Project 1

Shortest Path Algorithm with Heaps

Author Names

Zhang Tongchen（张童晨）

Wang Ziteng（王子腾）

Li Xiang（李想）

Date: 2020-03-29

Chapter 1: Introduction

Shortest path problem is a classical optimization problem among the research of graph theory. It aims at the generalized problem of finding the shortest path among vertices, which is why it has gained popularity from a wide range of fields including Computer Science, Traffic Engineering, Communication Engineering, etc.

Dijkstra algorithm is one of the most representative algorithms to solve the problem discussed above. As we all know, the Dijkstra algorithm involves finding the uncollected vertex with the smallest distance. If we traverse the whole graph every time trying to find that vertex, it might costs a lot of unnecessary time. Consequently, we implement the algorithm with four different kind of min-priority queues, the basic heap, the Fibonacci heap, the Skew heap and the Leftist heap every time finding the vertex with smallest distance.

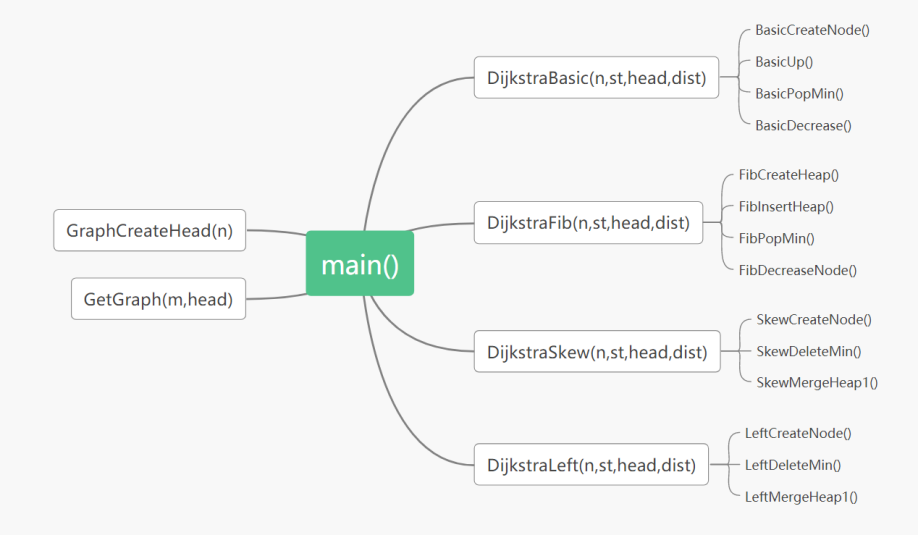
Generally speaking, we aim to find the best data structure for Dijkstra algorithm with four candidates discussed above. In order to measure the performance of those heaps precisely, we use a rather large data and sufficient amount of repetition to analyze the time complexity as well as measure the performance.

Chapter 2: Algorithm Specification

Our code can successfully solve the problem. And we are going to specify it with the following aspect.

1.sketch of the main program(includes data structure)

The function structure are as follows:



The whole problem generates the shortest path from the input data. The main function, which is in project1.c, is intended for user-interaction. Read in data from dataNY.txt and print out the results to solution.txt. The result is stored in the 'disk' array. The main program first call function GraphCreateHead() to create an array 'head' to store the head node for each point, as well as the function GetGraph() to read in m edges from input stream and build the graph. We use the idea of adjacency list. Beside the 'head' array, We use an extra structure to define the adjacent node. The data structure used is as follows:

```
typedef struct AdjNode* pAdjNode;

struct AdjNode
{
    pAdjNode next;      //next edge link to u
    int v,w;            //v the other point of the edge
                        //w the length of the edge
};
```

After building the graph, the program follows the user's instruction and call different version of Dijkstra algorithm. The parameters of each version of Dijkstra includes the number of vertices, the starting point which is generate by function rand(), the 'head' array which is used to represent the graph, and the 'dist' array for the output. The endpoint is used as a global variable.

2. Pseudocode of core algorithm and data structure

in different version of Dijkstra algorithm, we use different technics building the corresponding heap. However, the main idea of Dijkstra algorithm bears remarkable resemblance. **Consequently, we only makes a template over four different version of Dijkstra algorithm, yet mark out where the differences might be.**

1) Dijkstra algorithm(template)

```
void Dijkstra_template(int number, int start, Graph G, typedist *dist)
{
    Vertex V,W;
    for every vertex V {
        initialize dist[v] with infinite value;
        make V uncollected;
    }
    Create node of different heap with start vertex;
    Build heap with the node;            //different version
    dist[start]=0;
    while(heap is not empty){
        V=the node with the min distance and uncollected;
         //implemented by different version of function DeleteMin()
        if can't find such V
         break;
        make V collected;
        for each W adjacent to V and uncollected{
            if (dist[V]+dist from V to W<dist[W]){
                update dist[w];
                Insert V in the heap; //different version
            }
        }
      }
    clear the heap;
}
```

When checking each W adjacent to V and uncollected, there's a slight difference between {skew heap, leftist heap} and {basic heap, Fibonacci heap}. **The former pushes V whenever there's a better solution, the latter pushes V when it first become collected and decrease the value of the corresponding node by calling function BasicDecrease() or FibDecrease() if it has already been collected.**

2) Fibonacci heap(Basic heap)

Both basic heap and Fibonacci heap shares similar API. We illustrate the idea of Fibonacci heap with pseudocode, and basic heap could be easily understand.

Data structure for Fibonacci heap:

it's quite simple to understand as follows, MinNode represents the smallest node in the heap.

```
//point at a FibNode
typedef struct FibNode * pFibNode;
struct FibNode
{
    typekey key;
    int degree;
    int id;
    pFibNode left,right,child,father;
    int flag;           //1 means one child has been deleted
};

typedef struct FibHeap *pFibHeap;
struct FibHeap
{
    int num;
    // int MaxDegree;
    pFibNode MinNode;
};
```

Core algorithm

```
Fibonacci Heap

void FibInsertHeap(Heap H, Node p){
    if H is empty
        create the list;
    else {
        insert p into heap->MinNode;
        if(p is smaller than MinNode) p=MinNode;
    }
    H->num++;
}

Node FibPopMin(Heap H){
    if(H is empty) return NULL;
    for all the child of the MinNode{
        remove the child from the children list of MinNode;
        update the children list of MinNode;
        insert the child node back into H
    }
    remove the MinNode;
    FibConsolidate();
    //update the MinNode;
    //use function FibConsolidate() which union the the node with same degree,like
1011 + 1 = 1100
    update the number of node in the heap;
}
void FibDecreaseNode(Heap H, Node p, Keytype key){
    if(H ==NULL or p==NULL or key>previous key) return;
    p->key=key;
    if(p->key<p->father->key){
        remove p from list;
```

```
            Insert p into the root list;      //cut the node
            FibCascadeCut();
            //if father has a child deleted, cut father and cut the grandfather.
        }
        update MinNode;

void FibConsolidate(Heap h){
        int i;
        int size = max size of heap;
        for(i=0;i<n;i++){
            initialize the node array A[];
        }
        while(h->MinNode!=NULL){
            pFibNode x=h->MinNode;
            int d=x->degree;
            FibRemList(x);    //remove x from root list;
            if(x->right==x) h->MinNode=NULL;
            else h->MinNode=H->right;
            h->left=h->right=h;
            //update the minNode;
            while(A[d]!=NULL){
                pFibNode y=A[d];
                if(x->key>y->key) swap(x,y);
                Link y to x;
                A[d]=NULL;
                d++;
            }
            A[d]=x;
        }
        Add heap back into the root list;
        //quite simple ,won't show the detailed pseudocode
}
```

3) skew heap(leftist heap)

Since these two heaps are rather similar, we use skew heap as an example.

```
typedef struct SkewHeapNode* pSkewNode;
struct SkewHeapNode
{
    int key;
    int id;
    pSkewNode left,right;
    // int npl;
};
```

```
Node SkewDeleteMin(Node root){
    Node left=root->left, right=root->right;
    free(root);
    SkewMergeHeap1(left,right);
}
Node SkewMergeHeap1( Node a, Node b){
    if(a ==NULL) return b;
    if(b==NULL) return a;
    if(a->key<b->key) return SkewHeapMerge(a,b);
    else SkewHeapMerge(b,a);
}
Node SkewMergeHeap(Node a,Node b){
    if(a->son ==NULL) a->son=b;
    else {
        Node new=SkewMergeHeap1(a->right, b);
        a->right=new
        swap the children;
    }
    return a;
}
```

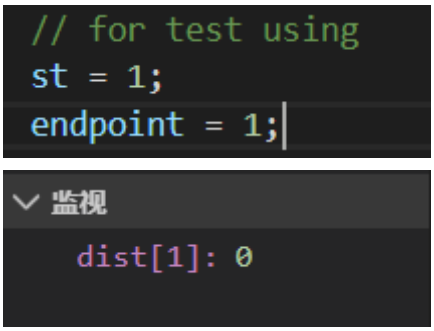
Chapter 3: Testing Results

1.Integrety testing

To test the correctness of our program, we use the testing data as followed, which is stored in “test.txt”.

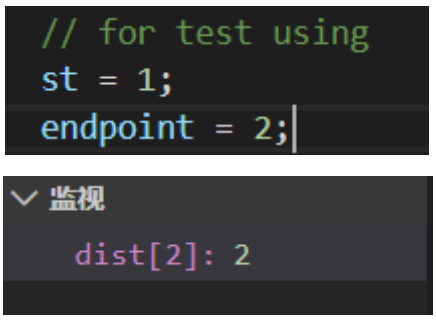
```
7 12
1 4 1
1 2 2
2 4 3
3 1 4
2 5 10
4 5 2
4 3 2
3 6 5
4 6 8
4 7 4
5 7 6
7 6 1
```

Case1: v1->v1 Test the distance to itself. **Expected answer:** 0



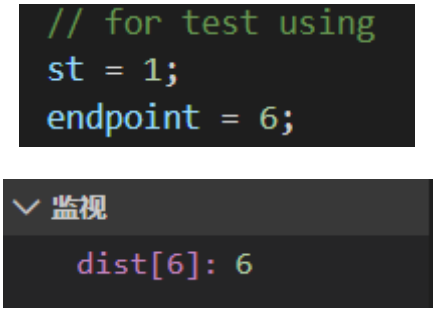
Result: correct.

Case2: v1->v2 Test a pair of adjacent vertices. **Expected answer:** 2 Path: 1->2



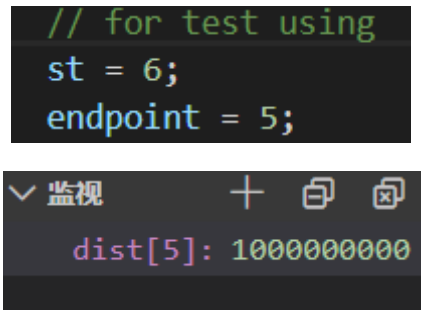
Result: correct.

Case3: v1->v6 Test the distance update when several paths exist. **Expected answer:** 6
Path: 1->4->7->6



Result: correct.

Case4: v6->v5 Test nonexistent path. **Expected answer:** INFDIST (1E9)
Path: nonexistent



Result: correct.

Conclusion: The logic and complement of our program works correctly.

2.Time testing

The testing data comes from *http://users.diag.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.NY.gr.gz*, **the data is in the dataNY.txt in the code folder.**

The testing result are listed as follows:

Time_cost(s)	50	100	200	300	500	1000	1500	2000
Basic Heap	1	4	8	13	20	47	71	96
Fibonacci Heap	2	5	12	19	29	68	109	141
Skew Heap	2	5	12	16	25	55	83	117
Leftist Heap	2	6	12	19	27	56	88	113

3.Map Comparison

We also measure the performance among different version of Dijkstra using different maps in order to show universality. The testing data, beside the NY group, comes from the following download links

http://users.diag.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.COL.gr.gz

http://users.diag.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.FLA.gr.gz

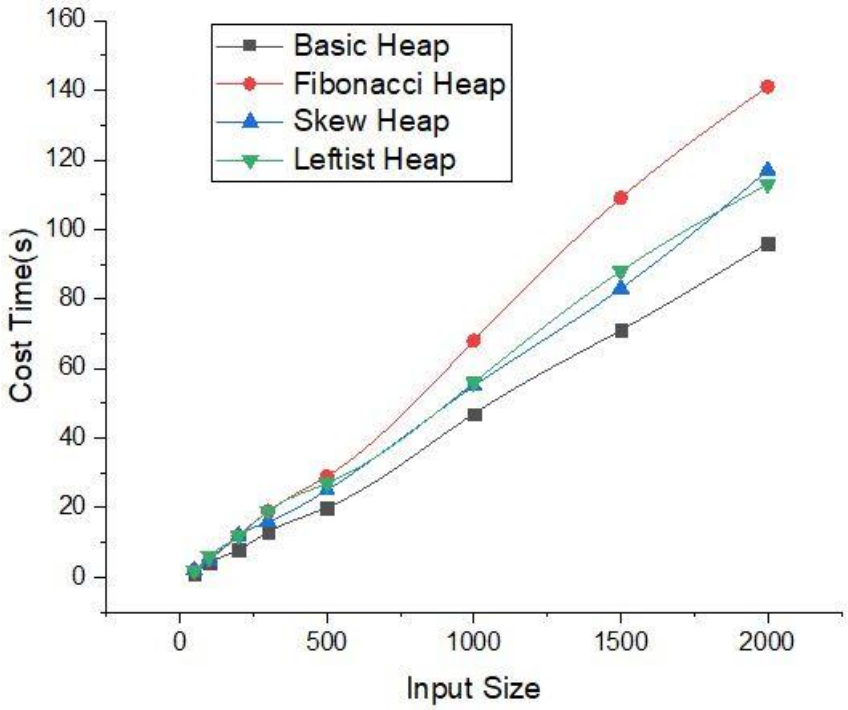
http://users.diag.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.NW.gr.gz

http://users.diag.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.NE.gr.gz

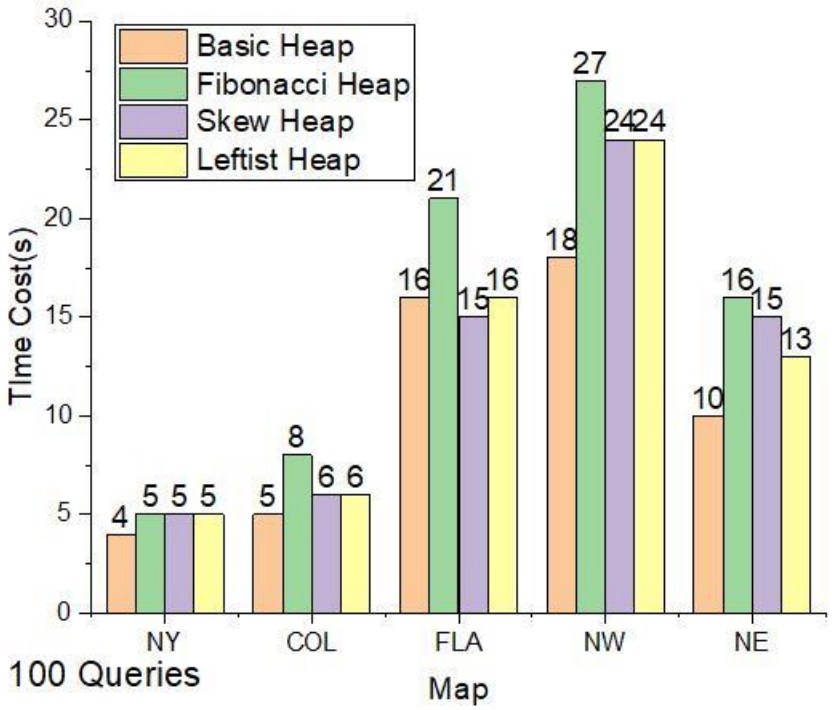
Time cost(s)	Basic Heap	Fibonacci Heap	Skew Heap	Leftist Heap
NY	4	5	5	5
COL	5	8	6	6
FLA	16	21	15	16
NW	18	27	24	24
NE	10	16	15	13

Chapter 4: Analysis and Comments

1. Time testing diagram



2. Map comparison diagram



Analysis

1.Space Complexity

Since the graph is sparse and enormous size of Vertices included, we use

adjacency list to store the Edge relation, which means the O(E) complexity. In Dijkstra algorithm, we use array to record the distance to each Vertice as well as keeping heap property, which costs O(V).

In all, the space complexity is O(E+V).

2.Time Complexity

For Dijkstra algorithm using different types of Heap:

1)Binary Heap

For each Vertice, insertion and deletion are operated once, which costs O(V*lgV) for V nodes. And for each edge, the comparision and update takes O(E*lgV) in total. In all, the time complexity is O((V+E)*lgV).

2)Fibonacci Heap

For V nodes, insertion costs O(1) and deletion costs O(lgV), which add up to O(V*lgV). For E edges, the update cost O(E) in total using cascading cut. In all, the time complexity is O(V*logV+E).

3)Leftist Heap

For V nodes, insertion and deletion costs O(lgV) by merging, summing up to O(V*lgV). For E edges the update cost O(E*lgV) using merging. So the time complexity is O((V+E)*lgV).

4)Skew Heap

Similar to Leftist Heap, the time complexity is O((V+E)*lgV).

Comments

According to the actual time cost, Fibonacci Heap takes more time than other structures, while the analysis shows it's supposed to be the fastest one. Here are the possible reasons:

1) The constant time cost of Fibonacci Heap is large in fact, like insertion operation;

2) The function calls are more frequetly, which cost more time;

3) Complicated pointer works and large amout of memory visit request;

4) Although the deletion operation takes O(lgV) for all structures, Fibonacci Heap takes much longer time. And for Dijkstra algorithm, the deletion operation are called frequently.

Appendix: Source Code (in C)

Located in the folder named **code**.

Declaration

We hereby declare that all the work done in this project titled " Shortest Path Algorithm with Heaps " is of our independent effort as a group.

Duty Assignments:

Programmer: Zhang Tongchen（张童晨）

Tester：Wang Ziteng（王子腾）

Report Writer：Li Xiang（李想）