Advanced Data Structures Project COP5536 Spring 2015

Name: Sagar Tewari

UFID:10488199

Email: sagartewari@ufl.edu

INDEX

1. **Compiling Instructions**

- Compiler information
    - Operating System - Ubuntu 14.04 LTS
    - Server - thunder.cise.ufl.edu
    - Standard gcc compiler used to compile the C++ code of the project
    - gcc version - 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

Steps to execute the code in (Server)thunder.cise.ufl.edu

1. Unzip the file in any directory.
2. Run "make" command on the terminal to run the makefile
3. The compiler will generate a warning for the Fibonacci file which can be ignored.
4. The code is ready to be executed
5. To run the first part (Dijkstra) -

    ./ssp filename source_node destination_node

6. To run the second part (Binary tries) -

    ./routing filename1 filename2 source_node destination_node

NOTE : The filenames need to have their extensions mentioned

filename0 = graph for first part

filename1 = graph for second part

filename2 = ip addresses for second part

NOTE : The second part doesn't run for larger input files provided but works perfectly for smaller ones

Class Descriptions and Structure overview:

Fibonacci.cpp

Edge::Edge(graph_nd* t, graph_nd* hd, double ln)

graph_nd::graph_nd(int d, double k)

graph_nd::graph_nd()

bool graph_nd::child_add(graph_nd *n)

bool graph_nd::Sib_add(graph_nd *nd)

bool graph_nd::remove()

void graph_nd::add_in_edge(Edge * edge)

void graph_nd::add_out_edge(Edge * edge)

graph_nd* graph_nd::Sibl_L()

graph_nd* graph_nd::Sibl_R()

FibonacciHeap::FibonacciHeap(graph_nd *r)

FibonacciHeap::~FibonacciHeap()

bool FibonacciHeap::isEmpty()

bool FibonacciHeap::vrtx_ins(graph_nd * n)

graph_nd* FibonacciHeap::findMin()

graph_nd* FibonacciHeap::minDelete()

bool FibonacciHeap::link(graph_nd* r)

void FibonacciHeap::keyDec(double delta, graph_nd* v)

ssp.cpp

int main(int argc, char *argv[]) [this is the main for part1]

{

reads the input file and has the dijkstra logic implemented in this block

}

trie.cpp

```
struct node

struct node *crNode()

void trie_add(struct node **root,char *str,int nhop)

void del(struct node *root)

int trim_trie(struct node **root)

void find_gateway(node **routing_table,int start,char *ip,int end)

void dec_to_bin(char *binaddr,char *tempbin, int n)

void convert(char *ip_addr,char *binaddr)

int main(int argc, char const *argv[])

{

file reading, routing and binary trie logic implemented here

}
```

Structure of the functions

fibonacci.cpp

The fibonacci.cpp file has the entire data structure for the fibonacci heap which is used for implementing Dijkstra agorithm for the first part of the project.

The detailed function description is given below :

- Edge::Edge(graph_nd* t, graph_nd* hd, double ln) -> It is used for declaring edges for adjacency graph (matrix).
- graph_nd::graph_nd(int d, double k) ->this is the overloaded constructor for various graph parameters.
- graph_nd::graph_nd() ->this is the graph constructor
- bool graph_nd::child_add(graph_nd *n) ->this function adds a child to a node and returns success upon successful completion
- bool graph_nd::Sib_add(graph_nd *nd) ->this program is used to add a sibling to a node and return true upon successful completion
- bool graph_nd::remove() ->removes a node by checking it's parent, left sibling and right sibling
- void graph_nd::add_in_edge(Edge * edge)v ->adds incoming egde to a node
- void graph_nd::add_out_edge(Edge * edge) ->adds outgoing edge to a node
- graph_nd* graph_nd::Sibl_L() ->this program is used to find the leftmost sibling of a specific node at a particular level
- graph_nd* graph_nd::Sibl_R() ->this program is used to find the rightmost sibling of a specific node at a particular level.
- FibonacciHeap::FibonacciHeap(graph_nd *r) -> used to initialize the fibonacci heap to work upon
- bool FibonacciHeap::isEmpty() ->used to check if the fibonacci heap is empty
- FibonacciHeap::~FibonacciHeap() -. deletes the heap once it goes out of bounds
- bool FibonacciHeap::vrtx_ins(graph_nd * n) -> inserting a vertex into the heap by checking the initial condition if the node is empty or not
- graph_nd* FibonacciHeap::findMin() ->returns the minimum node in the heap
- graph_nd* FibonacciHeap::minDelete() ->deletes the minimum node from the heap
- bool FibonacciHeap::link(graph_nd* r) ->used to link the nodes in a fibonacci heap
- void FibonacciHeap::keyDec(double delta, graph_nd* v) ->performs decrease key operation on specific nodes as required by the algorithm

ssp.cpp

This function contains the main function for the first part and is used to implement the Dijkstra's algorithm. It creates vectors for storing edges and vertices as adjacency list. It takes input files and source and destination nodes from console and implements the Dijkstra's algorithm.

trie.cpp

This file implements binary tries and the entire logic for the program. It takes in inputs, creates routing table, does longest prefix matching and trims the tree.

- struct node -> defining nodes for binary trie
- struct node *crNode() ->creates new trie nodes
- void trie_add(struct node **root,char *str,int nhop) ->adds ip addresses to the router specified by **root. nhop has the next_hop value
- void del(struct node *root) ->used in postorder trimming of trie
- int trim_trie(struct node **root) -> it is used to trim the tree by checking both the left and right subtree if they are zero.
- void find_gateway(node **routing_table,int start,char *ip,int end)

  +struct node *par -> points to the parent of crawling pointer, used to get next hop router
- void dec_to_bin(char *binaddr,char *tempbin, int n) ->converts a number to its binary representation
- void convert(char *ip_addr,char *binaddr) ->converts an ip address to 4 numbers and then to its binary representation
- int main(int argc, char const *argv[])

  +char ip_addr[16] = stores Ip addresses
  +char binaddr[33] = stores binary addresses

+int start = reads starting router

+int end = reads ending router

+struct node *routing_table[vertices] = initialize routing table

- The main function creates a routing table, calls Dijkstra (implemented in part1) to get the shortest distance to the next hop router.

makefile

This file is used to compile the entire code all at once instead of compiling each file separately and allows us to execute our programs following this.

OUTPUTS

- **PART 1**
  - The part 1 works good for the input files with 1000, 5000 and 1 million nodes.
  - It takes approximately 20 seconds for the code of part 1 to run for 1 million input nodes.
  - This implies that the fibonacci heap data structure has been implemented efficiently.

PART 2

The code for second part works fine for smaller input files provided in SAKAI in the first place but doesn't work for larger input files and gives segmentation error as it get's struck with calling ssp from part1.

NOTE - Do run the code for second part for smaller inputs provided in SAKAI.

CONCLUSION

The program works as expected for the first part. We get the shortest distance using Dijkstra's algorithm implemented by fibonacci heap.

The program works fine for smaller inputs of the second part and breaks weirdly for larger inputs due to memory allocation problems.