ADVANCED DATA STRUCTURES (COP 5536)

Spring, 2015

FINAL PROJECT REPORT

Kushagra Udai
UFID: 0937-7483
kudai@ufl.edu

<u>INTRODUCTION</u>

This project has two parts.

Part 1:

An implementation of Dijkstra's Single Source Shortest Path (**ssp**) algorithm for *undirected graphs* using a Fibonacci Heap as a min-priority queue. An *adjacency list* was used for the graph representation.

Part 2:

This implements a routing scheme (**routing**) for a network. Each router has an IP address and packets are forwarded to the next hop router by longest prefix matching using a binary trie. For each router in the network, ssp implemented in Part 1 is called to obtain shortest path from that router to each destination router. To construct the router table for R, for each destination Y, we examine the shortest path from R to Y and determine the router Z just after R on this path. This gives us a set of pairs <IP address of Y, next-hop router Z>. We Insert these pairs into a binary trie. Finally, we do a postorder traversal, removing subtries in which the next hop is the same for all destinations. Thus, multiple destinations having a prefix match and the same next hop will be grouped together in the trie.

Once there is the path, the next hop router is obtained and inserted into the leaf nodes of the router's binary trie for each of the destination.

This way the routing scheme hops from the start router to it's next hop and so on and so forth till it reaches the destination router.

Compiler Details/Execution Instructions:

Compiler:
gcc version 4.9.1 on Ubuntu 14.10.

Execution:

To execute the program, extract **Udai_Kushagra.tar.zip** using '**tar –zxvf**' in the terminal while in the same folder where the tar ball resides. Then type '**make**' in the extracted folder. A Makefile has been attached for compiling all the .cpp files (and the .h files) and creating .o files as well as the two executables: **ssp** and **routing**.

Once the executables are created, part 1 can be executed using:
./ssp file_name source_node destination_node
and part 2 using:
./routing file_name_1 file_name_2 source_node destination_node

PROGRAM STRUCTURE (PART 1)

Header Files:
- Node.h
- List.h
- fHeap.h
- Graph.h

Class Files:
- Node.cpp
- List.cpp
- fHeap.cpp
- Graph.cpp
- mainssp.cpp

Node.h/Node.cpp
This class defines a basic Fibonacci Heap node structure.

Attributes:
- **next (Node*)**: A pointer to the next node in the List in which the object belongs.
- **prev (Node*)**: A pointer to the previous node in the List in which the object belongs.
- **parent (Node*)**: A pointer to the parent node of the List in which the object belongs.
- **child (List*)**: A pointer to the List which has all the children of this node.
- **childcut (bool)**: A boolean value that defaults to false for all root nodes and all new nodes. Once a node loses a child, it is set to true. This is an important property for the cascading cut operation of a fibonacci heap.
- **data (int)**: This field stores the distance from the source vertex to this particular node. The value is relaxed untill the shortest distance is reached.
- **degree (int)**: This is also an integer field stores the number of children of the node, i.e. the number of nodes in the List pointed to by the child pointer.

Methods:
- **Node()**: The default constructor that initializes the integer attributes to 0, the pointer attributes to nullptr and the boolean attribute to false.
- **Node(Node *copyfrom, Node *copyto):** The constructor that copies all attributes from the copyfrom object to the copyto object.

List.h/List.cpp
This class defines a circular doubly linked list (List) that is used to link all the root nodes of the heap as well as the child nodes of each node.

Attributes:
- **head (Node*)**: This a pointer that points to the start node of the List.

Methods:

- **List():** Default constructor that initializes the head pointer to nullptr.
- **insertNode(Node*, Node*,bool):** This method, given the pointer to the minimum node in the root of the heap, creates and inserts a new node to the left of the minimum node, if the bool is true (which it is by default) or simply inserts the node without allocating new memory for it if the bool is false. It returns a pointer to the newly inserted node.
- **deleteNode(Node*, bool):** This deletes the given Node from the invoking List object. If the bool is true (which it is by default), it clears the memory occupied by that node, if not, it leases the memory alone so the node can be moved elsewhere given access to it’s pointer. It returns a pointer to the deleted node.
- **mergeLists(Node*):** This method is used merge two lists into one new list. It is typically called when a child List needs to be inserted into the root List, usually after an operation to extract minimum. It returns a pointer the to merged List.

fHeap.h/fHeap.cc

This class defines the basic structure and operations of a fibonacci heap. It involves the basic operations like Insert, Extract Minimum, Decrease Key and the complementary operations like Pairwise Combine and Cascading Cut.

Attributes:

- **minptr (Node*):** This pointer points to the node that has the minimum data in the roots List of the heap.
- **roots (List*):** This is a pointer to the root List of the heap. It has the head pointer information of the root List for the heap.

Methods:

- **fHeap():** This is a default constructor that initializes minptr to nullptr and roots to a new List with a nullptr head pointer.
- **isEmpty():** This method checks if the fHeap object invoking it is empty. It returns a bool which is true if it is empty and false if it is not.
- **insert(int):** Given a integer key, it creates a new Node and calls the insertNode function of the List class to insert into the root List of the heap. It returns a pointer to the inserted node.
- **extractMin():** This method implements the extract minimum functionality of a fibonacci heap. It assigns the node to be deleted as the minptr node and calls the deleteNode function of the List class. After that, it gets the child List of the deleted node. If the child List as well as the root List are empty, then the heap is emptied. If roots List is empty and the child List is not, then the new root List becomes the child List. If only the child List is empty, the root List remains. Otherwise the mergedLists function of List is called to merge both the root and child List.
  If the root List has just one node or is empty, minptr is set to that one node.
  If the root List has more than one node, pairwise combine is performed on the root nodes in the heap. It returns the extracted minimum Node pointer.
- **pairwiseCombine():** This method first initializes a hash map that has the key as the

degree and the value is the pointer to the node that has that degree. The minptr is set to the head node of the root List and a loop is executed which iterates through the nodes of the root List starting at the head. Inside the loop, another loop is initialized which checks whether the degree of the current node already exists in the hash map. If yes, depending on which node has the smaller value, one becomes the child of the other. The conflicting entry is deleted and the hashmap is again searched for conflicts with the new degree after the addition of the child. This joining continues till there are no more conflicts. The degree and the node is then inserted into the hashmap. In the meanwhile, with every iteration of the outer loop, the minptr is reassigned if there is a smaller value. This method does not return a value.

- **decreaseKey(Node*, int):** Given the pointer to the node, if the new reduced value is less than the already present value and less than the value of the node's parent, it is deleted from the child List and inserted into the root List using the List methods. Depending on the childcut value of the parent of the node, the next method (cascadingCut) is called. This method returns true if the decreaseKey operation succeeds and false otherwise.
- **cut(Node*,Node*):** This method removes the child to be cut from the child List of it's parent pointer and inserts it into the roots List of the fHeap using the insertNode and deleteNode functions of the List class. This method does not return a value.
- **cascadingCut(Node*):** This method is called in recursion until either a node in the root List is reached or the childcut value of the parent node becomes false. In the loop, the node is cut off from the child List and inserted into the root List using the cut method. This method does not return a value.

Graph.h/Graph.cc:

This class defines a graph by initializing an adjacency list representation. It also implements the Dijkstra's algorithm.

Attributes:

- **vertices (int):** This stores the number of vertices in the graph..
- **edges (int):** This stores the number of edges in the graph.
- **adjlist (unordered_map<int, unordered_map<int,int>>):** This is a hashmap with the key as a vertex(integer) and the value as another hashmap (key: vertex(int). value: edge weight(integer)). It is used to represent the adjacency list for the graph.

Methods:

- **Graph():** Default constructor which initializes both vertices and edges to 0.
- **insert(int,int,int):** This method inserts into the unordered map given the vertex1, vertex2 and the edge weight between them thereby creating the adjacency list for the graph.
- **dijkstra(int int,bool):** This method implements the ssp algorithm. It first defines hash maps that stores the <vertex, Node*> pair and a reverse <Node*, vertex> pair. Values are inserted into these every time a node is inserted into the heap for the first time. A path[] array is defined to store the previous vertex in the shortest path and the

shortest distance from source as a pair for a given vertex. The path[] values are initialized to -1 and INT_MAX Then, for source, the path distance is set to 0. After the source node has been inserted into the heap, the method loops till either the heap is empty or the end node is extracted from the heap. Inside the loop, extractMin() method of the fHeap class is called. For the extracted node, it's adjacent vertices are obtained. And their distance values are relaxed. If the relaxed values are lesser than the existing, the node is inserted into the heap (if it is not already present) else, its value is decreased using the decreaseKey method of the fHeap class. If the extracted node is equal to the given end node, the loop terminates. Now, if the bool is true (which it is by default), its distance is printed to the output stream as the ssp distance and the path from there to the source node is computed using the previous values and inserted into a stack. The stack is then popped to print the path from source to the end node. If the bool is false, the previous values are inserted into a vector and then the shortest computed distance is pushed to the end of the vector which is then returned.

mainssp.cc:
This contains the main function for executing part 1 of the project. A ifstream is initialized to input data from the input file. It also has an object of the Graph class. On reading each line from the file, the Insert() method of the Graph class is called so as to create the adjacency list. And lastly, the Dijkstra method is called, with the default bool.

PROGRAM STRUCTURE (PART 2)

Header Files:
- TrieNode.h
- Router.h

Class Files:
- TrieNode.cc
- Router.cc
- mainrouting.cc

TrieNode.h/TrieNode.cc
This class defines the basic structure of a node in a binary trie.

Attributes:
- **left (TrieNode*)**: This pointer points to the current node's left sub trie.
- **right (TrieNode*)**: This pointer points to the current node's right sub trie.
- **Data (int)**: This is a integer field that stores the data of the trie node.

Methods:
- **TrieNode():** The only method in this class is a default constructor that initializes the integer attribute to 0 and the pointer attributes to nullptr.
- **create(int, vector<string>, vector<int>):** This method creates a trie by inserting every vertex IP into the root trie using the insert method for the given source by calculating the nextHop value using the getNextHop method.
- **insert(string, int):** Given the root to a trie, and a IP string, level by level trie is

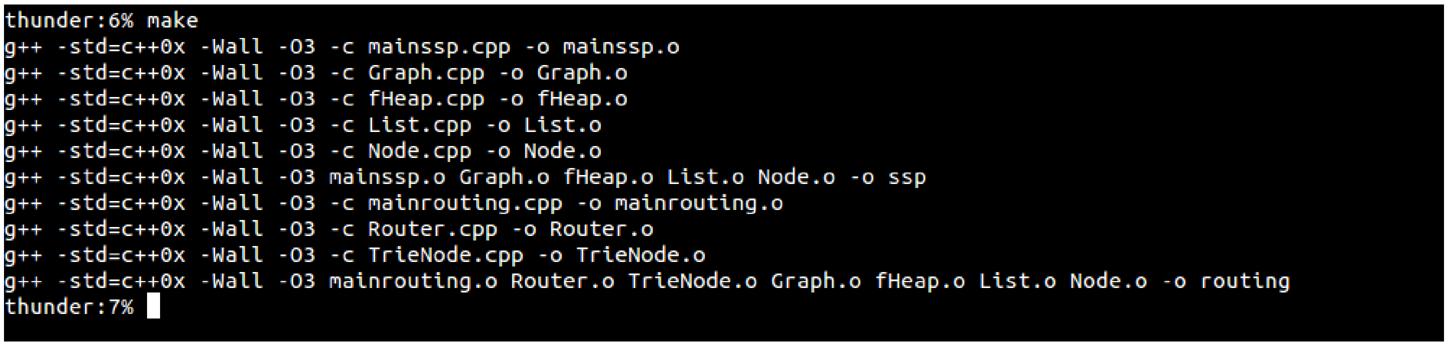
traversed if it exists, else new nodes are created in the path until the entire string has been represented. The last node that is reached/created is set as the leaf node and its value is set to the next hop value.

- **getNextHop(int, int, vector<int>):** As the name suggests, this method is used to find the next hop from a particular source to a particular destination. Here, the vector<int> argument is the previous vector that is returned by the Dijkstra method of the Graph class where it runs for all destinations. Now, given that vector<int>, the method loops through all destinations, finding the ssp from the source to each and uses that path to find the next hop. It finally returns the next hop for that source and destination.
- **condense(TrieNode*):** This method is recursed the same way as for a post fix traversal. Once it recurses, 3 conditons are checked for and the pointers are set accordingly. The conditions are that: 1) Either the left or the right subtree is null: In that case, the existing subtree is brought a level up and replaced with the local root. 2) Both the subtrees are null: In this case, nothing is done. 3) Neither of the subtrees are null but they are both the leaf nodes and they both have the same value. In this case, the local root is replaced by the left leaf node and the right one is deleted.
- **traverse(string):** Given the IP string, the trie is traversed according the bits of the IP till either the IP ends or a leaf node is reached. At every bit, the bit is printed as the prefix.

mainrouting.cc:

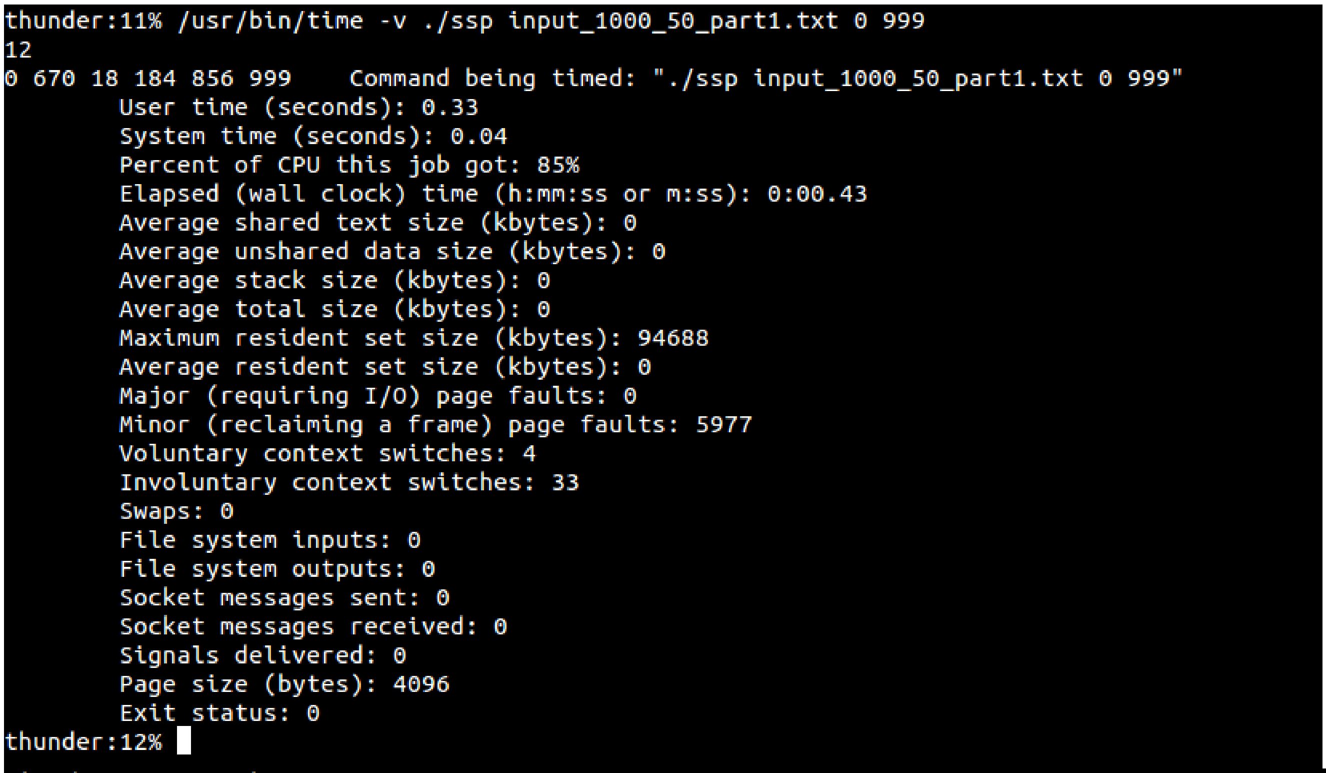
This contains the main function for executing part 2 of the project. A ifstream is intialized to input data from the input file. It also has an object of the Graph class. On reading each line from the file, the Insert() method of the Graph class is called so as to create the adjacency list. Then the second input file is read to take the IP addresses. Each IP is passed to the to32BitBinary which tokenizes into 4 number sets separated by '.'. A bitset function is applied on these tokens to convert the IP into a 32 bit binary. The binary IP is then returned as a string by the to32BitBinary function and stored into a vector by the main function.
Dijkstra method is then called for the start and end node to return the ssp distance and print it. After, for each of the vertices in the graph, the Router() object is initialized to and it's run method is called which then create all the tries and display the path and prefixes.

RESULTS:

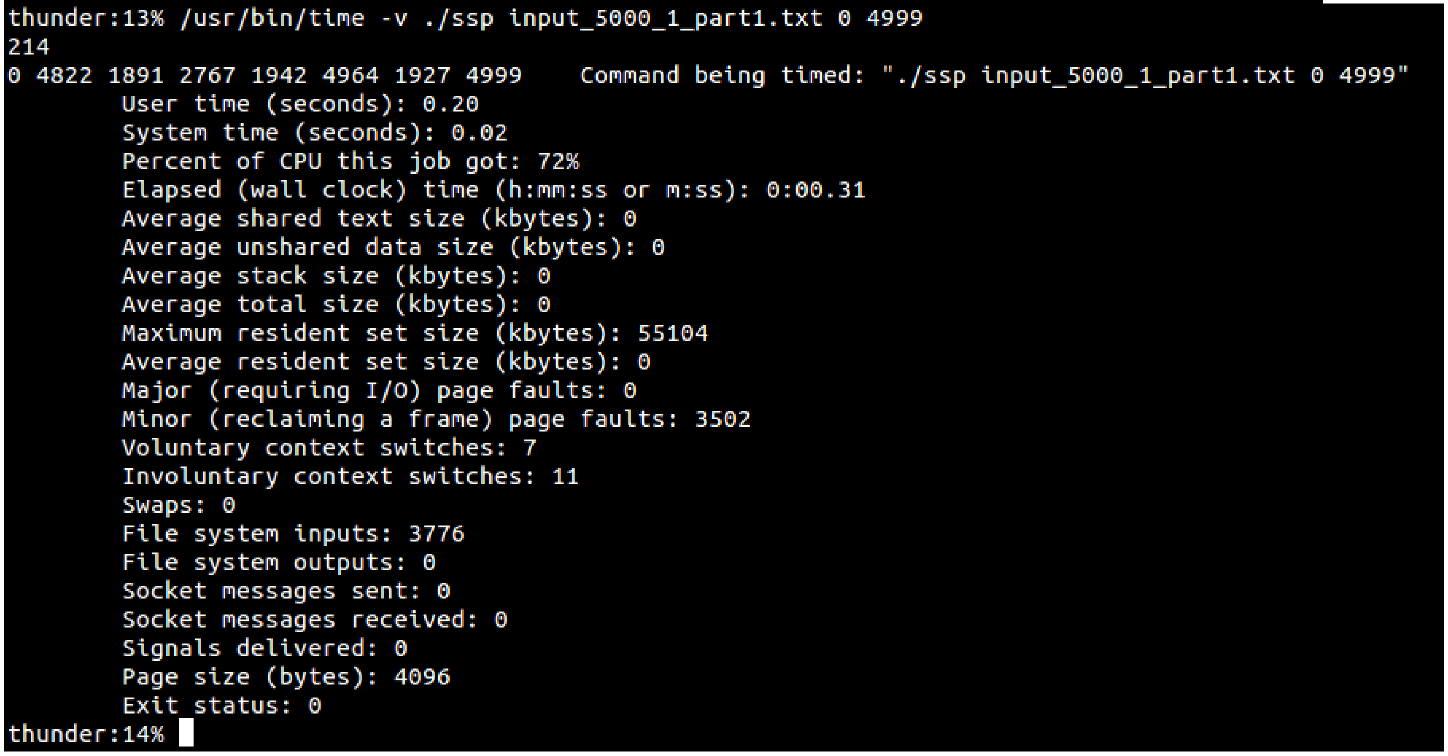


```
thunder:6% make
g++ -std=c++0x -Wall -O3 -c mainssp.cpp -o mainssp.o
g++ -std=c++0x -Wall -O3 -c Graph.cpp -o Graph.o
g++ -std=c++0x -Wall -O3 -c fHeap.cpp -o fHeap.o
g++ -std=c++0x -Wall -O3 -c List.cpp -o List.o
g++ -std=c++0x -Wall -O3 -c Node.cpp -o Node.o
g++ -std=c++0x -Wall -O3 mainssp.o Graph.o fHeap.o List.o Node.o -o ssp
g++ -std=c++0x -Wall -O3 -c mainrouting.cpp -o mainrouting.o
g++ -std=c++0x -Wall -O3 -c Router.cpp -o Router.o
g++ -std=c++0x -Wall -O3 -c TrieNode.cpp -o TrieNode.o
g++ -std=c++0x -Wall -O3 mainrouting.o Router.o TrieNode.o Graph.o fHeap.o List.o Node.o -o routing
thunder:7% 
```

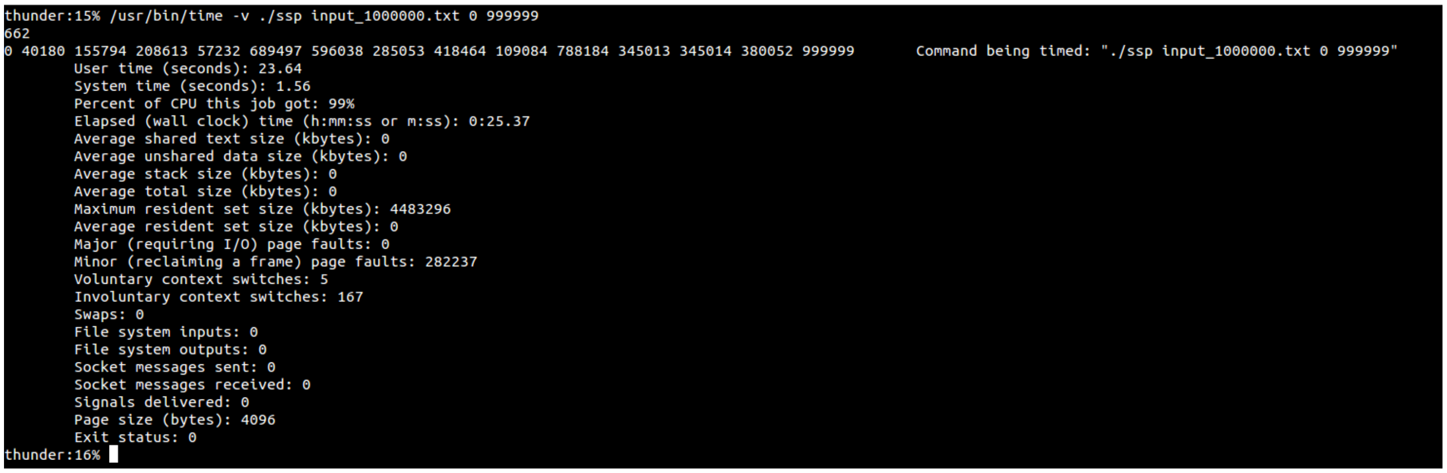
PART 1:



```
thunder:11% /usr/bin/time -v ./ssp input_1000_50_part1.txt 0 999
12
0 670 18 184 856 999    Command being timed: "./ssp input_1000_50_part1.txt 0 999"
        User time (seconds): 0.33
        System time (seconds): 0.04
        Percent of CPU this job got: 85%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.43
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 94688
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 5977
        Voluntary context switches: 4
        Involuntary context switches: 33
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
thunder:12%
```

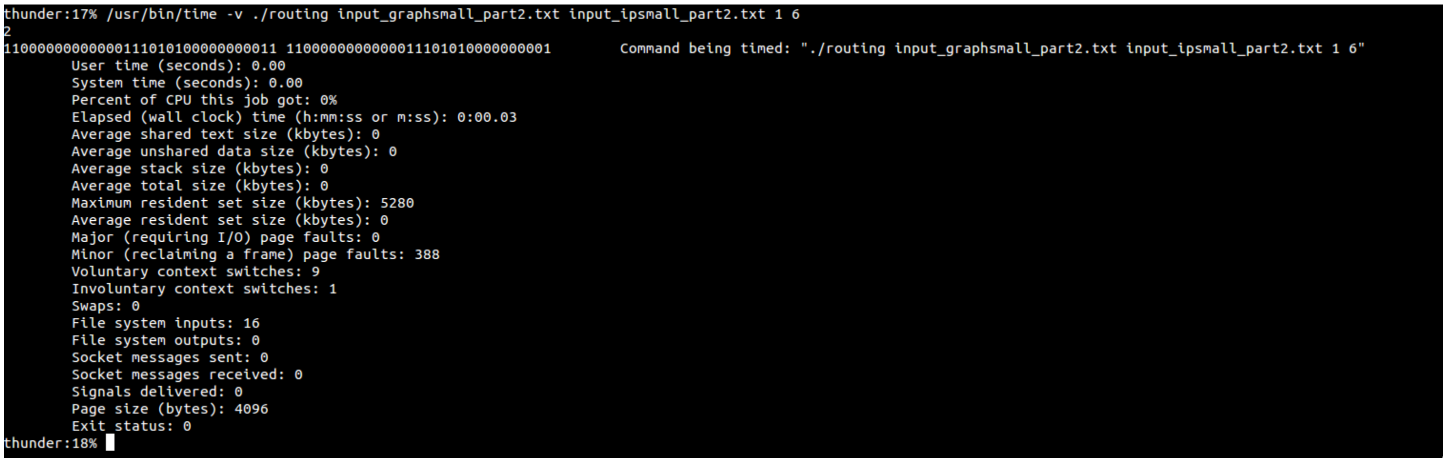


```
thunder:13% /usr/bin/time -v ./ssp input_5000_1_part1.txt 0 4999
214
0 4822 1891 2767 1942 4964 1927 4999    Command being timed: "./ssp input_5000_1_part1.txt 0 4999"
        User time (seconds): 0.20
        System time (seconds): 0.02
        Percent of CPU this job got: 72%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.31
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 55104
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 3502
        Voluntary context switches: 7
        Involuntary context switches: 11
        Swaps: 0
        File system inputs: 3776
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
thunder:14%
```

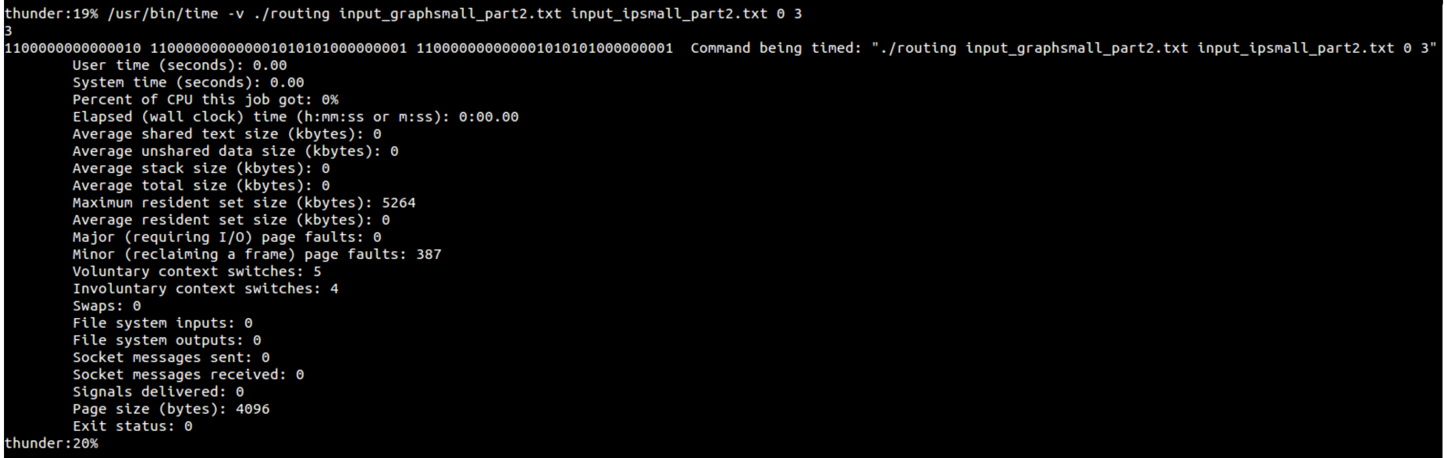


```
thunder:15% /usr/bin/time -v ./ssp input_1000000.txt 0 999999
662
0 40180 155794 208613 57232 689497 596038 285053 418464 109084 788184 345013 345014 380052 999999       Command being timed: "./ssp input_1000000.txt 0 999999"
        User time (seconds): 23.64
        System time (seconds): 1.56
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:25.37
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 4483296
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 282237
        Voluntary context switches: 5
        Involuntary context switches: 167
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
thunder:16%
```

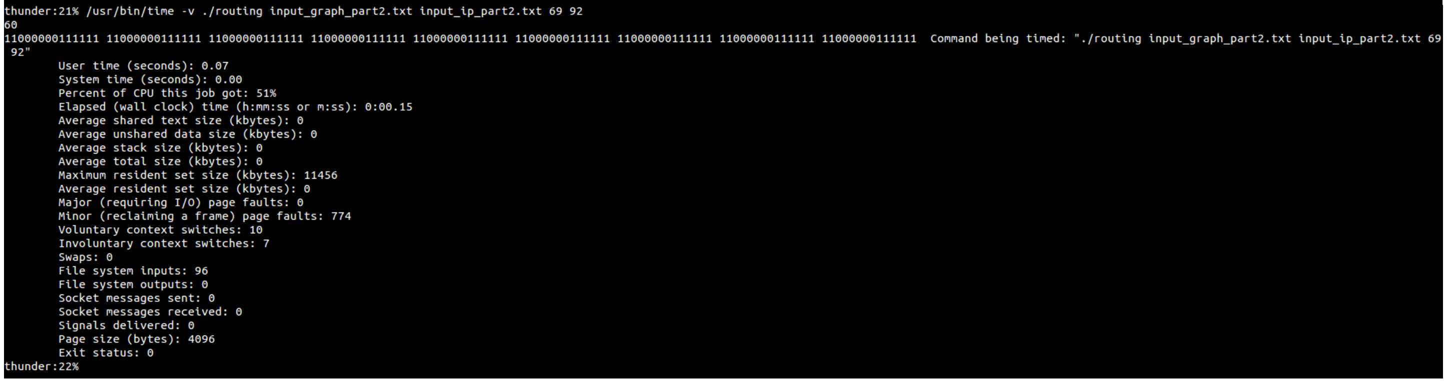
PART 2:



```
thunder:17% /usr/bin/time -v ./routing input_graphsmall_part2.txt input_ipsmall_part2.txt 1 6
2
11000000000000111010100000000011 11000000000000111010100000000001        Command being timed: "./routing input_graphsmall_part2.txt input_ipsmall_part2.txt 1 6"
        User time (seconds): 0.00
        System time (seconds): 0.00
        Percent of CPU this job got: 0%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.03
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 5280
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 388
        Voluntary context switches: 9
        Involuntary context switches: 1
        Swaps: 0
        File system inputs: 16
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
thunder:18%
```



```
thunder:19% /usr/bin/time -v ./routing input_graphsmall_part2.txt input_ipsmall_part2.txt 0 3
3
1100000000000010 11000000000000101010100000000001 11000000000000101010100000000001  Command being timed: "./routing input_graphsmall_part2.txt input_ipsmall_part2.txt 0 3"
        User time (seconds): 0.00
        System time (seconds): 0.00
        Percent of CPU this job got: 0%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 5264
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 387
        Voluntary context switches: 5
        Involuntary context switches: 4
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
thunder:20%
```



```
thunder:21% /usr/bin/time -v ./routing input_graph_part2.txt input_ip_part2.txt 69 92
60
11000000111111 11000000111111 11000000111111 11000000111111 11000000111111 11000000111111 11000000111111 11000000111111 11000000111111  Command being timed: "./routing input_graph_part2.txt input_ip_part2.txt 69
 92"
        User time (seconds): 0.07
        System time (seconds): 0.00
        Percent of CPU this job got: 51%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.15
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 11456
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 774
        Voluntary context switches: 10
        Involuntary context switches: 7
        Swaps: 0
        File system inputs: 96
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
thunder:22%
```