COP 5536

Final Project Report
Of

Name:Pooja Ahuja
UFID: 4996-3849
E-mail id: poojabahuja@ufl.edu

Advanced Data Structures- Project

- Part 1 implements Dijkstra's single source shortest path using the fibonacci heap data structure.

- Part 2 implements tries and compressed mapping using part 1

1. Working Environment:

System Requirements:

- JDK 1.5 or more
- 5 GB of RAM (For running million node graph)
- Operating System: Linux/Windows

Compilation and Running instructions:

The project has been compiled and tested on 'Thunder' server (thunder.cise.ufl.edu).
In the folder Ahuja_Pooja:

1. Run command 'make' to compile all the java files.
2. Now the folder should contain all the .class files.
3. Use the command "java ssp/routing Arguments" to run the code.

Note: If required, assign additional heap space to the by passing the VM argument -Xmx4000m. (If running on server, no need to assign extra memory for JVM)

Program Logic and Flow:-

Part1:

Classes: - ssp.java, RNode.java, Fibo_Node.java, fibonacci_heap.java

Main Class – ssp.java

1. dijkstra_function() creates the fibonacci_heap object.
2. RNode and Fibo_node() objects are mapped to each other created and added to a fibonacci heap.
3. dijkstraAlgorithm() prints out the weight of the shortest path along with the path.

Part2

Classes: - routing.java, TrieNode.java, TrieStruct.java

Main Class- routing.java

1. An RNode has a TrieStruct() object.
2. dijkstra_function() is called repeatedly. Also, the tries are populated.
3. Compression is done and the compressed nodes on the path are printed along with weight.

Structure of the program:

Classes used according to the program flow:-

Fibo_Node.java

public void addNewChildToHeapt(Fibo_Node fibo_node)

- It adds a fibo_node to its child list

public fibonacci_heap getParent(fibonacci_heap f)

- Returns the parent node of the fibonacci heap f

RNode.java

public void r_map(RNode node, int weight)

- It populates adjacency list of each RNode. It adds node to adjacency list and assigns weight to the RNode.

ssp.java

public static void main(String args[])

- The inputs are read from a file taken as an input in arguments.
- Then they are assigned to the adjacency list.
- Then Dijkstra's algorithm is called for the start node.

private static void createFiboHeap(RNode rnode)

- RNode and Fibo_node() objects are mapped to each other created and added to a fibonacci heap.

public static void dijkstra_function(RNode node) throws Exception

- Dijkstras algorithm is implemented via a Fibonacci heap.
- The vertices are stored in RNode.

- The start node in put inside the Fibonacci heap and all of its adjacency list elements with weights are relaxed and put in Fibonacci heap.
- Then, we delete the minimum element and now, find a new minimum, we take all elements from its adjacency list, and using decrease key, we relax their weights.(continues till we hit the destination node)

fibonacci_heap.java

public void addNode(Fibo_Node f_node)

- A new node is added to the fibonacci heap's top level Linked List.

public void deleteMinimumFunction()

- This performs the deleteMin() operation.
- 'start' points to the min node every time. It gets removed and start is assigned to the new min.
- pairWiseCombine () function is then called to restructure the heap.

public void add_toTopLevel(Fibo_Node node)

- It adds a node in the top level list. It is called by decreaseKey.

public void findNewMinimum()

- It ensures that the start points to the correct minimum node after operations.

public void pairWiseCombine ()

It implements the pairwise combine operation of Fibonacci heap. 'tracker' hashmap is maintained to track node degrees. It traverses the top level linked list and then combines the nodes with matching degrees.

public void decrease_key(Fibo_Node f_node, int val)

- It decreases f_node to value.
- It implements decrease_key operation of Fibonacci heap.
- If the key value of node n is less than its parent, we remove the node and add it to top level list.

- Cascading cut and childCut is also handled in this function

TrieNode.java

- Defines the structures for a Trie Node which is used in the Trie data structure constructed for routing.
- It contains 2 constructors- one default constructor and one parameterized constructor

TrieStruct.java

public void add(String ip_addr,int data)

- It adds a branch to the current trie.
- It adds the ip address branch and at the leaf, it has the pointer to the next hop.

public void compress_function(TrieNode root_node)

- It compresses the trie represented by root node recursively (in a postorder manner).

public String traverse(String str)

- It is used to traverse the current nodes trie for the destination node.
- ip address is represented by str.
- It returns the compressed path of the ip address.

routing.java

public static void main(String args[])

- It reads all input and maps the adjacency list accordingly. Additionally it also maps the ip addresses. It runs Dijkstras on each node, compressed trie is printed along with weight

public static void reset()

- Resets the values of the RNodes

public static void dijkstra_r(RNode node)throws Exception

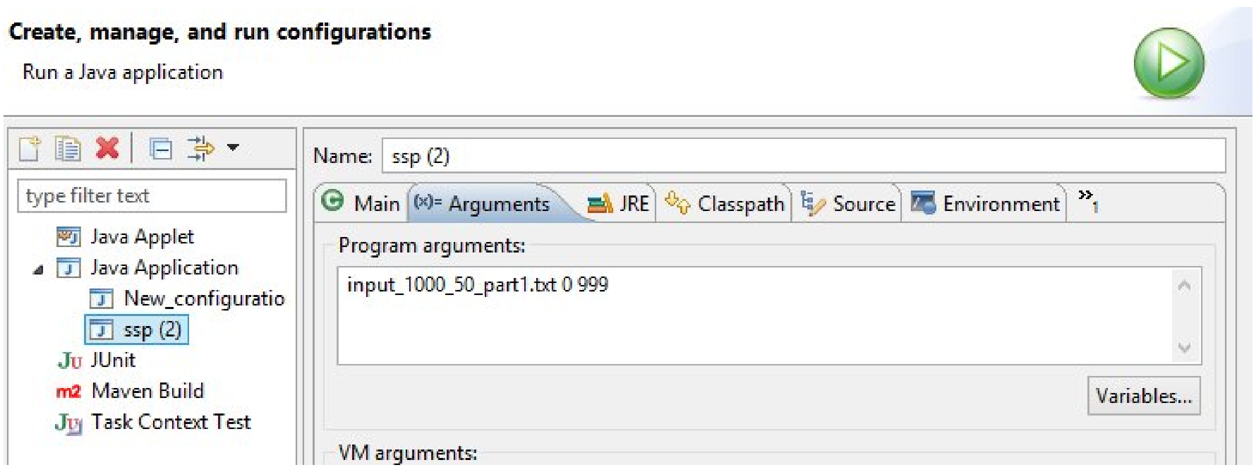
- Works same as ssp function.

Screenshots of outputs:

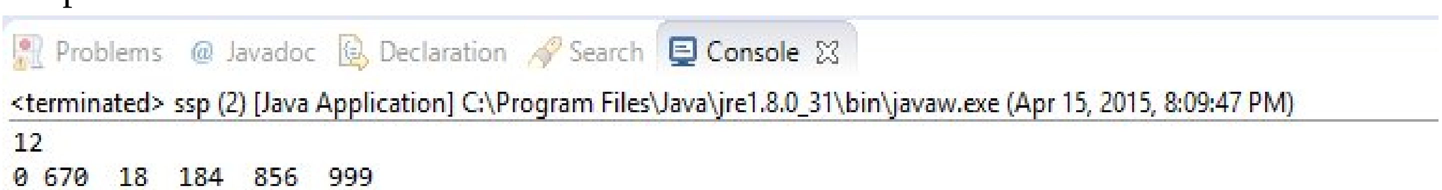
The following are the screenshots taken on the eclipse IDE after completing both the parts successfully.

Part 1 : SSP

1. For 1000 nodes:
   Input:

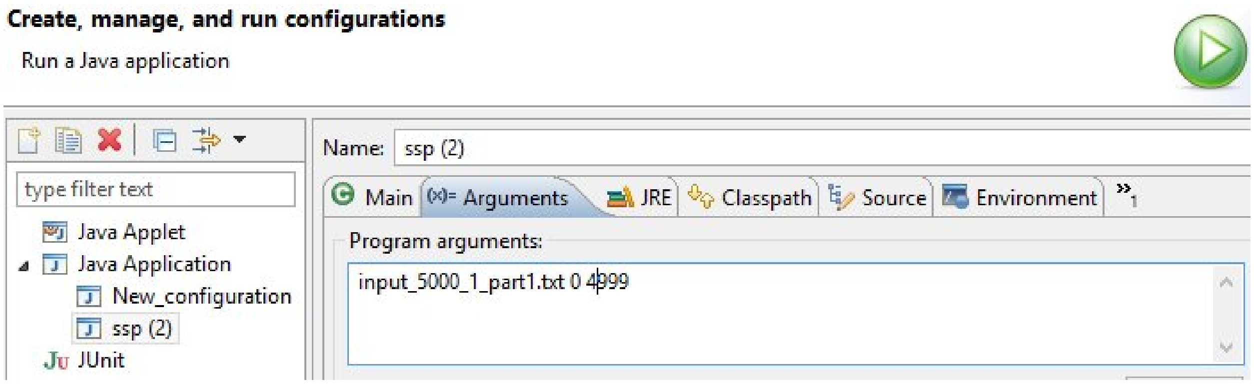


Output:

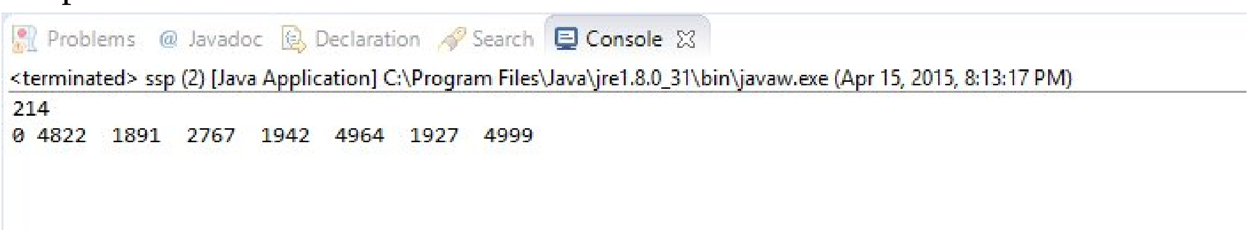


2. For 5000 nodes:

Input:

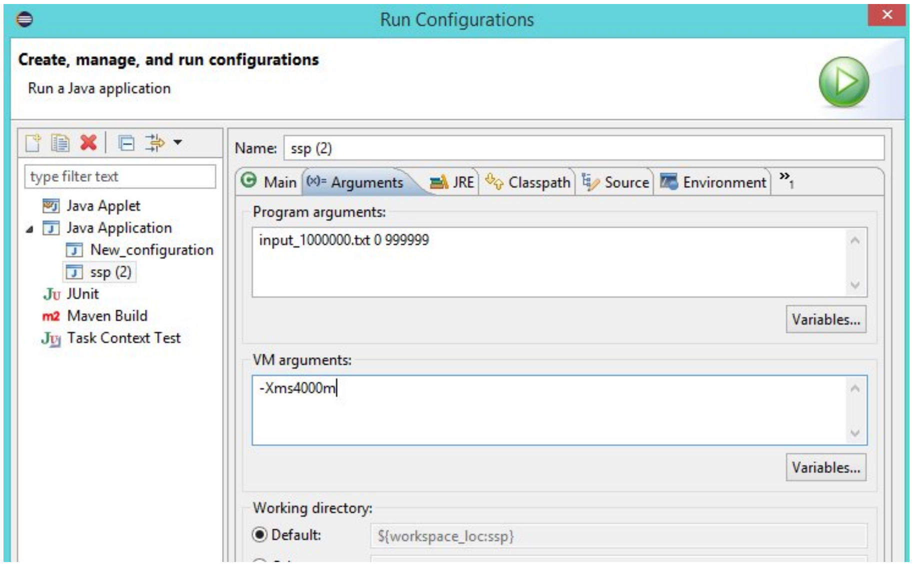


Output:

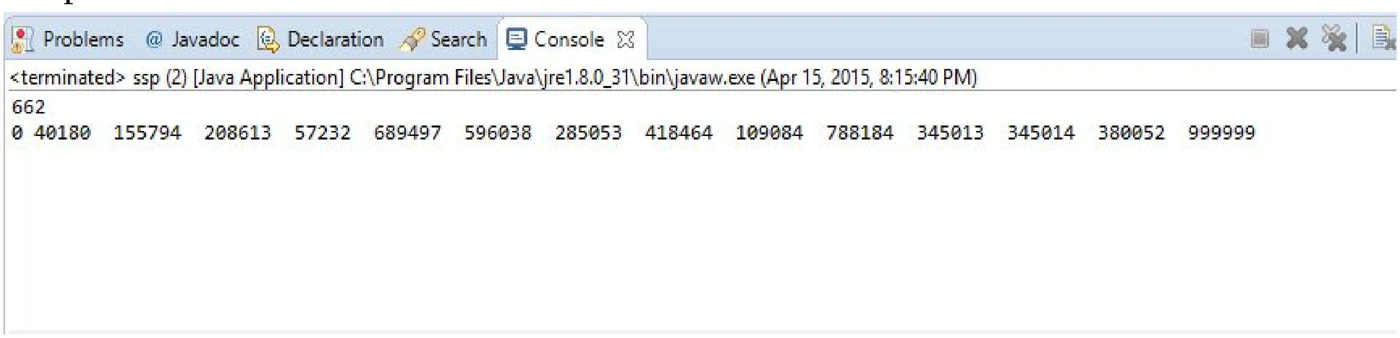


3. For 1 million nodes:

Input:



Output:

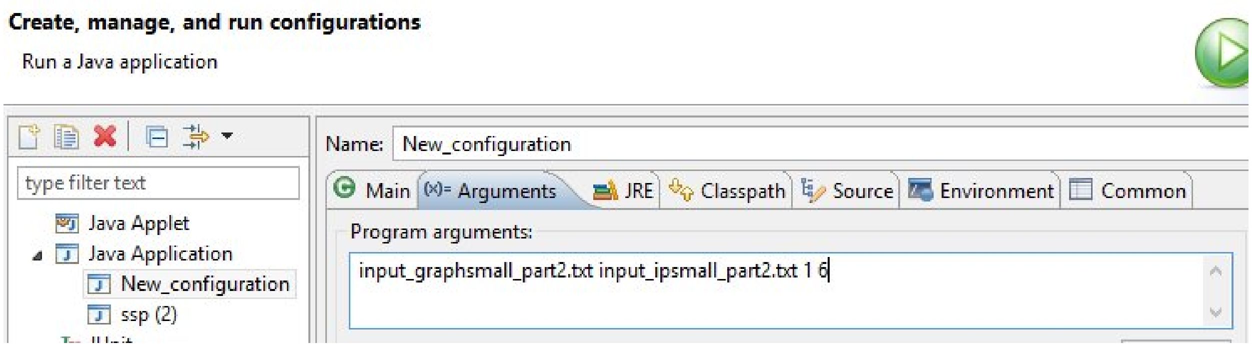


For million nodes, it takes around 5 minutes. However, most of the time is taken for file i/o operation. The dijkstras algorithm itself takes just 13 secs for million nodes.

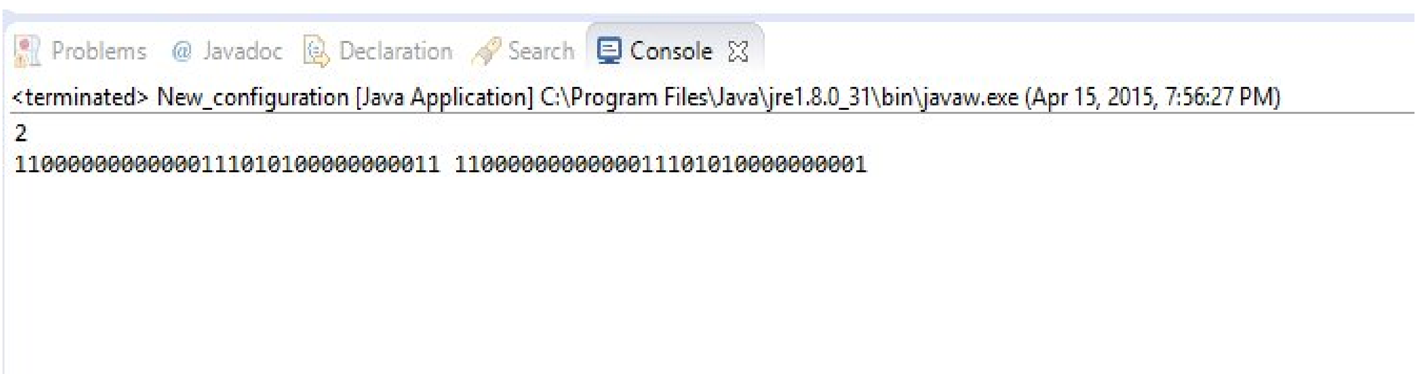
Routing:

1. For input_graphsmall_part2.txt and input_ipsmall_part2.txt:

Input:

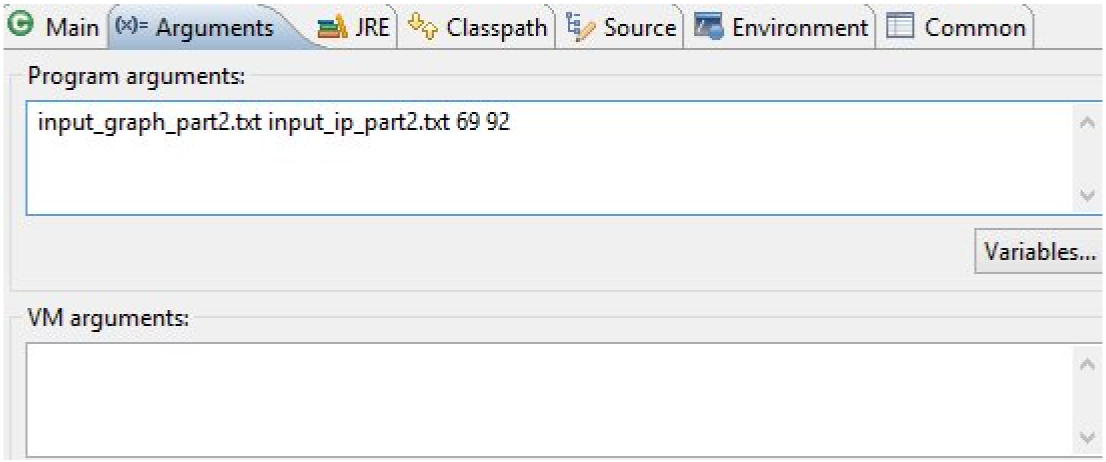


Output:

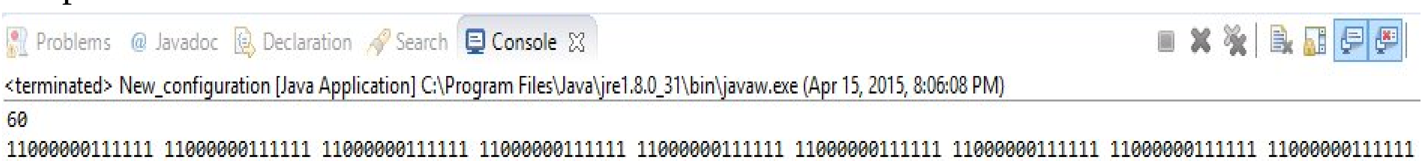


2. **For input_graph_part2.txt and input_ip_part2.txt:**

Input:



Output:



<u>**Conclusion:**</u>

- Successfully implemented Part 1 - Dijkstra’s single source shortest path using the fibonacci heap data structure.
- Successfully implemented Part 2 - Tries and compressed mapping using part 1