Homework 5

Introduction

GraphChi can run very large graph computations on just a single machine, by using a novel algorithm for processing the graph from disk. Compute the approximate diameter of undirected unweighted graphs using GraphChi framework.

Algorithms

I have implemented two version of approximate diameter estimation

1. **Sequential version using iFUB algorithm**
    - First convert the graph to undirected one
    - Then run the iFUB algorithm

ALGORITHM 1: iFUB

Input: A graph G, a node u, a lower bound l for the diameter, and an integer k
Output: A value M such that $D - M \leq k$

$i \leftarrow \text{ecc}(u)$;
$lb \leftarrow \max\{\text{ecc}(u), l\}$;
$ub \leftarrow 2\text{ecc}(u)$;
while $ub - lb > k$ **do**
  if $\max\{lb, B_i(u)\} > 2(i-1)$ **then**
    return $\max\{lb, B_i(u)\}$;
  else
    $lb \leftarrow \max\{lb, B_i(u)\}$;
    $ub \leftarrow 2(i-1)$;
  end
  $i \leftarrow i - 1$;
end
return lb;

ALGORITHM 2: 4-SWEEP

Input: A graph G
Output: A lower bound for the diameter of G and a node with (hopefully) low eccentricity

$r_1 \leftarrow$ random node of G or node with the highest degree;
$a_1 \leftarrow argmax_{v \in V} d(r_1, v)$;
$b_1 \leftarrow argmax_{v \in V} d(a_1, v)$;
$r_2 \leftarrow$ the node in the middle of the path between a_1 and b_1;
$a_2 \leftarrow argmax_{v \in V} d(r_2, v)$;
$b_2 \leftarrow argmax_{v \in V} d(a_2, v)$;
$u \leftarrow$ the node in the middle of the path between a_2 and b_2;
$lowerb \leftarrow \max\{\text{ecc}(a_1), \text{ecc}(a_2)\}$;
return $lowerb$ and u;

2. **Graphchi version using 2 sweep algorithm**
    - First Convert the graph to undirected graph
    - Find the components of the graph
    - Run BFS in each component choosing any vertex
    - Find the distant point and the distant vertex
    - Run BFS again from that vertex and find the distant vertex from it and the distance
    - The last distance is the approximate diameter of the graph.

Compile and Runtime Environment

- JAVA JDK 1.7
- MAVEN
- GraphChi Java version
- Environment : Hadoop pc

Evaluation

The results might vary as these are all approximate diameter and two algorithms are different. For the sequential one I am using iFUB algorithm and for the GraphChi version I am using modified 2-sweep.

Dataset	GraphChi Diameter	Sequential Diameter	GraphChi Running time (seconds)	Sequential Running time (sec) for 10 iteration
CA-CondMat.txt	15	15	70.625	30.092
com-amazon.ungraph.txt	47	47	495.447	270.994
CA-HepPh.txt	12	13	37.6	23.438
CA-GrQc.txt	17	17	13.6	2.306
CA-AstroPh.txt	14	14	62.289	10.4
CA-HepTh.txt	18	18	92.996	2.735

Analysis

Sequential version

This algorithm can find the approximate diameter of the graph very efficiently and it is also very fast. If graph data structure is optimum then it can perform faster. This algorithm uses 4 sweep algorithm to choose the starting point of the BFS. As starting point is an important factor of the algorithm, it uses 4-Sweep to find the initial source.

I am running the algorithm iteratively with random starting point. So, if the graph contains more than one component than we can check it very easily in different iteration. As, it starts from a random vertex so diameter may differ in different iteration. We can check the result of each iteration and find the approximate diameter which is the most frequent among them.

GraphChi version

In the first step to find the diameter we find the components of the graph. If the graph contains only one component than we can just run BFS twice to find the diameter. Because, we are using 2 sweep algorithm and it can also find the approximate diameter of a graph. But, if the graph contains more than one component, we have to run BFS in each components.

I have also used GraphChi program to find the components of the graph. It is very fast and find the components in a matter of seconds. My GraphChi program to find the components returns the smallest id vertex and the total number of vertices in that component.

When we get the smallest vertex of each components, we run BFS from that point. My GraphChi BFS algorithm finds the farthest point from the source and also saves that vertex. So, this is the first sweep of the algorithm. Then we start again from that point and run BFS. This time the algorithm will also find the furthest point from the source. This is the second sweep. And the distance we get from here is the

diameter of the graph. We do it for each of the components in the graph. So, in the end we have all the components with the diameter of each one. So, we can select the distant component as the graph diameter.

I have used 1 shard lo load data in memory. Iteration number for each BFS is 100. But, it converges before that. Iteration number to find components is 1000. But, it also converges before that. Because we are using schedulers to schedule the vertices which are neighbor of current vertex.

Conclusion

Both of these approaches are approximate and don't find the diameter in a deterministic way. At first I tried to use only BFS in each of the vertices of the graph to find the diameter. This process will find the diameter accurately. But, it is not a convenient way to find the diameter. Because, for a large graph the algorithm is very slow and event may not even end!

GraphChi helps us to do this work in our own PC. GraphChi can process computation on very large graphs by using a novel Parallel Sliding Windows -algorithm. The graph is split into P **shards**, which each contain roughly the same number of edges in sorted order. PSW can process a graph with mutable edge values efficiently from disk, with only a small number of non-sequential disk accesses, while supporting the asynchronous model of computation. PSW processes graphs in three stages: it 1) loads the graph from disk; 2) updates the vertices and edges; and 3) writes updated values to disk.

I have used the Java version of GraphChi. This java version is new and not rich as C++ version of GraphChi. But, unfortunately my GraphChi version of the algorithm is slower than the sequential one. It is very hard to fine tune the shard number to find a perfect combination. I tried different number of shards. But, the GraphChi version is still slow. But, it never crashes. Lastly I have used only 1 shard in this program. I also found the files it creates for shards are very irritating if you run it for different inputs.

How to run

Sequential Version

- From the command line go to the root folder
  - /home/2014280162/Homework5/Diameter_Sequential
- Clean package using "mvn clean"
- Build package using "mvn package"
- Run the program using **mvn exec:java -Dexec.mainClass="sequential.Diameter" -Dexec.args="inputDataFile numberOfIteration"**
  - mvn exec:java -Dexec.mainClass="sequential.Diameter" -Dexec.args="data/input 10"

GraphChi version

- From the command prompt go to the root folder
  - /home/2014280162/Homework5/Homework5_GraphChi
- Clean the package using "mvn clean"
- Build the project using "mvn package"
- Run by : **mvn exec:java -Dexec.mainClass="BigData.Homework5.Diameter" -Dexec.args="datainputFile numberOfShards"**
  - mvn exec:java -Dexec.mainClass="BigData.Homework5.Diameter" -Dexec.args=" data/input 10"