FACULTY OF ENGINEERING

Incremental calculation of shortest path in dynamic graphs

Distributed Systems

Presented by
1- Alaa Shehab 1
2- Bassant Ahmed 18
3- Rita Samir 25

4/6/2020

[This document contains the design and implementation of dynamic graph using RMI that allows multiple users to modify and query the graph in a thread safe environment]

Table of Contents

1 Problem Definition

It is required to implement a dynamic graph, allowing remote clients to submit batches, of both graph queries and updates, which are handed as a workload to the server.

The server is expected to process all batches concurrently and return query results to clients, which in turns require the server to accept more than one user at a time.

Requirements to be implemented:

- Implement two variants for the dynamic graph
- Add concurrency to maintain synchronization between batches
- Implement RMI to allow remote client batches
- Performance analysis
- Stress test for server to ensure its stability and that is provides reasonable response time for many clients

2 Algorithms

1. Dynamic graph version 1

   Reading edges from file and storing them in an adjacency list.
   Implement BFS using Queue for shortest path queries.

   Method : Add Edge

   IF Edge exist between v1 and v2 then return

   IF graph doesn't have v1 then
   graph.add(new list, v1)
   graph.get(v1).add(v2)

   IF graph doesn't have v2 then
   graph.add(new list, v2)

   Method : Delete Edge

   IF No edge exist between v1 and v2 then return

   graph.get(v1).remove(v2)

   Method : Find Shortest path

   queue <= first node of query

   WHILE queue is not empty DO
   head <= queue.dequeue()
   IF head == destination THEN return head.level
   List neighbours <= queue.get(head)
   FOR neighbour in neighbours DO
   IF neighbour is not visited then
   queue <= neighbour

2. Dynamic graph improved version
   Reading edges from file and storing them in a hashMap to decrease look up time O(1)
   and each vertex has a hashSet of neighbouring vertices

Method : Add Edge

```
IF   graph doesn't have v1 then
          graph.add(v1)
 graph.get(v1).add(v2)
```

Method : Delete Edge

```
IF   Graph does not contain v1  then
           return
graph.get(v1).remove(v2)
```

Method : Find Shortest path

```
queue <= first node of query

WHILE  queue is not empty DO
          head <= queue.dequeue()
          IF head == destination THEN return head.level
          Set neighbours <= queue.get(head)
          FOR neighbour in neighbours DO
                    IF neighbour is not visited then
                              queue <= neighbour
```

3. Concurrency

To make the graph concurrent with multiple clients applying different operations on it, we implemented the concurrency as in reader writer problem which is described as follows:

- When a client wants to find a route in the graph

where rc denotes the number of the clients who want to find a route in the graph and wrt is a mutex responsible for allowing a client to edit the graph.

```
wait (mutex);
rc ++;
if (rc == 1)
  wait (wrt);
signal(mutex);
...  FIND THE ROUTE
wait(mutex);
rc --;
if (rc == 0)
  signal (wrt);
signal(mutex);
```

- When a client wants to edit the graph

```
wait(wrt);
.
.  EDIT THE GRAPH
.
signal(wrt);
```

4. RMI
    a. Defining a remote interface (Graph.java)
    b. Implementing the remote interface (Server.java)
    c. Creating Stub and Skeleton objects from the implementation class using rmic (rmi compiler) (Client.java)
    d. Start the rmiregistry
    e. Create and execute the server application program
    f. Create and execute the client application program.

Server

- Export object of the server object (stub)
- Bind the remote object's stub in the registry with the binding name
- Initialize the graph and display the letter R

Client

- The main method accepts two arguments the IP of the host server (which also has the registry) and the path to the batch file of this client
- All the operations that are done in the main method for the client run on new thread as many clients can access the server simultaneously.
- For each client, get the registry and get the object stub by looking up the binding name

- Read the batch file to know which operation to run and then run this operation using the stub object.

4 Implementation

- Environment
  - Java Programming
  - Windows/Linux Platform
- Machines Specifications
  - Architecture: x86_64
  - System type: 64-bit operating system
  - Core(s) :5
  - CPU family: Intel
  - Core Generation: 8th Gen
  - CPU GHz: 1.6
- Network Type
  - Wireless LAN
- Data sets
  - Sigmoid competition dataset (to differentiate clearly between two graph variants)
    - 1,000,000 edge graph
    - 10,000 operation and query
- Runs
  - Number: 28 run performed
  - Min clients: 1
  - Max clients: 20
  - Operations: 35% add, 35% query, 30% delete
- Log file
  - server log file starts with client followed by the client IP and the client's action and at last the state of this action.

```
Client RMI TCP Connection(2)-192.168.1.4 added edge from 1 to 2 Successfully
Client RMI TCP Connection(2)-192.168.1.4 added edge from 2 to 3 Successfully
Client RMI TCP Connection(2)-192.168.1.4 added edge from 3 to 1 Successfully
Client RMI TCP Connection(2)-192.168.1.4 added edge from 4 to 1 Successfully
Client RMI TCP Connection(2)-192.168.1.4 added edge from 2 to 4 Successfully
The Graph was intialized Successfully
Client RMI TCP Connection(20)-192.168.1.4 added edge from 4 to 5 Successfully
Client RMI TCP Connection(11)-192.168.1.4 queried the shortest path from 5 to 1 Successfully
Client RMI TCP Connection(15)-192.168.1.4 queried the shortest path from 1 to 5 Successfully
Client RMI TCP Connection(24)-192.168.1.4 queried the shortest path from 1 to 3 Successfully
```

5 Results

All the following tests were performed on a 1,000,000 edge graph

Variant	Improved	Unimproved
Running 1500 Add/Query operation On 1,000,000 edge graph	13 minutes	15 minutes

Observation: Improved indeed is faster in case of having big workload

Frequency	5	10	20	5	10	20
Variant	Improved			Unimproved		
response/Client	2.4 ms	3.2 ms	2.3 sec	3.2 ms	6.8 ms	2.7 sec

Observation: As the frequency of requests increase the average response time per client increase, which is expected due to providing {Reader/Write} concurrency

Number of Clients	1	2	3	4	5	1	2	3	4	5
variant	Improved					Unimproved				
response/Client	1 ms	1.5 ms	2.3 ms	2.25 ms	2.5 ms	1 ms	1.5 ms	2.4 ms	2.55 ms	3.25 ms

Observation: As the frequency of requests increase the average response time per client increase, which is expected due to providing {Reader/Write} concurrency

Percentage of Op	0% Add	20% Add	50% Add	80% Add	100% Add	0% Add	20% Add	50% Add	80% Add	100% Add
variant	Improved					UnImproved				
response/Client	2 ms	2 ms	2 ms	2 ms	2 ms	7.6 ms	2.4 ms	2 ms	2 ms	2 ms

Observation: The percentage of add and delete doesn't affect the response time, which is expected

because they both require the same amount of time for both variants and only 1 can be performed at a time.

6 Conclusion

- Difference between graph variants is significant only when the operations are done on a large graph (1,000,000 + edges)
- Response time per client increases as the number of clients increase and as the frequency of requests increases
- Response time for add/delete operations doesn't change, however adding query operations within increases the response time considerably.
- Trying Bi-directional BFS or increasing the cache hit rate by storing vertices connected to each other near each other.