Breadth-first search with MapReduce

Ekimov Victor
University of Trento
MAT.174212
ekimov.victor@gmail.com

ABSTRACT

Actual paper is a **"Big Data"** course project at **University of Trento, Italy**. Following section is devoted to the transformation of a well-known serial **Breadth-first search** algorithm into a parallel version and their head-to-head comparison. Paper contains references to the source code freely available on **github** accounts as well as instructions on how to run them and what technologies have been used.

1. BFS WITH MAPREDUCE

In this section we will start from general introduction into graph data structure and inner implementation, followed by exploring the breadth-first search, one of the most well-known algorithm for graph processing. Serial version of the algorithm is provided by **"Algorithms, 4th Edition" by Robert Sedgewick and Kevin Wayne** [1], while parallel version will be backed up by our custom implementation on **GitHub: BFS-with-MapReduce**. Section includes benchmark and algorithms comparison to answer the question: does the parallel performs better?

1.1 Introduction to Graphs

Pairwise connections carry an important role in representing many real world relations and abstract structures. Whether a social network, maps, web site cross links or software dependency libraries, having a sound model to put objects together in a meaningful way helps discovering new facts about all of them at once, rather than individual piece at a time. Mathematics defines a graph theory for study such phenomena.

Definition 1. A *graph* is a set of *vertices* and a collection of *edges* that each connect a pair of vertices. [1]

A graph does not necessarily imply that every vertex is reachable from any other vertex, thus leaving a room for situations where graph becomes a set of connected components. In such way, two vertices might have nothing in common.

Definition 2. A *graph* is *connected* if there is a part from every vertex to every other vertex. [1]

The way two vertices are connected by an edge may fall into categories of:

- **self-loop** vertex is directly connected to itself
- **single edge** vertex is directly connected to another vertex by exactly one edge
- **parallel edges** vertex is directly connected to another vertex by more than one edge

Vertex not only can be connected to another vertex directly, but also through a set of different vertices, creating a path from one to another. Also, there might exist more that one way to connect two vertices by multiple paths.

Definition 3. A *path* in a *graph* is a sequence of vertices connected by edges. [1]

Graph may be of two forms:

- **directed** specifying the edge direction from one vertex to another
- **undirected** where no edge direction is implied (equivalent to bi-directional graph)

Direction or absence of direction influence the path construction.

Implementation of the graph can be done in two major approaches:

- **adjacency matrix** two-dimension boolean array of vertices, where "true" defines an edge between two vertices while "false" defines an absence of such edge.
- **array of adjacency lists** vertices array of lists of edges (each array index is a vertex id, while the content of such array cell is a list of other vertex ids directly connected to).

Adjacency matrix takes constant time to add an edge or to check the adjacency of two vertices, but requires $V * V$ space, which is inefficient, since most of the matrices tend to have much more vertices than edges.

Array of adjacency lists takes space proportional to $E+V$, constant time to add an edge, but checking the adjacency of two vertices takes a degree of vertex being checked.

For the future sections we will consider undirected, connected graph, using the array of adjacency lists implementation where each edge is considered to be a distance of 1 unit.

1.2 Serial Breadth-first search

BFS is a non-recursive algorithm, unlike Depth-first search (DFS). It is designed by choosing a source vertex, visiting all of its neighbours, then visiting all the neighbours of the first neighbour of the source vertex and so forth. Such implementation involves a queue to store all the vertices to be visited.

As an example, we will be using BFS to solve the single-source shortest path question, thus given a source vertex, find the shortest path to all other vertices.

Source vertex always has distance set to zero, path set to itself, color to GRAY (scheduled for processing) and is the first to be queued. Until the queue is not empty, dequeue the vertex to set the color to BLACK (processed) and for each of its neighbours, if the color is WHITE (not processed), then increment the distance, add parent vertex to the path, set the color to GRAY and enqueue for further procession. Figure 1 shows a pseudo-code for serial BFS algorithm.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Figure 1: Serial BFS algorithm

Let's explore the example of **tinyCG.txt** [1] where adjacency list corresponds to a Table 1.

Vertex	Neighbours
0	1, 2, 5
1	0, 2
2	0, 1, 3, 4
3	2, 4, 5
4	2, 3
5	0, 3

Table 1: An array of adjacency lists

Imagine the source vertex is colored as GRAY and queued for the next iteration. Figure 2 illustrates the initial state of graph processing algorithm.

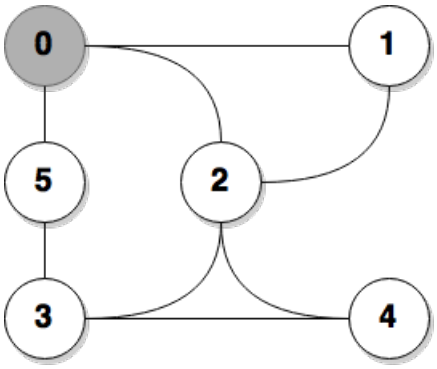

Figure 2: Iteration 0

Figure 3 enters the while loop and marks source vertex 0 as BLACK while its neighbours are added to the queue (1, 2 and 5) as GRAY.

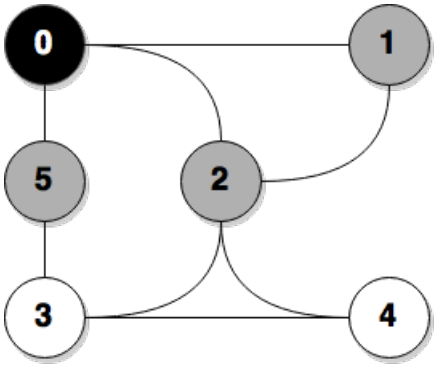

Figure 3: Iteration 1

Figure 4 dequeues the first neighbour of source vector, which happens to be 5 (order does not matter), marks it as BLACK and queues WHITE neighbours as GRAY (note that only vertex 3 is currently WHITE).

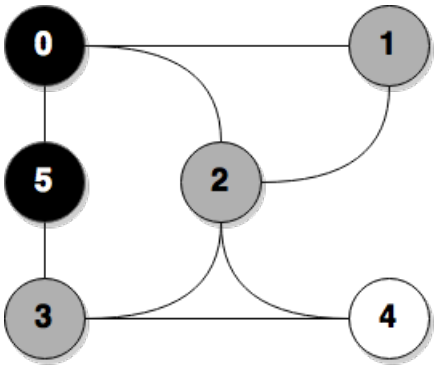

Figure 4: Iteration 2

Figure 5 dequeues the second neighbour of source vector, which happens to be 2, marks it as BLACK and queues WHITE neighbours as GRAY (note that only vertex 4 is

currently WHITE, while vertices 3 and 1 are already GRAY, even though they are neighbours of vertex 2).

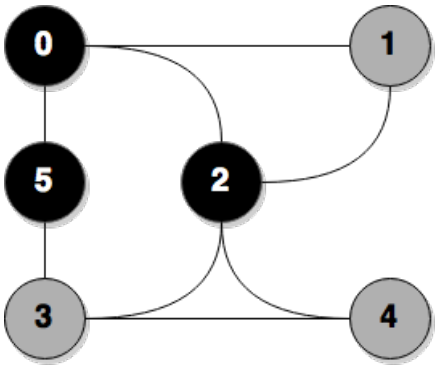

Figure 5: Iteration 3

Figure 6 dequeues the third neighbour of source vector, which happens to be 1, marks it as BLACK and continues iteration because it has no WHITE neighbours.

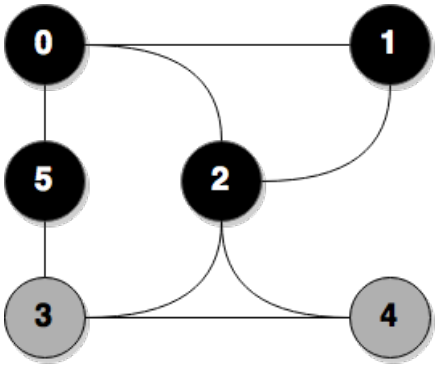

Figure 6: Iteration 4

Figure 7 dequeues the second level of vertices, added by vertex 5, which happens to be 3, marks it as BLACK and continues iteration because it has no WHITE neighbours.

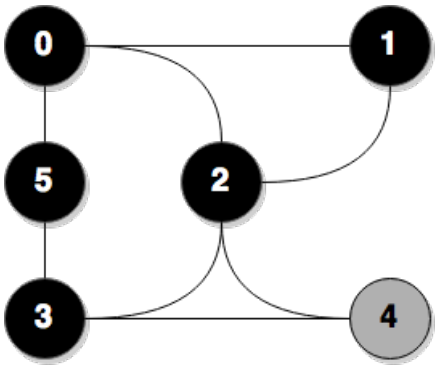

Figure 7: Iteration 5

Figure 8 dequeues the second level of vertices, added by vertex 2, which happens to be 4, marks it as BLACK and continues iteration because it has no WHITE neighbours.

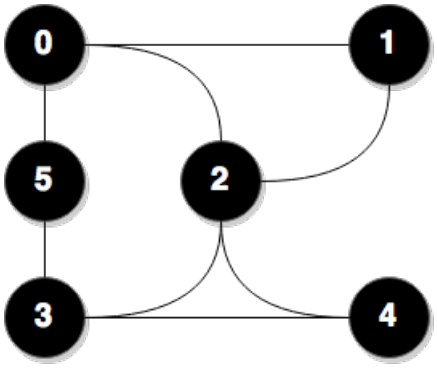

Figure 8: Iteration 6

Upon termination, BFS outputs the path from source vertex to every other vertices as in Table 2.

Vertex	Path to the source vertex
0	0
1	0, 1
2	0, 2
3	0, 5, 3 or 0, 2, 3 depending on the order
4	0, 2, 4, because other paths are longer
5	0, 5

Table 2: An array of adjacency lists

1.3 Parallel Breadth-first search

First observation of the serial BFS is that every new enqueue operation creates an iteration. That brings us to an idea of having iterative MapReduce algorithm, where outcome of the previous step will be fed as an input for the next step. While serial version explores the graph vertex by vertex, parallel explores level by level, which brings us reduced number of actual iterations (from 6 to 3).

Second observation is that parallel BFS will need to keep the graph structure and pass it along the iterations. Each node in a cluster need to see the entire graph to be able to reason about it. That requires a lot of read/write operations to restore the graph structure for the next iteration.

Third observation is the termination condition. For serial BFS termination depends on the emptiness of the queue, but for parallel BFS vertexes will reside on different node and there will be no "global" queue to check. Therefore, termination condition is computed after each iteration outside the MapReduce section. In connected graph, the number of iterations correspond to the diameter of the graph, which is the greatest distance between any vertices.

Given all these observations, Figure 9 shows a pseudocode for parallel BFS algorithm.

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)
1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         d_min ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d
8:             else if d < d_min then
9:                 d_min ← d
10:        M.Distance ← d_min
11:        Emit(nid m, node M)
```

Figure 9: Parallel BFS algorithm

Mapper emits a set of results, always including all vertices (GRAY as BLACK, others preserve the color) and optionally all their direct neighbours as new vertices with increased distance (in case if parent vertex was GRAY). Reducer receives a set of vertices grouped by vertex id and emits the one with the shortest distance and the darkest color (to indicate completeness of exploring the vertex).

Let's explore the example of **tinyCG.txt** [1] where adjacency list corresponds to a Table 1.

Imagine the source vertex is colored as GRAY. Then Figure 10 will illustrate the initial state of graph processing algorithm.

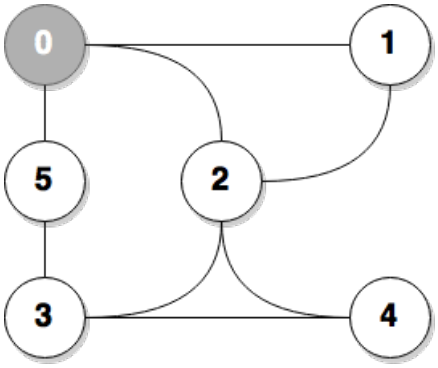

Figure 10: Iteration 0

Figure 11 mapper emits all vertices in the graph plus all the neighbours of every GRAY vertex. In our case vertices 1, 2 and 5 are emitted twice, first time with no changes as WHITE, second time as new GRAY vertices and incremented distance to the source. Reducer sees two entries for each of the 1, 2 and 5 vertices and emits only one new vertex for each key, which combines the shortest distance and darkest color.

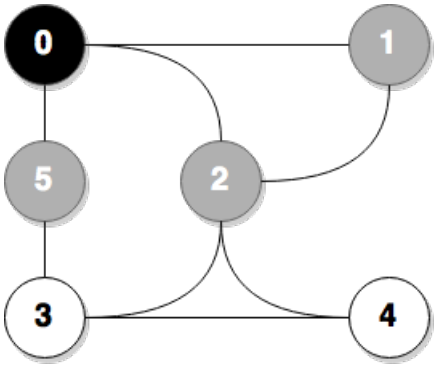

Figure 11: Iteration 1

Figure 12 mapper emits vertex 0 three times, 3 twice, 4 once on top of the normal emitting. Also, vertices 1 and 2 are connected, even if GRAY, they still perform extra emit of each other. All GRAY vertices are colored BLACK (1, 2 and 5).

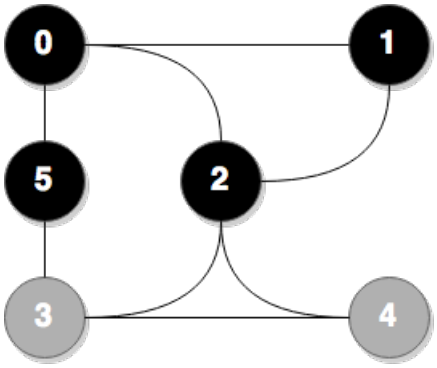

Figure 12: Iteration 2

Figure 13 mapper emits vertex 5 once and vertex 2 twice on top of the normal emitting. Also, vertices 3 and 4 are connected, even if GRAY, they still perform extra emit of each other. All GRAY vertices are colored BLACK (3 and 4).

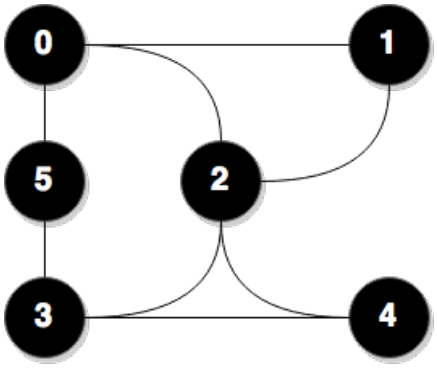

Figure 13: Iteration 3

No more GRAY vertices left, therefore BFS from source vertex outputs the path to every other vertices as in Table 2, similar to serial version.

1.4 Implementation of BFS with MapReduce

For implementation of the parallel BFS we are using Apache Spark 1.4.0, Java 7 and Scala 2.11 (for spark).

The entire implementation consist of 4 java classes:

- **BfsSpark** BFS algorithm on MapReduce
- **GraphFileUtil** utility for converting problem file from "Algorithm" book [1] into more convenient format
- **Vertex** represents vertex entity in the graph
- **Color** enum for coloring vertices

Vertex class is readable/writable from/to disk with information about id, direct neighbours, path to the source vertex, distance to the source vertex and current color (state of processing).

GraphFileUtil class is used to take the original problem file and convert it into initial input for BFS algorithm. Original file contains number of edges, which is irrelevant, but misses the second pair of edge, assuming undirected graph. For simplicity, it is useful to have all the neighbours for each vertex listed, not only one side of it. Class also initiates the state for source vertex with distance set to 0 and color to GRAY, while other vertices get color WHITE and positive infinity distance. Additionally, the path of each vertex is started with the source vertex and will finish with the id of itself when BFS terminates. New format corresponds to the "source" string of the Vertex class.

BfsSpark class initializes the spark application from **service.properties** file and goes through every problem file. Each of them is first converted with GraphFileUtil class and fed to the first iteration. After each iteration, intermediate computation is saved into a new problem file with underscore at the end of the name and iteration number. That file will be fed for the next iteration. While loop terminates once there is no more GRAY vertex present.

Mapper receives a list of lines from problem file and outputs a set of tuples with vertex id as key and vertex itself as value. First of all, vertex is reconstructed from the source string. If vertex happens to be GRAY, its neighbours are iterated through and added to the result set as new gray vertices with increased distance and modified path. No need to the the neighbours information, since reducer will recover them from original vertex. Every GRAY vertex becomes BLACK and is added to the result set to preserve the color.

Reducer receives vertices by id and recovers the neighbours list, chooses the darkest color, minimal distance and the path from the vertex having such minimal distance.

Imagine the source vertex is colored as GRAY. Table 3 illustrates the initial state of graph processing algorithm, corresponding to Figure 10.

Vertex	Neighbours	Path	Distance	Color
0	[1, 2, 5]	[0]	0	GRAY
1	[0, 2]	[0]	2147483647	WHITE
2	[0, 1, 3, 4]	[0]	2147483647	WHITE
3	[2, 4, 5]	[0]	2147483647	WHITE
4	[2, 3]	[0]	2147483647	WHITE
5	[0, 3]	[0]	2147483647	WHITE

Table 3: Iteration 0

Table 4 corresponds to Figure 11.

Vertex	Neighbours	Path	Distance	Color
0	[1, 2, 5]	[0]	0	BLACK
1	[0, 2]	[0, 1]	1	GRAY
2	[0, 1, 3, 4]	[0, 2]	1	GRAY
3	[2, 4, 5]	[0]	2147483647	WHITE
4	[2, 3]	[0]	2147483647	WHITE
5	[0, 3]	[0, 5]	1	GRAY

Table 4: Iteration 1

Table 5 corresponds to Figure 12.

Vertex	Neighbours	Path	Distance	Color
0	[1, 2, 5]	[0]	0	BLACK
1	[0, 2]	[0, 1]	1	BLACK
2	[0, 1, 3, 4]	[0, 2]	1	BLACK
3	[2, 4, 5]	[0, 5, 3]	2	GRAY
4	[2, 3]	[0, 2, 4]	2	GRAY
5	[0, 3]	[0, 5]	1	BLACK

Table 5: Iteration 2

Table 6 corresponds to Figure 13.

Vertex	Neighbours	Path	Distance	Color
0	[1, 2, 5]	[0]	0	BLACK
1	[0, 2]	[0, 1]	1	BLACK
2	[0, 1, 3, 4]	[0, 2]	1	BLACK
3	[2, 4, 5]	[0, 5, 3]	2	BLACK
4	[2, 3]	[0, 2, 4]	2	BLACK
5	[0, 3]	[0, 5]	1	BLACK

Table 6: Iteration 3

1.5 Performance comparison

Data sets with undirected graphs in it were taken from the "Algorithm" book [1]. Serial version was adopted from the same source, except that now we measure only the actual BFS algorithm using Stopwatch from Guava library, excluding the graph construction and intermediate steps. Parallel version is the one described in the previous section, runs with different number of workers, yet residing on the same machine. Once again, due to iterative nature of BFS, parallel version records the time only during the actual MapReduce stage and then sums it all up for each iteration.

Table 7 shows the comparison of serial vs parallel BFS. Tiny data set has 6 vertices and 16 edges, medium 250 vertices and 2,546 edges, large 1,000,000 vertices and 15,172,126 edges. Note that initial setup of spark takes about $\sim 3{,}4$ seconds, so the very first test will always be delayed by that time. Normally reruns differentiate by ~ 200 ms.

As we can see from the Table 7, adding more workers actually decreases the performance of the algorithm. It happens due to the "waste" generation and passing the graph structure among the nodes. Even with one worker, serial version outperforms the parallel one. You can traverse the large graph with serial BFS in time it takes to traverse two tiny with parallel BFS!

Problem	Serial	Parallel		
		1 worker	2 workers	10 workers
tinyCG	1,686 ms	569,1 ms	342,8 ms	1,610 s
mediumG	1,275 ms	2,914 s	3,924 s	20,94 s
largeG	1,170 s	none	none	none

Table 7: Serial vs Parallel BFS comparison

The reason for large data set to be unreachable is **OutOfMemoryError: Java heap space** o worker nodes during the reduce stage due to the amount of "waster" generated by the mapper. Recall that every mapper iteration generates final number of vertices as number of existing vertices (1,000,000) plus all the neighbours of the GRAY once. Such iterations continue up the the diameter of the graph, in our case could be over 30 times!

1.6 Conclusion

Compared to serial BFS, parallel BFS in MapReduce is a brute force approach which generates a lot of "waste" during the map phase that will be discarded later at reduce phase. Each iteration it re-emits all the nodes of GRAY vertex, often repeating the previous computations. Serial version is more efficient, giving the shortest path to each visited vertex on every iteration. However, the cost of such efficiency is maintaining a global queue of vertices, which is not an option in distributed environment.

2. REFERENCES

[1] R. Sedgewick and K. Wayne. Algorithms, 4th edition. 4:518–566, March 2011.