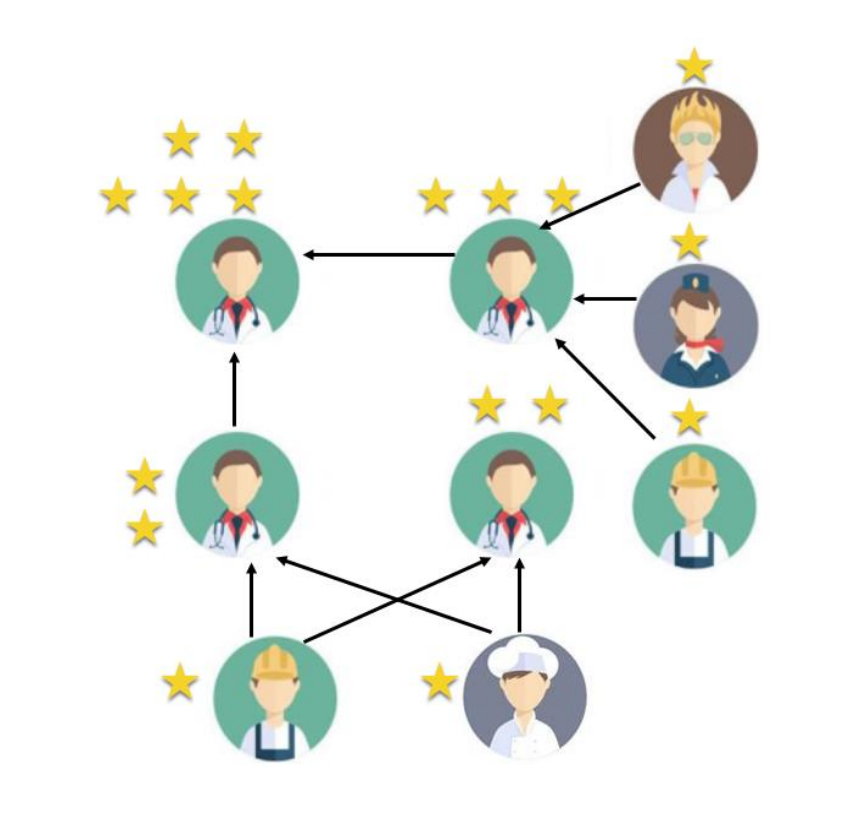
PageRank

软件学院《数据结构》讲义
内部使用

引例：社交网络中的影响力分析



Paper

Sergey Brin received his B.S. degree in mathematics and computer science from the University of Maryland at College Park in 1993. Currently, he is a Ph.D. candidate in computer science at Stanford University where he received his M.S. in 1995. He is a recipient of a National Science Foundation Graduate Fellowship. His research interests include search engines, information extraction from unstructured sources, and data mining of large text collections and scientific data.

Lawrence Page was born in East Lansing, Michigan, and received a B.S.E. in Computer Engineering at the University of Michigan Ann Arbor in 1995. He is currently a Ph.D. candidate in Computer Science at Stanford University. Some of his research interests include the link structure of the web, human computer interaction, search engines, scalability of information access interfaces, and personal data mining.

The Anatomy of a Large-Scale Hypertextual Web Search Engine

Sergey Brin and Lawrence Page

Computer Science Department,
Stanford University, Stanford, CA 94305, USA
sergey@cs.stanford.edu and page@cs.stanford.edu

We assume page A has pages T1...Tn which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also C(A) is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

$$PR(A) = (1-d) + d\,(PR(T1)/C(T1) + \ldots + PR(Tn)/C(Tn))$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one.

Patent

(12) **United States Patent**
Page

(10) **Patent No.:** **US 7,058,628 B1**
(45) **Date of Patent:** ***Jun. 6, 2006**

(54) **METHOD FOR NODE RANKING IN A LINKED DATABASE**

(75) Inventor: **Lawrence Page**, Stanford, CA (US)

(73) Assignee: **The Board of Trustees of the Leland Stanford Junior University**, Palo Alto, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 622 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/895,174**

(22) Filed: **Jul. 2, 2001**

Related U.S. Application Data

(63) Continuation of application No. 09/004,827, filed on Jan. 9, 1998.

(60) Provisional application No. 60/035,205, filed on Jan. 10, 1997.

5,754,939	A	5/1998	Herz et al.	455/4.2
5,832,494	A	11/1998	Egger et al.	
5,848,407	A	12/1998	Ishikawa et al.	
5,915,249	A	6/1999	Spencer	
5,920,854	A	7/1999	Kirsch et al.	
5,920,859	A	7/1999	Li	707/5
6,014,678	A	1/2000	Inoue et al.	
6,112,202	A	8/2000	Kleinberg	707/5
6,163,778	A	12/2000	Fogg et al.	707/10
6,269,368	B1	7/2001	Diamond	707/6
6,285,999	B1	9/2001	Page	707/5
6,389,436	B1	5/2002	Chakrabarti et al.	707/513
2001/0002466	A1	5/2001	Krasle	704/270.1

OTHER PUBLICATIONS

Yuwono et al., "Search and Ranking Algorithms for Locating Resources on the World Wide Web", IEEE 1996, pp. 164-171.
L. Katz, "A new status index derived from sociometric analysis", 1953, Psychometricka, vol. 18, pp. 39-43.
C.H. Hubbell, "An input-output approach to clique identification sociometry", 1965, pp. 377-399.

(Continued)

Primary Examiner—Uyen Le
(74) *Attorney, Agent, or Firm*—Harrity Snyder, LLP

Application US09/895,174 events

1997-01-10	Priority to US3520597P
2001-07-02	Application filed by Leland Stanford Junior University
2006-06-06	Application granted
2006-06-06	Publication of US7058628B1
2019-09-23	Adjusted expiration
Status	Expired - Lifetime

Show all events

PageRank算法

$$\mathrm{PR(u)} = \sum_{v \in D(u)} \frac{PR(v)}{|S(v)|}$$

PR(u)代表网页u的rank值

PR(v)代表网页v的rank值

D(u)代表指向u的所有网页，也就是网页图中u的入边源点集合

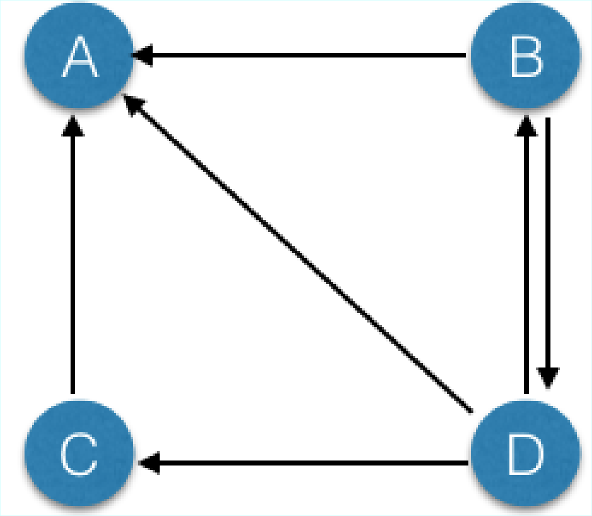
S(v)代表从v出发指向的网页集合，|S(v)|就是该集合的大小。

通过该公式计算，就可以得出网页集合中每一个网页的rank值。

PageRank算法

$$\mathrm{PR(u)} = \sum_{v \in D(u)} \frac{PR(v)}{|S(v)|}$$

阻尼系数d对于自引用页面的意义：阻尼系数赋予了用户继续点击页面与不再点击两种事件不同的比例，从而限制了自引用页面对于pagerank算法合理性的影响(因为如果不加阻尼系数)来限制的话，可能会出现最终经过迭代某个页面的PR值变为1的情况。



$$\mathrm{PR(A)} = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}$$

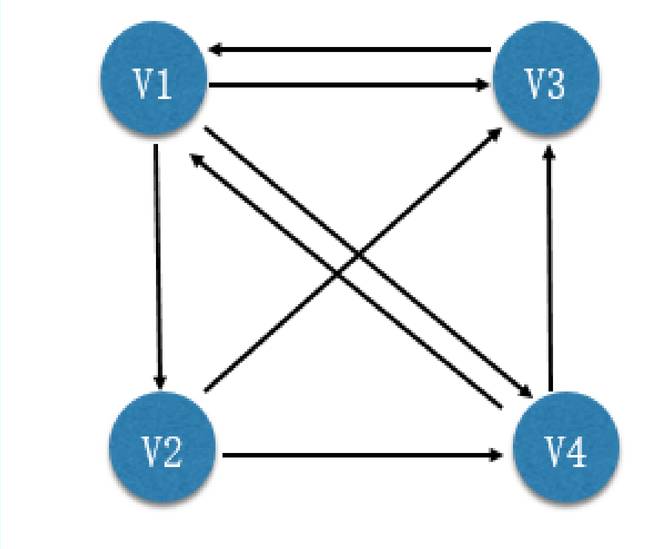
$$\mathrm{PR(u)} = 1 - \mathrm{d} + \mathrm{d} * \sum_{v \in D(u)} \frac{PR(v)}{|S(v)|}$$

阻尼系数d，d表示用户到达某个页面时继续浏览的概率，那么1-d就是用户停止点击的概率

$$\mathrm{PR(A)} = 0.15 + 0.85 * \left(\frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}\right)$$

迭代式的PageRank计算

$$\mathrm{PR}(u) = \sum_{v \in D(u)} \frac{PR(v)}{|S(v)|}$$



PR(v1)	PR(v2)	PR(v3)	PR(v4)
0.25	0.25	0.25	0.25

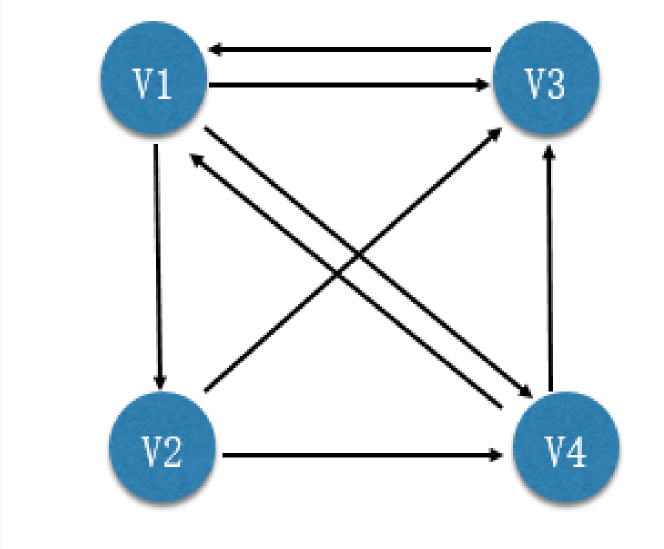
$$\mathrm{PR(v1)} = 0.25 * 1 + 0.25 * \frac{1}{2} = 0.37$$

$$\mathrm{PR(v2)} = 0.25 * \frac{1}{3} = 0.08$$

$$\mathrm{PR(v3)} = 0.25 * \frac{1}{3} + 0.25 * \frac{1}{2} + 0.25 * \frac{1}{2} = 0.33$$

$$\mathrm{PR(v4)} = 0.25 * \frac{1}{3} + 0.25 * \frac{1}{2} = 0.20$$

迭代式的PageRank计算



运行PageRank 4次后

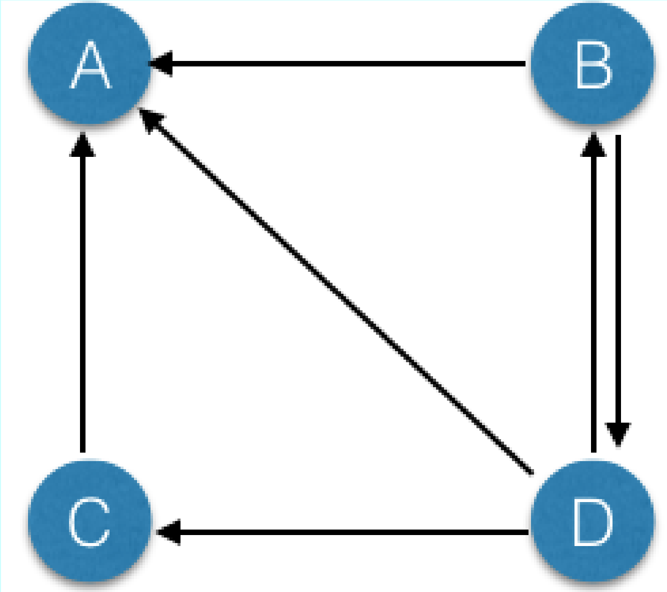
	Initial	Iter = 1	Iter = 2	Iter = 3	Iter = 4
PR(v1)	0.25	0.37	0.43	0.35	0.39
PR(v2)	0.25	0.08	0.12	0.14	0.11
PR(v3)	0.25	0.33	0.27	0.29	0.29
PR(v4)	0.25	0.20	0.16	0.20	0.19

运行PageRank 6次后

	Initial	Iter = 1	Iter = 2	Iter = 3	Iter = 4	Iter = 5	Iter = 6
PR(v1)	0.25	0.37	0.43	0.35	0.39	**0.39**	**0.38**
PR(v2)	0.25	0.08	0.12	0.14	0.11	**0.13**	**0.13**
PR(v3)	0.25	0.33	0.27	0.29	0.29	**0.28**	**0.28**
PR(v4)	0.25	0.20	0.16	0.20	0.19	**0.19**	**0.19**

基于矩阵的实现

矩阵实现优点：简单易行，并且很直观
缺点：PR问题一般会用到的矩阵为稀疏矩阵，因而直接使用矩阵进行存储会出现浪费较多空间的问题。



	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	0	0	1
D	0	1	0	0

	A	B	C	D
A	0	1/2	1	1/3
B	0	0	0	1/3
C	0	0	0	1/3
D	0	1/2	0	0

$$\mathrm{PR}(u) = \sum_{v \in D(u)} \frac{PR(v)}{|S(v)|}$$

直接使用1/S来表示每一条边的取值，从而使这个矩阵与PR值矩阵的乘积的结果就是新的PR值矩阵。

```cpp
template <typename T>
struct Row{
    int n;
    T *rdata;
    Row(){
        n = 0;
        rdata = NULL;
    }
    void set_row(int n, T *rdata){
        assert(n > 0);
        this->n = n;
        this->rdata = rdata;
    }
    T& operator[](int j){
        assert(j >= 0 && j < n);
        return rdata[j];
    }
    ~Row(){
        rdata = NULL;
    }
};
```

```cpp
template <typename T>
class Matrix{
protected:
    // m rows and n cols
    int m;
    int n;
    Row<T> *data;
public:
    Matrix(){
        m = n = 0;
        data = NULL;
    }
    Matrix(int m, int n, const T *_data = NULL){
        initialize(m, n, _data);
    }
```

```
void initialize(int m, int n, const T *_data){
    // ensure parameter _m and _n are all valid
    assert(m > 0 && n > 0);
    this->m = m;
    this->n = n;
    T *buf;
    assert((buf = (T*)calloc(m * n, sizeof(T))) != NULL);
    if(_data){
        memcpy(buf, _data, sizeof(T) * m*n);
    }
    // allocate memory for each row in a matrix
    assert((data = new Row<T>[m]) != NULL);
    for(int i = 0; i < m; ++i){
        data[i].set_row(n, buf+i*n);
    }
}
```

```
Matrix(const Matrix<T> &other){
    if(&other == this)return;
    this->m = other.row_size();
    this->n = other.col_size();
    // empty, don't need to copy data
    if(m <= 0 || n <= 0)return;
    initialize(m, n, NULL);
    for(int i = 0; i < m; ++i){
        for(int j = 0; j < n; ++j){
            data[i][j] = other[i][j];
        }
    }
}
```

```
Row<T>& operator[](int i) const{
        assert(i >= 0 && i < m);
        return data[i];
    }
```

```
Matrix operator*(const Matrix<T> &other){
        // precondtion
        assert(n == other.row_size());
        // res is result to be returned
        int res_col = other.col_size();
        // create res to store result
        Matrix<T> res(m, res_col);
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < res_col; ++j){
                res[i][j] = 0;
                // matrix A, B: sum of aik * bkj, k from 1 to n
                for(int k = 0; k < n; ++k){
                    res[i][j] += data[i][k] * other[k][j];
                }
            }
        }
        // overload operator=
        return res;
    }
```

使用了深拷贝！具体的深拷贝与浅拷贝的区别看那个课本版本的讲义。

```
// avoid shadow copy
    void operator=(const Matrix<T> &other){
        if(&other == this)return;
        // this is not empty
        if(this->m != 0){
            clear();
        }
        this->m = other.row_size();
        this->n = other.col_size();
        // empty, don't need to copy data
        if(m <= 0 || n <= 0)return;
        initialize(m, n, NULL);
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                data[i][j] = other[i][j];
            }
        }
    }
```

```
void operator+=(const Matrix<T> &other){
        // allow add itself
        if(m != other.row_size() || n != other.col_size())return;
        assert(m > 0 && n > 0 && data != NULL);
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                data[i][j] += other[i][j];
            }
        }
    }
```

```cpp
// avoid shadow copy
    void operator=(const Matrix<T> &other){
        if(&other == this)return;
        // this is not empty
        if(this->m != 0){
            clear();
        }
        this->m = other.row_size();
        this->n = other.col_size();
        // empty, don't need to copy data
        if(m <= 0 || n <= 0)return;
        initialize(m, n, NULL);
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                data[i][j] = other[i][j];
            }
        }
    }
```

```cpp
void operator+=(const Matrix<T> &other){
        // allow add itself
        if(m != other.row_size() || n != other.col_size())return;
        assert(m > 0 && n > 0 && data != NULL);
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                data[i][j] += other[i][j];
            }
        }
    }
```

```
int main(int argc, char * argv[]){
    int row1 = 6; int col1 = 6;
    Matrix<float> H(row1, col1, H_arr);
    // H.print();
    float R_arr[] = {0.2,
                     0.2,
                     0.2,
                     0.2,
                     0.2,
                     0.2};
    int row2 = 6; int col2 = 1;
    Matrix<float> R(row2, col2, R_arr);
    // basic way
    for(int i = 1; i <= iters; ++i){
        Matrix<float> res = H * R;
        printf("iter %d res is :\n", i);
        res.print();
        R = res;
    }
```

并行方法

```cpp
// get idx range of a row or a col
std::pair<int, int> get_range(int sub_id, int partitions, int total_len){
    assert(partitions > 0 && total_len > 0);
    int each_siz = total_len / partitions;
    int start = sub_id * each_siz;
    // > total_len is impossible
    int end = (sub_id + 1) * each_siz;
    // special case is the last partition
    end = (sub_id == partitions - 1)?total_len:end;
    return std::make_pair(start, end);
}
```

并行方法

```
for(int i = 1; i <= iters; ++i){
        Matrix<float> res(row1, col2);
        // rows are divided into partitions, thread num is partitions
        std::vector<std::thread> threads;
        threads.clear();
        for(int i = 0; i < partitions; ++i){
                threads.emplace_back([&](int th_i){
                        std::pair<int, int> l_rows = get_range(th_i, partitions, row1);
                        // sub1 is part of H_arr with fixed row range
                        int rows = l_rows.second - l_rows.first;
                        Matrix<float> sub1(rows, col1, H_arr+col1*l_rows.first);
                        // sub1 * R, sub_res is rows * col2
                        Matrix<float> sub_res = sub1 * R;
                        // copy data in sub_res into corresponding row range in res
                        for(int i = l_rows.first; i < l_rows.second; ++i){
                            for(int j = 0; j < col2; ++j){
                                res[i][j] = sub_res[i - l_rows.first][j];
                            }
                        }
                }, i);
        }
```

为何没有使用锁?

因为不同分组所包含的内容之间相互不影响。
因为每一行都是一个不同的点(网页)。

lambda 表达式

lambda 表达式定义了一个匿名函数，并且可以捕获一定范围内的变量。lambda 表达式的语法形式可简单归纳如下：

[capture] (params) opt -> ret { body; };

其中 capture 是捕获列表，params 是参数表，opt 是函数选项，ret 是返回值类型，body是函数体。

[capture]：1.[&]:表示以引用的方式传递参数
1.[=]：表示以拷贝的方式传递参数。

一个完整的 lambda 表达式看起来像这样：

```
auto f = [](int a) -> int { return a + 1; };
std::cout << f(1) << std::endl; // 输出: 2
```

Source: http://c.biancheng.net/view/3741.html

lambda 表达式

这个被嵌在lambda函数里面的特殊类类是函子

```
class CountEven
{
        int& count_;
        public:
        CountEven(int& count) : count_(count) {}
        void operator()(int val)
        {
                if (!(val & 1)) // val % 2 == 0
                {
                        ++ count_;
                }
        }
};
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each(v.begin(), v.end(), CountEven(even_count));
std::cout << "The number of even is " << even_count << std::endl;
```

Source: http://c.biancheng.net/view/3741.html

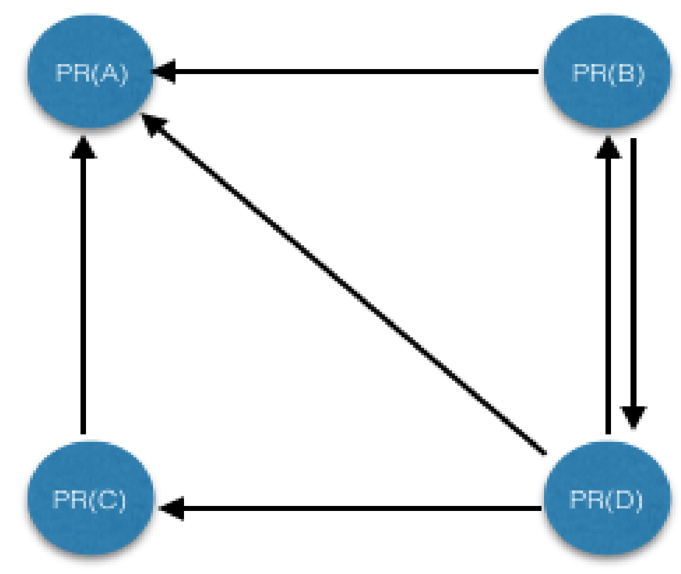
[&even_count]是在引用上面定义的event_count，定义的val就是前面的迭代的v中的数值

lambda 表达式

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each( v.begin(), v.end(), [&even_count](int val)
{
        if (!(val & 1)) // val % 2 == 0
        {
                ++ even_count;
        }
});
std::cout << "The number of even is " << even_count << std::endl;
```

Source: http://c.biancheng.net/view/3741.html

基于图结构的实现



G=(V,E,D)模型

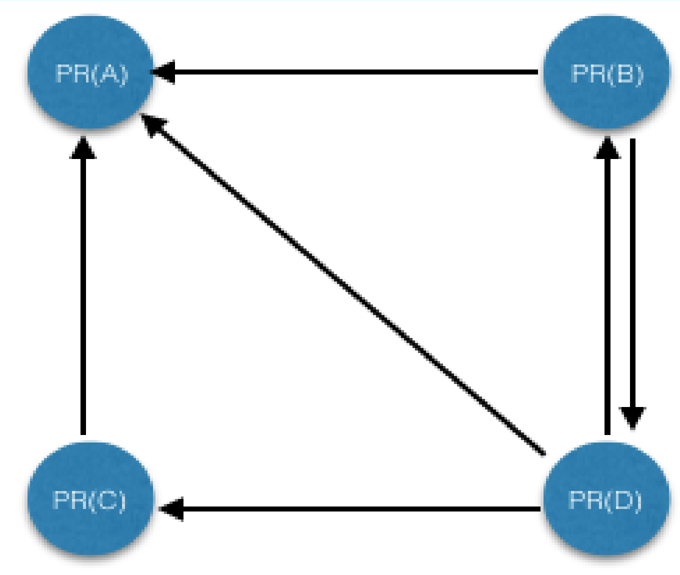
计算出度

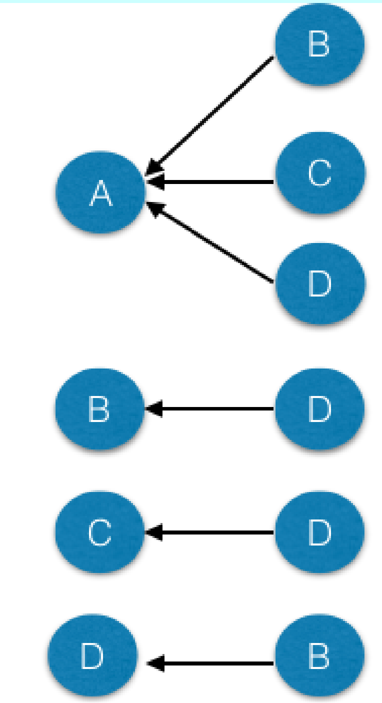
```
procedure PR_F(u, v)
    v.rank += u.rank / out_deg(u)
procedure VertexMap(u)
    u = 1-d + d*u.rank
procedure PageRank(G = (V, E, D))
    while i < iters
        for each v in V
            for each ngh u that satisfies (u, v) in E
                PR_F(u, v)
            VertexMap(v)
        i+=1
```

给每一个点赋一个阻尼系数，避免前面提到的自引用现象对于PR结果的影响。

基于图结构的实现

CSC: compressed sparse column，用紧凑压缩的方法来记录各个顶点的入边





源点数组

B	C	D	D	D	B

入度数组

3	1	1	1

源点范围数组

0	3	4	5	6

使用上述的紧凑压缩方法，可以减少不必要的存储空间。

基于图结构的实现-优化

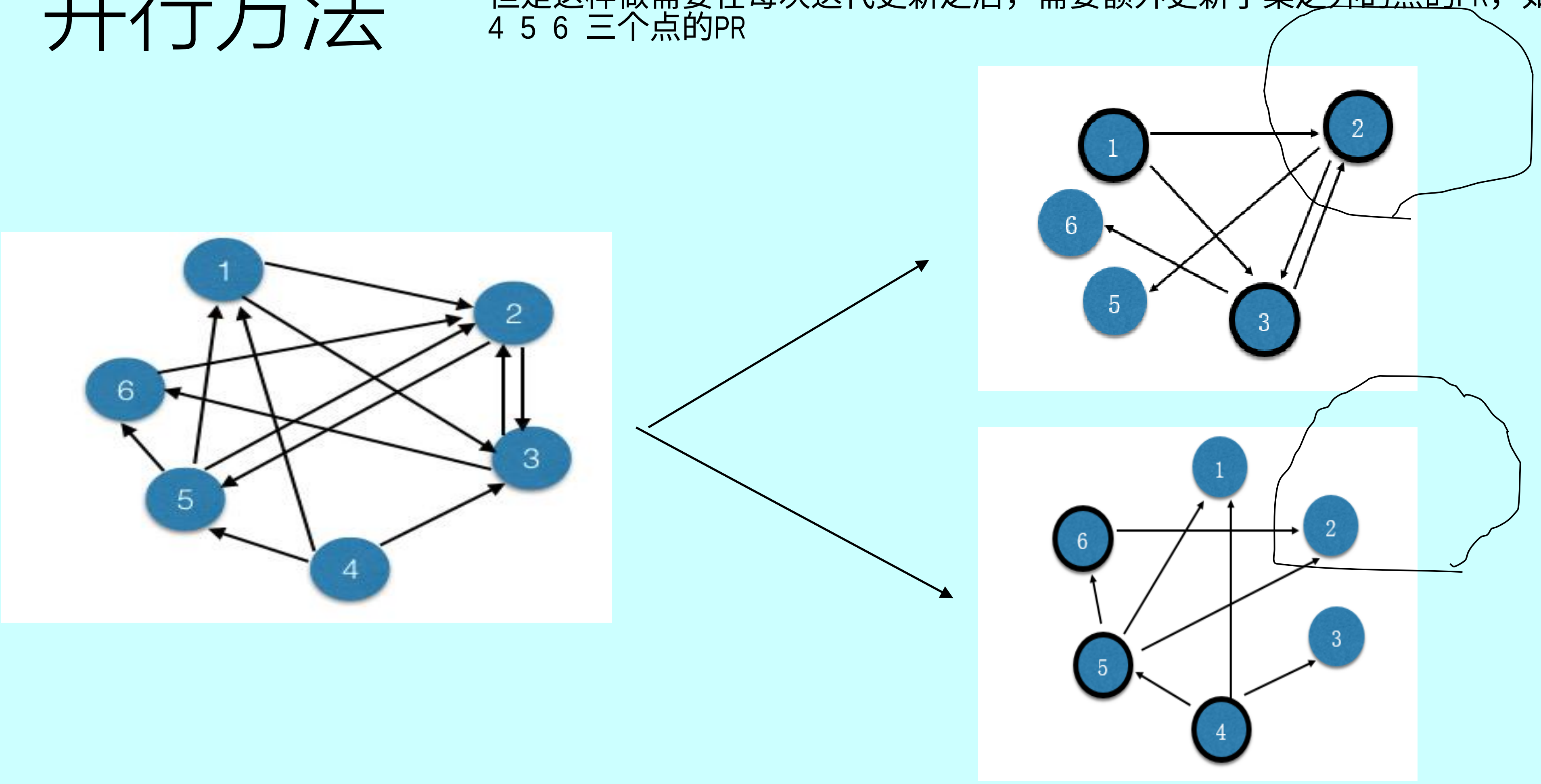
当某个顶点的rank值发生变化时，它的所有出边顶点的rank值自然需要被更新，也就是说所有活跃顶点的出边构成了活跃边集合。所以我们可以让活跃顶点主动的去更新所有邻居

```
procedure PageRank(G = (U, E, D))
    while i < iters
        // push mode
        for each active vertex u in U
            for each ngh v that satisfies (u, v) in E
               PR_F(u, v)
            VertexMap(v)
        i+=1
```

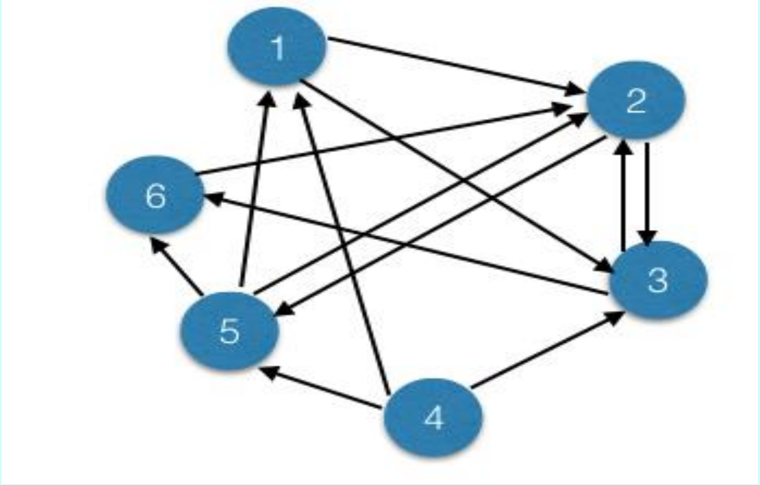
- push模型更适合活跃顶点较少的情况
- 在多线程情况下，push模式可能会同时对某个顶点进行更新（同步开销）

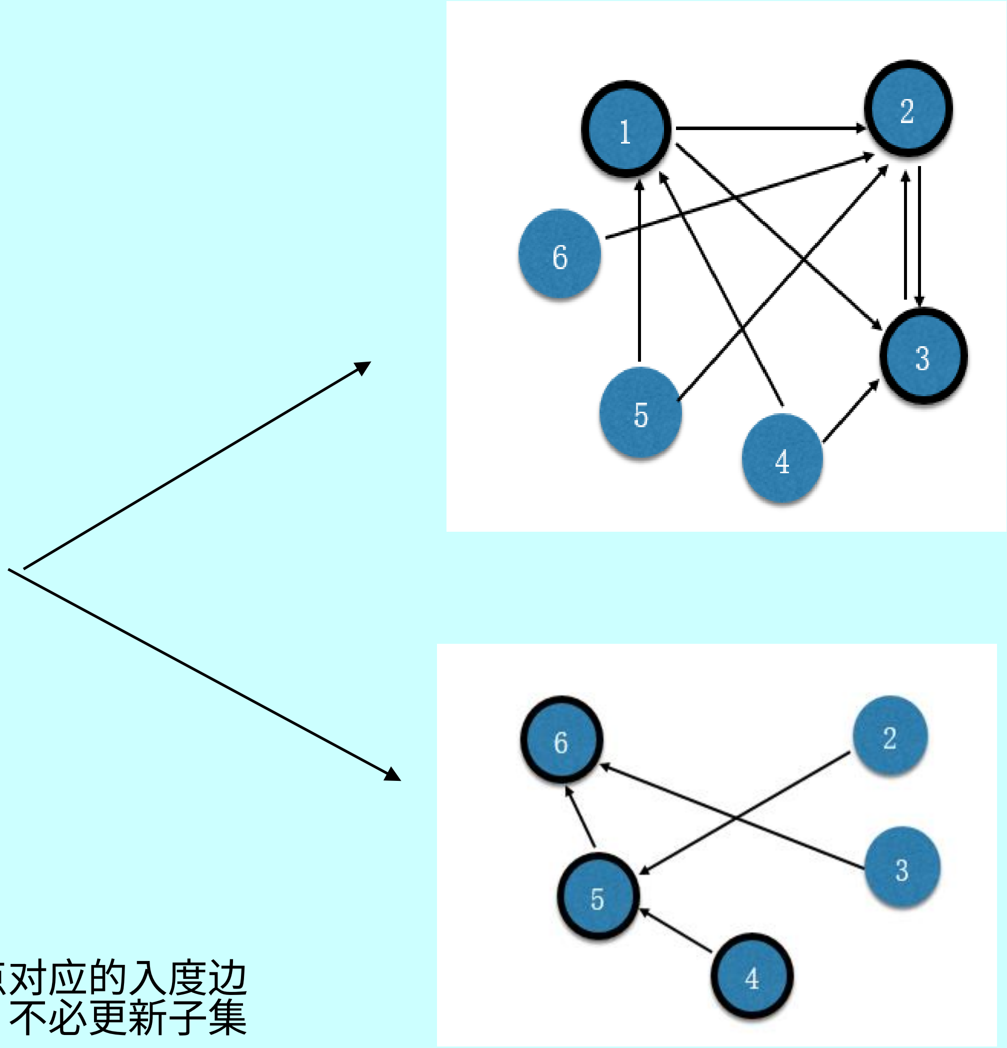
并行方法

第一种并行化的思路：
首先对所有点进行分组，之后将每个点的出度边包含在这个子集对应的thread里面
但是这样做需要在每次迭代更新之后，需要额外更新子集之外的点的PR，如图中的
4 5 6 三个点的PR



并行方法-改进





第二种并行化思路：
首先将点进行分组生成不同的点集，并将每个点对应的入度边考虑进去。这样可以使得每次进行PR值更新时，不必更新子集外的点的PR值。

Next

- Review