Report ADS Assignment-1

Performance analysis of Dijkstras algorithm using arrays and Fibonacci heap

Name: Saumitra Aditya **UFID:** 51840391 **email:** saumitraaditya@ufl.edu

Compiler: Netbeans IDE 7.4

System: Intel I-5 and 4 GB RAM.

Compilation and execution

Compiling- **javac -d out -sourcepath dijkstraa dijkstraa/dijikstra.java**

Where out is a directory which will house the compiled class files, dijkstraa is is the directory that contain all the java files and dijikstra.java is the main program of the project. Make sure to create the out directory before firing this command using mkdir out.

Executing- **java dijikstra -r 5000 10 0** / **java dijikstra -s /Desktop/T5.txt** /

java dijikstra -f /Desktop/T5.txt

Once the code has been compiled go to the out directory and fire the execution commands give the path to the input file as an argument.

Classes:

1. **Vertex**
   This class represents a vertex in a graph, it has an ID and a list of edges associated with it, the constructor for this class initializes the ID of the vertex which is either read from a file or randomly generated. Arraylist myEdges is a list containing all the edges associated with this vertex.
   The function retID() returns the ID of the vertex whereas myEdges is a public variable which can be accessed directly.

2. **Edge**
   This class represents an edge in the graph, it has 3 variables namely cost, vertex1 and vertex2 which represent the endpoint of the edges. The constructor initializes the edge given the cost, ID of vertex1 and vertex2. Methods retVertex1(), retVeretex2() and retCost() return the source vertex, destination vertex and cost respectively.

3. **Graph**
   The class graph is composed of an array of vertices 'Varr' , an array list of edges 'E', an array list of vertices 'V'. This class represents the graph as an Adjacency list which can be visualized as an array of vertices with each vertex having a list of associated edges dangling from it.
   Below is the description of the methods of the class
   a. **public Graph(int numV,int numE)**
      The constructor of graph takes the number of vertices and edges as an argument and initializes the array Varr, and the bit set Array which is basically used during addition of new edges to the graph to identify if they are duplicate and prevent their inclusion.

b. **public boolean AddEdge(Edge e)**
given an Edge e, it is added to the list of edges after performing a check that it is not duplicate. Next we extract the vertices of the edge and add it to the list of edges in the corresponding vertices (both the endpoints of the edge must have this edge in their list as we have a undirected graph), this is done by creating a new edge object with appropriate vertex1 and 2. So this where we basically make the adjacency list.

c. **private boolean CheckEdge(Edge e)**
this method uses the bit-set array to determine if an edge is duplicate if yes it returns true else it returns false.

d. **public boolean AddVertexS(Vertex v)**
This method is used to add a vertex to the array Varr. Returns true if it is successful in doing so.

e. **public Vertex[] fetchVerticesS()**
This method returns the array of vertices Varr[] .

f. **public ArrayList<Edge> fetchEdges()**
returns the list of edges E.

g. **retnumV(),retnumE()** return the number of vertices and edges respectively.

h. **public Vertex retVertex(int ID)** returns vertex object associated with the given ID.

i. **public void display()**
prints out the graph in a given format on the console.

4. **RandomData**
This class is used to generate a randomly connected graph given the number of nodes and density, it uses BFS to ensure that the graph is connected. The methods of this class are as declared below.

a. **public RandomData(int numNodes, double density)**
Given the number of nodes and density this method calculates the number of edges that should be present in the graph.

b. **public Graph generateGraph()**
This method initializes a graph object, which in turn prepares the array 'Varr' with the size good enough to accommodate all the vertices. It than makes use of Random class to create edges with random vertices and cost and adds them to the graph. Once the graph is created it calls the BFS method to ensure that the graph is connected if it's not, it repeats the whole process again until it gets a connected graph.

c. **private boolean BFS(Graph G, int r)**
This method runs a breadth first search on the passed graph starting from the initial vertex 'r'. It basically traverses all the nodes in the graph marking them on the way along. Once it is done we can again see if any of the vertex is left unmarked which points to the fact that the graph is not connected.

5. **Reader**
This class contains the functionality to read the given file and return the graph containing the edges and vertices in the graph. The methods of this class are as below.

a. **public Graph readf (String file) throws FileNotFoundException**
This method makes use of the scanner utility to read the file, it initializes a graph object and reads it till the end. While reading the graph it creates vertex and Edge objects along the

way and adds them to the graph. At the end we have constructed a graph which we return to the calling method.

6. **FibHeap**

   This class implements a Min. priority queue using Fibonacci heap, the algorithm used is as described in CLRS. The methods of this class are as below.

   a. **public class Entry**

      This class represents a node in the heap which can be visualized as a singleton doubly linked circular list, the attributes of Entry are –key, degree, marked, references to previous, next, parent and one of the child nodes and the value associated with the node.

   b. **public Entry(int key,Double value)**

      Constructor for the class which creates a new node given an integer key and Double value.

   c. **public FibHeap3(int Maxsize)**

      Constructor method for the heap need to pass the maximum number of elements we want to put in the heap.

   d. **public boolean contains(int key)**

      This method returns true if an element with the passed key is currently present in the heap.

   e. **public void insert (int key, Double value)**

      used to insert a new node in the heap, creates an Entry object and calls the linkLists method to meld this node in the top level list of the heap.

   f. **private Entry linkLists(Entry A, Entry B)**

      This method is used to meld an entry into the top level circular linked list, the parameters passed are the new entry and the known minimum top level entry .melding is carried out in O(1) time by adjusting the references of the list. The value returned is reference to smaller of the two entries.

   g. **public int delMin()**

      This method along with returning the minimum element deletes it from the heap. Next we have to make the min reference point to the current minimum element to this end we will have to traverse the entire top level list. Doing so is not time efficient so instead we start pairwise merging of same degree trees and continue till we have only one tree corresponding to a unique degree while doing so we also update the min pointer.

   h. **private void cascadingCut(Entry entry**)

      Implements the cascading cut functionality, cuts a node from its parent and melds it with the top-level list. If the parent was marked than cuts it so and goes on repeating the process recursively until sees an unmarked node.

   i. **public void decreaseKey(int key, double new_value)**

      Used to reset value of an element with the given key to the new value, if the new value is less than the value of the parent node we do a cascadingCut, if the new value is less than the value of the minimum node we set the min reference to this node.

7. **public class MinPQ**

   Implements a Min Priority queue using an array in which each operation takes O(n) time.

   a. **public class Node**

      The class for node element of the array contains the ID of the node its cost and methods to return ID and cost basically the ID will be VertexID and cost its recorded distance from the source node.

b. **public MinPQ(int maxsize)**
   Constructor for the class, given the maximum number of elements that we want the priority queue to accommodate is used to initialize the array which is used to implement the queue.
c. **public void insert(int ID, Double cost)**
   insert a new vertex into the array given its ID and distance from the source. create a new node to represent the vertex. if the array is initially empty the inserted element will be the first element. else we traverse the array in backward fashion till we get a node whose weight is less than weight of new node we also shift each node to next position to its right to make room for the newly created node now we have the position we insert the newly created node and increment the number of newly created nodes by 1.
d. **public boolean contains(int ID)**
   This method scans the array for a node with given ID if its found it returns true else returns false.
e. **public int delMin()**
   delMin operation takes O(n) time , first node in the array is the minimum node after deleting that node we have to shift all nodes one position left.
f. **public void decreaseKey(int ID, Double cost)**
   in this operation first we find the node whose weight has to be decreased than delete it from the array, shift all element one position left , re-insert the deleted node with the new key.

8. **public class DijkstraSimple/public class DijkstraFib**
   Both these classes are essentially identical except the fact the one uses Array based priority queue and the other Fibonacci heap backed priority queue. The signatures and functionality of all the methods is identical.
   a. **public DijkstraSimple(Graph G, int s) / public DijkstraFib(Graph G, int s)**
      pass the graph and source node to the constructor. Initialize the Dist and Edge Arrays , which have as many entries as the number of vertices in the graph G. initially the distance of all nodes from the source are marked to infinity, indicating that they are unreachable from the source node. the distance of source node from itself is marked as 0. relax vertices in order of distance from s. initialize the priority queue. insert source into PQ. while the queue is not empty we will go on inserting vertices into it than relax all the edges from it.
   b. **private void relax(Edge e)**
      in this method for a given vertex we note down and correct the weight of all of its neighbours, relaxing means correcting if the new known distance of a neighbour node via this node is less than the previously known distance we update it else leave it unchanged this is accomplished using the decreaseKey operation if the node was already in the queue or inserting it if it was not.
   c. **public double Dist(int v)**
      returns the distance of any node from the source.
   d. **public boolean hasPathTo(int v)**
      tells if there is a path from this node to source node.
9. **public class DijkstraA**
   a. **public static void main(String[] args) throws FileNotFoundException**
   This is the main class of the program which decides the flow depending upon the arguments given to the program. Takes the measurement of run times of both the modes and prints them and the output on the console. The functionality is very simple consisting of only if-else constructs and method calls to relevant classes depending upon the flow.

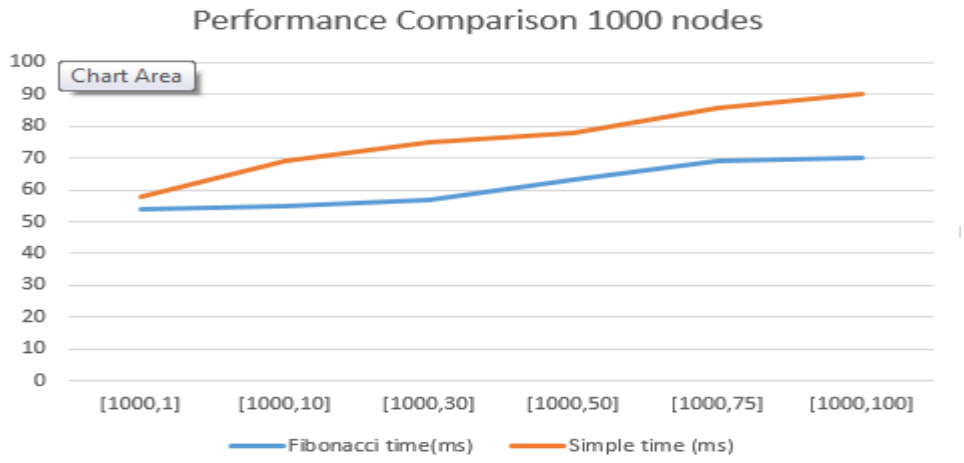
Expectation versus obtained results

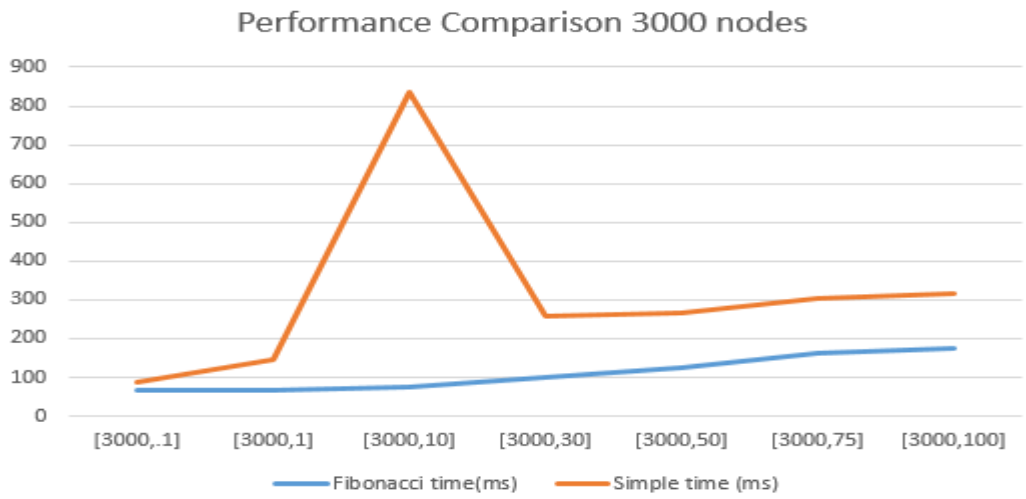
Expectations-

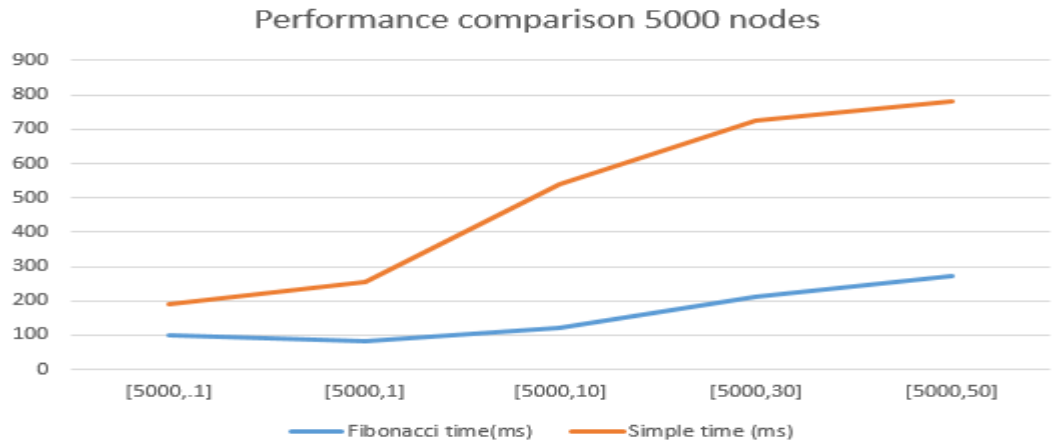
In the simple scheme all the operations take O(n) time , the most frequently used operations in dijkstra are insert, delMin and decreaseKey. The operation delMin is done V times where V is number of vertices, suppose V is equal to n than the complexity would be O(n^2) for it. decreaseKey operation is done E times where E is number of edges. Thus the overall complexity of performing Dijkstras using simple scheme should be O(n^2).

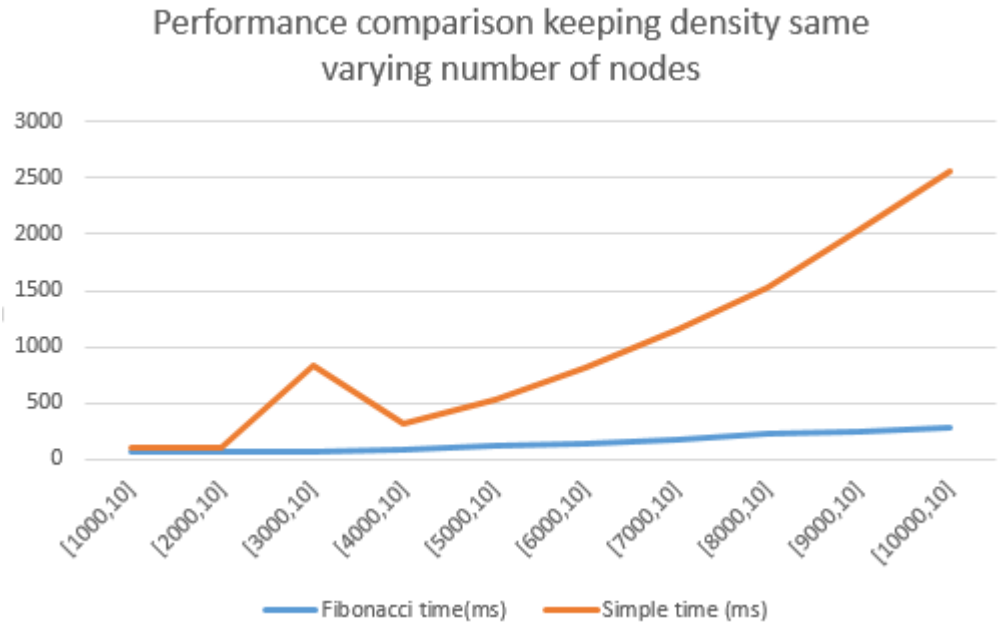
In the Fibonacci heap actual complexity of insert is O(1), delMin is O(n) and decrease key is also O(n) but the amortized complexities are O(1),O(log n) and O(1) respectively. Thus the expected result would be O(V *log V + E*1).

Actual results-









In the actual results obtained Fibonacci heap scheme performs better than the simple scheme. As we go on increasing the number of nodes the performance gap between the simple and Fibonacci scheme also widens I believe this is due to the fact the fact we are better able to exploit the amortized cost of the operations carried out using Fib Heap If you look at the shape of the graph it seems that the simple scheme is following a quadratic curve whereas the Fib scheme is almost close the X-Axis thus we can safely assume that for large values of Vertices simple scheme runs in O(n^2) time and FibHeap in O(V log V + E) time . However when the number of nodes is less the gap is not very large, this might be attributed to the fact that constant factors in this case are very large and cannot be ignored for example as the graph becomes more dense it itself becomes of the order of O(n^2). Other factors because of which Fibonacci heap is not as fast as it could be might be because of Java memory model which is inherently slower than C/C++, a number of constant factors involved might play an important role in this experiment.

For example in the case of [3000,10] where we have 3000 vertices and density is 10% , there is a very prominent spike where the performance of simple scheme suffers a big blow whereas FibHeap remains unaffected. This might actually be the case where simple scheme runs at close to its worst case performance. This is not an irregular result I ran the experiment several times and got matching values, This happens only on Netbeans while running on linux this behaviour is not repeated.

For experiment with 5000 vertices and over I was not able to execute with graphs having density more than 50% as my compiler kept running out of heap-space, This might be due to the fact the Graph is generated several times until we do not arrive at a connected graph (BFS succeeds) also for addition of every Edge to the graph we have to ensure that it is not a duplicate edge. These operations must be carried umpteen times to arrive at a final connected graph free of duplicate Edges thus exhausting the memory. However if it was given a file with 5000 or more nodes with higher densities it is supposed to run fine.