

# ADS Project

GARIMA SINGH

UFID: 5197-5877

## 1 PROJECT OBJECTIVE

---

To implement Dijkstra's algorithm (single source shortest path) using Fibonacci Heap and undirected graph using adjacency list and to use this to implement routing by longest prefix matching using binary tries.

## 2 PROJECT ENVIRONMENT

---

### 2.1 LANGUAGE: C++

### 2.2 IDE: DEVELOPED IN VISUAL STUDIO 2013

### 2.3 COMPILER USED IN LINUX: G++ WITH `-std=c++0x` FOR C++ 11 VERSION

### 2.4 INSTRUCTIONS TO RUN:

The 2 executables generated are ssp (for part1) and routing (for part2)

#### 2.4.1 For part1:

Command would be:

```
$/ssp graph_file src_node dest_node
```

And output is in the form:

total\_weight

src\_node node\_1 ... dest\_node

#### 2.4.2 For part2:

Command would be:

```
$/routing graph_file ip_file src_node dest_node
```

And output is in the form:

total\_weight

src\_prefix\_match node\_1\_prefix\_match ... node\_k\_prefix\_match

## 3 FUNCTION PROTOTYPES

---

### 3.1 FIBONACCIHEAP.H:

Class FibHeapNode

```
{
private:
FibHeapNode(int v, int c):nodeKey(c),nodeLabel(v),degree(0),mark(false),child(NULL),parent(NULL);
// Inserts other node after this node
    void insert(FibHeapNode* other);
// Remove this node from sibling list
    void remove();
// Add a child
    void addChild(FibHeapNode* other);
// Remove Child
    void removeChild(FibHeapNode* other);
public:
int key();
int data();
}

class FibonacciHeap {
protected:
// Insert a node into the HeapNode
    fNode insertNode(fNode newNode);
public:
FibonacciHeap():rootWithMinKey(NULL), count(0), maxDegree(0);

bool empty();

void removeMin();

void decreaseKey();
}
```

### 3.2 DIJKSTRA.CPP

```
void dijkstra(Graph * g,int srcNodeLabel,int destNodeLabel);
```

### 3.3 ROUTING.CPP

```
class BinTrie
{
public:

    BinTrie(bool isleaf);

    BinTrie(int d,bool isleaf);

}

//returns pointer to trie for ownerVertex
BinTrie * constructBinTrie(int ownerVertex, vector<int> prev);

//traverse final trie based on prefix matching of dest node IP and return next hop
int traverseTrie(int destNode,BinTrie * trie);

//modification of post order traversal to condense bin trie
void postOrder(BinTrie * root);

//delete subtries who have same next hop by calling postorder
void condenseBinTrie( BinTrie * trie);

//variation of dijkstra which returns previous array
vector<int> dijkstra(Graph * g,int srcNode,int destNode);

//to route the packet from source to destination(traverse tries of vertices on the way)
void route(int srcNode, int destNode, map<int, BinTrie *> binTrieMap)
```

## 4 PROGRAM STRUCTURE AND FLOW

---

### 4.1 PART 1: SSP

For first part we needed to implement a Fibonacci Heap and an undirected graph with adjacency list for each vertex.

The Fibonacci Heap class and its operations were implemented using the algorithm as given in Introduction to Algorithms by Cormen (et all.) and as discussed here:

<http://www.cse.yorku.ca/~aaw/Jason/FibonacciHeapAlgorithm.html>

Input command: `./ssp graph_file src_node dest_node`

From the input file graph\_file we read number of vertices and edges and information of each edge: end points and edge weight. This is used to populate the graph object which is a map of vertex as key and its adjacency list as value:

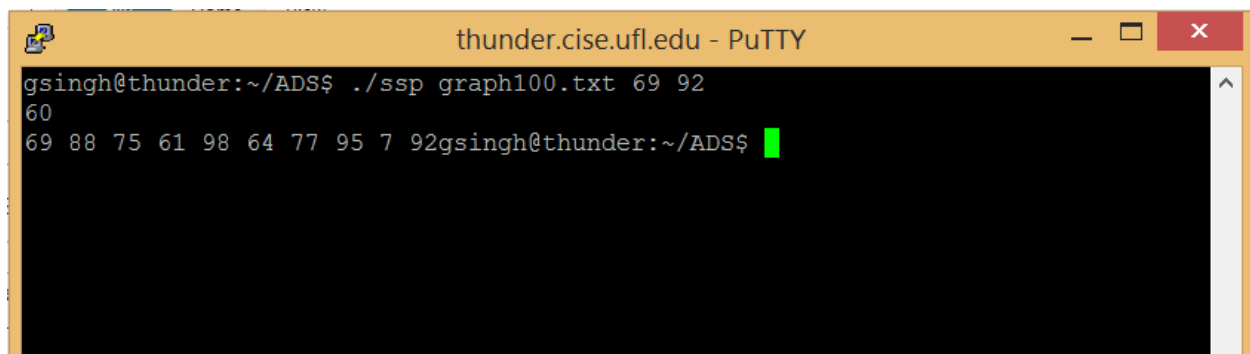
```
map<int, map<int, int> > vertices;
```

I then invoke the Dijkstra's algorithm by calling passing the graph object along with source and destination

```
void dijkstra(Graph * g,int srcNodeLabel,int destNodeLabel)
```

In the algorithm I populate 2 arrays distFromSource[] and prev[]. distFromSource stores distance of source vertex from all other vertices. So distFromSource[destNode] is the value of interest to us in the first part. prev[] stores the prev vertex to any vertex in the shortest path from source to that particular vertex. This is useful in printing the shortest path from source to destination.

At the beginning of the algorithm I initialize the distFromSource[] of vertices in the adjacency list of the source vertex with the edge weight and prev[] of these vertices with the source. For all other vertices , distFromSource[] stores a very large value(2000, since all weights are lesser than this) and prev with NULL. These values are then inserted into the Fibonacci heap and rest of the algorithms is performed removing minimum and decreasing key accordingly.



The screenshot shows a PuTTY terminal window titled "thunder.cise.ufl.edu - PuTTY". The terminal displays the following command and output:

```
gsingh@thunder:~/ADS$ ./ssp graph100.txt 69 92
60
69 88 75 61 98 64 77 95 7 92gsingh@thunder:~/ADS$
```

## 4.2 PART 2: ROUTING

In the second part we were given IP addresses of each vertex along with edge weight information. We needed to implement routing using Dijkstra's to find shortest path and by building a binary trie at each vertex (using binary representation of IP ) and then condensing it by longest prefix matching (by doing a postOrder traversal) .

Input command: `$/routing graph_file ip_file src_node dest_node`

The graph\_file is used to populate the graph object as before. The ip\_file contains a list of IPs for each vertex. This is read and used to populate a 2-d array:

```
int ** vertexToIPMap;
```

I then call the `vector<int> dijkstra(Graph * g,int srcNode,int destNode)` which returns the `prev[]` which is used to construct a binary trie. In `BinTrie * constructBinTrie(int ownerVertex, vector<int> prev)` we use the binary representation of the IP of all vertices other than the owner vertex and construct a binary trie for the owner vertex using the `prev[]` returned after running Dijkstra's algorithm for owner vertex, since this will tell us the next hop router in the shortest path from source vertex to all other vertices. A binary trie is essentially a binary tree for us with the exception that data is stored only in the leaf nodes and the internal nodes are used for prefix matching. We take a left if the ith bit of the IP is 0 or else we take a right and then finally in the leaf node we store the next hop router for this vertex.

I repeat this process for all vertices in the graph. The next step is to condense the binary tries.

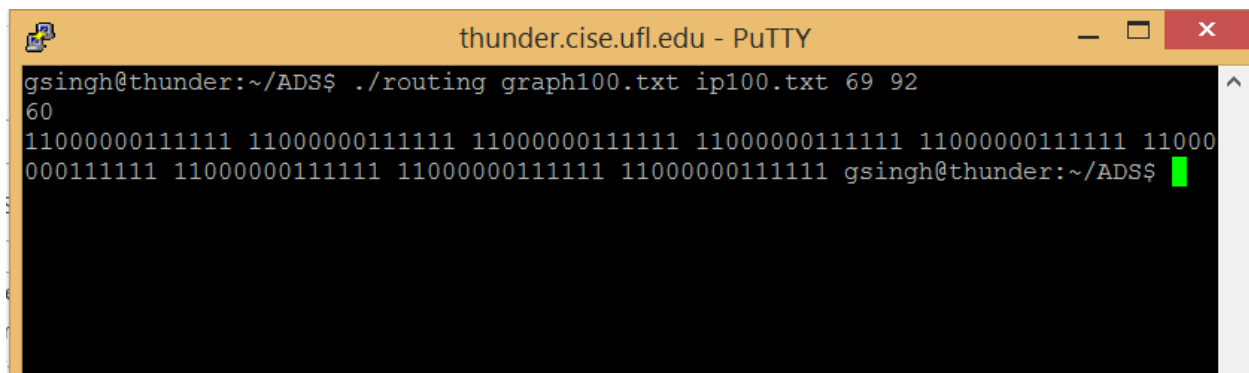
```
void condenseBinTrie( BinTrie * trie)
```

Condense means that if 2 children have the same value stored in them, the leaves can be condensed into their parent and the parent then becomes the new leaf. Also if there is only a single leaf, it too can be condensed into its parent and the parent can then become the new leaf. This is the idea of longest prefix matching and can be accomplished by postOrder traversal and performing the above checks for every node. Condensing the tree reduces the time required to traverse the binary trie to find the next hop router.

Finally I route the packet from the source to the destination in `void route(int srcNode, int destNode, map<int, BinTrie *> binTrieMap)`. We start with the source vertex and traverse its binary trie using the IP of the destination node and find the next hop router stored in the leaf. We then traverse the binary trie of this next hop again using the IP of the destination vertex in the function:

```
int traverseTrie(int destNode,BinTrie * trie)
```

We repeat this until the next hop is the destination router itself. This terminates the process.



```
thunder.cise.ufl.edu - PuTTY
gsingh@thunder:~/ADS$ ./routing graph100.txt ip100.txt 69 92
60
1100000011111 1100000011111 1100000011111 1100000011111 1100000011111 11000
00111111 1100000011111 1100000011111 1100000011111 gsingh@thunder:~/ADS$
```