Single execution EXPLoRa-C full Python

This execution method is the one from I started, with the EXPLoRa-C version implemented in Python.

Code files involved in the execution:

- **simple_deployment.py**
- **loraDirMulBSfading_uniformSF_collSF_v9_5.py**
- **explora_at_functionsMulBS_v9_5.py**

How to run:

In an environment with python type the command line

- "python .\simple_deployment.py"

Note that is mandatory to install all the libraries required

Files description:

- In the file **simple_deployment.py** there is the code to launch with python scripts the PowerShell commands, this file has been created to keep track of all the parameters needed to run Lorasim for the execution of a simulation. In the file are listed all the parameters with attached descriptions
- The file **loraDirMulBSfading_uniformSF_collSF_v9_5.py** contains the main Lorasim code divided into the following phases:
    - Scenario generation with the instantiation of one objects (threads) for each Node
    - Execution of the ADR algorithm for the calculation of the parameters to be given to each object (Node)
    - Execution of the simulation, launching in parallel one thread for every Node and collecting eventual collisions of the packets transmitted from every single Node
- In the file **explora_at_functionsMulBS_v9_5.py** there is the code of the EXPLoRa-C algorithm used as ADR by Lorasim

Execution description:

This execution is done entirely in the Python environment, so you don't have to perform any operations on the Java code.
In order to make the code as modular as possible, the code of the Lorasim simulator was modified so that the three different phases could also be executed individually.
In particular the parameter *nodes_file* (to give in input to Lorasim) present in the file **simple_deployment_explora_v_python.py** indicates to the simulator if to create a new scene (on the base of the other parameters in input) or if to use one already created.
In fact if the parameter *nodes_file* is initialized with the string "None", Lorasim will create a new scenario (operation that uses a lot of time), instead if it is initialized with the name of the file in which the data of a scenario are stored, Lorasim will read the file and will create the different objects (threads) for each node in the file.
It is important to specify that ALL files generated during the execution should be saved in the *python/simulation_files* directory. This folder is the same from which the Lorasim simulator in the file **loraDirMulBSfading_uniformSF_collSF_v9_5.py** will search for the file containing the simulation scenario indicated by the variable *nodes_file*.
In particular, the output of a complete Lorasim execution of the three phases will produce the following files:

- *(number of nodes)-nodes-raw.txt,* file that contains the information of each node in the scenario. In particular each row represents one node and the fields in the columns are:

1. The node unique ID
2. The node position on the x axe
3. The node position on the y axe
4. On couple of column (RSSI, SF) for each basestation in the scenario representing the RSSI and the starting Spreading Factor of one node seen by each basestation.

This file is the one to give as input with the *nodes_file* variable in order to avoid a new scenario creation and run the ADR and the simulation on a already created scenario.

- *basestation-(distanceBetweenGW).txt,* file that contains the information of each basestation in the scenario. In particular each row represent one basestation and the fields in the columns are:
    1. The basestation position on the x axe
    2. The basestation position on the y axe
    3. The basestation unique ID
- *(number of nodes)-nodes-raw-final-sfinput-(SF_input)-full-True-(distanceBetweenGW).txt*, file that has the same structure of the *(number of nodes)-nodes-raw.txt* but in this case the Spreading Factors associated to each basestation are the one given by the ADR algorithm
- *nodes-adr-(SF_input)-(number of basestations) -(number of nodes)-True.txt*, file that the same information of the *(number of nodes)-nodes-raw-final-sfinput-(SF_input)-full-True-(distanceBetweenGW).txt* file with other simulations results appended as columns

In addition the execution of Lorasim will print on the shell useful information about the performances like the average Data Extraction Rate (DER).

Single execution EXPLoRa-C Java

This execution allows Lorasim simulator to use the Java version of the EXPLoRa-C algorithm as ADR. This execution explanation can be used for both the classical EXPLoRa-C implementation and for the EXPLoRa-C functional implementation.

Code files involved in the execution:

- **simple_deployment.py**
- **loraDirMulBSfading_uniformSF_collSF_v9_5.py**
- **java Maven project**

Setup the code parameters:

- It is important to change the path in the **Main.java class** in order to allow the Java code to read the scenario from the same file of the Lorasim code.
  In particular at the start of each function (in the Main.java class) in which there is the code of a particular version of EXPLoRa-C (in this case we consider the
  public static int[] EXPLoRa_C_py_functional() function), there is a variable called *pathToNodeFile* in which should be inserted the path to the scenario created by Lorasim (e.g.
  *C:/********/python/simulation_files/(number of nodes)-nodes-raw.txt*)
- In the **simple_deployment.py** file change the value of the *javaPythonVersion* variable with one string between “java-start” or “java” depending of what version of EXPLoRa-C you would run, respectively the starting one or the functional one

How to run:

- Start the Java Maven project (recommended the usage of NetBeans)
  - At the start the maven will display a sort of Menù:
    Press 0+Enter to start the EXPLoRa_C_py algorithm
    Press 1+Enter to start the EXPLoRa_C_py_functional algorithm
    Press 2+Enter to start the EXPLoRa_CD_py
    Press 3+Enter to start the EXPLoRa_CD_geoDivision_py
    Press 4+Enter to start the EXPLoRa_CD_multiexecution_py
    Press 5+Enter to start the EXPLoRa_CD_geoDivision_splitVision_py
    Press 6+Enter to start the EXPLoRa_CD_splitVision_py
    Press 22+Enter to start the time performance comparation between EXPLoRa-C functional and multithread
    Press 222+Enter to open the gateway for the python interaction
- Digit 222 and press Enter
  - This will start the gateway used as cross language channel
- In an environment with python type the command line
  “python .\simple_deployment.py”

Files description:

- In the file **simple_deployment.py** there is the code to launch with python scripts the PowerShell commands, this file has been created to keep track of all the parameters needed to run Lorasim for the execution of a simulation. In the file are listed all the parameters with attached descriptions
- The file **loraDirMulBSfading_uniformSF_collSF_v9_5.py** contains the main Lorasim code divided into the following phases:
  - Scenario generation with the instantiation of one objects (threads) for each Node
  - Execution of the ADR algorithm for the calculation of the parameters to be given to each object (Node)

- Execution of the simulation, launching in parallel one thread for every Node and collecting eventual collisions of the packets transmitted from every single Node

- The **Java Maven Project** has the majority of functionalities in the *Main.java class*. In particular in this class there are several functions that are detailed in the following:
  - *public static void main()* : The main function is the first function to be performed at the start of the project, it contains the code to choose which operation you want to perform. In general it gives the possibility to execute the single different algorithms of EXPLoRa-C or to open the gateway for the communication with python Lorasim
  - *public static int[] EXPLoRa_C_py()* : EXPLoRa-C algorithm implemented exactly as described in the paper "Capture Aware Sequential Waterfilling for LoraWAN Adaptive Data Rate"
  - *public static int[] EXPLoRa_C_py_functional()* : EXPLoRa-C algorithm modified to fit into a quasi-functional view using the Java Stream library
  - *public static int[] EXPLoRa_CD_py()* : EXPLoRa-C multithreaded algorithm with division method based on RSSI
  - *public static int[] EXPLoRa_CD_geoDivision_py()* : EXPLoRa-C multithreaded algorithm with division method based on geometry
  - *public static void EXPLoRa_CD_multiexecution_py()* : EXPLoRa-C multithreaded algorithm with division method based on RSSI. This version is designed for oversized scenarios that do not fit entirely into RAM. In fact, only a subset of gateways and nodes are loaded into RAM. In particular are initially chosen subgroups of gateways and at every iteration are processed a subgroup by filtering from the scene of simulation the nodes with greater RSSI associated to the chosen gateways.
  - *public static int[] EXPLoRa_CD_geoDivision_splitVision_py()* : EXPLoRa-C Distributed algorithm with division method based on geometry
  - *public static int[] EXPLoRa_CD_splitVision_py()* : EXPLoRa-C Distributed algorithm with division method based on RSSI

  All the functions that run the various versions of EXPLoRa-C initially read the simulation scenario from a file created by the Lorasim simulator, for which the path must be indicated.
  The final output value of each of these functions is an array of integers filled with the spreading factors calculated by the algorithm in object, sorted so that the spreading factor at position X of the array is the one associated with the node with ID X.

Execution description:

All the part related to the python Lorasim simulator remains unchanged with respect to the single execution EXPLoRa-C full Python, except for the call to the ADR algorithm that in this case uses the py4j library to call the function of the Java Project Maven *public static int[] EXPLoRa_C_py_functional()* like shown in the following figure.

```
407             elif javaPythonVersion == "java":
408                 print("EXECUTING JAVA FUNCTIONAL VERSION OF EXPLORA-C")
409                 gateway = JavaGateway()
410                 resultingSFs = gateway.entry_point.EXPLoRa_C_py_functional(nrNodes)
411                 for i in range(SF_local.size):
412                     SF_local[i][0] = resultingSFs[i]
```

After the execution on the Java Project, the gateway will return the array with the calculated spreading factors and the Lorasim simulator will continue with the running of the simulation.

Single execution EXPLoRa-C with division method based on RSSI

This explanation encapsulates the execution of EXPLoRa-C multithreaded and Distributed versions that use the RSSI-based division method.

Code files involved in the execution:

- **simple_deployment.py**
- **loraDirMulBSfading_uniformSF_collSF_v9_5.py**
- **java Maven project**

Setup the code parameters:

- It is important to change the path in the **Main.java class** in order to allow the Java code to read the scenario from the same file of the Lorasim code.
  In particular at the start of each function (in the Main.java class) in which there is the code of a particular version of EXPLoRa-C (in this case we consider the
  public static int[] EXPLoRa_C_py_functional() function), there is a variable called *pathToNodeFile* in which should be inserted the path to the scenario created by Lorasim (e.g.
  *C:/********/python/simulation_files/(number of nodes)-nodes-raw.txt*)
- In the **simple_deployment.py** file change the value of the *javaPythonVersion* variable with one string between "java-cd-m1" or "java-cd-m1-splitVision" depending of what version of EXPLoRa-C you would run, respectively the multithread one or the Distributed one
- In the **simple_deployment.py** file change the value of the *javaEXPLoRaCDdivision* variable with the number of splits in which you would divide the simulation scenario (recommended the value 2 with all other variables at the default value)

How to run:

- Start the Java Maven project (recommended the usage of NetBeans)
  - At the start the maven will display a sort of menu:
    Press 0+Enter to start the EXPLoRa_C_py algorithm
    Press 1+Enter to start the EXPLoRa_C_py_functional algorithm
    Press 2+Enter to start the EXPLoRa_CD_py
    Press 3+Enter to start the EXPLoRa_CD_geoDivision_py
    Press 4+Enter to start the EXPLoRa_CD_multiexecution_py
    Press 5+Enter to start the EXPLoRa_CD_geoDivision_splitVision_py
    Press 6+Enter to start the EXPLoRa_CD_splitVision_py
    Press 22+Enter to start the time performance comparation between EXPLoRa-C functional and multithread
    Press 222+Enter to open the gateway for the python interaction
- Digit 222 and press Enter
  - This will start the gateway used as cross language channel
- In an environment with python type the command line
  "python .\simple_deployment.py"

Files description:

- In the file **simple_deployment.py** there is the code to launch with python scripts the PowerShell commands, this file has been created to keep track of all the parameters needed to run Lorasim for the execution of a simulation. In the file are listed all the parameters with attached descriptions

- The file **loraDirMulBSfading_uniformSF_collSF_v9_5.py** contains the main Lorasim code divided into the following phases:
    - Scenario generation with the instantiation of one objects (threads) for each Node
    - Execution of the ADR algorithm for the calculation of the parameters to be given to each object (Node)
    - Execution of the simulation, launching in parallel one thread for every Node and collecting eventual collisions of the packets transmitted from every single Node
- The **Java Maven Project** has the majority of functionalities in the *Main.java class*. In particular in this class there are several functions that are detailed in the following:
    - *public static void main()* : The main function is the first function to be performed at the start of the project, it contains the code to choose which operation you want to perform. In general it gives the possibility to execute the single different algorithms of EXPLoRa-C or to open the gateway for the communication with python Lorasim
    - *public static int[] EXPLoRa_C_py()* : EXPLoRa-C algorithm implemented exactly as described in the paper "Capture Aware Sequential Waterfilling for LoraWAN Adaptive Data Rate"
    - *public static int[] EXPLoRa_C_py_functional()* : EXPLoRa-C algorithm modified to fit into a quasi-functional view using the Java Stream library
    - *public static int[] EXPLoRa_CD_py()* : EXPLoRa-C multithreaded algorithm with division method based on RSSI
    - *public static int[] EXPLoRa_CD_geoDivision_py()* : EXPLoRa-C multithreaded algorithm with division method based on geometry
    - *public static void EXPLoRa_CD_multiexecution_py()* : EXPLoRa-C multithreaded algorithm with division method based on RSSI. This version is designed for oversized scenarios that do not fit entirely into RAM. In fact, only a subset of gateways and nodes are loaded into RAM. In particular are initially chosen subgroups of gateways and at every iteration are processed a subgroup by filtering from the scene of simulation the nodes with greater RSSI associated to the chosen gateways.
    - *public static int[] EXPLoRa_CD_geoDivision_splitVision_py()* : EXPLoRa-C Distributed algorithm with division method based on geometry
    - *public static int[] EXPLoRa_CD_splitVision_py()* : EXPLoRa-C Distributed algorithm with division method based on RSSI

  All the functions that run the various versions of EXPLoRa-C initially read the simulation scenario from a file created by the Lorasim simulator, for which the path must be indicated.
  The final output value of each of these functions is an array of integers filled with the spreading factors calculated by the algorithm in object, sorted so that the spreading factor at position X of the array is the one associated with the node with ID X.

Execution description:

All the part related to the python Lorasim simulator remains unchanged with respect to the single execution EXPLoRa-C Java.
The Java functions *EXPLoRa_CD_py()* and *EXPLoRa_CD_splitVision_py()* use the division method based on the RSSI value that each node has in association with each gateway. This division is implemented in the initial part of these functions and is divided into two phases, the first phase identifying a subset of gateways, and a second phase assigning each node in the simulation scenario to one of the gateways in the subset based on the RSSI that each node has in association with those gateways. The first phase of gateway selection is not the result of an algorithm but of an "almost manual" selection through a series of IFs.

Single execution EXPLoRa-C with division method based on geometry

This explanation encapsulates the execution of EXPLoRa-C multithreaded and Distributed versions that use the geometry-based division method.

Code files involved in the execution:

- **simple_deployment.py**
- **loraDirMulBSfading_uniformSF_collSF_v9_5.py**
- **node_clustering_on_position.py**
- **quadtree.py**
- **java Maven project**

Setup the code parameters:

- Note that in this case the path in the **Main.java class** could be not changed because the Java code takes as input the file generated by the **node_clustering_on_position.py**
- In the **simple_deployment.py** file change the value of the *javaPythonVersion* variable with one string between "java-cd-m2" or "java-cd-m2-splitVision" depending of what version of EXPLoRa-C you would run, respectively the multithread one or the Distributed one
- In the **simple_deployment.py** file change the value of the *javaEXPLoRaCDdivision* variable with the number of splits in which you would divide the simulation scenario (recommended the value 2 with all other variables at the default value)
- In the **simple_deployment.py** file change the values of the *javaEXPLoRaCDdivision_x* and *javaEXPLoRaCDdivision_y* variables respectively with the number of columns and rows in which splits the simulation scenario (recommended the value 2, 1 with all other variables at the default value)

How to run:

- Start the Java Maven project (recommend
- ed the usage of NetBeans)
    - At the start the maven will display a sort of menu:
      Press 0+Enter to start the EXPLoRa_C_py algorithm
      Press 1+Enter to start the EXPLoRa_C_py_functional algorithm
      Press 2+Enter to start the EXPLoRa_CD_py
      Press 3+Enter to start the EXPLoRa_CD_geoDivision_py
      Press 4+Enter to start the EXPLoRa_CD_multiexecution_py
      Press 5+Enter to start the EXPLoRa_CD_geoDivision_splitVision_py
      Press 6+Enter to start the EXPLoRa_CD_splitVision_py
      Press 22+Enter to start the time performance comparation between EXPLoRa-C functional and multithread
      Press 222+Enter to open the gateway for the python interaction
- Digit 222 and press Enter
    - This will start the gateway used as cross language channel
- In an environment with python type the command line "python .\simple_deployment.py"

Files description:

- In the file **simple_deployment.py** there is the code to launch with python scripts the PowerShell commands, this file has been created to keep track of all the parameters needed to run Lorasim for the execution of a simulation. In the file are listed all the parameters with attached descriptions
- The file **loraDirMulBSfading_uniformSF_collSF_v9_5.py** contains the main Lorasim code divided into the following phases:
    - Scenario generation with attached objects (threads) for each Node
    - Execution of the ADR algorithm for the calculation of the parameters to be given to each object (Node)
    - Execution of the simulation, launching in parallel one thread for every Node and collecting eventual collisions of the packets transmitted from every single Node
- The **Java Maven Project** has the majority of functionalities in the *Main.java class*. In particular in this class there are several functions that are detailed in the following:
    - *public static void main()* : The main function is the first function to be performed at the start of the project, it contains the code to choose which operation you want to perform. In general it gives the possibility to execute the single different algorithms of EXPLoRa-C or to open the gateway for the communication with python Lorasim
    - *public static int[] EXPLoRa_C_py()* : EXPLoRa-C algorithm implemented exactly as described in the paper "Capture Aware Sequential Waterfilling for LoraWAN Adaptive Data Rate"
    - *public static int[] EXPLoRa_C_py_functional()* : EXPLoRa-C algorithm modified to fit into a quasi-functional view using the Java Stream library
    - *public static int[] EXPLoRa_CD_py()* : EXPLoRa-C multithreaded algorithm with division method based on RSSI
    - *public static int[] EXPLoRa_CD_geoDivision_py()* : EXPLoRa-C multithreaded algorithm with division method based on geometry
    - *public static void EXPLoRa_CD_multiexecution_py()* : EXPLoRa-C multithreaded algorithm with division method based on RSSI. This version is designed for oversized scenarios that do not fit entirely into RAM. In fact, only a subset of gateways and nodes are loaded into RAM. In particular are initially chosen subgroups of gateways and at every iteration are processed a subgroup by filtering from the scene of simulation the nodes with greater RSSI associated to the chosen gateways.
    - *public static int[] EXPLoRa_CD_geoDivision_splitVision_py()* : EXPLoRa-C Distributed algorithm with division method based on geometry
    - *public static int[] EXPLoRa_CD_splitVision_py()* : EXPLoRa-C Distributed algorithm with division method based on RSSI

  All the functions that run the various versions of EXPLoRa-C initially read the simulation scenario from a file created by the Lorasim simulator, for which the path must be indicated.
  The final output value of each of these functions is an array of integers filled with the spreading factors calculated by the algorithm in object, sorted so that the spreading factor at position X of the array is the one associated with the node with ID X.
- The files **node_clustering_on_position.py** and **quadtree.py** contains the code used to divide the basestations and the nodes in disjoint subsets, based on their position in the space.

Execution description:

All the part related to the python Lorasim simulator remains unchanged with respect to the single execution EXPLoRa-C Java.
The **node_clustering_on_position.py** read the simulation scenario form the file generated by Lorasim, produces an intermediate file in the format needed by the functions in quadtree.py for splitting the scenario and calls the splitting function, producing the final file with the assignment of nodes and gateways to subareas.

The filed produced by the **node_clustering_on_position.py,** that are also the files that will take as input the **Java Project** will be stored in the *python/simulation_files/clustered_on_position* directory and are the following :

- *(number of basestations)-basestation-BwGw-(distanceBetweenGW)-BwGN-(distanceBetweenGW)-cluster-position-format.txt*
- *(number of basestations)-basestation-BwGw-(distanceBetweenGW)-BwGN-(distanceBetweenGW)-clustered.txt*
- *(number of nodes)-basestation-BwGw-(distanceBetweenGW)-BwGN-(distanceBetweenGW)-cluster-position-format.txt*
- *(number of basestations)-basestation-BwGw-(distanceBetweenGW)-BwGN-(number of nodes)-clustered.txt*

The Java functions *EXPLoRa_CD_geoDivision_py()* and *EXPLoRa_CD_geoDivision_splitVision_py()* read the split simulation scenario files produced by the **node_clustering_on_position.py** that are slight different from the original ones produced by Lorasim simulator, so the code to read these files is different from the one in the other functions.