Web Search, Télécom ParisTech
Web crawling

Pierre Senellart (`pierre.senellart@telecom-paristech.fr`)

15 March 2010

The purpose of this lab session is to program various robots for crawling the Web, according to different crawling strategies (depth-first search, breadth-first search, etc.), and gathering different kinds of information while crawling.

Labs environment

Most of the labs will use the Java programming language. You can use either the editor of your choice (e.g., `emacs`, `vim`, `kate`, `gedit`) and the command-line utilites `javac` and `java` to compile and run your Java programs, or the `eclipse` integrated development environment.

Necessary files for each lab session (such as the implementation of the `Robot` abstract class described next), and references to external websites, are available from the course website, on `http://pierre.senellart.com/enseignement/2009-2010/inf396/`. You are strongly advised to refer to the Java API documentation, `http://java.sun.com/javase/6/docs/api/`.

At the end of each lab session (or at the latest at 11:59pm on the same day), you are required to send by email all the code to the tutor responsible for the lab. This, along with observations made during the lab session itself, will be used to give a grade between A and E. The final grade for this course will be an average of the grades for each lab (C is enough for passing the course).

Labs assignments, are, by design, lengthy. It is not a requirement that you finish them. In particular, the "To go further" section is entirely optional. If you do a reasonable job during the 3 hours of the lab session, you will probably be awarded a B grade. You do not have to work overtime (but are of course welcome to do so).

The abstract class *Robot*

To simplify things, you are given the code of an abstract class `Robot` the different crawlers you implement must inherit of. This class provides basic crawling functionalities and declares methods that can be redefined in subclasses. It contains the following fields and methods with the *protected* access level (*private* fields and methods are not detailed since they are not accessible from subclasses):

Queue<String> candidates this will be used as a priority queue of the URLs to download next.

Set<String> done this is the set of all URLs already crawled.

final int initialQueueSize a constant that is to be used as an argument to the constructors of `PriorityQueue` and `PriorityBlockingQueue`.

Robot(String ua, long delay) the constructor of the `Robot` class. The first argument is the *user-agent name* of the robot, as used in the `User-Agent` HTTP header. The second argument is a delay, in milliseconds, between two successive queries to the same Web server.

abstract Comparator<String> comparator() this abstract method will be used to define the comparison order of the elements in the `candidates` priority queue.

abstract void initialize(String url) this abstract method is called for each URL of the initial seed.

abstract void dealWith(String url, Set<String> s) this abstract function is called each time a URL is processed. `url` is the current URL, while `s` is the set of URLs of all hyperlinks contained in `url`.

void executionLoop(Set<String> seed, long seconds) this (redefinable) method is called to start the crawl and should not return until the end of the crawl. The `seconds` parameter indicates the number of seconds to allocate to the crawl. After this delay has passed, the crawl should stop. The implementation of `executionLoop` in the `Robot` class just does some initialization, and calls the `initialize` method on each URL of the seed.

final void processURL(String url) this (final) method is to be called by the robot each time a URL has to be processed. It makes sure crawling ethics are respected, retrieves all links from this URL, and calls `dealWith`.

We use the following scheme for class hierarchies: An abstract subclass of `Robot` redefines the `executionLoop` method in order to indicate how the function `processURL` should be called on each candidate URL (e.g., a single thread browsing the `candidates` queue), and when the crawl should terminate; it is also responsible for initializing the `candidates` and `done` fields. An abstract subclass of this controlling subclass defines the `initialize`, `dealWith` and `comparator` methods in order to specify the crawling strategy used (e.g., breadth-first search). A subclass of this strategy subclass redefines the `initialize` and `dealWith` methods to actually do something with the crawled content (e.g., building a graph of the crawl). This is summarized in the following diagram.

`Robot`

↑

Controlling abstract subclass (redefines `executionLoop`)

↑

Crawling strategy abstract subclass (defines `initialize`, `dealWith`, and `comparator`)

↑

Actual crawler (potentially redefines `initialize` and `dealWith`)

Remark. In a language supporting multiple inheritance (unlike Java), the actual crawler could inherit from both a crawling strategy class and a controlling class. Other patterns are possible, such as making the controlling class an external class referred to in the actual crawler class.

1 Implementing a simple BFS crawler

1. Implement the controlling class `SingleThreadedRobot`, directly derived from `Robot`. You should provide a constructor (that initializes `candidates` and `done`), as well as a definition of the `executionLoop` method (that browses all candidate URLs, feeding them one by one to the `processURL` method). You can use a `java.util.PriorityQueue` for `candidates` (`comparator` will yield a Comparator that can be used to construct this priority queue), and a `java.util.HashSet` for `done`.

2. Implement the strategy class `BFSRobot`, directly derived from `SingleThreadedRobot`. You should provide a constructor and definitions of the `initialize`, `dealWith` and `comparator` method. In order to implement the comparison function (it should order first URLs of lesser depth), an additional data structure (e.g., a `HashMap`) can be added to the `BFSRobot` to remember the depth of each URL.

3. Implement a crawler class `ShowURLRobot`, directly derived from `BFSRobot`. This crawler will just output (on `System.out`) the list of URLs found during the crawl. You should provide a constructor, and a redefinition of the `dealWith` method.

4. Implement a class `ShowURL` with a `main` function that makes use of your `ShowURLRobot`. Test this function.

2 Various crawlers

1. Implement a crawler class `GraphExtractionRobot`, directly derived from `BFSRobot`. This crawler will output on `System.out` the list of URLs crawled along with some index for this URL, and on `System.err` the list of encountered hyperlinks, in the following format, one hyperlink per line: "$\mathit{index}_{\mathit{url}_1}$ $\mathit{index}_{\mathit{url}_2}$". Thus, we can have for instance on `System.out`:

   ```
   0 http://example.com/
   1 http://example.com/toto
   2 http://example.com/titi
   ```

   and on `System.err`:

   ```
   0 1
   0 2
   1 2
   ```

   Test this new crawling class.

2. Implement a strategy class `DFSRobot` that implements a depth-first search strategy. Test this new strategy class by making `GraphExtractionRobot` inherit from `DFSRobot`.

3. Implement a controlling class `MultiThreadedRobot` that starts off a number of threads, each thread processing different URLs. `candidates` can be defined as a `java.util.concurrent.PriorityBlockingQueue`. An introduction to concurrency control in Java can be found on `http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html`.

   Test and compare with `SingleThreadedRobot`.

3 To go further

1. Implement a limited-depth breadth-first search strategy.

2. Implement a crawler that retrieves and stores all text found in crawled Web pages. You might need an additional HTML parsing library, such as TagSoup (`http://www.ccil.org/~cowan/XML/tagsoup/`) or HTML Parser (`http://htmlparser.sourceforge.net/`).

3. Implement a controlling class with a single thread that makes use of Java asynchronous IO (cf. the classes of `java.util.concurrent`).