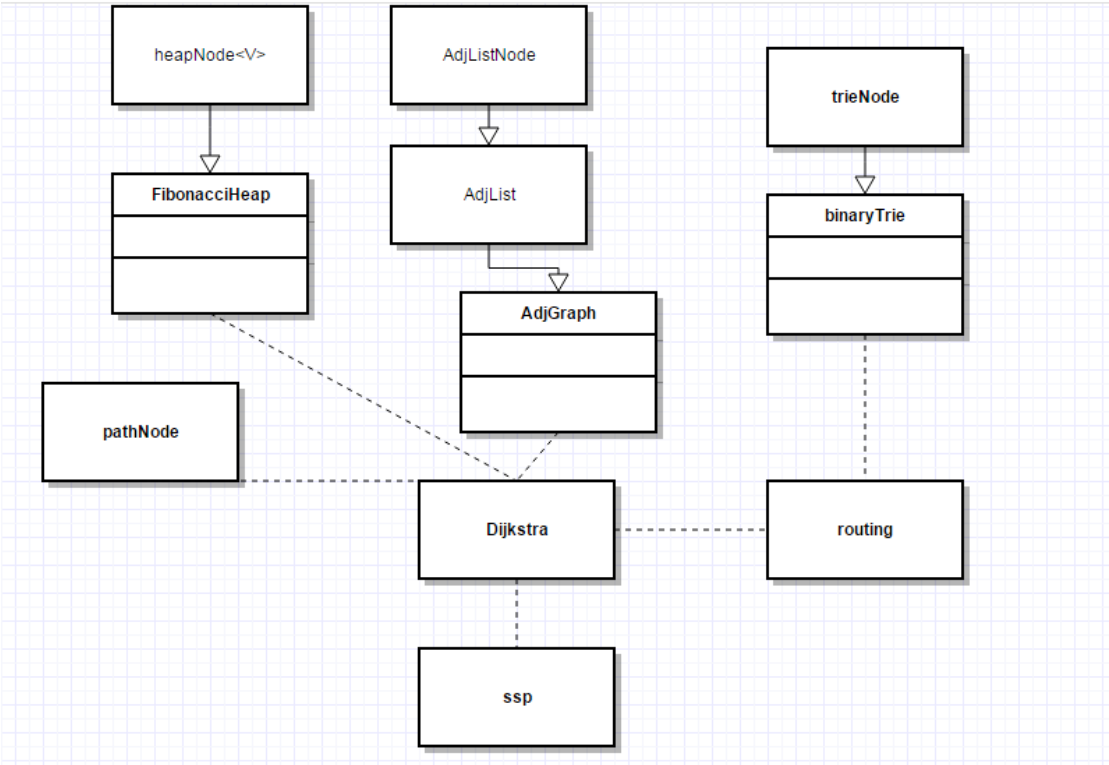
Report

1. Compiler: gcc, version: 4.8.2, support c++ 11.

2. Usage:
   >> make ssp (to make the ssp execution, which the default one)
   >> make routing (to make the routing execution)

   The produced execution will locate in the ./bin folder.
   Also, the .txt files should be put in the ./bin folder.
   >> cd bin
   >>./ssp filename src_node_# dst_node_#
   >>./routing graphfilename ipfilename src_node_# dst_node_#

3. Program Structure:



   The above diagram is not a real UML diagram. It is just to show the structure of the program. As the diagram shows, I define three nodes strcutre. One is heap node, one is adjacent list node, the other is path node, which is to store the route path.
   There are three data structures, one is Fibonacci heap, one is Adjacent graph. The Dijkstra algorithm will use these two data structures. The routing function which represents the router scheme, will invoke the Dijkstra function and use the binary trie.

4. Function Prototypes:

Fibonacci.h

```
//The structure represents the heap node;
template <class V>
struct heapNode{
//Member Functions
	// constructor
	heapNode();
	//copy constructor
	heapNode(heapNode& newNode);
	//operators overloading
	heapNode<V> &operator = (const heapNode &nodeB);
	// get previous node
	heapNode<V>* getPrev();
	// get next node
	heapNode<V>* getNext();
	// get child node
	heapNode<V>* getChild();
	// get parent node
	heapNode<V>* getParent();
	// get value
	V getValue();
	// if there is child cut
	bool ischildCut();
	// if has children
	bool hasChildren();
	// if has parent
	bool hasParent();
};
template <class V>
class FibonacciHeap{
public:
	// constructor
	FibonacciHeap();
	// insert a node
	heapNode<V>* insert(V);
	// merge two Fibonacci heaps
	void merge(FibonacciHeap&);
	// if the heap is empty
	bool isEmpty();
	// get the minimum node
	V getMinimum();
	// remove the minimum node
	V removeMinimum();
	// decrease key
	void decreaseKey(heapNode<V>* ,V );
```

```
        //search a node with specific value
        heapNode<V>* find(V value);
};
```

Adjgraph.h

```
//The structure to represent an adjacency list node
struct AdjListNode{
        //constructor
        AdjListNode(int d,int w);
        AdjListNode(int d, int w, AdjListNode *tmpN);
};

//The structure to represent an adjacency list
struct AdjList{
        AdjListNode *head;    // pointer to head of list
};

//The class represents the adjcent Graph
class AdjGraph{
//Member Functions
        //constructor
        AdjGraph(int n=0);

        // Please notice that it can only be used for undirected graph
        void addEdge(int src, int dest, int w);
        //The function to print the adjacenncy list representation of graph
        void printGraph();
       };
```

Dijkstra.h

```
class pathNode{
        //constructor
        pathNode();
        pathNode(int w, int v);

        //operators overloading
        pathNode &operator = (const pathNode &nodeB);
        bool operator < (pathNode nodeB);
        bool operator > (pathNode nodeB);
        bool operator <= (pathNode nodeB);
        bool operator >= (pathNode nodeB);
        bool operator == (pathNode nodeB);
```

```
};

//The function implement the single source all destination algorithm (Dijkstra's algorithm).
// the prev[] stores the previous node from destination node to source node
// the dist[] stores the distance from destination node to source node
void ShortestPaths(AdjGraph *graph, int v, int *prev,int *dist);

// This function transform the prev node representation of the path to next hop representation
of the path,
// which is needed by the router scheme.
// the nextHop[] stores the nextHop from source node to destination node
// the prev[] stores the previous node from destination node to source node
void getNextHop(int v, int numOfVex, int *prev, int *nextHop);
```

binartTrie.h

```
//The struct to represent the trie node;
class trieNode{
	trieNode(bool e = false,int b=-1,string v="",int nh=-1,trieNode *p=NULL,trieNode
*l=NULL,trieNode *r=NULL);
	~trieNode(){
	}
};


class binaryTrie{
	//constructor
	binaryTrie();
	//insert a node with a string (ip address) and a int (nextHop)
	bool insert(string,int);
	//remove a node with specific string (ip address)
	bool remove(string);
	//search a node with specific string (ip address)
	trieNode *search(string);
	//forward cleanup the nodes with less than two children
	bool fwdCleanup();
	//post order traversal to clean up the nodes with less than two children
	bool npClean();
	//to print the whole trie
	void print();
	//destructor
	~binaryTrie(){
	}
```

```
};
```