Mandatory Project 3: Indexed Heaps

Project Report

Niveditha Varadha Chandrasekaran
11/11/2016

CONTENTS

1. ABSTRACT

The aim of this project is to implement Indexed Priority Queues, Prim's MST algorithm, Dijkstra's Shortest Path algorithm and compare the performance of implementation of Prim's MST algorithm using Priority Queue of edges (using Java's Priority Queues) with implementation of Prim's MST algorithm using Priority Queue of vertices (using Indexed Heaps).

2. PROBLEM STATEMENT

This project aims at implementing the following:

a) Indexed priority queues
b) Prim 1 - Prim's MST algorithm (priority queue of edges; using Java's priority queues)
c) Prim 2 - Prim's MST algorithm (priority queue of vertices, using indexed heaps)
d) Dijkstra's algorithm for shortest paths using indexed heaps

3. METHODOLOGY

a) Prim 1 - priority queue of edges; using Java's priority queues:

Create a priority queue with weight of the edges as priority. Get the start vertex, assign its distance as 0 and mark it as seen and push all the edges from the start vertex into the priority queue. While the priority queue is not empty remove one edge from the queue and check if the vertices belonging to the edge are seen. If both the vertices are seen, then remove the next edge from the queue. If the one of the vertex is not seen then assign the vertex at the other end of the edge as its parent, add the edge weight to the weight of the MST and add all the edges of this vertex into the Priority Queue. Continue this until the Queue is empty. Return the weight of the MST.

b) Prim 2 - priority queue of vertices, using indexed heaps:

Create an Indexed priority queue with Vertex.distance as priority. Get the start vertex; assign its distance as 0. Create an array of vertices and assign initial index to the vertices. Use this array and build an indexed priority Queue. While the Queue is not empty, remove one vertex from the queue, mark it as seen and add its distance with the weight of MST. For all edges from that vertex u, get the other end vertex v. If v is not seen and edge weight is less than v.distance then assign weight of the edge to v.distance and make u as v.parent and perform decrease key operation on v (to assign correct index). Repeat this until the Queue is empty. Return the weight of the MST.

c) Dijkstra's algorithm for shortest paths using indexed heaps:

Create an Indexed priority queue with Vertex.distance as priority. Get the start vertex; assign its distance as 0. Create an array of vertices and assign initial index to the vertices. Use this array and build an indexed priority Queue. While the Queue is not empty, remove one vertex from the

queue, mark it as seen. For all edges from that vertex u, get the other end vertex v. If v is not seen and u.distance plus edge weight is less than v.distance then assign weight of the edge plus u.distance to v.distance and make u as v.parent and perform decrease key operation on v (to assign correct index). Repeat this until the Queue is empty.

4. DEVELOPMENT PLATFORM

IDE: Eclipse. Version: Mars.1 Release (4.5.1)
Java Version: 1.8
Operating System: Windows 10

5. SAMPLE INPUT/OUTPUT

Sample input:

5 7
1 5 8
1 4 7
1 3 6
4 3 3
3 5 6
5 3 2
5 2 1
1 2

Input Explanation:

Last line of input: s = 1, t = 2.
Prim's algorithm is run with source as s. Dijkstra's algorithm is run with s as source and distance to t is printed. A second optional parameter, given in the command line, controls which program is called, and whether verbose output is printed.

Sample output:

MST: 12
Distance: 1
Time: 8 msec.
Memory: 1 MB / 123 MB.

6. ANALYSIS OF RESULT

Running Time analysis Prim 1 and Prim 2 on the inputs:

Prim1: Priority queue of edges; using Java's priority queues Prim2: Priority queue of vertices; using indexed heaps			
g1.txt			
	Running time (ms)	**Memory Used(MB)**	**Memory Total(MB)**
Prim1	8	1	123
Prim2	9	1	123
g2.txt			
	Running time (ms)	**Memory Used(MB)**	**Memory Total(MB)**
Prim1	9	2	123
Prim2	19	2	123
g3.txt			
	Running time (ms)	**Memory Used(MB)**	**Memory Total(MB)**
Prim1	27	3	123
Prim2	20	3	123
g4-big.txt			
	Running time (ms)	**Memory Used(MB)**	**Memory Total(MB)**
Prim1	11171	802	1327
Prim2	1550	340	943

From the analysis, it is clear that, for dense graphs with million vertices and |V| * |V| edges, Prim 2 which is implemented by Priority Queue of vertices using Indexed Heap is more efficient and runs faster than Prim 1 which is implemented by Priority Queue of Edges using Java's Priority Queue.

The analysis shows that:

- The Performance of both Prim 1 and Prim 2 is almost the same for small graphs and moderately big graphs which is not dense.
- But as the graph size grows and becomes dense, the performance both in terms of memory and run time of Prim 2 (Priority Queue of vertices using Indexed Heap) is more efficient than that of Prim 1 (Priority Queue of Edges using Java's Priority Queue).

7. CONCLUSION

Indexed Priority Queue, Prim 1, Prim 2 and Dijkstra's Algorithm were implemented. For large and dense graphs it can be inferred that Performance of implementation of Prim's algorithm with Priority Queue of vertices using Indexed Heap is more efficient than that of Prim's algorithm with Priority Queue of Edges using Java's Priority Queue. Thus appropriate use of data structures and efficient usage indexes plays a key role in getting efficient running times as the input size increases and the graph becomes denser.

8. REFERENCES

- https://en.wikipedia.org/wiki/Priority_queue