Shortest Path Algorithm with Heaps

group 8: 颜晗 吴俊贤 钱于飞

Abstract: Firstly, we create two type of heaps, binary heap and Fibonacci heap. We have learned binary heap in class, but we need to learn Fibonacci heap by ourselves. After we know how to build heap, we use them to implement Dijkstra algorithm and test it. We use the USA road networks as our data source. Finally we analyzed the tested result and got the conclusion in this project.

1. Description of the project

In this project, we need to implement at least two heaps, then use them to implement the Dijkstra algorithm. After that we are demand to test them using the USA road networks. We should evaluate the performance of the heaps we use, and also find the relationship between run time and graph sizes.

2. Introduction to Fibonacci Heap

The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a "mergeable heap". Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

A Fibonacci heap is a collection of rooted trees that are min-heap ordered. Each node x has a pointer pointing its parent, also a pointer to any one of its child which we call the first child. The children of one node are linked together in a circular ,double linked list, using left and right pointer. If a node has only one child, then the left and right pointer point to itself. Last, we can see the root pointer points the minimum node in the root list which is also the minimum in the whole heap.

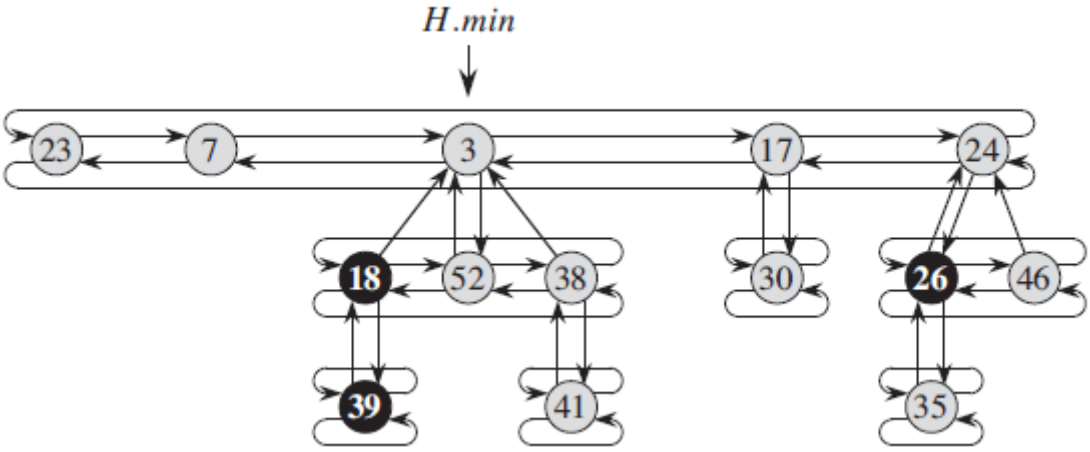

Figure 2.1 Fibonacci heap

```
//the node structure of Fibonacci heap
typedef struct Fibonacci_Node{
    int value;
    int degree;  //number of the children
    int mark;    //the flag of whether the children was
deleted
    struct Fibonacci_Node* left;
    struct Fibonacci_Node* right;
    struct Fibonacci_Node* child; //node of the first child
    struct Fibonacci_Node* parent; //node of the parent
}FibonacciNode;
typedef FibonacciNode* PtrFibonacciNode;
//the complete heap
typedef struct Fibonacci_Heap{
    int number; //the number of the nodes
    int maxdegree;
    PtrFibonacciNode min; //the pointer of the minimum
    PtrFibonacciNode *tmp; //tempary heap
}FibonacciHeap;
typedef FibonacciHeap* PtrFibonacciHeap;

```

Here are some operations of Fibonacci heap.

Insertion:

```
void InsertFibonacciHeap(MyHeap,value)
{
    malloc the space to the node;
    node->degree = node->mark = 0;
    node->value = value;
    node->child = node->parent = NULL;
    If MyHeap->min == NULL

```

```
        Create a root list for MyHeap containing just node;
        MyHeap->min = node;
    else insert node into MyHeap's root list
        If MyHeap->min->value > node->value
             update the min;
    MyHeap->number++;
}

```

It's a simple operation. When we want to insert a node to the heap,just insert the node to the root node linked list. If the inserted node is smaller than the old min node,we can update the min node as the inserted node.

Deletemin:

```
int DeleteminFibonacciHeap(MyHeap)
{
    z = MyHeap->min;
    if z != NULL
       for each child x of z
          add x to the root list of MyHeap;
          x->parent = NULL;
       remove z from the root list of MyHeap;
       If z->right == z
          MyHeap->min = NULL;
       Else MyHeap->min = z->right;
          consolidate(MyHeap); //details below
       MyHeap->number--;
    return z->value;
}
```

To begin with,we can remove the min node from the root node linked list,and add both the truncated child node and the child's brother to the root node linked list. In the second place,we remove the minimum heaps from the Fibonacci heap and add them to the temporary heap in turn. In addition we should merge the minimum heaps in the temporary heap with the same degree until the heap has no same degree minimum heap. Finally,we assign the temporary heap to the Fibonacci heap and update the min node.

Consolidate:

```
void ConsolidateFibonacciHeap(MyHeap)
{
    let MyHeap->tmp[0,NewMaxDegree] be a new array;
    for i = 0 to NewMaxDegree
        MyHeap->tmp[i] = NULL;
```

```
    for each node w in the root linked list of MyHeap
        x = w;
        d = x->degree;
        while MyHeap->tmp[d] != NULL
             y = MyHeap->tmp[d]; //another node with the same degree as x
             if x->value > y->value
                exchange x with y;
             Link(MyHeap,y,x);
             MyHeap->tmp[d] = NULL;
             d++;
        MyHeap->tmp[d] = x;
    MyHeap->min = NULL;
    for i = 0 to NewMaxDegree
       if MyHeap->tmp[i] != NULL
       create a root linked list for MyHeap containing just MyHeap->tmp[i];
             MyHeap->min = MyHeap->tmp[i];
       else insert MyHeap->tmp[i] into MyHeap's root linked list;
            if MyHeap->tmp[i]->value < MyHeap->min->value
                 MyHeap->min = MyHeap->tmp[i];
}
```

In this step,in which we reduce the number of trees in the Fibonacci heap,is consolidating the root linked list of MyHeap,which the call Consolidate(MyHeap) accomplishes. Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value:

1. Find two roots x and y in the root list with the same degree. Without loss of generality,let x->value ≤ y->value.
2. Link y to x:remove y from the root list,and make y a child of x by calling the Link procedure. This procedure increments the attribute x->degree and clears the mark on y.

The procedure Consolidate uses an auxiliary array tmp[0, NewMaxDegree] to keep track of roots according to their degrees.If tmp[i] = y,then y is currently a root with y->degree = i,which is useful to implement the corresponding operations.

Finally,we also need to copy the tmp to the MyHeap and update the min node.

3. The theoretical comparison

operation	Binary heap(worst)	Fibonacci heap(amortized)
Make - Heap	O(1)	O(1)
INSERT	O($logN$)	O(1)
MINIMUM	O(1)	O(1)
EXTRACT-MIN	O($logN$)	O($logN$)
UNION	O(N)	O(1)
DECREASE-KEY	O($logN$)	O(1)
DELETE	O($logN$)	O($logN$)

4. Other details

4.1 the implement of graph

```
// the data struct of graph
typedef struct Node *PtrToNode;
struct Node{            //As all the point has a index in array, we don't need to show which node the edge belong to.
    int Vertex;         //the point connect
    int Distance;       //the distance between them
    PtrToNode Next;     //the next edge
};

typedef struct Vnode{
    PtrToNode FirstEdge;
} AdjList[MaxVertex];   // The maximun graph size.

typedef struct GNode *PtrToGNode;
struct GNode{
int Nv;
int Ne;
AdjList G;
};
typedef PtrToGNode Graph;

```

4.2 the implement of Dijkstra

Here is the pseudo code of Dijkstra.

```
 int Dijkstra(Graph G , int v1 , int v2)
Mark the distance of every node to be infinity, mark every node unknown
    Mark the distance of v1 to be 0, mark it known and add it to heap
    While heap is not empty
       Deletemin from the heap
        If the deleted node is v2, return the distance of v2
       Go through all vertices that has an edge connected to the deleted node
            If the vertex is unknown, and that passing the deleted node can reduce the distance of this vertex, reduce the distance and add this node to heap
```

We have learned Dijkstra last term. Here what we need to do is just using the heap to find the minimum distance in one loop.

5. Test and result

The table below shows the average time taken to find the shortest path between 1,000 pairs for each road network. It seems that we cannot get the relationship between time and size directly from the data. But as we can see from the Figure 5.1 below, they are roughly linear.

network data	binary time(AVG)	Fibonacci time(AVG)
USA-road-d. NY	435.25(s)	82.35(s)
USA-road-d. COL	575.30(s)	124.70(s)
USA-road-d. FLA	1768.88(s)	330.31(s)
USA-road-d. NE	3902.42(s)	498.62(s)

network data	vertex number	edge number
USA-road-d. NY	264346	733846
USA-road-d. COL	435666	1057066
USA-road-d. FLA	1070376	2712798
USA-road-d. NE	1524453	3897636

The data in the table above were tested in a completely random situation. We also test in a special case: The source comes from 1~50000, and the sink comes from a network's last tens of thousands of points. The result is below.

	NY	COL	FLA	NE
binary	194.13(s)	423.58(s)	1302.40(s)	6004.16(s)
Fibonacci	44.60(s)	94.09(s)	224.01(s)	629.54(s)

As we can see, all the data are quite different from those in the above table. The first three are less and the NE is more. We can know that the networks may be many parts and those points whose index are far apart are just connected by some pairs. It may be a trick when create these data though they claim that these are real USA map.

How we get these test queries: Use the excel function RANDBTWEEN(), we can get the test data.

Below is a diagram of the running times vs. graph sizes.

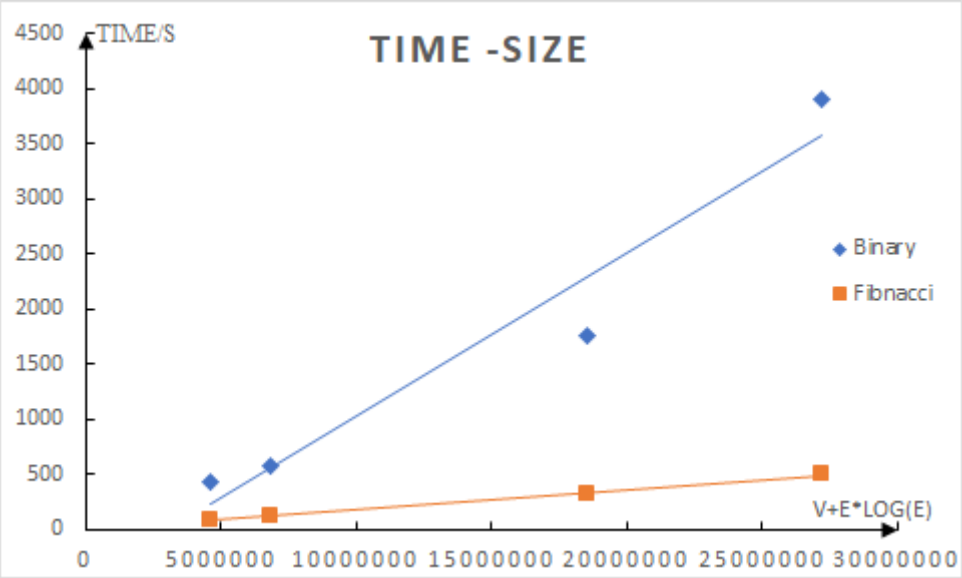

Figure 5.1

The Y-axis represents time, and the X-axis represents road network size which is replaced by $V + E * LOG(E)$. As the graph size become larger, the run time is larger. And the time is roughly linear with size, Fibonacci heap shows a better match.

As we can see, the time cost of Fibonacci heap is much less than binary heap. It may not be because Fibonacci heap is good, but because we implement the binary heap in a bad way.

As we can see below , after we delete the root(first) element, we need to adjust all the node in the heap two times which cost a lot of time. Fibonacci heap looks complex but in fact need little time when adjust, also the time cost is stable.

```
//the pseudo code of deletmin of binary heap
    Delete the first element
    Move all elements left one block
    Percolate up from the last element, if an element is
smaller than its father, switch them
```

6. Conclusion

The Fibonacci heap has a better performance in our test. And it also has a stable time cost when we run our program. Maybe we could implement more heap and implement them in a smarter way so that we could analyze this problem better.