HOMEWORK ASSIGNMENT #1 – XPATH, CRAWLING AND RANKING
MANAGEMENT OF BIG WEB DATA (FALL 2020-21)

GENERAL INSTRUCTIONS

- The work can be done in pairs or in singles.
- For the purpose of submission, your **username** will be <first_name>.<last_name>, or <first_name_of_member1>.<last_name_of _member1>.<first_name_of_member2>.<last_name_of _member2> in the case of a pair. E.g., alice.smith or alice.smith.bob.walker
- Your solution should be submitted in a single zip file named **<username>-hw1.zip**, through moodle. Only one member of the pair should submit.
- The zip will include an answers document in pdf format, named **<username>-hw1.pdf** make sure to write your name(s) on the top. The answers can be written in English or Hebrew.
- For every question that requires writing code, code should be written in python version 3.XX and submitted in a **.py** file. Try to keep the code readable as it will be graded.
- You are expected to submit an original work. Copying from references (e.g. existing crawlers) or looking at others' solutions is not acceptable. Do not publish your solution or save it in any public place – in cases of cheating, all involved students will be equally held liable.

TASKS

1. Consider the Wikipedia pages (in the English version, en.wikipedia.org) of professional tennis players, e.g., https://en.wikipedia.org/wiki/Andy_Ram. Look at the HTML sources of the pages and try to write **XPath expressions** (at most 10) with
    - **Input:** HTML contents of a Wikipedia page (in the English version, en.wikipedia.org) of any tennis player, e.g., https://en.wikipedia.org/wiki/Andy_Ram,
    - **Output:** URLs of *Wikipedia pages of related tennis players.* We define a related tennis player as one who was a partner, opponent or coach of the current player.

    The XPath expressions must be *legal* (check yourselves using python code), but they do not need to be *perfect*: you may miss some links (false negatives) or extract some wrong links (false positives). Try to write the best expressions you can find, to minimize false positives and negatives. **Provide the XPath expressions in the answers file,** and for each XPath expression give **one example** of a source URL and the output URL that was found.

    For example:
    The XPath "//a/@href" is too general. It finds 1898 links only in https://en.wikipedia.org/wiki/Andy_Ram

    The XPath //a[contains(text(),'Jonathan')]/@href" is better, and for https://en.wikipedia.org/wiki/Andy_Ram returns the URL of Andy's partner **/wiki/Jonathan_Erlich** (relative URL, actually pointing to https://en.wikipedia.org/wiki/Jonathan_Erlich)
    However, it also returns pages of unrelated players, e.g., /wiki/Jonathan_Stark_(tennis)

Pages outside Wikipedia http://www.protennisfan.com/2006/03/andy_ram_and_jo.html
And of course misses most of the related players (the Xpath has to work for *every* tennis player!).

So, in the answers file you could write:

//a[contains(text(),'Jonathan')]/@href
Example: given https://en.wikipedia.org/wiki/Andy_Ram the XPath returns
/wiki/Jonathan_Erlich

Tips:

- Check the structure of the page to extract links from specific parts of the page. You can use source viewers in your browsers (e.g., "inspect" option in Google Chrome)
- Check the URL structure to extract **only links inside en.wikipedia**
- Check textual context to find only names of players
- Check a few example pages, to make sure your expressions are general enough

2. Write a function `def tennisCrawler(url,xpaths)` whose purpose is to crawl pages of tennis players in Wikipedia
    a. The input `url` is a string containing the URL of the start page (e.g., https://en.wikipedia.org/wiki/Andy_Ram)
    b. The input `xpaths` is a list of strings representing legal XPath expressions
    c. The function will use the xpaths to extract a set of URLs from the web page.
    d. These URLs will be crawled in a combined BFS-DFS order:
        i. Do 3 DFS steps, by each time choosing a random link (from the set extracted by the Xpath expressions), and crawling the selected page.
        ii. Do a BFS step – crawl a page that is closest to the start page (breaking ties arbitrarily). For that, you will have to record for each page its minimum distance from the start page
    e. Mind crawling ethics and particularly, wait at least 3 seconds between page reads.
    f. In total, at most 80 URLs will be crawled in this manner, and only URLs of en.wikipedia.org. Avoid crawling the same URL twice!
    g. The function will return **a list of lists**. Each inner list will contain two strings: the first will be the *full* source URL, and the second will be the *full* URL of a page detected in the source URL by the crawler and that matched the Xpaths. For example, if /wiki/Jonathan_Erlich was extracted from the page https://en.wikipedia.org/wiki/Andy_Ram, the output will contain the list

```
['https://en.wikipedia.org/wiki/Andy_Ram', https://en.wikipedia.org/wiki/Jonathan_Erlich']
```

    The output should not contain repeated pairs, even if some link appears in the source page more than once.
    h. Submit the code in the file *Q2*.`tennisCrawler`.*py*

Tip:

- You can start from the simple crawler example from class, and change as needed.

3. We will compute a PageRank score for tennis players.
    a. Write a function `def tennisRank(listOfPairs, numIters)`
    b. `listOfPairs` is a list of lists in the format of the output of `tennisCrawler` from question 2.
    c. The function will treat each inner list [X, Y] as a link from X to Y.
    d. We will use iterative update rules to compute a PageRank Score for each player:
        i. Initialize page ranks to be $r_j^{(0)} = \frac{1}{N}$ for every page j
        ii. At each iteration, compute the rank of each player as

            $$r_j^{(t+1)} = 0.8 \sum_{i \to j} \frac{r_i^{(t)}}{d_i} + 0.2 \cdot \frac{1}{N} + 0.8 \sum_{i \in \text{deadEnds}} \frac{r_i^{(t)}}{N}$$

            This is the score page j gets from pages linking to it, based on their score in the previous iteration, and from random teleports. The last part of the sum accounts for dead end by assigning an equal probability to get from each dead-end to any other page. deadEnds is the set of pages without outgoing links, including pages that you discovered but did not crawl.
        iii. Repeat this `numIters` times
    e. The final PageRank of each page j is its $r_j^{(\text{numIters})}$ – i.e., the score computed for it in the final iteration.
    f. The function returns a dict where the keys are URLs and the values are the final score of each URL, e.g., {'https://en.wikipedia.org/wiki/Andy_Ram': 0.1, 'https://en.wikipedia.org/wiki/Jonathan_Erlich': 0.05 …}
    g. Submit the code in the file *Q3.tennisRank.py*
    h. In the answers file, write: what was the page with highest PageRank you got when the function was executed on the output of tennisCrawler('https://en.wikipedia.org/wiki/Andy_Ram',xpaths_from_question1)? Is setting `numIters=100` sufficient for the ranking process to converge in this case? Explain briefly.

Tips:

- There is no need to keep all the score vectors $r^{(0)}, r^{(1)}, r^{(2)}, \ldots$ You can save just $r^{(t)}$ and $r^{(t+1)}$, which you need for updating the scores.