

第十四章 PageRank

14.1 引例：社交网络中的影响力分析

PageRank [<https://en.wikipedia.org/wiki/PageRank>] [Page L, et al., 1999] 是针对网页链接的一种分析算法，它为网页集合内的每个网页分配权重，称为 rank 值，目的是为了衡量集合内网页的相对重要性。

除了网页排名外，PageRank 在社交网络中也有很好的应用。例如某个用户希望通过新浪微博找到医疗领域中最具影响力的专家，就可以先通过 PageRank 算法对该领域的用户进行影响力排名。在新浪微博中，每个用户都可以关注其他对象，被别人关注越多的用户人气往往越高，并且如果一个人气很高的用户 A 关注了用户 B，我们一般认为 B 的人气也很高。

下面简图表示了部分医疗领域人士的被关注度。图中用星星的数量表示对应用户的人气。可以发现，当用户被更多的其它用户关注时，人气往往会越高。例如右上的医生相对于下半部分的医生就更具人气。并且被更多医生关注的专家，人气也会更高。例如左上的专家和右上的专家，尽管右上专家被更多用户关注，但这些用户都是业外人士，人气较低，所以造成该专家的人气不及左上的专家。我们可以简单的用累加的方式来计算某个用户的影响力，例如用户 C 的粉丝为用户 A 和用户 B，那么用户 C 的影响力就可以认为是 A 和 B 的影响力之和，这种累加法便是简化版的 PageRank 算法。

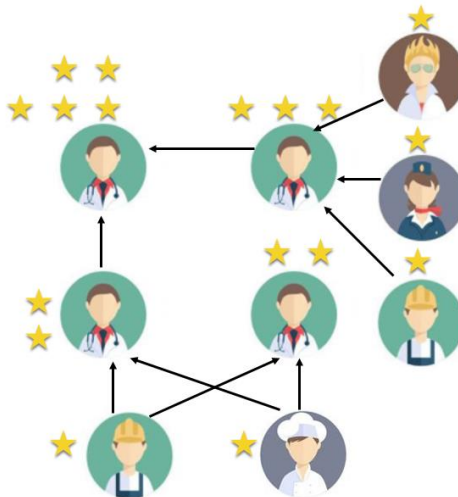


图 14.1 社交网络示例图

但简单累加并不能准确反映出个人影响力。例如图 14.2，用户 C1、C2 均为医疗领域人士，A 为一名业外人士，并且同时关注了 C1 和 C2；图 14.3 中，用户 C3 也为医疗领域人士，B 为一名业外人士。我们假设 A、B 两位在医疗领域内的影响力一样，按照简单累加法，C1（等于 A）的影响力和 C3（等于 B）的影响力就一样。但其实用户 B 在可以关注更多医生的前提下，只选择关注医生 C3，这就说明相对于 A 对 C1 的追随度，B 对 C3 的追随度要更高一些，而 A 和 B 的影响力又一样，所以可以认为 C3 比 C1 具有更高人气，这一点简单累加法就无法说明。



图 14.2

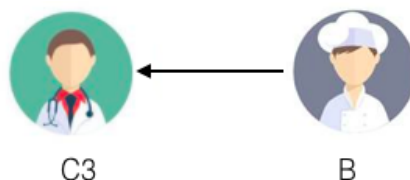


图 14.3

因此真实的 PageRank 算法绝不是简单累加，还需要考虑指向的其它网页（网页搜索中）或关注的其它用户（社交网络中），下面会做具体介绍。

PageRank 的计算模型可以基于矩阵，但矩阵的计算本身就是低效率的，因此只适合小规模数据。对于大规模的网页图或社交网络图，可以借助图模型计算。

14.2 PageRank 算法

14.2.1 算法介绍

PageRank 的一般计算公式为：

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

D(u)可以理解为可以访问到页面u的途径的方式，越多途径在一定程度上就说明了其rank应该比较高

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。

例如网页图结构 14.4，网页 A 同时被 B、C、D 指向。

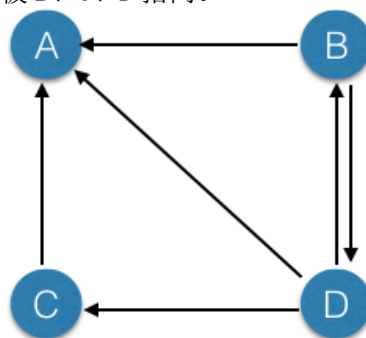


图 14.4 网页图结构

B 有 2 条出边，因此 B 将会传递 1/2 的 rank 值给 A，同理，C 只有一条出边，会传递全部的 rank 值给 A，D 传递 1/3 的 rank 值给 A。所以最终 A 的 rank 值可以表示为：

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

现实场景中，某个用户浏览网页时会有停止点击的时候，也就是说用户到达某个页面时会产生两种选择：继续浏览并点击网页中的超链接，另一种就是对当前页面失去兴趣，停止点击并随机浏览其它新的网页。对于这种情况，为了使 PageRank 更加精准，引入一个阻尼系数 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，修改后的 PageRank 公式为：

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

对于上面这个图，假设 d=0.85，A 的 rank 值可以进一步变为：

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

14.2.2 迭代式的 PageRank 计算

当所有顶点的 rank 值都被计算一次后，就完成了一次 PageRank 过程。为了结果的精准性，往往需要运行多次 PageRank 过程，当各个顶点的 rank 值变化微小并始终稳定在某个值附近时（此时达到收敛状态），才可以结束迭代。例如下面这个例子：

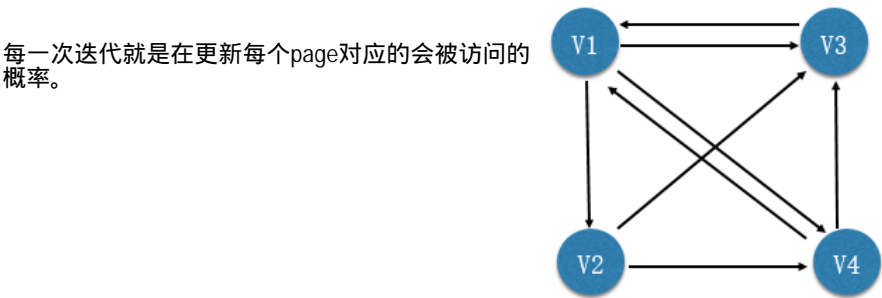


图 14.5 pagerank 迭代计算示例

图由 4 个顶点、8 条边构成。v1 到 v4 顶点的初始 rank 值如下：

表 14.1 v1-v4 初始 rank 值

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

根据 PageRank 计算公式，可以得出 v1 到 v4 的最新 rank 值为：

$$\begin{aligned} \text{PR}(v1) &= 0.25 * 1 + 0.25 * \frac{1}{2} = 0.37 \\ \text{PR}(v2) &= 0.25 * \frac{1}{3} = 0.08 \\ \text{PR}(v3) &= 0.25 * \frac{1}{3} + 0.25 * \frac{1}{2} + 0.25 * \frac{1}{2} = 0.33 \\ \text{PR}(v4) &= 0.25 * \frac{1}{3} + 0.25 * \frac{1}{2} = 0.20 \end{aligned}$$

同理可以得出运行四次 PageRank 过程后的结果：

表 14.2 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

运行五次后，可以发现迭代 5 和迭代 4 各顶点 rank 值的差距较小，可以猜测将要达到收敛状态。

表 14.3 运行 PageRank 5 次后

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

第六次迭代后，发现各顶点的 rank 值趋于稳定，因此可以判断算法达到收敛，迭代结束。

表 14.4 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	0.39

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

但在下面的实现中并没有根据收敛来判断程序是否终止，一方面是为了简化 PageRank 过程，另一方面是某些图很难收敛，为了避免死循环，下面代码均是预先设定阈值，在迭代次数达到阈值时，算法终止。

收敛

在高等数学中，收敛是研究函数的一个重要工具，是指随着自变量增加，函数值会聚于一点，向某一值靠近的状态。类比到 pagerank 中，就是指随着迭代次数增多，图中所有顶点的 rank 值趋于稳定的状态。

14.3 基于矩阵的实现

14.3.1 基本实现

图可以用邻接矩阵的方法表示，例如图 14.4 表示成如下矩阵：

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

第 1 列第 0 行为 1 则表示从 B 到 A 有一条边，0、1、2、3 列中的非 0 元素分别记录了 A、B、C、D 顶点的出边，而 0、1、2、3 行的非零元素则表示 A、B、C、D 的入边。各个顶点的当前 rank 值被存储在如下的初始向量中：

$PR(A)$
 $PR(B)$
 $PR(C)$
 $PR(D)$

这样的话，PageRank 计算就可以转化成矩阵向量的乘积。但在这之前，邻接矩阵还需要进一步的变形：对于任意一个非 0 的元素 a，假设它代表边 (u, v) 且 u 的出度为 $out_deg(u)$ ，则将 1 替换为 $1/out_deg(u)$ 。这是因为当该元素 a 与向量中的相应元素 ($PR(u)$) 相乘时，就代表将 u 顶点的当前 rank 值传递给 v，但 u 也会同时将值均等地传递给其它出边顶点，而不是全给 v，这里就用 $1/out_deg(u)$ 来表示这一点。上述例子中的邻接矩阵进一步转化为：

	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

最终，矩阵和向量的乘积结果就为一个 4*1 的向量，每一行分别代表一个顶点的最新 rank 值，例如结果的第 0 行（矩阵第 0 行和初始向量的相乘结果）为 A 的最新 rank 值，第 1 行（矩阵第 1 行和初始向量的相乘结果）为 B 的最新 rank 值。

那么如何用代码实现这种方法呢？因为所有操作都是基于矩阵进行，所以关键就在于矩阵的实现。

下面的代码实现了一个 Matrix 类，一共 m 行，每一行都有 n 列。Row 代表矩阵的某一行，这样实现的用途是为了两次重载 [] 操作符。如果直接将矩阵存储在二维数组中，并将该数组作为 Matrix 的包含数据，

在直接使用`[]`时，因为不检查边界范围，可能会产生数组越界的异常，是十分不安全的。这里用 Row 替代二维数组中的某一行，并为该类定义操作符`[]`操作，用来安全的获取特定行中相应位置的数据。同时，我们也希望 Matrix 类能像数组一样，直接通过`[]`操作符获取指定的行，因此再次对 Matrix 类重新定义`[]`操作符。这样，对于任意一个 Matrix 实例，例如 a，可以直接通过`a>[]`操作直接且安全的获取到指定行列的元素。

PageRank 计算过程中会涉及到矩阵的`*`和`+`，下面代码对这两个操作也进行了重载。这样，对于任意的两个 Matrix 实例，例如 a、b，如果要获得它们的乘积（或相加）结果，可以直接：`Matrix c = a*(+)` b，c 就保存了最终结果。下面代码重载了拷贝构造函数和赋值操作，将浅拷贝变为深拷贝。

```
1  #ifndef PAGERANK_INCLUDE_MATRIX_H
2  #define PAGERANK_INCLUDE_MATRIX_H
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <string.h>
7  namespace pagerank{
8  // Row represent a line in a matrix
9  template <typename T>
10 struct Row{
11     int n;
12     T *rdata;
13     Row() {
14         n = 0;
15         rdata = NULL;
16     }
17     void set_row(int n, T *rdata){
18         assert(n > 0);
19         this->n = n;
20         this->rdata = rdata;
21     }
22     T& operator[](int j){
23         assert(j >= 0 && j < n);
24         return rdata[j];
25     }
26     ~Row() {
27         rdata = NULL;
28     }
29 };
30 template <typename T>
31 class Matrix{
32 protected:
33     // m rows and n cols
34     int m;
35     int n;
36     Row<T> *data;
37 public:
38     Matrix() {
39         m = n = 0;
40         data = NULL;
41     }
42     Matrix(int m, int n, const T *_data = NULL) {
43         initialize(m, n, _data);
44     }
```

```

45 void initialize(int m, int n, const T *_data) {
46     // ensure parameter _m and _n are all valid
47     assert(m > 0 && n > 0);
48     this->m = m;
49     this->n = n;
50     T *buf;
51     assert((buf = (T*)calloc(m * n, sizeof(T))) != NULL);
52     if(_data) {
53         memcpy(buf, _data, sizeof(T) * m*n);
54     }
55     // allocate memory for each row in a matrix
56     assert((data = new Row<T>[m]) != NULL);
57     for(int i = 0; i < m; ++i) {
58         data[i].set_row(n, buf+i*n);
59     }
60 }
61 Matrix(const Matrix<T> &other) {
62     if(&other == this) return;
63     this->m = other.row_size();
64     this->n = other.col_size();
65     // empty, don't need to copy data
66     if(m <= 0 || n <= 0) return;
67     initialize(m, n, NULL);
68     for(int i = 0; i < m; ++i) {
69         for(int j = 0; j < n; ++j) {
70             data[i][j] = other[i][j];
71         }
72     }
73 }
74 ~Matrix() {
75     clear();
76 }
77 void clear() {
78     if(data != NULL && data[0].rdata != NULL) {
79         free(data[0].rdata);
80     }
81     if(data) {
82         delete[] data;
83     }
84     data = NULL;
85 }
86 Row<T>& operator[](int i) const {
87     assert(i >= 0 && i < m);
88     return data[i];
89 }
90 int row_size() const {
91     return m;
92 }
93 int col_size() const {
94     return n;
95 }
96 Matrix operator*(const Matrix<T> &other) {
97     // precondition

```

```

98     assert(n == other.row_size());
99     // res is result to be returned
100    int res_col = other.col_size();
101    // create res to store result
102    Matrix<T> res(m, res_col);
103    for(int i = 0; i < m; ++i) {
104        for(int j = 0; j < res_col; ++j) {
105            res[i][j] = 0;
106            // matrix A, B: sum of aik * bkj, k from 1 to n
107            for(int k = 0; k < n; ++k) {
108                res[i][j] += data[i][k] * other[k][j];
109            }
110        }
111    }
112    // overload operator=
113    return res;
114 }
115 // avoid shadow copy
116 void operator=(const Matrix<T> &other) {
117     if(&other == this) return;
118     // this is not empty
119     if(this->m != 0) {
120         clear();
121     }
122     this->m = other.row_size();
123     this->n = other.col_size();
124     // empty, don't need to copy data
125     if(m <= 0 || n <= 0) return;
126     initialize(m, n, NULL);
127     for(int i = 0; i < m; ++i) {
128         for(int j = 0; j < n; ++j) {
129             data[i][j] = other[i][j];
130         }
131     }
132 }
133 void operator+=(const Matrix<T> &other) {
134     // allow add itself
135     if(m != other.row_size() || n != other.col_size()) return;
136     assert(m > 0 && n > 0 && data != NULL);
137     for(int i = 0; i < m; ++i) {
138         for(int j = 0; j < n; ++j) {
139             data[i][j] += other[i][j];
140         }
141     }
142 }
143 void print() {
144     for(int i = 0; i < m; ++i) {
145         for(int j = 0; j < n; ++j) {
146             printf("%f ", data[i][j]);
147         }
148         printf("\n");
149     }
150 }

```

实现了从浅拷贝到深拷贝的转换，虽然性能下降，但是使用深拷贝就避免了拷贝只是拷贝了指针，但是指向的内容相同的情况的发生。因为对于这个算法情景，我们是需要动态的修改原始的PageRank矩阵的。

```
151     };
152     };
153 #endif
```

代码 14.1 Matrix 类

浅拷贝问题

在 C++ 中，包含指针数据的对象可能会产生浅拷贝问题。默认的拷贝构造函数或赋值函数只进行位拷贝而不是值拷贝。因此，当对象的指针数据被复制时，只是简单的进行指针数值的拷贝，而不是指针指向内存空间值的拷贝。在默认拷贝构造函数或赋值函数执行完成后，两个不同对象的指针数据将指向同一块内存空间，不仅会产生互相干扰的问题，并且在某个对象被析构掉时，另一对象将指向一块不存在的内存。

深拷贝

为了解决浅拷贝问题，可以重载拷贝构造函数和赋值函数，将指针数值的拷贝变为指向内存空间中值的拷贝，称为深拷贝。

基于实现好的 Matrix 类，PageRank 计算就很简单了，用两个 Matrix 实例分别表示邻接矩阵和向量，相乘便得到各个网页最新的 rank 值。例如对于上面的例子，在 PageRank.cpp 实现过程如下：

```
1  #include <utility>
2  #include <mutex>
3  #include <thread>
4  #include <vector>
5  #include "matrix.hpp"
6  using namespace pagerank;
7  // predefined threshold
8  static const int iters = 5;
9  static const float H_arr[] = {0, 0, 0.25, 0, 0, 1,
10                                1, 0, 0.25, 0, 0, 0.2,
11                                0, 0, 0, 0.5, 0, 1,
12                                0, 0, 0.25, 0, 1, 0.2,
13                                0, 1, 0.25, 0.5, 0, 0.5,
14                                1, 0, 1, 0, 1, 0};
15 int main(int argc, char * argv[]) {
16     int row1 = 6; int col1 = 6;
17     Matrix<float> H(row1, col1, H_arr);
18     // H.print();
19     float R_arr[] = {0.2,
20                      0.2,
21                      0.2,
22                      0.2,
23                      0.2,
24                      0.2};
25     int row2 = 6; int col2 = 1;
26     Matrix<float> R(row2, col2, R_arr);
27     // basic way
28     for(int i = 1; i <= iters; ++i) {
29         Matrix<float> res = H * R;
30         printf("iter %d res is :\n", i);
31         res.print();
32         R = res;
33     }
34 }
```

代码 14.2 基于 Matrix 类的 PageRank 实现

H_arr 为一个 const 型变量，存储了变形后邻接矩阵（除以出度后）的各个元素值。R_arr 存储了顶点的当前 rank 值。H、R 矩阵对象分别基于 H_arr、R_arr 数组构造自己的内部数据，最后的相乘结果存储在 res 矩阵中。预先设定的阈值为 5，当 PageRank 过程迭代 5 次后，循环结束。

14.3.2 并行方法

上面方法使用单个线程（主线程）完成所有顶点的 rank 值计算，虽然实现简单但程序不具备并发性。下面代码分配多个线程，每个线程只负责计算指定顶点的 rank 值，提高了程序的并发性：

```
1  #include <utility>
2  #include <mutex>
3  #include <thread>
4  #include <vector>
5  #include "matrix.hpp"
6  using namespace pagerank;
7  // predefined threshold
8  static const int iters = 5;
9  static const float H_arr[] = {0, 0, 0.25, 0, 0, 1,
10                                1, 0, 0.25, 0, 0, 0.2,
11                                0, 0, 0, 0.5, 0, 1,
12                                0, 0, 0.25, 0, 1, 0.2,
13                                0, 1, 0.25, 0.5, 0, 0.5,
14                                1, 0, 1, 0, 1, 0};
15  // get idx range of a row or a col
16  std::pair<int, int> get_range(int sub_id, int partitions, int total_len){
17      assert(partitions > 0 && total_len > 0);
18      int each_siz = total_len / partitions;
19      int start = sub_id * each_siz;
20      // > total_len is impossible
21      int end = (sub_id + 1) * each_siz;
22      // special case is the last partition
23      end = (sub_id == partitions - 1)?total_len:end;
24      return std::make_pair(start, end);
25  }
26  int main(int argc, char * argv[]){
27
28      int row1 = 6; int col1 = 6;
29      Matrix<float> H(row1, col1, H_arr);
30      // H.print();
31      float R_arr[] = {0.2,
32                       0.2,
33                       0.2,
34                       0.2,
35                       0.2,
36                       0.2};
37      int row2 = 6; int col2 = 1;
38      Matrix<float> R(row2, col2, R_arr);
39      // parallel way, should guarantee partitions <= row_size()
40      int partitions = 3;
41      if(partitions > row1){
42          fprintf(stderr, "partition is over rows range\n");
43      }
```

```

44     assert(partitions <= row1);
45     for(int i = 1; i <= iters; ++i){
46         Matrix<float> res(row1, col2);
47         // rows are divided into partitions
48         // thread num is partitions
49         std::vector<std::thread> threads;
50         threads.clear();
51         for(int i = 0; i < partitions; ++i){
52             threads.emplace_back([&](int th_i){
53                 std::pair<int, int> l_rows = get_range(th_i, partitions, row1);
54                 // subl is part of H_arr with fixed row range
55                 int rows = l_rows.second - l_rows.first;
56                 Matrix<float> subl(rows, col1, H_arr+col1*l_rows.first);
57                 // subl * R, sub_res is rows * col2
58                 Matrix<float> sub_res = subl * R;
59                 // copy data in sub_res into corresponding row range in res
60                 for(int i = l_rows.first; i < l_rows.second; ++i){
61                     for(int j = 0; j < col2; ++j){
62                         res[i][j] = sub_res[i - l_rows.first][j];
63                     }
64                 }
65             }, i);
66         }
67         for(int t = 0; t < partitions; ++t){
68             threads[t].join();
69         }
70         printf("iter %d res is :\n", i);
71         res.print();
72         R = res;
73     }
74     return 0;
75 }

```

代码 14.3 基于 Matrix 类的 PageRank 并行实现

partitions 指定了分配的线程数量，每个线程仅负责将邻接矩阵中指定行与矩阵 R 相乘，即只负责计算指定顶点的 rank 值。通过 get_range 函数，编号为 sub_id 的线程可以获得自己负责的行范围，存储在 std::pair 对象中。sub_res 临时存储了当前线程负责的顶点 rank 计算值，线程函数的最后一步就将 sub_res 中的结果存储至 res 矩阵中。因为各个线程负责的顶点范围不存在交集，所以这里当多个线程同时执行时，也不会产生任何问题。

14.4 基于图结构的实现

14.4.1 基本实现

图计算以图论为基础，将现实世界的问题抽象成图结构，并在这种数据结构上进行计算。通常，在图计算中，基本的数据结构表达就是： $G = (V, E, D)$ ， $V = \text{vertex}$ （顶点或者节点）， $E = \text{edge}$ （边）， $D = \text{data}$ （顶点或边上的数据）。图数据结构很好的表达了数据之间的关联性，很多实际问题中出现的问题都可以抽象成图来表示，从而以图论的思想或者以图为基础建立模型来解决问题，PageRank 就是图计算中一种很好的应用。

例如对于图 14.4，可以进一步转化为 $G = (V, E, D)$ 模型，这里的 D 就是指每个顶点的初始 rank 值，存储在顶点结构中。

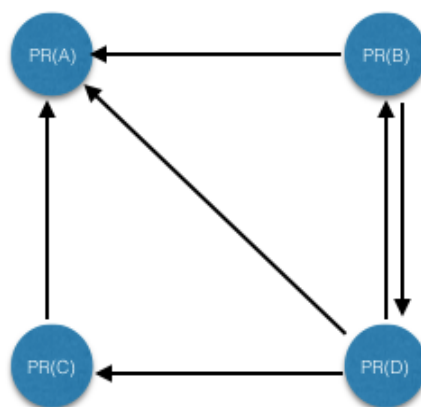


图 14.6 $G=(V, E, D)$ 模型

当要计算 A 的 rank 值时，得先知道 B、C、D 的 rank 值。A 依次获取各个邻居顶点的 rank 值之后，便可以按照之前介绍的 pagerank 算法计算自身值，其他顶点也同理。如下面的伪代码所示：

```
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)
```

这里，PR_F 定义了对于任意一条边(u, v)，v rank 值的计算规则。为了简化 PageRank 过程，这里简单的通过迭代次数来判定算法终止条件。

14.4.2 图结构表示方法

那么该如何在内存中表示如上图所示的 $G=(V, E, D)$ 模型呢？图的表示方法一般分为两种：邻接矩阵和邻接链表。邻接矩阵通过二维数组实现，这种方法实现简单，但当图比较稀疏时，矩阵中便会产生大量为 0 的元素，但其实真正记录边的是那些为 1 的值，会造成空间浪费；邻接链表通过链表数组实现，链表本身结构就大，也会带来存储开销。那么有没有一种方法既是通过数组实现，又能避免一般邻接矩阵的问题呢？CSC 和 CSR 就是很好的办法。

CSC 全称为 compressed sparse column，用紧凑压缩的方法来记录各个顶点的入边；CSR 全称为 compressed sparse row，同样也是利用压缩法来记录各个顶点的出边。

对于图 14.6，如果使用 CSC 表示的话：首先将所有的边按照目标顶点的顺序排列（从 A 到 D），得到下图：

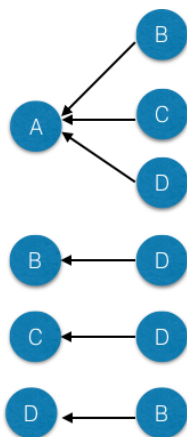


图 14.7 排序后的边

将所有边的源点按照上图的顺序存储至数组中：



图 14.8 源点数组

现在只要知道目标顶点 A、B、C、D 分别对应数组中的哪些源点，就可以确定出所有边。通过遍历所有边一次，可以得出每个顶点的入度：



图 14.9 入度数组

上图的数组记录了 A、B、C、D 的入度，通过这些入度也可以间接的得出每个顶点对应的源点范围。例如 A 的入度为 3，图 14.8 中 0、1、2 (0+3-1) 位置的元素便是 A 的入边源点，B 的入度为 1，3 (3+1-1) 位置的元素是 B 的入边源点，C、D 也同理，因此可以根据入度得出各个目标顶点的入边源点范围：



图 14.10 源点范围数组

在图 14.10 中，位置 0 为 0，位置 1 为 3，顶点 A 的入边源点范围就为 $[0, 3)$ ，即图 14.8 中 0 到 2 位置记录了 A 的所有入边源点；图 14.10 中位置 1、2 的元素为 3、4，顶点 B 的入边源点范围为 $[3, 4)$ ，即图 14.8 中 3 位置记录了 B 的入边源点；同理，图 14.10 中位置 2、3 的元素表示顶点 C 的入边源点范围，位置 3、4 的元素表示顶点 D 的入边源点范围。

CSR 与 CSC 类似，唯一不同的地方就是 CSR 记录了所有顶点的出边，而 CSC 记录了所有顶点的入边，这里就不详细介绍了。

14.4.3 优化

在上面的方法里，每次迭代都会对所有边执行 PR_F 操作，但对于某条边 (u, v) ，如果源点 u 达到收敛状态，它的 rank 值不会再发生变化，也就意味着它不会对 v 的 rank 值产生影响，因此，我们可以略过这样的边，仅对源点 rank 值发生改变的活跃边执行 PR_F 操作。这里引入一个活跃状态的概念：当某个顶点在当前迭代中的值发生变化时，我们就认为它在下次迭代中的状态为活跃。当一个顶点为活跃时，我们才对它进行相应的算法操作，否则直接略过。例如对于边 (x, y) ，只有当 x 的状态为活跃时，才执行 PR_F (x, y) 操作，并且我们称 (x, y) 为活跃边，这时因为 y 值发生了变化，所以将 y 在下次迭代中的状态设置为活跃，这样 y 就能后续地将自己的最新值传递给邻居节点。在引入“活跃”状态之后，上面的伪代码变化如下：

```

procedure PageRank( $G = (V, E, D)$ )
  while  $i < \text{iters}$ 
    for each  $v$  in  $V$ 
      for each ngh  $u$  that satisfies  $(u, v)$  in  $E$ 
        if  $u$  is active
          PR_F( $u, v$ )
          VertexMap( $v$ )
       $i++$ 

```

代码 14.5 基于图的 PageRank 伪代码 2

在上面伪代码中，尽管每次仅对源点 rank 值发生变化的边执行 PR_F 计算，但在这之前，仍然需要遍历 v 的每一条入边，以确定入边源点是否为活跃。那么有没有一种方法不需要 PR_F 之前的判断，就可以得出所有活跃边呢？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，此时的计算模型变化如下：

```

procedure PageRank( $G = (U, E, D)$ )
  while  $i < \text{iters}$ 
    // 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

```

代码 14.6 基于图的 PageRank 伪代码 3

可以发现，活跃顶点会将更新值主动的”推”给邻居，而不是由邻居来获取，我们称这种模式为”push”，邻居主动获取的模式（代码 14.5 所示）为”pull”。

当活跃顶点较少时（远远少于全部顶点数量，sparse），如果仅对活跃顶点进行 PR_F 操作，便可以大大减少操作次数，从而有效提高算法性能。因此，push 模型更适合活跃顶点较少的情况；当活跃顶点个数很多时（dense，例如，pagerank 算法在初始时认为所有顶点均为活跃），这时，push 模式或者 pull 模式都不能避免对所有边（u，v）进行操作。但是在多线程情况下，push 模式可能会同时对某个顶点进行更新（例如，v 顶点有两条入边（u1，v）和（u2，v），u1 和 u2 可能同时需要对 v 进行更新），并且活跃顶点越多，这样的竞争也就越多。但 pull 模式不存在这样的问题，因为每个顶点都是主动向源点获取数据，u1 和 u2 的数据可能会被同时读取，但这不会影响最终结果。因此，pull 模式更适合活跃顶点多的情况。

push VS pull

push 就是“推”，源节点主动地将自己的数据传递给所有目的节点；pull 即”拉”，目的节点主动地向所有源节点获取数据，以完成自身计算。两者最大的区别就在于数据传递方向不同。例如对于边 B→A，在 push 模式中，B 会将 PR(B) 主动传递给 A；在 pull 模式中，A 则主动向 B 获取 PR(B) 的值。

14.4.4 并行方法

为了提高算法的并发性，可以对整个图结构进行切割。单个线程（或线程组）只负责对单个子图上的顶点和边进行操作，通过多线程的并发执行，从而提高程序性能。

例如如下网络图结构 14.11，可以通过划分顶点的方式被分割成 2 个子图 14.11.1 和 14.11.2。首先将 6 个顶点划分成两个部分：1-3 号顶点和 4-6 号顶点，再将各个顶点的出边载入到各自所属的子图中。

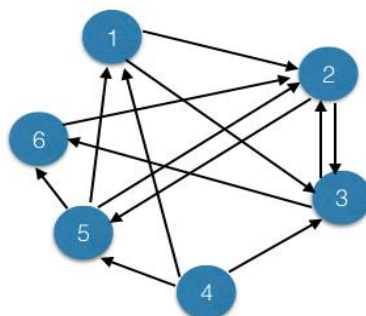


图 14.11 网络图结构

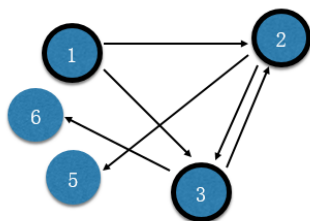


图 14.11.1 子图 1

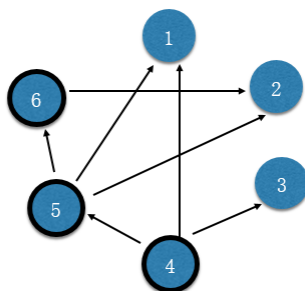


图 14.11.2 子图 2

在 push 模式下，线程 1 仅负责对 1-3 号顶点执行 PageRank 过程（上面伪代码所示），线程 2 仅负责对 4-6 号顶点执行 PageRank 过程。这里需要注意的是，可能存在多个源点同时更新某个目的顶点的情况（例如线程 1 在将顶点 1 的数据 push 给顶点 2 时，线程 2 同时将顶点 6 的数据 push 给了顶点 2），因此需要同步机制来确保数据的一致性。同步在确保数据一致性的同时，也会损耗性能，影响程序的并行效果。针对这个问题，我们对 push 模式下的 PageRank 算法进行进一步优化。

与之前划分顶点的方法不同的是，这里将各个顶点的入边载入到所属子图中，而不是出边。

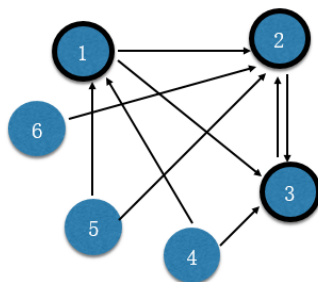


图 14.11.3 优化后的子图 1

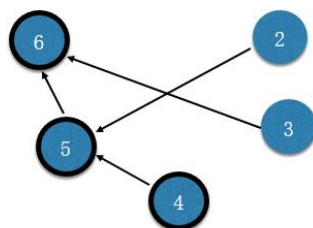


图 14.11.4 优化后的子图 2

线程 1 逐一访问子图 1 中的源点（按照从 1 到 6 的顺序），如果源点状态为活跃，就将该源点的当前数据 push 给子图 1 中相应的目的顶点，线程 2 执行相同操作。因为 2 个子图中不存在相同的目的顶点，所以线程 1 和线程 2 不可能同时对相同顶点进行更新操作，因此也就不需要额外的同步机制。

所有顶点的当前数据被连续的存储在内存数组中，线程 1 和线程 2 可能同时读取某个顶点的当前数据，但并不会修改该数据。如果每个线程都按照存储顺序依次访问各个顶点，便可以最大程度的减少 cache miss 次数，从而有效提高性能。

14.5 本章小结

PageRank 算法在现实生活中有很广泛的应用，例如网页排名和社交网络用户影响力分析。本章介绍了两种基本实现方法：基于矩阵实现和基于图实现。矩阵法主要利用了矩阵向量相乘的理论，更容易实现。基于图实现更适合较复杂的应用，例如万维网中庞大数量的网页和复杂的超链接关系。此外，本章节也对基本方法进行了拓展，给出了并行优化的方案。

14.6 练习题

本章14.4介绍了如何基于图结构来实现PageRank算法。14.4.4提供了两种划分策略来优化PageRank的并行计算，请你根据下面要求实现这部分。

第一种划分方法：首先将顶点平均划分成多个区域，再将各个区域中顶点的出边载入形成各自的子图（如图14.11.1、14.11.2所示）。请你实现该划分方法，并将划分好的子图以CSR（详见14.4.2）的形式存储。

另一种更适合push模型的划分方法：首先将顶点平均划分成多个区域，再将各个区域中顶点的入边载入形成各自的子图（如图14.11.3、14.11.4所示）。请你实现该划分方法，并将划分好的子图以CSC（详见14.4.2）的形式存储。

```
#define PART_MAX 10
class PartGraph{
public:
    std::string base_file;
    // assure partitions <= PART_MAX
    int partitions;
    Graph parted[PART_MAX];
    PartGraph();
    PartGraph(std::string base_file, int partitions);
    ~PartGraph();
    // 将base_file存储的图数据划分，结果存储在parted数组中
    // 第一种划分方法
    void part_graph_out();
    // 第二种划分方法（适合push）
    void part_graph_in();
};
// CSC/CSR representaion
class Graph{
public:
```



```

int vertices;
int edges;
/** CSC mode: all of edges are sorted by target first, source vertex of
    each edge is stored in this array following the sorted order.
    CSR mode: all of edges are sorted by source first, destination vertex
    of each edge is stored in this array following the sorted order.
    size is sizeof(Vertex) * edges */
Vertex *sorted_edges;
/** for each Vertex v, neighs[v] points to the start of its
    in-coming edges(CSC mode) or out-going edges(CSR mode) in
    sorted_edges. size is sizeof(Vertex*) * vertices */
Vertex **neighs;
// number of in-coming edges of each vertex
// size is sizeof(int) * vertices
int *in_degrees;
// number of out-going edges of each vertex
// size is sizeof(int) * vertices
int *out_degrees;
Graph();
~Graph();
// out-degree of Vertex v
int out_degree(Vertex v);
// in-degree of Vertex v
int in_degree(Vertex v);
// get i-th neighbor of Vertex v
Vertex get_neighbor(Vertex v, int i);
};

```

Project说明如下:

1. 按照PartGraph类的声明分别实现两种划分方法, 并将划分结果存储在Graph数组parted中。注意为了防止该数组溢出, 请确保划分总个数partitions<=PART_MAX。
2. 按照Graph类的声明实现CSR/CSC, 注意合理分配动态内存并妥善释放。
3. 分别用两种划分方法将原图划分为不同子图, 并在不同子图上运行PageRank算法(请使用push模型, 详见代码14.6), 输入数据请从Standard Network Analysis Project[<http://snap.stanford.edu/data/egonets-Facebook.html>]上下载ego-Facebook[<http://snap.stanford.edu/>]。
4. ego-Facebook以文本方式存储了社交网络的图结构, 请用该数据比较push模型在两种划分方法上的运行时间。

14.7 文献阅读

PageRank 最初来源于论文 [Page L, et al., 1999], 是最好的参考、答疑资料, 同时 [<https://en.wikipedia.org/wiki/PageRank>]对 PageRank 作了概括性的介绍。[Shun J, et al., 2013] 介绍了一个经典的图计算系统 Ligr。[Zhu X, et al., 2016]对 CSR 和 CSC 作了详细的介绍。14.4.4 中的优化方法借鉴了[Zhang K, et al., 2015]中的相关知识。

[<http://snap.stanford.edu/>] 和 [<http://snap.stanford.edu/data/egonets-Facebook.html>] 提供了课后项目相关数据的下载地址。