

# Web Ranking Algorithms<sup>\*</sup>

Zhiqiang Li<sup>1</sup>, Haoran Chen<sup>2,3</sup>, Xupeng Liu<sup>3</sup>, and Chengchen Rao<sup>4</sup>

South-east University

**Abstract.** Ranking pages plays a significant role in information retrieval. Search engine can list pages with highest relevance to the search key, seeks more websites which can be trusted. Web pages can also be categorized by their performance in linking (such as some hubs). We realize algorithms including PageRank, HITS, TrustRank and SALSA to measure different indicates of pages. After experiments, we evaluate their performance and precision. In the last, conclusions are drawn, which can contribute our further works. Other advanced methods should be studied and implement later in our plans.

**Keywords:** Search engine · PageRank · HITS · TrustRank · SALSA.

## 1 Problem definition

### 1.1 Problem Description

With the development of web science and computer science, how to filter reliable web site becomes a problem. The earliest search engines used the method of classified directory, which is manually classify web pages and sort out high-quality websites. Later, more and more web pages, artificial classification has become unrealistic. Search engines have entered the end of text retrieval, which computes the degree to which a user queries for keywords and web content to return search results. This method breaks the limit of the number, but the search results are not very good. Because there is always some web page back and forth some keywords to make their search ranking.

### 1.2 Related Work

The Google's founders, Larry Page and Sergey Brin, then graduate students at Stanford University, started working on sorting web pages. They drew on a common method used by academics to judge the importance of academic papers: the number of citations. It follows that the importance of web pages can also be evaluated in this way. the core idea of PageRank was born. The goal of personalized PageRank algorithm is to calculate the relevancy of all nodes with respect to user  $u$ . Start to walk from the node corresponding to user  $u$ , and each node will stop to walk with a 1-d probability and start again from  $u$ , or continue

---

<sup>\*</sup> Supported by organization x.

to walk with a probability of  $d$ , and choose a node from the node pointed by the current node to go downstream according to uniform distribution. In this way, after many rounds of swimming, the probability of each vertex being visited will converge and become stable, at which time we can use probability to rank. The calculation of subject-sensitive PageRank is mainly composed of two steps. The second step is to use the calculated topic PageRank score online to evaluate the similarity between the webpage and the user's query, so as to provide the user with the search results according to the similarity ranking. The following is a specific example to understand the calculation process of subject-sensitive PageRank. Hyperlink-Induced Topic Search (HITS; also known as hubs and authorities) is a link analysis algorithm that rates Web pages, developed by Jon Kleinberg. The idea behind Hubs and Authorities stemmed from a particular insight into the creation of web pages when the Internet was originally forming. Stochastic Approach for Link-Structure Analysis (SALSA) is a web page ranking algorithm designed by R. Lempel and S. Moran to assign high scores to hub and authority web pages based on the quantity of hyperlinks among them. TrustRank is a link analysis technique described in the paper Combating Web Spam with TrustRank by researchers Zoltan Gyongyi and Hector Garcia-Molina of Stanford University and Jan Pedersen of Yahoo!. The technique is used for semi-automatic separation of useful webpages from spam.

## 2 Algorithms

### 2.1 Data

Google Web graph

### 2.2 PageRank

The PageRank algorithm generally puts a PR value for each web page in advance (the PR value refers to the PageRank value). Since the PR value is physically a sense of the access probability of a web page, it is generally

$$\frac{1}{N} \quad (1)$$

, where  $N$  is the total number of web pages. In addition, in general, the sum of the PR values of all web pages is 1. If it is not 1, it is acceptable. Finally, the relationship between the PR values of different web pages is still correct, but the probability cannot be directly reflected. After the PR value is given in advance, iterates through the following algorithm until a smooth distribution is reached. There are three PageRank basic ideas. PageRank basic idea 1 is based on the situation that some webs have more than one chain. The formula below may not be accurate. Therefore, the PR value of  $A$  in this figure can be expressed as:

$$PR(A) = \frac{PR(B)}{k_{out}(B)} + \frac{PR(C)}{k_{out}(C)} \quad (2)$$

PageRank basic idea 2 is on another situation that some pages do not point to other webs, we use the formula 2 to calculate the page having no out chains. If the web only points to itself, the value of PR will always increase which is not reasonable. Therefore, the PR value of A in this figure can be expressed as:

$$PR(A) = \frac{PR(B)}{k_{out}(B)} + \frac{PR(C)}{N} \quad (3)$$

PageRank basic idea 3 is to solve this problem. To solve this problem. We imagine a person who browses the web randomly, and when he arrives at the C page, it is obviously not stupid to be trapped by the small tricks of the C page. We assume that he has a certain probability that the input URL will jump directly to a random web page, and the probability of jumping to each web page is the same. Therefore, the PR value of A in this figure can be expressed as:

$$PR(A) = \alpha \frac{PR(B)}{k_{out}(B)} + \frac{1 - \alpha}{N} \quad (4)$$

In general, the PR value of a web page is calculated as follows:

$$PR(A) = \alpha \sum_{P_j \in M_P} \frac{PR(p_j)}{L(p_j)} + \frac{1 - \alpha}{N} \quad (5)$$

According to the above formula, we can calculate the PR value of each web page, which is the final result when the iteration tends to be stable. The PageRank algorithm can be proved by Markov Convergence Theorem and here will not be explained too much.

### 2.3 HITS

Unlike the PageRank algorithm, the HITS algorithm runs after the user searches, so the HITS algorithm's processing object set is definitely much smaller. First, we need to determine this collection. The relationship between web pages in the whole Internet can be abstracted into a directed graph

$$G = (V, E) \quad (6)$$

When a search request is generated (may let the keyword be  $k$ ), we can take all the webpages containing the keyword  $k$ . The resulting set  $Q$  is the initial set and runs our HITS algorithm on this set. However, this collection has obvious drawbacks: this collection can be very large, large enough to contain millions of web pages, and this is obviously not the ideal collection size. So, we want to find a smaller set  $S$ , which satisfies the following conditions:  $S$  is indeed small enough.  $S$  contains a lot of pages related to the query.  $S$  contains many high quality authority pages. How to find this  $S$  collection? We assume that the user enters a keyword search and the search engine uses a text-based engine to search. Then we take the ranking (according to the relevance ranking) the top  $t$  ( $t$  generally

takes about 200) web pages as the initial collection, recorded as the root set  $R$ . This set satisfies the first two conditions we mentioned above, but it is far from satisfying the third condition. So we need to extend  $R$ . It is generally believed that a high-quality web page associated with a keyword, even if it is not in  $R$ , is likely to have some pages pointing to it in  $R$ . Based on this, our process of extending  $R$  is as follows (taken from Jon Kleinberg's paper): Subgraph( $q$ ,  $t$ ,  $d$ ): a query string.  $e$ : a text-based search engine.  $t$ ,  $d$ : natural numbers. Let  $R$  denote the top  $t$  results of  $e$  on  $q$ . Set  $S := R$ . For each page  $p \in R$  Let  $+(p)$  denote the set of all pages  $p$  points to. Let  $-(p)$  denote the set of all pages pointing to  $p$ . Add all pages in  $+(p)$  to  $S$ . If  $|(-(p))| \geq d$ , then Add all pages in  $-(p)$  to  $S$ . Else Add an arbitrary set of  $d$  pages from  $-(p)$  to  $S$ . End Return  $S$

At the beginning we let  $S = R$ . Then through the above method, we add all the pages pointed to by the web page in  $R$  to  $S$ , and then put a certain number of web pages pointing to the web pages in the  $R$  collection (the web page of each  $R$  can add up to  $d$  web pages pointing to it) is added to  $S$ . In order to ensure a suitable size of the  $S$  set,  $d$  cannot be too large, and is generally set to about 50. Usually, after the expansion, the size of the collection is 1000-5000 web pages, which satisfies the above three conditions. Before calculating the hub and authority values, we need to do a bit of  $S$ . We remove all links between pages under the same "domain name" (domain name refers to a website), because usually these links are only for people to switch between different pages under this website, such as navigation links within the website. In the HITS algorithm, these links are more likely to reflect the transfer relationship between the hub value and the authority value than the links between different websites. So we remove these links in the  $S$  collection to form a new set  $G$ . Now, you can start calculating the hub and authority values by 4. We use  $h(p)$  to represent the hub value of page  $p$ , and  $a(p)$  to represent the authority value of page  $p$ . First, the initial hub value  $h(p)$  of each page is 1 and the initial authority value  $a(p)$  is also 1. Then start the process of iterative calculation ( $n$  is the total number of pages in  $G$ ):

At the end of each iteration, normalization is required so that  $\sum h(p) = \sum a(p) = 1$ . The need for standardization will be explained in the Algorithm Proof section. We can set an iteration number upper limit  $k$  to control, or set a threshold, and the iteration ends when the change is less than the threshold. Then just return to the user a dozen or so pages with the authority value.

## 2.4 SALSA and TKC effect

Stochastic Approach for Link-Structure Analysis (SALSA) is a web page ranking algorithm designed by R. Lempel and S. Moran to assign high scores to hub and authority web pages based on the quantity of hyperlinks among them.

SALSA can be seen as an improvement of HITS.

It is computationally lighter since its ranking is equivalent to a weighted in/out degree ranking. The computational cost of the algorithm is a crucial factor since HITS and SALSA are computed at query time and can therefore sig-

nificantly affect the response time of a search engine. This should be contrasted with query-independent algorithms like PageRank that can be computed off-line.

SALSA is less vulnerable to the Tightly Knit Community (TKC) effect than HITS. A TKC is a topological structure within the Web that consists of a small set of highly interconnected pages. The presence of TKCs in a focused subgraph is known to negatively affect the detection of meaningful authorities by HITS.

The Twitter Social network uses a SALSA style algorithm to suggest accounts to follow.

## 2.5 TrustRank

TrustRank is a link analysis technique described in the paper Combating Web Spam with TrustRank by researchers Zoltan Gyongyi and Hector Garcia-Molina of Stanford University and Jan Pedersen of Yahoo!. The technique is used for semi-automatic separation of useful webpages from spam. Many web spam pages are created only with the intention of misleading search engines. These pages, chiefly created for commercial reasons, use various techniques to achieve higher-than-deserved rankings on the search engines' result pages. While human experts can easily identify spam, a manual review of the Internet is impractical.

One popular method for improving rankings is to increase the perceived importance of a document through complex linking schemes. Google's PageRank and other search ranking algorithms have been subjected to such manipulation.

TrustRank seeks to combat spam by filtering the web based upon reliability. The method calls for selecting a small set of seed pages to be evaluated by an expert. Once the reputable seed pages are manually identified, a crawl extending outward from the seed set seeks out similarly reliable and trustworthy pages. TrustRank's reliability diminishes with increased distance between documents and the seed set.

The logic works in the opposite way as well, which is called Anti-Trust Rank. The closer a site is to spam resources, the more likely it is to be spam as well.

The researchers who proposed the TrustRank methodology have continued to refine their work by evaluating related topics, such as measuring spam mass.

TrustRank algorithm can be described as follow:

As a first step, the algorithm calls function SelectSeed, which returns a vector  $s$ . The entry  $s(p)$  in this vector gives the "desirability" of page  $p$  as a seed page. In step (2) function Rank( $x, s$ ) generates a permutation  $x'$  of the vector  $x$ , with elements  $x'(i)$  in decreasing order of  $s(x'(i))$ . In other words, Rank reorders the elements of  $x$  in decreasing order of their scores.

Step (3) invokes the oracle function on the  $L$  most desirable seed pages. The entries of the static score distribution vector  $d$  that correspond to good seed pages are set to 1.

Finally, step (5) evaluates TrustRank scores using a biased PageRank computation with  $d$  replacing the uniform distribution. Note that step (5) implements a particular version of trust dampening and splitting: in each iteration, the trust score of a node is split among its neighbors and dampened by a factor

$$\alpha_B \tag{7}$$

---

```

function TrustRank
input
     $\mathbf{T}$     transition matrix
     $N$      number of pages
     $L$      limit of oracle invocations
     $\alpha_B$  decay factor for biased PageRank
     $M_B$    number of biased PageRank iterations
output
     $\mathbf{t}^*$    TrustRank scores
begin
    // evaluate seed-desirability of pages
    (1)  $\mathbf{s} = \text{SelectSeed}(\dots)$ 
    // generate corresponding ordering
    (2)  $\sigma = \text{Rank}(\{1, \dots, N\}, \mathbf{s})$ 
    // select good seeds
    (3)  $\mathbf{d} = \mathbf{0}_N$ 
    for  $i = 1$  to  $L$  do
        if  $\mathcal{O}(\sigma(i)) == 1$  then
             $\mathbf{d}(\sigma(i)) = 1$ 
    // normalize static score distribution vector
    (4)  $\mathbf{d} = \mathbf{d}/|\mathbf{d}|$ 
    // compute TrustRank scores
    (5)  $\mathbf{t}^* = \mathbf{d}$ 
    for  $i = 1$  to  $M_B$  do
         $\mathbf{t}^* = \alpha_B \cdot \mathbf{T} \cdot \mathbf{t}^* + (1 - \alpha_B) \cdot \mathbf{d}$ 
    return  $\mathbf{t}^*$ 
end

```

Fig. 1. TrustRank Algorithm

### 3 Experiments

#### 3.1 PageRank

We use Google web graph as our algorithm source, which contains 875713 nodes and 5105039 edges. Then we set stop condition about when iteration stops. We

```
# Directed graph (each unordered pair of nodes is saved once): web-Google.txt
# Webgraph from the Google programming contest, 2002
# Nodes: 875713 Edges: 5105039
# FromNodeId    ToNodeId
0    11342
0    824020
0    867923
0    891835
11342    0
11342    27469
11342    38716
11342    309564
11342    222178
```

Fig. 2.

set parameter bias which get the the sum of absolute value of the minus between pre-process node value and pro-process node value. If bias less than value we set, the iteration stop. Finally, we get outputs of our algorithms, including PageRank

```
    bias += abs(process_list[NI.GetId()] - original_list[NI.GetId()])
if bias <= 0.01:
    break
```

Fig. 3.

value list and page index which has the highest PageRank value.

#### 3.2 HITS

The input of HITS algorithm is also Google web graph. To measure when to stop, we introduce a parameter bias, which measure the difference between former hub list and latter hub list, former authority list and latter authority list. Finally, we get the outputs of this algorithm, two list. One is hub value list, the other is authority value list.

```

    bias = (float) 0.008990737543906807
    damp_factor = (float) 0.85
    original_list = (list) <class 'list'>: [2.5085825615024844e-05, 1.5227299041277566e-06, 2.68913175611924e-06, 5.438886476787059e-07, 6.281169813347545e-07, 1.27592996043555e-06, 1.712889f
    out_node = (TNGraphNode) <snap.TNGraphNode; proxy of <Swig Object of type 'TNGraphNode' at 0x000001F86C980810> >
    page_num = (int) 875713
    process_list = (list) <class 'list'>: [2.4844581841965065e-05, 1.513319795980814e-06, 2.6666703356778613e-06, 5.406920693108943e-07, 6.226503062769539e-07, 1.269568013020851e-06, 1.7128
    0000000 = (float) 2.4844581841965065e-05
    0000001 = (float) 1.513319795980814e-06
    0000002 = (float) 2.6666703356778613e-06
    0000003 = (float) 5.406920693108943e-07
    0000004 = (float) 6.226503062769539e-07
    0000005 = (float) 1.269568013020851e-06
    0000006 = (float) 1.7128899536720366e-07
    0000007 = (float) 4.5903896134596473e-07
    0000008 = (float) 3.3430120807090704e-07
    0000009 = (float) 1.8948845112496905e-07
    0000010 = (float) 2.2730084346252374e-06
    0000011 = (float) 5.226363081198658e-07
    0000012 = (float) 1.7128899536720366e-07
    0000013 = (float) 2.0284569670701207e-07
    0000014 = (float) 2.072559849405159e-07
    0000015 = (float) 2.0768790688273444e-07
    0000016 = (float) 3.434855809531197e-07
    0000017 = (float) 8.105391260504863e-07
    0000018 = (float) 1.7128899536720366e-07
    0000019 = (float) 2.533406478846446e-07
    0000020 = (float) 2.958959051604131e-07
    0000021 = (float) 7.782084452587508e-07
    0000022 = (float) 2.8526296331614176e-07
    0000023 = (float) 9.129193314937387e-06
    0000024 = (float) 1.7128899536720366e-07
    0000025 = (float) 1.7128899536720366e-07
    0000026 = (float) 1.7128899536720366e-07
    0000027 = (float) 1.7926446216781858e-06
    0000028 = (float) 9.371642646790033e-07

```

Fig. 4.

```

Connected to pydev debugger (build 191.6605.12)
0.0006582194468914516
597621
|

```

Fig. 5.



```

process_hub_list = (list) <class 'list'>: [2.3055379545768155e-06, 6.412548517369769e-05, 2.3115174361156127e-05, 3.411972347387698e-12, 4.697175666551019e-05, 2.9070140878637083e-06, 4.57340773443847...
0000000 = (float) 2.3055379545768155e-06
0000001 = (float) 6.412548517369769e-05
0000002 = (float) 2.3115174361156127e-05
0000003 = (float) 3.411972347387698e-12
0000004 = (float) 4.697175666551019e-05
0000005 = (float) 2.9070140878637083e-06
0000006 = (float) 4.57340773443847e-08
0000007 = (float) 5.471581135517801e-05
0000008 = (float) 8.387310424348443e-08
0000009 = (float) 3.468031053055277e-07
0000010 = (float) 2.4008684619628274e-06
0000011 = (float) 4.473191282651e-06
0000012 = (float) 3.411972347387698e-12
0000013 = (float) 0.00010490497264496614
0000014 = (float) 3.411972347387698e-12
0000015 = (float) 1.6413634174343264e-07
0000016 = (float) 5.953857716965177e-07

```

Fig. 6.

When bias is less than 1, we get the page which has the highest authority value is node 768091 and the other page which has the highest hub value is node 203748.

### 3.3 TrustRank

The input of this algorithm is top 55 pages from Google web graph ordered by PageRank.

Then we get outputs after processing and store them into a file.

Finally, we get 3824 pages which have high trust value.

### 3.4 SALSA

Firstly, we aim to generate two sets including hub set and authority set.

Then we form sets of connected graph by random walk. After getting each set, we find that it is not the complete set, so we use our approaches to refine them.

Finally, we calculate value by using existing formula. The outputs are below:

These are some results of SALSA value.

We can find that page 537039 has the most value in the first connected graph, with value 0.00023678...

## 4 Conclusion

In this paper, we use four algorithms to solve the problem of ranking pages. PageRank uses random walk model to simulate pages jumping, their ideas about

```

process_authority_list = [(list) <class 'list'>: [2.548162191767891e-05, 3.331322141046555e-05, 9.321578974264794e-06, 1.5405691138103454e-08, 4.1373510392404436e-07, 3.5677490411278123e-06, 8.1714799438304e-13, 3.0083957131607345e-06, 8.1714799438304e-13, 8.1714799438304e-13, 1.582395651050835e-06, 5.018963838900374e-07, 8.1714799438304e-13, 1.249068544070073e-06, 2.48650780358812e-07, 8.1714799438304e-13]]
0000000 = (float) 2.548162191767891e-05
0000001 = (float) 3.331322141046555e-05
0000002 = (float) 9.321578974264794e-06
0000003 = (float) 1.5405691138103454e-08
0000004 = (float) 4.1373510392404436e-07
0000005 = (float) 3.5677490411278123e-06
0000006 = (float) 8.1714799438304e-13
0000007 = (float) 3.0083957131607345e-06
0000008 = (float) 8.1714799438304e-13
0000009 = (float) 8.1714799438304e-13
0000010 = (float) 1.582395651050835e-06
0000011 = (float) 5.018963838900374e-07
0000012 = (float) 8.1714799438304e-13
0000013 = (float) 1.249068544070073e-06
0000014 = (float) 2.48650780358812e-07
0000015 = (float) 8.1714799438304e-13

```

Fig. 7.

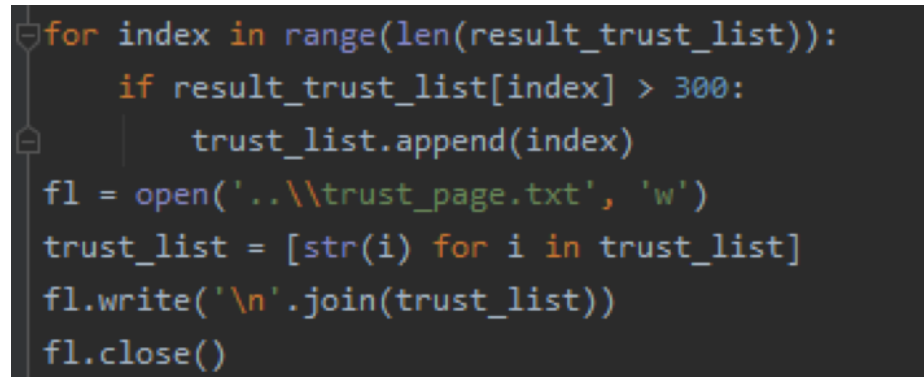
```

0.03338905469080529
768091
0.051840817967991545
203748

```

Fig. 8.

1	7314
2	21682
3	32163
4	41909
5	54147
6	57791
7	74397
8	163075
9	172133
10	173976
11	182121
12	185067
13	185821
14	191990
15	213432
16	226374
17	245186
18	277876
19	323129
20	324502
21	384666
22	396321
23	399699
24	402132



```

for index in range(len(result_trust_list)):
    if result_trust_list[index] > 300:
        trust_list.append(index)
fl = open('../trust_page.txt', 'w')
trust_list = [str(i) for i in trust_list]
fl.write('\n'.join(trust_list))
fl.close()

```

Fig. 10.

which page is more significant are heuristic. TrustRank chooses some pages which have high PageRank value and spreads trust value by random walking. HITS introduces two categories including hub and authority and tries to use these two categories to reinforce each other. Finally, we implement SALSA algorithm, which combine random walk and categories. Link relation propagation process helps a lot. Through this task, our understanding of distance is more profound. Algorithms implements and comprehension are more skilled. Our team has gained a lot.

## References

1. Author, F.: Article title. Journal **2**(5), 99–110 (2016)
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). <https://doi.org/10.1007/1234567890>
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, <http://www.springer.com/lncs>. Last accessed 4 Oct 2017

3803	911565
3804	911571
3805	912024
3806	912558
3807	912758
3808	912784
3809	912974
3810	913287
3811	913499
3812	913835
3813	913924
3814	914086
3815	914182
3816	914299
3817	914377
3818	914393
3819	915049
3820	915641

```
for NI in web_google_graph.Nodes():  
    #hub_set  
    if NI.GetOutDeg() != 0:  
        hub_list.append(NI.GetId())  
    #authority_set  
    if NI.GetInDeg() != 0:  
        authority_list.append(NI.GetId())
```

Fig. 12.

```

for authority_id in authority_list:
    if has_set[authority_id] == 1:
        continue
    to_hub_id = []
    origin_current_set = set()
    origin_current_set.add(authority_id)
    result_current_set = set()
    authority_node = web_google_graph.GetNI(authority_id)
    #get hub node
    for hub_id in authority_node.GetInEdges():
        if hub_id in hub_list:
            to_hub_id.append(hub_id)
    for hub_id in authority_node.GetOutEdges():
        if hub_id in hub_list:
            to_hub_id.append(hub_id)

    #return to authority node
    for hub_id in to_hub_id:
        hub_node = web_google_graph.GetNI(hub_id)
        #get authority set
        for new_authority_id in hub_node.GetInEdges():
            if new_authority_id in authority_list:
                origin_current_set.add(new_authority_id)
        for new_authority_id in hub_node.GetOutEdges():
            if new_authority_id in authority_list:
                origin_current_set.add(new_authority_id)

```

Fig. 13.

```

#refine_result_current_set
result_current_set = origin_current_set.copy()
while True:
    origin_current_set = result_current_set.copy()
    for authority_id in origin_current_set:
        to_hub_id = []
        authority_node = web_google_graph.GetNI(authority_id)
        # get hub node
        for hub_id in authority_node.GetInEdges():
            if hub_id in hub_list:
                to_hub_id.append(hub_id)
        for hub_id in authority_node.GetOutEdges():
            if hub_id in hub_list:
                to_hub_id.append(hub_id)
        # return to authority node
        for hub_id in to_hub_id:
            hub_node = web_google_graph.GetNI(hub_id)
            # get authority set
            for new_authority_id in hub_node.GetInEdges():
                if new_authority_id in authority_list:
                    result_current_set.add(new_authority_id)
            for new_authority_id in hub_node.GetOutEdges():
                if new_authority_id in authority_list:
                    result_current_set.add(new_authority_id)
        if len(result_current_set) == len(origin_current_set):
            break

```

Fig. 14.



```

#calculate value
authority_sum = len(authority_list)
connected_sum = len(result_current_set)
in_connected_sum = 0
for id in result_current_set:
    has_set[id] = 1
    in_node_sum = 0
    node = web_google_graph.GetNI(id)
    for hub_id in node.GetInEdges():
        if hub_id in hub_list:
            in_node_sum += 1
            in_connected_sum += 1
    for hub_id in node.GetOutEdges():
        if hub_id in hub_list:
            in_node_sum += 1
            in_connected_sum += 1
    result[id] = in_node_sum
#calculate each element value in set
for id in result_current_set:
    result[id] = (connected_sum * result[id]) / (authority_sum * in_connected_sum)

```

Fig. 15.

```

#calculate value
authority_sum = len(authority_list)
connected_sum = len(result_current_set)
in_connected_sum = 0
for id in result_current_set:
    has_set[id] = 1
    in_node_sum = 0
    node = web_google_graph.GetNI(id)
    for hub_id in node.GetInEdges():
        if hub_id in hub_list:
            in_node_sum += 1
            in_connected_sum += 1
    for hub_id in node.GetOutEdges():
        if hub_id in hub_list:
            in_node_sum += 1
            in_connected_sum += 1
    result[id] = in_node_sum
#calculate each element value in set
for id in result_current_set:
    result[id] = (connected_sum * result[id]) / (authority_sum * in_connected_sum)

```

Fig. 16.

```
01 0000007 = {int} 0
01 0000088 = {float} 1.043745951406959e-06
01 0000089 = {float} 1.0810225925286363e-06
01 0000090 = {int} 0
01 0000091 = {int} 0
01 0000092 = {float} 7.455328224335423e-08
```

Fig. 17.