Algorithm programming in VISIDIA

How to create an algorithm in VISIDIA?

An algorithm is defined in VISIDIA by a class. To implement a new algorithm, you must therefore create a new class in a new file. There are different way of simulating the execution of an algorithm over a network. VISIDIA let the user create a rewriting rule system thanks to the graphical interface so this documentation won't focus on it. Then, VISIDIA can simulate message passing algorithm (synchronous or not)and agent algorithm. A message passing algorithm can also be simulated over a real network.

Creation of the class

To create tour new class, you have to create a new file. The message passing algorithms are placed in the folder *visidia/algo* and the agents algorithms are placed in the *visidia/agents* folder. If you want to launch your algorithm on a network, put it in the *visidia/algoRMI* folder where the other one are situated. First define the package by the name of the folder (for example visidia.algoRMI)

Then, define your class as *public* and as extending from the class defining the type of your algorithm. If you want to create a classical asynchronous message passing algorithm, your class must *extends* the *Algorithm* class. If you want to use synchronization methods in your algorithm, extend the class *SyncAlgorithm*.

You also have to import the corresponding files where the class you extend is defined. For example, for message passing synchronous algorithm, you must import *visidia.simulation.SyncAlgorithm.*

Your Algorithm	Message passing, asynchronous	Message passing, synchronous	Message passing, asynchronous, over a network	Agent, asynchronous	Agent, synchronous
Package	visidia.algo	visidia.algo.synchronous	visidia.algoRMI	visidia.agents	visidia.agents
The class your algorithm must extend	Algorithm	SyncAlgorithm	AlgorithmDist	Agent	SynchronizedAgent
Import (from visidia.simulation)	Algorithm	SyncAlgorithm	AlgorithmDist	agents.Agent	agents.SynchronizedAgent

Creation of the algorithm

To create your algorithm, you have to overwrite the method *init* of the super class. In fact, you just have to write your code in a method called *init* with the following declaration : public void init().

This method will be called when your algorithm will be launched. It is the heart of your code. You can also create other methods in your file to simplify the code.

Example

```
package visidia.algoRMI;

import visidia.simulation.AlgorithmDist;
import visidia.misc.*;
      /**
       * there are different tools for all kinds of algorithms implemented in the
       * misc package
       */

public class MyClass extends AlgorithmDist {

      private static final long serialVersionUID = 7361958588127066919L;
      /**
       * this field has to be declared for serializable class. As
       * AlgorithmDist implments serializable, this must be done for
       * for all the algorithms distributed over a network
       */

      //variable declaration
      int firstVariable = 1;

      final int secondVariable = 2;

      public void init() {
      //inner code of your algorithm
      }

      public Object myMethod(Arguments Object) {
            //code of the method
            return;
      }
}
```

The VISIDIA API

In order to implement your algorithm, different methods have been created. The methods are specific to the class your algorithm extends. However, Algorithm and distributed algorithm classes have the same function description, they only differ in implementation.

Algorithms

General methods

```
protected final int getArity()
```

This method returns the number of neighbours of the node. The doors of the node are identified by an integer ranging from 1 to the node arity. This is the only way a node can distinguish it's neighbours.

```
protected final int nextDoor()
```

```
protected final int previousDoor()
```

Each node stores the last door by which it sent a message. Those methods get the following and respectively the preceding door.

```
protected final Integer getId()
```

All the nodes have a distinct identifier. However, be careful, this is not always the case in practice.

```
protected final int getNetSize()
```

This function returns the size of the graph. In a distributed algorithm, nodes shouldn't need to have the size of the graph to execute properly. However, it is sometimes necessary for the developer to have such an information.

Communication methods

The base of a distributed algorithm is the communication between the different nodes.

```
protected boolean sendTo(int door, Message msg)
```

The sendTo method is the base of the communication between two nodes. An algorithm can call this method to send a message to a neighbour. The neighbours are distinguished by the outgoing doors.

```
protected void sendAll(Message msg)
```

This method send a message to all neighbours.

```
protected Message receiveFrom(int door)
```

receiveFrom test if there is a message arrived on the specific door and return it. This method wait for a message to come.

```
protected Message receive(Door door)
```

Return the next message incoming on the node. The number of the port from on which the message comes is written in the `Door` object and can be accessed with `Door.getNum()`.

```
protected boolean anyMsg()
```

Test if the node has received a message. You can use this method if you don't want to block your algorithm.

```
protected boolean anyMsgDoor(int door)
```

Test if the node has received a message on the specific door.

```
protected boolean sendToNext(Message msg)
```

This method can only be used in oriented circles and send a message to the next node.

```
protected Message receiveFromPrevious()
```

As above, this method is specific to the oriented circles and get a message from the previous node.

```
protected Message receiveFrom(int door, MessageCriterion mc)
```

This function use the misc package as the type message criterion is defined there. This function allows to receive a message from a defined door that matches the criterion. There are different kinds of criterion, testing on the type of the message. To use them, create a criterion variable of the desired type : IntegerMessageCriterion, StringMessageCriterion, ArrowMessageCriterion, SyncMessageCriterion. This will only get the message of the specified type.

```
protected Message receive(Door door, MessageCriterion mc)
```

Return the next message incoming on the node matching the specific criterion. The number of the port from on which the message comes is written in the `Door` object and can be accessed with `Door.getNum()`.

Whiteboard methods

The whiteboards are places where informations are stored on the nodes or on the agents. Here we focus on node whiteboards. As far as the algorithm execution is concerned, there are no differences between storing a value in the whiteboard and declaring a variable for your algorithm. However, statistics can only be used on whiteboard's variable.

```
protected void putProperty(String key, Object value)
protected Object getProperty(String key)
```

Those functions are the accessors to the whiteboard.

Visualisation methods

Visidia enable a visualisation of the algorithms execution. However, you must include in your algorithm the way you want the execution to be visualized.

```
protected final void setDoorState(EdgeState st, int door)
```

This function will change the state of the door and therefore the colour of the edge. The parameter `st` of `EdgeState` type can be created by using `MarkedState(boolean b).` If the parameter b of the constructor is true, your edge will be marked, if it is false, you will remove the mark on the edge.

```
protected final void setEdgeColor(int door, EdgeColor color)
```

This function sets the colour of the outgoing edge to a specific colour. The parameter `color` of `EdgeState` type can be created by using `ColorState(Color c).` The `Color` type is defined if the file java.awt.Color of the java API.

Synchronised Algorithms

As the synchronous algorithms extend the asynchronous ones, you can use the methods previously described. However, synchronisation will cause differences in the communication management. Therefore, you shall not use the communication methods of the asynchronous algorithms.

Synchronisation methods

Synchronised algorithms function with a notion of pulse (or round). All the nodes will begin a new pulse at the same time. To do so, algorithms must call the method `void nextPulse().` This method will wait until all the algorithms have called this method and then return. If one algorithm does not call this method, the simulation will be blocked.

```
protected final int getPulse()
```

Each pulse have an integer identifier. You can check the identifier of the current pulse by a call to this method.

Communication methods

The synchronous communication methods work similarly to the asynchronous ones. However, the criterion used imply a verification on the current pulse.

```
protected boolean sendTo(int door, Message msg)
```

This method just add the current pulse to the message content and use the algorithm method.

```
protected void sendAll(Message msg)
```

Same remark as above.

```
protected final Message getNextMessage(DoorPulseCriterion dpc)
```

This method gets the next message that matches the `DoorPulseCriterion`. To create a `DoorPulseCriterion`, you can call the constructor with one of the following arguments : the door on which you want to receive the message, and/or the pulse identifier.

```
protected final boolean existMessage(DoorPulseCriterion dpc)
```

Return true if there exist a message matching the `DoorPulseCriterion` in the receive queue of the node.

```
protected final boolean anyMsg()
```

Return true if the node has received any message sent in the previous pulse.

```
protected final boolean anyMsgDoor(int door)
```

Return true if the node has received any message sent in the previous pulse and on the specified door.

```
protected final boolean anyMsgPulse(int pulse)
```

Return true if the node has received any message sent in the specified pulse.

```
protected final boolean anyMsgDoorPulse(int door, int pulse)
```

Return true if the node has received any message sent in the specified pulse on the specified door.

```
protected final Message receiveWait(Door door)
```

The node will wait until a message arrive. If no message arrived in the previous pulse, a nextPulse() call is done. The number of the port from on which the message comes is written in the `Door` object and can be accessed with `Door.getNum()`

```
protected final Message receiveWait()
```

The node will wait until a message arrive. If no message arrived in the previous pulse, a nextPulse() call is done. No information on the door is stored.

```
protected final Message receiveWait(int door)
```

The node will wait until a message arrive on the specified door. If no message arrived in the previous pulse, a nextPulse() call is done.

```
protected final Message receive(Door door)
```

Return the first message arrived in the previous pulse and write the door number in the `Door` object.

```
protected final Message receive(Door door, int pulse)
```

Return the first message arrived in the specified pulse and write the door number in the `Door` object.

```
protected final Message receive(int door)
```

Return the first message arrived in the previous pulse on the specified door.

```
protected final Message receive(int door, int pulse)
```

Return the first message arrived in the specified pulse on the specified door.

```
protected final void purge()
```

Delete all received messages. There is no way to recover those messages after this method execution. For performances reasons, it is recommended to delete messages that you are sure not to use.

Mobile Agents

Agents are autonomous calculator entity that moves in the network, using resources of the nodes when they arrive on them. They have a personal whiteboard and can use a whiteboard placed on the nodes to communicate. When a simulation uses agents, it cannot use messages. Agents can use different tools in order to simplify the code. First, they can use an agent mover that will automatize it's move. Then, you can implement an agent chooser that will allow you to place randomly your agent on the graph with your defined repartition.

Synchronous agents only have a slightly different API as there is no real communication between them. Therefore, they only have synchronisation methods that add to the classical ones.

General methods

```
public int getIdentity()
```

Return the identity of the vertex. This identity is unique in the graph.

```
public int getVertexIdentity()
```

Return the vertex, on which the agent is, unique identity.

```
public Collection agentsOnVertex()
```

Return a collection of the agents that are currently on the vertex.

```
public int getArity()
```

Return the number of neighbours of the node.

`public int getNetSize()`

Return the size of the graph. This function is available but remember that a node is rarely able to easily

`protected void sleep(long millis)`

Dis activate the node for a given amount of time. The value is given in milliseconds.

know it.

`public void cloneAgent()`

Create a new agent of the same type on the vertex. This agent will launch it's `init()` method.

`public void cloneAndSend(int door)`

Create a new agent of the same type and launch it on the specified door. This agent will launch it's `init()` method after it's move.

`public void createAgent(Class agClass)`

Create a new agent of the specified class on the vertex. This agent will launch it's `init()` method.

`public void createAgentAndSend(Class agClass, int door)`

Create a new agent of the specified class and launch it on the specified door. This agent will launch it's `init()` method after it's move.

Movement methods

To have the agent move, there is to different way of proceeding. You can force each move for the agent or use an agent mover. You can always force the move, even if the classical move uses an agent mover.

`public void moveToDoor(int door)`

Move the agent to the specified door.

`public int entryDoor()`

Return the door by which the agent came.

`public void moveBack()`

Move the agent back toward the door by which the agent came.

```
public void move()
```

Once you have defined an agent mover, you only have to use this method.

```
public void setAgentMover(String agentMoverClassName)
```

This method is the basic way to use a define agent mover. You only have to give the class name of the agent mover. There are different agent movers already defined in VISIDIA that you can use : LinearAgentMover (move will go to a non-visited vertex if possible), NoBackMover (LinearAgentMover that won't move backward if possible), RandomAgentMover (move to a random neighbour) and RandomWalk (move to a random neighbour half of the time or stay in place).

```
public void setAgentMover(AgentMover am)
```

This method set an agent mover for the agent by using it's class and not it's class name.

```
public AgentMover getAgentMover()
```

Return the class of the agent's agent mover.

Whiteboards methods

As we said before, there is two different whiteboards that the agent can use. There is therefore different accessors for the agent whiteboards and for the vertex one.

```
public Object getProperty(Object key)
```

Return the object associated with the `key` of the agent whiteboard. See the java documentation on hash table for further details.

```
public void setProperty(Object key, Object value)
```

Affect the object `value` to the `key`.

```
public Set getPropertyKeys()
```

Return the keys of the agent whiteboard.

```
public void setWhiteBoard(WhiteBoard wb)
```

Affect an existing whiteboard to the agent.

```
public void setWhiteBoard(Hashtable<Object,Object> defaults)
```

Affect a new whiteboard to the agent with default values.

`public WhiteBoard getWhiteBoard()`

Return the whiteboard of the agent.

`public Object getVertexProperty(Object key)`

Return the object associated with the `key` of the vertex whiteboard.

`public void setVertexProperty(Object key, Object value)`

Affect the object `value` to the `key`.

`public Set getVertexPropertyKeys()`

Return the keys of the vertex whiteboard.

`public void lockVertexProperties()`

As multiple agent can change vertex whiteboard, you must always assure that two modifications at the same time won't happen. This function puts a lock on the vertex whiteboard. You will return from this function with the assurance that no other agent will be able to lock the same vertex. If the vertex is already locked, this function waits.

`public void unlockVertexProperties()`

Remove your lock on the vertex whiteboard.

`public boolean vertexPropertiesLocked()`

Return true if the vertex is locked.

`public boolean lockVertexIfPossible()`

If the vertex is already locked, return false and does nothing, else return true and lock the vertex.

`public Agent getVertexPropertiesOwner()`

Return the agent locking the vertex or null if the vertex isn't locked.

Synchronisation method

This method can only be used in a synchronised agent.

`public void nextPulse()`

This method will call for a new pulse. An agent is blocked into this method until all agent have finished and called the `nextPulse()` method as well.

Agent Mover

There already exists defined agent mover but you can code a new one by following the API.

To implement an agent mover in order to use the method `move()` in your agent, you have to overwrite the method that choose the next door to go : `public abstract int findNextDoor() throws MoveException`. This has to be done by creating a new java file in visidia/agents/agentsmover. Then you only have to implement the way your mover will choose the next door to go.