The Giving Game: Documentation

The Giving Game

f

University of Amsterdam f

Julian Ruger

Contents

1. Introduction

In this document the usage of the simulator for the giving game will explained and the code will be documented. The usage of most variables will be explained inside the code itself. This document will focus on how each function is used and specific algorithms will be explained.

2. Instructions

Installation

requirements

Usage

Images to explain each button and graph etc. Python 3.4+

3. Code

This program is written with python. The code consists mostly of object oriented python code.

Back-end

The back end consists of the following files:
The usage of each function in each file and the most important variables will be explained.

Agent.py

The Agent class consists of multiple functions to update information about each agent. Each agent knows only about the things the agent is part of. For example each agent only knows about its own transactions. Each agent has an unique id which can be used as a index for some lists. The index of these lists is the same as the id to make navigating through these lists easy.
The following python modules are used:

- Numpy
- random

```
def give(self, receiving_agent, good):
```

Updates the transaction information of a certain agent after a good has been given.

Parameters:

receiving_agent: An *Agent* object that is receiving the good
good: A *Good* object that has been received by the receiving_agent

```
def receive(self, giving_agent, good):
```

Updates the transaction information of a certain agent after a good has been received.

Parameters:

giving_agent: An *Agent* object that is giving the good
good: A *Good* object that has been given by the giving_agent

Goods.py

The Good class consists of variables that determine what kind of good it is. Each good has an unique id, which can be used as the index of some lists. The index of these lists is the same as the id to make navigating through these lists easy.
For this class no modules are used.

Enviroment.py

The Enviroment class consists of multiple functions to create the enviroment of the giving game, to update variables after each transaction and to calculate the community effect. Most of the variables are stored in this class.
The following python modules are used:

- Numpy

```
def create_agents(self, N):
```

Creates N number of agents. The agents are stored in a list, *agents_list*.

Parameters:

N: An integer that defines the number of agents.

```
def create_goods(self, M, goods):
```

Creates M number of goods. The goods are stored in a list, *goods_list*.

Parameters:

M: An integer that defines the number of goods.

goods: A 2-dimensional list. Each entry consists of three integer values that define the type of the good.

```
def notify_agents(self):
```

Notifies the agents of the created goods.

```
def update_balancematrix(self, P, Q):
```

Updates the balance between P and Q in the balance matrix.

Parameters:

P: An *Agent* object that is giving.

Q: An *Agent* object that is receiving.

```
def notify_producer(self, good):
```

Notifies each producer of the life of the good.

Parameters:

good: An *Good* object.

```
produce_goods(self, output):
```

Goods are produced if the time until production reached zero, else it decreases the time until production by one.

Parameters:

output: An *Output* object used to update the GUI after a production.

```
def transaction(self, P, Q, good):
```

P gives a good to Q.

Parameters:

P: An *Agent* object that is giving.

Q: An *Agent* object that is receiving.

good: A *Good* object that is being given.

```
def select_agent(self, selectionrule, current_agent):
```

Select an agent using the selectionrule to receive a good.

Parameters:

selectionrule: An integer that defines which selection rule needs to be used. 0 is the random rule, 1 is the balance rule, 2 is the goodwill rule.

current_agent: An *Agent* object that is currently holding a good.

Returns: An *Agent* object

```
def select_start_agents(self)
```

Select randomly the agents to receive the goods at the start of the simulation. These agents also will be the producers if the goods are perishable.

```
def calculate_good_transaction_percentages(self, total_transactions):
```

Calculates the percentage of how many times each agent has given and received a specific good.

Parameters:

total_transactions: An integer, the total number of transactions that have taken place.

```
def calculate_communityeffect(self, goods_transactions, subgroup_size):
```

Calculates the percentage of transactions that take place in a subgroup with a specific size.

Parameters:

goods_transactions: A list that contains the transaction percentage of each good.

subgroup_size: An integer that defines the size of a subgroup.

Simulate.py

This file does not use any object oriented code. This file consists of multiple functions that call the necessary functions from other files like Enviroment.py. This file can be seen as the engine for the simulation. The creation of the enviroment, transactions and updating the UI take place in this file.
The following python modules are used:

- Numpy

```
def create_enviroment(N, M, goods_list, M_perishable, perish_period,
                      production_delay, nominal_value, parallel, selectionrule):
```

Creates the enviroment with all the agents and goods.

Parameters:

N: An integer that defines the number of agents

M: An integer that defines the numer of goods

goods_list: An 2-dimensianal list. Each entry consists of three integer values that define the type of the good.

M_perishable: An integer that defines the number goods that are perishable.

perish_period: An integer that defines the time before a good perishes.

production_delay: An integer that defines the time before a good is produced after it has perished.

nominal_value: An integer that defines the nominal value of the goods

parallel: An boolean that defines if the transaction should be executed parallel or one by one. True is parallel, False is one by one.

selectionrule: An integer that defines which selection rule needs to be used. 0 is the random rule, 1 is the balance rule, 2 is the goodwill rule.

```
def start_simulation(output, env, selectionrule): def simulate(nr_iterations, env, selectionrule, out
```

Starts the simulation.

Parameters:

output: An *Output* object that is used to update the GUI.

env: An textitEnviroment object.

selectionrule: An integer that defines which selection rule needs to be used. 0 is the random rule, 1 is the balance rule, 2 is the goodwill rule. ??????

```
def continue_simulation(env, selectionrule, output, total_transactions):
```

Continue the simulation after the simulation has been paused.

Parameters:

output: An *Output* object that is used to update the GUI.

env: An textitEnviroment object.

selectionrule: An integer that defines which selection rule needs to be used. 0 is the random rule, 1 is the balance rule, 2 is the goodwill rule. ??????

total_transactions: An integer. The total transactions after the simulation has been paused.

```
def onebyone(env, selectionrule, output, total_transactions):
```

Executes the transactions one by one.

Parameters:

output: An *Output* object that is used to update the GUI.

env: An textitEnviroment object.

selectionrule: An integer that defines which selection rule needs to be used. 0 is the random rule, 1 is the balance rule, 2 is the goodwill rule. ??????

total_transactions: An integer.

```
def parallel(env, selectionrule, output, total_transactions):
```

Executes the transactions parallel.

Parameters:

output: An *Output* object that is used to update the GUI.

env: An textitEnviroment object.

selectionrule: An integer that defines which selection rule needs to be used. 0 is the random rule, 1 is the balance rule, 2 is the goodwill rule. ??????

total_transactions: An integer.

Selection_rules.py

This file does not use any object oriented code. This file consists of the function used to select the next agent. The different algorithms for each selection rule will be explained below.
The following python modules are used:

- Numpy

```
def random_rule(N):
```

The random rule chooses an agent randombly

Parameters:

N: An integer that defines the number of agents.

Returns: An *Agent* object

```
def balance_rule(balance_matrix, current_agent, N):
```

Parameters:

balance_matrix: An 2-dimensional numpy array

current_agent: An *Agent* object.

N: An integer that defines the number of agents.

Returns: An *Agent* object

```
def goowill_rule():
```

Front-end

The front end consists of the following files:
The usage of each function/class in each file will be explained.

GUI_Input.py

The Input class is consists of multiple functions to handle the user input so that the parameters can be used to create the enviroment and start the simulation.
The following python modules are used:

- Numpy
- Qt

```
def setAgents(self, value):
```

GUI_Output.py

This file consists of multiple classes. Each class is used to create a part of the UI. The main class is the Output class.
The following python modules are used:

- Numpy
- Qt

```
class Output(QtGui.QMainWindow):
```

The Output class is the main class. This class creates the main window with options for the user to save data as an csv file.

```
class GUI(QtGui.QWidget):
```

The GUI class is the main layout class. This class creates all the different Qt layouts which can be filled with QWidgets.

```
class Tabs(QtGui.QTabWidget):
```

The Tabs class contains the layouts and widgets for the visualisation of the simulation. It contains the animated visualisation, a list of transactions and a graph containing the transaction percentages.

```
class ControlPanel(QtGui.QWidget):
```

The ControlPanel class contains all the controls to pause, start and delay the simulation.

```
class Results(QtGui.QWidget):
```

The Results class contains all controls to get to the most important data, the yield curve, the number of goods given and received by each agent and the community effect.

```
class AgentInfo(QtGui.QDialog):
```

The AgentInfo class contains functions to plot information about each agent.

```
class YieldCurve(QtGui.QDialog):
```

The YieldCurve class contains funtion to plot the yield curve for each pair of agents and each good.

Visualisation.py

This Canvas class consists of multiple function to create and animate the transaction during the simulation. The following python modules are used:

- Numpy
- VisPy which uses OpenGl

```
VERT_SHADER = """
#version 120
attribute vec3 position;
attribute vec4 color;
attribute float size;

varying vec4 v_color;
void main (void) {
    gl_Position = vec4(position, 1.0);
    v_color = color;
    gl_PointSize = size;
}
"""

FRAG_SHADER = """
#version 120
varying vec4 v_color;
void main()
{
    float x = 2.0*gl_PointCoord.x - 1.0;
    float y = 2.0*gl_PointCoord.y - 1.0;
    float a = 1.0 - (x*x + y*y);
    gl_FragColor = vec4(v_color.rgb, a*v_color.a);
}

"""
```

These shaders are needed for the visualisation of the agents and goods. More information about how shaders work can be found at:

```
def on_initialize(self, event):
```

4. Releases

5. Further research

Requirements

The following Python modules/frameworks are mandatory.

- Qt Framework
- MatplotLib
- Numpy