COT 5405: Programming Assignment 1 (Spring 2016)

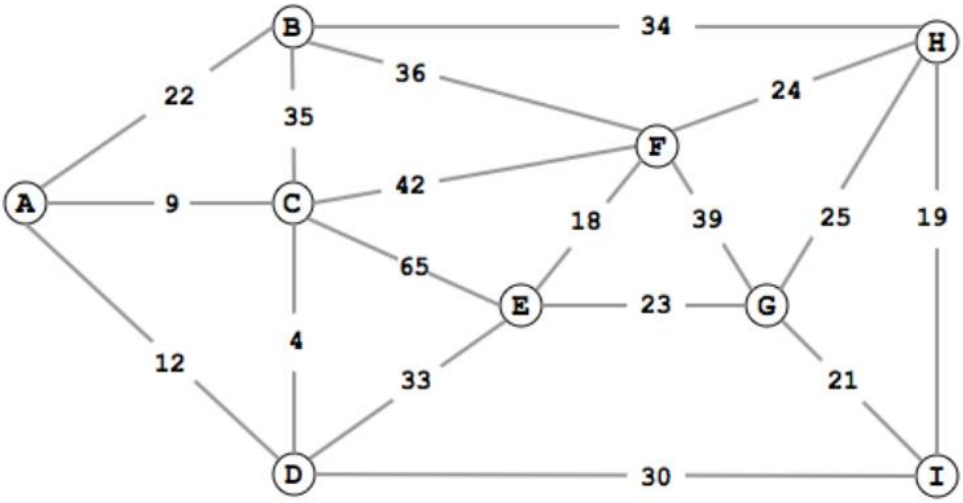
Report by: Sansiri Tarnpradab

Purpose

The purpose of this assignment is to investigate: 1) the impact of adjacency-list input representation on the runtime performance, and 2) the mechanism of three algorithms, namely *Heap sort, Dijkstra's algorithm,* and *Kruskal's algorithm* along with their efficiency in tackling problems and finding a proper solution.

Introduction

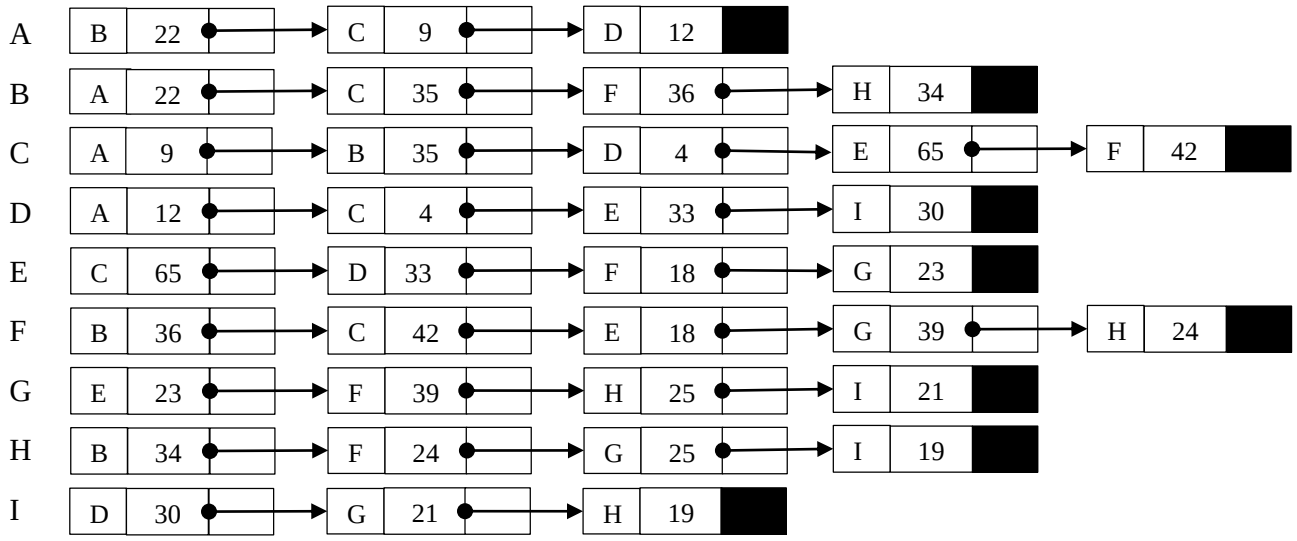
Problem 1: This problem requires to write down the adjacency matrix representation and link list representation of the given graph. Conceptually, the link list representation takes *less* space and time complexity than the adjacency matrix representation; that is, the adjacency list takes up $\Theta(V+E)$ space and identifying all edges takes $\Theta(V+E)$ time, whereas the adjacency matrix takes V^2 space and identifying all edges takes $\Theta(V^2)$ time. Both representations of the given graph are the following.



Adjacency matrix

	A	B	C	D	E	F	G	H	I
A	0	22	9	12	0	0	0	0	0
B	22	0	35	0	0	36	0	34	0
C	9	35	0	4	65	42	0	0	0
D	12	0	4	0	33	0	0	0	30
E	0	0	65	33	0	18	23	0	0
F	0	36	42	0	18	0	39	24	0
G	0	0	0	0	23	39	0	25	21
H	0	34	0	0	0	24	25	0	19
I	0	0	0	30	0	0	21	19	0

Adjacency list



Problem 2: This problem requires to sort all the edge values in the given graph in ascending order using heapsort approach. Heap sort has two major parts: 1) *Build the heap* (Heapify) by placing values in an array with the layout of a complete binary tree. For this assignment, we use the *Max Heap* property, which specifies that all nodes must be greater than or equal to each of its children, according to a comparison predicate defined for the heap. 2) *Sort the values* by repeatedly removing the largest element from the heap or the root of the heap, and inserting it into the array. The heap is updated after each removal to maintain the heap. Once all objects have been removed from the heap, the result is a sorted array. (Please run the program to see the results)

Problem 3: This problem requires to apply Dijkstra's algorithm to find the shortest path starting from node A to all other nodes. Conceptually, for a given source node in the graph, the algorithm finds the shortest path between that node and every other. The algorithm stops once the shortest path to the destination node has been determined. For this assignment, the start node is assigned to be node A; therefore, the Dijkstra's algorithm is applied to find the shortest paths to reach all other nodes including nodes B, C, D, E, F, G, H, and I. The distance between nodes is the edge value. (Please run the program to see the results)

Problem 4: This problem requires to apply Kruskal's algorithm to generate the Minimum Spanning Tree for the given graph. The mechanism is, firstly sort all the edges according the weight values, also for each individual node create a set to store each of them separately, thus resulting in the number of sets equals to the number of nodes. Then, for each sorted edge, check if its corresponding nodes are in the same set. If such nodes are in a different set, merge them into one set and move onto the next edge. This procedure is repeating until the final set number is one, implying that the MST is achieved with the minimum possible total weights on the MST edges. Noted that, throughout the process, any set(s) must be ensured that no cycle exists. (Please run the program to see the results)

Method

- ***Language used***:
  Java programming language

- ***Compilation instructions***:
  - To compile, use this command: **javac Runme.java**
  - To run, use this command: **java Runme argument**

  Where an "argument" to supply is the input text file. A path to the input file is needed in order to properly run this command. For example, the run command will look like this:
  java Runme C:\Users\Sansiri\workspace\Assignment1\Input.txt)

  For simplicity, the input file can be put in the same folder where the code files .java are contained so that the running command **java Runme Input.txt** can be run perfectly fine.

- ***Format of input***:

  The input is a text file. The first line **must** specify the number of nodes/vertices. For example, in the given graph the total number of nodes are 9, containing {A, B, C, D, E, F, G, H, I}. The remaining lines are similar to the adjacency-list representation in that, each line represents the vertex and its adjacency nodes. All the vertices in that same line are separated by space. An example is: A B,22 C,9 D,12 which indicates that A is the start node, B C and D are the adjacency nodes with the corresponding weight of 22 9 and 12 respectively. Each of the adjacency nodes and its weight **must** be comma-separated. Additionally, at the end of each line, **no** extra space is needed. The whole example of the input file looks like the following,

  9

  A B,22 C,9 D,12

  B A,22 C,35 F,36 H,34

  C A,9 B,35 D,4 E,65 F,42

  D A,12 C,4 E,33 I,30

  E C,65 D,33 F,18 G,23

  F B,36 C,42 E,18 G,39 H,24

  G E,23 F,39 H,25 I,21

  H B,34 F,24 G,25 I,19

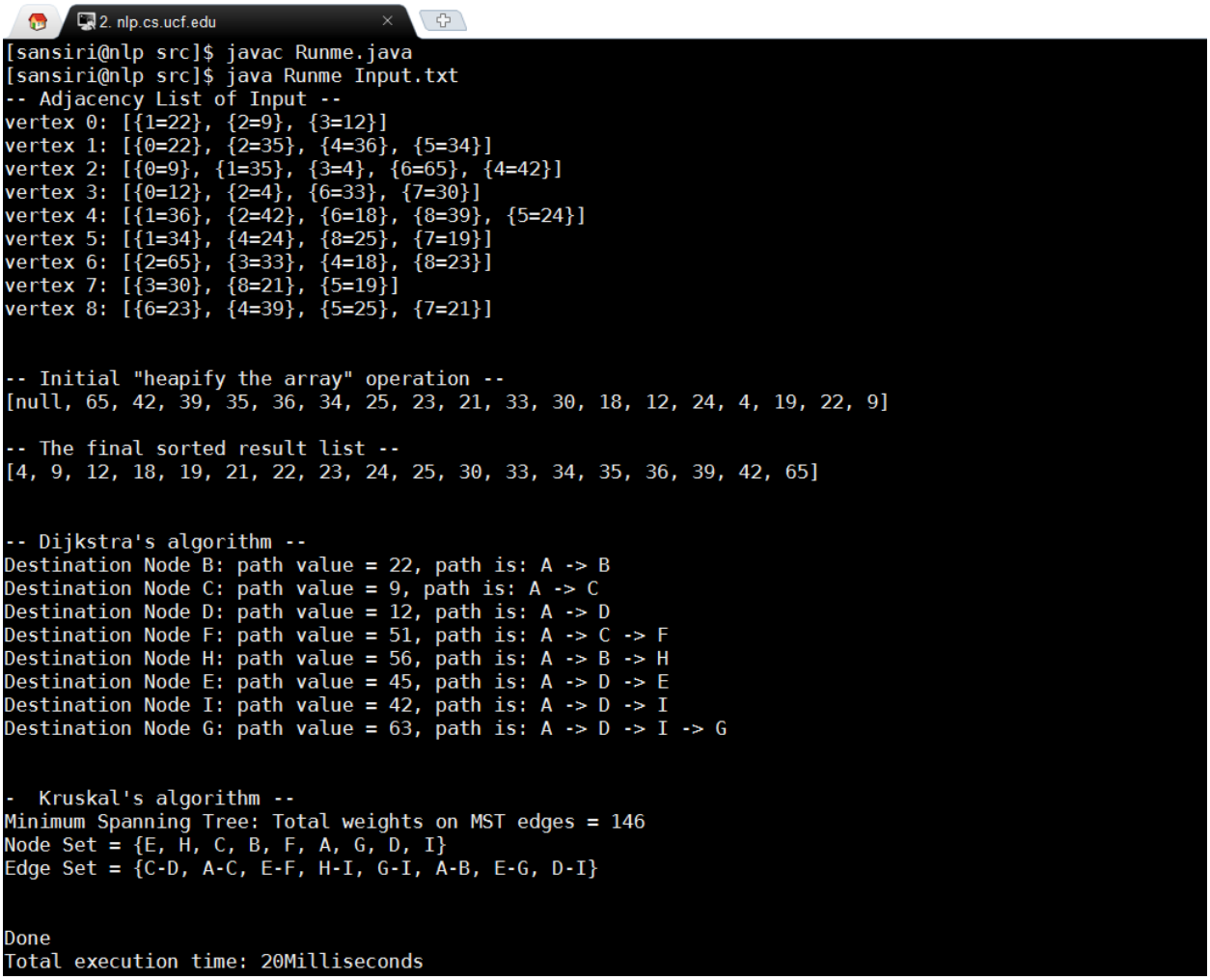
  I D,30 G,21 H,19

  The user can directly input the node name as a character, where each character **must** be of length one. The characters allowed include A-Z, a-z, and 0-9. The program will convert each character into type of integer later for the simpler programmability purpose. Any inputs with the format different from what was described above will cause the program to exit with the printed message "Bad input" to inform the user to recheck the input and try running the program again.

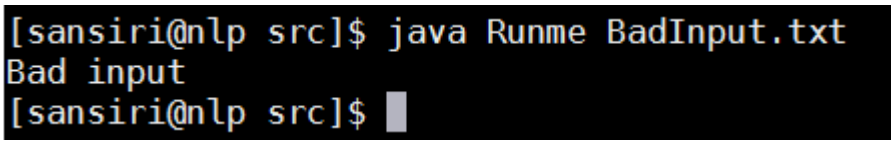
- ***Platform specifics***:
  Windows 8

- ***Algorithms to solve the problems***:
  Three algorithms are applied: Heap sort, Dijkstra's algorithm, and Kruskal's algorithm. Heap sort is used to sort the unsorted list of all the edge values. Dijkstra's algorithm is used to find the shortest path from the source node to all other nodes. Finally, Kruskal's algorithm is used to discover the Minimum Spanning Tree and its corresponding total weight values on MST edges.
- ***Outputs of test input data***:
  The output contains the following,
  1. Adjacency list of input, which displays each vertex as an integer.
  2. The list of edge values before the heapify operation.
  3. The list of edge values after being sorted with the heap sort approach.
  4. The output of the shortest path from source node (A) to all other nodes using Dijkstra's algorithm. The results display the destination node name, path value, and the path.
  5. The Kruskal's algorithm application output, containing the total weights on MST edges, the node set, and the edge set.

Below is the screenshot of the program output with the good input using SSH.



Following is the program output for the bad input using SSH.



- ***Discussions***:

The program includes the code of all three problems including an auxiliary function which prepare inputs for each algorithm, each of which can be discussed for its running time as follows.

- *Function preparing the input* to the linked list representation takes approximately O(VE) where V is the number of vertices and E is the number of edge because for each vertex, the program needs to iterate through each of its adjacent vertices of which the number could be of at most V-1 or E, hence leading to the running time of O(VE).
- *Heap sort algorithm* takes approximately O(n log n) where n here represents each weight value of the edge. This is because Heapify runs in O(log n) time as the heap has O(log n) levels, and the elements bring sifted moves up one level of the tree after a constant amount of work. Based on this, it is noticeable that:
    1. It takes O(n log n) time to build a heap, because we need to apply Heapify roughly n/2 times (to each of the internal nodes)
    2. It takes O(n log n) time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one Heapify.

    Therefore the total running time of Heap sort is O(n log n).

- *Dijkstra's algorithm* application takes O(VE log V) where V is the number of vertices and E is the maximum number of edges attached to a single node. The explanation is:

    1. Each vertex can be connected to (V-1) vertices, hence the number of adjacent edges to each vertex is V-1. Let's say E represents V-1 edges connected to each vertex.
    2. Finding and updating each adjacent vertex's weight is O(log(V)) + O(1) or O(log(V)).
    3. Hence, from step1 and step2, the time complexity for updating all adjacent vertices of a vertex is E*(logV) or E*logV.
    4. Hence, the time complexity for all V vertices is V * (E*logV) or O(VElogV).

- *Kruskal's algorithm* application takes O(E log E) where E is the number of sorted edges. This is because the edges have to be sorted first and it takes O(E log E) where it dominates the runtime for verifying whether the edge in consideration is a safe edge or not which would take O(E log V). Here, noted that I directly used the result generated by Heap sort algorithm which takes O(n log n) that is equivalent to O(E log E) as each n represents the edge E in this particular problem. And |E| > |V|; hence, the runtime for Kruskal's algorithm is O(E log E).

Finally, I made the program compute the total computation time. The result shows that the program's total computation takes roughly between 12 and 16 milliseconds.

- ***Implementation problems***:

  Implementation problems occurred due to the usage of an improper data structure in some files. I later found that such data structure slightly causes some complications when retrieving an input to compute for the results. Particularly, the data structure of type List<Map<Integer, Integer>> is used to store the adjacency list. Following is an overview of how the adjacency of input of the given graph looks like using List<Map<Integer, Integer>> data structure.

```
vertex 0: [{1=22}, {2=9}, {3=12}]
vertex 1: [{0=22}, {2=35}, {4=36}, {5=34}]
vertex 2: [{0=9}, {1=35}, {3=4}, {6=65}, {4=42}]
vertex 3: [{0=12}, {2=4}, {6=33}, {7=30}]
vertex 4: [{1=36}, {2=42}, {6=18}, {8=39}, {5=24}]
vertex 5: [{1=34}, {4=24}, {8=25}, {7=19}]
vertex 6: [{2=65}, {3=33}, {4=18}, {8=23}]
vertex 7: [{3=30}, {8=21}, {5=19}]
vertex 8: [{6=23}, {4=39}, {5=25}, {7=21}]
```

  The generated adjacency list is highly presentable and easy to understand; each row represents the specific node along with its list of adjacent nodes and weights. From the above example, the first row presents vertex 0 (Node A) and its list of adjacent nodes including 1(B), 2(C), 3(D) with the weight of 22, 9, and 12 respectively. However, as already mentioned that the issue is not about storing inputs in this data structure, but retrieving for the computation. Since each of these values is of the *object* datatype, oftentimes it requires additional steps to retrieve the real value. Such complication causes extra processing time which might impact the overall performance especially for the application of Dijkstra's algorithm.

  Another *minor* issue is the conversion of the node representation between *String* and *Integer*. That is, the code was written to support the String formatted input. Therefore, all the nodes that are inputted as String will be indexed as Integer when processed in the program. Given this scenario, whenever there is a need to convert datatype from String to Integer or vice versa, it requires an exclusive function for the conversion be able to return both datatypes. To do this, the generic method is used which also requires an additional Boolean argument to indicate type of an input to function, so that the function would return an output accordingly. It is also important to note that, the time taken to lookup any corresponding output might as well cause extra processing time.

- ***Possible implementation improvements***:

  The issue with data structure can be improved by using List<Node> instead of List<Map<Integer, Integer>>. Here, Node is the object which will store three values including *Node name, Node ID,* and *the list of all its adjacent nodes* (Noted that in the current implementation, there is already an object Node, yet it only contains the Node name and Node ID). With List<Node> to store adjacency list input, supposedly it should help make the query process simpler and the overall computation should be slightly faster as well. Other structures in the program are not really affected.