High Performance Computing II

ATHANASIOS AGRAFIOTIS*

Athanasios Agrafiotis. 2023. High Performance Computing II. 1, 1 (August 2023), 8 pages.

1 ABSTRACT

The assignment has as a main purpose to implement an algorithm using the initial system requirements that correspond to 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 compiler that the forefam framework has developed can be used to implement the algorithm and to compare it with the initial implementation. These implementations used C/C++ and the time execution was used as a metric.

2 INTRODUCTION

The paper introduces a technique that is able to compute graph structure properties. The problem that the paper examines is a sampling algorithm that makes use of the markov chain probabilities. The algorithm starts navigating through a graph of nodes and edges. The size of a graph is known as the number of nodes and number of edges. The purpose of the paper is to sample a limited number of nodes that reproduce the results of the original graph. The recommended number of samples is at twenty percent of the original graph.
The most well known algorithms that have been used are the breath first search and the random walk. In today's graphs, the network size tends to be enormous. In such case, the the breadth first search algorithm is computationally expensive. Some of the obstacles are that the breadth first has been shown that we 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 kinds of graphs as the number of edges has affected the random walk outcome.

The remainder of this paper is organized as follows, in section 3 related work we discuss related work on the api. Section 4,5 and 6 explain queue algorithms, compilers and parameters. In section 8 dataset provides information about the data. In section 9 results we provide our experimental results. Finally, in section 10 conclusion, we conclude the software outcome and provide suggestions for future implementation.

3 RELATED WORK

The Depth First Search is the most fundamental search algorithm used to explore nodes and edges of a graph. It runs at 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 useful, 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 for which edge it selects next until it cannot go any further, at which point it backtracks and continues its exploration. So a depth first search has to start on a

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

$

node 0. The start of our search is first search on node zero and arbitrarily goes to node 9 between the neighbors 1, 9. Then, from node 9, it moves to node 8 that no other neighbors. The algorithm searches until all the vertices have been visited. In case the algorithm has found a node that has already visited has backtracks to the first time the node has visited and moves to an unvisited node.
Database queries with the wrapping C/C++ code and its associated API layer. Database queries were automated together with the wrapping C/C++ allowed for integral optimization. The Forelem operations are tupled based and are used from this framework, the most well used loop structures: the forelem and whilelem loops. The tupl are referred as physical and virtual instances and are stored without any specific order in the algorithm, the loop structure; iterates over the loops. The outcome of the force and whole 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 lock 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.

's first search expands the frontier between discovered and undiscovered vertices uniformly across the breadth the frontier. That is, the algorithm discovers all vertices at distances k from s before discovering any vertices at of distance $k + 1$. Consider a graph where the source vertex A is the source and the next visited node is B in this algorithm before visiting C, D. For reason, it means that B, C, D are at a distance of one.
So, in a graph that starts a breathless search at a source node of zero, it visits all the nodes that are neighbors.

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
```

Next, it visits all nodes that are second neighbors of the source node. So it starts a breathless search for which node to visit next, handling a queue. So, as an example, in a graph, that a node '0' is considered 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)

graph,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 order to keep the network properties, they 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 the network. The source node is usually a node with high centrality. In this way, redundant information is avoided. Taking into consideration previous approaches, high degree nodes make the algorithm step more times. Random walk has shown to keep most of the network structure and reduce space and computation overhead, making our algorithms more efficient and scalable on large-scale networks.

The random walk selects a node (u) with a uniform random probability of $\frac{1}{n}$ 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 nodes cannot be visited more than once.

Algorithm 2 Random Walk [2]

```
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 valuep ∈ [0, 1]
  if p < (1 − Degv/Degu) then
    Append curr to walk
  end if
end for
```

6 LLVM

It is what is known as IR. Human readable IR is compiled down to a bitcode that makes use of assembles that represent the machine-level bitcode. The inverse of the assembly is known as disassemble. The most important component is one known as a Module , which is a container for functions, global variables and constants. The next component is called context, which is a container of modules and, finally, is the IR builder that allows you to generate IR. To build a compiler, that first is the creation of context and, on top of that, is the module. Using the

module, it is possible to define the global variables and function definitions, the whole system implements the code using the IR builder. The context manages the core and the LLVM's infrastructure, including the type of the data structure. The module is used to store all the information related to the LLVM module. Module is the Intermadiate Representation of objects. Each module implements the fundamental properties of the variables such as global and local , the list of functions and a list of libraries. A module maintains a number of variables that are used to store data in the module. And finally, the uniform API for creating instructions and inserting them into a basic block: either at the end of a BasicBlock, or at a specific iterator location.
The function creation requires the parameter types(the type of the function), the return (functions output), another tag is the varargs(is the varied number of arguments). Next is the function body that in LLVM consists of a number of blocks that leads to an optimization and can more clearly represented as a list of instructions. The first block is the entry block that handles the variable initialization or represents a math calculation. The next block is the branch instructions, which could be a number of comparison operators using if and else. If the condition is satisfied, if the function can proceed to the next block, that might differ in each iteration as it strongly depends on the condition. The next block is the return and where the function terminates. The function declaration is the header that has the function name, function type and, at last, the return type of the function. The compiler makes use of S-expressions that configure the variable name and number representation in the code scheme. The syntax tree is the configuration of data structures in code lists, vectors. The variables can be defined as three types of global variables that can be allocated outside of any function and they are mutable. The second is the register variable that follows the Static Single Assignment that cannot be assigned twice as the temporary memory is replaced with the new variable. Usually it represents temporary results of operations. In more detail, whenever a variable is called by the compiler, the compiler creates a new variable that might have the name of the first variable with an incremental number and keeps track of the stored parameters that will be optimized so the compile will work faster.

7 IMPLEMENTATION OF THE LOOP

The implementation of the loop was implemented as a separated iteration tools. The core of the iteration loop 'while' initialize with the structure of the loop(intialize, block ends). In each iteration a queue is used to push back the current bit digit with the next bit digit in an incremental way, until the final bit digit is the new insert point. During the iteration process the current bit digit and the next bit digit is temporary optimized from an implemented function optimize that keep temporary record of the stored parameters in blocks. The core of the optimization is the syntax tree that represents the numbers and the variables in a specific scheme.

```
    (
        (var z 138)
        (while (> z 0)
            (var x 250)
            (while (> x 0))
            (begin
                (set x (- x 1))
                (set z (- z 1))
                (printf "%d " x)))
);
```

Execution Time		
BFS	RW	LLVM(loop simulation)
361 sec.	360 microsecond	157 microsecond.

Table 1

8 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. For the purpose of the experiment, we generated a graph of 250 nodes and 1000 edges. The breadth first was crawled the whole graph and then recreated. The procedure was similar to the random walk algorithm and the outcome depicts the shortest distances without replacement. The random walk will discover a node that has already been visited once. The number of shortest distances paths differs between each algorithm. The shortest paths of the graph were calculated using an implementation of the Djikstra algorithm in (C++). In addition, an implementation of a loop was implemented as a Module that can run in the C++ programming language and the number of iterations was simulated to measure execution time. The execution time depends on the timestamp and it terminated until we finish the process of visiting node graphs.

9 RESULTS

Each of the algorithms was tested on simulation of both algorithms. The first algorithm, BFS, which is an approach that visits all nodes as iterates through a graph, depicts that the average shortest distance path was close to one half. A sample using the breadth the first search is not representative, with the shortest distance path close to one quarter, which introduces an error of approximately a quarter.

The next approach that was tested was the random walk implementation where the algorithm visits two hundred fifty nodes. The number of nodes that the algorithm visited was fifty-seven and the outcome depicts that the graph recreation and the average shortest path shows equal to zero points nine. The random walk was also executed by visiting a smaller sample. The number of nodes that the random walk has visited was nineteen and the result shows that both samples have the same shortest distance compared to each other

During the implementation, there was also the execution time was table 1. The breadth first search implementation is an algorithm that consists of a nested iteration that makes the process heavy, the random walk implementation has at least one iteration. In this case, two iterations were implemented in the random walk. The table depicts that both implementations differ and the random walk has a faster execution time than the breadth the first search. The LLVM compiler using clang that is implemented to run on C++ shows that the execution time can be remarkably better for both algorithms. The implementation of the forelem framework is similar using a syntax-tree and improves the execution time [1].

10 CONCLUSION

To sum up, two algorithms were tested that examine the properties of the shortest path in the whole graph and in smaller samples and an implementation of functions(LLVM compiler) that can execute the same algorithm. The result depicts that the random walk cannot be compared with the breadth first search to explore the whole graph. The error between the property of the shortest distance path of random walk and bfs can not be compared. The difference between the two samples of algorithms depicts high similarity , the error cannot be taken into consideration as the short distance is almost identical. All algorithms were implemented by the author the LLVM

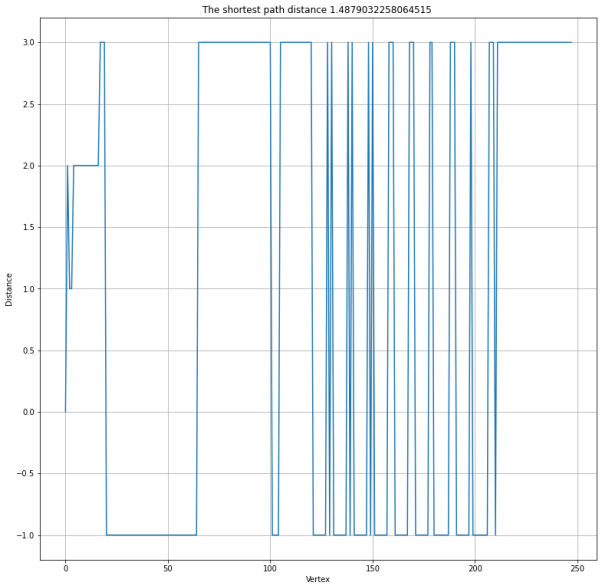

Fig. 1. shortest path distance BFS

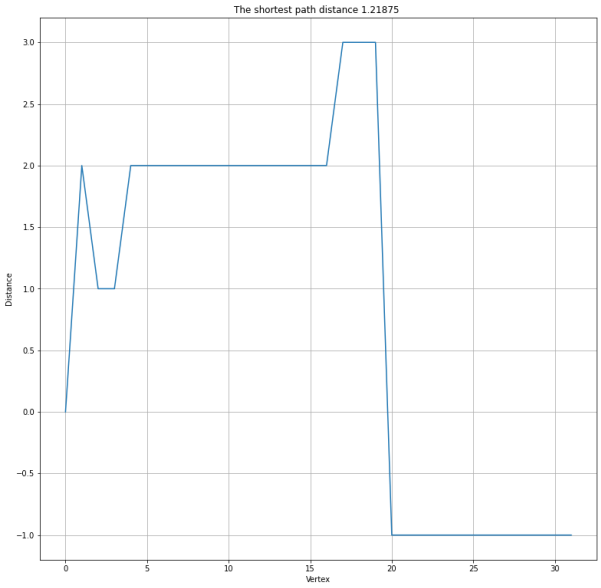

Fig. 2. shortest path distance BFS sample

Fig. 3. Breadth First Search

gives straightforward instructions for the creation of the iteration tools. The contribution of this paper is the implementation of an iteration module, the implementation of algorithms in C++ and at last the comparison.

The LLVM execution time that runs on C++ depicts that the LLVM iterates and will decrease the latency and can cover larger graphs to examine the properties. The forelem tool is similar with while implementation that is

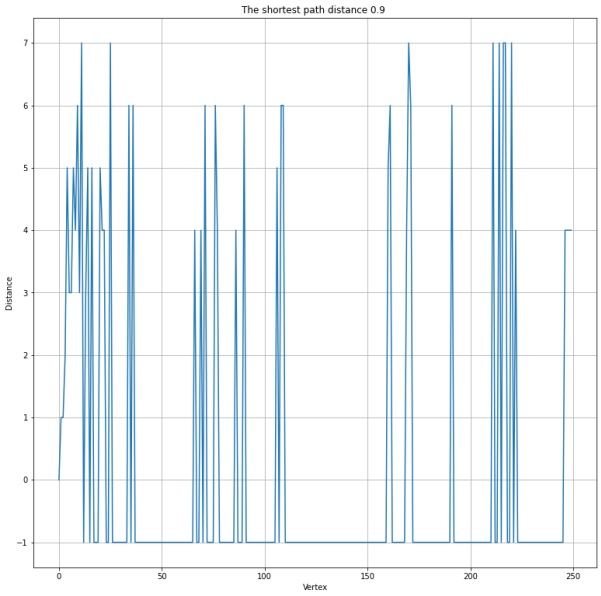

Fig. 4. shortest path distance RW

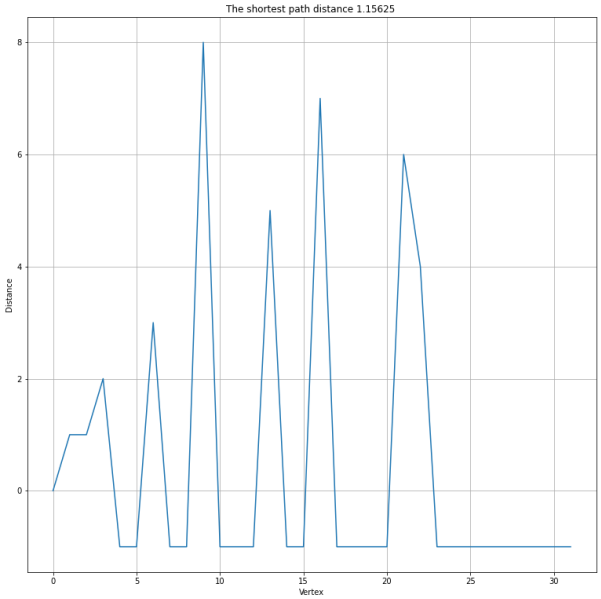

Fig. 5. shortest path distance RW sample

Fig. 6. Random Walk

demonstrated in this paper. The drawback that have not been tested with data transformation and different data structures.

-github link:https://github.com/Athaagra/HPComp

REFERENCES

[1] Kristian F. D. Rietveld and Harry A. G. Wijshoff. 2013. Forelem: A Versatile Optimization Framework For Tuple-Based Computations.
[2] 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