The Giving Game: Design Document

The Giving Game

April 14, 2015

University of Amsterdam

Julian Ruger 10352783

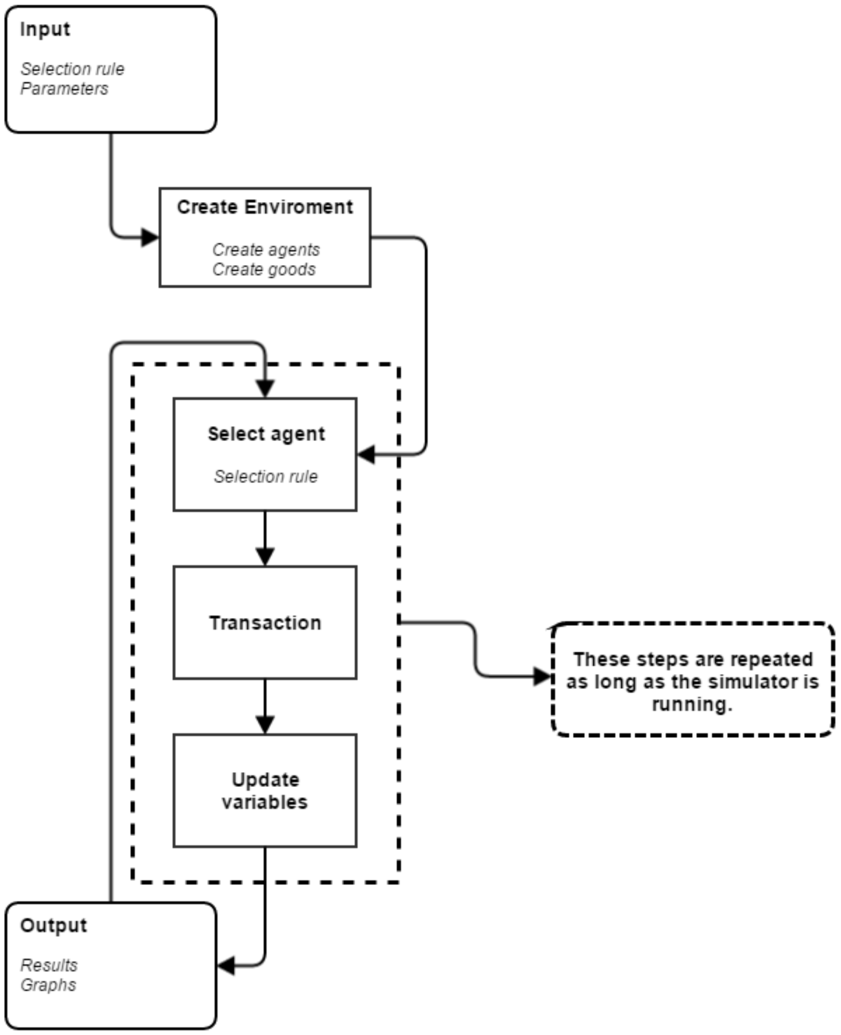
Contents

1. Introduction

I decided to create the simulator with Python(https://www.python.org/). I will be using the newest version of Python, Python 3.4, because this allows me to make use the full potential of Python. Python is a fairly easy programming language. The syntax allows me to write the program in fewer lines of code. With Python I do not have to specify the type of variables and the memory space is automatically allocated and deallocated. Python is also supported by alot of different packages and frameworks which can be used to write the code in even fewer lines. For example I will be using the Qt framework for the GUI. Python will save me alot of time and I only have about six or seven weeks so that is why I have chones Python. The only problem with python I might run into is the performance of the simulator. Because Python automatically allocates the memory space it might be less efficient using Python when I need to use alot of memory space. During the six/seven weeks of programming I will need to pay good attention to this problem to make sure the simulator will not get to slow at executing the tasks.

2. The simulation model

The simulator can be divided into three major parts: the algorithm implementations and the giving game itself (Back-end), the visualisation (results and graphs) and the GUI. The simulator can be seen as the following proces:



The following Input must be given to the simulator in order to get all the results (output):
Input

- Selection rules for example the random rule, the balance rule or goodwill rule.
- Parameters
  - Time, how many transactions should take place. Instead of this parameter I could also implement a start and stop button.
  - N number of agents
  - M number of goods, this parameter is not mandatory, the default number of goods will be 1.
  - The value of the goods and if the value decreases over time.
  - The community effect threshold with values in the range of [0, 1]. 0 meaning there is absolutely no community effect and 1 meaning that the transactions only take place in a subgroup. For example if the threshold would be 0.8 we can conclude that there is a community effect if 80/100 transactions takes place in a subgroup.
  - The above parameter introduces another parameter. We need to determine the size of a subgroup. The size of a subgroup should be in the range of [2, x]. The value of x should be determined during the literature study.

Output

- Results
  - Community effect, depending on the given threshold can the simulator conclude if there is a community effect with a simple yes or no answer.
  - The total number of transactions, balance of every agent, transactions of every agent
  - If a subgroup arose all the information about the agents in this subgroup should be given.

  Community effect, number of transactions etc.
- Graphs
  - For the community effect there should be a graph that shows the community effect over time.
  - For every agant pair and good a graph should be given if the user asks for it. This might be difficult to implement, because alot of information about the agents and goods must be stored into the memory. (Yield curve)
  - For every graph the corresponding function must be given.

During the development of the simulator I must take the following into account:

- Multiple selection rules must already be implemented and if a new selection rule or parameter is introduced it should not be difficult to add this to the existing simulator.
- Every agent should keep track of all his transactions.
- The simulator must be able to handle a large amount of agents ranging from 1000 to a maximum of 10000 agents.
- The results should be shown while the simulation is still running. If this will take too much time I could choose to only show the results if the user presses the stop/pause button.

3. Back-end

Important packages: Numpy
For every pair of agents (P, Q) we must keep track of the following things: The balance perceived by P and Q, How many times P and Q have given a good and received a good. For every single agent we must keep track of the following things: A list of all its transactions and the position of the agent (assuming an array of some sort will be used to store all the agents). For each good A and each par of agents (P, Q) we must keep track of the like factors of each good A rangning from [-1, 0] and the value of each good A rangning from $[0, \infty)$. To accomplish the above the following design decisions have been made:

- For the balance of each pair of agents (P, Q) a NxN matrix, where N is the total number of agents, will be used globally to store all the balance values of each pair.
- Each agent will have to store the following variables.
  - The position of the agent. For example this position is used in the NxN matrix explained above. This position will be an Integer ranging from [0, N] where N is the total number of agents.
  - A list of all its previous transactions. The transactions will be stored in a Python *list*.
  - A list of the number of goods given and received for every other agent. A Numpy array will be used to store these variables. The index of the array will be the indication for each agent.
- Each good will have to store the following variables.
  - The value of the good. This value is in the range of $[0, \infty)$.
  - If the value of the good decreases over time or not. A simple True or False will be used to determine this.
- For each good A a Python *list* will be used globally to store each good. The size of this list will be determined by the total number of goods.
- For each good A and each agent P a MxN matrix, where M is the total number of goods and N the total number of agents, will be used globally to store the like factor of every good perceived by every agent.

The pseudocode below shows the decisions made above. I will use an object oriented structure to accomplish the above decisions.
Agents

```
from numpy import *

class Agent:
    def __init__(self, position, N):
        self.position = position
        # List of transactions, for example: [(Action, Agent, Good)]
        # Where Action is either given or received, Agent is the agent
        # on the other side of the transaction and Good is de good that
        # has been transfered.
        self.listoftransactions = []
        # The number of goods given and goods received for each agent
        # are stored in the variable below. For example: array([(given, received)])
        # where given is the total number of goods given to the agent
        # with the position equal to the index and received the total number
        # of goods received from the agent.
        self.given_received = array([])
    def update_listoftransactions(self, new_transaction):
        self.listoftransactions.append(new_transaction)

    def give(self, receiving_agent, good):
        #The current agent gives to the receiving_agent.
        pass

    def receive(self, giving_agent, good):
        #The current agent receives from the giving_agent
        pass

    def update_given_received(self, position, previous_transaction):
        if previous_transaction == given:
            self.given_received[position][0] += previous_transaction
        elif previous_transaction == received:
            self.given_received[position][1] += previous_transaction
```

Goods

```
class Goods():
    def __init__(self, value, decreasing)
        self.value = value # The value of the good
        self.decreasing = decrasing # Either True or False

    def decrease(self, decreasing_factor):
        self.value = self.value * decreasing_factor
```

Creating the Giving Game enviroment

```
import Agents
import Goods
# A different file should contain all the functions for the selectio rules.
# Each selection rule will have one function, this function is called before
# every transaction.
import Selectionrules

def create_agents(N):
     # Create N agents by calling the __init__() from the Agents Class
     pass

def create_goods(M):
     # Create M product by calling the __init_() from the Goods Class
     pass

def transaction(P, Q):
     # Start a transaction between P and Q by calling give() and receive()
     # from the Agents Class
     # P give to Q
     P.give(Q)
     # Q reveives from P
     Q.receive(P)

def update_balancematrix(balancematrix):
     # Update the balance matrix after every transaction
     pass

def select_agent(selectionrule):
     # Call the right seletion rule to select the agent
     pass

def main():
     create_agents(N)
     create_goods(M)
     # Start the game
     P = current_agent #Choose a random agent
     While(Start):
          Q = select_agent(selectionrule)
          transaction(P, Q, good)
          P = Q

if __name__ == '__main__':
    main()
```

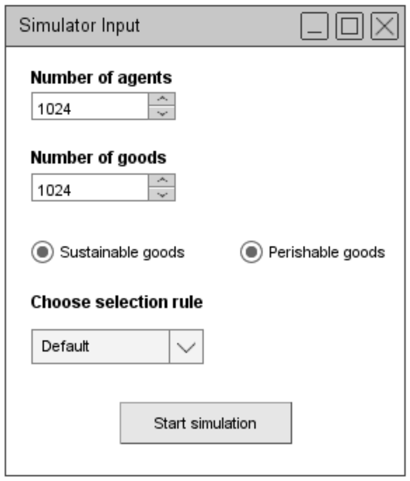
4. Visualisation

Important packages: Numpy, matplotlib, PyChart
With Either *matplotlib* or *PyChart* I can plot graphs. I will start with *matplotlib*, because this package is better documented and can be used together with the Qt framework which I will be using for the GUI. Together with *Numpy* I will be able to do all the necessary calculations and produce together with the Qt

framework well formatted results.

5. GUI

Important packages: VisPy Important framwork: Qt For the GUI I will be using the Qt framework. The GUI for the input of a few parameters will look something like this:



Simulator Input

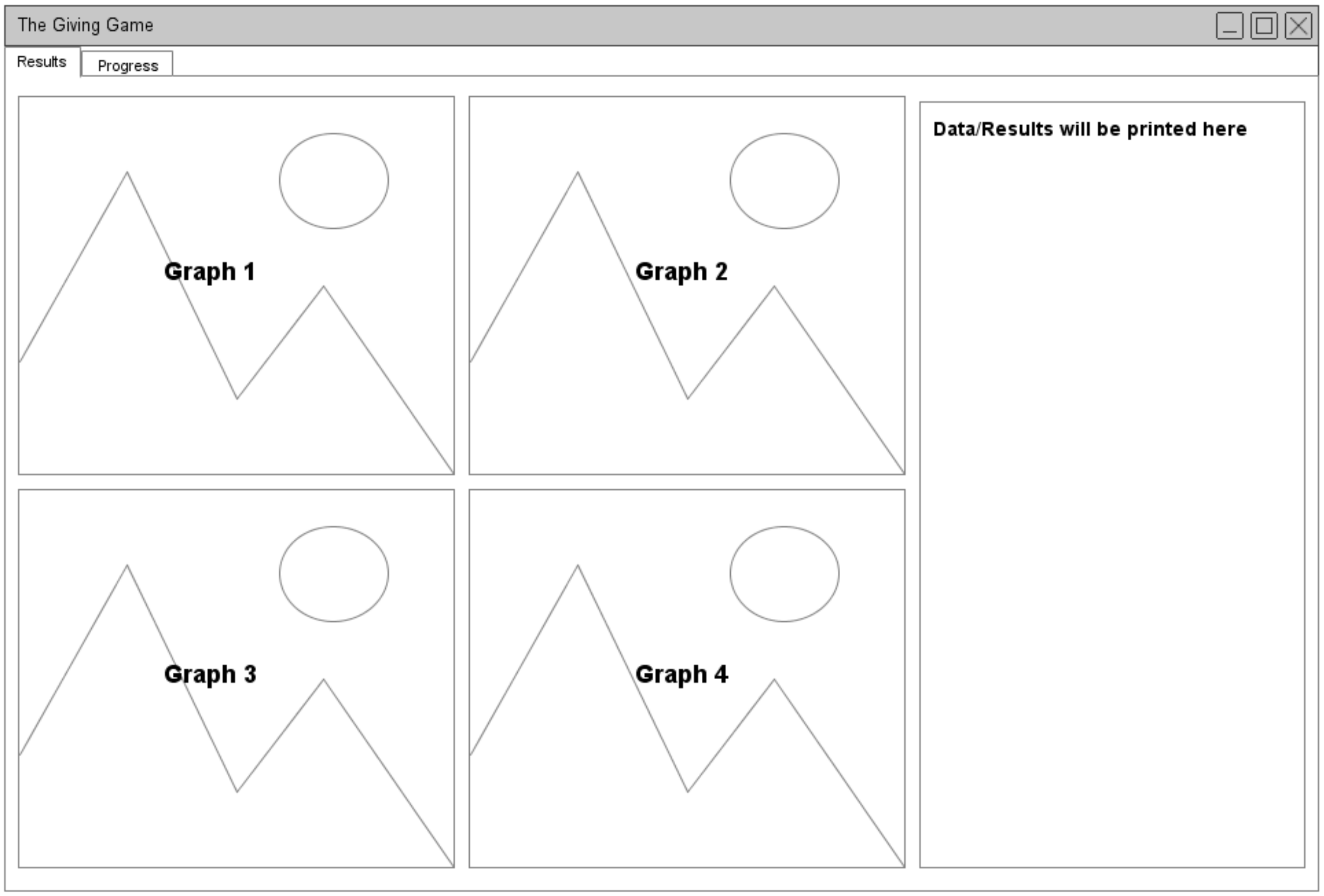
Number of agents: 1024

Number of goods: 1024

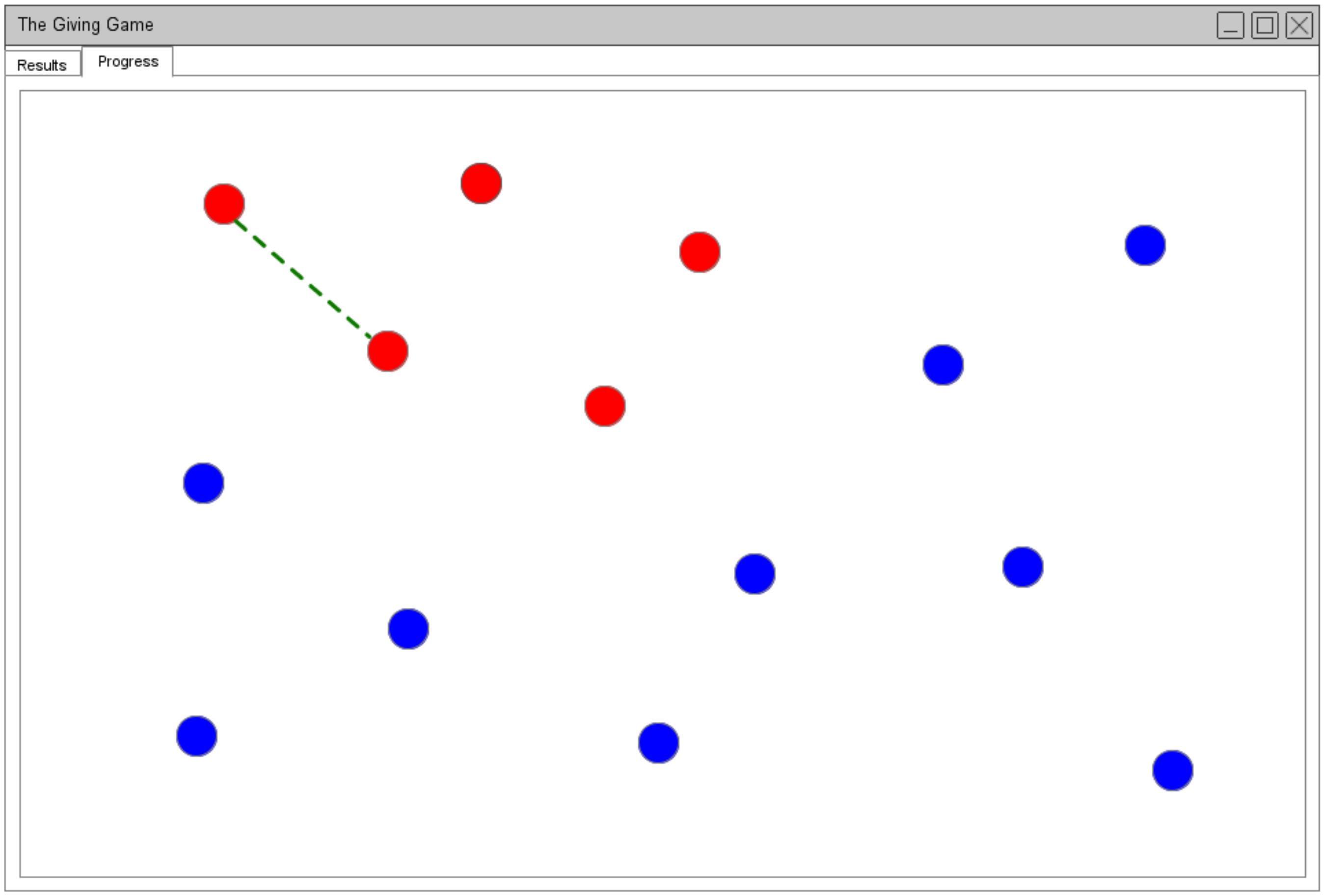
● Sustainable goods ● Perishable goods

Choose selection rule: Default

The results will look something like this:



The second tab will be used to show the following:



Here the user can see the progress of the giving game. The green dotted line shows a transaction. The color shows the subgroups. For example red could mean that the transactions take most of the time place in this subgroup and blue would mean the opposite. This visualisation of the progress will be implemented last. For this part of the visualisation I will be needing *OpenGL*. To accomplish this I will be making use of the package *VisPy*.

6. Possible additions

- I could add a variable for the most traded goods and base the results more on the goods to see how different kind of goods affect the results.

7. Changes and Updates

Goods

- The perish factor is now called the perish period because it is a natural number in the interval $[0, \infty$). These values are more logical if we think of time
- The production time is now called the production delay. This delay is a natural number in the interval $[0, \infty)$. The production delay starts when the good has perished.