A MapReduce-based approach for shortest path problem in large-scale networks

Sabeur Aridhi, Philippe Lacomme *, Libo Ren, Benjamin Vincent

Laboratoire d’Informatique (LIMOS), UMR CNRS 6158, Campus des Cézeaux, 63177 Aubière Cedex, France

ARTICLE INFO

Article history:
Received 24 March 2014
Received in revised form
10 February 2015
Accepted 10 February 2015

Keywords:
Shortest path
Large-scale network
MapReduce
Hadoop
Cloud computing

ABSTRACT

The cloud computing allows to use virtually infinite resources, and seems to be a new promising opportunity to solve scientific computing problems. The MapReduce parallel programming model is a new framework favoring the design of algorithms for cloud computing. Such framework favors processing of problems across huge datasets using a large number of heterogeneous computers over the web. In this paper, we are interested in evaluating how the MapReduce framework can create an innovative way for solving operational research problems. We proposed a MapReduce-based approach for the shortest path problem in large-scale real-road networks. Such a problem is the cornerstone of any real-world routing problem including the dial-a-ride problem (DARP), the pickup and delivery problem (PDP) and its dynamic variants. Most of efficient methods dedicated to these routing problems have to use the shortest path algorithms to construct the distance matrix between each pair of nodes and it could be a time-consuming task on a large-scale network due to its size. We focus on the design of an efficient MapReduce-based approach since a classical shortest path algorithm is not suitable to accomplish efficiently such task. Our objective is not to guarantee the optimality but to provide high quality solutions in acceptable computational time. The proposed approach consists in partitioning the original graph into a set of subgraphs, then solving the shortest path on each subgraph in a parallel way to obtain a solution for the original graph. An iterative improvement procedure is introduced to improve the solution. It is benchmarked on a graph modeling French road networks extracted from OpenStreetMap. The results of the experiment show that such approach achieves significant gain of computational time.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

MapReduce (Dean and Ghemawat, 2008) is a parallel programming model approving design of algorithms for cloud computing. Such framework favors processing of problems across huge datasets using a large number of computers. Hadoop is a MapReduce implementation dedicated to distributed computation over the web which has been first introduced for research promoted by Google. Hadoop is an open-source Apache project providing solutions for reliable and scalable distributed computing. It takes advantage of dedicated Hadoop Distributed File System (HDFS) for data storage and of MapReduce programming paradigm. Hadoop provides a data parallel processing paradigm on multiple computers across a cluster and offers the possibility of massively parallel processing.

The requirement for data-intensive analysis of scientific data across distributed clusters has grown significantly in recent years. By combining modern distributed computing technologies and operations research it is possible to reach a larger audience informed of, and benefit from, increasing amount of operations research software implemented and by this way to gain gradual quality. The MapReduce programming paradigm permits to investigate definitions of new approaches in operations research area with the objective to solve strongly large instances which could be difficult to address using a single-node approach due to both computational time and memory requirements. Thus, the adaptation of resource-intensive applications such as optimization algorithms (commonly used for solving operations research problems) to the cloud, require attention since the applications must be tuned to frameworks that can successfully take advantages of the cloud resources (Srirama et al., 2012).

The shortest path problem has received a considerable amount of attention for several decades. Many approaches used for solving operations research problems such as vehicle routing and scheduling problems are based on the shortest path algorithms. Moreover, the shortest path problem in large-scale real-road networks is the cornerstone of any real-world routing problem including the dial-a-ride problem (DARP), the pickup and delivery problem (PDP) and its dynamic variants. The most of efficient methods dedicated to these problems compute first the distance matrix. The construction of such matrix involves the shortest path calculation between each pair of

* Corresponding author. Tel.: +33 473 407 585.
E-mail addresses: sabeur.aridhi@gmail.com (S. Aridhi), placomme@isima.fr (P. Lacomme), ren@isima.fr (L. Ren), benjamin.vincent@isima.fr (B. Vincent).

http://dx.doi.org/10.1016/j.Engappai.2015.02.008
0952-1976/© 2015 Elsevier Ltd. All rights reserved.
nodes and it could be a time-consuming task on a large-scale network due to its size. The classical shortest path algorithm is not suitable to accomplish efficiently such task. Furthermore, those last years there have been a revival of interest for the shortest path problem used in the context of various transport engineering applications.

In this paper, we focus on the design of an efficient MapReduce-based approach for the shortest path problem in large-scale real-road networks. It is the first step to evaluate how the MapReduce framework can create an innovative way for solving operational research problems. The main idea of the proposed approach consists in partitioning the original graph into a set of subgraphs, then solving the shortest path on each subgraph in a parallel way to obtain a solution for the original graph. An iterative improvement procedure is introduced to improve an explored solution. Our objective is not to guarantee the optimality but to provide high quality solutions in acceptable computational time. The proposed approach is tested on a graph modeling French road network extracted from OpenStreetMap.

This paper is organized as follows. The following two subsections provide a short state-of-the-art on the shortest path problem and on the MapReduce framework. Section 2 presents the Hadoop architecture and the MapReduce programming model. The proposed MapReduce-based approach is detailed in Section 3. The experimental results and the comparison study can be found in Section 4, before the concluding remarks in Section 5.

1.1. The context of shortest path problem

In recent years, the shortest path problem has attracted much interest in numerous transportation applications. Without any doubt it could be attributed to the recent developments in Intelligent Transportation Systems (ITS), mainly in the field of Route Guidance System (RGS) and real time Automated Vehicle Dispatching System (AVDS). In both cases, a part of the whole system is used to compute the shortest path from an origin point to a destination one in a quick and accurate way.

In the applications mentioned above, the conventional optimal shortest path algorithms are not suitable because they are too greedy in computation time. The current RGS field has generated new interest in the use of heuristic algorithms to find shortest paths in a traffic network for real-time vehicle routing. Several heuristic search strategies have been developed in order to increase the computational efficiency of the shortest path algorithms.

Most of these heuristic search strategies came from the artificial intelligence (AI) field including but not limited to Hart et al. (1968), Nilsson (1971), Newell and Simon, (1972), and Pearl (1984). Here, the shortest path problem is used as a mechanism to validate the efficiency of these heuristics.


Let us note that the recent advances concern new extensions and complexity improvements. For example, the multi-objective shortest path problem is addressed by Umair et al. (2014). They introduced a single solution based on an evolutionary algorithm for the Multi-objective Shortest Path (MOSP). The second contribution of their paper concerns a state-of-the-art on the MOSP where publications are gathered into classes depending on the studied approaches. Some authors, including Tadao (2014), introduced new advances in complexity improvements of the classical shortest path algorithms based on bucket approaches. Cantone and Faro (2014) introduced lately a hybrid algorithm for the single source shortest path problem in the presence of few sources or few destinations of negative arcs.

For a recent survey it is possible to refer to Fu et al. (2006) proposed a state-of-the-art and examined the implementation and performance of numerous heuristic algorithms. Another survey of Garroppo et al. (2010) has focused on the multi-constrained approximated shortest path algorithms which are represented by works of Holzer et al. (2005), Wagner et al. (2005) and Gubichev et al. (2010).

1.2. MapReduce related works

Dean and Ghemawat (2008) showed in their paper that Google uses MapReduce for a wide variety of problems including but not limited to large-scale indexing, graph computations, machine learning or data mining from a huge set of indexed web pages. Cohen (2009) makes the first step into considering that MapReduce could be successfully applied for some specific graph problems, including computation of graph components, barycentric clustering, enumerating rectangles and enumerating triangles. MapReduce has also been benchmarked for scientific problems by Bunch et al. (2009). It has been proved that MapReduce could be an efficient alternative for simple problems including but not limited to the Marsaglia polar method for generating random variables. In their paper of 2011, Sethia and Karlapalem (2011), explore the benefits and possibilities of the implementation of multi-agents simulation framework on a Hadoop cloud. They concluded that the MapReduce-based framework they introduced is strongly efficient.

In Lin and Dyer (2010), the authors discuss scalable approaches to process large amounts of text with MapReduce and one of the addressed problems is the single-source shortest path. In this context, the authors discussed the adaptation of a MapReduce based breadth first search strategy to solve the single source shortest path problem. The main idea of this work is to emit paths along with distances in the map phase, so that each node will have its shortest path easily accessible at all times. The reduce phase consists of selecting the shortest path by choosing the path with minimum value of distance.

In Malewicz et al. (2010), the authors present PREGEL, a computational model suitable for large-scale graphs processing. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex centric approach is flexible enough to express a broad set of algorithms. Implementing a large-scale graph processing algorithm consists of creating a PREGEL program.

In Plimpton and Devine (2011), the authors present an MPI library that allows graph algorithms to be expressed in MapReduce. The main idea in this work is to implement typical MapReduce functionalities with MPI. Several graph algorithms have been evaluated such as PageRank, triangle finding, connected component identification, and single-source shortest path computation on large artificial graph.

The work presented in Kang and Faloutsos (2013) gives a description of PEGASUS, an open source Peta-graph mining library which performs typical graph mining tasks such as computing the diameter of a graph, computing the radius of each node and finding the connected components. The main idea of PEGASUS is the GIM-V primitive, standing for Generalized Iterative Matrix-Vector multiplication, which consists of a generalization of normal matrix-vector multiplication. PEGASUS customizes the GIM-V primitive and uses MapReduce in order to handle with important large-scale graph mining operations.

In the work of Kajdanowicz et al. (2014), three parallel techniques have been compared including MapReduce, its map-side join extension and Bulk Synchronous Parallel. Each technique was independently applied to solve different graph problems such as calculation of single source shortest paths (SSSP) and collective classification of graph nodes by means of relational influence propagation. In this work, the authors consider the SSSP in unweighted graphs. Thus, the shortest paths problem is interpreted as the computation of the minimal number of edges that form a path from a chosen vertex to every other vertex in an unweighted graph. One of the main conclusions of this comparative study is that MapReduce can still remain the
only mature enough alternative for parallel processing of graph algorithms on huge datasets.

In Aridhi et al. (2015), the authors present a MapReduce-based approach for the task of distributed frequent subgraphs mining. The proposed method relies on a density-based partitioning technique that considers data characteristics. It uses the densities of graphs in order to partition the input data. Such a partitioning technique allows for balanced computational load over the distributed collection of machines and replaces the default arbitrary partitioning technique of MapReduce.

2. Hadoop architecture and MapReduce model

2.1. Hadoop framework

Hadoop is an open-source Apache project which provides solutions for reliable and scalable distributed computing. The first component of hadoop distribution is the Hadoop Distributed File System (HDFS) for data storage. The second one is the wide spread MapReduce programming paradigm (Dean and Ghemawat, 2008). Hadoop provides a transparent framework for both reliability and data transfers. It is the cornerstone of numerous systems which define a whole ecosystem (see Fig. 1) around it.

The current version of Hadoop is built on a new version of MapReduce, called MapReduce 2.0 (MRv2) or YARN. The implementation of the previous versions of MapReduce is referred to as MRv1.

A MRv1 Hadoop framework consists of a single Master node and multiple Slave nodes. The Master node consists of a JobTracker, TaskTracker, NameNode and DataNode. Each Slave node consists of a DataNode and TaskTracker. One can note that for failure resilience purposes, secondary NameNodes can be used to replicate the data of the NameNode. The JobTracker assumes the responsibility of distributing the software configuration to the Slave nodes, scheduling the job's component tasks to available TaskTracker nodes, monitoring them and re-assigning tasks to TaskTracker nodes if required. It is also responsible for providing the status and diagnostic information to the client. The TaskTracker executes the task as defined by the JobTracker.

MapReduce framework has been subjected to numerous improvements for years and a great overhaul has been reported in hadoop-0.23. The new version of MapReduce is called MRv2 or YARN. In a MRv2 Hadoop framework, the two major functionalities of the JobTracker (resource management and job scheduling/monitoring) have been split into separate daemons: a global ResourceManager and per-application ApplicationMaster. The ResourceManager is a persistent service that receives and runs MapReduce jobs. The per-application ApplicationMaster consists on negotiating resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the tasks. In MRv2 the Resource Manager (respectively the Node Manager) replaces the JobTracker (respectively the Task Tracker).

2.2. Hadoop distributed file system (HDFS)

HDFS (http://hadoop.apache.org/hdfs/) is a distributed, scalable, and portable file system. Typically, HDFS is composed of one single NameNode which is a master server. It manages the file system namespace and it is linked to a Datanode to manage data storage.

Files are split into blocks of a fixed size, which are distributed into a cluster of one or several Datanodes. Such structure implies that all the blocks of one file can be saved into several machines. Thus, access to a file can generate access to several machines. One major drawback is the file loss due to the unavailability of one machine. HDFS solves this problem by replicating each block across several machine nodes. The metadata information consists of partitions of the files into blocks and the distribution of these blocks on different Datanodes. Datanodes perform block creation, deletion, and replication upon instruction coming from the NameNode.

2.3. MapReduce programming model

MapReduce is a programming model for processing highly distributable problems across huge datasets using a large number of computers. The central features of MapReduce are two functions, written by a user: Map and Reduce. The Map function takes as input a pair and produces a set of intermediate key-value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function. The Reduce function accepts an intermediate key and a set of values for that key. It merges these values together to form a possible smaller set of values.

In Hadoop terminology, a ‘job’ consists of the execution of a Mapper and Reducer on a block of data. Hadoop distributes the Map/Reduce invocations across slave nodes by partitioning the input data into splits. These input splits are processed in parallel on different slave nodes.

Fig. 2 shows an execution workflow of a MapReduce. The execution workflow is made-up of two main phases:

(a) The Map phase, which contains the following steps:
1. The input file is split into several pieces of data. Each piece is called a split or a chunk.
2. Each slave node hosting a map task, called a mapper, reads the content of the corresponding input split from the distributed file system.
3. Each mapper converts the content of its input split into a sequence of key-value pairs and calls the user-defined Map procedure for each key-value pair. The produced intermediate pairs are buffered in memory.

4. Periodically, the buffered intermediate key-value pairs are written to a local intermediate file, called segment file. In each file, the data items are sorted by keys. A mapper node could host several segment files and its number depends on the number of reducer nodes. The intermediate data should be written into different files if they are destined to different reducer nodes. A partitioning function ensures that pairs with the same key are always allocated to the same segment file.

(b) The Reduce phase, made of the following steps:
1. On the completion of a map task, the reducer node will pull over its corresponding segments.
2. When a reducer reads all intermediate data, it sorts the data by keys. All occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort will be used. The reducer then merges the data to produce for each key a single value.
3. Each reducer iterates over the sorted intermediate data and passes each key-value pair to the user's reduce function.
4. Each reducer writes its result to the distributed file system.

3. Shortest path resolution with MapReduce

3.1. A MapReduce-based approach

Srirama et al. (2012) clearly defined that scientific algorithms could be divided into four classes. Each class represents the difficulty in adapting an algorithm to the hadoop MapReduce paradigm:

- Class 1. Algorithms that can be adapted as a single execution of a MapReduce model.
- Class 2. Algorithms that can be adapted as a sequential execution of a constant number of MapReduce models.
- Class 3. Algorithms where the content of one iteration is represented as an execution of a single MapReduce model.
- Class 4. Algorithms where the content of one iteration is represented as an execution of multiple MapReduce models.

The framework, for the shortest path resolution, that we promote leads to the third class and requires several iterations of one MapReduce model but only one MapReduce execution per iteration. Algorithms belonging to class 3 are generally difficult to parallelize efficiently. The shortest path framework introduced here permits to solve the shortest path with high quality solutions.

The proposed shortest path computation schema (Fig. 3) is composed of four steps: the step 1 consists in partitioning the initial graph into a set of subgraphs. A partition procedure (see Algorithm 2: Partition Into Subgraph) is called to generate the subgraphs. It takes the parameters including the subgraph size, the starting and ending positions of the shortest path and calculates the initial graph.

Step 2 consists in using a MapReduce-based computation module for creating an initial solution which is represented as a set of shortest paths on the subgraphs. All intermediate solutions in the computation schema use such representation. The shortest path on the initial graph will be created in step 4 using a concatenation procedure (see Algorithm 8: Concat).

As stressed in Fig. 4, the MapReduce-based computation module consists in uploading the subgraphs (each one contains a starting and an ending node) to HDFS and starting the computation in a parallel way due to the job parallelization mechanism of MapReduce. The computation of the shortest path on a subgraph is achieved on a slave node using a Map procedure (see Algorithm 6: Start Node Job). All obtained shortest paths will be copied from slave nodes to the master node. A Reduce procedure is used to collect the shortest paths and create an intermediate solution on the master.

In step 3, the intermediate solution is downloaded from HDFS to a local directory before applying an iterative improvement. Each iteration consists in generating a new set of subgraphs using a second partition procedure. This procedure takes a recently created intermediate solution to determine the dimension (including subgraph size, the starting and ending nodes) of each subgraph. It uses then the MapReduce-based computation module to create an intermediate solution. This step is stopped when the maximal number of iterations is reached.

Finally, step 4 consists in concatenating all shortest paths in an intermediate solution to create a shortest path on the initial graph. In the following, we present the algorithm of our approach (see Algorithm 1) which takes 4 parameters:

- $L$: subgraph size in km;
- $x_1, y_1$: longitude and latitude of the origin (starting position);
- $x_2, y_2$: longitude and latitude of the destination (ending position).

In Algorithm 1, lines 8–9 illustrate step 1 (create initial data) which consists in a basic call to the first partition procedure (see Algorithm 2: Partition Into Subgraph). The main idea of this partition procedure consists in dividing the “straight line” from origin to destination into
several segments, each segment being considered as the diagonal of a subgraph.

Lines 10–17 illustrate step 2 of Algorithm 1. The procedure Start_Node_Job (see Algorithm 6) allows starting the computation of the shortest path on a subgraph using MapReduce engine. It takes a subgraph and the GPS coordinates of the starting and ending positions as parameters. It is possible that a starting or an ending position does not correspond to a node of the subgraph. Therefore, a projection step is necessary to compute the nearest node of the subgraph from each GPS position before starting the shortest path computation. A Dijkstra algorithm is adapted here for the shortest path computation. The obtained local shortest paths are ranged in an intermediate solution (denoted by $SP$) using the procedure Retrieved_Shortest_Path which is called at line 17. It is assumed that $SP[j]$ is the shortest path of the $j$th subgraph. Lines 19–50 highlight step 3 of Algorithm 1 and are dedicated to the iterative improvement of the intermediate solution. The second partition procedure uses Create_Graph (see Algorithm 5) to create the subgraph induced by the middle of the shortest path of the current iteration (lines 22–23).

Algorithm 1. Hadoop shortest path framework (HSPF).

1. procedure SPP_Hadoop_Framework
2. input parameters
3. L: subgraph size in km
4. $x_1, y_1$: latitude and longitude of the starting position (origin)
5. $x_2, y_2$: latitude and longitude of the ending position (destination)
6. Max_It: the maximal number of iterations
7. begin
8. // Step 1. Create initial data
9. $(G, Dx, Dy, Ax, Ay) := \text{Partition}_{\text{Into}}_{\text{Subgraph}}(L, x_1, y_2, x_2, y_2)$;
10. // Step 2. Computation of the initial shortest path
11. for $j := 1$ to $|G|$ do
12. $\leftarrow$ upload $G[j]$ from local to HDFS
13. // Step 3. Improvement
14. partition procedure 2
15. $G := \text{Create}_{\text{Graph}}(\text{mid}(j))$;
16. repeat $(N-1)$ iterations
17. Concatenation procedure
18. set of shortest paths $SP$
19. // Step 4. concatenate the set of shortest paths
20. download data from HDFS
21. set of shortest paths $SP$
22.\end{verbatim}}
\begin{algorithm}
\begin{algorithmic}
\State $G[j]$ \textbf{will be copied from HDFS to a slave node} \Comment{SP}
\State $SP'[j]:=$\text{Start\_Node\_Job}(G[j], Dx[j], Dy[j], Ax[j], Ay[j]);
\EndFor
\Comment{wait until hadoop nodes have achieved work}
\State $SP:=$\text{Retrieved\_Shortest\_Path (}SP')$;
\State // \textbf{Step 3. Improvement of the shortest path}
\For {$i := 2$ to $\text{Max\_It}$}
\State if ($i \% 2 = 0$) Then
\State for $j := 1$ to $|\text{SP'}| - 1$ do
\State $\{Mx[j], My[j]\}:=$\text{the middle of the shortest path $SP[j]$};
\State $\{Mx[j] + 1, My[j] + 1\}:=$\text{the middle of the shortest path $SP'[j+1]$};
\State $G[j]:=$\text{Create\_Graph}(Mx[j], My[j], Mx[j] + 1, My[j] + 1);
\Comment{$G[j]$ will be copied from HDFS to a slave node} \Comment{SP}
\EndFor
\Comment{wait until hadoop nodes have achieved work}
\State $\text{Concat} (SP, SP[j])$;
\State return $\text{Concat}(SP, SP', \text{Max\_It})$;
\Comment{// \textbf{Step 4. Concatenation of shortest path}}
\EndIf
\EndFor
\State // \textbf{Step 3. Load output files from the HDFS to local directory}
\State $SP'$ from the HDFS to local directory
\Else
\State for $j := 0$ to $|SP'|$ do \textbf{// size of $SP' = = nb-1$
\State if($j = 0$) $\{Mx[j], My[j]\}:=(x1, y1)$;
\State else $\{Mx[j], My[j]\}:=$\text{the middle of the shortest path $SP'[j]$};
\EndIf
\State if($j = |SP'|$) $\{Mx[j + 1], My[j + 1]\}:=(x2, y2)$;
\State else $\{Mx[j + 1], My[j + 1]\}:=$\text{the middle of the shortest path $SP'[j+1]$};
\EndIf
\State $G[j + 1]:=$\text{Create\_Graph}(Mx[j], My[j], Mx[j + 1], My[j + 1]);
\Comment{upload $G[j + 1]$ from local to HDFS} \Comment{SP}
\State $SP'[j+1]:=$\text{Start\_Node\_Job}(G[j], Mx[j], My[j], Mx[j + 1], My[j + 1]);
\EndFor
\Comment{wait until hadoop nodes have achieved work} \Comment{SP}
\State $SP:=$\text{Retrieved\_Shortest\_Path (}SP')$;
\State $\text{Upload $G[j]$ from local to HDFS}$ \Comment{SP}
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

3.2. \textbf{Illustrative example}

As mentioned before, step 1 of the proposed algorithm focuses on the creation of initial data. Considering the graph of Fig. 5(a), nodes I and J are starting and ending positions of the shortest path. To create the subgraphs, the straight line from node I to J will be partitioned averagely into four segments. Two extremities of each segment serve as starting and ending positions of each subgraph, and a subgraph is generated using the procedure \text{Create\_Graph}. Fig. 5(b) shows the obtained subgraphs. It is important to note that the starting position of the $i$th subgraph is the ending position of the $(i-1)$th subgraph. For each subgraph, the starting or ending position (stressed in red in Fig. 5(b)) could not be a node in a subgraph. Thus, a projection procedure will be used to project the starting/ending position to a node of subgraph before the shortest path computation.

In the step 2, each subgraph is assigned to one slave node by the MapReduce framework taking into account the utilization rate and the availability of the computers of the cluster. Depending on the number of resources in the cluster and of the number of subgraphs, a computer can archive one or several jobs.

A job consists in computing the shortest path from the starting position to the ending position on a subgraph using the procedure \text{Start\_Node\_Job}. All partial shortest paths computed on slave nodes are gathered on the master node and represent an intermediate solution as stressed. This is achieved by the procedure \text{Retrieved\_Shortest\_Path} at line 17 in Algorithm 1.

The concatenation of all shortest paths on subgraphs (denoted as local shortest paths) in an intermediate solution gives a shortest path on the initial graph (denoted as initial shortest path). This initial shortest path is not optimal since the starting/ending positions of the subgraphs were inserted arbitrarily inside the solution. Fig. 6(a) shows an initial shortest path from the position I to the position J, which is obtained by concatenating the set of local shortest paths (denoted by $SP$). One assumes that $SP[j]$ is the shortest path on the $j$th subgraph. We can note that the obtained initial shortest path passes through all positions stressed in red in Fig. 6(a), i.e. starting and ending positions of each subgraph.

The third step consists in the optimization of the initial shortest path considering the middle of each partial shortest path previously

**Fig. 5.** The First partition of subgraphs. (a) Initial graph and (b) set of subgraphs.
computed by slave nodes. The main idea of step 3 consists in adjusting the positions stressed in red in Fig. 6(a) (except for positions \( I \) and \( J \) which are respectively the starting and ending positions of the initial shortest path). For this purpose, an iterative improvement is implemented. As mentioned in Algorithm 1, the number of subgraphs is different between an odd iteration and an even iteration. Fig. 6(b) gives a shortest path at an odd iteration: considering \( SP \) the intermediate solution obtained at the precedent iteration. A node \( M_j \) (with its coordinates \( M_{xj}, M_{yj} \)) is the node corresponding to the middle of a shortest path \( SP[j] \) (the nearest node to the middle of the path). In the step of improvement, the subgraph creation consists in encompassing the nodes \( M_j \) and \( M_{j+1} \) (i.e. they are considered as starting and ending position of subgraph). In this example (see Fig. 6(b)), three subgraphs have been created.

As shown in Fig. 7(a), the obtained set of local shortest paths is denoted by \( SP' \). An intermediate solution obtained at an even iteration does not give an initial shortest path: the concatenations of all local shortest paths do not give the path from position \( I \) to position \( J \). It is necessary to include the path from the node \( I \) to \( M_1 \) and the path from \( M_4 \) to \( J \) in order to form a solution to the initial problem. Therefore, we have to use the set of local shortest paths obtained at the precedent iteration. Fig. 7(b) shows a shortest path on the initial graph.

3.3. Partition into subgraphs

3.3.1. Principle

The subgraphs partition procedure (in the initial data creation step) is described by procedure Partition Into Subgraph (Algorithm 2). It returns a set of subgraphs and receives three parameters:

- \( L \): length of the subgraph in km;
- \( x_1, y_1 \): longitude and latitude of the starting position (origin);
- \( x_2, y_2 \): longitude and latitude of the ending position (destination).

This procedure consists in dividing the “straight line” from origin to destination into several segments. Each segment is used to define a subgraph. It starts by computing the distance as the crow flies between the origin and the destination using the procedure

---

**Fig. 6.** The Second partition procedure. (a) An initial shortest path and (b) set of subgraphs.

**Fig. 7.** Solutions in different steps. (a) An intermediate solution and (b) the shortest path on the initial graph.
Compute_Distance (see Section 3.3.2). The obtained distance is then used to determine the number of subgraphs. Each subgraph is defined by its origin \((Px, Py)\) and destination \((Cx, Cy)\) positions. Note please, that the origin position of the \(i\)th subgraph is the destination of the \((i-1)\)th subgraph. The destination of the \(i\)th subgraph is obtained using the procedure Finding Partition Position (see Section 3.3.3). A subgraph is extracted from the OpenStreetMap by invocating of the procedure Create Graph (see Section 3.3.4). Note please that a predefined margin is used to increase the graph size in order to favor the connectivity between two successive subgraphs.

Algorithm 2. Partition Into Subgraph

```plaintext
procedure Partition Into Subgraph
1. input parameters
2. L: length of subgraph in km
3. x1, y1: longitude and latitude of the starting position
4. x2, y2: longitude and latitude of the ending position
5. begin
6. D = Compute_Distance (x1,y1,x2,y2);
7. d = \(\frac{D}{2}\); // diagonal length
8. nb = D/d;
9. Px = x1;
10. Py = y1;
11. for i = 1 to nb do
12.  if i = nb then
13.    Cx = x2;
14.    Cy = y2;
15.    else
16.    \((Cx,Cy) = \text{Finding partition position}(x1,y1,x2,y2,d,i)\);
17.  end if
18.  G[i] = Create Graph(Px,Py,Cx,Cy);
19.  Px = Cx;
20.  Py = Cy;
21.  end for
22.  Return G;
23. end
```

3.3.2 Distance between two coordinates GPS: Compute_Distance

The procedure Compute_Distance measures the distance as the crow flies between two GPS positions along the surface of the earth. Considering the earth as a perfect sphere and a great circle is the intersection of the sphere and a plane which passes through the center of the sphere. As shown in Fig. 8, the distance between \(A\) and \(B\) is the length of a minor arc of the great circle which passes \(A\) and \(B\), where \(O\) is the center of the sphere and \(r\) the radius. The length of the arc \(AB\) is equal to \(\frac{(\pi x 2 r \theta)}{2 \pi} = r \theta\), where \(\theta\) is the angle in radians between \(AO\) and \(OB\).

Assuming that the coordinates of the two points \(A\) and \(B\) are respectively \((\text{lat1, lng1})\) and \((\text{lat2, lng2})\), the differences in latitude and longitude are denoted as follows: \(\Delta \text{lat} = \text{lat2} - \text{lat1}\), \(\Delta \text{lng} = \ln g2 - \ln g1\). Using the haversine formula (Robusto, 1957; Sinnot, 1984), the angle \(\theta\) is calculated as follows: \(\theta = 2 \times \arcsin \sqrt{\sin^2(\frac{\Delta \text{lat}}{2}) + \cos(\text{lat1}) \cos(\text{lat2}) \sin^2(\frac{\Delta \text{lng}}{2})}\). The length of the arc \(AB\) is then equal to:

\[
\|AB\| = 2r \arcsin \sqrt{\sin^2(\frac{\Delta \text{lat}}{2}) + \cos(\text{lat1}) \cos(\text{lat2}) \sin^2(\frac{\Delta \text{lng}}{2})}
\]

3.3.3 Finding the dividing position: Finding_partition_position

As mentioned before, the partition into subgraphs consists in dividing the “straight line” from origin to destination into several segments. As shown in Fig. 9, the straight line is the shortest surface-path between the GPS coordinates (distance as the crow flies). It is in fact a minor arc of the great circle between two GPS positions.

Fig. 9(a) gives an example of the straight line to travel from Paris to Beijing using Google earth. The line travels across approximately Paris, north of Moscow, Siberia, Mongolia and Beijing on the classical 2D map representation (Fig. 9(b)). One can distinguish it from the line which links directly two nodes (noted projected line). More precisely, the distance as the crow flies of the arc and the projected line is respectively 8200 km and 9000 km. Clearly, the projected line
cannot be used as guideline to evaluate the position of a shortest path except for small scale graph where the difference between the arc and the projected line can be neglected.

In this paper, the arc is used as straight line to partition the subgraphs. As mentioned in Section 3.2, the straight line will be partitioned averagely into several segments. Two extremities of each segment serve as starting and ending positions of each subgraph. The destination position of ith subgraph is the origin of (i + 1)th subgraph.

Algorithm 3 describes the computation of partition positions on the straight line. It takes 4 parameters:

- d: the diagonal length of the subgraph in km;
- i: ith subgraph;
- Ax, Ay: longitude and latitude of the origin;
- Bx, By: longitude and latitude of the destination.

The computation of a partition position is composed of 4 steps.

The first step consists in converting the GPS position of the origin (point A) and the destination (point B) into Cartesian coordinate position. The formula of conversion is given by Algorithm 4.

The second step calculates the normal vector \((u, v, w)\) of the plane \(AB\). As described in Section 3.3.2, the length of \(\overline{AB} = r\theta\) where \(r\) is the radius of the earth and \(\theta\) is the angle in radians between \(AO\) and \(OB\). The length of \(\overline{AC}\) is known as \(d_{\text{rev}}\). Then the angle in radians between \(\overline{AO}\) and \(\overline{OC}\) (noted \(\gamma\)) can be obtained using the following equation:

\[
\gamma = \frac{\text{length of } \overline{AC}}{\text{length of } \overline{AB}} \times \frac{d_{\text{rev}}}{r}
\]

As shown in line 18, we have \(\gamma = \text{arcsin}(\frac{d_{\text{rev}}}{r})\). With the value of \(\gamma\), the position of point C can be obtained by applying a rotation matrix on the position of the point A.

If \((u, v, w)\) is a unit vector which represents the axle of rotation, then the rotation matrix \(R\) is given as follows:

\[
R = \begin{bmatrix}
(1 - \cos \gamma)u^2 + \cos \gamma & (1 - \cos \gamma)uv - w \sin \gamma & (1 - \cos \gamma)uw + v \sin \gamma \\
(1 - \cos \gamma)uv + w \sin \gamma & (1 - \cos \gamma)v^2 + \cos \gamma & (1 - \cos \gamma)vw - u \sin \gamma \\
(1 - \cos \gamma)uw - v \cos \gamma & (1 - \cos \gamma)vw + u \sin \gamma & (1 - \cos \gamma)w^2 + \cos \gamma
\end{bmatrix}
\]

Algorithm 3. Partition position computation

1. procedure Finding_partition_position
2. input parameters
3. \(d\): the diagonal length of subgraph in km
4. \(i\): ith subgraph
5. \(Ax, Ay\): longitude and latitude of the starting position (point A)
6. \(Bx, By\): longitude and latitude of the ending position (Point B)
7. begin
8. //step 1: convert the GPS position into Cartesian coordinate position
9. \((x1,y1,z1) := \text{GPS_to_cartesian}(Ax, Ay)\);
10. \((x2,y2,z2) := \text{GPS_to_cartesian}(Bx, By)\);
11. //step 2: calculate the normal vector \((u,v,w)\) of the plane \(OAB\)
12. \((x0,y0,z0) := (0,0,0)\); //the center of the earth (Point O)
13. \(u := (y1 - y2)z2 - z1y2 - y0z2\);
14. \(v := (z1 - z2)y2 - x1z2 - x0y2 - y0z2\);
15. \(w := (x1 - x2)y2 - y1z2 - z0x2 - x0z2\);
16. //step 3: finding the partition position \((x3,y3,z3)\) in Cartesian coordinates
17. \(r := \text{radius of the earth}\);
18. \(\gamma := \text{arcsin}(\frac{d_{\text{rev}}}{r})\); //angle of rotation
19. \(C := \cos(\gamma)\);
20. \(S := \sin(\gamma)\);
21. \(x3 := (((1 - C)\cos\gamma + C)\cos\gamma1 + ((1 - C)\sin\gamma - S)\sin\gamma1 + ((1 - C)\sin\gamma + S)\cos\gamma1\);
22. \(y3 := (((1 - C)\cos\gamma + C)\sin\gamma1 + ((1 - C)\sin\gamma - S)\cos\gamma1 + ((1 - C)\sin\gamma + S)\sin\gamma1\);
23. \(z3 := (((1 - C)\cos\gamma + C)\sin\gamma1 + ((1 - C)\sin\gamma - S)\cos\gamma1 + ((1 - C)\sin\gamma + S)\sin\gamma1\);
24. //step 4: convert the Cartesian coordinate position into GPS position
25. \(Kx := \text{arcsin}(y3/(R\cos(Kx)))\);
26. if \(Kx > 0\) then
27. \(Ky := \text{arcsin}(y3/(R\cos(Kx)))\);
28. else
29. \(Ky := \text{arcsin}(y3/(R\cos(Kx)))\);
30. end if
31. return \((Kx, Ky)\);
32. end

Algorithm 4. Conversion of a GPS position into a Cartesian position

1. procedure GPS_to_cartesian
2. input parameters
3. \(x1, y1\): longitude and latitude
4. begin
5. \(r := \text{radius of the earth}\);
6. \(x := \text{r}\cos(y1)\cos(x1)\);
7. \(y := \text{r}\cos(y1)\sin(x1)\);
8. \(z := \text{r}\sin(y1)\);
9. return \((x, y, z)\);
10. Fin

The last step consists in converting the position of point C from Cartesian coordinates into GPS coordinates.

3.3.4. Partitioning of a subgraph from the initial graph:
Create Graph procedure

Algorithm 5 describes the procedure for creation of a subgraph. The subgraph data is extracted from the OpenStreetMap source and it contains a set of nodes and a set of arcs. A subgraph is bounded by its origin and destination (input parameters). The boundary is increased by introducing a predefined margin that favors the connectivity between two successive subgraphs (lines 6–9). The creation of a subgraph consists in including all nodes/arc in the boundary (lines 10–18). The origin and destination is defined respectively as starting and ending position for shortest path computation (lines 19–20).

Algorithm 5. Creation of a subgraph

1. procedure Create_Graph
2. input parameters
3. \(Ax, Ay\): longitude and latitude of the origin
4. \(Bx, By\): longitude and latitude of the destination
5. begin
6. M := predefined margin
7. G := the initial graph data from OpenStreetMap
8. \((Ax',Ay')\); := update the position \((Ax,Ay)\) taking into account of M;
9. \((Bx',By')\); := update the position \((Bx,By)\) taking into account of M;
10. \(g:=\text{new the subgraph}\)
11. for each node \(n\) in \(G\) do
12. if \(n\) in the bound \((Ax',Ay',Bx',By')\) then
13. Put the node \(n\) into \(G'\);
14. for each arc \(e\) in \(G'\) and \(e\) connected to \(n\) and \(e\) not in \(g\) do
15. Put the arc \(e\) into \(g\);
16. end for
17. end if
18. end for
19. define \((Ax,Ay)\) as starting position for \(g\);
20. define \((Bx,By)\) as ending position for \(g\);
21. return \(g\);
22. end

3.4. Description of the Start_Node_Job procedure

This section focuses on the sub-problem which consists in managing the Hadoop nodes. This procedure (Algorithm 6) takes 3 parameters:

- \(g\): subgraph data;
- \(x1, y1\): longitude and latitude of the origin;
- \(x2, y2\): longitude and latitude of the destination.

The first step is the projection of the starting and ending positions of the nodes of the subgraph (lines 7–9 of Algorithm 6). The projection procedure is detailed in Section 3.4.1. The second step of the algorithm consists in computing the shortest path between two projected nodes. The shortest path computation procedure is described in Section 3.4.2.

Algorithm 6. Starting job on node

1. procedure Start_Node_Job
2. input parameters
3. \(g\): subgraph data
4. \(x1, y1\): longitude and latitude of the GPS position
5. \(x2, y2\): longitude and latitude of the destination
6. begin
7. // step 1. Compute projection
8. \(N1:=\text{Projection}(g,x1,y1)\);
9. \(Nj:=\text{Projection}(g,x2,y2)\);
10. // step 2. Compute shortest paths
11. \(SP:=\text{Shortest_Path_Computation}(g,Ni,Nj)\);
12. return \(SP\);
13. end

3.4.1. Projection of latitude/longitude position to a node of a subgraph

The projection procedure (Algorithm 7) consists in computing the nearest node of the subgraph for a given GPS position. It takes 2 parameters:

- \(g\): subgraph data;
- \(x1, y1\): longitude and latitude of the GPS position

The first step of Algorithm 7 is to find the nearest arc using a loop from line 7 to 13. As shown in Fig. 10, the distance from a GPS position \(P\) to an arc \((A,B)\) is defined using:

- The distance between the GPS position \(P\) and its projection \(e'\) on the arc if the orthogonal projection is possible, denoted by \(d_{P,e'}\);
- The distance between the GPS position \(P\) and the node \(A\), denoted by \(d_{P,A}\);
- The distance between the GPS position \(P\) and the node \(B\), denoted by \(d_{P,B}\).

The distance from \(P\) to an arc \((A,B)\) is the minimal value between \(d_{P,e'}, d_{P,A}\) and \(d_{P,B}\).

The second step (lines 14–20) of Algorithm 7 aims to find the nearest node. Note please that \((A',B')\) is the arc with the minimal distance from the GPS position \(P\). The distances from the GPS position to each node of the arc \((A',B')\) are computed, and the node with the smallest distance will be returned as the projected node.

Algorithm 7. Projection

1. procedure Projection
2. input parameters
3. \(g\): subgraph data
4. \(x1,y1\): longitude and latitude of the GPS position
5. begin
6. \(\text{min\_dist}\;:=\;+\infty\);
7. for each arc \((A,B)\) in \(g\) do
8. \(\text{dist}\;:=\text{distance form }\(x1, y1\) to arc \((A,B)\);
9. if \(\text{dist} < \text{min\_dist}\) then
10. \(\text{min\_dist}\;:=\text{dist}\);
11. \((A',B')\;:=\(A,B)\);
12. end if
13. end for
14. \(\text{dist\_PA}\;:=\text{distance form }\(x1, y1\) to node \(A')\);
15. \(\text{dist\_PB}\;:=\text{distance form }\(x1, y1\) to node \(B')\);
16. \(N1\;:=\(B')\);
17. if \(\text{dist\_PA} < \text{dist\_PB}\) then
18. \(N1\;:=\(A')\);
19. end if
20. return \(N1\);
21. end

3.4.2. Shortest path computation and its computational complexity

The procedure Shortest_Path_Computation is based on Dijkstra’s algorithm. It takes 3 parameters:

- \(g\): subgraph data;
- \(O\): the origin node;
- \(D\): the destination node.

The procedure Shortest_Path_Computation returns the shortest path form the origin node to the destination node on a subgraph. The proposed approach focuses on real-road networks which are very sparse graphs (Schultes, 2008). In a sparse graph, the degree of each of the nodes can be bounded by a constant, independent of the size of the graph. As stressed by Dreyfus and Law (1977) and Denardo (1982), Dijkstra’s algorithm is the most time efficient algorithm for computing shortest path exactly on sparse graphs.

It is possible to provide efficient implementation of Dijkstra’s algorithm using the heap data structure (Wirth, 1976; Levy, 1994; Kaplan, 1999; Reddy and Yang, 2003) in \(O(nm \log(n))\) where \(n\) is the number of nodes and \(m\) the number of outgoing arcs of a sparse graph. Note please that the degree of each of the node in a real-road network is usually smaller than five (Schultes, 2008). The complexity could be \(O(n \log(n))\) when \(n \gg 5\).
In the proposed approach, the shortest path computation is not realized on the initial graph but on the set of subgraphs. To gain insight into the potential gain in computation time achievable by the proposed, we assume that the number of subgraphs is exactly equal to the number of slave nodes, the size of the initial graph is $n'$ and the size of subgraph is $n"$. For each subgraph, the complexity for shortest path computation is about $O(n' \log(n''))$. If the number of iterations assigned to the proposed approach (denoted by $Max_{it}$ in the Algorithm 1) is set to 1, the relative efficiency of the proposed approach is $(O(n' \log(n')) / O(n' \log(n'')))$. Similar remarks have been achieved by Chou et al. (1998) who investigated the complexity using ratio of efficiency.

3.4.3. Description of the procedure Retrieved_Shortest_Path

As stressed before, this step collects first the set of shortest paths from all subgraphs and all paths in a single one taking advantages of the map/reduce scheme. To achieve this step, it is necessary to identify the shortest path on each subgraph (computed by slave nodes).

3.5. Description of the concatenation procedure: Concat

Fig. 11(a) shows an intermediate solution (denoted by $SP$) at an odd iteration, in which each square represents one subgraph and $SP[j]$ is the shortest path of the $j^{th}$ subgraph. Note please that a solution for the initial shortest path problem can be obtained by concatenating directly the set of shortest paths on subgraphs. As mentioned in Section 3.2, an intermediate solution obtained at an even iteration does not give an initial shortest path. Fig. 11(b) gives an intermediate solution at an even iteration (denoted by $SP'$). The concatenation of all elements in the $SP'$ gives the path from node $M_i$ to node $M_j$ (Fig. 11(b)). It is necessary to extend this path to reach the nodes $I$ and $J$ in order to obtain a solution for the initial problem. Note please that at the even iteration $(i + 1)$, node $M_i$ is the middle of the shortest path on the subgraph $j$ at iteration $i$, i.e. the middle of $SP[j]$. Thus, the path from the node $I$ to $M_1$ is the first half part of the shortest path on subgraph $1$ (i.e. $SP[1]$) at the iteration $i$. Idem, the path from $M_4$ to $J$ is the second half part of the shortest path on subgraph $4$ (i.e. $SP[4]$).

The concatenation procedure (Algorithm 8) takes three parameters including the intermediate solutions of the two last iterations and the maximal number of iterations:

- $SP$: the intermediate solution at the last odd iteration;
- $SP'$: the intermediate solution at the last even iteration;
- $Max_{it}$: the maximal number of iterations.

This procedure returns the obtained initial shortest path. In Algorithm 8, lines 13–19 consist in concatenating the set of shortest paths on subgraphs into an intermediate solution. If the maximal number of iterations is an even number, lines 9–12 and lines 20–22 allow extending the path to the starting and the ending positions.

Algorithm 8. Concatenation procedure

```plaintext
1. procedure Concat
2. input parameters
3. $SP$: the intermediate solution at the last odd iteration
4. $SP'$: the intermediate solution at the last even iteration
5. $Max_{it}$: the maximal number of iterations
6. begin
7. $SPP:=\emptyset$;
8. nb:=|$SP'$|;
9. if ($Max_{it}$%2=0) then
10. $SPP:=\text{first half part of the shortest path }SP[1]$;
11. nb:=|$SP'$|;
12. else
13. for $i:=1$ to nb do
14. if ($Max_{it}$%2=0) then
15. $SPP:=SPP \cup SP[i]$;
16. else
17. $SPP:=SPP \cup SP[i]$;
```

Fig. 11. Two types of intermediate solution. (a) the set of SP at an odd iteration $i$ and (b) the set of SP at an even iteration $i+1$. 
4. Numerical experiments

All the experiments have been performed on a Hadoop cluster of 10 nodes corresponding to a set of homogenous computers connected through a local area network. Each computer is based of 10 nodes corresponding to a set of homogenous computers connected through a local area network. Each computer is based on an AMD Opteron 2.3 GHz CPU under Linux (CentOS 64-bit). The number of cores used is set to 1 for all tests. The computational experiments have been achieved on Hadoop 0.23.9.

4.1. Instances of interest: 15 real life road trips

The used French road network graph has been extracted from OpenStreetMap. The route profile is set to pedestrian. A set of 15 instances have been created which are based on the same graph. Each instance consists of an origin and of a destination latitude/longitude position (see Table 1). For each instance, each subgraph size $D \in \{15, 30, 50, 100, 150, 200, \ldots\}$ kilometers is considered. Note please that the graph will not be split when $D = \infty$, i.e. it allows to get the optimal solution. The margin used for the subgraph creation is 5 km for each instance. The value of margin has been tuned after an experimental study which has proved that this value is large enough for real-road networks commonly available in industrialized countries. The French road network graph can be downloaded from: http://www.isima.fr/~iacomme/OR_hadoop/.

4.2. Influence of the subgraphs size on the results quality

Table 2 shows the computational results for the 7 values of the subgraph size ranging from 15 to 200 km. The optimal values

<table>
<thead>
<tr>
<th>Ins.</th>
<th>Opt.</th>
<th>$D=15$</th>
<th>$D=30$</th>
<th>$D=50$</th>
<th>$D=100$</th>
<th>$D=150$</th>
<th>$D=200$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Val.</td>
<td>Gap</td>
<td>Val.</td>
<td>GAP</td>
<td>Val.</td>
<td>GAP</td>
<td>Val.</td>
</tr>
<tr>
<td>1</td>
<td>87.1</td>
<td>96.8</td>
<td>11.11</td>
<td>92.3</td>
<td>5.95</td>
<td>87.1</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>841.2</td>
<td>-</td>
<td>-</td>
<td>574.3</td>
<td>6.87</td>
<td>557.7</td>
<td>3.78</td>
</tr>
<tr>
<td>3</td>
<td>537.4</td>
<td>574.3</td>
<td>6.87</td>
<td>557.7</td>
<td>3.78</td>
<td>540.5</td>
<td>2.25</td>
</tr>
<tr>
<td>4</td>
<td>355.6</td>
<td>381.6</td>
<td>7.31</td>
<td>375.1</td>
<td>5.49</td>
<td>369.6</td>
<td>3.95</td>
</tr>
<tr>
<td>5</td>
<td>883.8</td>
<td>954.4</td>
<td>7.99</td>
<td>933.2</td>
<td>5.59</td>
<td>926.4</td>
<td>4.82</td>
</tr>
<tr>
<td>6</td>
<td>948.2</td>
<td>1024.0</td>
<td>8.84</td>
<td>1007.5</td>
<td>6.25</td>
<td>989.6</td>
<td>4.37</td>
</tr>
<tr>
<td>7</td>
<td>349.6</td>
<td>377.0</td>
<td>7.85</td>
<td>372.4</td>
<td>6.55</td>
<td>361.8</td>
<td>3.51</td>
</tr>
<tr>
<td>8</td>
<td>750.1</td>
<td>814.3</td>
<td>8.56</td>
<td>792.5</td>
<td>5.66</td>
<td>782.7</td>
<td>4.35</td>
</tr>
<tr>
<td>9</td>
<td>611.7</td>
<td>639.2</td>
<td>4.50</td>
<td>682.2</td>
<td>11.52</td>
<td>639.7</td>
<td>4.57</td>
</tr>
<tr>
<td>10</td>
<td>773.1</td>
<td>852.5</td>
<td>15.65</td>
<td>828.2</td>
<td>12.35</td>
<td>806.6</td>
<td>9.42</td>
</tr>
<tr>
<td>11</td>
<td>264.2</td>
<td>282.6</td>
<td>6.95</td>
<td>273.1</td>
<td>3.36</td>
<td>272.2</td>
<td>3.00</td>
</tr>
<tr>
<td>12</td>
<td>192.6</td>
<td>206.8</td>
<td>7.36</td>
<td>199.0</td>
<td>3.31</td>
<td>197.8</td>
<td>2.72</td>
</tr>
<tr>
<td>13</td>
<td>152.4</td>
<td>-</td>
<td>-</td>
<td>167.7</td>
<td>10.01</td>
<td>163.1</td>
<td>7.01</td>
</tr>
<tr>
<td>14</td>
<td>872.3</td>
<td>974.4</td>
<td>11.70</td>
<td>941.3</td>
<td>7.91</td>
<td>921.0</td>
<td>5.59</td>
</tr>
<tr>
<td>15</td>
<td>763.3</td>
<td>830.8</td>
<td>8.84</td>
<td>824.2</td>
<td>7.97</td>
<td>807.9</td>
<td>5.84</td>
</tr>
<tr>
<td>AVG.</td>
<td>Gap</td>
<td>8.37</td>
<td>8.64</td>
<td>4.83</td>
<td>4.36</td>
<td>4.46</td>
<td>1.66</td>
</tr>
</tbody>
</table>
come from the results of $D = +\infty$. Column “Ins.” contains the instance identifier; column “Opt.” gives the optimal value obtained by a Dijkstra algorithm on the initial graph; column “Val.” reports the obtained value with 2 iterations; and “Gap” contains the gap in percent between the value and the optimal value of each size.

The results in Table 2 shows that the average gap has been improved from 8.37% to 1.11% by increasing the size of subgraphs. Moreover, for the instances with $D=200$ km, the optimal values were found for 6 over 15 instances. On the other hand, no feasible solutions are found for the instance number 2 when $D \leq 30$ and for the instance number 13 when $D \leq 15$.

The shortest path problem cannot be divided into sub-problems for Hadoop resolution with a guarantee of optimality. Moreover, depending on the subgraph size and on the graph nodes distribution some problems of connectivity could arise. An example is given on Fig. 12, where the shortest path cannot be computed since the node A (Montpellier) cannot be linked by a path to the node B (Marseille); there is no path in the subgraph presented by the square number 3 in Fig. 12. In such a situation, the process stops with a path composed by a set of non-connected partial shortest paths.

4.3. Influence of the subgraphs size on the computational time

Table 3 shows the computational time for different subgraph sizes with 2 iterations. Column “Ins.” contains the instance identifier; TT(s) gives the total computational time of the framework (data structure initialization plus projection on the graph plus shortest path execution) in seconds; TS(s) gives shortest path computational time in seconds.

The results in Table 3 show that the direct resolution presents an average computational time about 383.26 s and an average Dijkstra computational time is about 4 s. With $D = 15$, the total computational time of the Hadoop framework is about 96 s on average which is 4 times better than the direct resolution. The best speed factor is obtained for the square of size 100 km where the average computational time is about 58 s. The computational time required by the shortest path is less than 1% of the total time. It is possible to conclude that the data structure management and the graph projection are time consuming operations. This conclusion is not surprising since for example, the whole graph of instance 1 is about 5 GB of data to manage (382 s are required to initialize the data structure and realize the extra computation in memory and 0.19 s are needed to execute the shortest path computation).

4.4. Ratio of Hadoop framework efficiency versus a direct resolution

Table 4 shows the ratio between the direct resolution computational time and Hadoop framework computational time with 2 iterations. Column “Ins.” contains the number of instances; TT(s) gives the total computational time in seconds; Column “Ratio” gives Ratio between computational times of each approach.

The performance of the proposed approach in terms of both computational time and quality of the solution depends on the square size. For $D = 15$, the acceleration rate provided by the Hadoop framework is about 4.6 on average and some ratios are about 7 times depending on the instances. For example, for the instances 1 and 12, the ratio is respectively 7.1 and 7.8. The difference between instances probably depends on the graph structure, especially the density of

![Fig. 12. Failed computation of the shortest path with several subgraphs.](image-url)
the graph. Table 4 pushes us into considering that there exists an optimal square size favoring the computational time. This size is $D = 100$, where the maximal ratio (on average) of 6.7 has been obtained. As shown in Table 4, too large square (with 200 km of size) decreases the ratio and allows lower performances.

4.5. Influence of the number of Hadoop slave nodes

The second topic of interest is the influence of the number of slave nodes on the computational time. Table 5 gives the impact of the number of slave nodes on the computational time for $D = (15, 50, 100, 200)$. We report the computational time with one slave node (column "1-sn"), 4 slave nodes (column "4-sn"), and 8 slave nodes (column "8-sn"). The average computational time varies strongly and it depends on the number of slaves. For example, the computation time of instance 1 has decreased from 84 s to 54 s with a number of slaves from 1 to 8 with $D = 15$. Similar remarks hold for all instances. However, increasing the number of slaves does not have a positive impact on the computational time. Let us consider for example the instance 4 for $D = 50$ where the computational time remains equal for 4 slaves and 8 slaves. Such numerical analysis pushes us into considering that for each instance, and for each subgraph size, there exist an optimal number of slaves that depends on the instance and on the parameter D.

4.6. Influence of the number of iterations assigned to the improvement step of the algorithm

Table 6 shows the impact of the number of iterations on the quality of solutions. In this table, we report the relative gap between the obtained value and the optimal value for $D = (15, 50, 100, 200)$ with one iteration (column "1-it.") and two iterations (column "2-it.") and three iterations (column "3-it."). It is possible to state that the proposed algorithm offers the best price-performance ratio.

The number of iterations assigned to the part 3 of the algorithm is responsible for the solution improvement. On average, for square size of 15 km, the average gap, decreases from 20.77% to 8.59% which represents a ratio of improvement of 2.4. Similar remark holds for all square sizes. Let us note that solutions about less than 1% of the optimal solution (on average) are provided with 3 iterations and $D = 200$.

5. Concluding remarks

The hadoop framework is a new approach for operational research algorithms. Our contribution stands at the crossroads of the optimization research community and the MapReduce community. We proved that there is a great interest in defining new approaches taking advantages of research in the area of information system, data mining and parallelization. A MapReduce-based approach is proposed for solving the shortest path problem in large-scale real road networks. The proposed approach is tested on a graph modeling the French road network extracted from OpenStreetMap. The experimental results show that such approach achieves significant gain of computational time. As future work, we are interested in adapting Operational Research algorithms to such framework, especially for those which could be more easily to take advantages of parallel computing.

### Table 4

<table>
<thead>
<tr>
<th>Ins.</th>
<th>Proposed approach with a Dijkstra algorithm</th>
<th>Direct resolution</th>
<th>TT(s)</th>
<th>TT(s)</th>
<th>Ratio</th>
<th>TT(s)</th>
<th>TT(s)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>D=15</td>
<td>D=50</td>
<td></td>
<td>D=100</td>
<td>D=200</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TT(s)</td>
<td>TT(s)</td>
<td></td>
<td>TT(s)</td>
<td>TT(s)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>383</td>
<td></td>
<td>49</td>
<td>78</td>
<td>54</td>
<td>71</td>
<td>48</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>391</td>
<td></td>
<td>127</td>
<td>3.1</td>
<td>62</td>
<td>63</td>
<td>58</td>
<td>6.7</td>
</tr>
<tr>
<td>3</td>
<td>398</td>
<td></td>
<td>106</td>
<td>3.7</td>
<td>61</td>
<td>6.4</td>
<td>59</td>
<td>6.6</td>
</tr>
<tr>
<td>4</td>
<td>386</td>
<td></td>
<td>75</td>
<td>5.1</td>
<td>52</td>
<td>7.4</td>
<td>58</td>
<td>6.7</td>
</tr>
<tr>
<td>5</td>
<td>389</td>
<td></td>
<td>138</td>
<td>2.8</td>
<td>73</td>
<td>5.3</td>
<td>65</td>
<td>6.0</td>
</tr>
<tr>
<td>6</td>
<td>388</td>
<td></td>
<td>149</td>
<td>2.6</td>
<td>75</td>
<td>5.2</td>
<td>74</td>
<td>5.2</td>
</tr>
<tr>
<td>7</td>
<td>384</td>
<td></td>
<td>73</td>
<td>5.3</td>
<td>52</td>
<td>7.4</td>
<td>53</td>
<td>7.2</td>
</tr>
<tr>
<td>8</td>
<td>389</td>
<td></td>
<td>121</td>
<td>3.2</td>
<td>65</td>
<td>6.0</td>
<td>56</td>
<td>7.0</td>
</tr>
<tr>
<td>9</td>
<td>398</td>
<td></td>
<td>102</td>
<td>3.8</td>
<td>55</td>
<td>7.0</td>
<td>55</td>
<td>7.0</td>
</tr>
<tr>
<td>10</td>
<td>389</td>
<td></td>
<td>110</td>
<td>3.5</td>
<td>56</td>
<td>6.9</td>
<td>56</td>
<td>6.9</td>
</tr>
<tr>
<td>11</td>
<td>385</td>
<td></td>
<td>61</td>
<td>6.3</td>
<td>50</td>
<td>7.7</td>
<td>55</td>
<td>7.0</td>
</tr>
<tr>
<td>12</td>
<td>384</td>
<td></td>
<td>54</td>
<td>7.1</td>
<td>51</td>
<td>7.4</td>
<td>58</td>
<td>6.6</td>
</tr>
<tr>
<td>13</td>
<td>384</td>
<td></td>
<td>47</td>
<td>8.2</td>
<td>52</td>
<td>7.4</td>
<td>59</td>
<td>6.5</td>
</tr>
<tr>
<td>14</td>
<td>389</td>
<td></td>
<td>119</td>
<td>3.3</td>
<td>69</td>
<td>5.6</td>
<td>59</td>
<td>6.6</td>
</tr>
<tr>
<td>15</td>
<td>389</td>
<td></td>
<td>122</td>
<td>3.2</td>
<td>61</td>
<td>6.4</td>
<td>63</td>
<td>6.2</td>
</tr>
<tr>
<td>AVG.</td>
<td>387.0</td>
<td></td>
<td>96.9</td>
<td>4.6</td>
<td>59.3</td>
<td>6.6</td>
<td>58.4</td>
<td>6.7</td>
</tr>
</tbody>
</table>

### Table 5

<table>
<thead>
<tr>
<th>Ins.</th>
<th>Proposed approach</th>
<th>$D=15$</th>
<th>$D=50$</th>
<th>$D=100$</th>
<th>$D=200$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1-sn</td>
<td>4-sn</td>
<td>8-sn</td>
<td>1-sn</td>
</tr>
<tr>
<td>1</td>
<td>84</td>
<td>49</td>
<td>54</td>
<td>55</td>
<td>48</td>
</tr>
<tr>
<td>2</td>
<td>613</td>
<td>127</td>
<td>76</td>
<td>247</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>457</td>
<td>106</td>
<td>65</td>
<td>181</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>281</td>
<td>75</td>
<td>53</td>
<td>122</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>689</td>
<td>138</td>
<td>76</td>
<td>277</td>
<td>73</td>
</tr>
<tr>
<td>6</td>
<td>746</td>
<td>149</td>
<td>87</td>
<td>318</td>
<td>75</td>
</tr>
<tr>
<td>7</td>
<td>274</td>
<td>73</td>
<td>51</td>
<td>119</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>573</td>
<td>121</td>
<td>69</td>
<td>228</td>
<td>65</td>
</tr>
<tr>
<td>9</td>
<td>444</td>
<td>102</td>
<td>108</td>
<td>171</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>530</td>
<td>110</td>
<td>65</td>
<td>203</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>216</td>
<td>61</td>
<td>49</td>
<td>87</td>
<td>50</td>
</tr>
<tr>
<td>12</td>
<td>179</td>
<td>54</td>
<td>51</td>
<td>81</td>
<td>52</td>
</tr>
<tr>
<td>13</td>
<td>137</td>
<td>47</td>
<td>48</td>
<td>66</td>
<td>52</td>
</tr>
<tr>
<td>14</td>
<td>614</td>
<td>119</td>
<td>76</td>
<td>234</td>
<td>69</td>
</tr>
<tr>
<td>15</td>
<td>601</td>
<td>122</td>
<td>73</td>
<td>221</td>
<td>61</td>
</tr>
<tr>
<td>AVG.</td>
<td>429.2</td>
<td>96.9</td>
<td>66.7</td>
<td>174.0</td>
<td>59.3</td>
</tr>
</tbody>
</table>
platform as Greedy Randomized Adaptive Search Procedure (GRASP) or Genetic Algorithm.

References


