COP 5536 Advanced Data Structures
Spring 2014 Assignment #1 (Programming Project)

Dijkstra's Algorithm Implementation Using Simple Scheme and Fibonacci Heap Scheme Report

Author: Harshini Gottumukkala UFID: 2938-5682 UF Email: harshinig@ufl.edu

1. Working Environment:

1. Hardware Requirement:
- Hard Disk space: Minimum 5GB
- Memory: 512 MB Minimum

2. Operating System:
- Win32 and Above versions like WINDOWS 98/NT/XP/VISTA/7/8
-Linux

3. Compiler:
- Standard Java Compiler (JDK)

2. Compile Process

1.Copy the contents of the folder called "source code" into a separate directory. Set the command line to point to this folder.

2. With above specified compilers, to compile Java file execute command below:

javac dijkstra.java

3. After compiling successfully, compiler will generate .class files for all the classes. To run the main class, execute command below:

java dijkstra parm1 parm2 parm3 parm4

4. Parameters can be given as follows:
   a. **parm1** is the input mode. It can be

   i. **-r** :random input mode
   ii. **-s** :file input mode using simple scheme
   iii. **–f**:file input mode using Fibonacci heap scheme

b**. parm2** can be
i.number of nodes(vertices) for random mode
ii. file name for file input modes (sampleinput1 and sampleinput2 are the two sample files provided)

c. **parm3** is only for random mode and is the density.

d. **parm4** is only for random mode and is source node.

3. Classes Used

Node class: Class defined for the structure of a node of the graph.Contains label of the node and a list to store the adjacent nodes.
AdjacentNode class: Class for the adjacent nodes to be stored in the adjacency list of a node. Contains label and weight of the adjacent node.
FibonacciHeapNode class: Class for the structure of a Fibonacci heap node. Contains the key, value, degree, parent, left sibling, right sibling, child and cut fields
dijkstra class: Class containing the main method and all associated methods

4. Function Prototypes

Function: public static void Add(FibonacciHeapNode f)
Arguments: Pointer to a fibonacci heap node
Return value: void
Description: to add the node to the Fibonacci heap and update pointers
---

Function: public static Insert(int key, int value)
Arguments: key and value to be inserted
Return value: Pointer to Node created
Description: For given key, value pair it creates a FibonacciHeapNode object and calls Add() method to add it to the heap.

Function: public static FibonacciHeapNode DeleteMin ()
Arguments:none
Return value: reference to minimum Fibonacci Heap node
Description: Deletes the root node i.e the node with least key value and returns it. Calls Join() method to join two trees of equal degree and updates min element i.e root.

Function: public static void DecreaseKey(FibonacciHeapNode node,int newKey)
Arguments: Node to be updated and the new key to be inserted into that node
Return value: void
Description: Updates the key value of a given node with a new key value and applies the cascading cut .

Function: public static void Remove(FibonacciHeapNode n)
Arguments:Reference to node to be removed
Return value: void
Description: Removes the given node and its subtree from the heap and updates pointers
---

Function: public static void Join(FibonacciHeapNode node)
Arguments: Reference to node to be joined with root of tree of equal degree
Return value: void
Description: Joins the tree with root as given node with a tree of equal degree by making one tree subtree of the other
---

Function: public static void GenerateRandomGraph(int n, int e)
Arguments: Number of nodes and number of edges
Return value: void
Description: Generates a graph of given number of nodes and edges using Random number generator and adjacency list representation. Calls DepthFirstSearch() method to check if graph is connected. If connected, the variable connected is set to true.

Function: public static void DepthFirstSearch(Node thisNode)
Arguments: Starting node for depth first search
Return value: void
Description: Performs depth first search on given graph in a recursive way

---

Function: public static Boolean EdgeExists(int e1,int e2)
Arguments: the labels of the two nodes connected by an edge
Return value: true or false
Description: Check if an edge exists in a graph and return true if it does, return false if it doesn't.

Function: public static void FileInputGraph()
Arguments: none
Return value: void
Description:Takes number of nodes, source node, number of edges, edges and cost from a file and generates corresponding graph
---

Function: public static void ShortestPath(int length,int source)
Arguments: number of nodes and source node
Return value: void
Description: Computes next shortest distance from source by calling GetNextNode() method and updates the distances of neighbouring nodes. Finally the distance array consists of the shortest distances from all nodes to the source node.

Function: public static void GetShortestPath()
Arguments: none
Return value: void
Description: Computes time taken to execute ShortestPath() and prints the time.

Function: public static void ShortestPathFibonacciHeap(int length,int source)
Arguments: number of nodes and source node
Return value: void
Description: Computes next shortest distance from source by calling DeleteMin() method and updates the distances of neighbouring nodes by calling DecreaseKey() method on neighbouring nodes. Finally the distance array consists of the shortest distances from all nodes to the source node.

Function: public static void GetShortestPathFibonacciHeap()
Arguments: none
Return value: void
Description: Computes time taken to execute ShortestPathFibonacciHeap() and prints the time.

Function: public static void PrintDistance(String m)
Arguments: input mode
Return value: void
Description: Prints the distance array in the file input or random mode.
---

5. Structure of Program

The program begins the execution from the main function. Initially it checks for the input mode specified as command line argument.

1. **If mode= "-r"** ,
   a. Read in the number of nodes,density and source node from the command line.
   b. Call **GenerateRandomGraph()** to generate a random graph and repeat the process until a connected graph is generated. The graph is stored in the array Nodes[].
   c. Call **GetShortestPath()** which calls the **ShortestPath()** function and also prints time taken to execute it. The ShortestPath() function calculates the shortest distance from source node to all nodes for the generated random graph and stores them in the distance array. The distance array is printed by calling **PrintDistance()** function.
   d. Call **GetShortestPathFibonacciHeap()** which calls the **ShortestPathFibonacciHeap()** function and also prints time taken to execute it. The ShortestPathFibonacciHeap() function calculates the shortest distance from source node to all nodes for the generated random graph using a Fibonacci Heap and stores them in the distance array. The distance array is printed by calling **PrintDistance()** function. (To not display distance arrays, comment out the calls to PrintDistance() in the main function in source code.)
   e. The time taken to execute either schemes can now be compared .

2. **If mode= ”-s”** ,

   a.Call **FileInputGraph()** to generate a graph using the nodes and edges from the specified file.

   b. Call **GetShortestPath()** which calls the **ShortestPath()** function and also prints time taken to execute it. The ShortestPath() function calculates the shortest distance from source node to all nodes for the graph and stores them in the distance array. The distance array is printed by calling **PrintDistance()** function.

3. **If mode= ”-f”** ,

   a.Call **FileInputGraph()** to generate a graph using the nodes and edges from the specified file

   b. Call **GetShortestPathFibonacciHeap()** which calls the **ShortestPathFibonacciHeap()** function and also prints time taken to execute it. The ShortestPathFibonacciHeap() function calculates the shortest distance from source node to all nodes for the graph using a Fibonacci Heap and stores them in the distance array. The distance array is printed by calling **PrintDistance()** function.

6. Summary Of Result Comparison

Expectations-

1. The simple scheme uses arrays to store and update the shortest distance values . Searching an array for the shortest distance takes O(n) time and hence, for “n” searches ,it takes O(n^2) time. Complexity of using simple scheme is O(n^2).(where n=number of nodes)

2. The Fibonacci heap scheme uses a Fibonacci heap to store and update the distance values. Order of decreaseKey is O(1) and order of DeleteMin is O(logn) . We do DeleteMin’s for “n” times and decreaseKey for “e” times and hence, complexity of this scheme is O(nlogn+ e). (where n=number of node,e=number of edges)

3.The Fibonacci heap scheme should give a better performance than simple scheme as n increases. For dense graphs, performance will be similar as e=n^2 and performance of Fibonacci heap degrades to O(n^2).
For sparse graphs, the Fibonacci Heap scheme should give a better performance.

Actual Performance-

Performance is measured on Linux AMD FX™-8320 Eight Core Processor(64-Bit). For given number of nodes and density, the table gives time taken to compute shortest paths in simple scheme and in Fibonacci heap scheme.
The values for time taken have been averaged by running each case 10 times and dividing total time in each case by 10.

a.Table Of Results:-

Number of Nodes n, Density d	Time for Simple Scheme (in milli seconds)	Time for Fibonacci Heap Scheme (in milli seconds)
n=1000 , d=0.1%	-	-
n=1000 , d=1%	10.4	20.9
n=1000 , d=50%	37.7	57
n=1000 , d=70%	42	52.3
n=1000 , d=100%	23	37.4
n=3000 , d=0.1%	32.8	58.9
n=3000 , d=1%	33.9	62
n=3000 , d=50%	134	171
n=3000 , d=70%	137	163
n=3000 , d=100%	141.5	167.3
n=5000 , d=0.1%	82.33	124.3
n=5000 , d=1%	103.7	142.9
n=5000 , d=50%	227.1	319
n=5000 , d=70%	314	357.6
n=5000 , d=100%	401.4	437.9

b.The actual results vary slightly from the expected results as the simple scheme is giving better performance than Fibonacci Heap scheme. This may be due to various factors like

i.System processing speed and configuration

ii.The Join operation which is a costly operation is performed for all DeleteMin operations and may slow down the performance of the Fibonacci Heap.

iii.The Cascading Cut operation which is a costly operation is done for all DecreaseKey operations and may slow down the performance of the Fibonacci Heap.

7. Conclusion

The Fibonacci Heap scheme gives a better average performance over simple scheme when implementing the Dijkstra's algorithm. For sparse graphs, the improvement is significant as number of edges is less. For dense graphs , the performance of Fibonacci heap degrades to O(n^2) and is similar to the simple scheme.