Shortest Path Algorithm with Heaps

Chapter1 Introduction

Dijkstra algorithm is a common method to find the shortest path. However, this method requires finding the point with the smallest unknown distance when updating the EntryTable. In order to shorten the running time of dijkstra algorithm, ordinary heap and Fibonacci heap are used for optimization, and the optimization quality of the two methods is compared.

Chapter2 Run time table

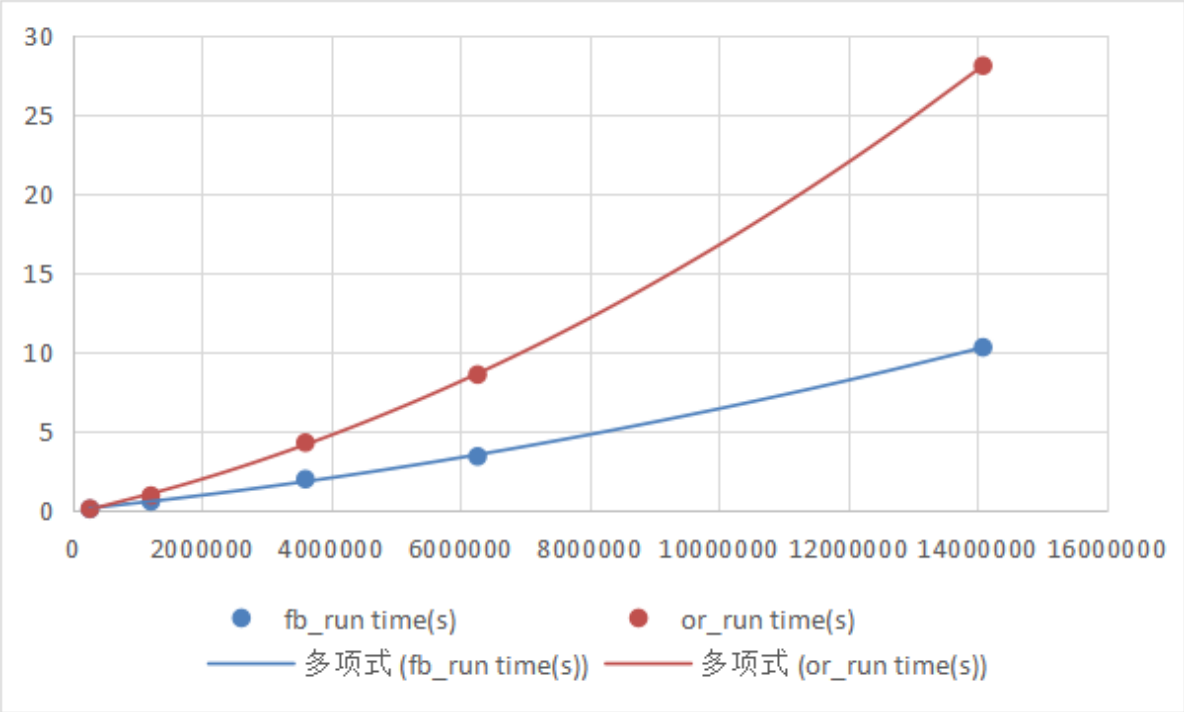
Fibonacci min heap

num of vertexs	num of edges	time(s)	download link of data set
264346	733846	0.123	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NY.gr.gz
1207945	2840208	0.602	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NW.gr.gz
3598623	8778114	2.007	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.E.gr.gz
6262104	15248146	3.439	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.W.gr.gz
14081816	34292496	10.326	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CTR.gr.gz

Ordinary min heap

num of vertexs	num of edges	time(s)	download link of data set
264346	733846	0.14	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NY.gr.gz
1207945	2840208	0.965	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.NW.gr.gz
3598623	8778114	4.309	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.E.gr.gz
6262104	15248146	8.604	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.W.gr.gz
14081816	34292496	28.096	http://www.diag.uniroma1.it//challenge9/data/USA-road-d/USA-road-d.CTR.gr.gz

Chart



纵轴坐标是运行时间，横轴坐标是input size

Chapter3 Analysis and Comments

As can be seen from the graph, the effect of Fibonacci heap optimization is better than that of ordinary heap. The lines between points in a graph can be approximated by a polynomial curve.

However, this intuitive analysis is not very accurate, because its abscissa only includes the number of points, and to know the exact time complexity requires a detailed analysis.

For the normal heap, let the number of points be V and the number of edges be E, because when updating the EntryTable, we need to loop V times, and the time complexity of the correction heap is O(logE), so the time complexity of the normal heap is O(VlogE).

The Fibonacci heap is a collection of ordered trees of the smallest heaps in computer science. It has similar properties to the binomial heap and can be used to implement merge priority queues. Operations that do not involve deleting elements have O(1) amortized time. The number of extract-min and Delete is more efficient when compared to others. Each Decrease-key in a dense graph requires only O(1) amortization time, which is a huge improvement over O(LGN) in binomial heap.

The key idea of Fibonacci heaps is to delay heap maintenance as much as possible. For example, when inserting a new node into the Fibonacci heap or merging two Fibonacci heaps, the tree is not merged but left to the extract-min operation.So the time complexity is O(E+Vlog(V)).

Chapter4 Appendix

pj2_nheap.h

```
#ifndef _PJ3_NHEAP_H
#define _PJ3_NHEAP_H
#define NotAVetex (0)

typedef int Vertex;
```

```
typedef struct Adjlist *List;
struct Adjlist//有向图的邻接表
{
    List NextNode;
    Vertex n;
    int dist;
};
struct TableEntry
{
    List Header;
    int Known;
    int Dist;
    Vertex Path;
};
typedef struct TableEntry Table[30000000];
int m, n;
Vertex Start;
Table T;
struct n_Heap
{
    int heap[70000000];
    int position[70000000];
    int size;
};
#define INF (100000000)
struct n_Heap heap;
void swap(int p1, int p2);//交换堆中的节点
void Initial(Table T);//初始化记录最短路径的表
void ReadGraph(Table T);//读入有向图
void n_Dijkstra(Table T);//使用普通最小堆对Dijkstra算法进行优化
void PrintPath(Table T, Vertex V);
void initial_nheap();//初始化堆
int Mindis(int fa, int ls, int rs);//找到父节点和两个子节点中的最小节点
void p_down(int v);//percolate down
void p_up(int v);  // percolate down
#endif
```

pj2_nheap.c

```
#include <stdio.h>
#include <stdlib.h>
#include "pj2_nheap.h"
void swap(int p1, int p2)
{
    int t = heap.heap[p1];
    heap.heap[p1] = heap.heap[p2];
    heap.heap[p2] = t;
    t = heap.position[heap.heap[p1]];
    heap.position[heap.heap[p1]] = heap.position[heap.heap[p2]];
    heap.position[heap.heap[p2]] = t;
}
int Mindis(int fa, int ls, int rs)
{
    int min = fa;
    int mind = T[heap.heap[fa]].Dist;
    if (ls <= heap.size && mind > T[heap.heap[ls]].Dist)
    {
```

```
        mind = T[heap.heap[ls]].Dist;
        min = ls;
    }
    if (rs <= heap.size && mind > T[heap.heap[rs]].Dist)
    {
        mind = T[heap.heap[rs]].Dist;
        min = rs;
    }
    return min;
}
void p_down(int v)
{
    int min;
    while (1)
    {
        min = Mindis(v, 2 * v, 2 * v + 1);
        if (min != v)
        {
            swap(v, min);
            v = min;
        }
        else
            break;
    }
}
void p_up(int v)
{
    int min;
    while (1)
    {
        int fa = v  / 2;
        if(fa==0)
            break;
        min = Mindis(fa, 2 * fa, 2 * fa + 1);
        if (min != fa)
        {
            swap(fa, min);
            v = fa;
        }
        else
            break;
    }
}

void Initial(Table T)
{
    for(int i=1;i<=n;i++)
    {
        T[i].Header = (List)malloc(sizeof(struct Adjlist));
        T[i].Header->NextNode=NULL;
        T[i].Known = 0;
        T[i].Dist=INF;
        T[i].Path =NotAVetex;
    }
    T[Start].Dist=0;
}
void ReadGraph(Table T)
{
```

```
    for(int i=0;i<n;i++)
    {
        int v1,v2,dis;
        scanf("%d %d %d",&v1,&v2,&dis);
        List p= T[v1].Header;
        while(p!=NULL){
            p=p->NextNode;
        }
        p=(List)malloc(sizeof(struct Adjlist));
        p->n=v2;
        p->dist=dis;
    }
}

void n_Dijkstra(Table T)
{
    T[Start].Dist=0;
    initial_nheap();
    Vertex V,W;
    for(int i =heap.size/2;i>0;i--)
    {
        p_down(i);
    }
    // int cnt=0;
    while(1)
    {
        if(heap.size>0)
            V=heap.heap[1];
        else
            V=NotAVetex;
        if(V==NotAVetex)
            break;
        swap(1,heap.size--);
        p_down(1);
        T[V].Known=1;
        List p=T[V].Header->NextNode;
        while(p!=NULL){
            if(!T[p->n].Known){
                if(T[V].Dist+p->dist<T[p->n].Dist)
                {
                    T[p->n].Dist = T[V].Dist + p->dist;
                    T[p->n].Path = V;
                    p_up(heap.position[p->n]);
                }

            }
            p=p->NextNode;
        }

    }
}
void PrintPath(Table T,Vertex V)
{
    if(T[V].Path!=NotAVetex)
    {
        PrintPath(T,T[V].Path);
        printf("to");
    }
```

```
    printf("%d",V);
}
void initial_nheap()
{
    heap.size = m;
    for (int i = 1; i <= heap.size; i++)
    {
        heap.heap[i] = i;
        heap.position[i]=i;
    }
}
```

pj2_fheap.h

```
#ifndef _FIBONACCI_HEAP_H_
#define _FIBONACCI_HEAP_H_
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "pj2_fheap.h"
#include "pj2_nheap.h"
#define LOG2(x) ((log((double)(x))) / (log(2.0)))

typedef int Type;

//定义斐波那契堆的节点
typedef struct _FibonacciNode
{
    Type key;
    int degree;
    struct _FibonacciNode *left;
    struct _FibonacciNode *right;
    struct _FibonacciNode *child;
    struct _FibonacciNode *parent;
    int marked;
} FibonacciNode, FibNode;

//定义斐波那契堆
typedef struct _FibonacciHeap
{
    int keyNum;
    int maxDegree;
    struct _FibonacciNode *min;
    struct _FibonacciNode **cons;
} FibonacciHeap, FibHeap;

//移去斐波那契堆的节点
void remove_node(FibNode *node);

//增加节点
void add_node(FibNode *node, FibNode *root);

//创建一个斐波那契堆
FibHeap *make_heap();
```

```
//创建一个斐波那契堆的节点
FibNode *make_node(Type key);

//插入一个节点
void insert_node(FibHeap *heap, FibNode *node);

//插入一个值（通过插入节点实现）
void insert_key(FibHeap *heap, Type key);

//将min从根链表中移除
FibNode *remove_min(FibHeap *heap);

//连接节点和根
void link(FibHeap *heap, FibNode *node, FibNode *root);

//创建consolidate所需空间
void make_cons(FibHeap *heap);

//合并斐波那契堆的根链表中左右相同度数的树
void consolidate(FibHeap *heap);

 // 移除最小节点，并返回移除节点后的斐波那契堆
FibNode *min_off(FibHeap *heap);

//释放最小节点空间
void min_free(FibHeap *heap);

//更新度数
void renew_degree(FibNode *parent, int degree);

void cut_heap(FibHeap *heap, FibNode *node, FibNode *parent);
void cascading(FibHeap *heap, FibNode *node);

//将斐波那契堆heap中节点node的值减少为key
void decrease_heap(FibHeap *heap, FibNode *node, Type key);
FibNode *search_node(FibNode *root, Type key);
FibNode *search_heap(FibHeap *heap, Type key);

//删去键值为key的节点
void delete_key(FibHeap *heap, Type key);

//采用斐波那契堆来优化Dijkstra算法
void f_Dijkstra(Table T);
#endif
```

pj2_fheap.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "pj2_fheap.h"
#include "pj2_nheap.h"

#define LOG2(x) ((log((double)(x))) / (log(2.0)))
```

```
 void remove_node(FibNode *node)
{
    node->left->right = node->right;
    node->right->left = node->left;
}

 void add_node(FibNode *node, FibNode *root)
{
    node->left = root->left;
    root->left->right = node;
    node->right = root;
    root->left = node;
}

FibHeap *make_heap()
{
    FibHeap *heap;

    heap = (FibHeap *)malloc(sizeof(FibHeap));
    if (heap == NULL)
    {
        printf("Error: make FibHeap failed\n");
        return NULL;
    }

    heap->keyNum = 0;
    heap->maxDegree = 0;
    heap->min = NULL;
    heap->cons = NULL;

    return heap;
}

 FibNode *make_node(Type key)
{
    FibNode *node;

    node = (FibNode *)malloc(sizeof(FibNode));
    if (node == NULL)
    {
        printf("Error: make Node failed\n");
        return NULL;
    }
    node->key = key;
    node->degree = 0;
    node->left = node;
    node->right = node;
    node->parent = NULL;
    node->child = NULL;

    return node;
}


 void insert_node(FibHeap *heap, FibNode *node)
{
    if (heap->keyNum == 0)
        heap->min = node;
```

```
    else
    {
        add_node(node, heap->min);
        if (T[node->key].Dist < T[heap->min->key].Dist)
            heap->min = node;
    }
    heap->keyNum++;
}


void insert_key(FibHeap *heap, Type key)
{
    FibNode *node;

    if (heap == NULL)
        return;

    node = make_node(key);
    if (node == NULL)
        return;

    insert_node(heap, node);
}


 FibNode *remove_min(FibHeap *heap)
{
    FibNode *min = heap->min;

    if (heap->min == min->right)
        heap->min = NULL;
    else
    {
        remove_node(min);
        heap->min = min->right;
    }
    min->left = min->right = min;

    return min;
}

 void link(FibHeap *heap, FibNode *node, FibNode *root)
{
    remove_node(node);
    if (root->child == NULL)
        root->child = node;
    else
        add_node(node, root->child);

    node->parent = root;
    root->degree++;
    node->marked = 0;
}

 void make_cons(FibHeap *heap)
{
    int old = heap->maxDegree;
```

```
    heap->maxDegree = LOG2(heap->keyNum) + 1;

    if (old >= heap->maxDegree)
        return;

    heap->cons = (FibNode **)realloc(heap->cons,
                                     sizeof(FibHeap *) * (heap->maxDegree + 1));
}


 void consolidate(FibHeap *heap)
{
    int i, d, D;
    FibNode *x, *y, *tmp;

    make_cons(heap);
    D = heap->maxDegree + 1;

    for (i = 0; i < D; i++)
        heap->cons[i] = NULL;

    while (heap->min != NULL)
    {
        x = remove_min(heap);
        d = x->degree;
        while (heap->cons[d] != NULL)
        {
            y = heap->cons[d];
            if (T[x->key].Dist > T[y->key].Dist)
            {
                tmp = x;
                x = y;
                y = tmp;
            }
            link(heap, y, x);
            heap->cons[d] = NULL;
            d++;
        }
        heap->cons[d] = x;
    }
    heap->min = NULL;

    for (i = 0; i < D; i++)
    {
        if (heap->cons[i] != NULL)
        {
            if (heap->min == NULL)
                heap->min = heap->cons[i];
            else
            {
                add_node(heap->cons[i], heap->min);
                if (T[(heap->cons[i])->key].Dist < T[heap->min->key].Dist)
                    heap->min = heap->cons[i];
            }
        }
    }
}
```

```
FibNode *min_off(FibHeap *heap)
{
    if (heap == NULL || heap->min == NULL)
        return NULL;

    FibNode *child = NULL;
    FibNode *min = heap->min;
    while (min->child != NULL)
    {
        child = min->child;
        remove_node(child);
        if (child->right == child)
            min->child = NULL;
        else
            min->child = child->right;

        add_node(child, heap->min);
        child->parent = NULL;
    }

    remove_node(min);
    if (min->right == min)
        heap->min = NULL;
    else
    {
        heap->min = min->right;
        consolidate(heap);
    }
    heap->keyNum--;

    return min;
}

void min_free(FibHeap *heap)
{
    FibNode *node;

    if (heap == NULL || heap->min == NULL)
        return;

    node = min_off(heap);
    if (node != NULL)
        free(node);
}

 void renew_degree(FibNode *parent, int degree)
{
    parent->degree -= degree;
    if (parent->parent != NULL)
        renew_degree(parent->parent, degree);
}


 void cut_heap(FibHeap *heap, FibNode *node, FibNode *parent)
{
    remove_node(node);
    renew_degree(parent, node->degree);
```

```
    if (node == node->right)
        parent->child = NULL;
    else
        parent->child = node->right;

    node->parent = NULL;
    node->left = node->right = node;
    node->marked = 0;
    add_node(node, heap->min);
}


 void cascading(FibHeap *heap, FibNode *node)
{
    FibNode *parent = node->parent;
    if (parent != NULL)
        return;

    if (node->marked == 0)
        node->marked = 1;
    else
    {
        cut_heap(heap, node, parent);
        cascading(heap, parent);
    }
}


 void decrease_heap(FibHeap *heap, FibNode *node, Type key)
{
    FibNode *parent;

    if (heap == NULL || heap->min == NULL || node == NULL)
        return;

    if (T[key].Dist >= T[node->key].Dist)
    {
        return;
    }

    node->key = key;
    parent = node->parent;
    if (parent != NULL && T[node->key].Dist < T[parent->key].Dist)
    {
        cut_heap(heap, node, parent);
        cascading(heap, parent);
    }
    if (T[node->key].Dist < T[heap->min->key].Dist)
        heap->min = node;
}


 FibNode *search_node(FibNode *root, Type key)
{
    FibNode *t = root; // 临时节点
    FibNode *p = NULL; // 要查找的节点

    if (root == NULL)
```

```
        return root;

    do
    {
        if (t->key == key)
        {
            p = t;
            break;
        }
        else
        {
            if ((p = search_node(t->child, key)) != NULL)
                break;
        }
        t = t->right;
    } while (t != root);

    return p;
}


 FibNode *search_heap(FibHeap *heap, Type key)
{
    if (heap == NULL || heap->min == NULL)
        return NULL;

    return search_node(heap->min, key);
}


void delete_key(FibHeap *heap, Type key)
{
    FibNode *node;

    if (heap == NULL || heap->min == NULL)
        return;

    node = search_heap(heap, key);
    if (node == NULL)
        return;

    Type min = heap->min->key;
    decrease_heap(heap, node, min - 1);
    min_off(heap);
    free(node);
}

void f_Dijkstra(Table T)
{
    T[Start].Dist = 0;
    FibHeap* heap=make_heap();
    for(int i=1;i<=m;i++)
    {
        insert_key(heap,i);
    }
    Vertex V, W;
    for(int i=1;i<=m;i++)
    {
```

```
        V=heap->min->key;
        min_off(heap);
        T[V].Known = 1;
        List p = T[V].Header->NextNode;
        while (p != NULL)
        {
            if (!T[p->n].Known)
            {
                if (T[V].Dist + p->dist < T[p->n].Dist)
                {
                    T[p->n].Dist = T[V].Dist + p->dist;
                    T[p->n].Path = V;
                    delete_key(heap,V);
                    insert_key(heap,V);
                }
            }
            p = p->NextNode;
        }
    }
}
```

test.c

```
#include <stdio.h>
#include <stdlib.h> //  exit() 函数
#include <time.h>
#include <string.h>
#include "pj2_nheap.h"//普通最小堆优化
#include "pj2_fheap.h"//斐波那契最小堆优化
clock_t start, end;//记录时间的开始和结束
int char2int(char *a)//将字符串转换为整数
{
    int i = 0;
    int sum = 0;
    while (*(a + i) != '\0')
    {
        sum = (*(a + i) - '0') + sum * 10;
        i++;
    }
    return sum;
}
int main()
{
    char c[1000];
    Start = 1;
    FILE *fptr;
    if ((fptr = fopen("USA-road-d.NY.gr", "r")) == NULL)//填入要打开的文件地址，如果
文件打不开，则报错
    {
        printf("Error! opening file");
        exit(1);
    }
    int cnt = 0;
    while (!feof(fptr))
    {
        fscanf(fptr, "%s", c);//每隔一个空格读入
        if (!strcmp(c,"c"))//“c”代表注释信息，跳过
```

```
            continue;
        if (!strcmp(c, "p"))//代表vertex和edge的信息
        {
            fscanf(fptr, "%s", c);
            fscanf(fptr, "%s", c);
            m = char2int(c);//m在pj3_nheap.h里定义，储存vertex数
            fscanf(fptr, "%s", c);
            n = char2int(c); // n在pj3_fheap.h里定义，储存edge数
            Initial(T);
        }
        if (!strcmp(c, "a"))
        {
                //读入有向边
                int v1, v2, dis;
                // scanf("%d %d %d", &v1, &v2, &dis);
                fscanf(fptr, "%s", c);
                v1 = char2int(c);
                fscanf(fptr, "%s", c);
                v2 = char2int(c);
                fscanf(fptr, "%s", c);
                dis = char2int(c);
                List p = T[v1].Header;
                while (p->NextNode != NULL)
                {
                    p = p->NextNode;
                }
                p->NextNode = (List)malloc(sizeof(struct Adjlist));
                p->NextNode->NextNode=NULL;
                p->NextNode->n = v2;
                p->NextNode->dist = dis;
        }
    }
    fclose(fptr);
    start = clock();
    n_Dijkstra(T); //要测试斐波那契堆的时间，则写成"f_Dijkstra(T)";要测试普通最小堆的时
间，则写成"n_Dijkstra(T)"
    end = clock();
    printf("time=%f\n", (double)(end - start) / CLOCKS_PER_SEC); //输出时间

    return 0;
}
```