Advanced Data structures

Laboratory Project 3
Shortest Path Algorithm with Heaps

Date：2019-3-30

}

Chapter1 Introduction

It is an existing phenomenon that shortest path problems, which are ones of the most fundamental combinatorial optimization problems with many applications, have been researched and discussed since years ago. In the last semester, we have learnt a few shortest path algorithms using graphs. To be frank, it was a little inefficient. However, now we have learnt priority queue in this semester, which could be used to realize the Dijkstra's shortest path algorithm in a relatively much more efficient and effective way.

In this project, we use Fibonacci heap and heap to implement our algorithm, using American city maps as examples. And we compare them with each other.

Chapter2 Algorithm Specification

2.1 class definition

First, we define a class to represent the graph as a map. We also define a data structure to represent the Fibonacci heap, which has 4 pointer members, which points to its left sibling, right sibling, its child and parent respectively. It also has an integer member as its degree and a pair element to represent the value of the node.

The related code is listed as follows:

```
  class HeapGraph
  {
  public:
      HeapGraph(int Vertex, int edges); // Constructor Function
      void Addedge(int start, int end, int distance);//Add a New edge into the Graph
      void ShortestPath(int source); // Get the minimum distance of each node
      vector<int> distance; // Initialize each vertex distance
  private:
      int Vertex; // the value of vertex
      int source; // the origin of the graph
      int edges; // the number of edges
      list<pair<int, int> >*adj_List; // The adj_List save the distance and end
  };
struct FibNode
{
        FibNode* left; // left sibling
        FibNode* right; // right sibling
        FibNode* parent;
        FibNode* child;
        pair<int, int> element; // Value of the node
        int degree; // Degree of the node
};
```

2.2 priority queue based on heap

We use priority_queue, heap and vector as several tools to realize the Dijkstra algorithm. *priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > >Heap;* The origin priority queue looks like this. Then using Dijkstra algorithm that we have learnt, we may solve the problem. We push the pairs, and then loop until all the vertex have been labeled to shortest path to each node. The algorithm is implemented according to the increasing order of the lengths of the generated paths.

The related code is listed as follows:

```
Heap.push(make_pair(0, source)); // Push into the array
distance[source] = 0;

while (!Heap.empty()) // Loop until all the vertex have been labeled
{
    // Heap.top() returns pair<int,int>
    // The first int refers to the vertex with the minimum distance
    // The second int refers to the distance
    int start = Heap.top().second;
    Heap.pop();

    for (auto i : adj_List[start])
    {
        int end = i.first; // end refers to the another vertex
        int dist = i.second; // dist refers to the old distance

        if (distance[end] > distance[start] + dist)
        {
            distance[end] = distance[start] + dist;
            Heap.push(make_pair(distance[end], end));
        }
    }
}
```

2.3 the Fibonacci heap part

The Fibonacci heap is also a collection of binomial trees, but it is flatter than min-priority queue. A Fibonacci node has 4 pointer members, which points to its left sibling, right sibling, its child and parent respectively. It also has an integer member as its degree and a pair element to represent the value of the node. Then we use it to define a “Fib graph” class. It has many functions to maintain the Fibonacci heap. However, those are like the ones in the min-priority queue which are to main the binomial trees in it. The specific operations including “link”, “consolidate” are illustrated in the class. Using Fibonacci heap implement the Dijkstra algorithm is like using min-priority queue. Push the pair into the array and then loop until all the vertex have been labeled.

The related code is listed as follows:

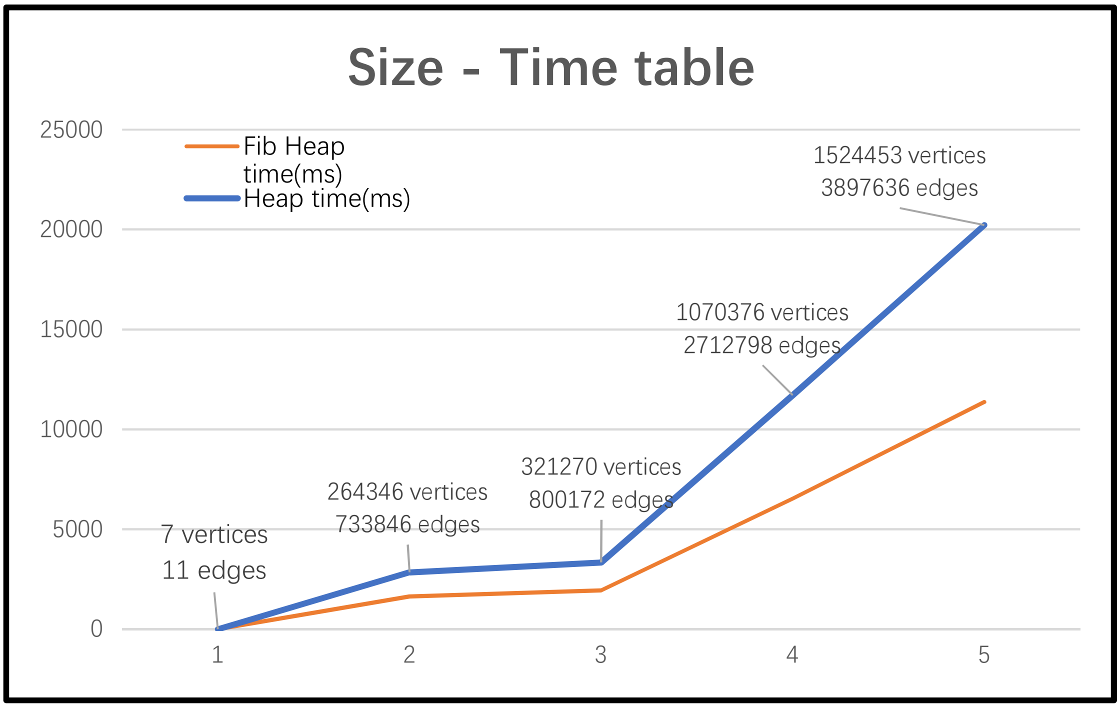
```
class FibGraph
{
public:
	FibGraph(int Vertex, int edges); // Initialize the newheap
	void Push(pair<int, int>element); // Push a new node into the heap
	pair<int, int> Pop(void); // Pop the minimum node from the heap
	vector<int>distance; // save the distance of each node;
	void Addedge(int start, int end, int distance); //Add a new edge into the graph
	void ShortestPath(int source); // Get the minimum distance of each node
private:
	int Vertex; // the number of vertex in the graph
	int edges; // the number of edges in the graph
	struct FibNode* Minnode = NULL; // the FibNode pointer to the mininum number
	int numbernode; // the number of nodes in the graph
	list<pair<int, int>>*adj_List; // the adjacent Lis
	void Link(FibNode* Anode, FibNode* PNode); // Link the two Node
	void Consolidate(void); // merge the pieces after pop
};

void FibGraph::ShortestPath(int source)
{

	Push(make_pair(source, 0)); // Push into the array
	distance[source] = 0;

	while (numbernode != 0) // Loop until all the vertex have been labeled
	{
		// The first int refers to the vertex with the minimum distance
		// The second int refers to the distance
		auto element = Pop();
		int start = element.first;

		for (auto i : adj_List[start])
		{
			int end = i.first; // end refers to the another vertex
			int dist = i.second; // dist refers to the old distance

			if (distance[end] > distance[start] + dist)
			{
				distance[end] = distance[start] + dist;
```

```
                Push(make_pair(end, distance[end]));
            }
        }
    }
}
```

Chapter3 Testing Results

3.1 test results

Size - times table					
vertices	7	264346	321270	1070376	1524453
edges	11	733846	800172	2712798	3897636
Heap time(ms)	0.0118	2844	3340	11711	20222
Fib Heap time(ms)	0.0023	1641	1932	6530	11368



3.2 Check

The program is logically sensible and the results of all the tests prove to be correct. All test cases reinforce the fact that the program solves the problem properly.

Chapter4 Analysis and Comments

The Fibonacci heap is a collection of binomial trees in computer science. It has better analytic performance than the binomial heap and can be used to implement merge priority queues. An operation that does not involve deleting an element has a flat time of O(1). The number of extractions - reductions and deletions is much less efficient than others. The dense map requires only O(1) of equalization time for each key reduction, which is a huge improvement compared to the O(lg n) of the binomial.

The search time in the Fibonacci heap is O(1), but the setup takes O(N) time, so the time complexity of the experimental Fibonacci heap is O(N).The time complexity of the smallest heap is O(NlgN).

Declaration: We hereby declare that all the work done in the project titled "Performance Measurement" is of our independent effort as a group.

Group assignment:

Programmer:王钟毓

Tester:彭子帆

Document writer:陈宇威