算法基础实验报告

> PB19051035周佳豪

实验设备和环境

C++11与vscode

实验内容及要求

斐波那契堆

- 本次实验需要编写斐波那契堆的基本操作，具体包括插入一个结点、抽取最小结点、合并根链表、关键词减值、删除一个结点。然后进行step1-7的操作，并进行事件分析。
- 表示堆结点以及堆的数据结构为：

```
typedef struct heap_node
{
    int key;
    int degree;
    list<struct heap_node *> children;
    struct heap_node *parent;
    struct heap_node *left;
    struct heap_node *right;
    bool mark;
} heap_node;

typedef struct heap
{
    struct heap_node *min;
    list<struct heap_node *> roots;
    int n;
} heap;
```

- 实验中使用`vector<heap_node *> N(1001)`记录每个堆结点的地址。
- 插入一个结点：

```
void heap_insert(heap *H, int x)
{
    heap_node *node = new heap_node();
    N[x] = node;
    node->degree = 0;
    node->parent = nullptr;
    node->children.clear();
    node->mark = false;
    node->key = x;
    if (H->min == nullptr)
    {
        H->min = node;
        H->roots.push_back(node);
        node->left = node->right = node;
    }
    else
    {
        H->roots.push_back(node);
        heap_node *temp = H->min->left;
        node->right = H->min;
        H->min->left = node;
        temp->right = node;
        node->left = temp;
        if (node->key < H->min->key)
            H->min = node;
    }
    H->n = H->n + 1;
}
```

此时，结点还未分配。

- 插入一个结点到根链表：

```
void addNodeInRoots(heap *H, heap_node *node)
{
    H->n = H->n + 1;
    node->parent = nullptr;
    if (H->min != nullptr)
    {
        heap_node *temp = H->min->left;
        node->right = H->min;
        H->min->left = node;
        temp->right = node;
        node->left = temp;
    }
    else
    {
        node->right = node->left = node;
    }

    H->roots.push_back(node);
}
```

此时，结点已经分配。

- 从根链表删除一个结点

```
heap_node *removeNodeFromRoots(heap *H, heap_node *node)
{
    auto iter = find(H->roots.begin(), H->roots.end(), node);
    H->roots.erase(iter);
    H->n = H->n - 1;
    heap_node *left_node = node->left;
    heap_node *right_node = node->right;
    left_node->right = right_node;
    right_node->left = left_node;
}
```

- 返回堆中最小关键词

```
int heap_minimum(heap *H)
{
    return H->min->key;
}
```

- 向不在根链表的结点中插入一个孩子

```
void addChildInParent(heap *H, heap_node *p, heap_node *child)
{
    if (p->children.size() == 0)
    {
        child->left = child->right = child;
    }
    else
    {
        auto right_node = p->children.front();
        auto left_node = p->children.back();
        left_node->right = child;
        child->left = left_node;
        right_node->left = child;
        child->right = right_node;
    }
    p->children.push_back(child);
    p->degree++;
    child->parent = p;
    H->n = H->n + 1;
}
```

- 合并的辅助操作heap_link，使y成为x的孩子

```
void heap_link(heap *H, heap_node *y, heap_node *x)
{
    removeNodeFromRoots(H, y);
    addChildInParent(H, x, y);
    y->mark = false;
}
```

- 合并堆的根链表

```
void consoliDate(heap *H)
{
    int max_degree = 0;
    max_degree = H->n;
    map<int, heap_node *> A;
    for (int i = 0; i <= max_degree; i++)
        A[i] = nullptr;
    vector<heap_node *> temp_roots;
    temp_roots.assign(H->roots.begin(), H->roots.end());
    for (heap_node *w : temp_roots)
    {
        heap_node *x = w;
        int d = x->degree;
        while (A[d] != nullptr)
        {
            heap_node *y = A[d];
            if (x->key > y->key)
            {
                heap_node *temp = y;
                y = x;
                x = temp;
            }
            heap_link(H, y, x);
            A[d] = nullptr;
            d = d + 1;
        }
        A[d] = x;
    }
    H->min = nullptr;
    H->roots.clear();
    for (int i = 0; i <= max_degree; i++)
    {
        if (A[i] != nullptr)
        {
            if (H->min == nullptr)
            {

                addNodeInRoots(H, A[i]);
                H->n = H->n - 1;
                H->min = A[i];
            }
            else
            {
                addNodeInRoots(H, A[i]);
                H->n = H->n - 1;
                if (A[i]->key < H->min->key)
                    H->min = A[i];
            }
        }
    }
}
```

- 删除一个结点的孩子

```
heap_node *removeChildFromNode(heap *H, heap_node *p, heap_node *child)
{
    auto iter = find(p->children.begin(), p->children.end(), child);
    if (iter == p->children.end())
    {
        return nullptr;
    }
    p->children.erase(iter);
    p->degree--;
    auto left_node = child->left;
    auto right_node = child->right;
    left_node->right = right_node;
    right_node->left = left_node;
    H->n = H->n - 1;
    return child;
}
```

- 抽取最小结点

```
heap_node *heap_extract_min(heap *H)
{
    heap_node *node = H->min;
    if (node != nullptr)
    {
        vector<heap_node *> temp;
        temp.assign(node->children.begin(), node->children.end());
        for (auto child : temp)
        {
            removeChildFromNode(H, node, child);
            addNodeInRoots(H, child);
        }
        removeNodeFromRoots(H, node);
        if (node->right == node)
            H->min = nullptr;
        else
        {
            H->min = node->right;

            consoliDate(H);
        }
    }
    return node;
}
```

- 堆的合并

```
heap *heap_union(heap *H1, heap *H2)
{
    heap *H = make_heap();
    H->min = H1->min;
    H->roots.assign(H1->roots.begin(), H1->roots.end());
    H->roots.insert(H->roots.end(), H2->roots.begin(), H2->roots.end());
    if (H1->min == nullptr || (H2->min != nullptr && H2->min->key < H1->min-
>key))
        H->min = H2->min;
    H->n = H1->n + H2->n;
    return H;
}
```

- 关键词减值

```
void heap_cut_node(heap *H, heap_node *x, heap_node *y)
{
    auto child = removeChildFromNode(H, y, x);
    addNodeInRoots(H, child);
    child->mark = false;
}

void cascading_heap_cut(heap *H, heap_node *y)
{
    auto z = y->parent;
    if (z != nullptr)
    {
        if (!y->mark)
        {
            y->mark = true;
        }
        else
        {
            heap_cut_node(H, y, z);
            cascading_heap_cut(H, z);
        }
    }
}

void heap_decrease_key(heap *H, heap_node *x, int k)
{
    if (k > x->key)
        return;
    x->key = k;
    auto y = x->parent;
    if (y != nullptr && x->key < y->key)
    {
        heap_cut_node(H, x, y);
    }

    if (x->key < H->min->key)
        H->min = x;
}
```

- 删除一个结点

```
void heap_delete_node(heap *H, heap_node *x)
{
    heap_decrease_key(H, x, -1);
    heap_extract_min(H);
}
```

- 从文件中读写数据

```
    string inpath = "../input/2_1_input.txt";
    string outpath_result = "../output/result.txt";
    string outpath_time = "../output/time.txt";
    ifstream infile;
    ofstream outfile_result, outfile_time;
    infile.open(inpath);
    outfile_result.open(outpath_result);
    outfile_time.open(outpath_time);
    vector<int> H1_insert;
    vector<int> H2_insert;
    vector<int> H3_insert;
    vector<int> H4_insert;
    int temp = 0;
    string buffer;
    while (!infile.eof())
    {
        getline(infile, buffer);
        temp++;
        if (temp <= 50)
        {
            H1_insert.push_back(stoi(buffer));
        }
        else if (temp >= 51 && temp <= 150)
        {
            H2_insert.push_back(stoi(buffer));
        }
        else if (temp >= 151 && temp <= 300)
        {
            H3_insert.push_back(stoi(buffer));
        }
        else if (temp >= 301 && temp <= 500)
        {
            H4_insert.push_back(stoi(buffer));
        }
    }
```

- step1

```
    LARGE_INTEGER t1, t2, tc;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);
    heap *H1 = make_heap();
    for (auto val : H1_insert)
        heap_insert(H1, val);
    heap *H2 = make_heap();
    for (auto val : H2_insert)
        heap_insert(H2, val);
    heap *H3 = make_heap();
    for (auto val : H3_insert)
        heap_insert(H3, val);
    heap *H4 = make_heap();
    for (auto val : H4_insert)
        heap_insert(H4, val);

    QueryPerformanceCounter(&t2);
    double time_step1 = double(t2.QuadPart - t1.QuadPart) /
(double)tc.QuadPart;
```

- step2

```
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);
    outfile_result << "H1" << endl;

    heap_insert(H1, 249);
    outfile_result << H1->n << ",";

    heap_insert(H1, 830);
    outfile_result << H1->n << ",";

    temp_int = heap_minimum(H1);
    outfile_result << temp_int << ",";

    heap_delete_node(H1, N[127]);
    outfile_result << H1->n << ",";

    heap_delete_node(H1, N[141]);
    outfile_result << H1->n << ",";

    temp_int = heap_minimum(H1);
    outfile_result << temp_int << ",";

    heap_decrease_key(H1, N[75], 61);
    outfile_result << H1->min->key << ",";

    heap_decrease_key(H1, N[198], 169);
    outfile_result << H1->min->key << ",";

    temp_node = heap_extract_min(H1);
    outfile_result << temp_node->key << ",";

    temp_node = heap_extract_min(H1);
    outfile_result << temp_node->key << endl;

    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time << "step2:" << endl;
    outfile_time << "time = " << time * 1000 << "ms" << endl;
```

- step3

```
    outfile_result << "H2" << endl;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);

    heap_insert(H2, 816);
    outfile_result << H2->n << ",";

    temp_int = heap_minimum(H2);
    outfile_result << temp_int << ",";

    heap_insert(H2, 345);
    outfile_result << H2->n << ",";

    temp_node = heap_extract_min(H2);
    outfile_result << temp_node->key << ",";

    heap_delete_node(H2, N[504]);
    outfile_result << H2->n << ",";

    heap_delete_node(H2, N[203]);
    outfile_result << H2->n << ",";

    heap_decrease_key(H2, N[296], 87);
    outfile_result << H2->min->key << ",";

    heap_decrease_key(H2, N[278], 258);
    outfile_result << H2->min->key << ",";

    temp_int = heap_minimum(H2);
    outfile_result << temp_int << ",";

    temp_node = heap_extract_min(H2);
    outfile_result << temp_node->key << endl;
    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time << "step3: " << endl;
    outfile_time << "time = " << time * 1000 << "ms" << endl;
```

- step4

```
    outfile_result << "H3:" << endl;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);

    temp_node = heap_extract_min(H3);
    outfile_result << temp_node->key << ",";

    temp_int = heap_minimum(H3);
    outfile_result << temp_int << ",";

    heap_insert(H3, 262);
    outfile_result << H3->n << ",";

    temp_node = heap_extract_min(H3);
    outfile_result << temp_node->key << ",";

    heap_insert(H3, 832);
    outfile_result << H3->n << ",";

    temp_int = heap_minimum(H3);
    outfile_result << temp_int << ",";

    heap_delete_node(H3, N[134]);
    outfile_result << H3->n << ",";

    heap_delete_node(H3, N[177]);
    outfile_result << H3->n << ",";

    heap_decrease_key(H3, N[617], 360);
    outfile_result << H3->min->key << ",";

    heap_decrease_key(H3, N[889], 353);
    outfile_result << H3->min->key << endl;

    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time << "step4:" << endl;
    outfile_time << "time = " << time * 1000 << "ms" << endl;
```

- step5

```
    outfile_result << "H4:" << endl;

    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);

    temp_int = heap_minimum(H4);
    outfile_result << temp_int << ",";
    heap_delete_node(H4, N[708]);
    outfile_result << H4->n << ",";

    heap_insert(H4, 281);
    outfile_result << H4->n << ",";

    heap_insert(H4, 347);
    outfile_result << H4->n << ",";

    temp_int = heap_minimum(H4);
    outfile_result << temp_int << ",";

    heap_delete_node(H4, N[415]);
    outfile_result << H4->n << ",";

    temp_node = heap_extract_min(H4);
    outfile_result << temp_node->key << ",";

    heap_decrease_key(H4, N[620], 354);
    outfile_result << H4->min->key << ",";

    heap_decrease_key(H4, N[410], 80);
    outfile_result << H4->min->key << ",";

    temp_node = heap_extract_min(H4);
    outfile_result << temp_node->key << endl;

    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time << "step5: " << endl;
    outfile_time << "time = " << time * 1000 << "ms" << endl;
```

- step6

```
    heap *H12 = heap_union(H1, H2);
    heap *H34 = heap_union(H3, H4);
    heap *H5 = heap_union(H12, H34);
```

- step7

```
    outfile_result << "H5:" << endl;

    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);

    temp_node = heap_extract_min(H5);
    outfile_result << temp_node->key << ",";

    temp_int = heap_minimum(H5);
    outfile_result << temp_int << ",";

    heap_delete_node(H5, N[800]);
    outfile_result << H5->n << ",";

    heap_insert(H5, 267);
    outfile_result << H5->n << ",";

    heap_insert(H5, 351);
    outfile_result << H5->n << ",";

    temp_node = heap_extract_min(H5);
    outfile_result << temp_node->key << ",";

    heap_decrease_key(H5, N[478], 444);
    outfile_result << H5->min->key << ",";

    heap_decrease_key(H5, N[559], 456);
    outfile_result << H5->min->key << ",";

    temp_int = heap_minimum(H5);
    outfile_result << temp_int << ",";

    heap_delete_node(H5, N[929]);
    outfile_result << H5->n << endl;

    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time << "step7: " << endl;
    outfile_time << "time = " << time * 1000 << "ms" << endl;
```

- 代码的善后处理

```
    infile.close();
    outfile_result.close();
    outfile_time.close();
    for (int i = 0; i < N.size(); i++)
        delete N[i];
    delete H1;
    delete H2;
    delete H12;
    delete H3;
    delete H4;
    delete H34;
    delete H5;
```

- 结果与分析
  - result.txt

    ```
    H1
    51,52,20,51,50,20,20,20,20,25
    H2
    101,8,102,8,100,99,10,10,10,10
    H3:
    2,3,150,3,150,6,149,148,6,6
    H4:
    1,199,200,201,1,200,1,5,5,5
    H5:
    6,9,490,491,492,9,11,11,11,490
    ```

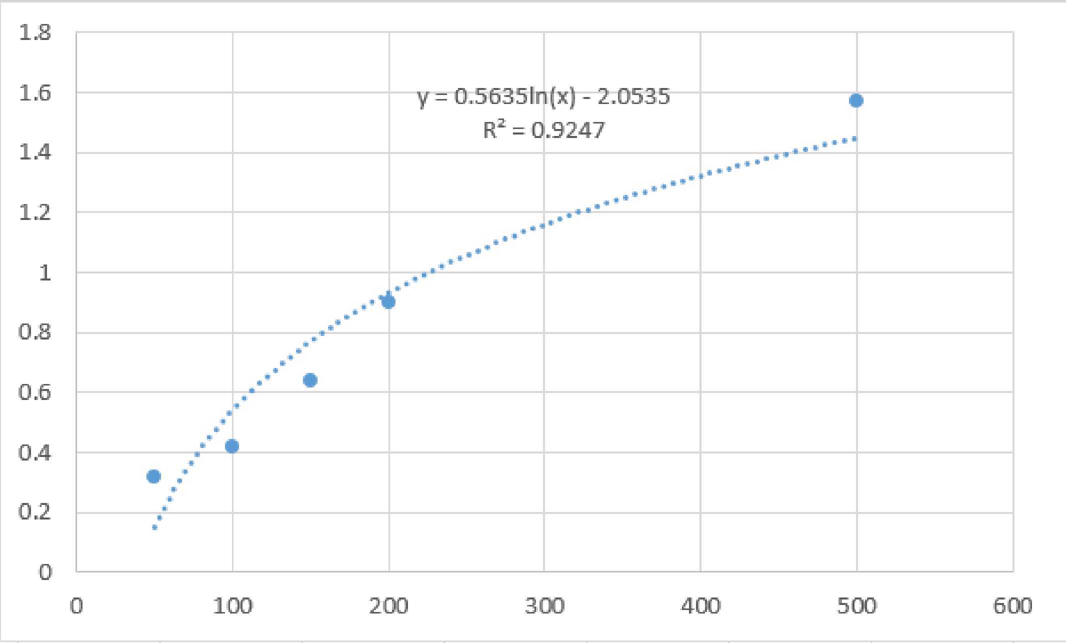
  - time.txt

其中time为5此time的平均值,单位ms

Heap.N	time1	time2	time3	time4	time5	time
50	0.3857	0.3949	0.2594	0.2811	0.2657	0.3174
100	0.5563	0.4261	0.3634	0.3702	0.3861	0.4204
150	0.8922	0.5922	0.5187	0.565	0.6233	0.6383
200	0.7642	0.9231	0.772	0.7474	1.2854	0.8984
500	1.4474	2.0826	1.4308	1.4295	1.447	1.5675

拟合结果与分析：

N	time（ms)
50	0.3174
100	0.4204
150	0.6383
200	0.8984
500	1.5675



而理论值即运行时间为O(logN)，其中N为堆中的结点个数。由于R^2=0.9247，故拟合结果较好，即实际运行时间也很好的服从了O(logN)

用于不相交集合的数据结构

- 本次实验需要建立一个不相交的数据结构，然后通过将有亲戚关系的人合并，最后得到总共的家族数。
- 实验所用的数据结构为：

```
typedef struct Node
{
    int rank;
    struct Node *parent;
} Node;

typedef struct Forest
{
    list<Node *> roots;
    int count;
} Forest;
```

通过Forest的count即可得到家族数。

- MAKE-SET操作

```
Node *make_set(Forest *F)
{
    Node *node = new Node();
    node->rank = 0;
    node->parent = node;
    F->roots.push_back(node);
    F->count++;
    return node;
}
```

- UNION操作

```
Node *union_tree(Forest *F, Node *x, Node *y)
{
    Node* x_root = find_set(x);
    Node* y_root = find_set(y);
    if(x_root != y_root)
        link(F, x_root, y_root);
}
```

此时需要先判断x与y是否已经在一棵树上了，然后再进行合并。

- LINK操作

```
Node *link(Forest *F, Node *x, Node *y)
{
    if (x->rank > y->rank)
    {
        y->parent = x;
        auto iter = find(F->roots.begin(), F->roots.end(), y);
        F->roots.erase(iter);
        F->count--;
    }
    else
    {
        x->parent = y;
        auto iter = find(F->roots.begin(), F->roots.end(), x);
        F->roots.erase(iter);
        F->count--;
        if (x->rank == y->rank)
```

```
            y->rank++;
    }
}
```

- FIND-SET操作

```
Node *find_set(Node *x)
{
    if (x != x->parent)
    {
        x->parent = find_set(x->parent);
    }
    return x->parent;
}
```

- 从文件中读取数据，并将数据存储在二维数组中

```
    string inpath = "../input/2_2_input.txt";
    string outpath_result = "../output/result.txt";
    string outpath_time = "../output/time.txt";
    int n_10[10][10], n_15[15][15], n_20[20][20], n_30[30][30], n_25[25]
[25];
    ifstream infile;
    ofstream outfile_result, outfile_time;
    infile.open(inpath);
    outfile_result.open(outpath_result);
    outfile_time.open(outpath_time);
    string buffer;
    vector<int> T;
    while (getline(infile, buffer))
    {
        string temp;
        istringstream data(buffer);
        while (data >> temp)
        {
            T.push_back(stoi(temp));
        }
    }
    //cout<<"T.size() = "<<T.size()<<endl;
    int count = 0;
    while (count < T.size())
    {
        if (count <= 99)
        {
            for (int i = 0; i < 10; i++)
                for (int j = 0; j < 10; j++)
                    n_10[i][j] = T[count++];
        }
        else if (count >= 100 && count <= 324)
        {
            for (int i = 0; i < 15; i++)
                for (int j = 0; j < 15; j++)
                    n_15[i][j] = T[count++];
        }
        else if (count >= 325 && count <= 724)
        {
```

```
        for (int i = 0; i < 20; i++)
            for (int j = 0; j < 20; j++)
                n_20[i][j] = T[count++];
    }
    else if (count >= 725 && count <= 1349)
    {
        for (int i = 0; i < 25; i++)
            for (int j = 0; j < 25; j++)
                n_25[i][j] = T[count++];
    }
    else if (count >= 1350 && count <= 2249)
    {
        for (int i = 0; i < 30; i++)
            for (int j = 0; j < 30; j++)
                n_30[i][j] = T[count++];
    }
}
```

- 对五组数据分别计算家族数以及统计时间

```
double time;
LARGE_INTEGER t1, t2, tc;

//n=10
Forest *F_10 = new Forest();
F_10->count = 0;
F_10->roots.clear();
vector<Node *>person_10;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
for(int i = 0; i < 10; i++)
{
    person_10.push_back(make_set(F_10));

}
for (int i = 0; i < 10; i++)
{
    for (int j = i + 1; j < 10; j++)
    {
        if(n_10[i][j]==1)
        {
            union_tree(F_10,person_10[i],person_10[j]);
        }

    }
}
QueryPerformanceCounter(&t2);
time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
outfile_time<<"n=10 time = "<<time * 1000<<"ms"<<endl;
cout<<"F_10->count="<<F_10->count<<endl;
outfile_result<<"n=10 family numer is "<<F_10->count<<endl;


//n=15

Forest *F_15 = new Forest();
F_15->count = 0;
```

```
F_15->roots.clear();
vector<Node *>person_15;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
for(int i = 0; i < 15; i++)
{
    person_15.push_back(make_set(F_15));

}
for (int i = 0; i < 15; i++)
{
    for (int j = i + 1; j < 15; j++)
    {
        if(n_15[i][j]==1)
        {
            union_tree(F_15,person_15[i],person_15[j]);
        }

    }
}
QueryPerformanceCounter(&t2);
time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
outfile_time<<"n=15 time = "<<time * 1000<<"ms"<<endl;
cout<<"F_15->count="<<F_15->count<<endl;
outfile_result<<"n=15 family numer is "<<F_15->count<<endl;


//n=20

Forest *F_20 = new Forest();
F_20->count = 0;
F_20->roots.clear();
vector<Node *>person_20;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
for(int i = 0; i < 20; i++)
{
    person_20.push_back(make_set(F_20));

}
for (int i = 0; i < 20; i++)
{
    for (int j = i + 1; j < 20; j++)
    {
        if(n_20[i][j]==1)
        {
            union_tree(F_20,person_20[i],person_20[j]);
        }

    }
}
QueryPerformanceCounter(&t2);
time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
outfile_time<<"n=20 time = "<<time * 1000<<"ms"<<endl;
cout<<"F_20->count="<<F_20->count<<endl;
outfile_result<<"n=20 family numer is "<<F_20->count<<endl;

//n=25
```

```
    Forest *F_25 = new Forest();
    F_25->count = 0;
    F_25->roots.clear();
    vector<Node *>person_25;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);
    for(int i = 0; i < 25; i++)
    {
        person_25.push_back(make_set(F_25));

    }
    for (int i = 0; i < 25; i++)
    {
        for (int j = i + 1; j < 25; j++)
        {
            if(n_25[i][j]==1)
            {
                union_tree(F_25,person_25[i],person_25[j]);
            }

        }
    }
    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time<<"n=25 time = "<<time * 1000<<"ms"<<endl;
    cout<<"F_25->count="<<F_25->count<<endl;
    outfile_result<<"n=25 family numer is "<<F_25->count<<endl;

    //30
    Forest *F_30 = new Forest();
    F_30->count = 0;
    F_30->roots.clear();
    vector<Node *>person_30;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);
    for(int i = 0; i < 30; i++)
    {
        person_30.push_back(make_set(F_30));

    }
    for (int i = 0; i < 30; i++)
    {
        for (int j = i + 1; j < 30; j++)
        {
            if(n_30[i][j]==1)
            {
                union_tree(F_30,person_30[i],person_30[j]);
            }

        }
    }
    QueryPerformanceCounter(&t2);
    time = double(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    outfile_time<<"n=30 time = "<<time * 1000<<"ms"<<endl;
    cout<<"F_30->count="<<F_30->count<<endl;
    outfile_result<<"n=30 family numer is "<<F_30->count<<endl;
```

- 最后关闭文件
- 实验结果与分析
    - result.txt

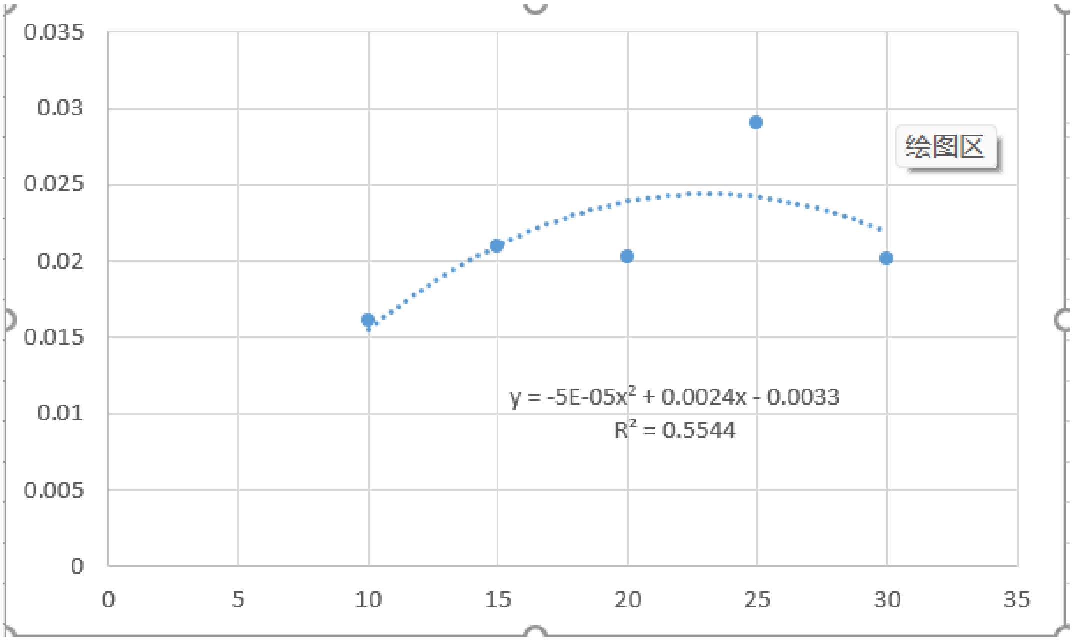
      ```
      n=10 family numer is 3
      n=15 family numer is 3
      n=20 family numer is 2
      n=25 family numer is 1
      n=30 family numer is 5
      ```

    - time.txt

      time单位为ms

n	time1	time2	time3	time4	time5	time
10	0.0229	0.0144	0.0144	0.014	0.0146	0.0161
15	0.0315	0.0202	0.0144	0.0152	0.0236	0.021
20	0.0234	0.0155	0.0273	0.0229	0.0125	0.0203
25	0.0429	0.0283	0.0203	0.0209	0.0328	0.029
30	0.0224	0.0164	0.021	0.024	0.0168	0.0201

拟合结果与分析：



可见二次多项式拟合的结果很差，

```
for (int i = 0; i < 25; i++)
    {
        for (int j = i + 1; j < 25; j++)
        {
            if(n_25[i][j]==1)
            {
                union_tree(F_25,person_25[i],person_25[j]);
            }

        }
    }
```

这是由于只有两者有亲戚关系的时候才需要合并，此外union也有条件（不在同一集合才能合并），故只能得到运行时间的上界为O(n^2)，不能得到确切的运行时间