算法基础project2实验报告

PB19071535徐昊天

一.实验内容

1. 斐波那契堆。
2. 家族数。

二.实验设备和环境

- windows操作系统
- vscode、dev-c++
- Excel

三.实验方法和步骤

1.斐波那契堆

- 处理文件

  本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。涉及文件处理的代码如下：

```
FILE *fp1, *fp2, *fp3;
// 打开输入输出文件
if((fp1 = fopen("../input/2_1_input.txt","r"))==NULL)
{
    printf("cannot open input file\n");
    exit(0);
}
if((fp2 = fopen("../output/result.txt","w"))==NULL)
{
    printf("cannot open result.txt\n");
    exit(0);
}
if((fp3 = fopen("../output/time.txt","w"))==NULL)
{
    printf("cannot open time.txt\n");
    exit(0);
}
```

```
    ......
    ......
    ......
    for (i = 0; i < 50; i++)
    {
        fscanf(fp1, "%d", &val);
        num = insert(H1, val);
    }
    for (i = 0; i < 100; i++)
    {
        fscanf(fp1, "%d", &val);
        num = insert(H2, val);
    }
    for (i = 0; i < 150; i++)
    {
        fscanf(fp1, "%d", &val);
        num = insert(H3, val);
    }
    for (i = 0; i < 200; i++)
    {
        fscanf(fp1, "%d", &val);
        num = insert(H4, val);
    }
    ......
    ......
    ......
    //关闭输入输出文件
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
```

读取结点的 `key` 值并通过 `INSERT` 操作建立斐波那契堆H1-H4。

- 定义结构体

  根据课本中斐波那契堆与结点的成员和具体功能，分别定义结构体如下：

```
typedef struct FibNode
{
    struct FibNode *p;
    struct FibNode *child;
    struct FibNode *left;
    struct FibNode *right;
    int key;
    int degree;     // 孩子数目
    int mark;   // 自从上一次成为另一个结点的孩子后，是否失去过孩子
}FibNode;

typedef struct H
{
    int n;
    FibNode *min;
}H;
```

`FibNode` 为结点结构体， `H` 为斐波那契堆结构体。

- 创建一个新的斐波那契堆

分配并返回一个斐波那契堆对象H，其中H.n=0且H.min=NIL，H中不存在树，具体代码如下：

```
H *make_heap()
{
    H *Heap = (H*)malloc(sizeof(H));
    Heap->min = NULL;
    Heap->n = 0;
    return Heap;
}
```

- 插入一个结点

将结点x插入斐波那契堆H中，假设该结点已经被分配，x.key已经被赋值，具体代码如下：

```
int insert(H *heap, int x)
{
    FibNode *node;
    node = (FibNode*)malloc(sizeof(FibNode));
    node->p = NULL;
    node->key = x;
    node->child = NULL;
    node->mark = 0;
    node->degree = 0;
    if (heap->min == NULL)
    {
        node->right = node;
        node->left = node;
        heap->min = node;
    }
    else
    {
        node->right = heap->min->right;
        node->left = heap->min;
        heap->min->right->left = node;
        heap->min->right = node;
        if (x < heap->min->key)
            heap->min = node;
    }
    heap->n++;
    nodes[x] = node;
    return heap->n;
}
```

函数返回插入x节点后斐波那契堆H的节点数目。

- 两个斐波那契堆的合并

将两个斐波那契堆的根链表链接，确定最新的最小结点，具体代码如下：

```
H *Union(H *H1, H *H2)
{
    H *heap;
    heap = make_heap();
```

```
    heap->min = H1->min;
    int i, j;
    i = j = 0;
    H1->min->right->left = H2->min->left;
    H2->min->left->right = H1->min->right;
    H1->min->right = H2->min;
    H2->min->left = H1->min;
    if (H1->min == NULL || (H2->min != NULL && H2->min->key < H1->min->key))
        heap->min = H2->min;
    heap->n = H1->n + H2->n;
    return heap;
}
```

- 抽取最小结点

  首先将最小结点的每个孩子变为根结点，并从根链表中删除该最小结点。然后通过把具有相同度数的根结点合并的方法来链接成根链表，直到每个度数至多只有一个根在根链表中，具体代码如下：

```
FibNode *extract_min(H *heap)
{
    FibNode *z = heap->min;
    FibNode *x, *y;
    int i;
    nodes[heap->min->key] = NULL;
    if (z != NULL)
    {
        if (z->child != NULL)
        {
            y = z->child;
            for (i = 0; i < z->degree; i++)
            {
                x = y;
                y = y->right;
                x->p = NULL;
                x->left = heap->min;
                x->right = heap->min->right;
                heap->min->right->left = x;
                heap->min->right = x;
            }
        }
        z->child = NULL;
        z->degree = 0;
        z->left->right = z->right;
        z->right->left = z->left;
        heap->n--;
        if (z == z->right)
            heap->min = NULL;
        else
        {
            heap->min = z->right;
            consolidate(heap);
        }
    }
    return z;
}
```

需调用一个辅助过程 CONSOLIDATE ，通过一个辅助数组来记录根结点对应的度数的轨迹，进而减少斐波那契堆中树的数目，具体代码如下：

```
void consolidate(H *heap)
{
    int n = log2(heap->n);
    int d;
    FibNode *A[n+1];
    FibNode *x, *y, *swap;
    for (int i = 0; i <= n; i++)
        A[i] = NULL;
    int root = 1;
    FibNode *w = heap->min->left;
    FibNode *w_bro = heap->min;
    do
    {
        x = w_bro;
        d = x->degree;
        if (w == w_bro)
            root = 0;
        w_bro = w_bro->right;
        while (A[d] != NULL)
        {
            y = A[d];
            if (x->key > y->key)
            {
                swap = x;
                x = y;
                y = swap;
            }
            link(heap, y, x);
            A[d] = NULL;
            d++;
        }
        A[d] = x;
    }while(root);
    heap->min = NULL;
    for (int i = 0; i <= n; i++)
    {
        if(A[i] != NULL)
        {
            if(heap->min == NULL)
            {
                heap->min = A[i];
                heap->min->p = NULL;
                heap->min->left = heap->min;
                heap->min->right = heap->min;
            }
            else
            {
                A[i]->right = heap->min->right;
                heap->min->right->left = A[i];
                A[i]->left = heap->min;
                heap->min->right = A[i];
                if (A[i]->key < heap->min->key)
                    heap->min = A[i];
            }
        }
    }
}
```

调用 LINK 过程，具体代码如下：

```
void link(H *heap, FibNode *y, FibNode *x){
    FibNode *node;
    y->left->right=y->right;
    y->right->left=y->left;
    if(x->child==NULL){
        x->child=y;
        y->right=y;
        y->left=y;
        y->p=x;
    }
    else{
        node=x->child;
        node->right->left=y;
        y->right=node->right;
        y->left=node;
        node->right=y;
        y->p=x;
    }
    x->degree++;
    y->mark=0;
}
```

- 关键字减值

假定从一个链表中移除一个结点不改变被移除的结点的任何结构属性，具体代码如下：

```
FibNode *decrease_key(H *heap, FibNode *x, int k)
{
    if (k > x->key)
    {
        printf("new key is greater than current key!\n");
        exit(0);
    }
    nodes[k] = x;
    nodes[x->key] = NULL;
    x->key = k;
    FibNode *y;
    y = x->p;
    if(y != NULL && x->key < y->key)
    {
        cut(heap, x, y);
        cascanding_cut(heap, y);
    }
    if (x->key < heap->min->key)
        heap->min = x;
    return heap->min;
}
```

调用 CUT 过程和 CASCADING-CUT 过程。 CUT 过程切断x与其父节点y之间的链接，使x成为根结点； CASCADING-CUT 过程执行级联切断操作，沿着树一直递归向上，直到它遇到根结点或者一个未被标记的结点。具体代码如下：

```
void cut(H *heap, FibNode *x, FibNode *y)
```

```
{
    if (x == y->child)
    {
        if (x->right != x)
            y->child = x->right;
        else
            y->child = NULL;
    }
    x->p = NULL;
    y->degree--;
    x->left->right = x->right;
    x->right->left = x->left;
    x->right = heap->min->right;
    x->left = heap->min;
    heap->min->right->left = x;
    heap->min->right = x;
    x->mark = 0;
}

void cascanding_cut(H *heap, FibNode *y)
{
    FibNode *z;
    z = y->p;
    if (z != NULL)
    {
        if (y->mark == 0)
            y->mark = 1;
        else
        {
            cut(heap, y, z);
            cascanding_cut(heap, z);
        }
    }
}
```

- 删除一个结点

  删除操作可直接通过调用其他函数实现，具体代码如下：

```
int delete(H *heap, FibNode *x)
{
    decrease_key(heap, x, 0);
    FibNode *y = extract_min(heap);
    nodes[x->key] = NULL;
    return heap->n;
}
```

  返回值为删除结点后斐波那契堆H的结点数目。

- 记录算法运行时间

  利用头文件 `#include<windows.h>` 中 `QueryPerformance` 工具记录时间，结果以微秒为单位，实现代码如下：

```
LONGLONG begin, over, freq;
LARGE_INTEGER number;
QueryPerformanceFrequency(&number);
freq = number.QuadPart;
QueryPerformanceCounter(&number);
begin = number.QuadPart;
......
......
......
QueryPerformanceCounter(&number);
over = number.QuadPart;
fprintf(fp3, "step2:%lld\n", (over - begin)*10000000/freq);
```

用于生成的时间乘以了 `10000000` ，故得到时间单位为 `10^(-4)ns` 。

2.家族数

- 处理文件

本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。涉及文件处理的代码如下：

```
    FILE *fp1, *fp2, *fp3;
    // 打开输入输出文件
    if((fp1 = fopen("../input/2_2_input.txt","r"))==NULL)
    {
        printf("cannot open input file\n");
        exit(0);
    }
    if((fp2 = fopen("../output/result.txt","w"))==NULL)
    {
        printf("cannot open result.txt\n");
        exit(0);
    }
    if((fp3 = fopen("../output/time.txt","w"))==NULL)
    {
        printf("cannot open time.txt\n");
        exit(0);
    }
    ......
    ......
    ......
    for (i = 0; i < 5; i++)
    {
        int family[N][N];
        for (j = 0; j < N; j++)
        {
            for (k = 0; k < N; k++)
            {
                fscanf(fp1, "%d", &family[j][k]);
            }
        }
        ......
        ......
        ......
        //关闭输入输出文件
        fclose(fp1);
```

```
        fclose(fp2);
        fclose(fp3);
    }
```

将N个人之间的亲戚关系读入 `family` 矩阵。

- 定义结构体

  根据课本中不相交集合森林结点的成员和具体功能，定义结构体如下：

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

- 实现不相交集合森林

  通过 `UNION` 过程合并不相交集合森林， `UNION` 操作有两种情况，取决于两棵树的根是否有相同的秩，若无相同的秩，则让较大秩的根成为较小秩的根的父结点，秩不变化；否则任选一个作为父结点，并使其秩加一；通过路径压缩手段实现不相交集合森林代码如下：

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

void link(Node *x, Node *y)
{
    if (x->rank > y->rank)
    {
        y->p = x;
    }
    else
    {
        x->p = y;
        if (x->rank == y->rank)
            y->rank = y->rank + 1;
    }
}

void Union(Node *x, Node *y)
{
    link(find_set(x), find_set(y));
}
```

- 遍历家族关系

  通过两层循环遍历存储家庭关系的矩阵，若值为1则表示两者存在亲戚关系，需被合并入同一个不相交集合森林中，具体实现代码如下：

```
        for (j = 0; j < N; j++)
        {
            for (k = j; k < N; k++)
            {
                if (k != j)
                {
                    if ((family[j][k] == 1 || family[k][j] == 1 ) && find_set(node[j]) !=
find_set(node[k]))
                    {
                        Union(node[j], node[k]);
                    }
                }
            }
        }
```

- 计算家族数

  通过两层循环判断结点之间 `find_set` 结果是否相同，若存在新的不相交集合森林，则家族数加一，具体实现代码如下：

```
        for (j = 0; j < N; j++)
        {
            int flag = 1;
            for (k = 0; k < j; k++)
            {
                if (find_set(node[j]) == find_set(node[k]))
                    flag = 0;
            }
            if (flag == 1)
                num++;
        }
```

- 记录算法运行时间

  利用头文件 `#include<windows.h>` 中 `QueryPerformance` 工具记录时间，结果以微秒为单位，实现代码如下：

```
LONGLONG begin, over, freq;
LARGE_INTEGER number;
QueryPerformanceFrequency(&number);
freq = number.QuadPart;
QueryPerformanceCounter(&number);
begin = number.QuadPart;
......
......
......
QueryPerformanceCounter(&number);
over = number.QuadPart;
fprintf(fp3, "step2:%lld\n", (over - begin)*10000000/freq);
```

  用于生成的时间乘以了 `10000000` ，故得到时间单位为 `10^(-4)ns` 。

四.实验结果和分析

1.斐波那契堆

- step2-5,7的10个ops结果

  执行程序后打印的结果如下图所示：

126-徐昊天-PB19071535-project2 > ex1 > output > 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

```

- 时间复杂度分析

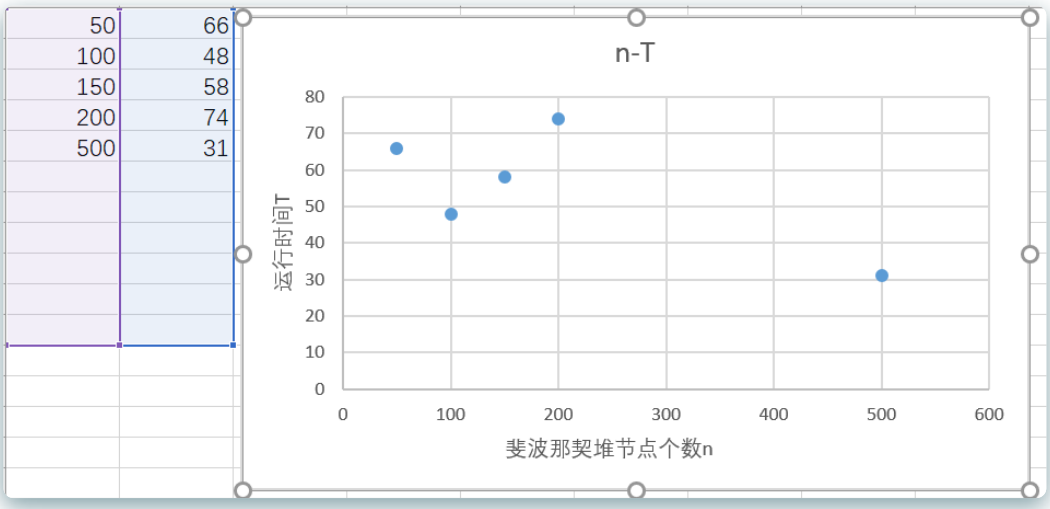
  程序运行后 `time.txt` 中时间如下图所示：

126-徐昊天-PB19071535-project2 > ex1 > output > time.txt

```
step2:66
step3:48
step4:58
step5:74
step7:31

```

  将数据在excel中散点图如下图所示：



根据散点图中算法运行时间与斐波那契堆结点个数的关系可知，实际时间复杂度与理论情况偏差较大。

理论时间复杂度与根链表节点数成正比，而实际运行时间n=50时结点数最小而运行时间偏大；合并得到 H5 后根链表结点数最大，而时间最小；n=100、150、200时运行时间趋于线性关系，较为符合理论情况。

偏差原因：测试数据样本较小，存在较大偶然性和随机性，关键语句执行次数未必符合理论情况，故难以得到和理论情况接近的结果。

2.家族数

- 家族数结果

  运行程序后打印结果如下图所示：

  126-徐昊天-PB19071535-project2 > ex2 > output > result.txt

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

- 时间复杂度分析

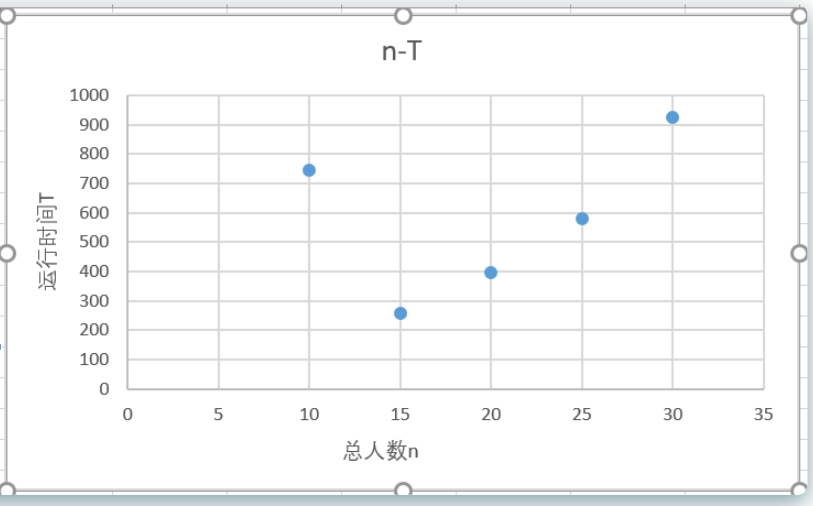
  程序运行后 time.txt 中时间如下图所示：

  126-徐昊天-PB19071535-project2 > ex2 > output > time.txt

```
n=10 时间:747
n=15 时间:256
n=20 时间:397
n=25 时间:582
n=30 时间:925
```

  将数据在excel中散点图如下图所示：

10	747
15	256
20	397
25	582
30	925



根据散点图中算法运行时间与总人数的关系可知，实际时间复杂度与理论情况存在一定偏差。

根据实现算法，理论时间复杂度为O(n^2)，而实际运行时间n=10时人数最小而运行时间较大；n=15、20、25、30时运行时间较为符合理论情况。

偏差原因：测试数据样本较小，存在较大偶然性和随机性，关键语句执行次数未必符合理论情况，故难以得到和理论情况接近的结果。

五.实验收获与感想

1. 强化了对书本上有关斐波那契堆和不相交集合森林算法的理解，通过对多种测试数据的执行，对算法的时间复杂度有了更深刻的理解。
2. 通过处理较为复杂的数据结构及其ADT，提升了对C语言的运用和调试能力。