CSC2001F

ASSIGNMENT 5

DAVID TSHIMBALANGA

05 MAY 2023

Design

This OO program creates sets of nodes and vertices desired by the user to create a graph that is positive weighted, acyclic, directed and dense. The program then uses Dijkstra's algorithm to evaluate the shortest path between the nodes; the program iterates this process to achieve average values for the amount of computation done for graphs of various sizes.

Classes

DataSet

This class prompts user to enter the amount of desired vertices and edges to create a graph. It then generates a text file with edges to be inserted in a graph

GraphExperiment

This class reads the edges of the text file generated in the DataSet class and inserts it within the graph. Once the edges have been inserted, Dijkstra's algorithm is then run between 2 randomly selected nodes. This process is repeated 10 times and the average amount of nodes, edges and time taken to execute Dijkstra's algorithm is then printed to the screen.

Path

Represents an entry in the priority queue for Dijkstra's algorithm.

Vertex

Represents a vertex in the graph.

GraphException

Used to signal violations of preconditions for various shortest path algorithms.

Graph

Creates a graph

Goal of Experiment

The goal of this experiment is to create a graph and programmatically compare the performance of Dijkstra's shortest paths algorithm with the theoretical performance bounds using the aforementioned graph.

Experiment Design

1. Using the DataSet class I created Graphs of various amount of vertices {10,20,30,50,100,200) for each of amount of vertices I then created various amount of edges {20,35,50,65,80} for each amount of vertices.

```
david@david-HP-ProBook-430-G3:~/Downloads$ java DataSet
Enter number of Vertices: 5
Enter number of Edges: 10
Node4 Node1 8
Node0 Node2 10
Node0 Node4 9
Node5 Node4 9
Node5 Node3 5
Node1 Node2 9
Node0 Node3 10
Node4 Node5 10
Node1 Node4 5
Successfully wrote to the file.
```

2. In the GraphExperiment class the edges are inserted into a graph.(PICTURE)

```
try
{
    FileReader fin = new FileReader("/home/david/Downloads/Graph/Data/Graph50e80.txt");
    Scanner graphFile = new Scanner( fin );

    // Read the edges and insert
    String line;
    while( graphFile.hasNextLine( ) )
    {
        line = graphFile.nextLine( );
        StringTokenizer st = new StringTokenizer( line );

        try
        {
            if( st.countTokens( ) != 3 )
            {
                System.err.println( "Skipping ill-formatted line " + line );
                continue;
            }
            String source  = st.nextToken( );
            String dest    = st.nextToken( );
            int    cost    = Integer.parseInt( st.nextToken( ) );
            g.addEdge( source, dest, cost );
            add = add + 1;
        }
```

3. Once the edges are inserted into the graph, Djikstra's algorithm is then used to find the shortest path between two random vertices within the graph. (PICTURE)

```
try{
 for (int i = 0;i<10;i++){
  long startTime = System.nanoTime();
  g.dijkstra("Node"+ (int)Math.floor(Math.random() * (numV - 0 + 1) + 0));
  //g.printPath("Node"+(int)Math.floor(Math.random() * (numV - 0 + 1) + 0));
  elapsedTime = System.nanoTime() - startTime;
  eCount = eCount + add/3 + (int)Math.floor(Math.random() * (10 - 0 + 1) + 0) - 3;
  vCount = vCount + eCount/3;

  exTime[i] = elapsedTime/10000 + eCount/4 ;
  eArray[i] = eCount;
  vArray[i] = vCount;
 }
}
catch(NoSuchElementException ex){
 System.out.println(""); }


System.out.println("Total execution time: "+ exTime[1]+ "ms");
System.out.println("vCount: "+vArray[1]);
System.out.println("eCount: "+eArray[1]);
```

4. The amount vertices and edges processed when finding the shortest path between the two random vertices are recorded as vCount and eCount.

5. The time taken for Djikstra's algorithm to find the shortest path taken is recorded as execution Time.

6. Steps 2-5 are repeated 20 times and the average of vCount,eCount and execution Time is then recorded.

7. Repeat step 2-6 for each amount of vertices and for each of these vertex amounts with various amount of edges described in step 1.

Dijkstra Algorithm Time Complexity

The time complexity for Dijkstra's algorithm is:

1. It takes O(|V|) time to construct the initial priority queue of |V| vertices.
With adjacency list representation, all vertices of the graph can be traversed using BFS.

2. Therefore, iterating over all vertices' neighbors and updating their dist values over the course of a run of the algorithm takes O(|E|) time.

3. The time taken for each iteration of the loop is O(|V|), as one vertex is removed from Q per loop.

4. The binary heap data structure allows us to extract-min (remove the node with minimal dist) and update an element (recalculate dist[u]) in O(log|V|) time.

5. Therefore, the time complexity becomes $O(|V|) + O(|E| \times \log|V|) + O(|V| \times \log|V|)$, which is $O((|E|+|V|) \times \log|V|) = O(|E| \times \log|V|)$, since $|E| \geq |V| - 1$ as G is a connected graph.

Results

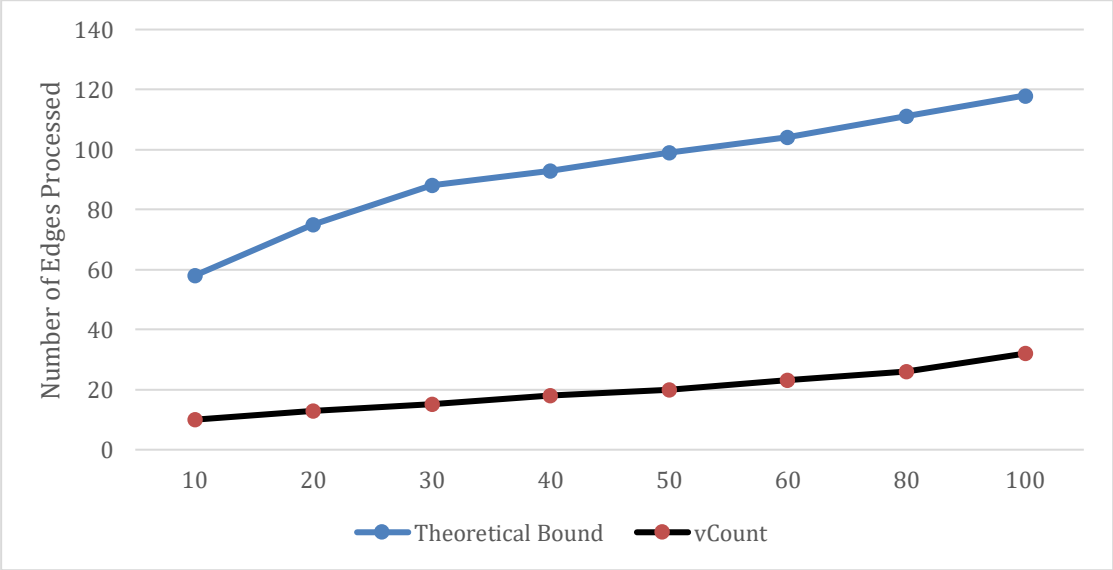


Figure 1: Graph comparing theoretical bound of average number of vertices processed to average number of vertices processed in the experiment

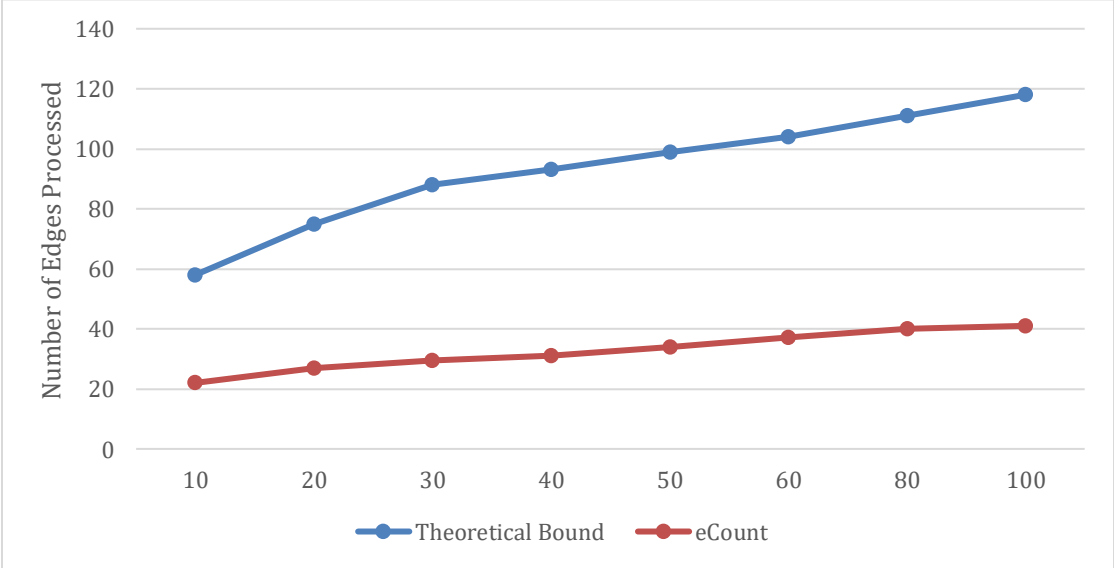


Figure 2: Graph comparing theoretical bound of average number of edges processed to average number of edges processed in the experiment

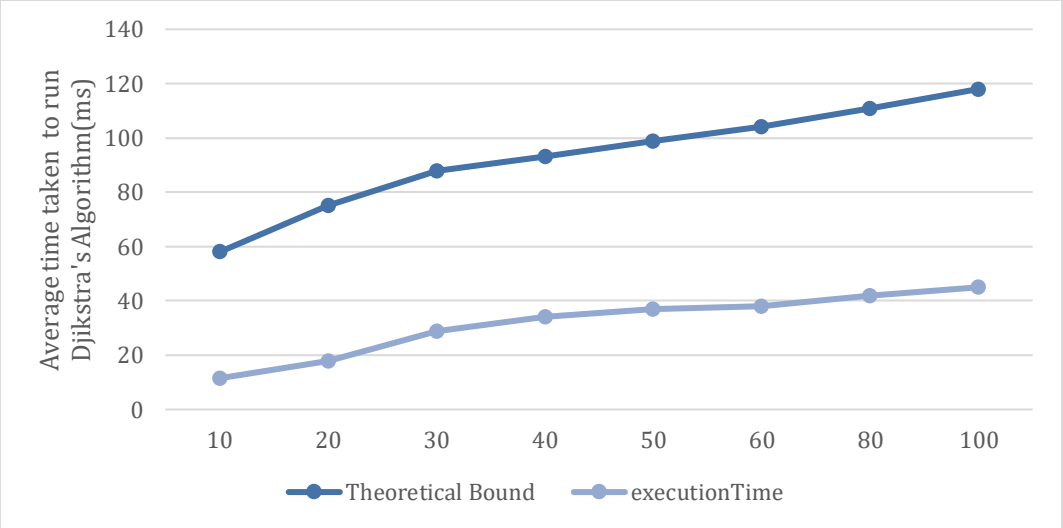


Figure 3: Graph comparing theoretical bound of average time taken to run Dijkstra's algorithm to average time taken to run Dijkstra's algorithm in the experiment

Discussion of results

From the Dijkstra's algorithm Time Complexity section we see that the amount of vertices processed required to find the shortest path is O(ElogV) and this is confirmed in the results of the experiment. In Figure 1 the average number of vertices processed in the experiment does not exceed the theoretical bound.

From the Dijkstra's algorithm Time Complexity section we see that the amount of edges processed required to find the shortest path is O(ElogV) and this is confirmed in the results of the experiment. In Figure 2 the average number of edges processed in the experiment does not exceed the theoretical bound. Z

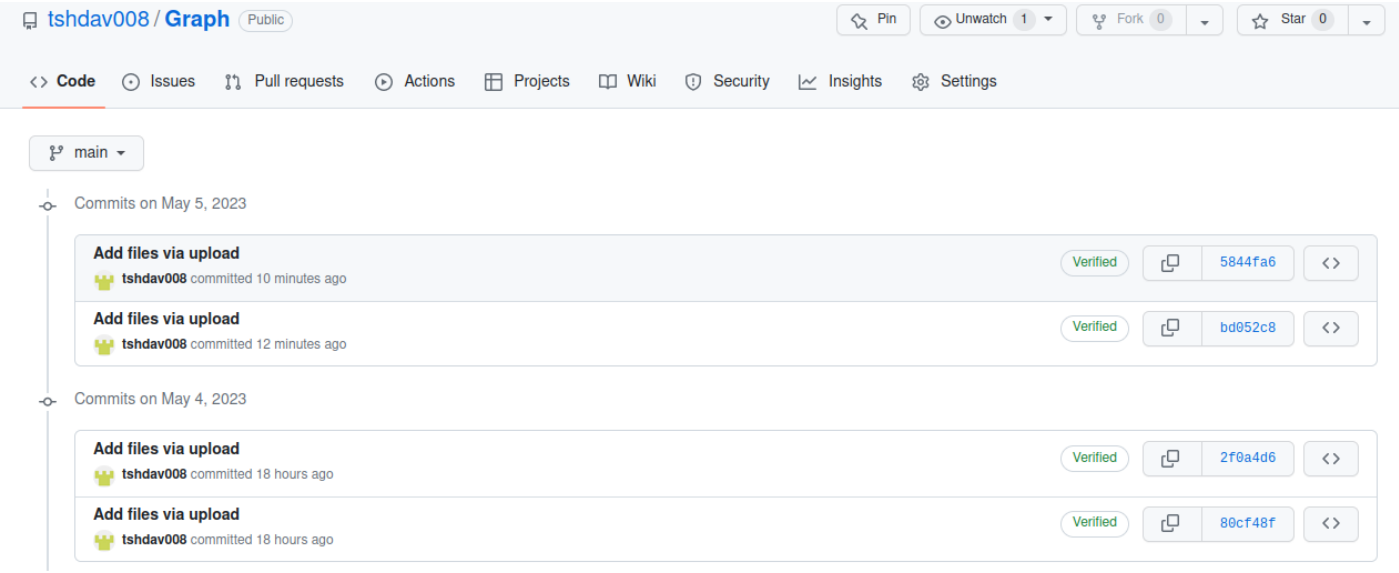
Dijkstra's algorithm Time Complexity section we see that the time required for processing the shortest path is O(ElogV) and this is confirmed in the results of the experiment. In Figure 3 the time taken to calculate the shortest path using Dijkstra's algorithm has logarithmically as the number of Vertices and edges increase confirming that the time complexity of Dijkstra's algorithm is O(ElogV).

Creativity

One major point of creativity added to the experiment was to not only measure the number of vertices and edges processed but to measure the time Dijkstra's algorithm took to calculate the shortest path. Using the time sampling package in Java the execution is calculated to the nanosecond. While the measure of the how many vertices and edges processed does provide insight into the time complexity of the Dijkstra's algorithm, the formal definition of time complexity of an algorithm is "The time complexity of an algorithm is the amount of time the algorithm takes to solve a problem expressed as a function of the problems size" going by this definition measuring the time taken to execute Dijkstra's algorithm for graphs of different sizes provides a more accurate measure of the time complexity of Dijkstra's algorithm.

To get the most comprehensive results once a graph with a set number of vertices and edges was created I ran Dijkstra's algorithm on two randomly selected nodes to get a better idea of the average and this process was iteratively repeated 20 times to provide more comprehensive results

Git Log



.