the *commands* set from the non-empty rule.
    * Make then sets the creation time of T to be larger than that of any existing object (thus making T the youngest object).
- Make never rebuilds something unnecessarily.

For this assignment, "executing" commands simply means printing them verbatim.

As usual, in response to input errors (in format, for example), you should output an error message on the standard error output and exit with a non-zero exit code. Do not output to the standard error output otherwise. Errors include missing prerequisites with no build instructions, attempting to build a target that has more than one non-empty commands set, syntax errors in the input files (see also §2.2), and circular dependencies. Error messages must occupy one line and contain the word "error" in any combination of cases.

2.2 Running Make

To run the Make client, a user will type

```
java make.Main [ -f MAKEFILE ] [ -D FILEINFO ] TARGET ...
```

where `[...]` indicates an optional argument. *MAKEFILE* (default `Makefile`) is the name of the makefile. *FILEINFO* (default `fileinfo`) describes the current existing objects and their modification times (see below). Each *TARGET* is processed as a target in order.

The *FILEINFO* argument is a file containing lines of the form

NAME CHANGEDATE

where *CHANGEDATE* is an integer indicating a time (the larger the time, the younger the named object). On a Unix system, this would be the number of seconds since the Epoch (the beginning of 1 January 1970), not counting leap seconds (making 2038 (2^31 seconds past the Epoch) the Unix Y2K year). However, for us, the only thing that matters is that larger is later.

The first line of the *FILEINFO* file contains only a date (that is, an integer), indicating the current time, and larger than any *CHANGEDATE* in the file.

3 Client: Trip finder

I suspect you all have or have used some sort of GPS device by now, or gotten directions from the Google$^{\text{tm}}$ map service. Such systems "know" about a network of roads, and given two end points (and perhaps some waypoints in between) will pick out a shortest route. This next client will be a stripped-down version. Specifically, given a map and a request—a sequence of two or more points on the map—it will produce a shortest route from the first point to the second, third, and so forth.

The map file will be in free format (a sequence of "words" containing no whitespace and separated by whitespace—blanks, tabs, newlines, etc.). Information comes in two forms: location entries, introduced by the letter 'L', and road entries, introduced by the letter 'R'.

- Location entries have the form

  $\texttt{L} \quad C \ X \ Y$

  where C designates a place and X and Y are floating-point numbers. This means that the place named C is at coordinates (X, Y) (we're sort of assuming a flat earth here.) There may only be one location entry for any given C.

- Distance entries have the form

  $\texttt{R} \quad C_0 \quad N \quad L \quad D \quad C_1$

  where each C_i designates a place, N is the name of a road L is a numeric distance (a floating-point number), and D is one of the strings NS, EW, WE, SN. Each C_i must be declared in a previous location entry. For example, the entry

  ```
  R Montara US_1 56.0 NS Santa_Cruz
  ```

  would mean that *US_1* nominally runs North to South from Montara to Santa Cruz, for a distance of 56.0 miles, and that there are no map points in between (I guess we have a rather idiosyncratically sparse map). Our road connections will all be two way, so the sample entry also indicates a South to North-running route (SN) from Santa Cruz to Montara. There will always be at most one road between any two locations. As in real life, the designations of direction (NS, SN, etc.) on road segments are *not* related to the (X, Y) positions of the locations they join[1]

Requests will come from a file each of whose lines contains the names of two or more points on the map, separated by commas and whitespace. For example,

```
Berkeley, San_Francisco, Santa_Cruz
Santa_Cruz, San_Jose, Stockton
```

Strip any whitespace from before and after each of these strings.

In response, your program is supposed to print out a route in this format for each request:

[1]Consider, for example, taking I80 from where it intersects with University Avenue towards Sacramento. At that point, it travels north and slightly west, but it is officially east-bound I80.

```
From Berkeley:

1. Take University_Ave west for 0.1 miles.
2. Take Martin_Luther_King_Jr Way south for 1.2 miles.
3. Take Ashby_Ave west for 1.8 miles.
4. Take I-580 west for 1.0 miles.
5. Take I-80 west for 8.4 miles to San_Francisco.
6. Take US-101 south for 34.5 miles.
7. Take CA-85 south for 13.3 miles.
8. Take CA-17 south for 22.2 miles.
9. Take CA-1 north for 1.0 miles to Santa_Cruz.
```

When two adjacent segments use the same road with the same orientation (NS, EW), join them into one segment. So instead of printing

```
3. Take Ashby_Ave west for 1.0 miles.
4. Take Ashby_Ave west for 0.8 miles.
5. ...
```

print

```
3. Take Ashby_Ave west for 1.8 miles.
4. ...
```

As usual, in response to input errors (in format, for example), you should output an error message on the standard error output and exit with a non-zero exit code. Do not output to the standard error output otherwise.

3.1 Running Trip Finder

To run the Trip client, a user will type

```
java trip.Main [ -m MAP ] [ -o OUT ] [ REQUESTS ]
```

MAP (default `Map`) is the name of the file containing the map data. *REQUESTS* (default, the standard input) contains the requests. *OUT* (default, the standard output) receives the solution.

4 The Graph Package

You should implement these two clients using a graph package whose interface we've supplied. The package `graph` will export several classes (marked public in the skeleton) and no other public classes. You can add classes to the package as long as they are not public. You can add members to the public classes, as long as they are overridings of public methods or are not public or protected. You can change methods, but not the signatures (types of parameters and return values) of public or protected methods. We will test your graph package separately, exercising methods that might not be used by either of the clients you implement, so unit tests will be particularly useful. Likewise, we will test your clients using *our* graph package, so that your clients must rely only on the published spec, and not on any special behavior of your particular implementation.

5 Your Task

The staff directory contains skeleton files for this project in `proj3`.

Please read *General Guidelines for Programming Projects.*

Be sure to include tests of your program (that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them. To help with testing and debugging, we will provide our own version of the program, so that you can test your program against ours (we'll be on the lookout for illegal moves). More details will follow.

We will expect your finished programs to be workmanlike, and of course, will enforce the mechanical style standards. Make sure all methods are adequately commented—meaning that after reading the name, parameters, and comment on a method, you don't need to look at the code to figure out what a call will do. Don't leave debugging print statements lying around. In fact, don't use them; learn to use the debugger (either `gjdb` or that of `Eclipse` or your favorite Java-system vendor).

The nature of the graph package is such that you'll probably want to do extensive JUnit testing on it. Always feel free to add private setup procedures for testing purposes only that set up conditions (such as graphs) that are used by several different tests.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that

```
make check
```

runs your tests.

6 Advice

Use `svn` or the `hw` command regularly. This provides backup. It's the safest way to transport things from a copy of your project on a local computer to (`commit`) and from (`update`) the instructional machines. Frequent commits will limit the amount of work you have to do when a file gets messed up. Use `svn status` or `hw status` to check for files that should be added and to make sure all necessary commits have been done.

Use of `make style` not only encourages proper formatting, but also shows you missing documentation or left-over internal comments that might indicate work that still needs to be done, and can find certain error-prone constructs or bad practices.

The skeleton provides the great bulk of the code for the two applications: Trip and Make. There are a few places for you to fill in the use of the graph interface and a few other details. Your main problem here will be understanding the existing code (a big part of programming in real life).

The graph package will constitute the bulk of your work. First, understand the intent of everything in the package. Again, this will mean reading and understanding existing code—an API in this case. Be sure to ask us about anything that is unclear. Next, choose your data structures, making sure they will suffice to implement everything. Do the "...`Graph`" classes first. The ...`Traversal` classes are built on them, and the ...`ShortestPaths` classes in turn are built on those.

We will supply staff implementations of the Make and Trip clients that you can test against your graph package with the same integration tests as for your own implementations of these applications. Likewise, we will supply a staff implementation of the graph package that you can use to test your implementations of the two applications. As usual, none of these staff implementations is the standard: this specification and the documentation comments on public parts of the graph package are the standard.

For the Make client, an obvious way to proceed is to have a graph with its vertices representing targets and containing the commands needed to build a target. There may be more than one rule per target, so you'll have to keep a mapping from targets to vertices so that you can accumulate all the necessary information in each one vertex for each target. The edges then represent dependencies; there is an edge from each target to each of its dependencies. As indicated in lecture, a depth-first post-order traversal of this graph visits the vertices (targets) in the order they must be built. You can keep track in the vertices whether they have been rebuilt, that is (post-)visited yet, so that you can check for circularity during the visit of a vertex simply by looking at the vertex's successors (which must necessarily all be marked) and checking that each has been visited. For a successor vertex to be marked but not yet visited must mean that there is a path from that successor back to the vertex being visited and also from the vertex being visited to that successor node: a circularity.

The trip client is an obvious application of A* search. The vertices are cities and the edges are roads. You can use the coordinates of the cities to get a heuristic estimate of distance.