Parallel Programing Assignment 2

Submission Guidelines

To reduce likelihood of misunderstandings, please follow these guidelines:

1. Work can either be done individually or in pairs.
2. Submission is through the moodle (lamda) system.
3. Make sure that your solution (for part III) compiles and runs without any errors and warnings on BIU servers.
4. In the first line of every file you submit, write in a comment your id and full name. For example: "/ *123456789 Israela Israeli* /".
5. Not using parallel computation (i.e `<thread>/<pthread.h>` libraries for part III, and parallel algorithms for parts I,II) in even one task will result in an automatic 0 for the grade. In other words, sequential code is unacceptable.

General Background

The goal of this exercise is to practice both parallel thinking, and parallel programming (using c/c++ and `<pthread.h>/<thread>`)

- even though we worked only with c POSIX threads you are allowed to use `c++` threads and `c++` thread pool
- you are recommended to use a thread pool for the code part of the assignment, you may use the one we showed you, or you may use any other implementation of you choosing.
- This exercise is composed of 3 unrelated parts. The first 2 parts are about theoretical parallel algorithms, and the third part is about parallel coding in c/c++ .
- you **MUST** add -
  - pseudo code for each theoretical question.
  - explanation that shows why your algorithm is correct
- you **MUST** provide a makefile to compile your project with the dependencies you choose.

Environment & Submission

1. You will need to submit a pdf file named parts_I_II.pdf with the solution for parts I, II.
2. You will need to submit your code for part 3 in a zip file named part_III.zip
   This zip file must include all your code, and your makefile for the code
3. Your zip file must be organized in such a way, that when using the unzip command (in the terminal of the biu planet servers), all the files must be located directly inside the

same folder that contains the zip file. In other words, after opening a terminal in the folder that contains your zip file, and then running:

```
$ unzip part_III.zip
$ ls
```

The ls command has to print all your files, and not just list a directory where all your files are located.

> ⚠ **Warning**
>
> Make sure that after unzipping your file (as explained above) on the biu planet servers, then your file compiles with the "make" command with no errors, and after compiling, make sure that your code runs (on the biu planet servers).

Part 1 - A

Determine whether a given mathematical equation containing parentheses has balanced parentheses or not. Describe an algorithm that accepts the equation as input and returns true if the parentheses are balanced, and false otherwise.
Achieve a time complexity of O(log n).
Example:
Input: "(3 *(4 + 2)* - *1)*"
Output: true
Input: "((5 2) / (4 - 2)"
Output: false

- The equation may contain numbers, arithmetic operators (+, -, ∗, /), and parentheses.
- Balanced parentheses mean that every opening parenthesis has a corresponding closing parenthesis, and they are properly nested.

Part 1 - B

write an algorithm that given an array A and a number b finds the number of continuous sub arrays which their sum is b.

for example : given $A = [1, 7, 3, 5, 2, 6]$ and $b = 8$ the out put will be 3 for the sub arrays : $[1, 7], [3, 5], [2, 6]$

try to make this algorithm as efficient as possible.

Part 2

Given a rooted tree with n nodes, each node has a unique identifier ranging from 0 to n-1. Each node i has a pointer to its parent node, and the root node points to itself.

Your task is to determine, for each node i, its k-th ancestor (the ancestor at distance k from the node). If no such ancestor exists, return -1 for that node.

for example given the tree

```
    0
   / \
  1   2
 / \
3   4
```

Parent pointers: $[0, 0, 0, 1, 1]$
$k = 2$

The expected output for the k-th ancestors is: $[-1, -1, -1, 0, 0]$

- for this question you must provide a detailed pseudo code which emphasis the logic of the tree traversal. using a common pseudo code language is the best (you can check how some algorithms were described using pseudo code language in the lectures or practical sessions).
- try to achieve a parallel runtime of O(log k) for all cases.
- notice that the algorithm should work for general trees as well (not just binary).

Part 3 - Page Ranking algorithm

PageRank (PR) is an algorithm used in real-life by Google, to rank web pages in their search results. In a nutshell, according to Google, this is how the algorithm works:

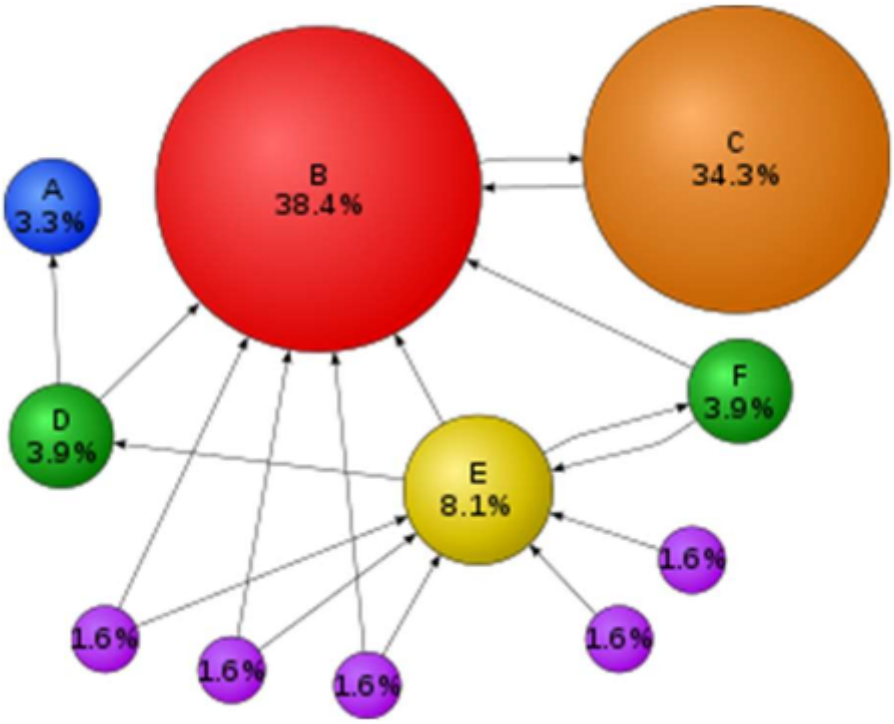
"PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites"

- To be precise, note that Google uses additional algorithms, but this algorithm is still used & it is the first one which was used!

In simple words, PR algorithm gives each web page a "score"; for a vertex (webpage) A, we mark its score, page rank, as: $PR(A)$. . The page rank is used to understand the relative importance of each web page, compared to other web pages. Higher PR $\rightarrow$ more relevant!

- Note that hyperlinks from a webpage to itself are ignored, and if there are multiple links between two web pages, we still have a single edge between them.

- Below is a simple illustration of the PR algorithm. The percentage shows the perceived importance, and the edges represent hyperlinks, as explained above:



n a perfect world, we could have just counted the number of links to a web page; however, in this case, as owners of an unpopular web page, **we could just create millions of dummy web pages linking to our web page → fool the system!**

The formula uses a model of a "random surfer" who reaches their target site after several clicks, then switches to a random web page. In simple words, the PageRank score of a web page reflects the chance that the random surfer will land on that web page by clicking on a link.

> **Additional explanation not required for the exercise, if you're interested to understand:**
>
> The PageRank theory holds that an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue following links is a damping factor d. The probability that they instead jump to any random page is 1 - d. Various studies have tested different damping factors, but it is generally assumed that the damping factor will be set around 0.85.

The sequential algorithm:

First, let's see the PR score formula for a single web page:

The PageRank Formula of a page v

$$Rank(v) = \frac{d}{N} + (1-d)\left(\sum_{u_i} \frac{Rank(u_i)}{outlink(u_i)} + \sum_{u_j} \frac{Rank(u_j)}{N}\right)$$

- d: Jump factor.
- N: Number of webpages
- u_i: Pages with links to v.
- u_j: Pages without outlinks.
- $Rank(u_i)$: The rank of the page u_i in the previous iteration.
- $outlinks(u_i)$: The number of pages u_i is pointing to.

Explanation for tricky parts:

- The first sigma: for all websites u_i that are pointing to v, we want to take their ranks and divide them by how many websites they are pointing to (and this indeed makes sense – if a page is pointing to a lot of other pages, the importance of each of that reference should decrease).
- The second sigma: relocating the probabilities for a vertex that doesn't have any outgoing links (edges). And it makes sense; if we land at a website without any outgoing links, we will have to quit the current page and type in the web address go to any other websites at random → that's why the sum is divided by N (number of webpages), because we want to uniformly redistribute the probabilities to all other websites.

Pseudocode for the sequential algorithm:
Here, $PageRank(G, n)$ receives as an input a graph G and n (the number of iterations). Notice that we initialize the rank of all pages to be $\frac{1}{N}$, where N is the total number of all web pages:

```
   PageRank(G, n)
1     d ← 0.15
2     N ← number of nodes in G
3     For all nodes v in G:
4        Initialize Rank(v) = 1/N
5     While n > 0:
6        For all v with links from u_i and u_j with no outlinks:
7           Rank(v) = d/N + (1 − d) · (∑ Rank(u_i)/outlink(u_i) + ∑ Rank(u_j)/N)
8        n = n - 1
9     return Rank
```

Runtime:

For a graph with N vertices and M directed edges, having n iterations of this algorithm will give a running time of $O(n * (N + M))$.

This is because in every iteration, we have to traverse every vertex and all edges that comes out of the node, which means that for each of the $i = 1, \ldots, N$ vertices, there are j steps to perform, where the sum of all j is M.

Why should we bother to parallelize?
The PageRank algorithm sound great and intuitive; however, there are nearly 2 billion websites online! It is clear that running such a sequential algorithm on a graph with 2 billion vertices is not a good idea, so what can we do? Parallelize it!

your task is to create a parallel page rank function which will compute the Rank array for each vertex in the Graph.

your API should contain only one function in the header file

```
void PageRank(Graph* g, n , float* rank) // result will be returned to rank array.
```

- you may write the function in `c` or `c++`
- don't submit a main file , we will take care of that, just make sure the Make file consider a main file when compiling the project.
- it is recommended to use a Thread pool for this code
- do not use `openMP` only `pthreads` or `std::thread`
- implement a struct graph for this project
  - you may use the implementation showed in the sessions or a one of your own.
  - make sure to name the struct you use Graph (for automation purposes)
  - for you convenience, a graph implementation will be uploaded as part of the exercises (you can modify it as much as you want)
  - think about a graph representation that can be more beneficial for this algorithm

Links with helpful information:
https://en.wikipedia.org/wiki/PageRank
https://churchill-aloha.medium.com/pagerank-algorithm-explanation-code-2fb6c0389bed
https://www.youtube.com/watch?v=meonLcN7LD4