

IDC 2018

15 – 17 October 2018 – Bilbao, Spain



UNIVERSIDAD
DE MÁLAGA

Multi-Objective Big Data Optimization with jMetalSP

Antonio J. Nebro
PhD in Computer Science
antonio@lcc.uma.es

Cristóbal Barba-González
PhD Student
cbarba@lcc.uma.es



About this tutorial

- jMetalSP is
 - A Java-based framework for multi-objective Big Data optimization with metaheuristics
 - It combines
 - jMetal: optimization engine (<https://github.com/jMetal/jMetal>)
 - Apache Spark: unified analytics engine for large-scale data processing (<http://spark.apache.org/>)
 - Open source project (MIT license)
- Web site
 - GitHub: <https://github.com/jMetal/jMetalSP>



About this tutorial

- Purpose of this tutorial
 - Offer a practical view of the jMetalSP framework
 - Description of its main components (algorithms, problems, streaming runtime, streaming data processing, algorithm data consumers)
 - Examples of dynamic versions of well-known algorithms (e.g., NSGA-II) and newer proposals (e.g., InDM2).
 - How to use jMetalSP to solve a dynamic optimization problem
 - Show how you can use jMetal to:
 - Configure and run dynamic multi-objective metaheuristics with and without Spark

Table of contents

- Who, why, what, when
- Background
- Working with jMetalSP
- Case study
- Further developments

Who, why, what, when

- Antonio J. Nebro
 - PhD in Computer Science (1999)
 - Associate professor (University of Málaga – Spain)
 - Khaos research group (<http://khaos.uma.es/en>)
- Research interests
 - Multi-objective optimization
 - Parallel metaheuristics
 - Applications to real-world problems (bioinformatics, civil engineering, Big Data)
 - Software tools (jMetal, jMetalSP, jMetalPy)



Who, why, what, when

- Cristóbal Barba González
 - PhD student
 - Researcher (University of Málaga – Spain)
 - Khaos research group (<http://khaos.uma.es/>)
- Research interests
 - Multi-objective Big Data optimization
 - Parallel metaheuristics
 - Applications to real-world problems (bioinformatics, civil engineering, Big Data)
 - Software tools (jMetal, jMetalSP)
 - Web Semantic



Who, why, what, when

- Khaos research group

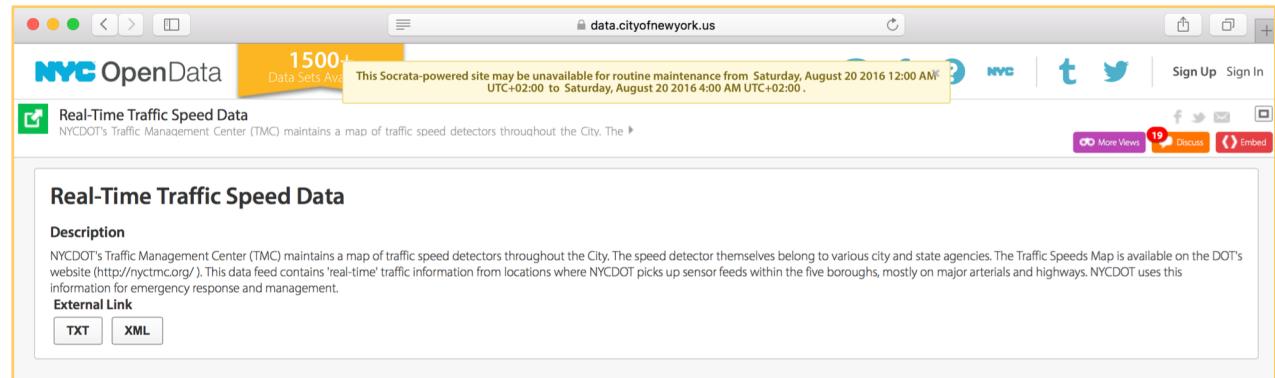


Who, why, what, when

- **Research Lines:**
 - Data Management and Integration (Relational Data Bases, NoSQL, Linked Data, Open Data)
 - Data Analysis (Data Mining, Test Mining, Information Recovery, Optimization Algorithms, Big Data Analytics)
 - Semantic Annotation. Integrated Data Analysis
 - Semantics (Scalable Reasoning, Semantic Relations Discovery, Semantic Assignment to Mobile Objects, Semantic Driven Contents Recommendation, Web Semantics for E-Sciences, Smart Data for Big Data interpretation)
 - **Metaheuristics. Multi-objective Optimization. Big Data Optimization**
- **Applications (multi-disciplinary domains):**
 - Life Sciences and Biomedicine
 - Cultural Patrimony and Tourism
 - E-commerce
 - Smart Cities

Who, why, what, when

- We have a lot of experience with jMetal
 - Java-based software for multi-objective optimization with metaheuristics
 - Project started in 2006
 - <https://github.com/jMetal/jMetal>
- We started to work with Apache Spark in 2015
 - Spark has a streaming engine
- Availability of open data -> real-time traffic speed data of New York



Who, why, what, when

- “It is of great interest to investigate the role of evolutionary multiobjective optimization (EMO) techniques for the optimization and learning involving big data, in particular, the ability of EMO techniques to solve dynamic multi-objective big data analytics problems.”

*Call for papers of Applied Soft Computing, Special Issue on
Evolutionary Multi-objective Optimization and Applications
in Big Data (2016)*

jMetalSP: a framework for dynamic multi-objective big data optimization

Autores Cristóbal Barba-González, José García-Nieto, Antonio J Nebro, José A Cordero, Juan J Durillo, Ismael Navas-Delgado, José F Aldana-Montes

Fecha de publicación 2018/8/1

Revista Applied Soft Computing

Who, why, what, when

- What we where interested in:
 - Defining a dynamic bi-objective transportation problem using real-time traffic data of New York processed in streaming
 - Solving the problem with a dynamic evolutionary algorithm
- Our approach:
 - To design a software platform to deal with these kinds of problems:
jMetalSP
 - The algorithms and problems are provided by jMetal
 - The streaming facilities are provided by Spark

Dynamic multi-objective optimization with jmetal and spark: a case study

Autores José A Cordero, Antonio J Nebro, Cristóbal Barba-González, Juan J Durillo, José García-Nieto, Ismael Navas-Delgado, José F Aldana-Montes

Fecha de
publicación 2016/8/26

Conferencia International Workshop on Machine Learning, Optimization and Big Data

Who, why, what, when

- We started the project in 2016
- A redesign of the architecture was carried out in 2017

Design and architecture of the jMetalSP framework

Autores Antonio J Nebro, Cristóbal Barba-González, José García Nieto, José A Cordero, José F Aldana Montes

Fecha de 2017/7/15

Conferencia Proceedings of the Genetic and Evolutionary Computation Conference Companion

- The jMetalSP project is continuously evolving and we are working actively on it

Table of contents

- Who, why, what, when
- **Background**
- Working with jMetalSP
- Case study
- Further developments

What is multi-objective optimization?

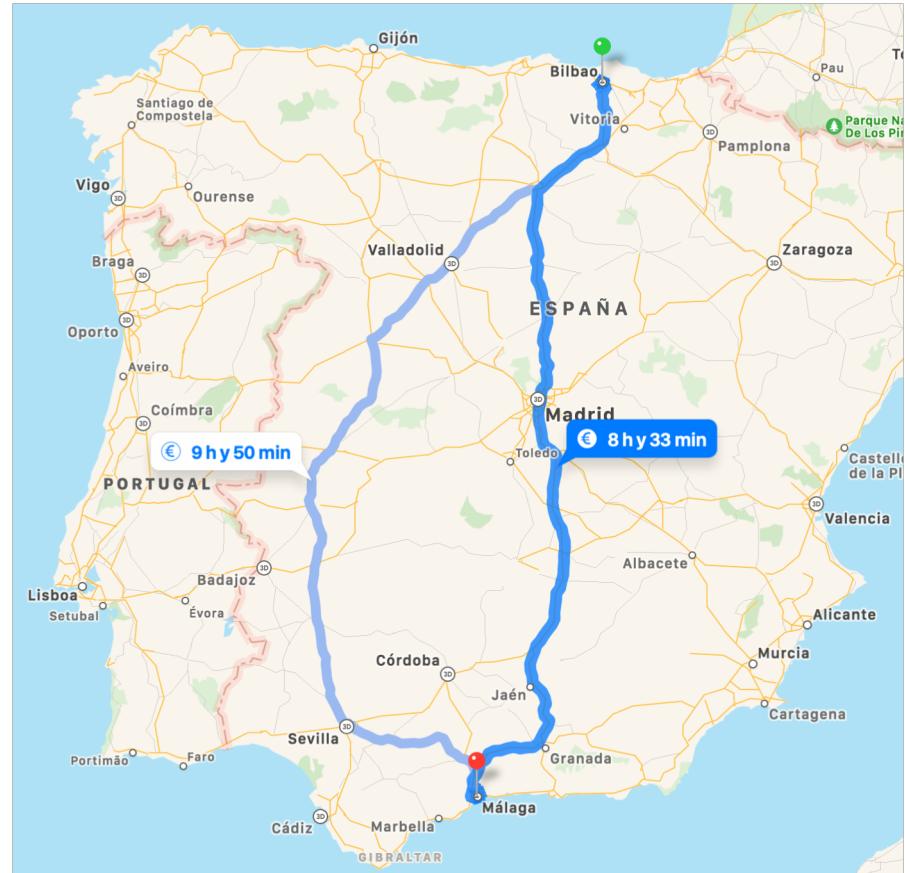
- Many real-world optimization problems require to optimize **more than one** objective or function at the same time
 - These objectives are usually in conflict among them
 - Improving one means worsening the others



- Multi-objective (or multi-criteria) optimization
 - Discipline focused in solving multiobjective optimization problems (MOPs)

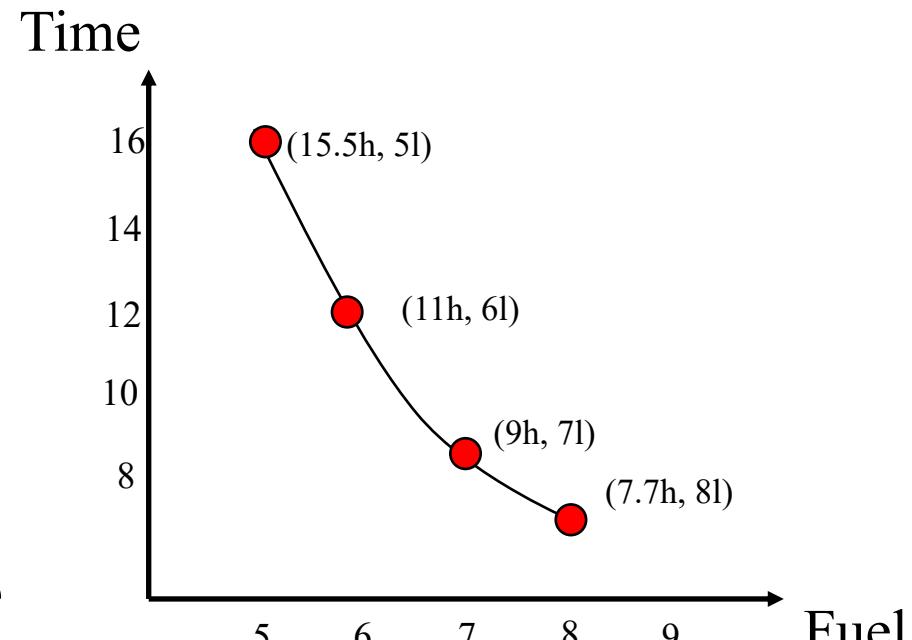
An example

- Example: travelling by car from Málaga to Bilbao (932 km)
 - Objective 1:
 - Minimizing time
 - Objective 2:
 - Minimizing fuel
 - Constraints:
 - Max. speed: 120 km/h
 - Min. speed: 60 km/h
 - Decision variable:
 - Mean car speed



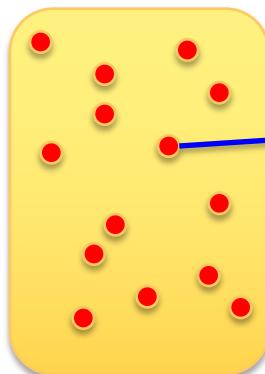
An example

- Example: travelling by car from Málaga to Bilbao (932 km)
 - We assume ideal conditions (i.e., we can keep a constant speed)
 - Extreme solutions
 - Time: 7.7 hours, fuel: 8 litres
 - Time: 15.5 hours, fuel: 5 litres
 - Other solutions
 - Time: 9 hours, fuel: 7 litres
 - Time: 12 hours, fuel: 6 litres
 - As velocity is a continuous variable
 - There are infinite compromise solutions

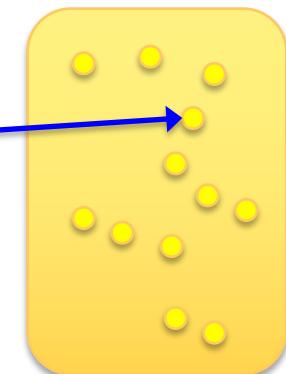


Single vs Multi-Objective Optimization

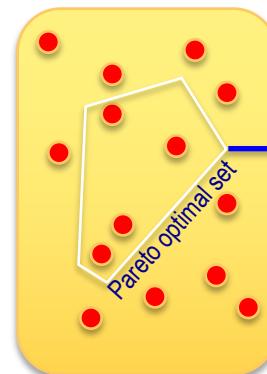
- In single-objective optimization (SO)
 - The optimum is a solution
- In multi-objective optimization (MO)
 - The optimum (**Pareto optimal set**) is a set of (**non-dominated**) solutions



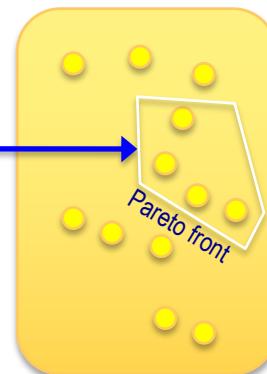
X
(Solution space)



F(X)
(Objective space)



X
(Solution space)



F(X), G(X), ...
Objective space

Concepts: Dominance and non-dominance

- In single-objective optimization
 - We look for a single solution
 - The concept of “A better than B” is trivial
- In multi-objective optimization
 - We are not restricted to find a unique optimal solution
 - the concept of “A better than B” is not trivial

A	2	3	4	5
B	4	6	5	7

A	3	7	4	8
B	2	1	2	5

A	1	9	4	5
B	3	6	5	7

A is better than B

A	3	7	4	8
B	2	1	2	5

A	1	9	4	5
B	3	6	5	7

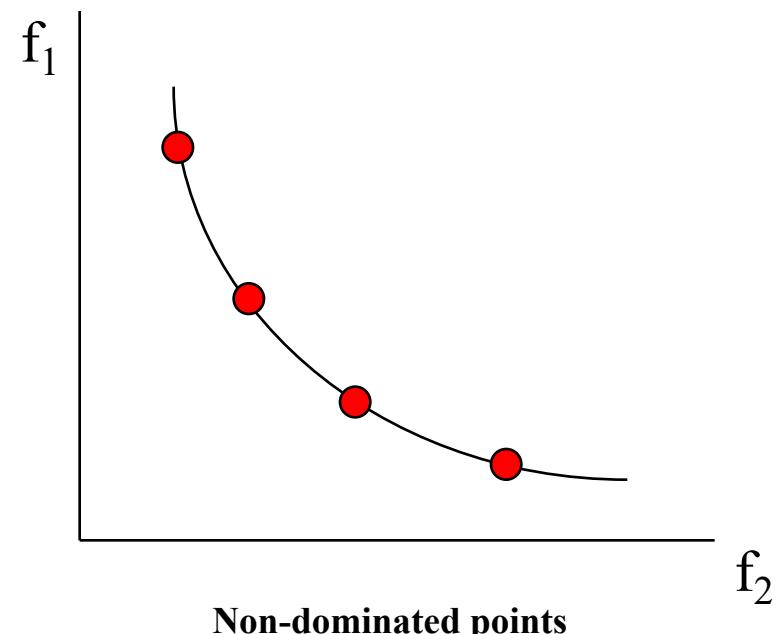
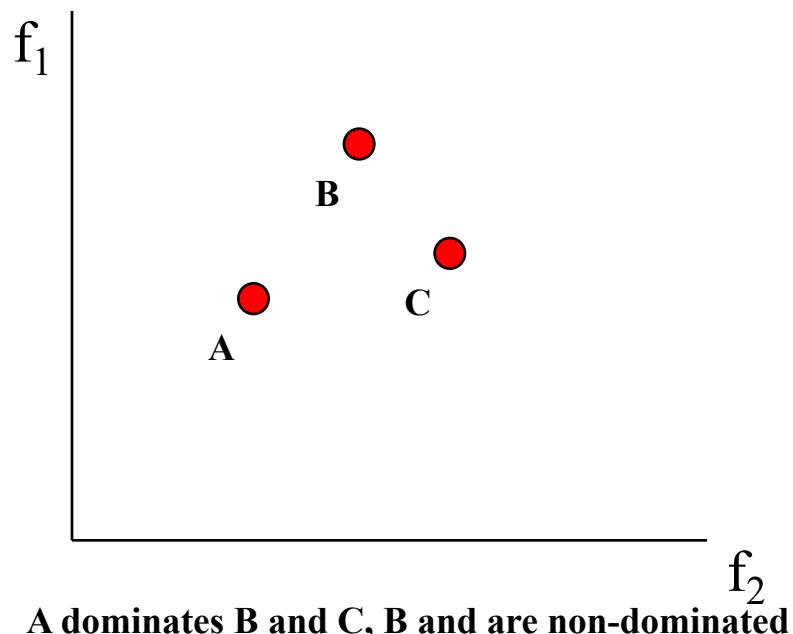
B is better than A

None is better

A and B are NON-DOMINATED

Concepts: Dominance and non-dominance

- Examples



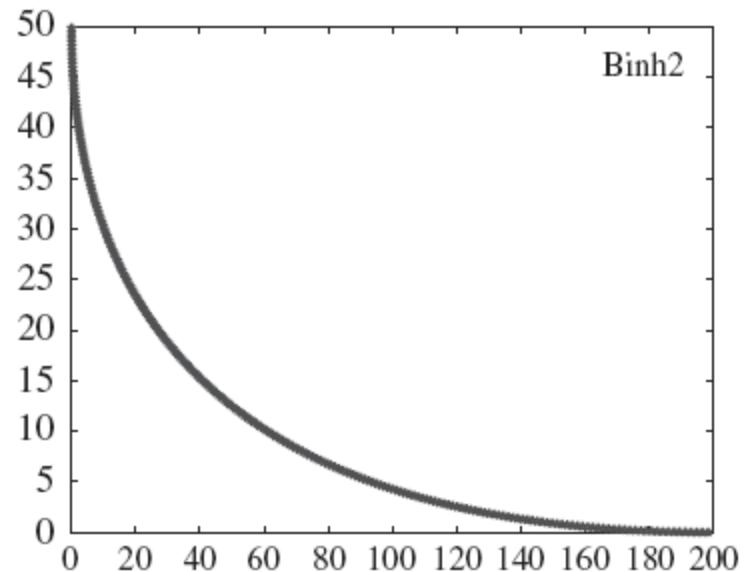
Concepts: Pareto optimal set, Pareto optimal front

- The solution of a MOP is a set composed of those non-dominated solutions that cannot be dominated by any other solution outside that set: **Pareto Optimal Set**
- The correspondence of the Pareto optimal set in the objective space is known as **Pareto Optimal Front** (or just **Pareto Front**)

$$\begin{aligned}\text{Min } F &= (f_1(\vec{x}), f_2(\vec{x})) \\ f_1(\vec{x}) &= 4x_1^2 + 4x_2 \\ f_2(\vec{x}) &= (x_1 - 5)^2 + (x_2 - 5)^2\end{aligned}$$

Subject to:

$$\begin{aligned}g_1(\vec{x}) &= (x_1 - 5)^2 + x_2^2 - 25 \leq 0 \\ g_2(\vec{x}) &= -(x_1 - 8)^2 - (x_2 + 3)^2 + 7.7 \leq 0 \\ 0 &\leq x_1 \leq 5 \\ 0 &\geq x_2 \leq 3\end{aligned}$$

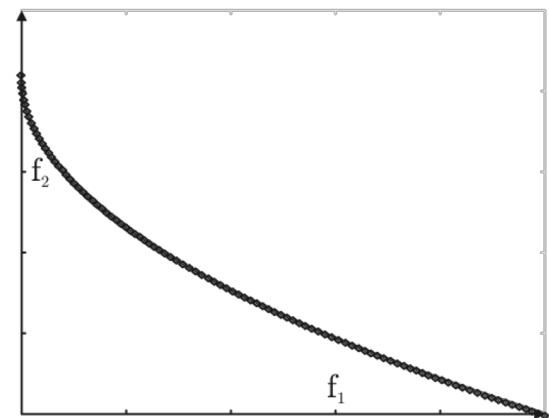
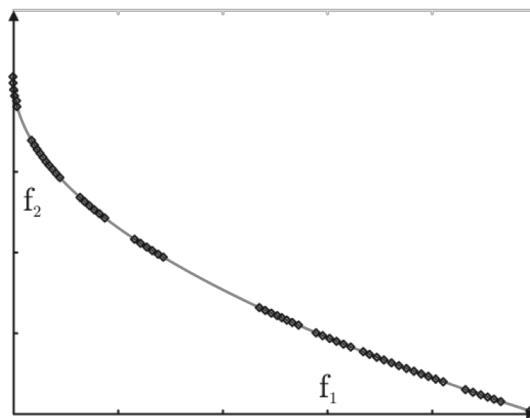
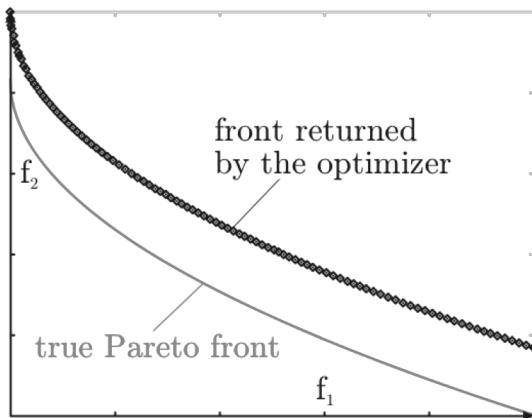


Goals of Multi-Objective Optimization

- The ideal goal is to obtain the Pareto front
- Unfortunately, this is unpractical in real-world problems
 - NP-hard complexity, non-linearity, epistasis, ...
 - Frequently, exact techniques are not useful
- Alternative: Use non-exact algorithms
 - E.g. *Metaheuristics*
 - These techniques provide an *approximation* to the Pareto front

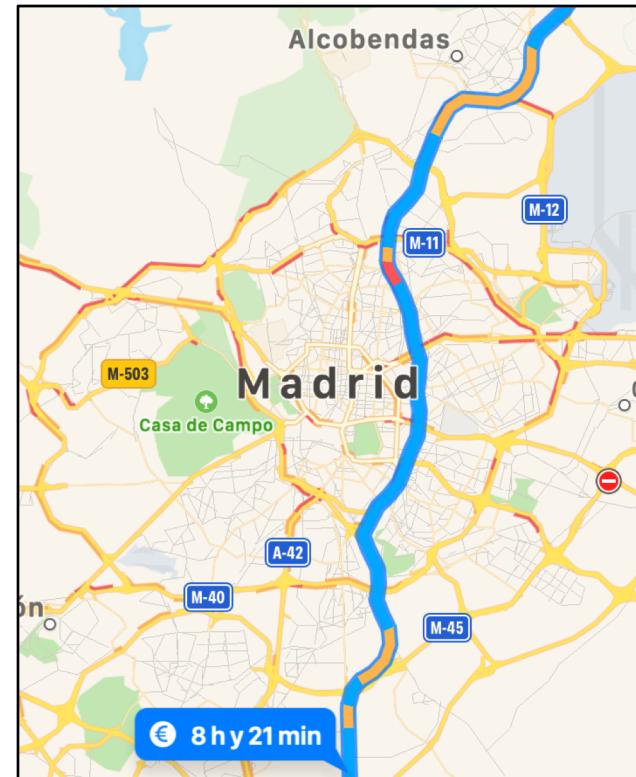
Goals of Multi-Objective Optimization

- Main goal when using non-exact techniques:
 - Obtaining an approximation to the Pareto front with two properties
 - Convergence
 - Diversity



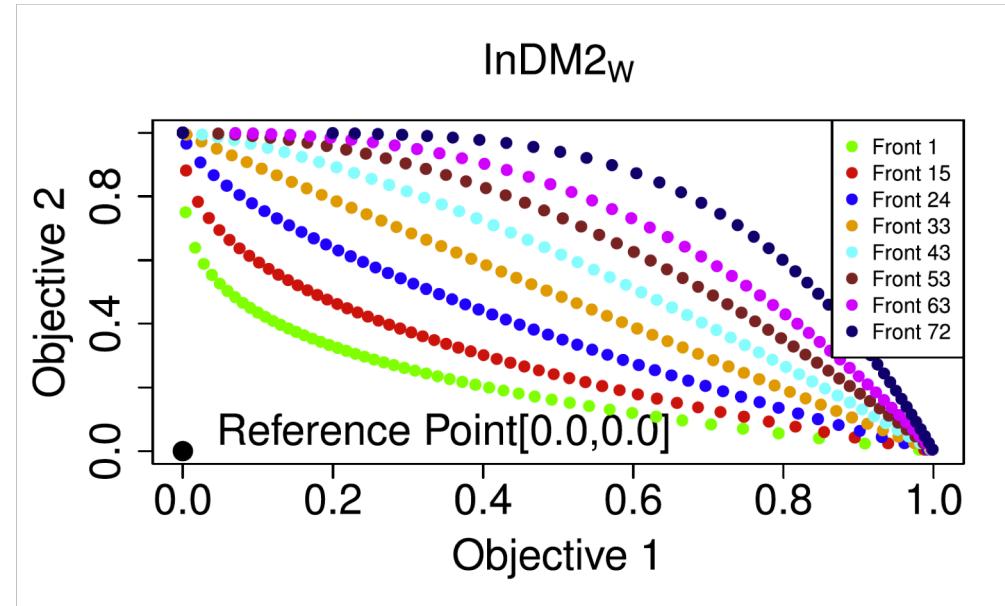
Dynamic Multi-Objective Optimization Problems

- Example: travelling by car from Málaga to Bilbao (932 km)
 - What happen if we take into consideration real-traffic information?
 - Sources:
 - Cameras
 - Sensor data
 - Web services
 - The problem data
 - Change with time
 - The optimization problem
 - Becomes **dynamic**



Dynamic Multi-Objective Optimization Problems

- The changes can affect:
 - The Pareto set
 - The Pareto front
 - Both
 - None of them, but the problem formulation can change



Metaheuristics

- Components (Greek terms):
 - Meta: “higher level”
 - Heuristic: “to find”, generic method to solve a problem
- Metaheuristic:
 - *High-level strategy that combines a set of underlying simpler operation techniques (usually heuristics) aimed at obtaining a more powerful procedure*
- They have become very popular optimization techniques
 - Evolutionary algorithms
 - Particle swarm optimization
 - Ant colony optimization
 - ...

Metaheuristics

- Pseudo-codes

Algorithm 1 Template of a metaheuristic

```
1:  $A(0) \leftarrow \text{GenerateInitialSolutions}()$ 
2:  $t \leftarrow 0$ 
3:  $\text{Evaluate}(A(0))$ 
4: while not StoppingCriterion( ) do
5:    $S(t) \leftarrow \text{Generation}(A(t))$ 
6:    $\text{Evaluate}(S(t))$ 
7:    $A(t+1) \leftarrow \text{Update}(A(t), S(t))$ 
8:    $t \leftarrow t + 1$ 
9: end while
```

Algorithm 2 Template of an Evolutionary Algorithm (EA) algorithm.

```
1:  $P(0) \leftarrow \text{GenerateInitialPopulation}()$ 
2:  $t \leftarrow 0$ 
3:  $\text{Evaluate}(P(0))$ 
4: while not StoppingCriterion( ) do
5:    $P'(t) \leftarrow \text{Selection}(P(t))$ 
6:    $P''(t) \leftarrow \text{Reproduction}(P'(t))$ 
7:    $\text{Evaluate}(P''(t))$ 
8:    $P(t+1) \leftarrow \text{Replacement}(P(t), P''(t))$ 
9:    $t \leftarrow t + 1$ 
10: end while
```

Dynamic Metaheuristics

- Adopted approach
 - If a change in the problem is detected the algorithm must do a restart
 - Restart strategies
 - Some solutions can be removed
 - The deleted solutions must be replaced by new ones

Algorithm 1 Template of a metaheuristic

```
1:  $A(0) \leftarrow \text{GenerateInitialSolutions}()$ 
2:  $t \leftarrow 0$ 
3:  $\text{Evaluate}(A(0))$ 
4: while not  $\text{StoppingCriterion}()$  do
5:    $S(t) \leftarrow \text{Generation}(A(t))$ 
6:    $\text{Evaluate}(S(t))$ 
7:    $A(t+1) \leftarrow \text{Update}(A(t), S(t))$ 
8:    $t \leftarrow t + 1$ 
9: end while
```

Change detection



Apache Spark

- Spark is a popular technology in Big Data applications



APACHE
Spark™ *Lightning-fast unified analytics engine*

Download Libraries Documentation Examples Community Developers Apache Software Foundation

Apache Spark™ is a unified analytics engine for large-scale data processing.

Speed

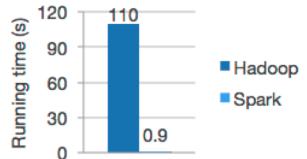
Run workloads 100x faster.

Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

Ease of Use

Write applications quickly in Java, Scala, Python, R, and SQL.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python, R, and SQL shells.



The chart shows the running time in seconds for logistic regression. Hadoop takes approximately 110 seconds, while Spark takes only 0.9 seconds.

System	Running time (s)
Hadoop	110
Spark	0.9

Logistic regression in Hadoop and Spark

```
df = spark.read.json("logs.json")
df.where("age > 21")
.select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

Latest News

- Spark 2.3.2 released (Sep 24, 2018)
- Spark+AI Summit (October 2-4th, 2018, London) agenda posted (Jul 24, 2018)
- Spark 2.2.2 released (Jul 02, 2018)
- Spark 2.1.3 released (Jun 29, 2018)

[Archive](#)


APACHECON
North America
September 24-27, 2018
Montréal, Canada

[Download Spark](#)

Built-in Libraries:

- SQL and DataFrames
- Spark Streaming
- Mlib (machine learning)
- GraphX (graph)

[Third-Party Projects](#)

<http://spark.apache.org/>

Apache Spark

- Spark includes two streaming engines:
 - RDD-based (DStreams)
 - Structured streaming (dataframe-based API)
- DStreams (RDD-based API)
 - Streaming data is processed as micro-batches



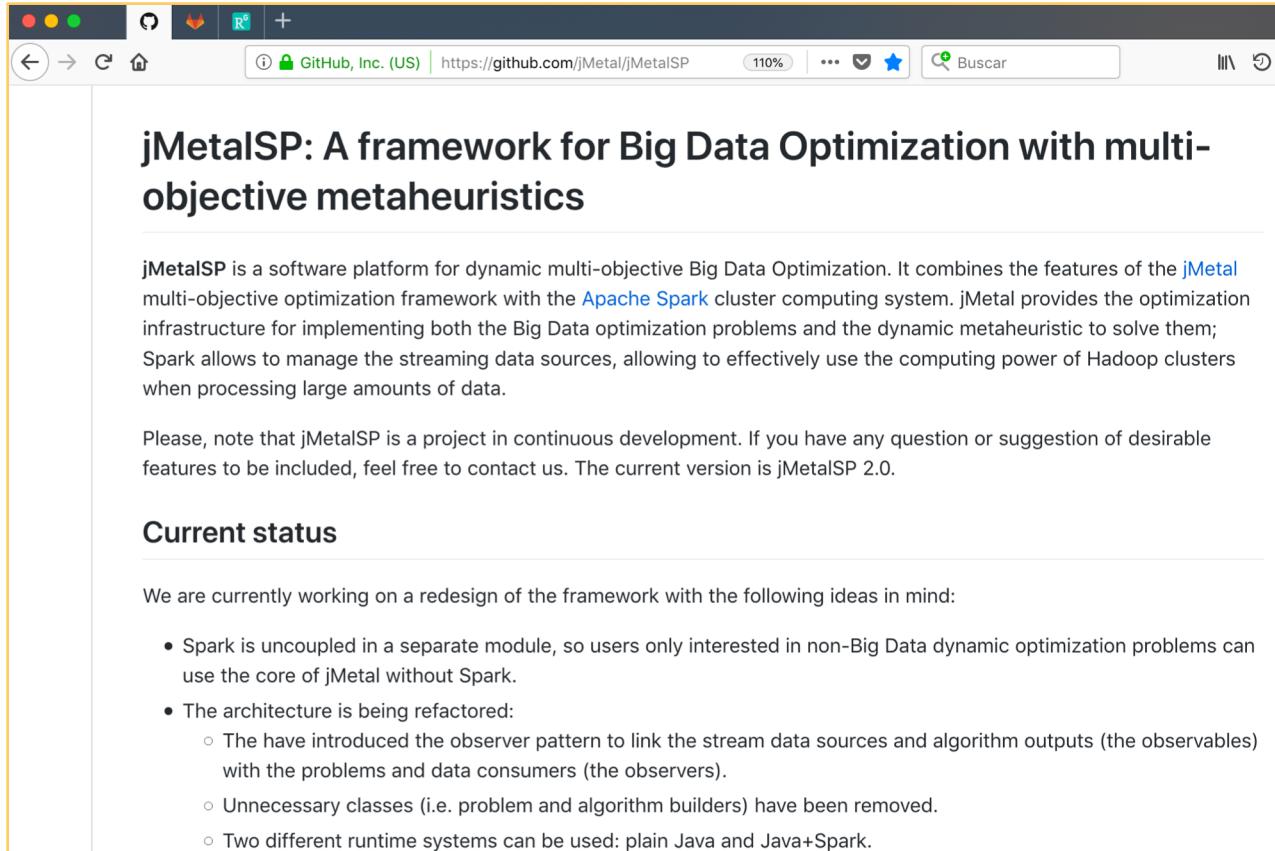
Table of contents

- Who, why, what, when
- Background
- **Working with jMetalSP**
- Case study
- Further developments

Software requirements

- Requirements to run jMetal
 - Java JDK 8+
 - Maven
 - An IDE (IntelliJ Idea, Eclipse, Netbeans)

jMetalSP: a Maven Project Hosted in GitHub



jMetalSP: A framework for Big Data Optimization with multi-objective metaheuristics

jMetalSP is a software platform for dynamic multi-objective Big Data Optimization. It combines the features of the [jMetal](#) multi-objective optimization framework with the [Apache Spark](#) cluster computing system. jMetal provides the optimization infrastructure for implementing both the Big Data optimization problems and the dynamic metaheuristic to solve them; Spark allows to manage the streaming data sources, allowing to effectively use the computing power of Hadoop clusters when processing large amounts of data.

Please, note that jMetalSP is a project in continuous development. If you have any question or suggestion of desirable features to be included, feel free to contact us. The current version is jMetalSP 2.0.

Current status

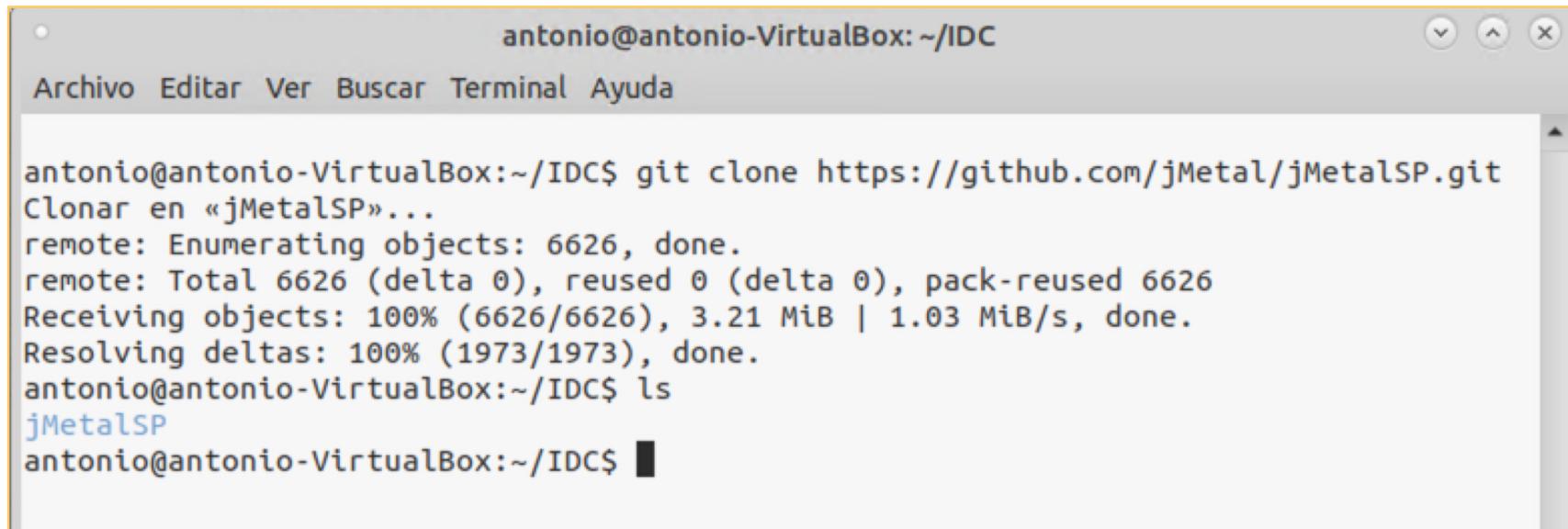
We are currently working on a redesign of the framework with the following ideas in mind:

- Spark is uncoupled in a separate module, so users only interested in non-Big Data dynamic optimization problems can use the core of jMetal without Spark.
- The architecture is being refactored:
 - We have introduced the observer pattern to link the stream data sources and algorithm outputs (the observables) with the problems and data consumers (the observers).
 - Unnecessary classes (i.e. problem and algorithm builders) have been removed.
 - Two different runtime systems can be used: plain Java and Java+Spark.

<https://github.com/jMetal/jMetalSP>

jMetalSP: a Maven Project Hosted in GitHub

- Getting the full source code of the project

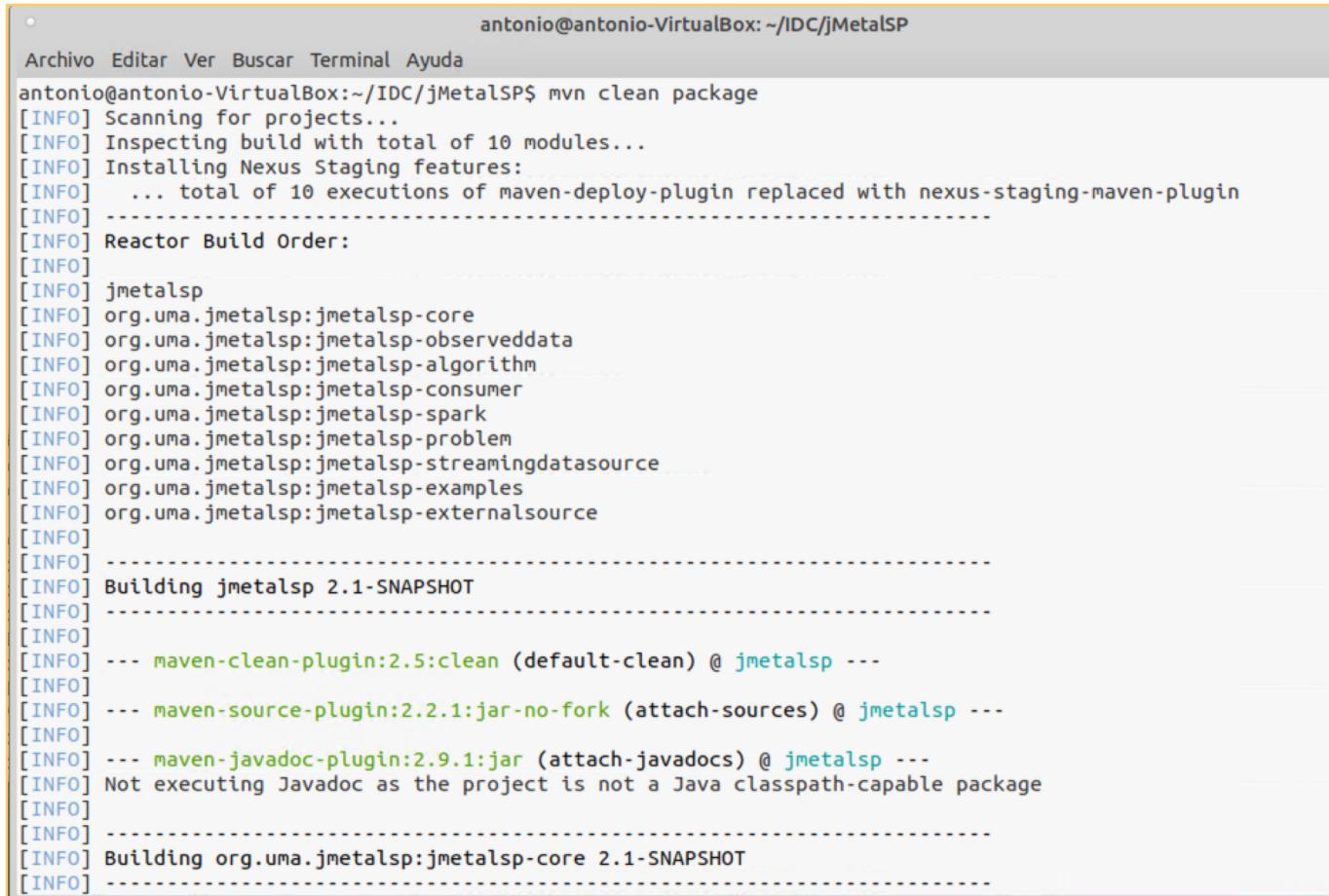


```
antonio@antonio-VirtualBox: ~/IDC
Archivo Editar Ver Buscar Terminal Ayuda

antonio@antonio-VirtualBox:~/IDC$ git clone https://github.com/jMetal/jMetalSP.git
Clonar en «jMetalSP»...
remote: Enumerating objects: 6626, done.
remote: Total 6626 (delta 0), reused 0 (delta 0), pack-reused 6626
Receiving objects: 100% (6626/6626), 3.21 MiB | 1.03 MiB/s, done.
Resolving deltas: 100% (1973/1973), done.
antonio@antonio-VirtualBox:~/IDC$ ls
jMetalSP
antonio@antonio-VirtualBox:~/IDC$ █
```

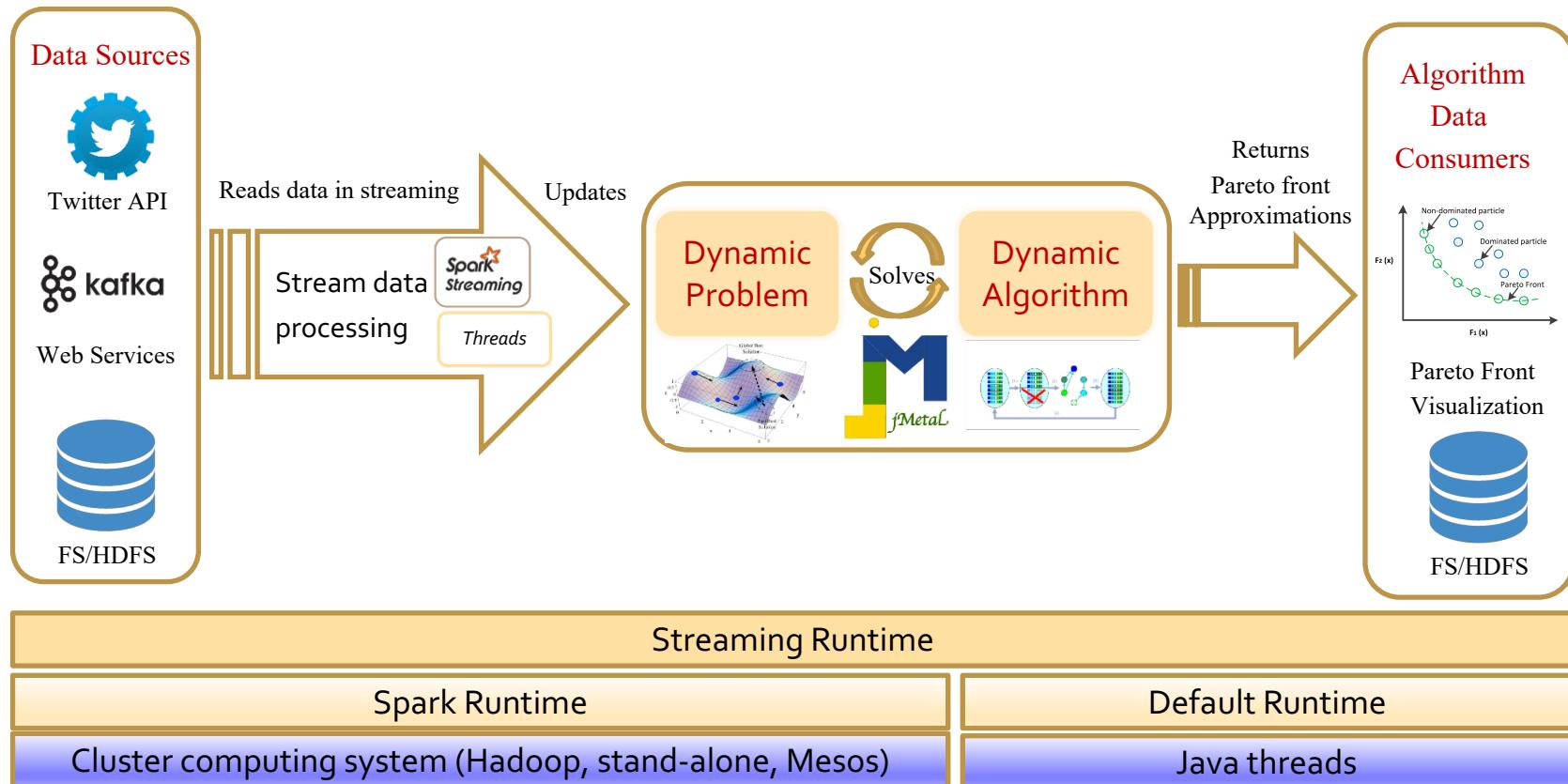
jMetalSP: a Maven Project Hosted in GitHub

- The project can be built from the command line using Maven

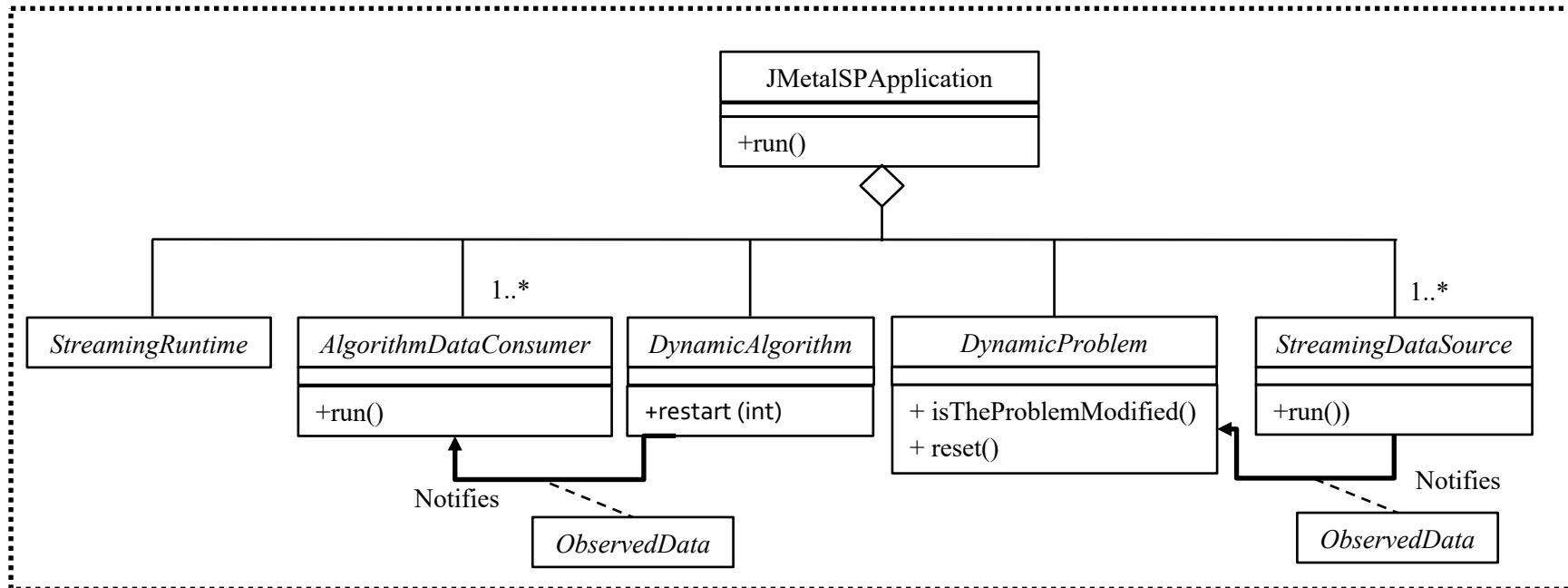


```
antonio@antonio-VirtualBox: ~/IDC/jMetalSP$ mvn clean package
[INFO] Scanning for projects...
[INFO] Inspecting build with total of 10 modules...
[INFO] Installing Nexus Staging features:
[INFO]   ... total of 10 executions of maven-deploy-plugin replaced with nexus-staging-maven-plugin
[INFO]
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] jmetalsp
[INFO] org.uma.jmetalsp:jmetalsp-core
[INFO] org.uma.jmetalsp:jmetalsp-observeddata
[INFO] org.uma.jmetalsp:jmetalsp-algorithm
[INFO] org.uma.jmetalsp:jmetalsp-consumer
[INFO] org.uma.jmetalsp:jmetalsp-spark
[INFO] org.uma.jmetalsp:jmetalsp-problem
[INFO] org.uma.jmetalsp:jmetalsp-streamingdatasource
[INFO] org.uma.jmetalsp:jmetalsp-examples
[INFO] org.uma.jmetalsp:jmetalsp-externalsource
[INFO]
[INFO] -----
[INFO] Building jmetalsp 2.1-SNAPSHOT
[INFO] -----
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ jmetalsp ---
[INFO] --- maven-source-plugin:2.2.1:jar-no-fork (attach-sources) @ jmetalsp ---
[INFO] --- maven-javadoc-plugin:2.9.1:jar (attach-javadocs) @ jmetalsp ---
[INFO] Not executing Javadoc as the project is not a Java classpath-capable package
[INFO]
[INFO] -----
[INFO] Building org.uma.jmetalsp:jmetalsp-core 2.1-SNAPSHOT
[INFO] -----
```

jMetalSP (2.0) Architecture



jMetalSP (2.0) UML class diagram



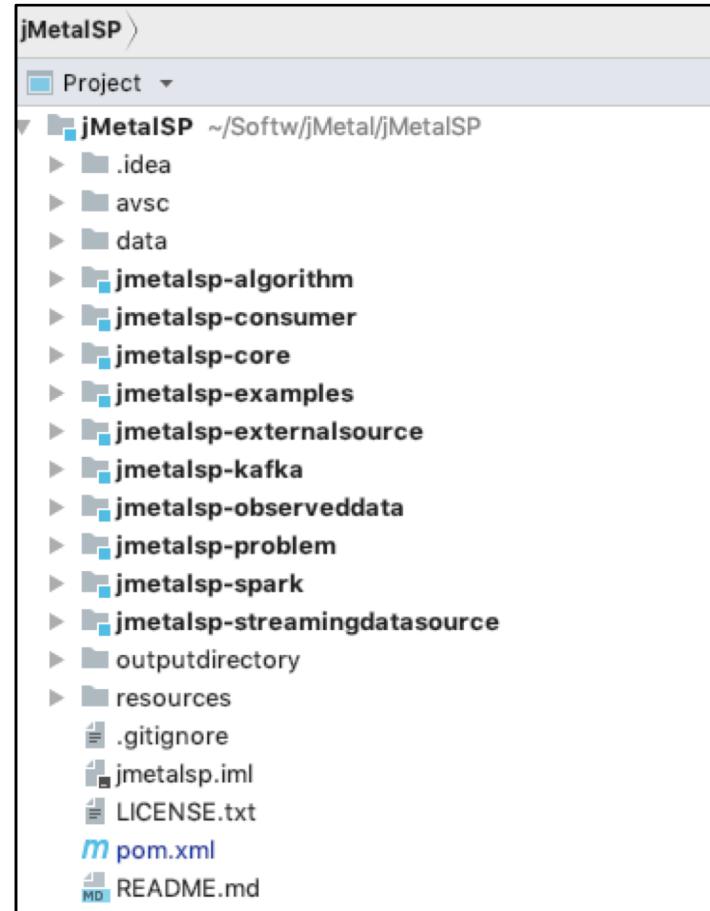
```

// Declaring application components
...
JMetalSPApplication<> application = new JMetalSPApplication<>();

application.setStreamingRuntime(new SparkRuntime())
.setProblem(problem)
.setAlgorithm(algorithm)
.addStreamingDataSource(streamingDataSource1, problem)
.addStreamingDataSource(streamingDataSource2, problem)
.addAlgorithmDataConsumer(dataConsumer1)
.addAlgorithmDataConsumer(dataConsumer2)
.run();
  
```

jMetalSP: a Maven Project Hosted in GitHub

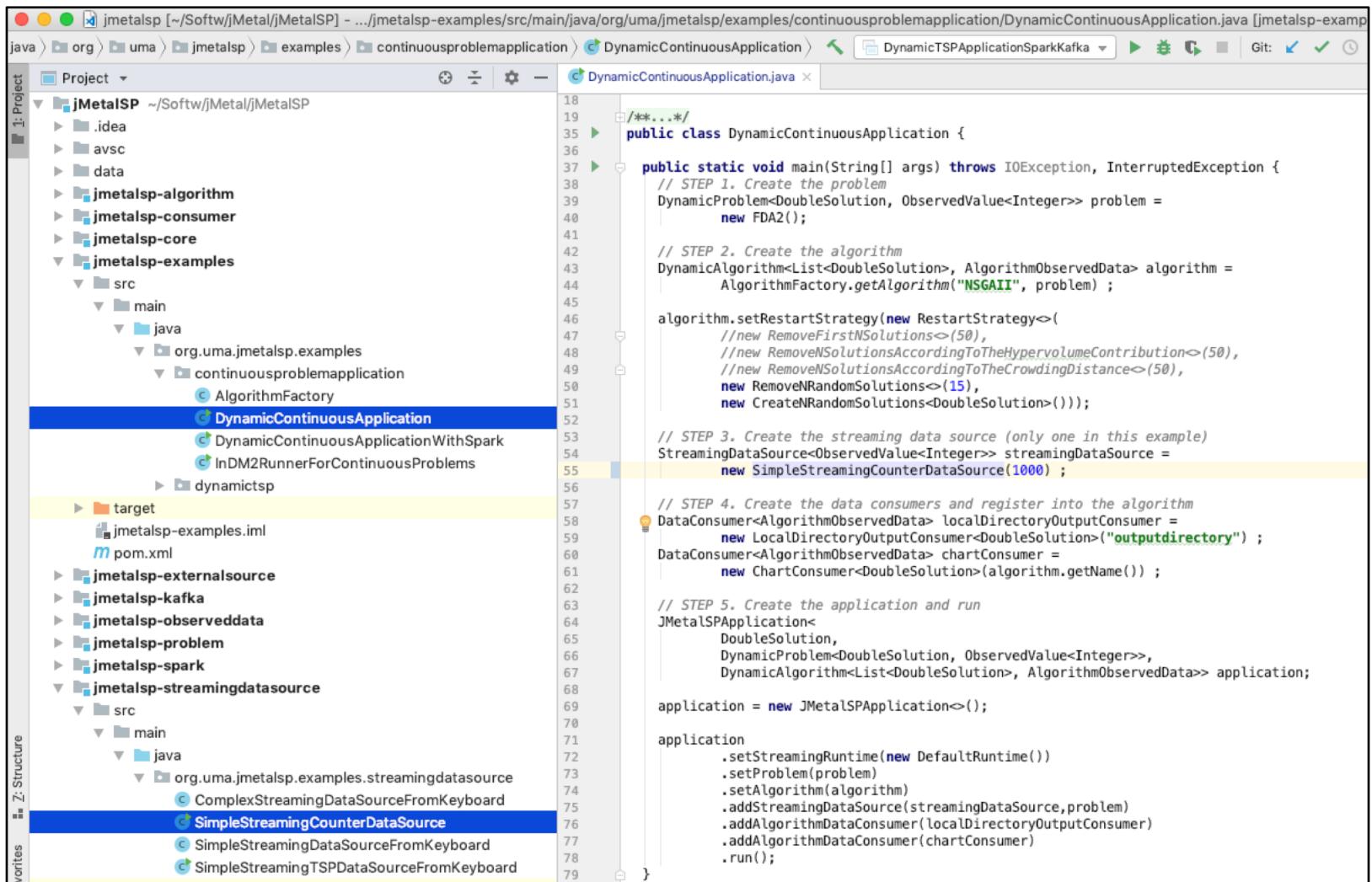
- jMetalSP is composed of a number of 10 Maven sub-packages
- In the current version (jMetalSP 2.0):
 - Multi-Objective algorithms (7)
 - Benchmark problems (5)
 - Streaming runtimes (2)



Algorithms included

- Dynamic versions of
 - NSGA-II
 - NSGA-III
 - MOCell
 - SMPSO
 - WASF-GA (Decision making)
 - R-NSGA-II (Decision making)
 - InDM2 (Decision making, interactive)

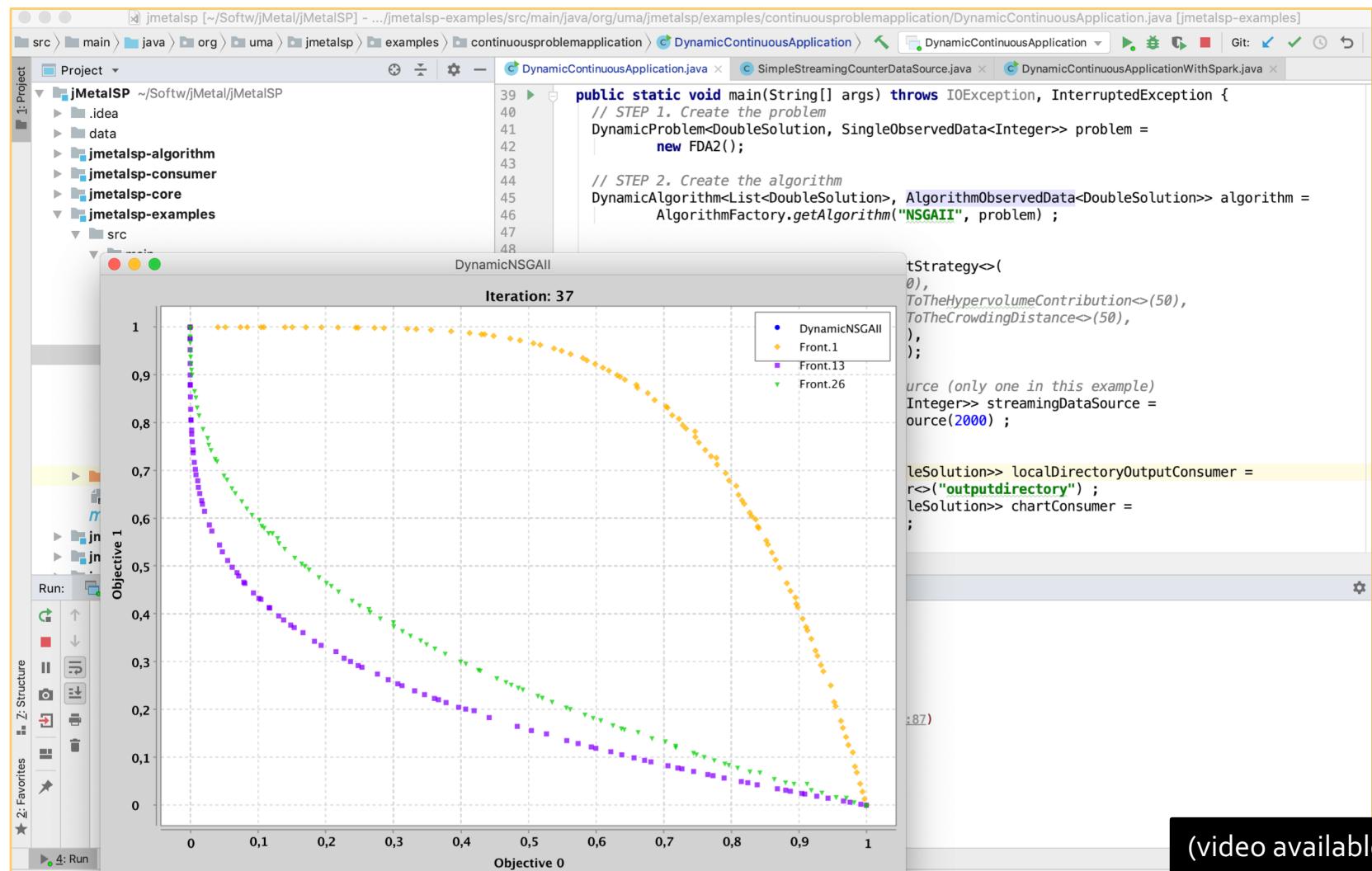
Example 1: DNGSA-II, FDA2, Default Runtime



The screenshot shows an IDE interface with the following details:

- Project View:** Shows the project structure under "jMetalSP". The "DynamicContinuousApplication" class is selected.
- Code Editor:** Displays the Java code for "DynamicContinuousApplication.java". The code implements a step-by-step process for creating a dynamic continuous application using the jMetalSP framework.
- Code Content:** The code includes:
 - // STEP 1. Create the problem
 - DynamicProblem<DoubleSolution, ObservedValue<Integer>> problem =
new FDA2();
 - // STEP 2. Create the algorithm
 - DynamicAlgorithm<List<DoubleSolution>, AlgorithmObservedData> algorithm =
AlgorithmFactory.getAlgorithm("NSGAII", problem);
 - algorithm.setRestartStrategy(new RestartStrategy<>(
//new RemoveFirstNSolutions<>(50),
//new RemoveSolutionsAccordingToTheHypervolumeContribution<>(50),
//new RemoveSolutionsAccordingToTheCrowdingDistance<>(50),
new RemoveRandomSolutions<>(15),
new CreateNRandomSolutions<DoubleSolution>()));
 - // STEP 3. Create the streaming data source (only one in this example)
 - StreamingDataSource<ObservedValue<Integer>> streamingDataSource =
new SimpleStreamingCounterDataSource(1000) ;
 - // STEP 4. Create the data consumers and register into the algorithm
 - DataConsumer<AlgorithmObservedData> localDirectoryOutputConsumer =
new LocalDirectoryOutputConsumer<DoubleSolution>("outputdirectory") ;
 - DataConsumer<AlgorithmObservedData> chartConsumer =
new ChartConsumer<DoubleSolution>(algorithm.getName()) ;
 - // STEP 5. Create the application and run
 - JMetalSPApplication<
DoubleSolution,
DynamicProblem<DoubleSolution, ObservedValue<Integer>>,
DynamicAlgorithm<List<DoubleSolution>, AlgorithmObservedData>> application;
 - application = new JMetalSPApplication<>();
 - application
.setStreamingRuntime(new DefaultRuntime())
.setProblem(problem)
.setAlgorithm(algorithm)
.addStreamingDataSource(streamingDataSource,problem)
.addAlgorithmDataConsumer(localDirectoryOutputConsumer)
.addAlgorithmDataConsumer(chartConsumer)
.run();

Example 1: DNGSA-II, FDA2, Default Runtime



The screenshot shows the jMetalSP IDE interface. On the left, the Project Explorer displays the project structure under the `DynamicContinuousApplication` package. In the center, a chart titled "Iteration: 37" plots Objective 1 against Objective 0. The chart shows four fronts: "DynamicNSGAII" (blue dots), "Front.1" (orange diamonds), "Front.13" (purple squares), and "Front.26" (green triangles). The axes range from 0 to 1. On the right, the code editor shows the `DynamicContinuousApplication.java` file. The code initializes a problem using `FDA2()` and creates an algorithm using `AlgorithmFactory.getAlgorithm("NSGAII", problem)`. It also defines a strategy with hypervolume and crowding distance contributions, sets up a streaming data source, and configures local directory output and chart consumers.

```
public static void main(String[] args) throws IOException, InterruptedException {
    // STEP 1. Create the problem
    DynamicProblem<DoubleSolution, SingleObservedData<Integer>> problem =
        new FDA2();

    // STEP 2. Create the algorithm
    DynamicAlgorithm<List<DoubleSolution>, AlgorithmObservedData<DoubleSolution>> algorithm =
        AlgorithmFactory.getAlgorithm("NSGAII", problem);

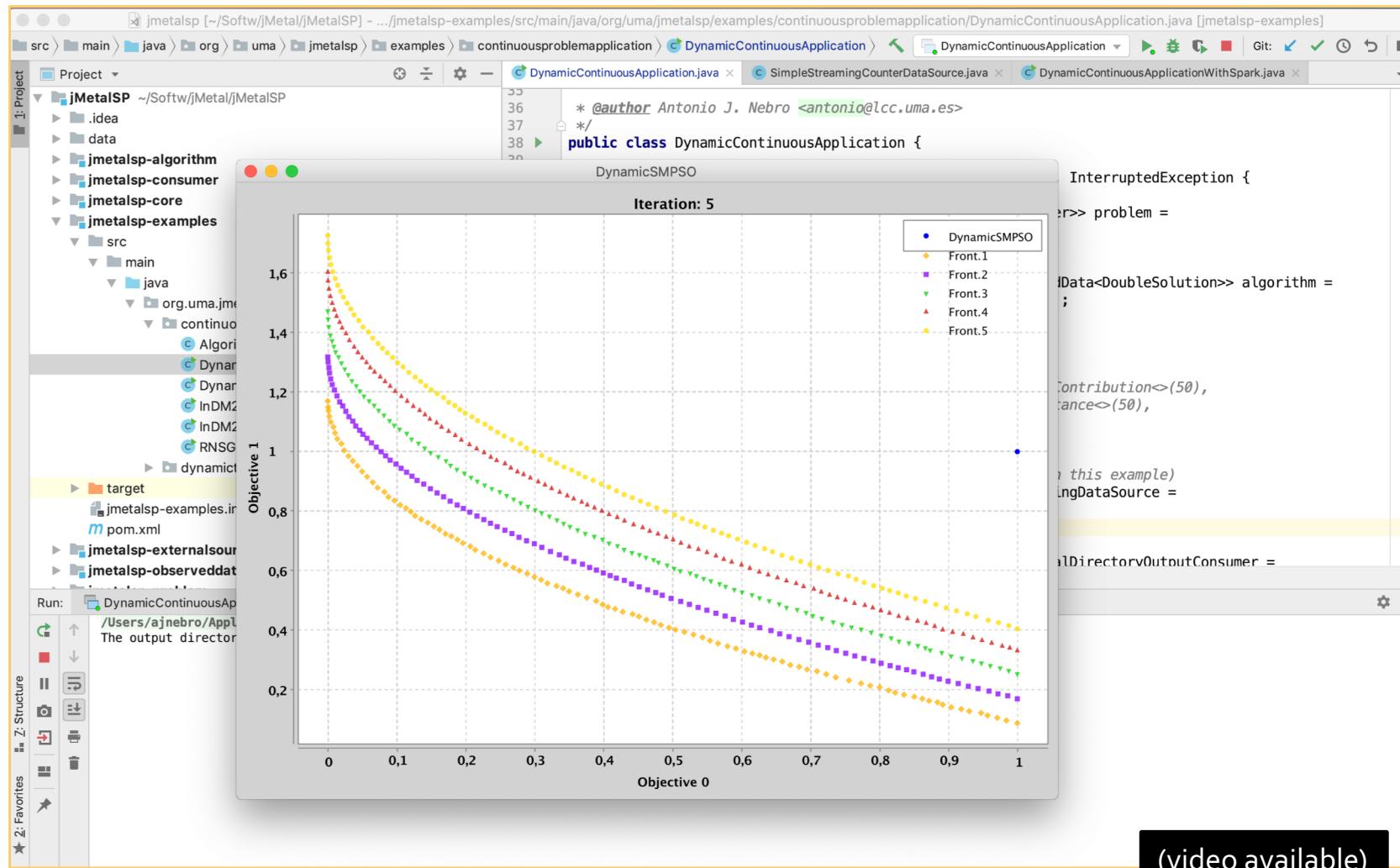
    tStrategy<(
    0),
    ToTheHypervolumeContribution<(50),
    ToTheCrowdingDistance<(50),
    ),
    );
}

// (only one in this example)
Integer>> streamingDataSource =
source(2000) ;

DoubleSolution>> localDirectoryOutputConsumer =
r>>("outputdirectory") ;
DoubleSolution>> chartConsumer =
;
```

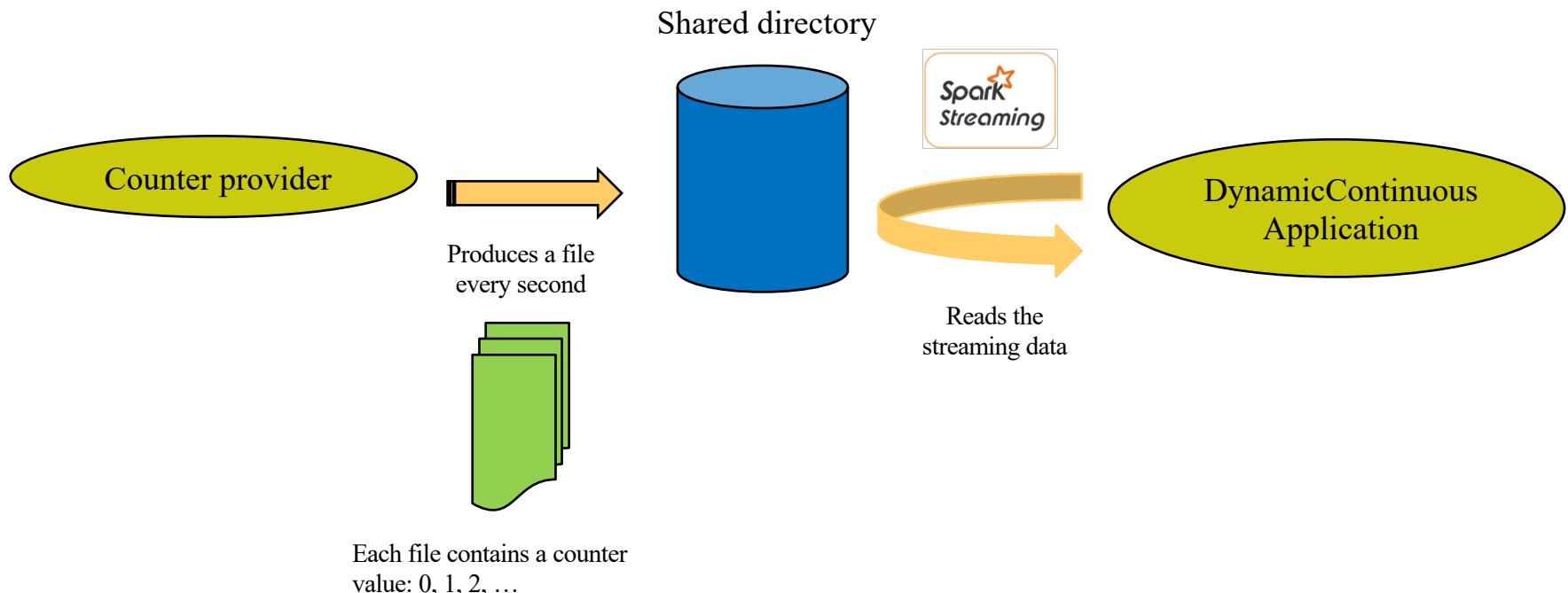
(video available)

Example 2: DSMPSO, FDA3, Default Runtime



(video available)

Example 3: DSMPSO, FDA2, Spark Runtime



Example 3: DSMPSO, FDA2, Spark Runtime

Screenshot of IntelliJ IDEA showing the code editor for a Java project named jmetalsp. The project structure on the left includes modules like jmetalsp-examples, jmetalsp-externalsource, and jmetalsp-spark. The code editor on the right displays a class named SimpleSparkStreamingCounterDataSource.java. The code implements a streaming data source using Apache Spark. It defines a constructor that takes an observable and a directory name, and overrides the run() method to process data from a text file stream. A lightbulb icon is shown over the reduce operation in line 49. The bottom right pane shows the main() method of a JMetalsPApplication, which initializes a sparkConf, sets a streaming runtime, and runs the application.

```

public SimpleSparkStreamingCounterDataSource(
    Observable<SingleObservedData<Integer>> observable,
    String directoryName) {
    this.observable = observable;
    this.directoryName = directoryName;
}

public SimpleSparkStreamingCounterDataSource(String directoryName) {
    this(new DefaultObservable<>(), directoryName);
}

@Override
public void run() {
    JMetalLogger.logger.info( msg: "Run method in the streaming data source invoked");
    JMetalLogger.logger.info( msg: "Directory: " + directoryName);

    JavaStream<Integer> time = streamingContext
        .textFileStream(directoryName)
        .map(Integer::parseInt);

    time.foreachRDD(numbers -> {
        if (numbers.rdd().count() > 0) {
            int value = numbers.reduce((value1, value2)-> value1);
            //int value = numbers.collect().get(0) // Does not work after Spark 1.6
            System.out.println("Value: " + value);
            observable.setChanged();
            observable.notifyObservers(new SingleObservedData<Integer>(value));
        }
    });
}

@Override
public Observable<SingleObservedData<Integer>> getObservable() {
    return observable;
}

```

```

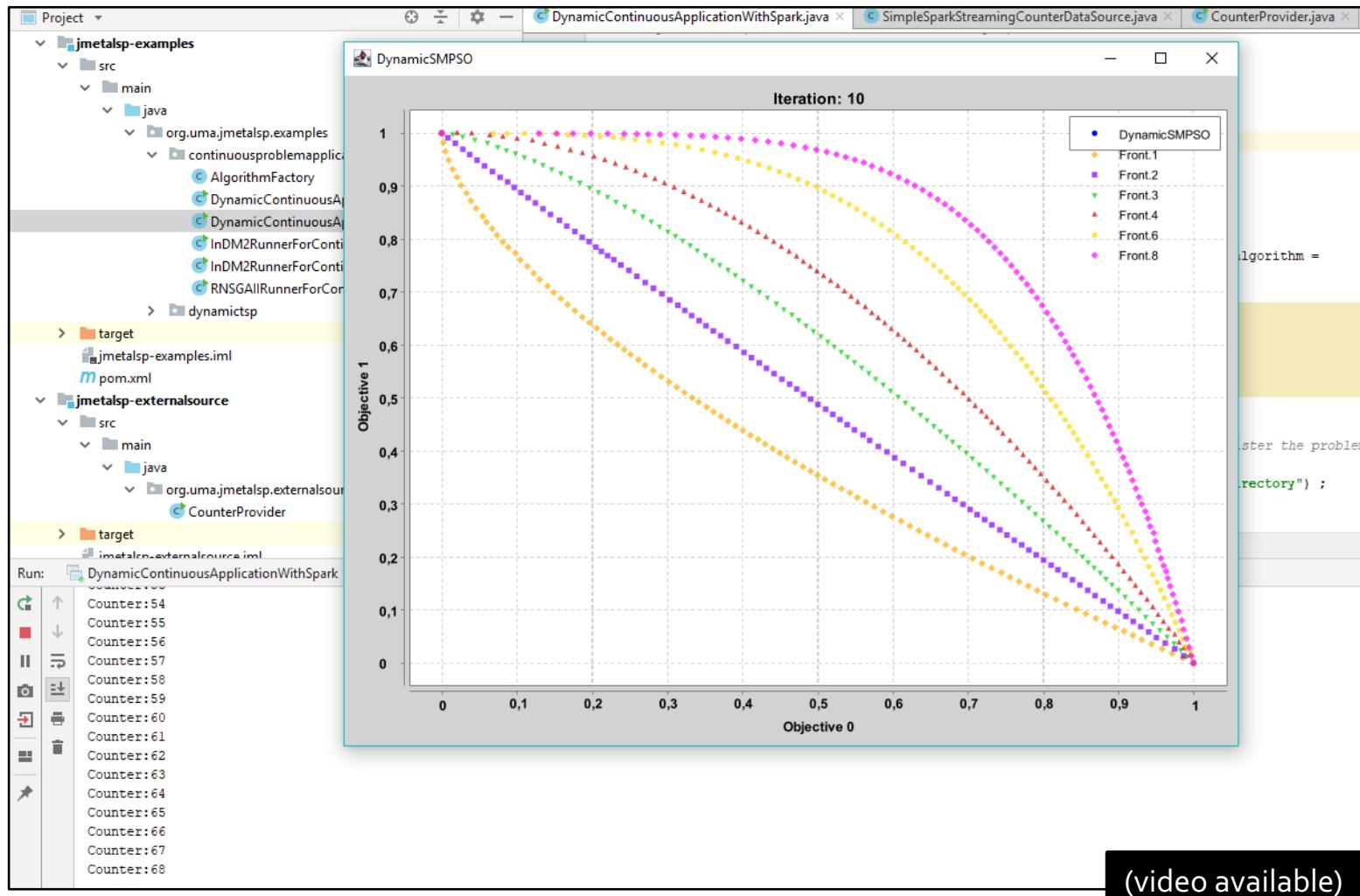
application = new JMetalsPApplication<>();

SparkConf sparkConf = new SparkConf()
    .setAppName("SparkApp")
    .setSparkHome("F:\\spark-2.3.1-bin-hadoop2.7")
    .setMaster("local[4]" );

application.setStreamingRuntime(new SparkRuntime( duration: 1, sparkConf))
    .setProblem(problem)
    .setAlgorithm(algorithm)
    .addStreamingDataSource(streamingDataSource,problem)
    .addAlgorithmDataConsumer(localDirectoryOutputConsumer)
    .addAlgorithmDataConsumer(chartConsumer)
    .run();

```

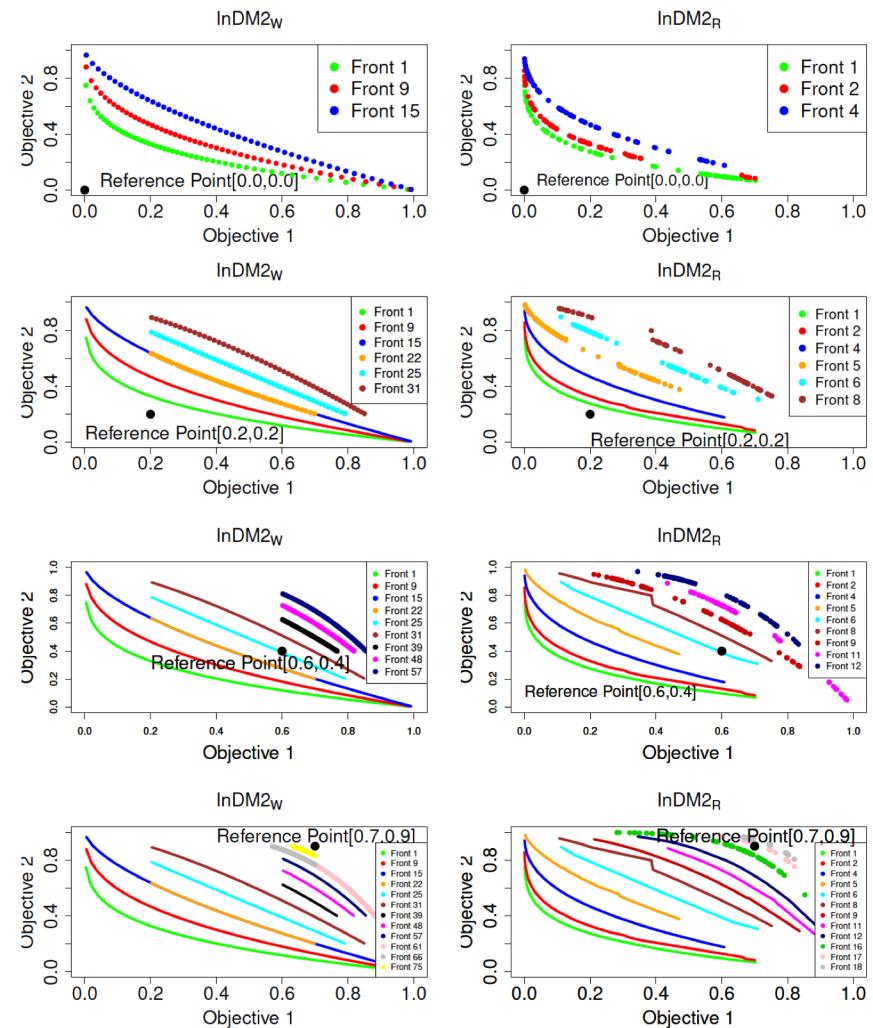
Example 3: DSMPSO, FDA2, Spark Runtime



(video available)

Example 4: InDM2, FDA2, Default runtime

- InDM2
 - Interactive Dynamic Multi-objective Decision Making algorithm
- Intended for
 - Large executions
 - Guide the search to the desired region



A.J. Nebro, A.B. Ruiz, C. Barba-González, J. García Nieto,
 M.M. Roldán-García, M. Luque, J.F. Aldana Montes.
*InDM2: Interactive Dynamic Multi-Objective Decision
 Making Using Evolutionary Algorithms. Swarm and
 Evolutionary Computation*, Vol. 40, pp: 184:195. June
 2018.

(video available)

Table of contents

- Who, why, what, when
- Background
- Working with jMetalSP
- **Case study**
- Further developments

Case Study: Real Time Traffic Problem

- Traveling Salesman Problem (TSP) based on New York City's real-time traffic data.
- Data are from New York City Open Data Website (<https://opendata.cityofnewyork.us/>).



The screenshot shows the homepage of the NYC Open Data website. At the top, there is a navigation bar with links for Home, Data, About, Learn, Alerts, Contact Us, Blog, and Sign In. On the far left of the header is the NYC Open Data logo. To the right of the header is a search bar with the placeholder "Search all NYC.gov websites". Below the header, a large blue banner features the text "Open Data for All New Yorkers" in white. Below this text is a paragraph of descriptive text: "Where can you find public Wi-Fi in your neighborhood? What kind of tree is in front of your office? Learn about New York City using NYC Open Data." At the bottom of the banner is a search bar with the placeholder "Search Open Data for things like 311, Buildings, Crime". To the right of the banner is a video player showing a woman wearing a hard hat and holding a clipboard, with the text "NYC Open Data: Public Data at Work" above it.

How You Can Get Involved

Case Study: Real Time Traffic Problem

- This is a classical academic problem to which we have incorporated a combination of real nodes (street geo-locations) and real traffic data from the city of New York.
- Dynamic version of the bi-objective Traveling Salesman Problem (DTSP).
- The goal is to minimize the travel time and the distance to cover all the points of the instance.

Case Study: Real Time Traffic Problem

- DTSP problem is composed of 93 locations and 315 communications between them.
- The links are bi-directional, so the resulting DTSP is asymmetric.
- To determine the distance between points, we have used a Google service.



Case Study: Real Time Traffic Problem

- <https://data.cityofnewyork.us/resource/i4gi-tjb9.json>

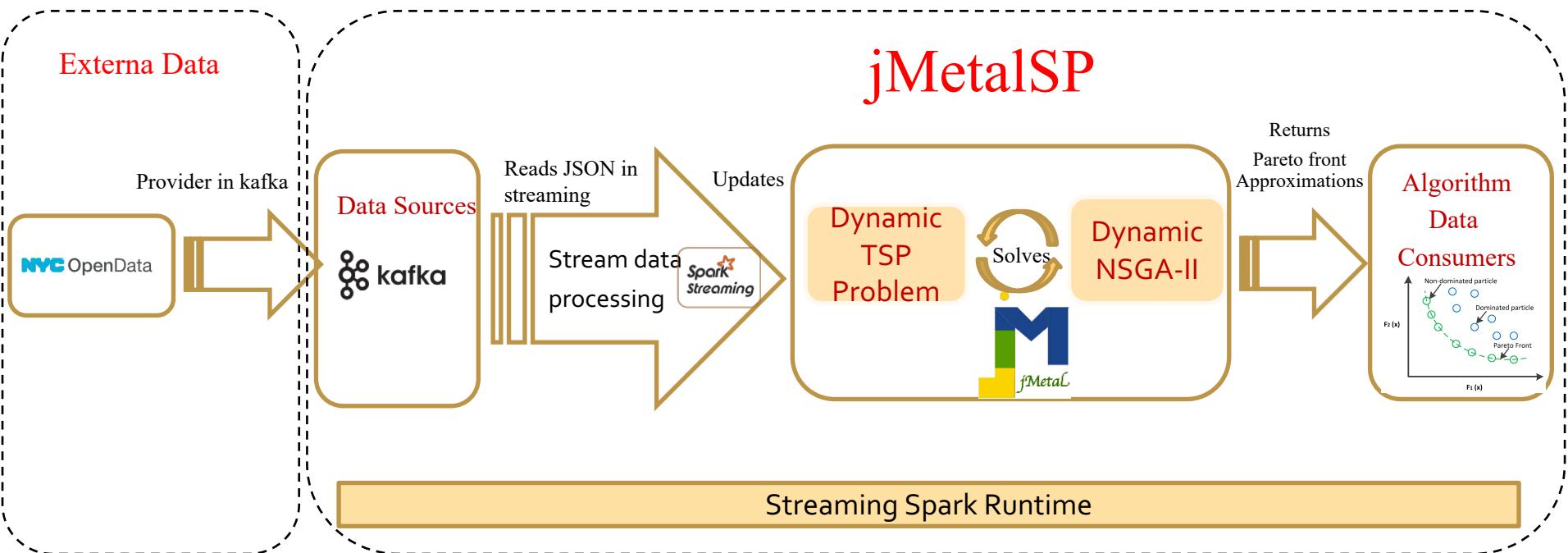
```
{  
  "name": "NY Traffic",  
  "fields": [  
    {  
      "name": "id",  
      "type": "int"  
    },  
    {  
      "name": "speed",  
      "type": "double"  
    },  
    {  
      "name": "travel_time",  
      "type": "int"  
    },  
  ]}
```

```
{  
  "name": "status",  
  "type": "int"  
},  
{  
  "name": "encoded_poly_line",  
  "type": "string"  
},  
{  
  "name": "link_name",  
  "type": "double"  
}]}
```

Case Study: Real Time Traffic Problem

- **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.
- **LinkName:** description of the link location and end points (a string).

Case Study: Real Time Traffic Problem



Case Study: Real Time Traffic Problem

```
SimpleSparkStructuredKafkaStreamingTSP.java ×
```

```
102
103
104     JavaDStream<List<ParsedNode>> nodes=stream.map(value ->{
105
106         List<ParsedNode> result= new ArrayList<>();
107         final JSONArray parser = new JSONArray(value.value());
108         for (int i = 0; i < parser.length(); i++) {
109             try{
110                 final JSONObject object = parser.getJSONObject(i);
111                 ParsedNode pNode = new ParsedNode(
112                     object.getInt( key: "id"),
113                     object.getDouble( key: "speed"),
114                     object.getInt( key: "travel_time"),
115                     (object.getDouble( key: "status")==0.0d),
116                     object.getString( key: "encoded_poly_line"),
117                     object.getString( key: "link_name"),
118                     GoogleDecode.decode(object.getString( key: "encoded_poly_line"))
119                 );
120
121                 if(nodeDistances.containsKey(pNode.getId())){
122                     pNode.setDistance(nodeDistances.get(pNode.getId()));
123                 }
124                 result.add(pNode);
125             }catch (Exception ex){
126                 ex.printStackTrace();
127             }
128         }
```

Case Study: Real Time Traffic Problem

```
SimpleSparkStructuredKafkaStreamingTSP.java x

194     nodes.foreachRDD(aux-> {
195         if (aux != null && aux.rdd().count()>0) {
196             List<ParsedNode> pNodes = aux.reduce((key, value) -> value);
197             for (ParsedNode node : pNodes) {
198                 if (hashNodes.get(node.getId()) != null) {
199                     ParsedNode nodeAux = hashNodes.get(node.getId());
200                     if (nodeAux.isStatus() != node.isStatus()) {
201                         if (node.isStatus()) {
202                             node.setDistance(Integer.MAX_VALUE);
203                             node.setTravelTime(Integer.MAX_VALUE);
204                         } else if (nodeDistances.containsKey(node.getId())) {
205                             node.setDistance(nodeDistances.get(node.getId()));
206                         }
207                         node.setDistanceUpdated(true);
208                         node.setCostUpdated(true);
209                         nodeAux.setStatus(node.isStatus());
210                     }
211                     if (node.getTravelTime() != nodeAux.getTravelTime()) {
212                         nodeAux.setTravelTime(node.getTravelTime());
213                         node.setCostUpdated(true);
214                     }
215                 } else {
216                     hashNodes.put(node.getId(), node);
217                     addManualEdges();
218                 }
219                 TSPMatrixData matrix = generateMatrix(node);
220                 observable.setChanged();
221                 observable.notifyObservers(new ObservedValue<>(matrix));

```

Case Study: Real Time Traffic Problem

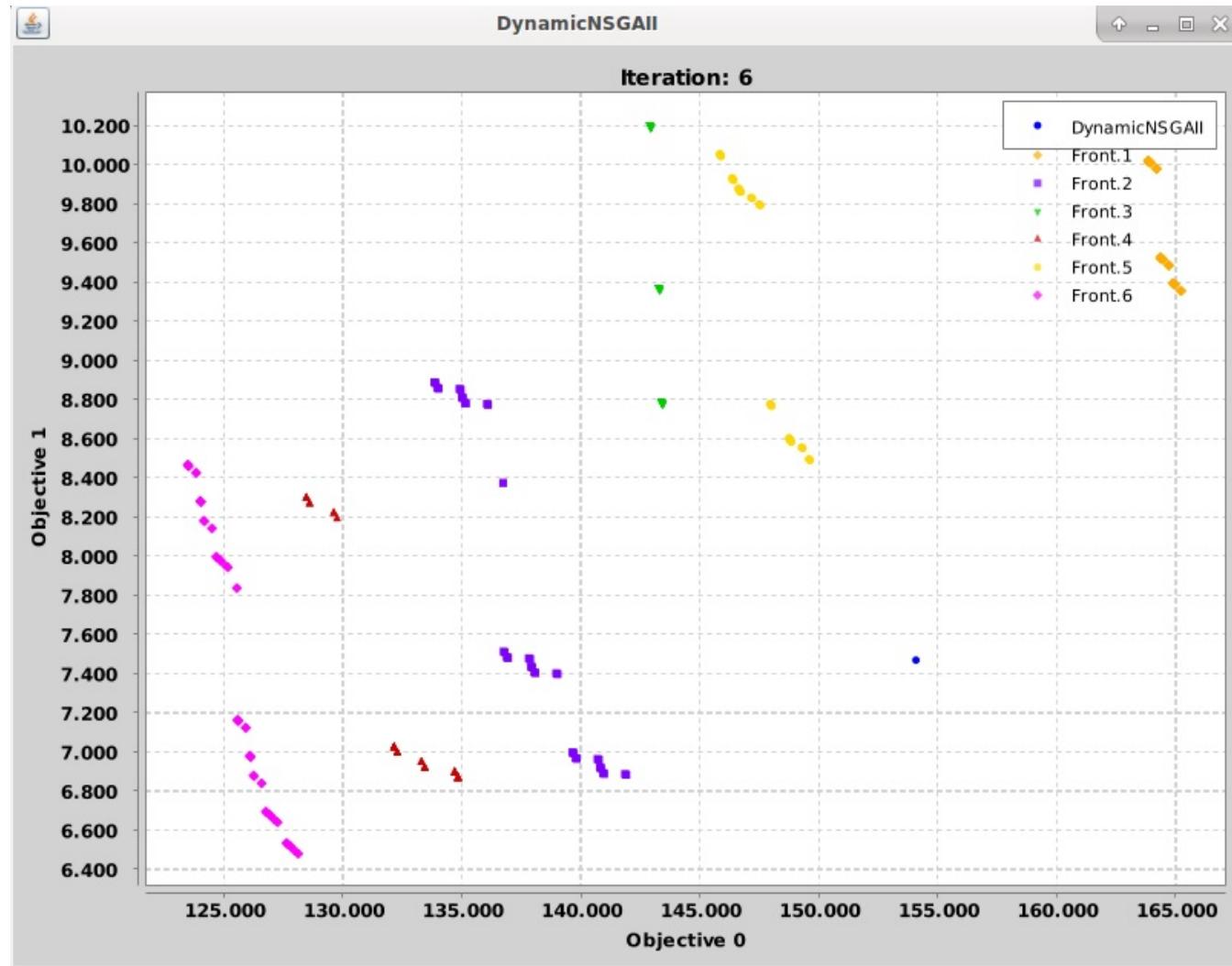


Table of contents

- Who, why, what, when
- Background
- Working with jMetalSP
- Case study
- **Further developments**

Further developments

- jMetalSP has two types of users
 - Designers of dynamic multi-objective algorithms
 - Users requiring the use of a streaming engine for processing large amounts of data
- For the second group
 - The API of Spark streaming will be replaced by Spark Structured Streaming
 - There are other streaming projects
 - Flink
 - Kafka Streams
- **Line of work 1:** incorporating these systems into jMetalSP

Further developments

- All the components of jMetalSP run as threads of a same process
 - This can limit the usefulness of jMetalSP applications
 - The streaming processing could be near sensors generating the data
 - The optimization algorithm could run on a high performance computer
 - The graphical output could be visualized on a Web application
- **Line of work 2:** using Kafka to decouple all the jMetal components

Further developments

- We have used two applications with jMetalSP
 - Dynamic benchmark problems
 - Transportation problem
 - One data source with an update frequency of about 1 minute
- Line of work 3: find real-world problems

Thanks for your attention ☺

- Comments and suggestions are welcome

