Dynamic Multi-Objective Optimization with jMetal and Spark: A Case Study

José A. Cordero[1], Antonio J. Nebro[2(✉)], Cristóbal Barba-González[2], Juan J. Durillo[3], José García-Nieto[2], Ismael Navas-Delgado[2], and José F. Aldana-Montes[2]

[1] European Organization for Nuclear Research (CERN), Geneva, Switzerland
[2] Khaos Research Group, Ada Byron Research Building, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, Málaga, Spain
antonio@lcc.uma.es
[3] Distributed and Parallel Systems Group, University of Innsbruck, Innsbruck, Austria

Abstract. Technologies for Big Data and Data Science are receiving increasing research interest nowadays. This paper introduces the prototyping architecture of a tool aimed to solve Big Data Optimization problems. Our tool combines the jMetal framework for multi-objective optimization with Apache Spark, a technology that is gaining momentum. In particular, we make use of the streaming facilities of Spark to feed an optimization problem with data from different sources. We demonstrate the use of our tool by solving a dynamic bi-objective instance of the Traveling Salesman Problem (TSP) based on near real-time traffic data from New York City, which is updated several times per minute. Our experiment shows that both jMetal and Spark can be integrated providing a software platform to deal with dynamic multi-optimization problems.

Keywords: Multi-objective optimization · Dynamic optimization problem · Big data technologies · Spark · Streaming processing · jMetal

1 Introduction

Big Data is defined in a generic way as dealing with data which are too large and complex to be processed with traditional database technologies [1]. The standard *de facto* plaftorm for Big Data processing is the Hadoop system [2], where its HDFS file system plays a fundamental role. However, another basic component of Hadoop, the MapReduce framework, is loosing popularity in favor of modern Big data technologies that are emerging, such as Apache Spark [3], a general-purpose cluster computing system that can run on a wide variety of distributed systems, including Hadoop.

Big Data applications can be characterized by no less than four V's: Volume, Velocity, Variety, and Veracity [4]. In this context, there are many scenarios that

© Springer International Publishing AG 2016
P.M. Pardalos et al. (Eds.): MOD 2016, LNCS 10122, pp. 106–117, 2016.
DOI: 10.1007/978-3-319-51469-7_9

can benefit from Big Data technologies although the tackled problems do not fulfill all the V's requirements. In particular, many applications do not require to process amounts of data in the order of petabytes, but they are characterized by the rest of features. In this paper, we are going to focus on one of such applications: dynamic multi-objective optimization with data received in streaming. Our purpose is to explore how Big Data and optimization technologies can be used together to provide a satisfactory solution for this kind of problems.

The motivation of our work is threefold. First, the growing availability of Open Data by a number of cities is fostering the appearance of new applications making use of them, as for example Smart city applications [5,6] related to traffic. In this context, the open data provided by the New York City Department of Transportation [7], which updates traffic data several times per minute, has led us to consider the optimization of a dynamic version of the Traveling Salesman Problem (TSP) [8] by using real data to define it.

Second, from the technological point of view, Spark is becoming a dominant technology in the Big Data context. This can be stated in the Gartner's Hype Cycle for Advanced Analytics and Data Science 2015 [9], where Spark is almost at the top of the Peak of Inflated Expectations.

Third, metaheuristics are popular algorithms for solving complex real-world optimization problems, so they are promising methods to be applied to deal with the new challenging applications that are appearing in the field known as Big Data Optimization.

With these ideas in mind, our goal here is to provide a software solution to a dynamic multi-objective optimization problem that is updated with a relative high frequency with real information. Our proposal is based on the combination of the jMetal optimization framework [10] with the Spark features to process incoming data in streaming. In concrete, the contributions of this work can be summarized as follows:

- We define a software solution to optimize dynamic problems with streaming data coming from Open Data sources.
- We validate our proposal by defining a dynamic bi-objective TSP problem instance and testing it with both, synthetic and real-world data.
- The resulting software package is freely available[1].

The rest of the paper is organized as follows. Section 2 includes a background on dynamic multi-objective optimization. The architecture of the software solution is described in Sect. 3. The case study is presented in Sect. 4. Finally, we present the conclusions and lines of future work in Sect. 5.

2 Background on Dynamic Multi-Objective Optimization

A multi-objective optimization problem (MOP) is composed of two or more conflicting objective or functions that must be minimized/maximized at the same

[1] https://github.com/jMetal/jMetalSP.

time. When the features defining the problem do not change with time, many techniques can be used to solve them. In particular, multi-objective evolutionary algorithms (EMO) have been widely applied in the last 15 years to solve these kinds of problems [11,12]. EMO techniques are attractive because they can find a widely distributed set of solutions close to the Pareto front (PF) of a MOP in a single run.

Many real world applications are not static, hence the objective functions or the decision space can vary with time [13]. This results in a Dynamic MOP (DMOP) that requires to apply some kind of dynamic EMO (DEMO) algorithm to solve it.

Four kinds of DMOPs can be characterized [13]:

- Type I: The Pareto Set (PS) changes, i.e. the set of all the optimal decision variables changes, but the PF remains the same.
- Type II: Both PS and PF change.
- Type III: PS does not change whereas PF changes.
- Type IV: Both PS and PF do not change, but the problem can change.

Independently of the DMOP variant, traditional EMO algorithms must be adapted to transform them into some kind of DEMO to solve them. Our chosen DMOP is a bi-objective formulation of the TSP where two goals, distance and traveling time, have to be minimized. We assume that the nodes remain fixed and that there are variations in the arcs due to changes in the traffic, such as bottlenecks, traffic jams or cuts in some streets that may affect the traveling time and the distance between some places. This problem would fit in the Type II category.

3 Architecture Components

In this section, the proposed architecture is described, giving details of the two main components; jMetal Framework and Spark.

3.1 Proposed Architecture

We focus our research on a context in which: (1) the data defining the DMOP are produced in a continuous, but not necessarily constant rate (i.e. they are produced in streaming), (2) they can be generated by different sources, and (3) they must be processed to clear any wrong or inconsistent piece of information. These three features cover the velocity, variety and veracity of Big Data applications. The last V, the volume, will depend on the amount of information that can be obtained to be processed.

With these ideas in mind we propose the software solution depicted in Fig. 1. The dynamic problem (multi-objective TSP or MSTP) is being continuously dealt with a multi-objective algorithm (dynamic NSGA-II), which stores in an external file system the found Pareto front approximations. In parallel, a component reads information from data sources in streaming and updates the problem.

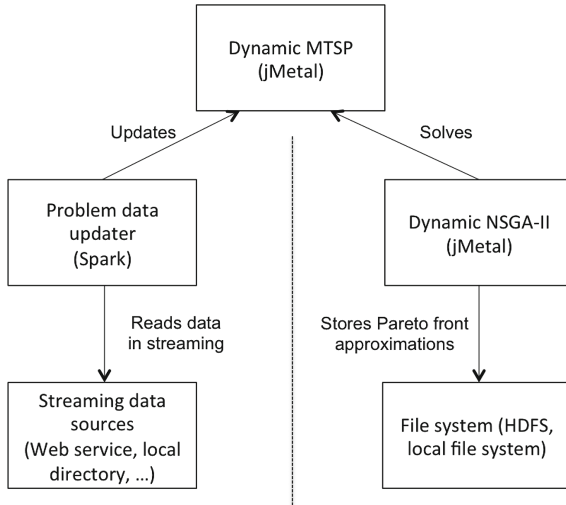

Fig. 1. Architecture of the proposed software solution.

To implement this architecture, the problem and the algorithm are jMetal objects, and the updater component is implemented in Spark. We provide details of all these elements next.

3.2 The jMetal Framework

jMetal is Java object-oriented framework aimed at multi-objective optimization with metaheuristics [10]. It includes a number of algorithms representative of the state-of-the-art, benchmark problems, quality indicators, and support for carrying out experimental studies. These features have made jMetal a popular tool in the field of multi-objective optimization. In this work, we use jMetal 5 [14], which has the architecture depicted in Fig. 2. The underlying idea in this framework is that an algorithm (metaheuristic) manipulates a number of solutions with some operators to solve an optimization problem.

Focusing on the `Problem` interface, it provides methods to know about the basic problem features: number of variables, number of objectives, and number of constraints. The main method is `evaluate()`, which contains the code implementing the objective functions that are computed to evaluate a solution.

Among all the MOPs provided by jMetal there exist a `MultiobjectiveTSP` class, having two objectives (distance and cost) and assuming that the input data are files with TSPLIB [15] format. We have adapted this class to be used in a dynamic context, which has required three changes:

1. Methods for updating part or the whole data matrices have to be incorporated. Whenever one of them is invoked, a flag indicating a data change must be set.

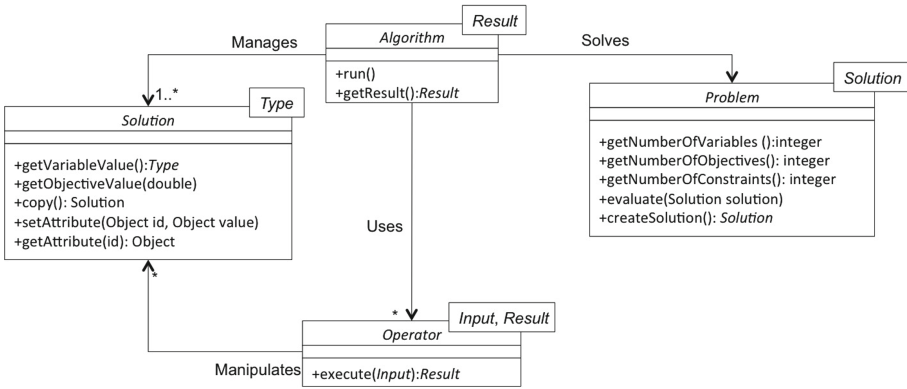

Fig. 2. jMetal 5.0 architecture.

2. The aforementioned methods and the `evaluate()` method have to be tagged as synchronized to ensure the mutual exclusion when accessing to the problem data.
3. A method is needed to get the status of the data changed flag and another one to reset it.

In order to explain how to adapt an existing EMO algorithm to solve a DMOP, we describe next the steps that are required to develop a dynamic version of the well-known NSGA-II algorithm [16]. A feature of jMetal 5 is the inclusion of algorithm templates [14] that mimic the pseudo-code of a number of multi-objective metaheuristics. As an example, the `AbstractEvolutionaryAlgorithm` template (an abstract class) contains a `run()` method that includes the steps of a generic EMO algorithm as can be observed in the following code:

```
1. @Override public void run() {
2.  List<S> offspringPopulation;
3.  List<S> matingPopulation;

4.  population = createInitialPopulation();
5.  population = evaluatePopulation(population);
6.  initProgress();
7.  while (!isStoppingConditionReached()) {
8.    matingPopulation = selection(population);
9.    offspringPopulation = reproduction(matingPopulation);
10.   offspringPopulation =
                     evaluatePopulation(offspringPopulation);
11.   population = replacement(population, offspringPopulation);
12.   updateProgress();
13. }
14.}
```

The implementation of NSGA-II follows this template, so it defines the methods for creating the initial population, evaluating the population, etc. To imple-

ment a dynamic variant of NSGA-II (DNSGA-II), the class defining it can inherit from the NSGA-II class and only two methods have to be redefined:

- `isStoppingConditionReached()`: when the number of function evaluations reaches its limit (stopping condition), a file with the Pareto front approximation found is written, but instead of terminating the algorithm, a re-start operation is carried out and the algorithm begins again.
- `updateProgress()`: after an algorithm iteration, a counter of function evaluations is updated. In the DNSGA-II code, the data changed flag of the problem is checked. If the result is positive, the population is re-started and evaluated and the flag is reset.

3.3 Apache Spark

Apache Spark is a general-purpose distributed computing system [3] based on the concept of Resilient Distributed Datasets (RDDs). RDDs are collections of elements that can be operated in parallel on the nodes of a cluster by using two types of operations: transformations (e.g. map, filter, union, etc.) and actions (e.g. reduce, collect, count, etc.). Among the features of Spark (high level parallel processing programming model, machine learning algorithms, graph processing, multi-programming language API), we use here its support for streaming processing data. In this context, Spark manages the so called `JavaDStream` structures, which are later discretized into a number of RDDs to be processed.

There are a number of streaming data sources that Spark can handle. In our proposal, we choose as a source a directory where the new incoming data will be stored. We assume that a daemon process is iteratively fetching the data from a Web service and writing it to that source directory. The data will have the form of text files where each line has the following structure: a symbol 'd' (distance) or 't' (travel time), two integers representing the coordinates of the point, and the new distance/time value.

The pseudo-code of the Spark+jMetal solution is described next:

```
 1. DynamicMultiobjectiveTSP problem <- intializeProblem();
 2. File outputDirectory <- createOutputDirectory();
 3. DynamicNSGAII algorithm = initilizeAlgorithm(problem, outputDirectory);
 4. startAlgorith(algorithm);
 5. SparkConf sparkConf = new SparkConf().setAppName("SparkClient");
 6. JavaStreamingContext streamingContext =
        new JavaStreamingContext(sparkConf, Durations.seconds(5));
 7. JavaDStream<String> lines =
        streamingContext.textFileStream(inputDataDirectory);
 8. JavaDStream<Map<>> routeUpdates = lines.map(s -> {return parsed lines});
 9. routeUpdates.foreachRDD(
        s -> {list = s.collect();
        list.foreach(items -> {updateProblem(problem, items.next()})});

10. streamingContext.start();
11. streamingContext.awaitTermination();
```

The first steps initialize the problem with the matrices containing data of distance and travel time, and to create the output directory where the Pareto front approximations will be stored (lines 1 and 2). Then the algorithm is created and its execution is started in a thread (lines 3 and 4). Once the algorithm is running, it is the turn to start Spark, what requires two steps: creating a `SparkConf` object (line 5) and a `JavaStreamingContext` (line 6), which indicates the polling frequency (5 s in the code).

The processing of the incoming streaming files requires three instructions: first, a text file stream is created (line 7), which stores in a `JavaDStream<String>` list all the lines of the files arrived to the input data directory since the last polling. Second, a map transformation is used (line 8) to parse all the lines read in the previous step. The last step (line 9) consists in executing a `foreachRDD` instruction to update the problem data with the information of the parsed lines.

The two last instructions (lines 10 and 11) start the Spark streaming context and await for termination. As a result, the code between lines 7 and 9 will be iteratively executed whenever new data files have been written in the input data directory since the last polling.

We would like to remark that this pseudo-code mimics closely the current Java implementation, so no more than a few lines of code are needed.

4 Case Study: Dynamic Bi-objective TSP

To test our software architecture in practice we apply it to two scenarios: an artificial dynamic TSP (DTSP) with benchmark data and another version of the problem with real data. We analyze both scenarios next.

4.1 Problem with Synthetic Data

Our artificial DTSP problem is built from the data to two 100 TSP instances taken from TSPLIB [15]. One instance represents the distances and the other one the travel time. To simulate the modification of the problem, we write every 5 s a data file containing updated information.

The parameter settings of the dynamic NSGA-II algorithm are the following: the population size is 100, the crossover operator is PMX (applied with a 0.9 probability), the mutation operator is swap (applied with a probability of 0.2), and the algorithm computes 250,000 function evaluations before writing out the found front and re-starting. As development and target computer we have used a MacBook Pro laptop (2,2 GHz Intel Core i7 processor, 8 GB RAM, 256 GB SSD) with MacOS 10.11.3, Java SE 1.8.0_40, and Spark 1.4.1.

Figure 3 depicts some fronts produced by the dynamic NSGA-II throughout a given execution, starting from the first one (`FUN0.tsv`) up to the 20th one (`FUN20.tsv`). We can observe that the shape of the Pareto front approximations change in time due to the updating operation of the problem data. In fact, each new front contains optimized solutions with regards to the previous

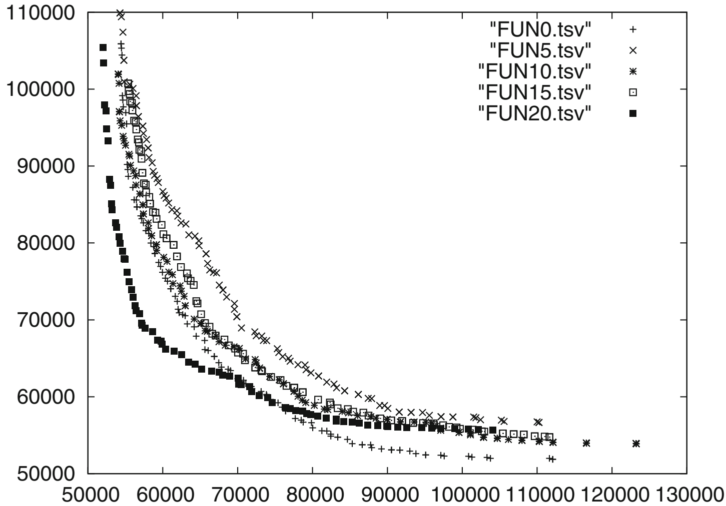

Fig. 3. Evolution of the Pareto front approximations yielded by the dynamic NSGA-II algorithm (FUNx.tsv refers to the xth front that have been computed).

ones, which lead us to suggest that the learning model of the optimization procedure is kept through different re-starts, although detecting the changes in the problem structure.

Once we have tested our approach on a synthetic instance, we now aim at managing real-world data, in order to show whether the proposed model is actually applicable or not.

4.2 Problem with Real Data

The DTSP with real-world data we are going to define is based on the Open Data provided by the New York City Department of Transportation, which updates the traffic information several times per minute[2]. The information is provided as a text file where each line includes, among other data:

- `Id`: link identifier (an integer number)
- `Speed`: average speed a vehicle traveled between end points on the link in the most recent interval (a real number).
- `TravelTime`: average time a vehicle took to traverse the link (a real number).
- `Status`: if the link is closed by accidents, works or any cause (a boolean).
- `EncodedPolyLine`: an encoded string that represents the GPS Coordinates of the link. It is encoded using the Google's Encoded Polyline Algorithm [17].
- `DataAsOf`: last time data was received from link (a date).
- `LinkName`: description of the link location and end points (a string).

[2] At the time of writing this paper, the data can be obtained from this URL: http://207.251.86.229/nyc-links-cams/LinkSpeedQuery.txt.

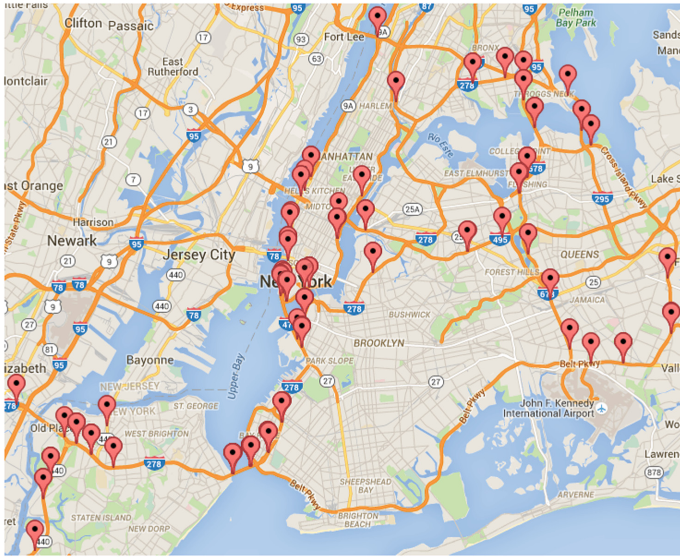

Fig. 4. Real dynamic TSP instance: 93 nodes of the city of New York.

As the information is given in the form of links instead of nodes, we have made a pre-processing of the data to obtain a feasible TSP. To join the routes, we have compared the GPS coordinates. A communication between two nodes is created when there is a final point of a route and a starting point of another in the same place (with a small margin error). After that, we iterate through all the nodes removing those with degree <2. The list of GPS coordinates is decoded from the `EncodedPolyLine` field of each link.

As as result, the DTSP problem is composed of 93 locations and 315 communications between them, which are depicted in Fig. 4. We have to note that the links are bi-directional, so the resulting DTSP is asymmetric. In this regard, it is worth mentioning that we have approached the TSP by following a vector permutation optimization model, as commonly done in population based metaheuristics.

To determine the distance between points, we have used a Google service [18] that, given two locations, it returns the distance between them; this step only is carried out to initialize the graph. The initial travel time and speed are obtained from the first data file read.

As in the synthetic DTSP, a daemon process is polling the data source every 30 s, parsing the received information, and writing the updates directly into the cost and distance matrices that Spark is using to calculate the results. If a route has a status 1, meaning that it has been closed to circulation, we assume an infinite cost and distance for that route.

After running our software for 40 min, the number of data requests to the traffic service was 77, from which 37 returned no information. In those cases

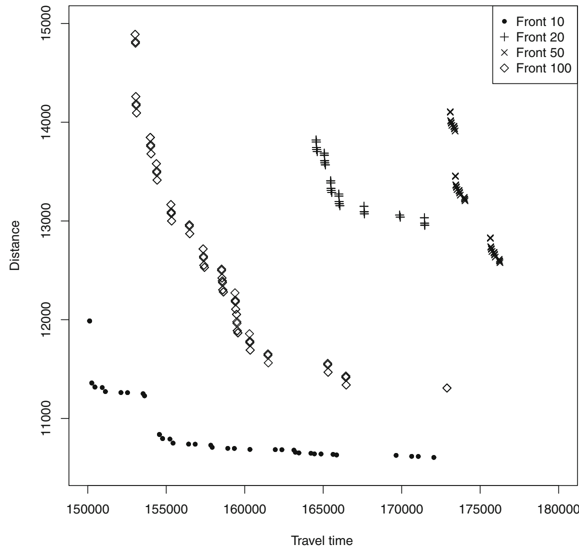

Fig. 5. Pareto front approximations obtained when solving the real data DTSP (Front x refers to the xth front that have been obtained).

where there was new information, the number of updates ranged between 141 and 238.

Figure 5 shows the Pareto front approximations obtained after 10, 20, 50, and 100 runs of the dynamic NSGA-II algorithm. Similarly to the obtained results in the synthetic instance, in this case the fronts of solutions are refined through the optimization process. Therefore, the learning model also is kept when dealing with real-world data.

4.3 Discussion

An analysis at a glance of our architecture could indicate that the use of Spark does not justify selecting it to be included in our system, because we are not taking advantage of many of its features. However, we would like to note what we are presenting here is a first approximation to the problem and many extensions are possible. For example, thinking in Smart City applications and our considered optimization problem, more data sources could be used for a more precise problem formulation, such as weather forecasting, social networks, GPS data from mobile phones in cars, etc. This would require more demanding storage (e.g. in HDFS) and more computing power for processing and integrating all the data (e.g. a Hadoop cluster), and in this scenario not only the streaming feature of Spark will be useful, but its high performance computing capabilities.

From the optimization point of view, we have presented a simple adaptation of a well-known algorithm, NSGA-II, to solve the dynamic TSP problem. In case of defining a more realistic problem, with more data sources, a more sophisticated

algorithm could be designed to take advantage of problem-specific data, e.g. by incorporating a local search or new variation operators. Furthermore, we must consider that in a real scenario, if the traveler starts to visit the nodes while the application is running, the traversed arcs should be removed from the problem, so it would be simplified and then it would not be a TSP anymore.

It is clear that a complete solution of the considered optimization problem would require additional components, such as a visualization module able of displaying the evolution of the Pareto front approximations and enabling to choose a particular trade-off solution from them. We consider that this kind of components are orthogonal to our software architecture. As the produced fronts are being stored in secondary storage, an external program can be developed to display them as needed.

5 Conclusions and Future Work

In this paper, we have presented a software solution to deal with a Big Data Optimization problem by combining the jMetal optimization framework with the Spark cluster computing system and we have demonstrated how to apply it to solve a concrete example: a dynamic multi-objective TSP.

Our motivation has been driven by the availability of Open Data sources, the raise of Spark as distributed computing platform in clusters and Hadoop systems, and the utilization of the jMetal framework to provide the infrastructure to deal with multi-objective optimization problems.

We have presented two case studies that consider a bi-objective formulation of the problem, where the total distance and time travel are goals to be minimized. First, a synthetic version of the problem, based on benchmark data, to test the working of the system; second, a real instance created from Open Data of the city of New York. In this latter case, the TSP nodes correspond to real locations, and the problem data is updated in streaming.

Defining more realistic problems, including additional data sources, as well as considering other Smart city related problems are matters of future works.

Acknowledgments. This work is partially funded by Grants TIN2011-25840 (Ministerio de Ciencia e Innovación) and P11-TIC-7529 and P12-TIC-1519 (Plan Andaluz de Investigación, Desarrollo e Innovación). Cristóbal Barba-González is supported by Grant BES-2015-072209 (Ministerio de Economía y Competitividad).

References

1. Editorial: Community cleverness required. Nature **455**, 1 (2008)
2. White, T.: Hadoop: The Definitive Guide, 1st edn. O'Reilly Media Inc., Sebastopol (2009)
3. Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, Berkeley, CA, USA, HotCloud 2010, pp. 10. USENIX Association (2010)

4. Marr, M.: Big Data: Using SMART Big Data Analytics and Metrics to Make Better Decisions and Improve Performance. Wiley, Hoboken (2015)
5. Nam, T., Pardo, T.: Smart city as urban innovation: focusing on management, policy, and context. In: Proceedings of the 5th International Conference on Theory and Practice of Electronic Governance, ICEGOV 2011, pp. 185–194. ACM (2011)
6. Garcia-Nieto, J., Olivera, A., Alba, E.: Optimal cycle program of traffic lights with particle swarm optimization. IEEE Trans. Evol. Comput. **17**, 823–839 (2013)
7. NYCDOT: New York City traffic speed detectors data set (2016). http://nyctmc.org
8. Papadimitriou, C.H.: The Euclidean travelling salesman problem is NP-complete. Theor. Comput. Sci. **4**, 237–244 (1977)
9. Gartner Inc.: Gartner's hype cycle for advanced analytics and data science (2015). https://www.gartner.com/doc/3087721/hype-cycle-advanced-analytics-data
10. Durillo, J., Nebro, A.: jMetal: a java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011)
11. Deb, K.: Multi-objective Optimization Using Evolutionary Algorithms. Wiley, New York (2001)
12. Coello, C., Lamont, G., van Veldhuizen, D.: Multi-objective Optimization Using Evolutionary Algorithms, 2nd edn. Wiley, New York (2007)
13. Farina, M., Deb, K., Amato, P.: Dynamic multiobjective optimization problems: test cases, approximations, and applications. IEEE Trans. Evol. Comput. **8**, 425–442 (2004)
14. Nebro, A., Durillo, J.J., Vergne, M.: Redesigning the jMetal multi-objective optimization framework. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion 2015, pp. 1093–1100. ACM, New York (2015)
15. Reinelt, G.: TSPLIB - a traveling salesman problem library. INFORMS J. Comput. **3**, 376–384 (1991)
16. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**, 182–197 (2002)
17. Google Inc.: Encoded polyline algorithm format (2016). https://developers.google.com/maps/documentation/utilities/polylinealgorithm
18. Google Inc.: Google maps distance matrix API (2016). https://developers.google.com/maps/documentation/distance-matrix