High Performance Computing II

ATHANASIOS AGRAFIOTIS*

Athanasios Agrafiotis. 2023. High Performance Computing II. 1, 1 (July 2023), 4 pages.

1 ABSTRACT

The assignment has as a main purpose to implement an algorithm using the initial system requirements that corresponds to the a random walk algorithm. The main algorithm implementation is to optimize the performance of the agent during the navigation in a graph. In this paper we show that the original Forelem framework that can be used to implement the algorithm and to compare with the initial implementation. These implementations are used C/C++ and the time execution was used as a metric.

2 INTRODUCTION

The paper introduces a technique that is able to compute a graph structure properties. The problem that the paper examines is a sampling algorithm that makes use of the markov chain probabilities. The algorithm starts navigate through a graph of nodes and edges. The size of the graph is known as the number of nodes and number of edges. The purpose of the paper is to sample a limit number of nodes that reproduce the results of the original graph. The recommended number of sample is at the twenty percent of the original graph.
The most well known algorithms that have been used is the breath first search and the random walk. In today graphs the network size tends to be enormous, in such case the the breadth first search algorithm is computational expensive. Some of the obstacles is that the breadth first has shown that overestimate the shortest path length and the random walk it create loss of information as the sampling of nodes is totally random. In addition the the random walk is not the same efficient for all kind of graphs as the number of edges have affected the random walk outcome.

3 RELATED WORK

The Depth First Search is most fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of O(V+E) and is often used as a building block in other algorithms. By itself the DFS is not all that usefull, but when augmented to perform other tasks such as count connected components, determine connectivity, or find bridges/articulation points then DFS really shines. As the dfs expands suggests, a depth first search plunges depth first into a graph without regard regard for which edge it selects next until it cannot go any further, at which point it backtracks and continues it's exploration. So a depth first search has to start on a node 0. The start our depth first search on node zero and arbitrarily goes to node 9 between the neighbors 1, 9. Then from node 9 moves to node 8 that no other neighbors. The algorithm searches until all the vertices have visited in case the algorithm has finds a node that have already visited backtracks to the first time the node have visited and move to an unvisited node.
Database queries with the wrapping C/C++ code and it's associated API layer. Database queries were automated together with the wrapping C/C++ allowed for integral optimization. The Forelem operations are tupled based

* Author's address: Athanasios Agrafiotis, athaagrak@gmail.com.

$

and are used from this framework, the most well used loop structures: the forelem and whilelem loops. The tuple are refered as a physical and virtual instances and are stored without any specific order that the algorithm, the loop structure iterate over the loops. The outcome of the forelem and whilelem optimize the implementation , automatically.

4 BREADTH FIRST SEARCH

The Breadth first Search (BFS) is a fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of O(V+E) and is often used as a building nlock in other algorithms. The BFS algorithm is particularly useful for one thing: finding the shortest path on unweighted graphs. A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbors. Breadth first search expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$. Consider a graph that the source vertex A is the source and next visited node is B of this algorithm before visiting C, D. For this reasons, means that B, C, D is at a distance of one.
So in a graph that starts a breathless search at a source node of zero visits all the nodes that are neighbors. Next

Algorithm 1 Breadth First Search [**?**]

```
1: function solve(s)
2: q = queuedatastructurewithenqueueanddequeue
3: visited = [false, .., false]
4: visited[s] = true
5: prev = [null, .., null]
6: while !q.isEmpty() do
7:    node = q.dequeue()
8:    neighbours = g.get(node)
9:    for (next : neighbours) : do
10:      if !visited[next] : then
11:         q.enqueue(next)
12:         visited[next] = true
13:         prev[next] = node
14:      end if
15:   end for
16: end while
17: return prev
```

it visit all nodes that are second neighbors of the source node. So it starts a breathless search from which node to visit next handling a queue. So as an example in a graph that a node ’0’ consider as a source node. 0 < − − −−
In the next step the algorithm explores
11 (unvisited nodes)
7 (unvisited nodes)
9 (unvisited nodes) < − − −(new visited node)
0< − − − (visited)

Using a queue the BFS algorithm uses a queue data structure to track which node to visit next. Upon reaching a new node that algorithm adds it to the queue to visit it later. The queue data structure works just like a real world queue such as a waiting line at a restaurant. People can either enter the waiting line (enqueue) or get seated (dequeue).

5 RANDOM WALK

In to keep the network properties, the proposed a high-degree biased variable-length random walk algorithm. The random walk backtracks to the nodes with higher degree nodes and provides more information about network. The source node is usually a node with high centrality in this way redudant information is avoiding. Taking in consideration previous approaches high degree nodes makes the algorithm to step more times. Random walk has shown to achieve to keep most of the network structure and reduce space and computation overhead making our algorithm more efficient and scalable on large-scale networks. The random walk select a node (u) with a uniform random probability $\frac{1}{n}$ from a graph V. In the next iteration the algorithm picks one of the unvisited neighbors of node u in the case that the number of neighbors are k, node u has degree K_u, include one of it's neighbors with probability $\frac{1}{K_u}$. The process repeats until the desired length of the random walk is reached. Self avoiding node cannot be visited more than once.

Algorithm 2 Random Walk [1]

Input: Network G(V,E) max walk length l_{max}
Node sequence walk
Initialize walk to $[u]$
$l = minDeg(u), L_{max} + 1$
for $i = 0$ **do**
  $curr = walk[-1]$
  Select a node u uniformly from neighbors of curr
  Append u to walk
  Generate a random value$p \in [0, 1]$
  if $p < (1 - \frac{Degv}{Degu})$ **then**
    Append curr to walk
  end if
end for

6 FORELEM SPECIFICATION FOR BREADTH FIRST SEARCH AND RANDOM WALK

Recall that the Forelem framework allows "random" execution with no specific order for a specific number of times, in contrast the classic implementation of the for loop that the order of operations are fixed. The main difference between the Forelem implementation and the classic For is that the computation must be reduced to it's core.
The classical while loop continues to execute until the criteria has satisfied. This naturally corresponds to using a whilelem loop in the Forelem specification, which by definition terminates as soon as all tuples result in a no-op operation. In the the algorithm of breadth first search loops through the queue that have all the nodes of the graph. In the first step discover the neighbors of the current node that has dequeued. In the second step, the data is the neighbor is not in the visited nodes will be added.

The second algorithm makes use only of one loop that iterates for specific number of steps until to required length.

Algorithm 3 Random Walk [1]

Input: Network G(V,E) max walk length l_{max}
Node sequence walk
Initialize walk to $[u]$
$l = minDeg(u), L_{max} + 1$ $i = 0$
$curr = walk[-1]$
Select a node u uniformly from neighbors of curr
Append u to walk
Generate a random value$p \in [0, 1]$
if $p < (1 - \frac{Degv}{Degu})$ **then**
  Append curr to walk
end if

Algorithm 4 Breadth First Search [**?**]

function solve(s)
$q = queuedatastructurewithenqueueanddequeue$
$visited = [false, .., false]$
$visited[s] = true$
$prev = [null, .., null]$ $!q.isEmpty()$
$node = q.dequeue()$
$neighbours = g.get(node)$ $(next : neighbours)$:
if $!visited[next]$: **then**
  $q.enqueue(next)$
  $visited[next] = true$
  $prev[next] = node$
end if

return prev

7 EXPERIMENTS

To evaluate the performance of the derived bfs and random walk implementation we ran several experiments, using the first two implementations and a the implementations using the forelem and whilelem.

8 RESULTS

REFERENCES

[1] Yunyi Zhang, Zhan Shi, Dan Feng, and Xiu-Xiu Zhan. 2019. Degree-biased random walk for large-scale network embedding. *Future Generation Computer Systems* 100 (2019), 198–209. https://doi.org/10.1016/j.future.2019.05.033