Graph Cycle Detection Using PySpark

Itamar Shamshins, Asmaa Sayah, Ron Amihai
itamar.shamshins@post.runi.ac.il,asmaa.sayah@post.runi.ac.il,ron.amihai@post.runi.ac.il
Reichman University
Israel

ABSTRACT

Graphs are a ubiquitous data structure in a variety of applications, including social networks, financial transactions, computational biology, and mathematical modeling. As a result, detecting cycles in graphs is essential for solving real-world problems, such as identifying relationships between users or detecting fraud networks. Distributed cycle detection algorithms are useful when dealing with large graphs that are processed across multiple nodes of a distributed computing system.

In this paper, we present our implementation of a distributed cycle detection algorithm by *Rocha, Thatte.*[1]. The algorithm is designed to run on a large-scale distributed computing framework such as PySpark. We evaluate our implementation on synthetic graphs, demonstrate its distributed nature and correctness in detecting cycles.

Our work aims to advance distributed graph processing by providing a tool for researchers and practitioners working with large-scale graphs.

KEYWORDS

Big data, Spark, Graph, Cycle Detection, Cluster, Distributed, GraphFrames

ACM Reference Format:
Itamar Shamshins, Asmaa Sayah, Ron Amihai. 2018. Graph Cycle Detection Using PySpark. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cycle detection algorithms are fundamental in graph theory and have numerous applications in various fields. Patterns within graphs can be used to solve complex problems such as social media mining and fraud detection.

For instance, *Bodaghi and Rostampour* [2] have shown that cycle detection algorithms can be used to detect car-insurance fraud by parties that may seem unrelated at first. Similarly, *Vsubelj and Bajec* [3] have used cycle detection algorithms to identify fraudulent activities in banking and financial services, such as circular transactions between bank accounts to hide the source of money (money laundering).

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Cycle detection algorithms also have well-known applications in cryptography and science. *Brent and Pollard* [4] have shown in the early 1980's that cycle detection algorithms can be used to discover the factors of Fermat's 8^{th} number ($2^{256}+1$). *Nester and Shparlinski* [5] have shown the effectiveness of cycle detection algorithms in factorization of integers for cryptography.

Foreign-exchange arbitrage detection is another application of cycle detection algorithms. Sets of currency exchange rates across multiple assets are easy to detect as a cycle with a negative sum of edge weights when the logarithm of the exchange rate is used to weight individual edges. *Cui et al.* [6] have demonstrated the effectiveness of cycle detection algorithms in detecting foreign-exchange arbitrage opportunities.

The focus of this paper is to implement a distributed algorithm capable of detecting cycles on large-scale, sparse directed graphs. To achieve this goal, we will leverage the bulk synchronous message model as used in the paper by *Rocha, Thatte.*[1] to identify cycles.

2 RELATED WORK

Numerous papers and tools for large-graph processing have been released in the last decade. A notable example is Google's Pregel, which has been the inspiration for systems such as Facebook's Giraph and Spark's GraphX [7] [8] [9].

GraphX, contributed to the Spark Project by UC-Berkeley in 2014, exposes a graph data structure and API using Spark RDDs. It also includes a set of algorithms for basic graph operations, such as Shortest Paths, but has the limitation of the underlying RDDs being immutable and therefore not suitable for graph applications which require updating in real-time.

GraphFrames, by *Dave et al.*, a newer framework (2016) presented by a cross industry team from MIT, UC-Berkeley, Uber, Databricks and Microsoft, exposes Graph structures using Spark DataFrames, is mutable and has extended functionality when compared with GraphX [10]

While these (and other) frameworks implement a few well-known algorithms internally, they do not offer ready-made implementations for all applications. Specifically, we could not find any implementation for cycle detection on GraphFrames nor on GraphX. However, these frameworks do offer APIs and tools for implementing user-specific algorithms for users who require to perform a specific computation, such as the one described in this paper.

3 METHODS AND IMPLEMENTATION

In this work, we implemented an algorithm which is based on the *bulk synchronous parallel (BSP)* model for message passing between graph vertices.

Under BSP, graph data is split across a distributed system by partitioning batches of vertices to be processed on different nodes, thus enabling significant scalability.

BSP provides a general framework for parallel computing via messaging, making it an ideal choice for large-scale graph processing when the scale of the graph is larger than shared memory on a single machine. This approach has been shown to be highly effective in various parallel computing applications, as discussed by Leslie et al. [11].

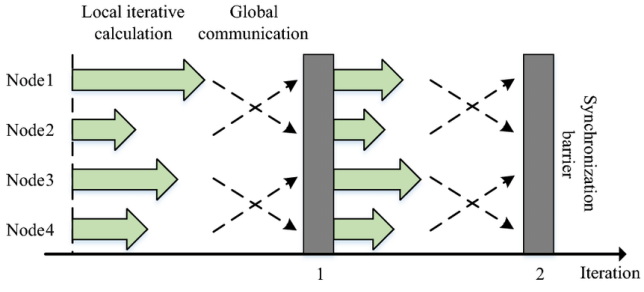
3.1 The BSP model

The BSP *(bulk synchronous parallel)* model is a parallel computing model that enables efficient processing of large-scale data sets by dividing the computation into a series of synchronized *supersteps*.

At the heart of the BSP model is the concept of a *superstep*, which acts as a unit of computation and coordination between machines running the parallel execution.

A superstep consists of three internal steps:

(1) **Parallel computation:** each processor in the parallel system performs its assigned task on a portion of the data set. This step is designed to be highly parallelizable, with minimal communication between processors.
(2) **Communication:** processors exchange data with each other to ensure that each processor has the information it needs for the next superstep. This communication step is crucial for ensuring that the computation can continue in the next superstep.
(3) **Synchronisation barrier:** ensures that all processors have completed their computations before the next superstep begins. This step is essential for maintaining consistency in the computation and avoiding race conditions.



3.2 Cycle Detection - Overview:

Let S be a *superstep* in the execution of the algorithm:

- As S begins, each vertex in the graph will receive messages sent to it by its neighbors during the previous step $(S-1)$, and **might** send messages to its neighbors, which they will receive at the next step $(S+1)$.
- In the first superstep (S_1), each vertex v_i sends all of its neighbors a message containing itself (v_i), and receives no messages.
- From the second superstep (S_2) and until the end of the algorithm, each vertex may both receive and send messages from/to its neighbors, as long as certain conditions are met.
- Once a cycle is detected, the messages' sequence that contains it is dropped. Alternatively, for all message sequences without cycles, eventually all vertices halt and the algorithm ends.

3.3 Superstep details :

1. Parallel computation At the beginning of a superstep (S_i), any running vertex v inspects all the messages it has received from neighbors. Messages are always in the form of a sequence of vertex ids $(v_1, v_2, ..., v_k)$.

If no messages are received then v *halts* and does not participate in further computation.

When a message is received, the vertex checks whether:

(1) Its id is the first in the sequence (i.e. $v = v_1$). If so, a cycle has been detected and is reported as needed. The algorithm ends.
(2) If v is present in the sequence (i.e. $v = v_i, i \in 2, 3, ..., k$) then the sequence contains a cycle but some other vertex would have identified it in a previous step. The vertex *halts*.

2. Communication

If none of the above is true, then execution continues -

v appends itself to the sequence $(v_1, v_2, ..., v_k, v)$ and forwards it to all of its neighbors.

3. Barrier

Vertex v waits for the next message to arrive. If no message arrives at the next superstep the vertex deactivates and halts. Otherwise, it commences superstep S_{i+1}.

3.4 Complexity

The total number of iterations of the algorithm is the number of vertices in the longest path in the graph $(O(E))$, plus a few more steps for deactivating vertices $(O(1))$

3.5 Intuition: detecting cycles by message passing

Figure 1 presents an example of the execution of the algorithm. In iteration i = 3, all the three vertices detect the cycle [2, 3, 4]. We ensure that the cycle is reported only once by emitting the detected cycle only from the vertex with the least identifier value in the ordered sequence, which is the vertex 2 in the example.

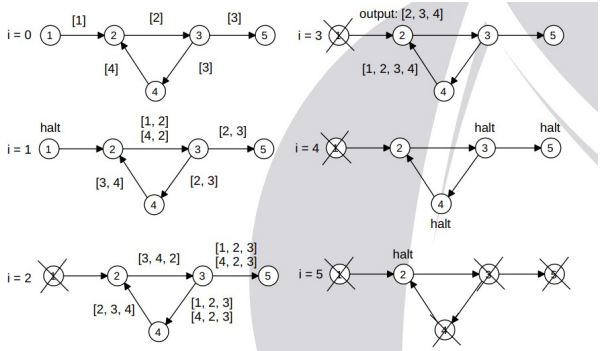


Figure 1: Example of the algorithm for detecting cycles by message passing.

(image from Rocha and Thatte) [1]

3.6 Pseudo-code:

Algorithm 1 Pseudocode for the compute function of the distributed cycle detection algorithm. The algorithm takes G as input, and for each superstep i the function COMPUTE($M_i^{(v)}$) is executed for each active vertex v.

```
1:  function COMPUTE(M_i^(v))
2:      if i = 0 then
3:          for each w ∈ N+(v) do
4:              send (v) to w
5:      else if M_i^(v) = ∅ then
6:          deactivate v and halt
7:      else
8:          for each (v_1, v_2, ..., v_k) ∈ M_i^(v) do
9:              if v_1 = v and min{v_1, v_2, ..., v_k} = v then
10:                 report (v_1 = v, v_2, ..., v_k, v_{k+1} = v)
11:             else if v ∉ {v_2, ..., v_k} then
12:                 for each w ∈ N+(v) do
13:                     send (v_1, v_2, ..., v_k, v) to w
```

(image from Rocha and Thatte) [1]

3.7 Datasets

To evaluate the performance and correctness of our cycle detection implementation, we will use synthetic graphs with specific properties. Synthetic graphs offer a controlled environment where we can manipulate the graph's properties to test the algorithms' behavior in different scenarios.

Graphs were generated as follows:

- We used NetworkX, a Python graphs-package, to generate random graphs. Specifically, we used the '"fast gnp" random graph function
- Each graph is defined by 2 properties:
  - Number of vertices
  - A probability p for an edge to exist between any set of 2 vertices in the graph

Since Rocha-Thatte's algorithm is designed for sparse graphs, we generally used graphs with a probability of 3%-5% of edge incidence.

Once a graph was generated, a cycle detection run was made and results recorded during run time. Recorded data included the following:

- Graph properties; numbers of edges and vertices
- Number of discovered cycles and their paths
- Time (in seconds) to complete each full iteration (i.e. a BSP iteration of processing-messaging-barrier)
- Configuration of the underlying hardware used in the test

Since detection runs were using unique random graphs, a second dataset was prepared once the original data was collected. This second set grouped and averaged all data on the basis of the cluster-size and provider.

More specifically, the metrics were aggregated on the basis of 2 configuration parameters:

- Per-infrastructure basis, either Google Colab or Google Cloud Dataproc
- Number of worker machines

All google Colab runs were considered as single-machine runs, with 2 vCPUs each.

All Dataproc runs were either 2-nodes or 5-nodes. All worker machines (and the driver) were 4 vCPUs and 16GB RAM, each.

This information is presented later, in the evaluation section.

3.8 Implementation Details

Our implementation relies on Apache Spark (pySpark) as the data repository, and GraphFrames for graph-representation and vertex-to-vertex communications. Code is available on a github repository: ***primary link*** *(secondary link: https://tinyurl.com/nuthm73s)*

3.8.1 *Cycle detection and outbound message construction.*

The main body of work (other than vertex-to-vertex communications, which is handled by GraphFrames) is the analysis of incoming paths from all neighbors of any given vertex.

Incoming paths are a list of vertex-ids which are aggregated into a list-of-lists (paths from all neighbors). An example of such an incoming message would be as follows:

[[5, 38, 8, 35], [31, 38, 8, 35], [0, 30, 8, 35], [12, 30, 8, 35], [3, 12, 8, 35], [36, 23, 8, 35], [33, 22, 14, 35], [2, 22, 14, 35], [31, 22, 14, 35], [34, 10, 13, 35], [22, 10, 13, 35], [12, 10, 13, 35], [14, 35, 26, 13], [3, 35, 26, 13], [2, 35, 26, 13], [8, 35, 26, 13]]

Each inner list is a path sent to the receiving vertex from one of its neighbors, and the outer list aggregates message to that vertex from all its neighbors.

Message complexity:

The number of messages and their size grows with each step of the algorithm. At step 10, all paths are of length 10 and each vertex can receive messages originating from all other vertices that are 10 steps away. As such, the number of messages can grow exponentially, depending on the density of the graph.

3.8.2 *Path analysis:*

To determine whether a cycle is detected, and whether it should be reported, the algorithm requires we search for the id of the receiving vertex in paths sent to it from its neighbors. The specific conditions are outlined under sections 3.6 and 3.7.

We implemented this logic in 3 different ways (configurable with a parameter):

(1) DataFrame queries
(2) Flat paths analysis
(3) Nested Paths analysis

i. DataFrame Queries:

Under this method the algorithm uses only dataframe sql-like commands such as .where(), .withColumn() etc.

All paths are analyzed using spark functions such as explode(), array_contains() and element_at(). This method may be appropriate for users that are constrained from using UDFs, but is the slowest option available.

ii. Flat paths analysis

This method uses a combination of DataFrame commands and a UDF for analysis of incoming paths. The nested structure of inbound messages is first split into multiple dataframe rows (using explode()) and then a UDF loads each path into a numpy array to search for cycles and construct outbound messages.

This is the second slowest method, since explode() is a costly operation. Also, path data is loaded into numpy arrays on an individual basis, which is also costlier than necessary.

iii. Nested paths analysis

Under this method, incoming lists of paths are loaded into numpy multi-dimensional arrays and analyzed using vector operations. This is the fastest option.

3.8.3 *Inter-vertex message handling:*

Serialization and Compression:

Since the size messages grows significantly with each iteration of the algorithm, we also employ data serialization and compression on outbound messages. Once the messages are received at the other end, they are decompressed and deserialized into a list of lists.
We used pickle for serialization and zlib for compression - both being standard packages with an efficient implementation in C.

Structure correction:

An artefact of GraphFrames aggregation is that the nested messages could have a "jagged" structure (i.e. internal arrays of unbalanced length) - this is corrected as part of the decompression operation when the aggregated messages are processed.

4 EVALUATION

As described under **3.8 Datasets**, numerous metrics were collected during each test run. The metrics were *aggregated and averaged.*

The rationale behind this approach is that multiple random graphs would offer a better benchmark than a single structure.

Averaging over the size and run time of each test, relative to number of worker machines, gives us a view of the *average iteration time, per size of graph when given a specific infrastructure configuration.*

In the plots below we show results of both individual runs as well as the average-runs.

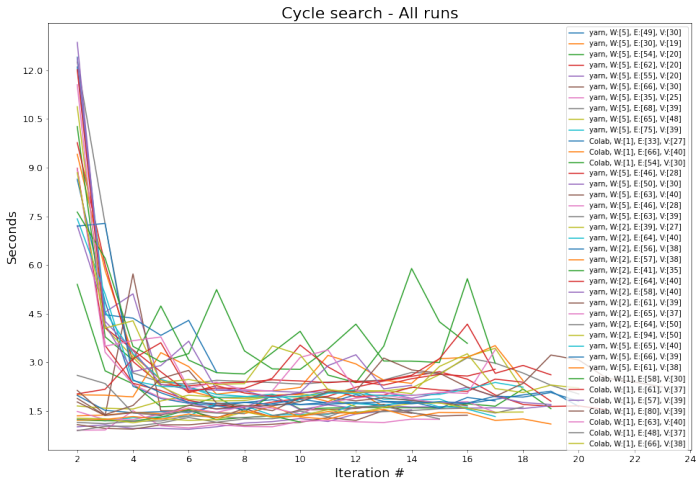
4.0.1 *Correctness of implementation.*

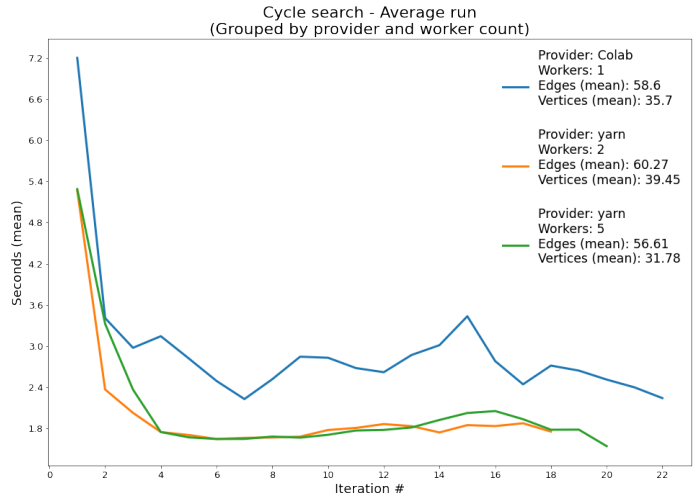
We ran several controlled tests (on non-random graphs) to demonstrate correctness of implementation. The assumptions underlying these tests are:

- The original algorithm is theoretically correct (as proven in the paper by *Rocha and Thatte*)
- Given the simple nature of this algorithm, if our implementation is correct for several different use-cases, then it will remain correct for all use cases. That is, we are demonstrating correctness, not proving it mathematically.
- If errors occur due to infrastructure issues (i.e. a node in the cluster suffers a failure), this does not negate the fundamental correctness of the cycle detection code

Tests covered the following use cases:

- A circle graph - each edge is connected only to one neighbor and the last node connects to the first node (contains one cycle of length E)
- Graphs with:
  - partially-overlapping cycles and a tail
  - "nested" cycles and a tail.
  - non-overlapping cycles
  - no cycles at all
  - disjoint vertices groups





5 DISCUSSION

Scalability of the distributed algorithm:

Generally speaking, a cluster of workers handled graph analysis faster than a service such as Colab. This was expected since the clusters had significantly more resources than a Colab notebook does.

However, as can be seen in the plots above, there was not a significant difference in run-time when using clusters with 2 or 5 nodes (all nodes were identical). The reason for this is described in greater detail below, but comes down to the algorithm encountering issues before node resources were fully utilized.

As such, more work would be required to improve this implementation's ability to fully utilize cluster resources.

Message complexity as a limiting factor:

A key limitation of the Rocha-Thatte algorithm became apparent once testing of this implementation was performed.

Namely, that an underlying assumption of the algorithm is that computation and memory complexity of large-graph processing can be offloaded to 'messaging complexity' as a means to scale out.

In other words, since the graph would be broken up to multiple machines it is assumed that individual cluster-members would

be able to make simple paths-analysis and send their conclusions to all neighbors, without a limitation on the overall graph size. Furthermore, the maximum run-time of the algorithm would only be $O(E)$, which is better than most other algorithms.

However, in reality the vertices would be required to send an exponentially increasing set of messages to all neighbors thereby quickly overwhelming the message processing pipeline of the cluster.

In our work we used compression to mitigate this effect, though with limited success. Multiple spark-cluster configurations (2,3,5,6,7 nodes) quickly suffered stack overflows when edge-counts went over 100+, which is not a graph size which justifies usage of a distributed environment.

Run-time analysis shows that resource usage on individual workers was not a constraint. That is, messaging complexity saturated the cluster long before individual processing at each node reached capacity.

One caveat to the conclusion above is that it is possible that the overflows were due to some easily-resolvable issue with a specific component in the spark stack (or in GraphFrames). If so, the behavior described above should be easily resolvable.

Future work:

To resolve the issues described above, several steps can be taken.

First, if a specific component of the cluster environment is causing stack overflows, it should be possible to implement a workaround. One suspect is the GrpahFrames framework which was used to facilitate messaging.

Second, more refined messaging approaches can be implemented. One example is forwarding paths to neighbors using a fine-grained message streaming approach rather than sending a complete set of all incoming paths in one message. This will utilize network capacity to its fullest, without causing parts of the software stack to crash.

Last, usage of a cluster with a much larger number of machines which are also smaller in size (i.e. a swarm of small machines) would probably better tackle the issue of message complexity, since under this paradigm an individual node does not require much processing power and instead we want to spread out communications across as large a number of nodes as possible.

6 CONCLUSIONS

In this work we implemented a distributed cycle-detection algorithm, based on work from *Rocha and Thatte* [1].

The feasibility of partitioning a graph and analyzing it for cycles on a pyspark/graphframes stack was demonstrated.

Some underlying challenges related to the usage of the BSP model in iterative cycle detection algorithms was encountered and discussed.

Further work is warranted in order to overcome the messaging barrier in larger graphs; possibly by replacing the vertex-messaging framework with a more robust implementation to avoid stack overflows when processing exponentially increasing message broadcasts.

REFERENCES

[1] Rodrigo Rocha and Bhalchandra Thatte. Distributed cycle detection in large-scale sparse graphs. 08 2015.

[2] Arezo Bodaghi and Babak Teimourpour. Automobile insurance fraud detection using social network analysis. In *Applications of Data Management and Analysis*, pages 11–16. Springer, 2018.

[3] Lovro Šubelj, Štefan Furlan, and Marko Bajec. An expert system for detecting automobile insurance fraud using social network analysis. *Expert Systems with Applications*, 38(1):1039–1052, 2011.

[4] Richard P Brent and John M Pollard. Factorization of the eighth fermat number. *Mathematics of Computation*, 36(154):627–630, 1981.

[5] A. Nesterenko. Cycle detection algorithms and their applications. *Journal of Mathematical Sciences*, 182, 04 2012.

[6] Zhenyu Cui and Stephen Michael Taylor. Circular arbitrage detection using graphs. *Stevens Institute of Technology School of Business Research Paper*, 2018.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Joy Arulraj. Apache giraph, https://giraph.apache.org/.

[9] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. *OSDI*, pages 599–613, 01 2014.

[10] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the fourth international workshop on graph data management experiences and systems*, pages 1–8, 2016.

[11] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, aug 1990.