

# Lab 8: 图搜索 BFS 算法及存储优化

PB21020718 曾健斌

## 一、实验内容

针对图,根据给定的数据选择合适的存储方式(邻接矩阵和邻接表中的一种)进行存储,并进行图的广度优先遍历的过程。

**数据集 1**: 使用 `data.txt` 中的数据,看做无向图,选择合适的方式进行存储(提示:其特征为节点数较少而边比较密集),并以 A 为起始顶点输出遍历过程。

**数据集 2**: twitter 真实数据集,数据集规模如下:

twitter\_small: Nodes 81306, Edges 1768149, 有向图;

twitter\_large: Nodes 11316811, Edges 85331846, 有向图。

对 twitter\_small,选择一种合适的存储方式存储数据,并输出 BFS 的遍历时间。twitter\_large 不做要求。

提示:图可能不是连通图,且可能有重复边。

## 二、算法实现

### 相关结构体与枚举类

```
1 // bfs颜色
2 enum color_t { WHITE, GRAY, BLACK };
3
4 // 边结构体,用于构造器,本实现中有向边与无向边均使用该结构体
5 struct Edge {
6     int u, v;
7 };
8
9 // 无向图邻接矩阵实现中使用的结构体,后续将使用它实例化二重vector
10 struct Vertex {
11     color_t color;
12     int distance;
13     int parent;
14 };
15
16 // 链表,用于邻接链表实现中作为邻接链表
17 struct linklist {
18     int v;
19     linklist *next;
20     linklist *previous;
21 };
22
23 // 在邻接链表实现中作为顶点序列
24 struct VertexAdj {
25     color_t color;
26     int distance;
27     int parent;
28     linklist *adjacent;
29     int flag; // flag字段为考虑到可能有多个连通子图设立的字段
```

## 邻接链表实现

```

1  class GraphAdjList {
2  public:
3      // 用于顶点检索，使用无序的哈希map提高效率
4      unordered_map<int, VertexAdj> vertexs;
5      int V, E;
6      vector<Edge> edges;
7      // 构造器，本实现构造器比较繁琐
8      GraphAdjList(vector<Edge> edges) : V(0), E(edges.size()) {
9          for (auto e : edges) {
10             this->edges.push_back(e);
11         }
12         // 对每一条边进行以下考虑
13         for (auto e : edges) {
14             if (vertexs.find(e.u) == vertexs.end()) { // 若该顶点已存在
15                 vertexs[e.u].color = WHITE;
16                 vertexs[e.u].distance = -1;
17                 vertexs[e.u].parent = -1;
18                 vertexs[e.u].adjacent = new linklist;
19                 vertexs[e.u].adjacent->previous = vertexs[e.u].adjacent;
20                 vertexs[e.u].adjacent->next = new linklist;
21                 vertexs[e.u].adjacent->next->v = e.v;
22                 if (vertexs.find(e.v) == vertexs.end()) { // 判断边的尾结点是否
存在
23                     vertexs[e.v].color = WHITE;
24                     vertexs[e.v].distance = -1;
25                     vertexs[e.v].parent = -1;
26                     vertexs[e.v].adjacent = new linklist;
27                     vertexs[e.v].adjacent->previous = vertexs[e.v].adjacent;
28                     vertexs[e.v].adjacent->next = nullptr;
29                     V++;
30                     vertexs[e.v].flag = 0; // 初始化为0,后续用于存储bfs源点值
31                 }
32                 // 维护邻接链表
33                 vertexs[e.u].adjacent->next->previous =
vertexs[e.u].adjacent;
34                 vertexs[e.u].adjacent->next->next = nullptr;
35                 vertexs[e.u].flag = 0;
36                 V++;
37             }
38             else { // 顶点已存在则直接修改邻接链
39                 int fg = 0; // fg是考虑到重复边设置的标志位
40                 linklist *p = vertexs[e.u].adjacent;
41                 while (p->next != nullptr) {
42                     p = p->next;
43                     if (p->v == e.v) { // 判断为重复边，放弃该边
44                         fg = 1;
45                         break;
46                     }
47                 }
48                 if (!fg) {
49                     p->next = new linklist;

```

```

50         p->next->v = e.v;
51         if (vertexs.find(e.v) == vertexs.end()) { // 同样判断尾结
点是否存在
52             vertexs[e.v].color = WHITE;
53             vertexs[e.v].distance = -1;
54             vertexs[e.v].parent = -1;
55             vertexs[e.v].adjacent = new linklist;
56             vertexs[e.v].adjacent->previous =
vertexs[e.v].adjacent;
57             vertexs[e.v].adjacent->next = nullptr;
58             vertexs[e.v].flag = 0;
59             V++;
60         }
61         p->next->previous = p;
62         p->next->next = nullptr;
63     }
64 }
65 }
66 }
67 ~GraphAdjList(){}
68 void bfs(int s);
69 };

```

## bfs 实现（邻接链表）

```

1 void
2 GraphAdjList::bfs(int s){
3     queue<int> Q; // 用于bfs的队列
4     vertexs[s].color = GRAY;
5     vertexs[s].distance = 0;
6     vertexs[s].parent = -1;
7     Q.push(s);
8     while (!Q.empty()) { // 重复入队出队涂色等操作实现bfs
9         int u = Q.front();
10        Q.pop();
11        for (linklist *tmp = vertexs[u].adjacent; tmp->next != nullptr;) {
12            tmp = tmp->next;
13            if (vertexs[tmp->v].color == WHITE) {
14                vertexs[tmp->v].color = GRAY;
15                vertexs[tmp->v].distance = vertexs[u].distance + 1;
16                vertexs[tmp->v].parent = u;
17                vertexs[tmp->v].flag = s;
18                Q.push(tmp->v);
19            }
20        }
21        vertexs[u].color = BLACK;
22    }
23 }

```

## 邻接矩阵实现

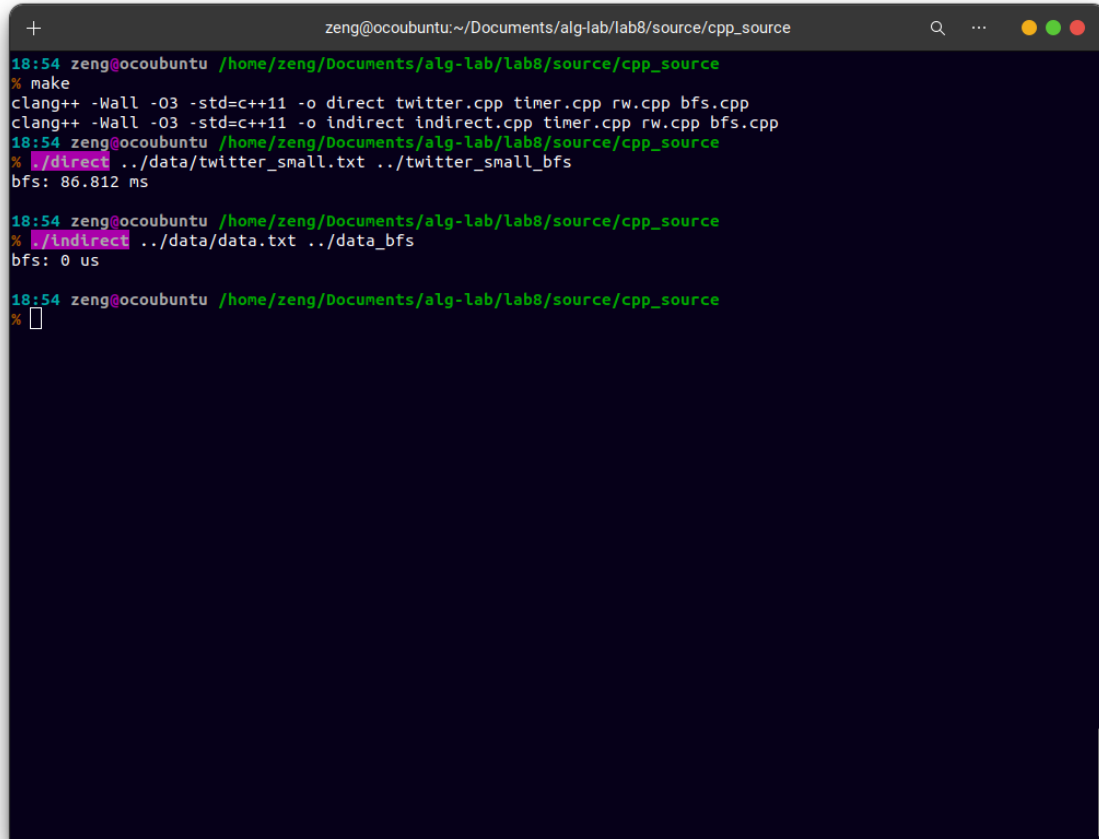
```
1 class GraphAdjMat {
2 public:
3     int V, E;
4     vector<Edge> edges;
5     vector<vector<int>> adj; // 维护一个二维邻接矩阵
6     vector<Vertex> vertexs;
7
8     // 邻接矩阵实现的构造器相对简单，需要注意的是动态维护矩阵大小
9     GraphAdjMat(vector<Edge> edges) : V(0), E(edges.size()) {
10         for (auto e : edges) {
11             this->edges.push_back(e);
12             V = max(V, max(e.u, e.v)); // 维护最大顶点序号
13         }
14         adj.resize(V); // 维护矩阵大小
15         for (auto e : edges) {
16             adj[e.u - 1].push_back(e.v - 1);
17             adj[e.v - 1].push_back(e.u - 1);
18         }
19         vertexs.resize(V);
20         for (auto i = 0; i < V; i++) {
21             vertexs[i].color = WHITE;
22             vertexs[i].distance = -1;
23             vertexs[i].parent = -1;
24         }
25     }
26     ~GraphAdjMat() {}
27     void bfs(int s);
28 };
```

## bfs 实现（邻接矩阵）

```
1 // 此bfs与邻接链表处的实现思路一致，不再多言
2 void
3 GraphAdjMat::bfs(int s){
4     queue<int> Q;
5     vertexs[s].color = GRAY;
6     vertexs[s].distance = 0;
7     vertexs[s].parent = -1;
8     Q.push(s);
9     while (!Q.empty()) {
10         int u = Q.front();
11         Q.pop();
12         for (auto v : adj[u]) {
13             if (vertexs[v].color == WHITE) {
14                 vertexs[v].color = GRAY;
15                 vertexs[v].distance = vertexs[u].distance + 1;
16                 vertexs[v].parent = u;
17                 Q.push(v);
18             }
19         }
20         vertexs[u].color = BLACK;
21     }
```

### 三、实验结果

两个数据集耗时：



```
zeng@ocoubuntu: ~/Documents/alg-lab/lab8/source/cpp_source
18:54 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab8/source/cpp_source
% make
clang++ -Wall -O3 -std=c++11 -o direct twitter.cpp timer.cpp rw.cpp bfs.cpp
clang++ -Wall -O3 -std=c++11 -o indirect indirect.cpp timer.cpp rw.cpp bfs.cpp
18:54 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab8/source/cpp_source
% ./direct ../data/twitter_small.txt ../twitter_small_bfs
bfs: 86.812 ms

18:54 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab8/source/cpp_source
% ./indirect ../data/data.txt ../data_bfs
bfs: 0 us

18:54 zeng@ocoubuntu /home/zeng/Documents/alg-lab/lab8/source/cpp_source
% 
```

输出 bfs 结果至文件

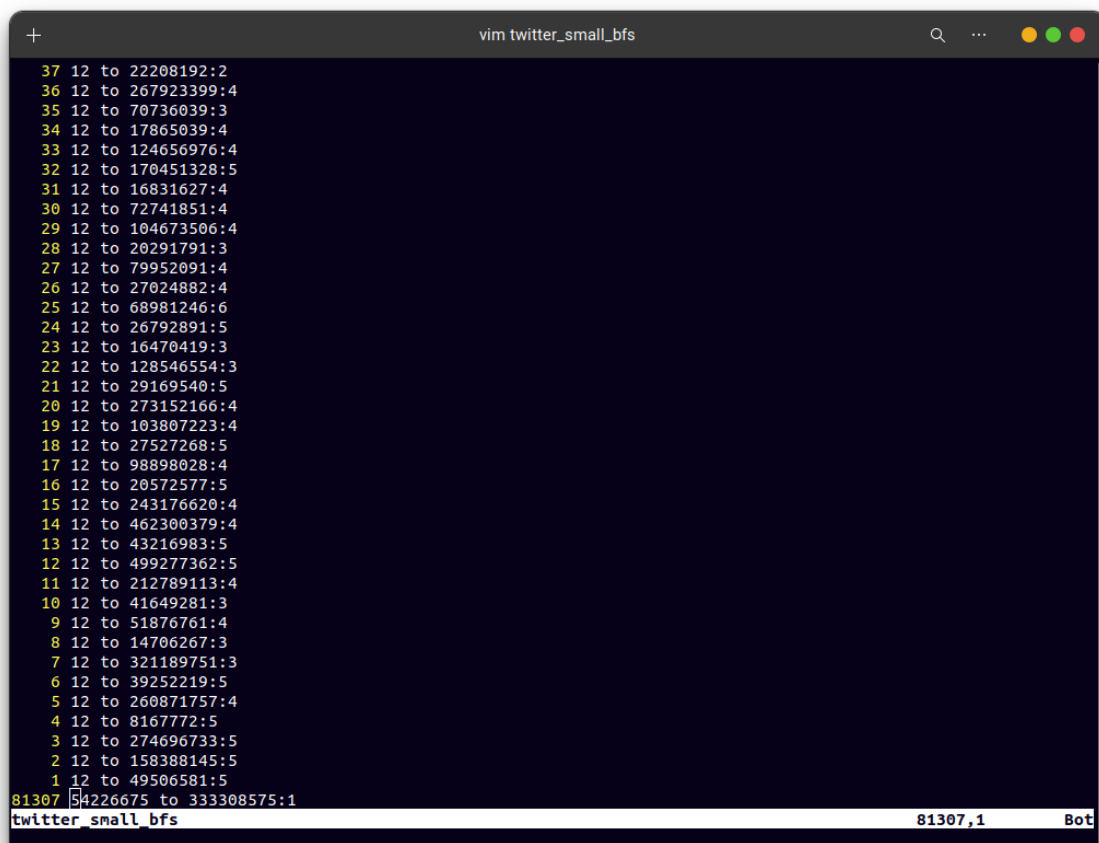
[illegible]

```

+
vim twitter_small_bfs
1 Vertex distance for file ../data/twitter_small.txt
1 vertex #: 81306
2 edge #: 2420766
3 12 to 99847264:4
4 12 to 102436604:4
5 12 to 14757212:4
6 12 to 556108152:4
7 12 to 46142925:4
8 12 to 110009988:4
9 12 to 367339979:4
10 12 to 27971297:6
11 12 to 44195043:5
12 12 to 163196038:4
13 12 to 80055873:5
14 12 to 29420247:5
15 12 to 206920297:6
16 12 to 30495820:6
17 12 to 378664141:6
18 12 to 18396706:6
19 12 to 25465315:5
20 12 to 78537310:4
21 12 to 267268763:4
22 12 to 35554964:5
23 12 to 132550281:4
24 12 to 103710274:4
25 12 to 19153872:4
26 12 to 116399875:4
27 12 to 125866482:4
28 12 to 57081366:4
29 12 to 252367584:4
30 12 to 22796259:4
31 12 to 18554046:4
32 12 to 15784870:4
33 12 to 28372127:4
34 12 to 270562594:4
35 12 to 94452438:4
36 12 to 274750412:4
37 12 to 22078387:4
twitter_small_bfs
"twitter_small_bfs" 81307L, 1405717B
1,1 Top

```

其中数据集 `twitter_small` 有两个连通分量



```
37 12 to 22208192:2
36 12 to 267923399:4
35 12 to 70736039:3
34 12 to 17865039:4
33 12 to 124656976:4
32 12 to 170451328:5
31 12 to 16831627:4
30 12 to 72741851:4
29 12 to 104673506:4
28 12 to 20291791:3
27 12 to 79952091:4
26 12 to 27024882:4
25 12 to 68981246:6
24 12 to 26792891:5
23 12 to 16470419:3
22 12 to 128546554:3
21 12 to 29169540:5
20 12 to 273152166:4
19 12 to 103807223:4
18 12 to 27527268:5
17 12 to 98898028:4
16 12 to 20572577:5
15 12 to 243176620:4
14 12 to 462300379:4
13 12 to 43216983:5
12 12 to 499277362:5
11 12 to 212789113:4
10 12 to 41649281:3
9 12 to 51876761:4
8 12 to 14706267:3
7 12 to 321189751:3
6 12 to 39252219:5
5 12 to 260871757:4
4 12 to 8167772:5
3 12 to 274696733:5
2 12 to 158388145:5
1 12 to 49506581:5
81307 5 4226675 to 333308575:1
twitter_small_bfs 81307,1 Bot
```

## 四、实验总结

本次实验中邻接矩阵的实现较为简单，而邻接链表的实现相对复杂

- 邻接矩阵使用二重 vector 实现
- 邻接链表使用 unordered\_map 与链表结构体实现，构造器较为复杂，但是在降低时间成本上颇有效果

关于存储方式选择的考虑

- 对于比较小的数据集使用邻接矩阵，因为该存储方式比较简单，而且由于数据点序号密集，我使用了与最大序号一样大小的矩阵
- 但是在对 twitter\_small 的实验中，以上实现会把内存跑爆
- 对于后者，由于其边比较稀疏，且顶点数大，顶点序号分散的情况，采用邻接链表实现显然是更好的选择