EXP2 算法基础实验报告

廖佳怡 PB19151776

1. 实验设备和环境

编译环境：Windows10

编程语言：C++

机器内存：16.0GB

时钟主频：2.30GHz

2. 实验内容及要求

- 实验2.1 斐波拉契堆
  - 实验内容：

    实现斐波拉契堆的基本操作：INSERT, MINIMUM, UNION, EXTRACT-MIN, DECREASE-KEY, DELETE

    - step1：通过INSERT（要操作的数见input.txt）建立斐波那契堆H1-H4。
    - step2:在H1下完成以下10个ops:
      INSERT(H1,249);INSERT(H1,830);MINIMUM(H1);DELETE(H1,127);DELETE(H1,141);
      MINIMUM(H1);
      DECREASE-KEY(H1,75,61);DECREASE-KEY(H1,198,169); EXTRACT-MIN(H1);EXTRACT-MIN(H1);
    - Step3:在H2下完成以下10个ops:
      INSERT(H2,816); MINIMUM(H2); INSERT(H2,345); EXTRACT-MIN(H2);DELETE(H2,504);DELETE(H2,203);
      DECREASE-KEY(H2,296,87);DECREASE-KEY(H2,278,258);MINIMUM(H2);EXTRACT-MIN(H2);
    - Step4:在H3下完成以下10个ops：
      EXTRACT-MIN(H3); MINIMUM(H3);INSERT(H3,262); EXTRACT-MIN(H3);INSERT(H3,830);MINIMUM(H3);
      DELETE(H3,134);DELETE(H3,177);DECREASE-KEY(H3,617,360);DECREASE-KEY(H3,889,353);
    - Step5:在H4下完成以下10个ops：
      MINIMUM(H4); DELETE(H4,708);
      INSERT(H4,281);INSERT(H4,347);MINIMUM(H4);DELETE(H4,415);
      EXTRACT-MIN(H4); DECREASE-KEY(H4,620,354);DECREASE-KEY(H4,410,80);
      EXTRACT-MIN(H4);
    - Step6:将H1-H4进行UNION成H5
    - Step7:在H5完成如下10个ops：
      EXTRACT-MIN(H5); MINIMUM(H5); DELETE(H5,800);
      INSERT(H5,267);INSERT(H5,351); EXTRACT-MIN(H5);
      DECREASE-KEY(H5,478,444);DECREASE-KEY(H5,559,456);MINIMUM(H5);
      DELETE(H5,929);
    - 分别统计step2-5,7的运行时间，并画图分析。

- 实验要求：

  Ex2.1/input/
  2_1_input.txt：共500行，每行是一个1-1000之间的整数，前50个是H1进行INSERT操作的数值，第51-150个是H2进行INSERT操作的数值，第151-300个是H3进行INSERT操作的数值，第301-500个代表H4进行INSERT操作的数值。
  Ex2.1/output/
  result.txt：step2-5,7的10个ops结果（中间用逗号分割）
  注：INSERT操作,DELETE操作返回H.N;
  EXTRACT-MIN操作,DECREASE-KEY操作返回H.min;
  格式：
  H1
  10个ops结果
  格式：H2
  ...
  H5
  10个ops结果
  time.txt：格式：step2的运行时间
  step3的运行时间
  step4的运行时间
  step5的运行时间
  step7的运行时间

- 实验2.2 朋友圈
  - 实验内容：

    有 N 个人（N=10/15/20/25/30）。其中有些人是亲戚，有些则不是。他们的亲戚关系具有传递性。如果已知 A 是 B 的亲戚，B 是 C 的亲戚，那么我们可以认为 A 也是 C 的亲戚。所谓的家族，是指所有亲戚的集合。

    - 给定一个 N * N 的矩阵 M，表示不同人之间的亲戚关系。如果M[i][j] = 1，表示已知第 i 个和 j 个人互为亲戚关系，否则不是亲戚关系。你必须输出所有人中的家族数。
    - 必须用并查集解决，并要实现按秩合并，路径压缩两种优化手段。
    - 记录不同N值的运行时间，并画图分析。
  - 实验要求：

    2_2_input.txt（已给出）：
    10*10*矩阵（当*N=10*时矩阵*M*）（不同矩阵之间以换行符间隔）
    *15*15矩阵
    20*20*矩阵
    *25*25矩阵
    30*30矩阵
    Ex2.2/output/
    result.txt：统计n取不同值时的家族数
    格式：n=10 家族数
    n=15 家族数
    ...
    time.txt：求解n取不同值时家族数的时间（格式类似上面result.txt）

3. 方法和步骤

Exp1

数据结构：

结点Node和斐波拉契堆Heap

```
class Node
{
    public:
        int key;
        int degree;
        Node* p;
        std::unordered_set<Node* > child;
        bool mark;
};
class Heap
{
    public:
        int n;
        Node* min;
        list<Node*> root_list;
};
```

斐波拉契堆上操作Util

```
class Util
{
    public:
        Heap* Fib_Init_Heap();
        int Fib_Heap_Insert(Heap* ,int );
        Heap* Fib_Heap_Union(Heap* ,Heap* );
        int Fib_Heap_Decrease_Key(Heap*,Node*,int);
        int Fib_Heap_Min(Heap*);
        int Fib_Heap_Extract_Min(Heap*);
        int Fib_Heap_Delete(Heap*,Node*);
    private:
        void Consolidate(Heap* );
        void Cut(Heap*,Node*,Node*);
        void Cascading_Cut(Heap*,Node*);
        Node* Fib_Heap_Link(Heap*,Node* , Node*);
        Node* Fib_Make_Node(int);
};
```

操作实现：

- 创建一个新的斐波那契堆

```
Heap* Util::Fib_Init_Heap()
{
    Heap* heap=new Heap;
    heap->n=0;
    heap->min=NULL;
    return heap;
}
```

- 插入一个结点

创建一个结点并将它插入根链表。

```
int Util::Fib_Heap_Insert(Heap* H,int val)
{
    Node* x=Util::Fib_Make_Node(val);
    node_map.insert({val,x});
    x->degree=0;
    x->p=NULL;
    x->mark=false;
    if(H->min==NULL)
    {
        H->root_list.push_back(x);
        H->min=x;
    }
    else
    {
        H->root_list.push_back(x);
        if(x->key < H->min->key)
            H->min=x;
    }
    H->n=H->n+1;
    return H->n;
}
```

- 寻找最小结点

```
int Util::Fib_Heap_Min(Heap* H)
{
    return H->min->key;
}
```

- 两个斐波拉契堆的合并

  两个根链表的合并

```
Heap* Util::Fib_Heap_Union(Heap* H1,Heap* H2)
{
    Heap* H = new Heap;
    H->min=H1->min;
    H->root_list.splice(H->root_list.end(),H1->root_list);
    H->root_list.splice(H->root_list.end(),H2->root_list);
    if(H1->min==NULL || H2->min!=NULL && H2->min->key < H1->min->key)
    {
        H->min=H2->min;
    }
    H->n=H1->n+H2->n;
    return H;
}
```

- 抽取最小结点

  先把z的所有孩子结点移到根链表，再合并根链表中度相同的结点

```
int Util::Fib_Heap_Extract_Min(Heap* H)
{
    Node* z=H->min;
    if(z)
    {
        for(auto x : z->child)
        {
            H->root_list.push_back(x);
            x->p=NULL;
        }
        H->root_list.remove(z);
        node_map.erase(z->key);
        if(H->root_list.size()==0)
        {
            H->min=NULL;
        }
        else
        {
            H->min=NULL;
            Util::Consolidate(H);
        }
        H->n=H->n-1;
    }
    return z->key;
}
```

辅助函数Link把y从根链表中移出，挂到x下。前提是x与y度数相同且保持小根堆性质。

```
Node* Util::Fib_Heap_Link(Heap* H,Node* y,Node* x)
{
    //H->root_list.remove(y);
    x->child.insert(y);
    x->degree++;
    y->mark=false;
    y->p=x;
    return y;
}
```

辅助函数Consolidate合并度数相同的点并重建根链表，找到最小值。

```cpp
void Util::Consolidate(Heap* H)
{
    std::array<Node* , max_D> A;
    for(int i=0;i<max_D;i++)
    {
        A[i]=NULL;
    }
    std::unordered_set<Node* >remove_set;
    for(auto w : H->root_list)
    {
        Node* x=w;
        if(remove_set.find(x)!=remove_set.end())
            continue;//x已被移除
        int d=x->degree;
        while(A[d])
        {
            Node* y=A[d];
            if(x->key > y->key)
            {
                remove_set.insert(Fib_Heap_Link(H,x,y));
                x=y;
            }
            else
                remove_set.insert(Fib_Heap_Link(H,y,x));
            A[d]=NULL;
            d++;
        }
        A[d]=x;
    }
    H->min=NULL;
    H->root_list.clear();
    for(int i=0;i<max_D;i++)
    {
        if(A[i])
        {
            if(H->min==NULL)
            {
                H->root_list.push_back(A[i]);
                H->min=A[i];
            }
            else
            {
                H->root_list.push_back(A[i]);
                if(A[i]->key<H->min->key)
                {
                    H->min=A[i];
                }
            }
        }
    }
}
```

- 关键字减值

```
int Util::Fib_Heap_Decrease_Key(Heap* H,Node* x,int k)
{
    if(k > x->key)
    {
        cout<<"new key is greater than current key"<<endl;
    }
    node_map.erase(x->key);
    node_map.insert({k,x});
    x->key=k;
    Node* y=x->p;
    if(y && x->key < y->key)
    {
        Util::Cut(H,x,y);
        Util::Cascading_Cut(H,y);
    }
    if(x->key < H->min->key)
    {
        H->min=x;
    }
    return H->min->key;
}
```

辅助函数Cut和Cascading_Cut用来剪切结点挂到根链表上并修改mark域。

```
void Util::Cut(Heap* H,Node* x,Node* y)
{
    y->child.erase(x);
    y->degree--;
    H->root_list.push_back(x);
    x->p=NULL;
    x->mark=false;
}
void Util::Cascading_Cut(Heap* H,Node* y)
{
    Node* z=y->p;
    if(z)
    {
        if(y->mark==false)
        {
            y->mark=true;
        }
        else
        {
            Util::Cut(H,y,z);
            Util::Cascading_Cut(H,z);
        }
    }
}
```

- 删除一个结点

  先将该结点的关键字降到最小，再抽取最小值。

```
int Util::Fib_Heap_Delete(Heap* H,Node* x)
{
    Util::Fib_Heap_Decrease_Key(H,x,0);
    Util::Fib_Heap_Extract_Min(H);
    return H->n;
}
```

Exp2

按秩合并

使具有较少结点的树的根指向具有较多结点的树的根。对于每个结点，维护一个秩，它表示该结点高度的一个上界，在UNION操作中，可以让具有较小秩的根指向具有较大秩的根

路径压缩

在FIND-SET中使查找路径中的每个结点直接指向根，不改变任何结点的秩。

- MAKE-SET

```
void MAKE_SET(int x)
{
    p[x]=x;
    r[x]=0;
}
```

- UNION

```
void UNION(int x,int y)
{
    LINK(FIND_SET(x),FIND_SET(y));
}
```

```
void LINK(int x,int y)
{
    if(r[x]>r[y])
        p[y]=x;
    else
    {
        p[x]=y;
        if(r[x]==r[y])
            r[y]++;
    }
}
```

- FIND-SET

```
int FIND_SET(int x)
{
    if(x!=p[x])
    {
        p[x]=FIND_SET(p[x]);
        return p[x];
    }
}
```

4. 结果和分析

Exp1

- 结果如下：

```
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
```

- 时间分析
  - 理论时间复杂度：

    创建一个新的斐波拉契堆，插入一个结点，寻找最小结点，两个斐波拉契堆的合并，关键字减值的摊还代价都为$O(1)$。

    抽取最小结点，删除一个结点的摊还代价都为$O(lgn)$。

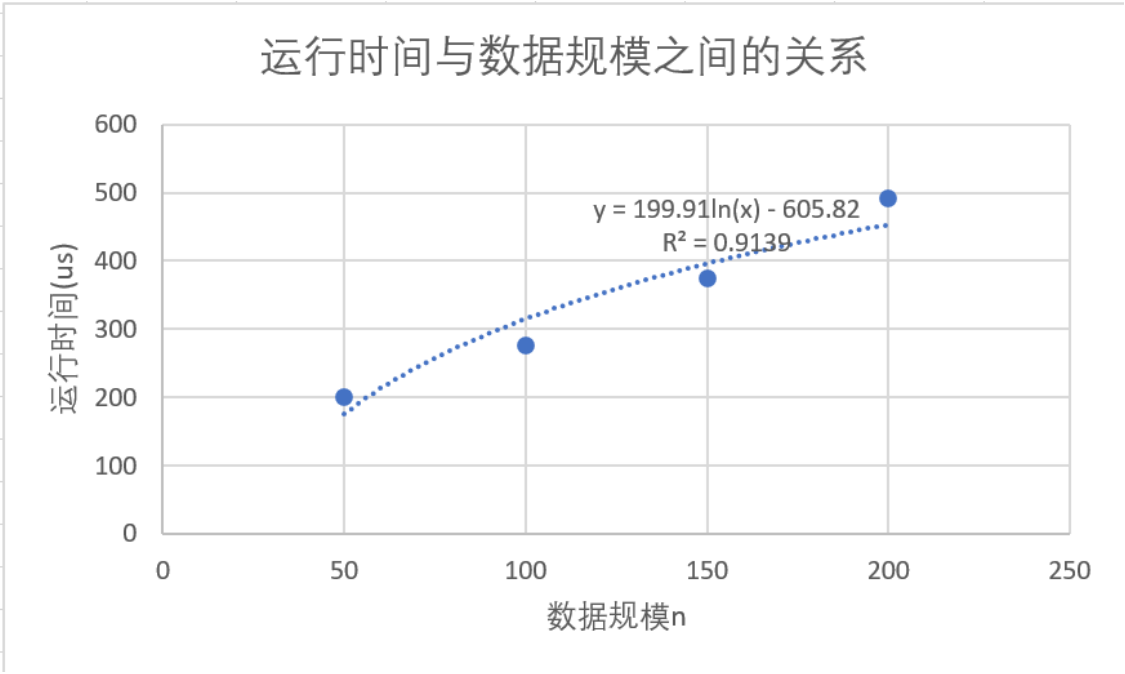
    故每个step的理论时间都为$O(lgn)$。

  运行结果：

```
step2的运行时间200.7us
step3的运行时间275.1us
step4的运行时间373.5us
step5的运行时间490.9us
step7的运行时间107.9us
```

  画图：

  将step2-5的运行时间与数据规模之间的关系画在下图中，可见符合$O(lgn)$的理论分析，拟合优度为0.9139。



Exp2

- 结果如下：

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

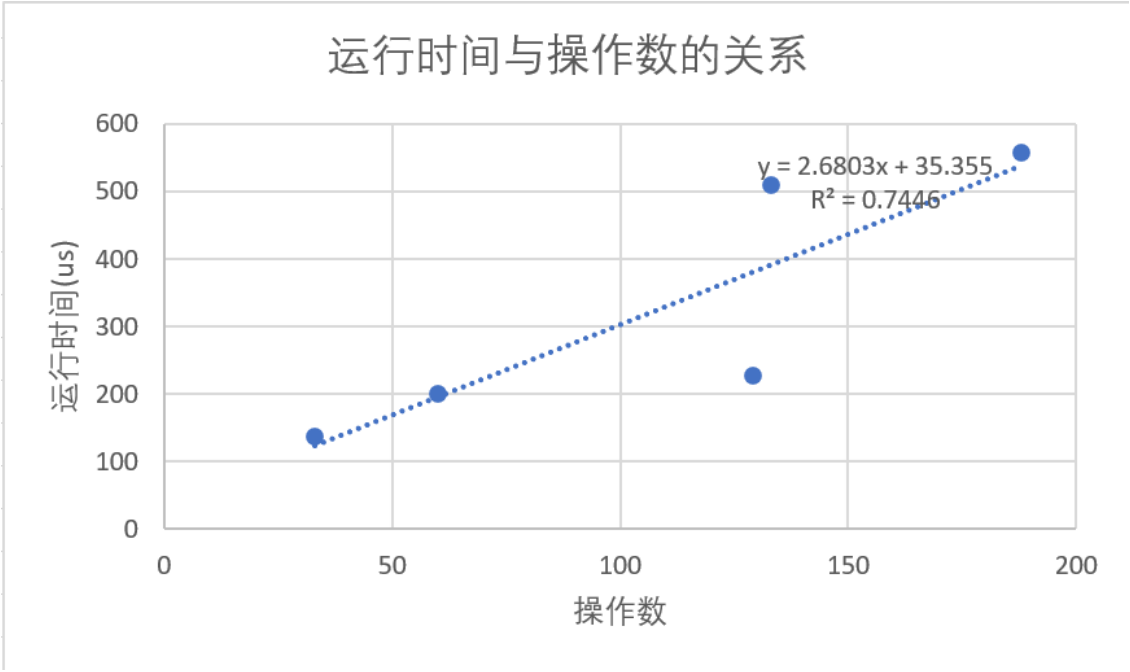
- 时间分析
  - 理论时间复杂度：

    当使用按秩合并与路径压缩时，最坏情况的运行时间为$O(m\alpha(n))$，其中m是操作数，$\alpha(n)$是一个分段函数，当$8 \leq n \leq 2047$时，$\alpha(n) = 3$。且在所有实际应用中，$\alpha(n) \leq 4$，其运行时间与m呈线性关系，但严格来说它是超线性的。
  - 运行结果：

```
n=10 137.1us
n=15 200.9us
n=20 227.1us
n=25 558.5us
n=30 508.6us
```

  - 画图：

    将实验得到的运行时间与操作数进行拟合，绘图可见，其基本符合理论分析中的线性关系，且拟合优度为$R^2 = 0.7446$。


运行时间与操作数的关系
600
500
400
300
200
100
0
运行时间(us)
0
50
100
150
200
操作数
y = 2.6803x + 35.355
R² = 0.7446