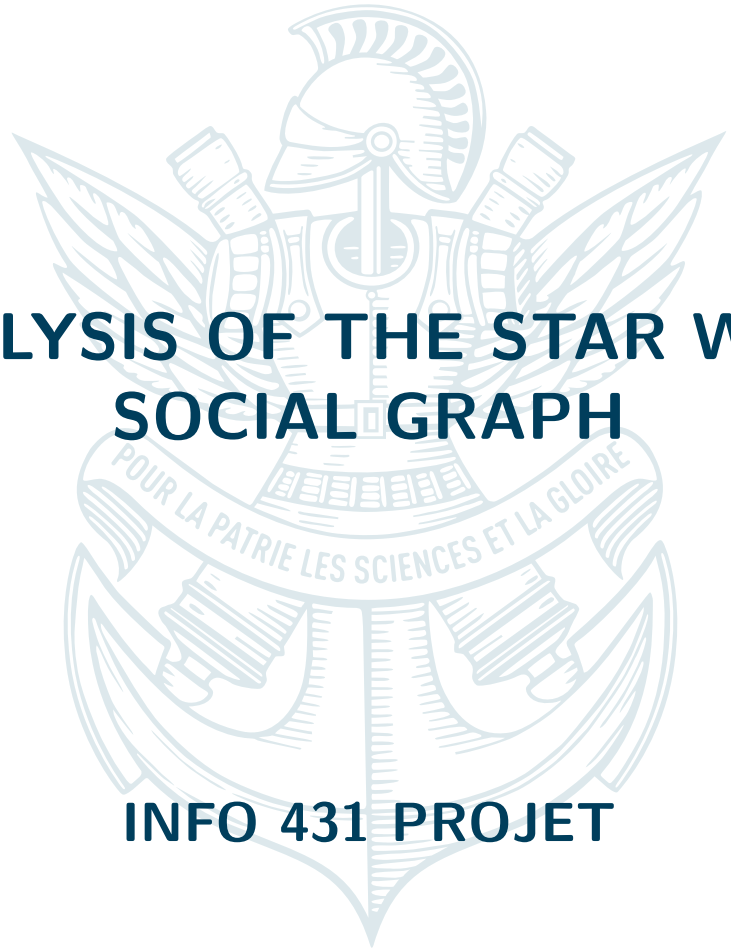

ANALYSIS OF THE STAR WARS SOCIAL GRAPH

INFO 431 PROJET

7 avril 2017

Jiefeng Fang,
Yayun Dai

1 Introduction

First of all, we would like to provide you our git address just to facilitate your lecture in case of ambiguity. https://github.com/Datoclement/StarWarSocialGraphAnalysis

The goal of this project is to study the social graph of Star Wars characters based on the rule that a character A is linked to another one B if its name appears on A's associated wiki page. We've developed a parallel search robot that traverses the social graph built from Wookieepedia and saved on disk the visited k-neighborhood using the *Breadth First Search* algorithm.

2 Algorithm

Our search program contains two parts : the first part is to scrap all the characters based on the categories ; the second part is the parallel search of the k-neighborhood.

2.1 Part I : Get the characters

As there is no common index to reach every character page on the site, to be sure that we were actually scrapping all the characters, we had to first fetch all the categories indexing the characters such as Category :Individuals. From this root category, we used the Breadth First Search Algorithm to get all the subcategories until it fetches all the associated unique characters.

In order to compare the performances, we've written two versions of codes for this process, a sequential one and a concurrent one. For each version, we have a HashSet *list* to store all the characters that we find, a HashSet *visited* to record the categories that we've already encountered, a queue q to put in the new nodes i.e. the new subcategories during the search.

For each node n polled from q, we have a function *findSubCategories*, which is to find the subcategories of n and add them to q ; a function *findCharacters*, which is to find the characters associated directly to n. Here's how the two functions work.

findSubCategories(String psc, Queue q) : *psc* is the source code of the page of this node. To find the subcategories, we search in the part labeled as "mv-subcategories" of the source code all the links that contain "/wiki/".
findCharacters(String psc) : To find the characters, we search in the part labeled as "mv-pages" of the source code all the links that contain "/wiki/".

2.1.1 Sequential version

The sequential version is a normal BFS algorithm, shown as in Figure 1.

```
public SequentialCharacterScraper(){
    this.list = new HashSet<String>(40000);
    this.visited = new HashSet<String>(40000);
    Queue<String> q = new LinkedList<String>();
    q.add("Category:Individuals");
    while(!q.isEmpty()){
        String str = q.poll();
        if(this.visited.contains(str)) continue;
        this.visited.add(str);
        String pageSourceCode = new SourceCode(str).content;
        this.findSubCategories(pageSourceCode,q);
        this.findCharacters(pageSourceCode);
    }
}
```

Figure 1 – SequentialCharacterScapper

2.1.2 Concurrent version

Suppose that there is n available processors, we can create n threads, each taking an subcategory from the queue and find its subcategories and its characters. When the queue is empty, the thread has to wait for 300ms to make sure that there will be no more nodes to be added.

```
Thread[] ts = new Thread[n];
for(int i=0;i<n;i++){
    System.out.println("Processor "+i+" is created.");
    ts[i] = new Thread(new Runnable(){
        public void run(){
            String str = "";
            while(true){
                str = q.poll();
                if(str==null){
                    try{
                        Thread.sleep(300);
                    }
                    catch(Exception e){e.printStackTrace();}
                    str = q.poll();
                    if(str==null)break;
                }
                if(visited.contains(str))continue;
                visited.add(str);
                String pageSourceCode = new SourceCode(str).content;
                ConcurrentCharacterScraper.this.findSubCategories(pageSourceCode,q);
                ConcurrentCharacterScraper.this.findCharacters(pageSourceCode);
            }
        }
    });
    ts[i].setName("Thread-"+i);
    ts[i].start();
}
```

Figure 2 – ConcurrentCharacterScapper

2.2 Part II : Parallel Breadth First Search

To implement a parallel BFS, we use two queues which we note as q1 and q2 here : q1 is used to store the k-neighborhood and q2 is to store the k+1-neighborhood. Still, we have n threads, each taking a node from q1 and putting its neighbors in q2 until q1 becomes empty.

We implemented two versions of the search algorithm : an online one and an offline one. The online one is to search directly the neighbors in the source code of the website ; the offline one searches in a text file created in advance which has listed all the 1-neighbors of every character. Therefore, the difference between the two versions is the way to find the neighbors.

2.2.1 Online Search

Online search finds the neighbors through the website :

```
public HashSet<String> findNeighbors(String character){
    HashSet<String> n = new HashSet<String>();
    String pageData = new SourceCode(character).content;
    int head = pageData.indexOf("<article");
    int tail = pageData.indexOf("</article>", head);
    int cur = head;
    while(true){
        cur = pageData.indexOf("\"/wiki/", cur+1);
        if((cur == -1) || (cur > tail)) return n;
        cur += 7;
        int end = pageData.indexOf('\"', cur+1);
        String url = pageData.substring(cur, end);
        if(this.characters.contains(url))
            n.add(url);
    }
}
```

Figure 3 – findNeighbors

2.2.2 Offline Search

Offline search finds the neighbors through a text file that contains all the neighbors of every character. The file is created in advance by visiting the website of every character that we've found in the first part and scrapping its neighbors. In order to be more efficient, we've also used a parallel program to complete this file.

3 Performance

As we have two completely different pipelines to solve the problem, we do the performance test for both of them separately, comparing methods in sequential version and in concurrent version.

3.1 Test Result (Average)

As is shown below, using parallel algorithm can, in terms of average performance, is always not a bad choice in our case, if it is not better.

Preprocess	Scraper of Characters	Scraper of Graph
Sequential	76.9s	----
Parallel	65.2s	24.3min

Online	Yoda depth=1	Yoda depth=2	Yoda depth=3
Sequential	0.9s	22.3s	110.1s
Parallel	0.5s	9.1s	83.8s

Offline	Yoda depth=1	Yoda depth=2	Yoda depth=3
Sequential	137ms	138ms	163ms
Parallel	94ms	107ms	139ms

Offline	Yoda depth=4	Yoda depth=5
Sequential	176ms	180ms
Parallel	161ms	175ms

4 Conclusion

If we run our program on Yoda with a depth of 5, the program terminates in about 15 minutes, which is actually the same order as the total scraping time of the total graph. This fact makes the pre-scraping of complete social graph much more favourable than to search on the Internet each time we run our program.

Moreover, the test result proves to be highly dependant on the speed of Internet, as a result of which our test results are provided only for reference.

Finally, we also see a better performance with a parallel algorithm than with a sequential one, in spite of the fact that our personal computers have only 4 available processors. We have chosen a parallel algorithm that is adapted to our case - only a limited number of processors are available. Actually, we see also some interesting algorithms where we use a larger number of processors and produce a much better theoretical result, e.g. http ://supertech.csail.mit.edu/papers/pbfs.pdf. The analysis of these algorithms are beyond this project, but we still encourage readers to have a try.