Description of the code

```
6
7   beta = float(input("input beta:"))
8   iterator_times = int(input("input iterator time:"))
9   time_start=time.time()
10  total_nodes = 875713
11  Edges = 5105039
12
```

Line 7- 9: Prepare the variables. And prompt the users to input the value beta and iterator times. Also, I use python time.time() function to get the program start time.

Line 13 – 21: Read the file and initial some lists for store the data

I defined some functions .
The first function is that if the new node coming then we got True. Because we need use it to judge the new node coming or not. I checked the web_Google.file, it shows that like {0,1},{0,2},{0,3},{1,2}. If 0 is the start node, then he will be displayed centrally, which is useful for us to handle the probability of him going to each interface. For this case probability is 1/3, when 0->1,0->2,0->3. Only If the new nodes appear, I do the calculation, otherwise not.

```
def is_changed(node,old_node):
    if node == old_node:
        return False
    else:
        return True
```

Write all() and largest():
They are used to write output to Pagerank.txt and Top_10_nodes.txt separately.

Calculate():

```
def calulate(Matrix,v0,v1):
    v0 = Matrix*v0 + v1
    return v0
```

$$\mathbf{v}' = \beta M\mathbf{v} + (1-\beta)\mathbf{e}/n$$

```
for line in read_file:
    if line[0] != '#':
        split_line = line.strip().split('\t')
        '''
```

Read file.

Because the web_Google.txt file contains some comments at the beginning.

If '#' is the start character, then we ignore it. After comments, we should be split the line.

Split_line[0] is the start node, Split_line[1] is the link node.

```
    if flag == True:
        flag = False
        node = split_line[0]
        count = 1
        page_set.append(split_line)
```

Only read the first data line. Count is used for count how many links for one node has and use it for calculate the probability. Page_set[ ] is used to save every line of the one single node. Flag is making the program only read the first line.

Else: if the node[0] still not change, then I still save it to page_set[ ].

Else if : the new node coming, then it passes the function is_changed(). After that, we should be calculated the probability of older node. And save the data amounts for each row and the index of each row for sparse matrix.

```
    elif is_changed(split_line[0],node) and flag is not True:
        prob = 1 /(count)
        cor = beta * prob
        for i in range(count):
            str = (page_set[i])
            row.append(str[0])
            col.append(str[1])
            if max_node_num < int(str[0]):
                max_node_num = int(str[0])
            else:
                max_node_num = max_node_num
            if max_node2_num<int(str[1]):
                max_node2_num = int(str[1])
            else:
                max_node2_num = max_node2_num
            mtx_dt.append(cor)
        page_set.clear()
        page_set.append(split_line)
        node = split_line[0]
        count = 1
```

page_set Page.

Flag:

count:

e.g
0, 11342
0, 824020
0, 867923
0, 891835
} => same node

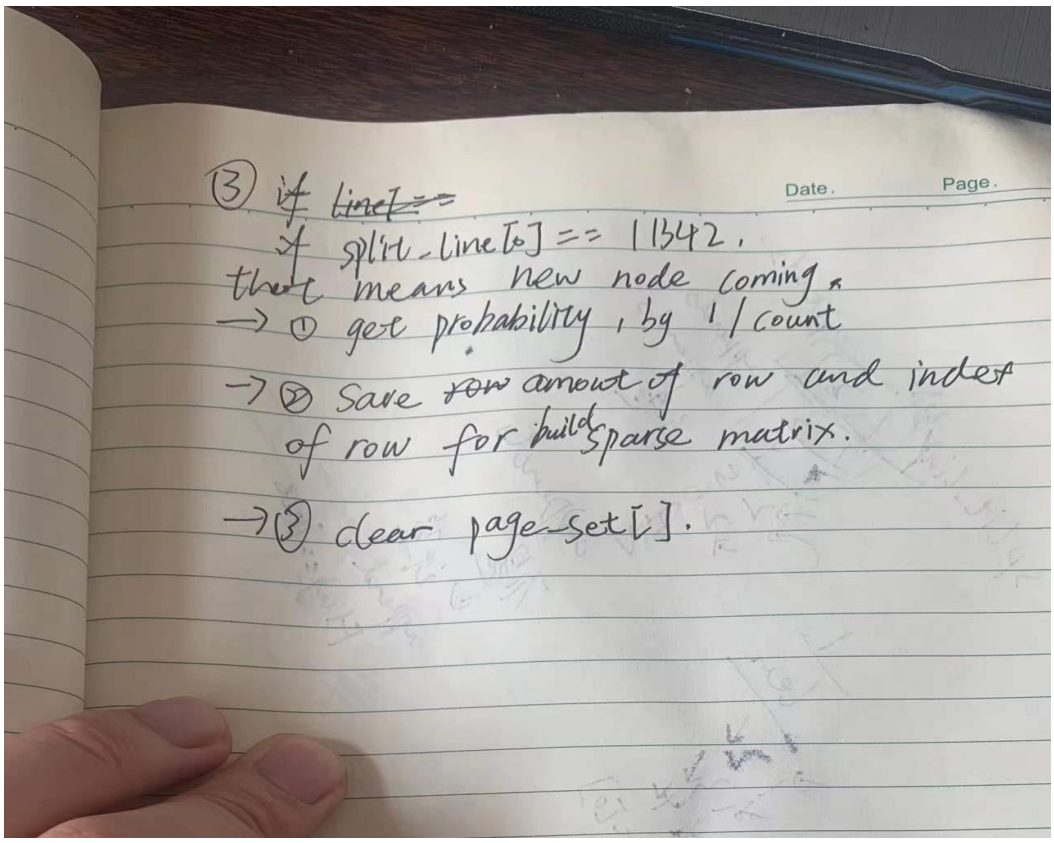
11342, 0
11342, 27469 } => new node coming
11569, 1

Case1: read first data
Case2: if the new node coming
Case3: for still the same node coming

① If Flag = True, read fist line and put it in page_set, Then change the flag to False. This flag just help me identify the node

② Suppose for case 3:
if split_line[0] == 0,
We think it they are same node
then put it to page_set
for first four line, the page_set[]

page_set[0] [0, 11342]
[1] ---
[2] ----
page_set[3] [0, 891835]



Because it contains millions of row node. I cannot build the matrix by the normal way. As a result, I used csr_matrix () to build the matrix.

```
max_node = max(max_node2_num,max_node_num)+1
Matrix = sparse.csr_matrix((mtx_dt,(col,row)), shape=(max_node, max_node))
```

Max_node is the max node. It used to for the sparse matrix shape.
Then get the Matrix.

Line 98-105:
Iterator the Matrix.

$$\mathbf{v}' = \beta M \mathbf{v} + (1 - \beta)\mathbf{e}/n$$

User can define the iterator times by themselves.
And sort the result from largest to smallest. Finally get the top_10 largest nodes.

Show the run time by float 3.

```
execution_time = round(execution_time_og,3)

print('time cost',execution_time_og,'second')
```