COP5615 PROJECT ASSIGNMENT 2

Sanil Sinai Borkar
{*sanilborkar@ufl.edu*}

April 15, 2015

1 Dijkstra's Shortest Path Algorithm Using Fibonacci Heaps

1. To run this program, navigate to the folder and type make ssp
2. On the command line, type

$$./ssp < filename >< source\_vertex >< destination\_vertex >$$

1.1 CLASSES USED

- *Fibonacci Node Implementation: FibNode.h FibNode.cc*
  - Fibonacci Node structure

int vNumber	Vertex number
int data	Weight of the edge with this node as the vertex
int degree	Degree
Node* parent	Pointer to the parent
Node* child	Pointer to a child
Node* left	Pointer to the left sibling
Node* right	Pointer to the right sibling
bool isChildCut	Indicates child-cut status
$std :: vector < adjListNode* >$ adjList	Adjacency List
Node* prevNode	Previous node to this on the shortes path
BinTrie T	Binary trie object at this node

  - Node(int value, int weight)
    - Constructor to create a new Fibonacci node
    - Input: the key-value pair
    - Return Value: Pointer to the created node
  - void AddtoAdjList(Node* dest, int weight)
    - Adds a node to the adjacency list of the invoking node
    - Input: Pointer dest to the node to be inserted, the distance between the invoking node and dest
    - No return value

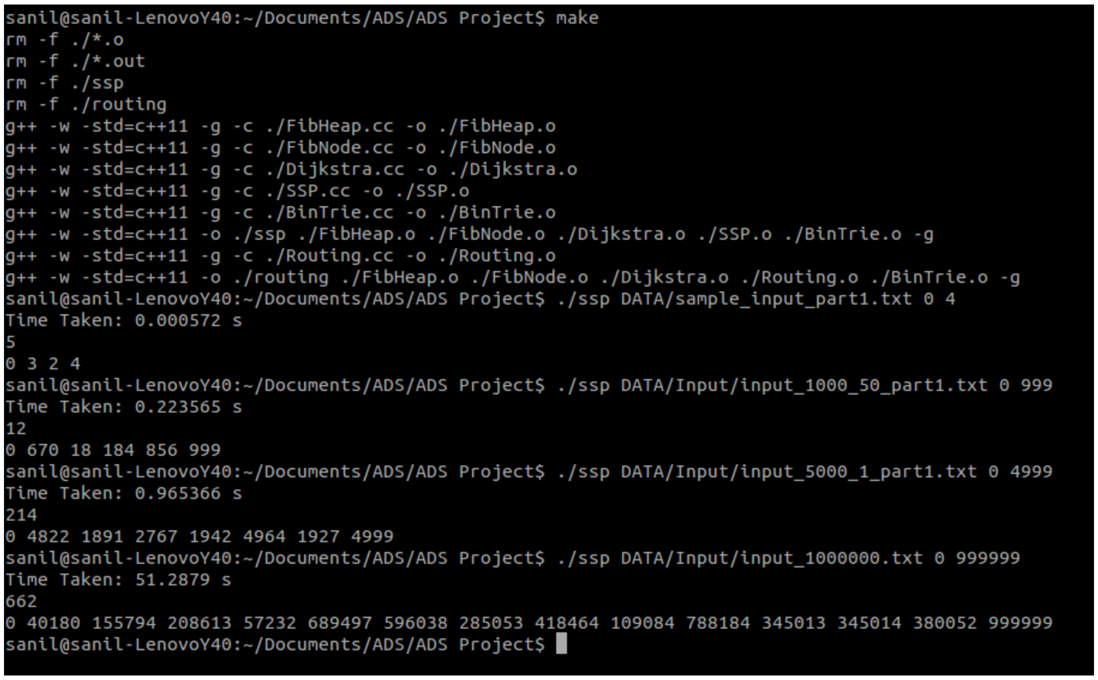
- bool AddAsChild(Node *n)
    - Adds the node pointed to by n as a child of the invoking node
    - Input: Pointer to the node that needs to be inserted as the child
    - Return Value: True, if successful; false, otherwise
- void RemoveFromHeap()
    - Removes the invoking node from the heap
    - No input
    - No return value

- *Fibonacci Heap Implementation: FibHeap.h FibHeap.h*
    - bool Insert (Node* ptr)
    - Inserts a node pointer to by ptr in the heap
    - Input: Pointer to the node to be inserted
    - Return Value: True, if successful; false, otherwise
    - void DecreaseKey (Node *ptr, int newValue)
        - Decreases the key value of the node pointed to by ptr
        - Input: Pointer to the node whose key is to decreased, the new key value
        - No return value
        - Node* RemoveMin ()
            - Removes the minimum node from the heap, and returns it
            - No input
            - Return Value: Pointer to the minimum node that was just removed
        - Node* Remove (Node* ptr)
            - Removes an arbritrary node pointed to by ptr
            - Input: Pointer to the node to be removed
            - Return Value: Pointer to the node that was just removed
        - void PairwiseCombine (Node *ptr, std::vector¡Node*¿ &vNodeDegrees)
            - Combine the individual trees of same degree in the heap in a pair-wise manner
            - Input: Pointer to the invoking node, a vector containing the degrees of all the individual trees present in the heap
            - No return value
        - void CascadingCut (Node *p)
            - Perform a cascading cut
            - Input: Pointer to the node on which cascading cut needs to be invoked
            - No return value
- *Dijkstra's Algorithm Implementation: Dijkstra.h Dijkstra.cc*
    - bool Initialize(char* filename, int srcVertex, int &V, int &E)
        - Initializes the graph, the vertices and the adjacency lists.

        * Input: Filename that contains the graph information, source vertex, references to integers that will hold the number of vertices and edges in the graph, respectively.
        * Return Value: True, if successful; false, otherwise

    – int FindSSP(int source, int dest, std::vector¡int¿ &verticesOnSP)
        * Find the shortest path using Djikstra's algorithm
        * Input: Source vertex, destination vertex, reference to a vector that will contain the vertices on the shortest path found
        * Return Value: The shortest path distance

- *SSP: ssp.cc*
    – Contains the *main()* function
    – Input: Filename, source vertex, destination vertex
    – Initialize the graph and adjacency lists
    – Call the shortest path from the source to the destination (passed in the command line arguments)
    – Print the weight of the shortest path and the vertices on this
    – Running Time (on an average, screenshots shown in Fig 1):
        * 5 vertices graph = 0.000572 seconds
        * 1000 vertices graph = 0.223565 seconds
        * 5000 vertices graph = 0.965366 seconds
        * 1M vertices graph = 51.2879 seconds



```
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ make
rm -f ./*.o
rm -f ./*.out
rm -f ./ssp
rm -f ./routing
g++ -w -std=c++11 -g -c ./FibHeap.cc -o ./FibHeap.o
g++ -w -std=c++11 -g -c ./FibNode.cc -o ./FibNode.o
g++ -w -std=c++11 -g -c ./Dijkstra.cc -o ./Dijkstra.o
g++ -w -std=c++11 -g -c ./SSP.cc -o ./SSP.o
g++ -w -std=c++11 -g -c ./BinTrie.cc -o ./BinTrie.o
g++ -w -std=c++11 -o ./ssp ./FibHeap.o ./FibNode.o ./Dijkstra.o ./SSP.o ./BinTrie.o -g
g++ -w -std=c++11 -g -c ./Routing.cc -o ./Routing.o
g++ -w -std=c++11 -o ./routing ./FibHeap.o ./FibNode.o ./Dijkstra.o ./Routing.o ./BinTrie.o -g
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./ssp DATA/sample_input_part1.txt 0 4
Time Taken: 0.000572 s
5
0 3 2 4
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./ssp DATA/Input/input_1000_50_part1.txt 0 999
Time Taken: 0.223565 s
12
0 670 18 184 856 999
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./ssp DATA/Input/input_5000_1_part1.txt 0 4999
Time Taken: 0.965366 s
214
0 4822 1891 2767 1942 4964 1927 4999
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./ssp DATA/Input/input_1000000.txt 0 999999
Time Taken: 51.2879 s
662
0 40180 155794 208613 57232 689497 596038 285053 418464 109084 788184 345013 345014 380052 999999
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$
```

Figure 1: Average Running Time for Dijkstra's SSP Algorithm

2 Internet Routing Using Binary Tries

1. To run this program, navigate to the folder and type make routing
2. On the command line, type
   $./routing < graph\_filename >< ip\_address\_filename >< source\_vertex >< destination\_vertex >$

2.1 CLASSES USED

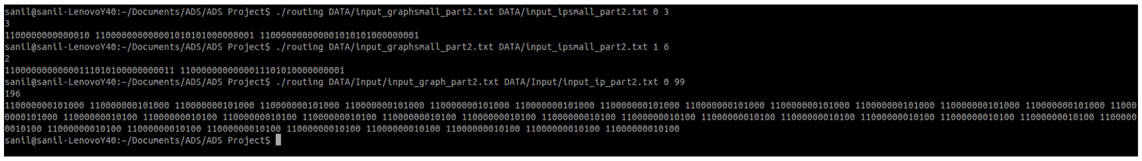
- All the classes mentioned in PROGRAM 1, except ssp.cc.
  - *Binary Trie Implementation: BinTrie.h BinTrie.cc*
    - Binary Trie Node Structure

string data	Key
string value	Value
int bitPosition	The bit position on which considered for this node
int nodeType	Indicates if it is an element or branch node
TrieNode* left	Pointer to the left child
TrieNode* right	Pointer to the right child
  - void RemoveCommonSubTries(TrieNode* p, TrieNode* r)
    - Removes the common sub-tries and compacts the trie
    - Input: A pointer to a node and another to its child
    - No return value
  - bool IsEmpty()
    - Checks if the binary trie is empty
    - No Input
    - Return Value: True, if empty; false, otherwise
  - bool Insert (string key, string value, bool isPostOrder)
    - Insert a $< key, value >$ pair in the trie
    - Input: Key, value, a boolean variable which if true will call RemoveCommonSubtries to compact the trie
    - Return Value: True, if successful; false, otherwise
  - string LongestPrefixMatch (string key, string &value)
    - Finds the longest matching prefix to the key
    - Input: Key to be searched, reference to an integer that will hold the matching value
    - Return Value: The matched prefix
  - TrieNode(int type)
    - Constructor to a trie node
    - Input: Indicates if the node is a branch or an element node
    - Return Value: Pointer to the created node
  - TrieNode(string d, string val, int type)
    - Constructor to a trie node
    - Input: The key-value pair, and an integer that indicates if the node is a branch or an element node

        * Return Value: Pointer to the created node

- *Dijkstra's Algorithm Implementation: Dijkstra.h Dijkstra.cc*
    - In addition to the functions mentioned in PROGRAM 1 under Dijkstra's Algorithm Implementation, the following functions were also implemented as per the requirements of PROGRAM 2.
        * void LoadIP(char* filename)
            · Loads the IP Addresses of the nodes/vertices from the file.
            · Input: Filename containing the IP address information.
            · No return value
        * int InitializeNode (int source, int dest)
            · Finds the next hop from the source to the destination using Dijkstra's shortest path algorithm
            · Input: Source vertex, destination vertex
            · Return Value: Next hop vertex number from source on the shortest path to the destination. Return a -1 if no next hop found.
        * void FindRoute(int source, int dest)
            · Once the routing tables have been initialized across the entire network , use Binary Tries to find the longest matching prefix to reach from source to the destination
            · Input: Source vertex, destination vertex
            · No return value
        * void BuildTrie (int source)
            · Builds a binary trie at the invoking node by inserting $< destination\_ip\_address, next\_hop >$ key-value pairs
            · Input: Source vertex
            · No return value

- *Routing: routing.cc*
    - Contains the *main()* function
    - Intialize the nodes as in the case of Program 1
    - From each source, call Dijkstra's algorithm to find its shortest path from all the other vertices.
    - Retrieve the next hop node on this path.
    - Build a trie at this node by inserting $< destination\_ip\_address, next\_hop\_node >$
    - After the entire network is initialized this way, perform the following steps:
        * Go to the source node.
        * Traverse the trie to find the longest matching prefix to the destination IP address. This will return us the next hop node.
        * Set source = next hop node
        * Sum the distance into the total distance
        * Repeat till we reach the destination
    - Print out the weight of the source-destination path, and the prefixes found in the process.
    - Screenshots shown in Fig 2



```
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./routing DATA/input_graphsmall_part2.txt DATA/input_ipsmall_part2.txt 0 3
3
1100000000000010 1100000000000010101010000000001 1100000000000010101010000000001
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./routing DATA/input_graphsmall_part2.txt DATA/input_ipsmall_part2.txt 1 6
2
1100000000000011101010000000011 1100000000000011101010000000001
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ ./routing DATA/Input/input_graph_part2.txt DATA/Input/input_ip_part2.txt 0 99
196
110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 110000000101000 11000
0000101000 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 1100000
0010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100 11000000010100
sanil@sanil-LenovoY40:~/Documents/ADS/ADS Project$ 
```

Figure 2: Internet Routing Using Binary Tries