CS39202 - Database Management Systems Lab

Indian Institute of Technology Kharagpur

Department of Computer Science and Engineering

Term Project - Large Scale Graph Processing using

Authors:
B Aarsha Sai (20CS10016)
Manami Mondal (20CS10033)
Ch Krishna Venkat (20CS10019)
N Surya Prakash Reddy (20CS10038)
Virinchi Mourya Peddireddy (20CS10075)

Date: April 16, 2023

1 Objective

The objective of this project is to take a large graph from the SNAP repository for graphs, process them and allow running simple queries. The queries we implemented are

- Count: Gives the number of nodes and the number of edges in the graph.
- Neighbors: Given a node, returns the neighbors of that node as a list.
- Triangle count: A triangle is a set of three nodes that are connected to each other. This query takes a node as input and returns the number of triangles that this particular node is involved in.
- : Page Rank: Page rank is the order of probability that starting from any node and ending up at a particular node.
- Static Page Rank: The user is allowed to choose the number of iterations that the page rank algorithm is run. Using this, the algorithm gives the top 10 ranked nodes.
- Dynamic Page Rank: The user is allowed to choose a convergence value, reaching which the algorithm concludes. Convergence is said to be achieved when the sum of differences between the probability values in the current step and previous step is less than or equal to the convergence value.
- Shortest Path: Returns the shortest distance between two nodes that are taken as user input.
- Connected Components: The number of components that are connected, i.e, each pair of nodes in the component has a path between each other.
- Strongly Connected Components: The number of components that are strongly connected, i.e, each pair of nodes in the component has a directed path between each other.

2 Methodology

2.1 Count

For getting the count of nodes and neighbors, we access the nodes and edge members of the graph class.

```
case "count" =>
  println(s"Number of vertices: ${graph.vertices.count()}")
  println(s"Number of edges: ${graph.edges.count()}")
```

Figure 1: Count code

2.2 Neighbors

For the Neighbors, we filter the edges in the graph which have the source as the given node.

```
case "neighbors" =>
  val vertexId = scala.io.StdIn.readLine("Enter vertexId: ").toLong
  //check if vertex exists
  if (graph.vertices.filter(v => v._1 == vertexId).count() == 0) {
    println("Vertex does not exist")
  } else {
    //get neighbors
    //start time
    val startTime = Calendar.getInstance().getTimeInMillis
    val neighbors = graph.edges.filter(e => e.srcId == vertexId).map(_.dstId).collect()
    //end time
    val endTime = Calendar.getInstance().getTimeInMillis
    //print time taken
    println("Time taken: " + (endTime - startTime) + " ms")
    println(s"Neighbors of $vertexId: ${neighbors.mkString("(", ", ", ")")}")
  }
```

Figure 2: Neighbors code

2.3 Triangle

For finding triangles in the code, we use the TriangleCount method and run it with the graph as input and select the number of triangles.

```
case "triangles" =>
  //triangle count with graphx
  //start time
  val startTime = Calendar.getInstance().getTimeInMillis
  val numTriangles = TriangleCount.run(graph).vertices.map(_._2).reduce(_ + _) / 3
  //end time
  val endTime = Calendar.getInstance().getTimeInMillis
  //print time taken
  println("Time taken: " + (endTime - startTime) + " ms")
  println(s"Number of triangles in the graph: $numTriangles")
```

Figure 3: Triangles code

2.4 Static Page Rank

For Static page rank, we set the number of iterations in the PageRank object and run with the graph as the input. Then we choose the first 10 vertices in order of probabilties.

```
case "pagerank" =>
  //input number of iterations
  val numIterations = scala.io.StdIn.readLine("Enter number of iterations: ").toInt
  //start time
  val startTime = Calendar.getInstance().getTimeInMillis
  val pageRankGraph = PageRank.run(graph, numIterations)
  val top10 = pageRankGraph.vertices.top(10)(Ordering.by(_._2))
  //end time
  val endTime = Calendar.getInstance().getTimeInMillis
  //print time taken
  println("Time taken: " + (endTime - startTime) + " ms")
  println("Top 10 vertices by PageRank:")
  top10.foreach { case (id, rank) => println(s"$id\t$rank") }
```

Figure 4: Static Page Rank code

2.5 Dynamic Page Rank

For dynamic page rank, we set the tolerance value and run the pageRank object's run until convergence method so that we only stop when the sum of changes in probs is less than tolerance.

```
case "dpr" =>
  //input tolerance
  val tolerance = scala.io.StdIn.readLine("Enter tolerance: ").toDouble
  //start time
  val startTime = Calendar.getInstance().getTimeInMillis
  val pageRankGraph = PageRank.runUntilConvergence(graph, tolerance)
  val top10 = pageRankGraph.vertices.top(10)(Ordering.by(_._2))
  //end time
  val endTime = Calendar.getInstance().getTimeInMillis
  //print time taken
  println("Time taken: " + (endTime - startTime) + " ms")
  println("Top 10 vertices by PageRank:")
  top10.foreach { case (id, rank) => println(s"$id\t$rank") }
```

Figure 5: Dynamic Page Rank code

2.6 Shortest Path

For finding the shortest paths, we

- use the ShortestPaths object method with the destination vertices set to the destination given by the user and run with the graph as input.
- We then filter the paths and take the path which matches the destination node and print the shortest distance between source and destination.

```
case "sssp" =>
  //input source and destination vertex
  val source = scala.io.StdIn.readLine("Enter source vertex: ").toLong
  val destination = scala.io.StdIn.readLine("Enter destination vertex: ").toLong
  //check if source and destination are valid
  if (graph.vertices.filter(v => v._1 == source).count() == 0) {
    println("Source vertex does not exist")
  } else if (graph.vertices.filter(v => v._1 == destination).count() == 0) {
    println("Destination vertex does not exist")
  } else {
    //start time
    val startTime = Calendar.getInstance().getTimeInMillis
    //run sssp
    val sssp = ShortestPaths.run(graph, Seq(destination))
    //end time
    val endTime = Calendar.getInstance().getTimeInMillis
    //print time taken
    println("Time taken: " + (endTime - startTime) + " ms")
    //print shortest path
    val path = sssp.vertices.filter(v => v._1 == source).map(_._2).collect()(0)
    println(s"Shortest path from $source to $destination: ${path(destination)}")
  }
```

Figure 6: Shortest Path code

2.7 Connected Components

For connected components, we call the ConnectedComponents object and print the number of connected components in the graph.

```
//connected components
case "cc" =>
  //start time
  val startTime = Calendar.getInstance().getTimeInMillis
  //run connected components
  val cc = ConnectedComponents.run(graph, 10).vertices
  //end time
  val endTime = Calendar.getInstance().getTimeInMillis
  //print time taken
  println("Time taken: " + (endTime - startTime) + " ms")
  //print number of connected components
  println("Number of connected components: " + cc.map(_._2).distinct().count())
  //print largest connected component
  val largestCC = cc.map(_._2).countByValue().maxBy(_._2)._1
  println("Largest connected component: " + largestCC)
  //print number of vertices in largest connected component
  println("Number of vertices in largest connected component: " + cc.filter(_._2 == largestCC).count())
```

Figure 7: Connected Components code

2.8 Strongly Connected Components

For connected components, we call the StronglConnectedComponents object and print the number of connected components in the graph.

```
//strongly connected components
case "scc" =>
  //start time
  val startTime = Calendar.getInstance().getTimeInMillis
  //run strongly connected components
  val scc = StronglyConnectedComponents.run(graph, 10).vertices
  //end time
  val endTime = Calendar.getInstance().getTimeInMillis
  //print time taken
  println("Time taken: " + (endTime - startTime) + " ms")
  //print number of strongly connected components
  println("Number of strongly connected components: " + scc.map(_._2).distinct().count())
  //print largest strongly connected component
  val largestSCC = scc.map(_._2).countByValue().maxBy(_._2)._1
  println("Largest strongly connected component: " + largestSCC)
  //print number of vertices in largest strongly connected component
  println("Number of vertices in largest strongly connected component: " + scc.filter(_._2 == largestSCC).count())
```

Figure 8: Strongly Connected Components code

3 Results

3.1 Count

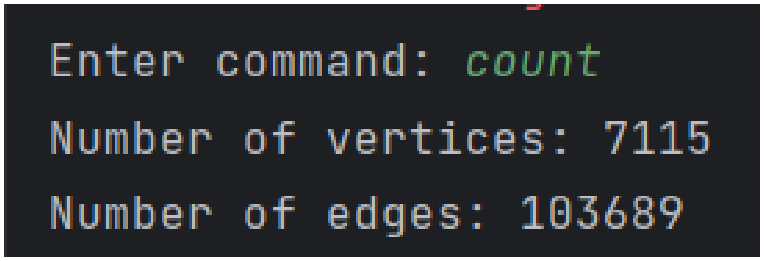

Figure 9: Count results

3.2 Neighbors

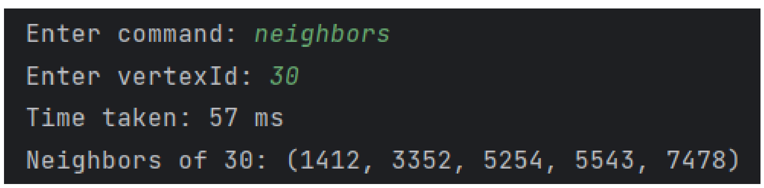

Figure 10: Neighbors results

3.3 Triangle

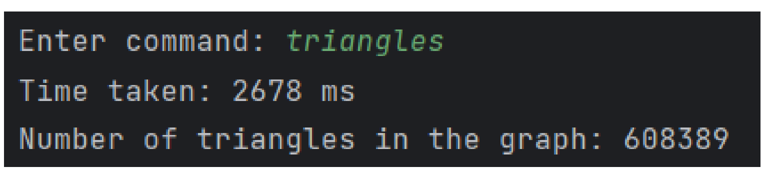

Figure 11: Triangle results

3.4 Static Page Rank

```
Enter command: pagerank
Enter number of iterations: 10
Time taken: 1744 ms
Top 10 vertices by PageRank:
4037    32.76139259035081
15  26.25300495761945
6634    26.16452443488649
2625    23.511515933026356
2398    18.728389390669694
2470    17.838985178368716
2237    17.708040334723627
4191    16.223671535354253
7553    15.511778549594181
5254    15.350069106265575
```

Figure 12: Static Page Rank results

3.5 Dynamic Page Rank

```
Enter command: dpr
Enter tolerance: 0.0001
Time taken: 3264 ms
Top 10 vertices by PageRank:
4037    32.780742393891785
15  26.181746574767544
6634    25.51855014072694
2625    23.36100468516942
2398    18.559437057562366
2470    17.957604768296473
2237    17.76401205997492
4191    16.135404511532666
7553    15.436932186578405
5254    15.297497713728962
```

Figure 13: Dynamic Page Rank results

3.6 Shortest Path

```
Enter command: sssp
Enter source vertex: 30
Enter destination vertex: 4191
Time taken: 1526 ms
Shortest path from 30 to 4191: 2
```

Figure 14: Shortest Path results

3.7 Connected Components

```
Enter command: cc
Time taken: 1406 ms
Number of connected components: 24
Largest connected component: 3
Number of vertices in largest connected component: 7066
```

Figure 15: Connected Components results

3.8 Strongly Connected Components

```
Enter command: scc
Time taken: 3659 ms
Number of strongly connected components: 5816
Largest strongly connected component: 3
Number of vertices in largest strongly connected component: 1300
```

Figure 16: Strongly Connected Components results

4 References

Large Scale Graph Processing Book