Graph Analytics Project Report

Pierpaolo Agamennone Francesco Fulco Gonzales

Abstract

The following is the report for the High Performance Computing: Graph and Data Analytics project, for the track graph query. The aim of this document consists in explaining and justifying our implementation choices for the CSR and HashJoin data structures, the query algorithms and the testing thereof. We have used the relationships from the Pokec dataset to test and benchmark our program.

1 buildFromFile

In both the implementation of CSR and HashJoin we chose to build both data structures when the program is first executed, thus allowing all subsequent queries to be immediately executed, as the speed of execution of the queries is far more important than the time it takes to boot the program in most cases.

Following are the benchmarks for time and memory to load the dataset into the data structures.

TABLE 1: Time and Memory needed to load the dataset from disk into the data structures

	Time (s)	Memory (MB)
CSR	9.5	1825
HashJoin	10	2053
Both	17.6	2145

These tests were performed loading the whole dataset in memory. Memory refers to the RAM occupied by the program using one data structure at a time in the first two rows, and both data structures in the third row.

As we can notice the time and memory it takes to build the two data structures are very similar, hence the load time wouldn't impact the choice of one over the other. The surprising and unexplainable finding was that the memory for both data structures wasn't the sum of the two.

1.1 CSR

In readInput we leverage the java.util.ArrayList data structure to dynamically add all vertices to idx and ptr as we read them, and at the end of the read operation, once we know the length of both lists we transform them into actual arrays. This gives us a boost in performance and memory consumption, since integer arrays are more compact than ArrayLists.

1.2 HashJoin

For the HashJoin we used the java.util.HashMap, and we decided to load the graph immediately into the data structure using an intermediate tabular structure as we thought that it would have slowed down the building of the structures without any significant benefit. The hashmap is the ideal data structure to retrieve the outneighbors of a node as the search of a key is constant, without taking overflow into account.

The "value" of the hashmap is a LinkedList that contains all the outneighbors of a vertex, which is the key of the hashmap. In a previous implementation we used an ArrayList instead, which is more compact and uses less memory, as it relies on the Array data structure, however, it also slows down the insertion, since at each insert it has to check if the underlying Array is full and is especially expensive when it is indeed full and has to copy all elements to a new, bigger Array. The LinkedList avoids this problem altogether and speeds up insertion, which is the only operation we perform on this data structure.

2 getNeighbors

This method is the core of the whole program, as it contains the algorithm to retrieve the neighbors given a vertex.

2.1 CSR

The CSR implementation of getNeighbors uses the standard algorithm used for this data structure, hence we don't deem it interesting enough to be explained here.

2.2 HashJoin

In the HashJoin, to retrieve a node's neighbors we do a simple search using the node as key, since its corresponding "value" in the map is the list of its outneighbors, and key search in the HashMap takes constant time.

3 traverse

The traverse method, given a CSR and a vertex ID, returns a list of all the neighbors of that vertex.

This is implemented leveraging the getNeighbors method, and can use one of two algorithms: Breadth First Search or Depth First Search. We implemented both of these algorithms as we assumed they would give different performances for different depths, and we could find a threshold to switch to one or the other based on the given depth. We went ahead and tested our hypothesis, the results are reported in the table below.

TABLE 2: Performace of the BFS and DFS algorithms on queries of different depths

	Depth	Avg Time per query (ms)
BFS	10	2685
	6	12
	2	1.9
DFS	10	732
	6	9
	2	1.4

These tests report the average time taken by each type of query averaged over 100 queries and were done loading 1000 relationships, as our machine could not handle higher loads.

We can notice that DFS always wins over BFS, that is, because we didn't take full advantage of the BFS, which is fit for parallelization and would get a great speed gain from it.

4 join

This is the HashJoin corresponding method to the CSR traverse. It implements the same recursive algorithm and is quite similar to the former thanks to the abstraction of the getNeighbors method, that hides the differences between the two data structures.

Below are the benchmarks using the join method for our queries.

TABLE 3: Performance of the join algorithm on queries of different depths

	Depth	Avg Time per query (ms)
join	10	1421
	6	5.6
	2	1.2

These tests were performed with the same modality specified before, i.e. average over 100 queries, loading 1000 relationships.

These tests make clear that the join performs best for low to medium depth queries.

5 QueryEngine

Finally we implemented a way to parse a query written in the PGQL language, and to calculate the numeric value of the depth of the query. We achieved this using a regular expression.

6 Comparisons

Analyzing all the data reported, we can say that the HashJoin performs very well on low to medium depth queries, whereas the CSR with the DFS algorithm, is best suited for deep queries. Instead the BFS doesn't seem useful in this specific implementation as we did not take advantage of parallelism.

7 Heuristics

Based on these assessments we came up with a heuristic to choose the best data structure and algorithm for any given query. We identified the threshold to switch from DFS to join through trial an error. The threshold we found is 8, if a query has depth lesser than 8 we will apply the join algorithm, if it is higher or equal we will use the DFS.

TABLE 4: Performance of the heuristic on queries of different depths

	Depth	Avg Time per query (ms)
heuristic	10	713
	6	5.3
	2	1.1

8 Benchmarking & Tools

The project was written in java using maven as a build tool and the IntelliJ Idea IDE. We did not use any external libraries.

The script to benchmark the program was written in python using the library psutil to monitor the process' memory usage.

The tests were done one an Intel Core i7-6700HQ processor on an Ubuntu 20.04 Virtual Machine with 6 vCPU and 8 GB of RAM.

9 Improvements

The software could benefit from some improvements, namely the use of multithreading, especially for the BFS as previously mentioned, and also to run multiple queries in parallel. Another improvement could be the implementation of a custom HashMap, that avoids the overhead of java.util.HashMap, since we don't need most of its features. An improvement to the benchmarking would be monitoring the memory used for each query.