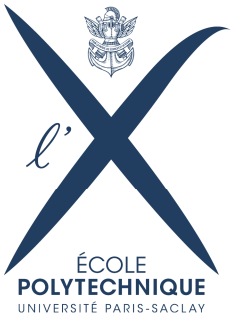
STAR WARS SOCIAL GRAPH

INF431

07/04/2017

Pierre-Elie Bélouard
Melchior d'Harcourt
X2015



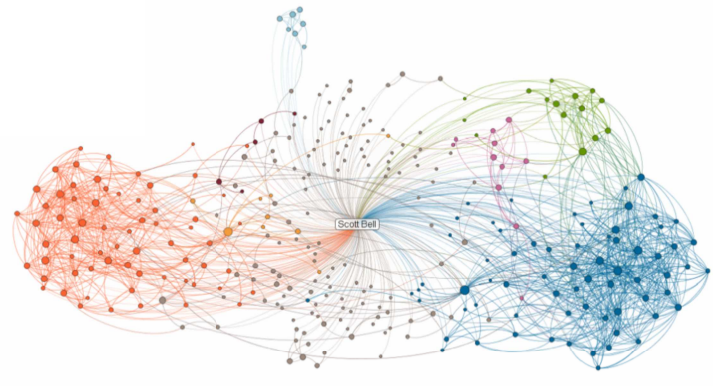
1
INTRODUCTION

The virtual **Star Wars world** is composed of millions of characters. Several thousands of them have the honour to have an article in Wikia, the Star Wars online encyclopedia. Wikia is composed of about 140 000 pages about objects of this virtual world : characters (Luke Skywalker, Darth Sidious, Han Solo, ...), places, cities, planets (Coruscant), historical periods, ... On each Wikia page, you can find some links to other pages, meaning that there is a direct link between the subject of the current page and the object of the link.

The question asked in this project is the following : assuming we consider one specific character having a page in Wikia, and with the datas given by the pages and the links in Wikia, could we create a program returning the distance from the departure character to all the Star Wars characters having a Wikia page ?

First of all, let us say some definitions.

— A **Star Wars object** is the subject of a Wikia Page. It is not necessarily a character.
— A **star Wars character** is a Star Wars object who is a character (not necessarily a human : it could be a droid, ...).
— A character A is **linked to** a character B if there is a link to A-Wikia page in B-Wikia page.
— The **distance** from character A to character B is defined as following : it is the smallest integer k such that there exists some characters $A_0 = A, A_1$, ..., $A_k = B$ such that $\forall i = 0,...,k-1$, character A_i is linked to character $A_i + 1$.
— **Star Wars Social graph** : the oriented graph whose nodes are Star Wars characters and whose connections are defined by the relation "is linked to" just above. You will find below one example of social graph, in the banking world.



It is obvious that a **bread first search algorithm** starting from the given character will give the solution.

But how could we implement this in an efficient way on the Java Virtual Machine ?

2
INTERACTION BETWEEN THE JAVA VIRTUAL MACHINE AND THE INTERNET WITH JSOUP LIBRARY

The first difficulty we had to face was the way to **interact with the internet**. We used the **Jsoup library** and created a special class (*PageParser*) which takes an url adress of the Wikia page of a Star Wars object and does the following tasks :

- connects to the internet and gets the html code of the page (Jsoup.connect)
- checks if the object is a character. To do this, we have remarked that all the characters (and only them) have, in their Wikia pages, a box with the items "Biographical informations", "following description", and "chronological and political information". To determine if the object is a character, we also have to read the html code and return true if we read these items.
- If the object is a character, gives the url adresses of the links which are in the character Wikia page. In the html code, a link to another WikiapAGE is identified with the code "a[href≙wiki]". We also read the html code and select the following links. Take note that we select far too much links than we have to (for examples, links to url adresses related to the same character). To try not to visit too much links, we do not consider the links containing some special characters which could not appear in the url adress of an object main page.

How many data do we theorically exchange with wikia server to connect to all these pages ? Wikia contains 140000 pages. Assuming that the average weight of a wikia Page is the same of the average weight of a Wikipedia page (several kilobytes) and using the open-sources statistics provided by Wikipedia (see the attached graphic), doing the whole job would **theorically need no more than** 1 **GB** to exchange with the server. In the practice, we visit each page once, but we could visit some pages wich are not Wikia Pages but also pages related to Wikia pages so it could needs up **to** 20 **GB data**.

3 WHY IS A CONCURRENT PROGRAM A GOOD SOLUTION TO THE PROBLEM ?

It could seems natural to compute a simple program, marking the vertice at a distance i from the departure point, and then put the vertice linked to these vertice (let call them i-vertice)in a Set (let name it $i+1$-Set). We then visit the vertice in the $i+1$-Set. The distance between the departure character and the characters corresponding to the vertice in the $i+1$-set which have note been marked yet is $i+1$. We mark them and continue the search at the step $i+2$.

This seems to be an efficient implementation on a single computer, but it is actually not. Indeed, it takes time to the Wikia server to answer the Jsoup request. A Timer has made us realize that there are, in average per request and if we use the sequential code, 0.3 s in which the processor is doing nothing but waiting for the answer of the server. And because we visit about one million pages, it would take several days to do the job !

Here comes the concurrency to help us solving the problem in a reasonable time. The idea is exactly the same, but we create **one thread per page to visit** (this will need some synchronization). Each thread sends a request to the Wikia server. While waiting for the answer, and contrary to the sequential program, the processor is not doing nothing, because other threads are working.

We also created a **semaphore** (the shared value int workingThreads) to avoid that too many threads work together. The optimal number of working threads is not very clear, because there are two different contrary trends. If a lot of threads are working at the same moment, it reduces the likelihood for the processor to do nothing but waiting for an answer from the server, but with too many threads working together and trying to take the locks at the same moment, the program will be very slow. We have done the simulation with a maximal number of 100 threads working together (the value in the final variable MAXIMAL). If we had more time, we would have done some simulations with other values of MAXIMAL in order to find **the most efficient one**.

4 THE DATA STRUCTURES AND THE ARCHITECTURE OF THE PROGRAM WE IMPLEMENTED

Here are the **classes** we use :

— *WookiePage* is the Java representation of a wikia page. It has, among others, the fpllowing fields : url adress, name of the object the page is about, the boolean isCharacter (true if the object is a character). The order on the WookiePage is the dictionary order on the url adress

— *PageParser* is the interface with the internet , we have already described it in the first section

— *PageExploration* implements the Runnable Interface. It takes a url adress. While running, it will use PageParser to connect to the Wikia page, and mark the pages linked to the WookiePage in order to visit them at the next step.
— *Manager* is the main class : it initializes, launches the threads (no more than 100 working at the same time, in the right order), deals with the coordination between the threads, and impresses the new characters found at the end of each round.

Here are the **shared variables** used :
— A semaphore workingThreads (protected by the the same lock and having the same condition variable as toVisit) : there can be no more than MAXIMAL threads working together.
— A shared treeSet t (with a lock) containing all the url pages already visited. It is to prevent some pages to be read several times.
— A shared stack impression (with a lock) : contains all the characters which have been found at a distance k and which need to bee printed by the class Manager.
— A shared stak toVisit : contains all the url adresses of the Wikia pages which have been found at this step and which need to be visited at the next round.
— A shared counters array canal (protected by a lock and with a condition variable) : canal[k] is the number of threads launched at the round k and which have not finished their work yet. When canal[k]= 0, Manager could print the characters found during this round and then starts a new round.

Take note that the synchronization procedures we've chosen are quite elementary : each shared object is protected by an **own lock**, and we use two **condition variables** : the first one is to inform the main class that less than MAXIMAL threads are working (so it could launch new threads) and the second one is to inform that all the threads of a given round have finished their jobs, so Manager could impress the characters found at this round and the launch the threads for the next round. The synchronization does not need to be elaborate because the time spent working on the shared objects is a very small proportion of the total runtime. The majority of this time is spent waiting for the answer of the server or reading the html code.

5 CONCLUSION

Building the social graph of a community is a problem which could be solved by a sequential program in a very simple but unefficient way. Using concurrency enables us to solve the problem in a quicker way, but it needs much more work and precautions, although the synchronizations we've chosen are quite elementary. Let us finish by remarking that the algorithm we implemented could be adapted to the real world using the datas provided by Wikipedia or some social network (Facebook, Twitter, LinledIn, ...). The analysis of the social graph is an excellent way to know a lot about the different groups, the differents influence spheres of the society. It could be used to predict how a trend could be spread in a country or in the world.