NARRATIVE AGENTS AS A REPORTING MECHANISM FOR AGENT-BASED MODELS

by

Brent D. Auble

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Arts
Interdisciplinary Studies

Committee:

_____ Director

_____

_____

_____ Program Director

_____ Dean, College of Humanities and Social Sciences

Date: _____ Spring Semester 2015
George Mason University
Fairfax, VA

Narrative Agents as a Reporting Mechanism for Agent-Based Models

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Arts in Interdisciplinary Studies at George Mason University

By

Brent D. Auble
Bachelor of Science
Lafayette College, 2004

Director: Andrew Crooks, Assistant Professor & Director of Graduate Studies;
Department of Computational Social Science

Spring Semester 2015
George Mason University
Fairfax, VA

Dedication

To my wonderful wife Pam for putting up with my endless schooling, and to her and my family for all of their love and support.

Acknowledgements

To Dr. Andrew Crooks, thank you for supporting me through this process and prodding me along when I needed it.

Dr. Robert Axtell, thank you for mentioning the idea of narrative agents to me initially and then encouraging me to pursue the development of this approach.

Dr. William Kennedy, thank you for being a wonderful sounding board and excellent professor.

LMI, for its ongoing support of my academic pursuits. Over the years, my previous employers and clients have been unfailingly supportive of me pursuing my studies in computational social science, which I have greatly appreciated.

Table of Contents

List of Tables

List of Figures

Abstract

NARRATIVE AGENTS AS A REPORTING MECHANISM FOR AGENT-BASED MODELS

Brent D. Auble, MAIS

George Mason University, 2015

Thesis Director: Dr. Andrew Crooks

Agent-Based Modeling (ABM) is an approach for building computer models of social situations where computer agents interact with each other within a computer-generated environment. The agents have limited information and the environment can change, simulating complex situations, and interactions between agents and the environment can result in unexpected "emergent" behaviors. One value of ABMs is that they allow for collection of all details of the characteristics and behavior over time of every individual agent and the environment, theoretically enabling the analysis of micro-level interactions between individuals and within small groups. In practice, however, the volume of raw data generated by each run of a model (thousands of which might be done to test a range of parameters) makes it difficult to identify unusual interactions, and analysis of models ends up being done by aggregating data and reporting overall trends.

This thesis explores using computer-generated narratives describing the behavior of an individual agent as an alternate method for evaluating the micro-level behavior of an ABM. In addition, approaches are demonstrated for identifying the agents whose narratives are most worth the time to read.

Introduction

A good story is worth a thousand data points. Agent-Based Modeling (ABM) suffers from a data problem – too much data. Unlike most scientists, agent-based modelers have complete information about the behavior they study, which can easily lead to data overload. As a result, they tend to report their data by aggregating the outcomes of individual behaviors and using typical measures developed for messy, real-world data. For example, an ABM of a country's economy may report a "Gross Domestic Product" (GDP), which readily allows for comparison to the real country. Applying quantitative approaches to the detailed data in the model could still result in a lot of analysis that is difficult for humans to understand without ready comparison to commonly used real-world measures. One solution to this challenge is to have the computer model generate a narrative describing the behavior of individual agents or groups of agents. The number of narratives generated may be large, but people are readily able to understand stories.

In this thesis, I discuss how using computer-generated narratives increase the value and understandability of agent-based models. I also describe and demonstrate several techniques for generating narratives.

Agent-based modeling is an approach for understanding social and biological systems through the use of computer simulations that explicitly model the behavior of individuals ("agents"). Because an ABM is entirely self-contained and computer-generated, it – theoretically – has the ability to capture and report every action of every individual at every time step in the model. All of the information the agent used to make a decision about its behavior is also available. This should allow for a great deal of detailed analysis of an ABM's behavior. The reality, however, is that type of detailed analysis rarely happens. For example, in the March 2015 issue of the Journal of Artificial Societies and Social Simulation[1], there were 18 peer reviewed articles, only one of which (Kirke & Miranda, 2015) performed analysis of the behavior of individual agents. There are a number of reasons why individual agent level analysis of ABMs is rare. First, in many of fields, the expectation is for there to be aggregate-level analysis, and any other approach is difficult to get accepted. Second, a lot of data is generated by each run of a model, which is difficult to analyze. Third, each model is typically run many hundreds, thousands, or millions of times in order to study a range of possible behaviors and outcomes. Fourth, tools and methodologies for analyzing low-level individual behavior for large numbers of individuals do not exist.

The value of a narrative generated from the behavior of agents within the model is that it gets into the rich micro-level behavior of the model in a form that is naturally attractive to

[1] Journal of Artificial Societies and Social Simulation, Volume 18, Issue 2, March 2015, http://jasss.soc.surrey.ac.uk/18/2/contents.html

humans: we like stories. A researcher can examine the narratives to find "interesting" stories, which may lead to a deeper quantitative analysis of specific data. The stories may also allow the researcher to identify flaws in the design or implementation of the model.

In this thesis I develop, describe, and provide examples of methodologies for computationally (without human intervention) generating narratives about the behavior of an ABM. There are at least four possible approaches for narrative generation:

1. an individual agent with actions that are independent of their previous actions (i.e. they have no memory);
2. an individual agent with memory of its past actions that influence later behavior;
3. an omniscient "journalist" reporting on logical groups of agents (e.g. by geography, industry); and
4. an overall narrative about the behavior of the model, describing model-wide trends and possibly the actions of a few notable agents.

For this thesis, I develop and explore the first approach, which was programmed using the Zero-Intelligence Traders model (Gode & Sunder, 1993, 1997) as the basis. This model is well known within the fields of computational social science and agent-based computational economics[2].

[2] A Google Scholar search on 4/25/2015 for the Gode & Sunder (1993) article listed 1351 citing articles, 629 of which included the terms "agent-based model".

The remainder of the thesis is structured as follows. First, a review of the literature related to narrative and reporting on ABMs is presented to set the scene and research contribution for the thesis. Second, a description of the Zero-Intelligence Traders model, how it behaves, how it was built, how the narrative is generated, and how the model was tested. Third, a description and examples of the model's narrative and data outputs is presented. Fourth, discussion of model narrative generally, storage and analysis implications, and when it is worth the effort to add narrative generation to a model. Fifth, a discussion of potential future avenues of research on narrative generation. Finally, a summary of the work on this thesis.

Literature Review

The idea of using narrative to describe the behavior of an agent-based model is not new. Axelrod (1997) made the point that ABMs typically deal with situations that are "path-dependent, meaning that history matters." Further, that describing the history of the model is an obvious way to analyze a model. He discussed three approaches to describing a model's history. First is as a chronological series of events. Second is that the "history can be told from the point of view of a single actor." And that "this is often the easiest kind of history to understand and can be very revealing about the ways in which the model's mechanisms have their effects over time." Finally that the "history can be told from a global point of view," such as describing large-scale patterns like the distribution of wealth over time.

After carrying out an extensive literature review, there appears to be very little published literature describing the use of agent-generated narrative as a means of reporting on the behavior of the model. Millington et al. (2012) used narratives to describe the behavior of breeding behavior in bird colonies, but the researchers generate the narratives themselves after the completion of model runs. They do not have the computer model or the agents generate the narratives programmatically. However, their working definition of narrative is useful: "a narrative (text) is constructed from the modeler's understanding

(discourse) which is in turn derived from sequences of modeled events (the fabula)." In this case, the "modeler's understanding" is programmed into the model prior to the operation of the model and the generation of sequences of modeled events. Other examples of model narratives can be found in Rand (2005) and Cioffi-Revilla et al. (2013), but in each case, human intervention was required to produce the narrative at the completion of the model run.

A review of several general books on ABM and social simulation (Gilbert 2008, Gilbert & Troitzsch, 2005; North & Macal, 2007) does not reveal any discussion of agent narrative. North & Macal devote an entire chapter to "Understanding and Presenting ABMS Results," but it does not include any discussion of narrative. Two recently published introductions to ABM (Railsback & Grimm, 2011; Wilensky & Rand, 2015) have no mention of narratives.

Work has been done on having agents generate language as the purpose of the model (Dautenhahn & Coles, 2001) and as a means of communication between agents (Guerin & Pitt 2000). Neither of these approaches are directly relevant to having agents generate narrative in order to describe the behavior of the agents or the model.

The field of computer gaming offers some inspiration. Madden (2009) presents an approach for generating narrative describing the actions of players (both human- and computer-controlled) in a massively-multiplayer online role playing game. In order to do

this, he created a computer-controlled agent ("non-player character" or NPC) that was visible to the people playing the game, allowing the players to know they were being observed for reporting purposes (and potentially causing them to modify their behavior). The observer NPC generated narrative description in real time of the current actions of the players and their interactions (friendly or combat) with other players and computer-controlled NPCs. These descriptions were available immediately to the players being observed and potentially any other people playing the game. Madden's (2009) methodology is useful, although there are sufficient differences that it cannot be adopted directly for ABMs. First, because he is generating narratives describing the actions of human players, there is no ability to describe "why" the behavior happened – with agents, the why is known. Second, his observer agent likely affects the behavior of the human players. Finally, the observer is constrained to describing current actions, and does not generate narrative of behavior over time nor the aggregate behavior of groups of agents.

From personal discussions, I know of work done by a collaborator of Robert Axtell in a large-scale economic model using journalist agents to report on agent activities, so the concept certainly is not novel. However, this specific work has not yet been published, and there is nothing else specifically describing computer-generated narrative about agents in an ABM.

Another way of thinking about narrative is as a method for designing and communicating a model. There have been discussions of the visual aspects of ABMs. For example

Grimm (2002) described the value of using model visualization as a method for both debugging a model, and for communicating its behavior to a wider audience. Kornhauser et al. (2009) give detailed guidelines for the visual design of ABMs. There is not yet a similar set of guidelines for using narrative to describe an ABM, but the work of Millington et al. (2012), and, hopefully, this thesis provide a start.

Approach

In order to explore various approaches to having agents generate narrative, I modified an existing agent-based model to incorporate narrative. The model is the Zero-Intelligence (ZI) Traders model (Gode & Sunder 1993, 1997), and is based on a version of the model developed in Java by Robert Axtell.

Zero-Intelligence Traders Model Overview

One of the classic agent-based models is the Zero-Intelligence (ZI) Traders model of Gode and Sunder (1993, 1997). What makes it a classic is that it is both technically simple (agents do no reasoning and only have one behavior – attempting to make a trade) and it shows that a core tenet of economic theory can be explained without recourse to complex economic structures, such as a Walrasian tâtonnement (Walras, 1926).

ZI Traders has simple buyer and seller agents with their own individual maximum purchase price and minimum sale price (respectively), randomly paired up and randomly selecting a price below or above those prices (respectively) and only trading if they can meet somewhere between those prices. If a trade is possible, the buyer and seller trade at a random price in the middle of their offers and consequently drop out of the trading

pool. Figure 1 shows a buyer with a maximum purchase price of 70 making an offer of 60, and a seller with a minimum sale price of 30 making an offer of 40. Because the seller's offer was lower than the buyer's offer a trade can occur, with the actual trade price being randomly determined between the two offers – in this case 55.

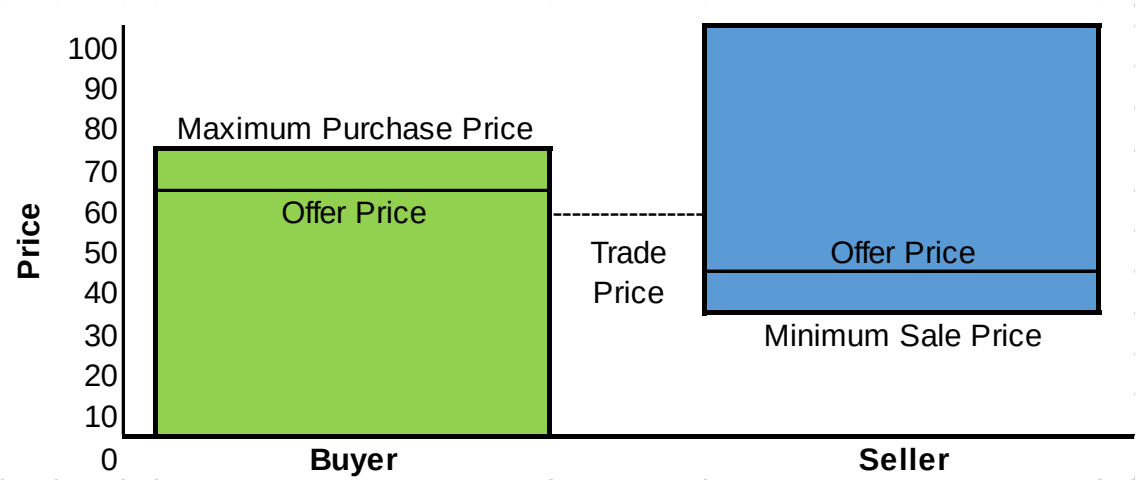

Figure 1: Example Buyer and Seller Prices in a Successful Trade.

At the end of the model run, the number, average, and standard deviation of the successful trades is printed, along with how long the model took to run. For the purpose of showing that it is possible to achieve the market clearing price predicted by economic theory – solely through the use of buyers and sellers with no knowledge of the market – the average and standard deviation are sufficient[3]. A simple average, however, does not provide much insight into the behavior of the model. One way to gain deeper insight into a model is by having agents generate their own narrative describing their behavior.

[3] Although, as Anscombe (1973) showed so beautifully, mean and standard deviation should never be considered sufficient without examining the actual distribution of the data.

This thesis describes a version of ZI Traders which includes agents who generate their own narratives about attempted trades[4]. The agents, even though they have "zero-intelligence" are capable of generating narratives. This is the first approach to narrative generation using individual agents with actions that are independent of their previous actions (i.e. they have no memory).

Model Steps

The multi-threaded ZI Traders model operates as follows:

1. **Generate Agents:** The agent population is generated based on parameters provided. A set number of buyer agents are created, each with an ID and a single property: their "value", which is a random number between zero and a set "maximum value". Next, a set number of seller agents are created, each with an ID and a single property: their "cost", which is a random number between zero and a set "maximum cost". Typically the number of buyer agents will be identical to the number of seller agents, and the buyers' "maximum value" will be identical to the sellers' "maximum cost", but the model does not require that. Of course, if the buyers' values are considerably lower than the sellers' costs, there

[4] Note, that portions of this thesis are adapted from previous work I did for a class on Advanced Agent-Based Modeling taught by Robert Axtell in Fall 2011, where I programmed the agents in the Zero-Intelligence Traders model to generate narrative. That work inspired this thesis topic, which include a refinement of the earlier work and an expansion to include the identification of the agent's whose narratives are most worth reviewing.

will be fewer successful trades. The IDs and values for each of the buyers are written to a comma-separated values (CSV) file, and the IDs and costs for each of the sellers are written to a separate CSV file.

2. **Assign Agents to Threads:** The buyer and seller agents are distributed evenly between however many treads are desired (1 or more, based on a model parameter), and each thread is set to attempt an equal number of transactions (defined by a total number of transactions parameter divided by the number of threads). Each thread and set of buyer/sellers is independent – only buyers and sellers on the same thread have the opportunity to trade with each other.
3. **Conduct Trade Attempts:** One buyer who has not completed a trade is randomly selected, and forms a bid price, which is a random number between zero and that agent's maximum value. One seller who has not completed a trade is randomly selected, and forms a sell price, which is a random number between the agent's minimum cost and the maximum cost. If the buyer's bid price is greater than the seller's sell price, then a random number is chosen between the two values and that becomes the completed transaction price. For example, a buyer is chosen with a maximum value of 20, and that buyer sets a bid price of 16. A seller is chosen with a minimum cost of 10 and sets a sell price of 15. Since the buyer's bid price is greater than the sell price, a trade can be completed, and a random number is chosen, say 15.432, between the two as the transaction price. If the bid and sell prices had been reversed, then no transaction would be possible. Once a buyer or seller completes a trade, it will no longer be selected to

try another trade. If it does not complete a trade, it can be selected again – even the next pass through.

4. **Record Trade Attempt:** Whether a trade attempt was successful or not, each agent (buyer and seller), records its own record of the trade attempt. The record is simply the information available to the agent, along with its own narrative about the trade attempt. The transaction record includes (if known) the ID of the buyer, the ID of the seller, the buyer's value, they buyer's offer, the seller's minimum cost, the seller's offer, the sale price (for a completed transaction), a Boolean value indicating whether the trade was completed, a time stamp, and a narrative. Obviously, a buyer will not know the seller's minimum cost, it will only know the seller's offer, and vice versa. The narrative generation is discussed below. Each agent generates a running average of the average amount it offered in each trade attempt, although this is only used as a value in its own narrative – the average does not affect the amount an agent offers in future trade attempts. Each thread also maintains its own set of aggregate data on successful trades, including the total number of trades, the sum of all trade prices, the squared sum of all trade prices, and the minimum and maximum trade prices. In the event of a successful trade, the information is added to the thread's trade data. Note that individual trades are not recorded, they are simply added into the running aggregate values.
5. **Save Transaction Records:** After each of the threads has completed its set of trade attempts, the program saves a CSV file with each of the buyer agents' trade

attempt records, and a separate CSV file with each of the seller agents' trade attempt records.

6. **Rank Agents:** Agents are ranked on a set of "interestingness" features (see below for more information), and narrative is generated for each agent who ranks in the top N (e.g. 10) agents in how they measure on that features. For example, one feature is final trade price. Those agents (buyers and sellers) who completed a trade for the lowest N prices, the highest N prices, and the N prices closest to the overall average trade price have that noted in the beginning of their individual narrative.
7. **Generate and Save Narratives:** Then separate text files (one for buyers and one for sellers) are generated containing a consolidated narrative for each agent. The consolidated narrative starts with a summary of the number of trade attempts the agent made and whether it was ultimately successful in completing a trade, followed by notes indicating whether the agent ranked in the top N on any interesting features, and then writes out the narratives from each of the agent's attempted trades in increasing chronological order. Two separate files are generated with copies of the narratives of any buyers and sellers who ranked in the top N on any features, sorted in descending order by the number of top N rankings of each of the agents.
8. **Summarize Model Run:** Once the files have been saved, the program combines the aggregate values from each thread and prints out the overall number of successful transactions, average price and standard deviation. The total amount of

time the model required to run is printed, along with how long the trading portion of the model took and how long the narratives took to generate. Then the program is finished.

Model Structure

The Java implementation of the ZI Traders model consists of six files/classes[5]:

1. **ZITraders**.java, the main portion of the program, including the majority of the logic for running the trades, multi-threading, ranking agent features, and generating data and narrative files.
2. **AgentPopulation**.java, an object for generating and managing arrays of buyer and seller agents.
3. **BuyerAgent**.java, an object for managing the data and behaviors of each individual buyer agent. This module also generates all of the individual buyer narrative.
4. **SellerAgent**.java, an object for managing the data and behaviors of each individual seller agent. This module also generates all of the individual seller narrative.

[5] The original code supplied by Rob Axtell consisted of five of these files, TradeAttempt is new. As a rough indication of the degree of modification done to the code, the following is a list of the files with their original number of lines of code followed by their current number: ZITraders (226/1335), AgentPopulation (59/122), BuyerAgent (34/743), SellerAgent (36/657), TradeAttempt (0/205), and Data (81/81). While this shows an increase in lines of code from 436 to 3143, roughly seven-fold, line counts are horribly imprecise since they include comments, commented out test code, blank lines, and code that seemed like it might be useful, but was never ultimately used, so I would not recommend reading too much into the actual numbers.

5. **TradeAttempt**.java, an object for maintaining a record of each individual trade attempt for each buyer and seller, independently (i.e., each buyer and seller creates their own record of the trade attempt, which is stored by each agent independently).
6. **Data**.java, an object that aggregates quantitative information about the results of successful trades and calculates the resulting average trade price and standard deviation of trade prices, for example.

The model was tested in Java 8, but should run in any version back to Java 5.

Narrative Generation

Narratives are based on the information the agent has available to it about its current trade attempt, the number of attempts it has had and the average amount it has offered in previous trade attempts, if it has had any. Variations in the generated narrative are based on whether the agent is a buyer or seller, if the trade was successful, the number of trades the agent has attempted, the difference between the amount it offered and its value or cost, and if successful, the difference between the transaction price and its value or cost. The buyer narrative is structured identically, but varies in its language appropriately (e.g., happier if it gets a lower price).

The narratives start with a general reaction to whether the trade was successful or not, which varies based on how many prior trade attempts there have been for the agent. For example, “My first trade attempt was successful!” or “My fifth attempt failed. This is getting depressing.” For unsuccessful trade attempts, the range in emotional reactions on the part of the agents is based loosely on Elisabeth Kübler-Ross’s (1969) “Five Stages of Grief”:

1. Denial
2. Anger
3. Bargaining
4. Depression
5. Acceptance

Although, I made the reaction to the second unsuccessful trade something which might be analogous to “bargaining”, and continued in order from there. So the reaction to the third unsuccessful trade is denial, the fourth is anger, the fifth is depression, the sixth through tenth are acceptance and after the tenth, it becomes neutral or perhaps puzzlement (“Will this ever end with a successful trade?”). The Kübler-Ross model was chosen since it gives a simple progression in emotional states and is well enough known that it would likely resonate with people reading the narratives. It is certainly not intended for the agents to convey a similar level of emotion to what a person my feel in the case of loss or a death – these agents are only modeling the selling of products after all.

This reaction is followed by a description of who the offer was made to, how much the offer was and what percentage that is over or under the agent's cost or value (as appropriate).

For successful trades, the narrative ends with the agent's reaction to the trade amount based on the variance between its cost/value and the sale price. If the difference is less than 10%, the agent wonders if it could have gotten a better deal. If the difference is between 10% and 30%, it feels good about the trade. And if the difference is greater than 30% it is ecstatic about the trade.

For unsuccessful trades, the agent first says what its average offer was, and the percentage difference from its cost or value. It then ends with similar reaction to a successful trade based on the difference between its offer and its cost or value. If the difference is less than 10%, the agent does not think it could have offered a much better deal. If the difference is between 10% and 30%, it feels that its offer was reasonable. And if the difference is greater than 30% it wonders if it asked for too much.

One issue with this kind of narrative is that the agents at the low end of the price range (closer to zero) are always more likely to offer or demand greater than 30%, so their narratives will be skewed. Of course, at that end of the spectrum, they are outliers.

Please note that while agents do have memory of prior transactions and generate a running average of the offer prices they made, they do not use any of that information when generating bid prices – they remain "zero intelligence."

Identifying Interesting Agents

Given that there are thousands of agents, and each agent generates some amount of narrative (even if just to say that there were no transaction attempts), it is important to identify the agents whose narratives are most worth the time for a person to review. I refer to this as finding the "interesting" agents.

The process for finding interesting agents consists of three steps:

1. Defining the measures available in the model;
2. Determining which metrics can be calculated from those measures; and
3. Deciding which metric values are likely to indicate the most interesting agents.

The first step, defining the measures, involves understanding which elements of the model can be quantified. These are frequently going to be numeric variables associated with an agent or counts of things that agents do, such as interact with other agents. In the case of the ZI Traders model, here are the measures:

1. Buyer's Maximum Purchase Price
2. Buyer's Offer Price

3. Seller's Minimum Sale Price
4. Seller's Offer Price
5. Trade Price (identical for buyers and sellers)
6. Trade/Transaction Attempts

The second step is to determine which metrics can be calculated from the measures. The metrics can range from simple counts of the number of trade attempts, for both successful and unsuccessful buyers and sellers, to more involved ones such as the trade prices closest to the overall average trade price or the range of prices offered by a buyer or seller in all of their trade attempts. Table 1 lists all of the metrics used within the ZI Traders model.

The third step is to decide which metric values are likely to indicate the most interesting agents. In general, this will be those values at the extremes of distributions, or, as in the case of the trade price, it can also be the values closest to the average trade price, which is done to make sure that a person is not looking only at extremes, but also more "typical" agents. The way this is handled in the ZI Traders model is to identify the "Top N" items in the distribution, where N is usually 10, so the agents with the 10 highest or lowest values for any metric are identified as interesting. The next section, on the Ranking Process, describes how the model actually identifies those agents.

Table 1: Ranked Features / Metrics

Metric	Buyer / Sellers	Trade Success	Sample Rank Narrative
Buyer's Maximum Purchase Price - Seller's Minimum Sale Price	Both	Successful	* This agent ranks in Nth place for lowest difference between the buyer's maximum purchase price and the seller's minimum sale price among buyers who made a successful trade, with a difference of 10.65.
Buyer's Offer Price - Seller's Offer Price	Both	Successful	* This agent ranks in Nth place for lowest difference between the buyer's offer price and the seller's offer price among buyers who made a successful trade, with a difference of 0.34.
|Trade Price – Overall Average Price|	Both	Successful	* This agent ranks in Nth place for lowest difference between the final trade price and the overall average transaction price (50.02), with a difference of 0.18.
Buyer's Maximum Purchase Price	Buyer	Successful	* This agent ranks in Nth place for lowest maximum purchase price among buyers who made a successful trade.
Buyer's Offer Price	Buyer	Successful	* This agent ranks in 10th place for lowest successful offer price among buyers.
Trade Price	Buyer	Successful	* This agent ranks in Nth place for lowest successful trade price among buyers.
Seller's Minimum Sale Price	Seller	Successful	* This agent ranks in Nth place for highest minimum sale price among sellers who made a successful trade.
Seller's Offer Price	Seller	Successful	* This agent ranks in 10th place for highest successful offer price among sellers.
Trade Price	Seller	Successful	* This agent ranks in Nth place for highest successful trade price among sellers.
Trade Attempts	Both	Successful	* This agent ranks in Nth place for number of attempts before successfully making a trade. With tie: * This agent ranks in Nth place for number of attempts before successfully making a trade (tied with 10 others).
Trade Attempts	Both	Unsuccessful	With tie: * This agent ranks in Nth place for number of unsuccessful trade attempts without making a trade (tied with 20 others).
Offer Prices	Both	Successful	* This agent ranks in Nth place for the range of prices offered (97.09 from max to min) in trade attempts.

Ranking Process

In order to identify interesting agent features, a process is run at the completion of the model to identify the ordered ranking of various metrics regarding agent performance. Across buyers and sellers, there are 12 metrics calculated as shown in Table 1. The general process is as follows:

1. Loop through all of the relevant agents (buyers/sellers, successful/unsuccessful traders) and calculate the metric.
2. Count the number of times each metric value occurs. In practice, the only metric where this matters is the number of trade attempts since that it the only integer metric. That is because the odds of having multiple identical numbers among values derived from one or more randomly selected 64-bit floating point Java double variables is extremely low. However, this is still done for each metric. An example of the counts for successful buyer trade attempts is shown in Table 2.
3. Sort the metric values in the appropriate order and rank the values in order of importance/"interestingness" from 1 (the most important) to however many unique values there are for the metric. Whether the sorting is done from low values to high values or vice versa is dependent on the context in which it will be used. A higher number of trade attempts is more interesting for both buyers and sellers, whether they are successful or unsuccessful at completing a trade. However, trade price interestingness is different for buyers and sellers, with a

lower price being more interesting for buyers and a higher price more interesting for sellers. The sorting is done for each of these metrics by storing the counts of the values in a Java TreeMap key-value data structure[6], which sorts records automatically as they are stored (in either ascending or descending order, as appropriate for the metric). The key is the metric value (e.g. number of trade attempts), while the value is the number of agents who had that metric value. The TreeMap also allows for quick retrieval of the values by hashing the key. The ranks of each metric value are stored in a separate TreeMap or HashMap (in the case of trade attempts since the number of items is small).

4. Determine the "Top N" value of the metric. "Top N" is a threshold number (e.g., the top 10) for how many of the highest ranked agents are identified for each metric in the "interesting narratives" files. No less than N agents will be reported as interesting on the metric, and ties will be reported as such (although only likely for trade attempt counts). As an example, Table 2 shows the ranks of the number of trade attempts. If N is 10, then the Top N value for the metric will be 5 trade attempts since only 6 buyers had 6 or more trade attempts, but 15 had 5 or more (as shown in the running total column). The Top N value for each metric is stored as the value of the metric that must be equaled or exceeded for an agent to qualify as interesting on that metric. This is done to simplify the identification of

[6] The Java TreeMap (https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html) is "a Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms."

interestingness when generating agent narratives – rather than calculate the agent's score on the metric, then figure out the rank from the score, the score itself is the determinant of interestingness.

5. Use the Top N value to indicate whether the agent is interesting when generating the agent's narrative.

Table 2: Example Numbers of Successful Buyer Trade Attempts

Rank	Number of Trade Attempts	Numbers of Buyers	Running Total
1	7	2	2
2	6	4	6
3	5	9	15
4	4	39	54
5	3	82	136
6	2	189	325
7	1	331	656

Verification & Validation

With any form of simulation, it is important to make sure that the model accurately reflects what it is simulating and to make sure that the model itself is working correctly and as intended. As described by Balci (1998), the process of determining whether the model accurately reflects what it is simulating is called validation, and ensuring correct operation is called verification. Testing is done to assess the validity of the model and also for verification that the model is behaving properly.

This model was tested extensively and iteratively over the course of its development. Each addition of new code was tested by adding in additional lines of code that printed out immediate results for display in the Java output window or that stored the generated data in output files for verification at the completion of a model run. Additional verification is done by storing and saving data in duplicate – buyers and sellers each have copies of the data from all of their trade attempts – which allows for comparison of the data files to make sure the values are being calculated and stored correctly. This approach did help catch some coding errors in the course of developing the model. Buyer and seller agents are implemented separately, but they are close to identical in many of their features and behaviors. This allowed for development of code for one type of agent and the copying of that code to the other type, with minor modifications made to handle the differences in behavior. However, there were times when lines of code that needed modification were missed. In these cases, a careful review of the data files and the narratives helped to identify errors in the code.

The narratives themselves were excellent tools for debugging and verification. Since the narratives contain values for each trade attempt, a running average, and summary values, it is possible to do a manual check that the correct numbers are being calculated. When errors were found (and they were), it became a guide to what portion of the code needed to be reviewed. Inconsistency in the narrative values led to both opening up the data files to check if the values presented in the narrative matched the values stored in the data, and to a review of the relevant code to find the bug.

Since the primary purpose of this model is to demonstrate the possibility of having an ABM generate narrative, the focus has not been on demonstrating the validity of the model. However, the model does consistently produce a "price … close to the point of intersection of the supply and demand functions" (Gode & Sunder, 1993), which provides a degree of validation that the model is faithfully recreating the behavior of the standard zero-intelligence traders model.

Results & Findings

Any ABM run generates a considerable amount of data, and while the purpose of this thesis is to describe how to generate narrative to understand the behavior of individual agents within a model, aggregate analysis of the model is still necessary and valuable. This section will start with an aggregate-level look at the buyers and sellers and some analysis of the results of one model run, followed by examples of the individual-level narratives generated by the model.

Aggregate Results and Analysis

At the completion of a model run, summary text is displayed showing the number of successful trades, the average and standard deviation of the trade prices, the number of interesting buyers and sellers, and how long the model took to run. Here is an example:

Final stats: 656 transactions at 48.42 average price; 20.43 standard deviation.

There were 83 interesting buyers and 84 interesting sellers.

The model ran in 269 milliseconds.

Generating and saving narrative files took 1054 milliseconds.

Total elapsed time: 1323 milliseconds

BUILD SUCCESSFUL (total time: 1 second)

The record of all transaction attempts also provided a useful set of information. The following charts are based on a run of the model with 5000 buyers, 5000 sellers, 10,000 transaction attempts and a maximum sale/bid price of 100.

Figure 2 shows the distribution of buyers by their maximum purchase price (rounded to the nearest integer) in the blue bars, accompanied by the number of those buyers who completed a successful trade (the red bars). As can be seen from the chart, there is close to a uniform distribution of maximum purchase prices across the buyers, averaging at 50 buyers per integer price, with peaks at 64 buyers (with prices of 41 and 96) and the smallest number of buyers, 23, at 0. Zero being the smallest is not surprising given that all of the numbers are rounded up from decimal places of greater than or equal to 0.5 and down from less than 0.5, and there are no negative values to round up to zero. The 32 100s is a little more surprising, given an expected count of 25, but not unreasonable. As can be seen from the red bars indicating the number of successful traders at each rounded maximum purchase price, the buyers who were more likely to be successful were those more willing to trade at the higher end of the price range, while very few at the lower end were successful at making trades. Of the 2498 buyers with a rounded maximum purchase

price of 50 or lower, only 76 were successful at making a trade (11.6% of the 656 successful trades).

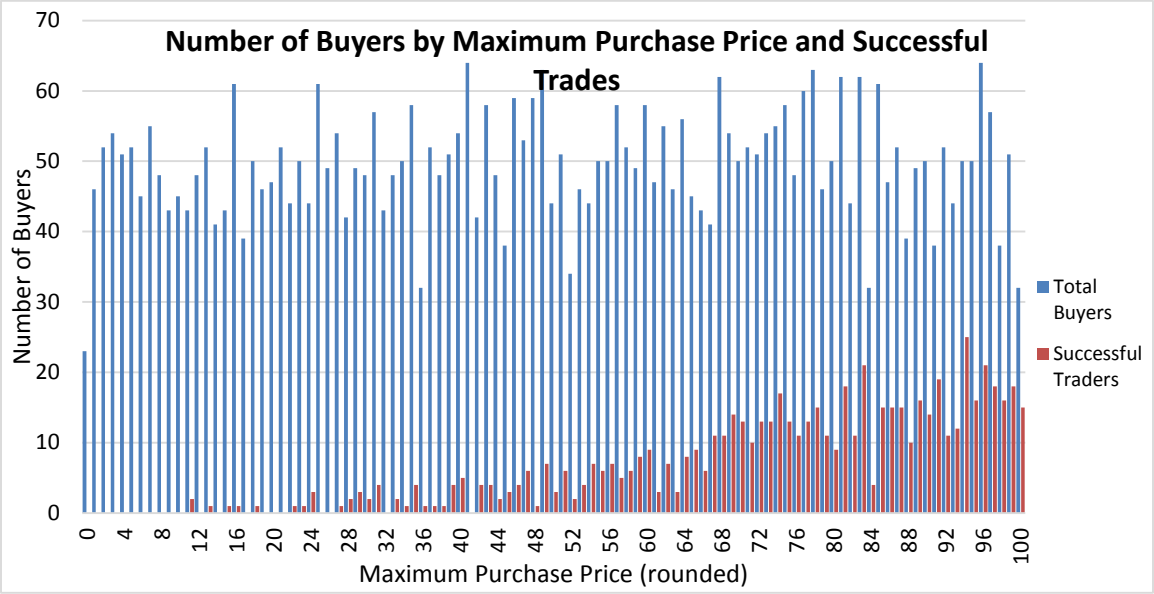

Figure 2: Number of Buyers by Maximum Purchase Price and Successful Trades

Figure 3 shows the distribution of sellers by their minimum sale price (rounded to the nearest integer) in the blue bars, accompanied by the number of those sellers who completed a successful trade (the red bars). As can be seen from the chart, there is close to a uniform distribution of minimum sale prices across the sellers, averaging at 50 sellers per integer price, with peak of 75 sellers at a price of and the smallest number of sellers, 24, at 100, followed by 36 at 0 (for the same reason as the buyers). As can be seen from the red bars indicating the number of successful traders at each rounded minimum sale price, the sellers who were more likely to be successful were those more willing to trade at the lower end of the price range, while very few at the higher end were successful at

making trades. Of the 2501 sellers with a rounded minimum sale price of 50 or higher, only 67 were successful at making a trade (10.2% of the 656 successful trades).

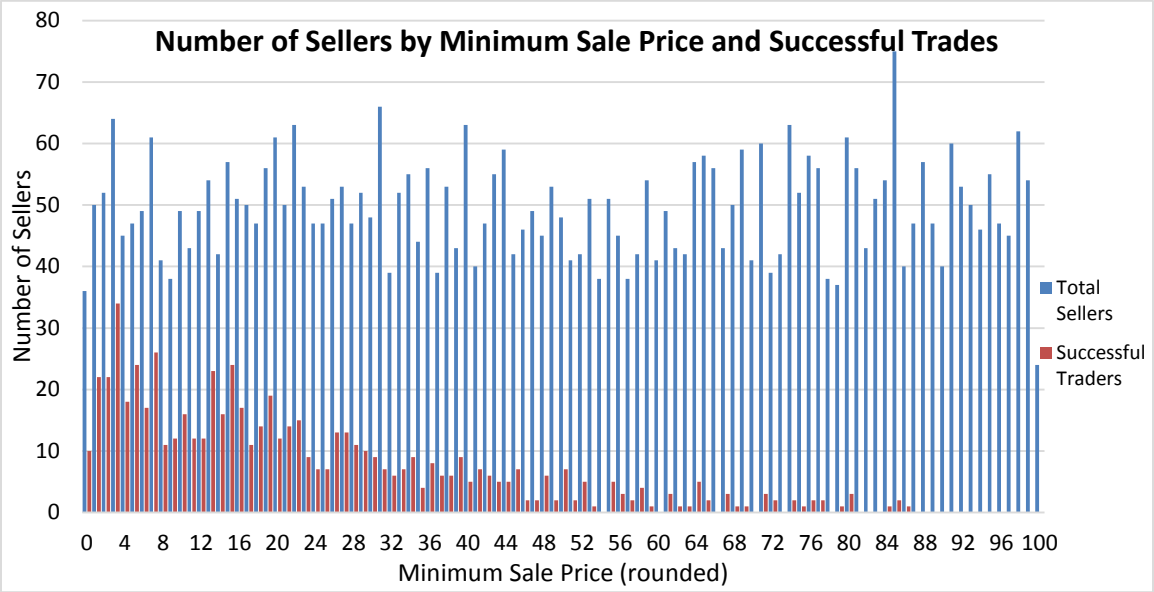

Figure 3: Number of Sellers by Minimum Sale Price and Successful Trades

The distribution of the number of transaction attempts attempted by each buyer and seller is shown in Figure 4 and Figure 5, and because there were twice as many transaction attempts as buyers or sellers, the average number of transaction attempts per buyer or seller is two.

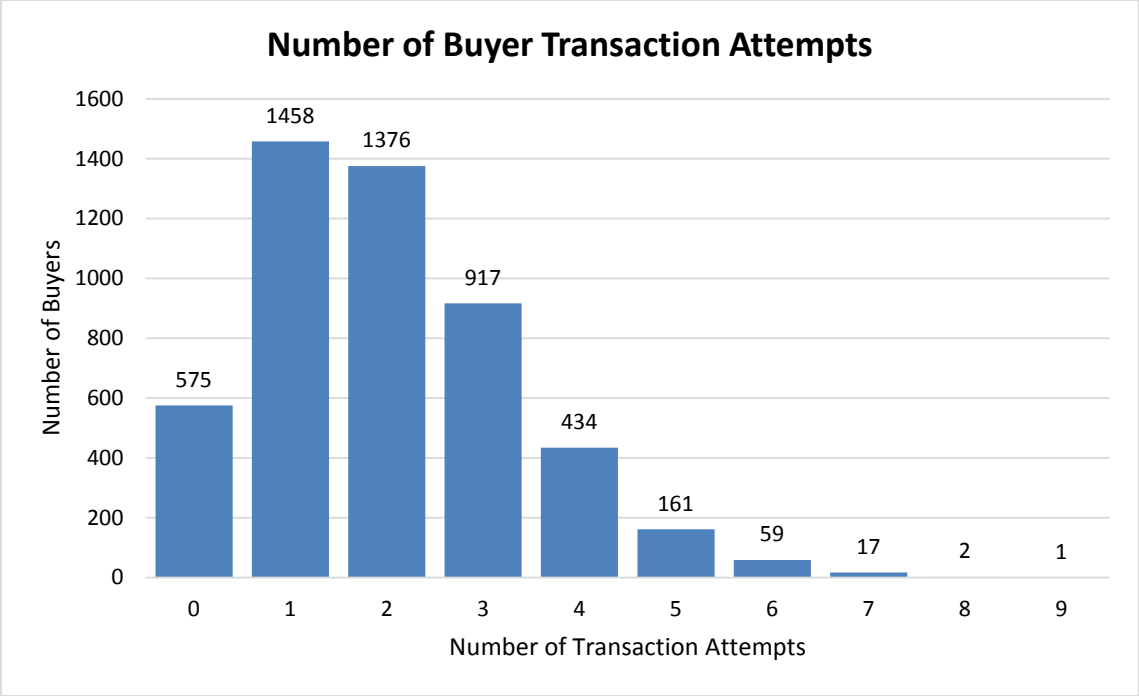

Figure 4: Number of Transaction Attempts by Buyers

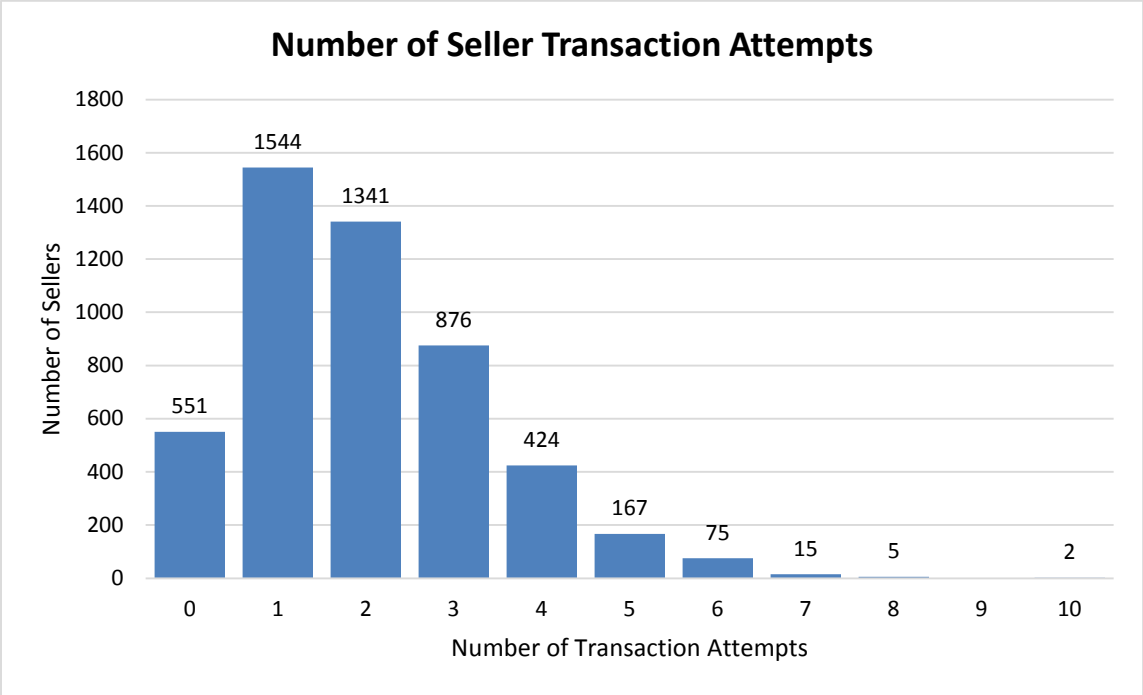

Figure 5: Number of Transaction Attempts by Sellers

The final trade prices, as shown in Figure 6, follow something close to a Gaussian distribution, which is what is expected. Most trades should be close to the middle of the range of possible prices given the structure of the model. The exceptions, like the single trade at 67 or the 15 trades at 30, are not particularly surprising given that there were only 656 successful trades, which were distributed between 101 possible bins (although only 93 were used).

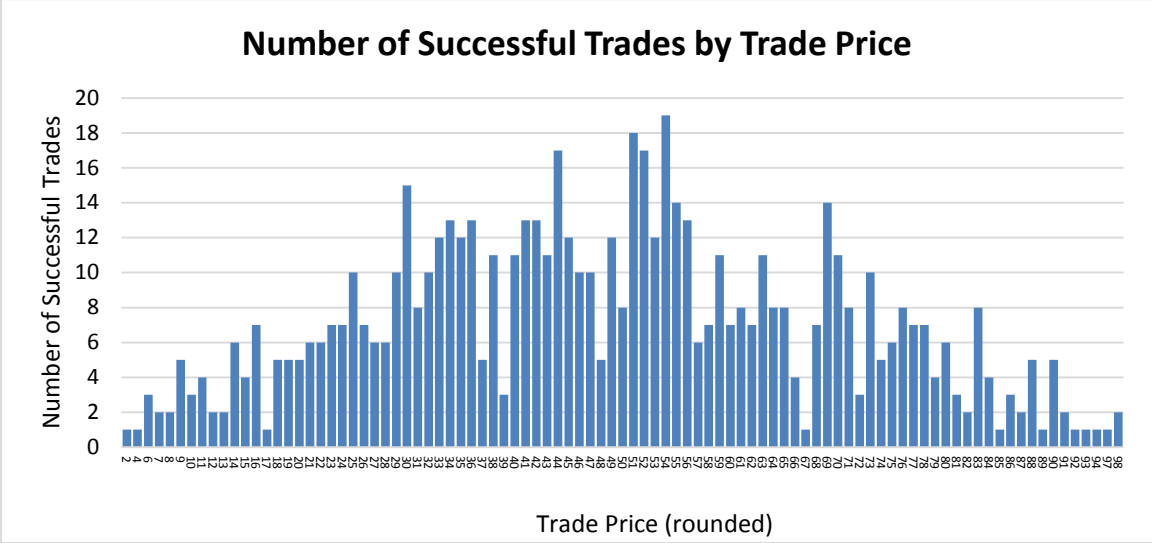

Figure 6: Number of Successful Trades by Trade Price

As shown in Figure 7, for cases where a trade was successfully completed, there can be a wide range in the difference (rounded to the nearest integer) between the buyer's maximum purchase price and the seller's minimum sale price. Other than a slight prevalence in the middle of the distribution, with a mode of 48 and a mean of 53.05, there is no typical difference between the starting point of buyers and sellers. Three buyer/seller pairs had a rounded difference of 6 between the buyer's maximum purchase

price and the seller's minimum sale price, meaning the trade was very unlikely, while two had a difference of 97, indicating buyers who would pay almost anything getting paired up with sellers who would sell for almost nothing. In the case of the two pairs with a 97 difference, one traded at 5.62 and the other at 49.16.

Figure 7: Number of Successful Trades by Difference in Buyer Maximum Purchase Price & Seller Minimum Sale Price

Figure 8 shows the number of successful trades by the rounded difference between the buyer's and the seller's offer prices. In this case, there are many more successful trades where the offer prices were close together, and fewer where the differences were large. This is unsurprising given that the buyers' and sellers' starting values were drawn from a uniform distribution, so any random pairing of them should have more likelihood of being closer to each other in value rather than further away.

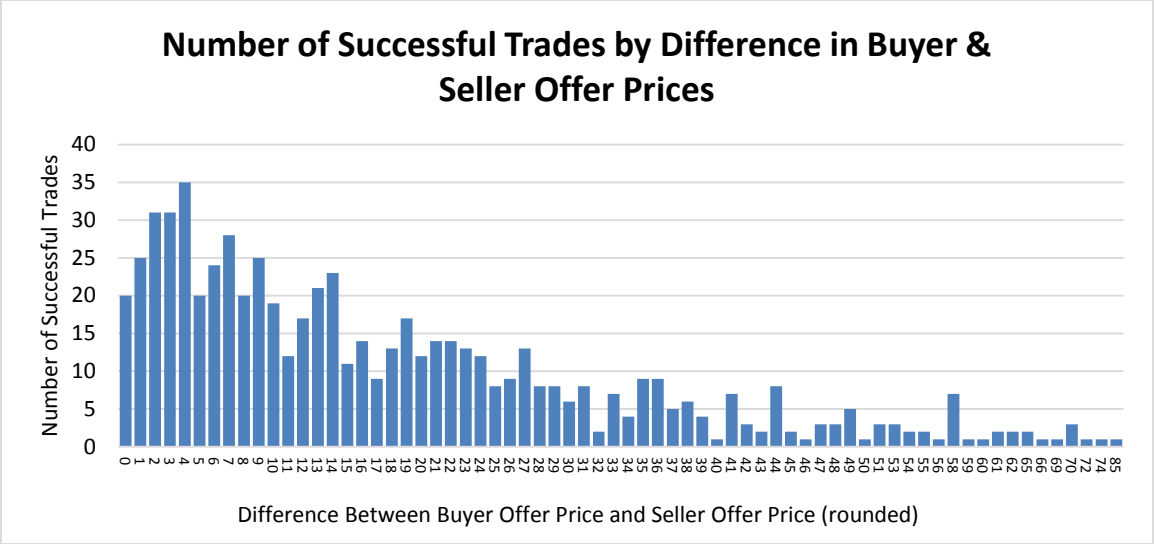

Figure 8: Number of Successful Trades by Difference in Buyer & Seller Offer Prices

Narrative Results

While the aggregate analysis gives a good overall sense of the behavior of the model, there are details that are easily missed and questions that cannot be answered without examining the history and characteristics of individual agents. For example, Figure 4 shows that one buyer succeeded at making a trade on its ninth attempt, but that figure cannot answer questions such as, "*why did it take nine times for one agent to make a trade?*" or, "*what are the characteristics of that agent*?" The generated narratives do provide insight into the behavior of the model at an individual agent level.

The question, "*since this information can be gotten from the individual agent data, why bother turning it into narrative*?" is reasonable to ask. The answer is that the narrative does at least a couple things. First, it makes the data more interesting to read, both by

generating a story of sorts and by packaging the information in a coherent, connected format. Second, the process of generating narrative – or more accurately developing the algorithms that turn events and data from the model into narrative – requires the developer to identify the most important elements of an agent's history and think about how to turn those elements into pieces of a story. This involves highlighting certain elements, such as the number of trade attempts or the offer price, and figuring out other elements that do not add to the ongoing narrative and can be excluded, such as noting the buyer's maximum purchase price in every transaction attempt narrative.

Here is an example consolidated seller narrative, with four interesting features:

> *Narrative for seller 1417, who has a minimum sale price of 84.35362886814184:*
>
> *4 trades have been attempted. The range of offers made by this seller was 12.05155579. A successful trade was completed with buyer 3735 for 90.48294221466527.*
>
> ** This agent ranks in 8th place for highest successful trade price among sellers.*
> ** This agent ranks in 3rd place for highest successful offer price among sellers.*
> ** This agent ranks in 3rd place for highest minimum sale price among sellers who made a successful trade.*

** This agent ranks in 10th place for lowest difference between the seller's minimum sale price and the buyer's maximum purchase price among sellers who made a successful trade, with a difference of 11.353388426391035.*

Tue Apr 14 01:50:48 EDT 2015: My first trade attempt wasn't successful. I offered buyer 1417 97.05, which is 15.05% over my minimum sell price -- I think that was reasonable.

Tue Apr 14 01:50:48 EDT 2015: Two failed attempts. Third time is the charm... hopefully. I offered buyer 1417 91.93, which is 8.98% over my minimum sell price -- I couldn't go much lower! My average offer was 94.49, which is 12.02% over my minimum.

Tue Apr 14 01:50:48 EDT 2015: The third time was not the charm. I think I've been reasonable. This time, I offered buyer 1417 99.94, which is 18.48% over my minimum sell price -- I think that was reasonable. My average offer was 96.31, which is 14.17% over my minimum.

Tue Apr 14 01:50:48 EDT 2015: After 4 attempts, I was finally able to make a trade. I offered buyer 1417 87.89, which is 4.20% over my minimum sell price. This wasn't too bad of a trade. I got 7.27% over my minimum. The sale price was 90.48. I wonder if I could've gotten a better deal?

The seller was successful in making a trade on its fourth attempt, and had a nice range of reactions based on various offer percentages. Its minimum sale price was 84.35362886814184.

It is the intersection of multiple characteristics shown here through the ranking text in each interesting agent narrative that shows where narrative provides information that aggregate analysis does not. In the case of the seller above, it has four interesting features – it ranked in the top 10 seller agents on those characteristics – and it helps tell the story of how an agent with a minimum sale price of more than 84 was able to make a successful trade. The story is basically that this agent, even though it had the third highest successful offer price and minimum sale price, was able to find a buyer, that had a higher maximum purchase price (about 11.35 higher than the seller's minimum), placing this seller in 10th place for smallest difference between those prices. This agent got lucky, but there were others who likely were even luckier.

Here is an example consolidated buyer narrative, with three interesting features:

> *Narrative for buyer 4540, who has a maximum purchase price of 19.637677770011063:*

6 trades have been attempted. The range of offers made by this buyer was 16.92220128. A successful trade was completed with seller 1419 for 9.789312916173731.

** This agent ranks in 3rd place for number of attempts before successfully making a trade (tied with 3 others).*
** This agent ranks in 5th place for lowest successful offer price among buyers.*
** This agent ranks in 3rd place for lowest maximum purchase price among buyers who made a successful trade.*

Tue Apr 14 01:50:48 EDT 2015: My first trade attempt wasn't successful. I offered seller 3176 1.93, which is 90.18% under my maximum -- I wonder if it was too little?

Tue Apr 14 01:50:48 EDT 2015: Two failed attempts. Third time is the charm... hopefully. I offered seller 4829 1.69, which is 91.42% under my maximum -- I wonder if it was too little? After 2 trade attempts, my average offer was 1.81, which is 90.80% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: The third time was not the charm. I think I've been reasonable. This time, I offered seller 3242 18.61, which is 5.24% under my

maximum -- I couldn't go much higher! After 3 trade attempts, my average offer was 7.41, which is 62.28% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: What the heck, is nobody willing to trade with me?! This was my fourth failed attempt. I offered seller 3699 16.66, which is 15.17% under my maximum -- I think that was reasonable. After 4 trade attempts, my average offer was 9.72, which is 50.50% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: My fifth attempt failed. This is getting depressing. I offered seller 1902 14.14, which is 27.98% under my maximum -- I think that was reasonable. After 5 trade attempts, my average offer was 10.60, which is 46.00% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: After 6 attempts, I was finally able to make a trade. I offered seller 1419 10.77, which is 45.16% under my maximum. This was a great trade! I paid an unbelievable 50.15% under my maximum. The sale price was 9.79. I almost feel guilty about the deal I got from the seller... Almost.

This buyer was successful in making a trade on its sixth try. Its maximum purchase price was 19.637677770011063.

Finally, here is an example of a consolidated buyer narrative for a buyer who did not complete a successful trade even after eight attempts:

Narrative for buyer 55, who has a maximum purchase price of 25.908776065489192:

8 trades have been attempted. None have been successful.

** This agent ranks in 1st place for number of unsuccessful trade attempts without making a trade (tied with 6 others).*

Tue Apr 14 01:50:48 EDT 2015: My first trade attempt wasn't successful. I offered seller 1436 14.11, which is 45.53% under my maximum -- I wonder if it was too little?

Tue Apr 14 01:50:48 EDT 2015: Two failed attempts. Third time is the charm... hopefully. I offered seller 4916 16.38, which is 36.79% under my maximum -- I wonder if it was too little? After 2 trade attempts, my average offer was 15.25, which is 41.16% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: The third time was not the charm. I think I've been reasonable. This time, I offered seller 836 20.34, which is 21.50% under my

maximum -- I think that was reasonable. After 3 trade attempts, my average offer was 16.94, which is 34.61% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: What the heck, is nobody willing to trade with me?! This was my fourth failed attempt. I offered seller 747 21.17, which is 18.27% under my maximum -- I think that was reasonable. After 4 trade attempts, my average offer was 18.00, which is 30.52% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: My fifth attempt failed. This is getting depressing. I offered seller 1621 10.89, which is 57.98% under my maximum -- I wonder if it was too little? After 5 trade attempts, my average offer was 16.58, which is 36.02% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: My 6th try and still no luck. I'll just keep trying. I offered seller 2707 14.73, which is 43.15% under my maximum -- I wonder if it was too little? After 6 trade attempts, my average offer was 16.27, which is 37.20% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: My 7th try and still no luck. I'll just keep trying. I offered seller 3682 2.04, which is 92.12% under my maximum -- I wonder if it was too little? After 7 trade attempts, my average offer was 14.24, which is 45.05% under my maximum.

Tue Apr 14 01:50:48 EDT 2015: My 8th try and still no luck. I'll just keep trying. I offered seller 939 25.83, which is 0.29% under my maximum -- I couldn't go much higher! After 8 trade attempts, my average offer was 15.69, which is 39.46% under my maximum.

In this narrative, one can see almost the full range of emotional reactions to unsuccessful trades. This buyer's maximum purchase price was 25.908776065489192, so it should have had a reasonable chance of making a trade, but it offered on average 15.69 by its eighth try. It seems to have gotten into a rut of offering relatively low bids and having trouble finding sellers willing to take small amounts.

Note that the narratives contain a mixture of numbers shown to full available decimal place precision (which is what is used by Java) and ones rounded to two decimal places. This was done deliberately to make sure there is sufficient detail where potentially relevant (e.g. in the agent introduction and rankings), but two digits in the trade attempt narratives to improve readability. The distinction can also be thought of as being between the narrative, where the story is more important, and the meta-narrative – the story about the story – where the precision is more important.

Discussion

Even in a simple model where agents have “zero-intelligence”, it is possible to generate useful narratives that provide deeper insight into the behavior of the model. In the case of the ZI Traders model, the agents do no reasoning about their offers and have no knowledge about the market, including their own personal history; however, it is reasonably easy to have the agents generate a narrative reaction to each trade attempt using minimal information about their past history (number of previous trade attempts and average offer price). Their knowledge of their history does not have any impact on the behavior of the agent (although it could, if desired), so the agents remain “zero-intelligence”.

The narratives are simple, but they are not random – the only variability is in the trade itself and the agent’s history. Given that, it could add some variety to make the language of responses randomly variable. It might be possible to add some additional variability to the narratives based on characteristics of the agents themselves. For example, by adding various personality types to the agents – some could have “big” personalities, others could be meek, or more prone to anger. The agents at either end of the price spectrum could also react differently than those in the middle.

In addition, there could be variability in the thresholds for reactions to differences in offer amount and the agent's cost/value. The under 10%, 10-30% and greater than 30% categories are arbitrary, although they seem to each get used with sufficient frequency that they appear to be reasonable. However, there is no reason that those categories could vary for each agent either randomly, or based on personality type, and/or on their cost/value. Some people might think that a 10-30% offer was fair, while others feel that 6-15% is fair and anything over that is too much/little. There is certainly plenty of room for both calibration and variability. Calibration, in this case, could involve running the model a number of times and analyzing the distribution of all offers or all successful offers to determine the thresholds for each response category. The thresholds could be decided by dividing the differences into simple thirds, or by using a Gaussian distribution and where most of the results fall plus or minus one standard deviation (~68%) and the extremes are greater than or less than one standard deviation (~16% each). This would give better justification for the categories rather than the arbitrary decision made when building the model.

At the moment, the model simply saves the narratives at the end of the model run. It might be valuable to report some narratives while the model is running in order to get a real-time sense of the model's behavior (although on my computer with the parameters noted above, the model took about 1300ms to run, so there is not much room for "real-time" reporting). It might be valuable to select a subset of agents to report on – perhaps

some with more interesting personalities or outliers along with “normal” agents. This becomes a much more significant concern as the size of the model grows.

Storing Data and Narrative

Each run of the model generates a set of files storing the data and narratives, although there are parameters for turning off this functionality. The names of the files contain a date and time stamp (to the second), so each run of the model generates new files without overwriting older versions. Table 3 lists the files that are saved with each model run.

The files generated as part of this process are large. For model runs with 5,000 sellers, 5,000 buyers and 10,000 transaction attempts, each of the two narrative text files and the two transaction record CSV files was approximately 3MB in size (for a total of ~12MB), and the other five smaller files come to another 550KB. With larger sets of agents and more transactions, the file size and related memory requirements for the agents could easily become prohibitive.

Table 3: Example Files Generated by Each Model Run

File Name	Type	Description	Size (Bytes)
Interesting Seller Narratives 2015-04-17 1744-22.txt	Narrative (txt)	Any seller narratives that met the criterion for at least one "interestingness" factor	117,284
Interesting Buyer Narratives 2015-04-17 1744-22.txt	Narrative (txt)	Any buyer narratives that met the criterion for at least one "interestingness" factor	114,028
Seller Narratives 2015-04-17 1744-22.txt	Narrative (txt)	All seller narratives (whether successful or not, even if no trades were attempted)	3,283,026
Buyer Narratives 2015-04-17 1744-22.txt	Narrative (txt)	All buyer narratives (whether successful or not, even if no trades were attempted)	3,349,554
Completed Transactions 2015-04-17 1744-22.csv	Data (csv)	Data on all completed transactions, excluding narrative	88,395
Seller Transactions 2015-04-17 1744-22.csv	Data (csv)	Data on all seller transaction attempts (whether successful or not, even if no trades were attempted) , including narrative	3,284,434
Buyer Transactions 2015-04-17 1744-22.csv	Data (csv)	Data on all buyer transaction attempts (whether successful or not, even if no trades were attempted) , including narrative	3,335,029
Sellers 2015-04-17 1744-22.csv	Data (csv)	The ID and minimum sale price for all sellers	119,832
Buyers 2015-04-17 1744-22.csv	Data (csv)	The ID and maximum purchase price for all buyers	119,814
		Total Size:	13,811,396

If the model was increased to 500,000 sellers, 500,000 buyers and 1 million transaction attempts, the resulting files would likely total approximately 100 times the size of the ones in these runs, or 1.4GB, and it does not take much from there to quickly fill up hard drives on typical computers. Given that, it is reasonable to consider how to scale back on the amount of data collected or consider other methods of capturing the data.

One possibility is to write the transaction information including the narratives, to a database, which need not be located on the same computer as the one running the model. In this case, the agents would not retain a complete record of their own transaction attempts. If an agent needed its complete history, it could query the database; however, given that the narratives rely on the number of transaction attempts and average offer, it makes sense for the agents to maintain that information in the same way the threads maintain similar information for all successful transactions that occur in the thread.

Another possibility is to have a smaller number of journalist agents who report on themselves and/or other agents. They could identify the other agents to report on through various methods such as statistical sampling of the population, random selection or by building up a social network – possibly generated natively by any agents who attempt to trade with each other.

Finally, it might be worthwhile to consider passing messages outside of the system running the model. This would offload the handling and storage of the transaction data to a separate computer (similar to using a database on a separate server), at the potential expense of significant network traffic. One model for this is Twitter – each agent could "tweet" its transaction attempt and narrative. A private Twitter-like server would need to be set up, using, for example, GNU Social[7], and existing Twitter analysis or text mining

[7] http://www.gnu.org/software/social/

tools, such as Python NLTK[8] or the tm[9] and twitteR[10] packages for R, could be used to analyze the data.

When to Generate Narrative

Adding the functionality for generating narrative to an ABM requires a considerable amount of effort, so it is not something that can be casually added to a model. In this case, the original model, which did not generate any narrative or store details of individual agent histories, totaled 436 lines of code, while the final version of this model had approximately 3143. Of the approximately 2700 additional lines of code, about two-thirds were dedicated to generating narrative, with the remaining third primarily dedicated to storing and saving records of individual agent trade attempts, which would be done even if not generating narrative. However, that still means that approximately twice as much code was needed to save detailed model data compared to simply running the model, and 4 times as much code was needed to generate narrative.

This brings up the question, "*when is it reasonable to expend the effort needed to add narrative to an ABM?*" Other than as an example of the possibility of having an ABM generate narrative, I would not suggest that a model such as ZI Traders is a good candidate for adding narrative. The behaviors of the agents are too simple and the

[8] http://www.nltk.org/
[9] http://cran.r-project.org/web/packages/tm/
[10] http://cran.r-project.org/web/packages/twitteR/index.html

interactions between agents are trivial, while the features that make agents interesting are easy to extract from a saved transaction attempt record. In addition, the outcome of the model does not vary much from run to run – with 5,000 buyers, 5,000 sellers, a price range of 0 to 100, and 10,000 trade attempts, there are consistently about 650 successful trades with an average trade price of 50. Changing these parameters does not result in a different outcome other than in predictable ways: increasing or decreasing the price range still results in a mathematically predictable average trade price; increasing trade attempts will increase the number of successful trades, but not change the average trade price; and increasing the number of buyers and sellers has no impact on the average trade price. Those characteristics of the model certainly do not suggest that making the effort to program narrative generation requiring more than twice the code needed to run the model and save the data is worthwhile. However, they do suggest the characteristics that may make a model a good candidate for adding narrative.

Characteristics of an ABM that suggest it may be worthwhile investing the effort to automate narrative generation include:

1. Meaningful interactions between agents
2. An agent decision process that leads to different behaviors, and would benefit from making it explicit
3. Variability in model outcomes not driven directly by input parameters (e.g., non-linear results)
4. The model will be run many times to understand variations in model behavior

5. People other than the developer(s) will be using the model
6. The model exhibits unexpected and initially unexplainable behaviors. In addition to the narrative identifying unusual behavior, the process of developing the narrative may expose errors in the code.
7. Generally, when the model is complicated enough to justify the additional effort.

Ultimately, the decision on whether to have a model generate its own narrative is up to the modeler(s) and their determination of whether the value derived from the narrative is worth the considerable effort necessary to create it. Key factors for this decision are the amount of time and resources available to program the narrative generation and the effectiveness of the narrative in communicating the results of the model to its intended audience (which may be the people funding the development of the model).

Future Research

As mentioned in the introduction of this thesis, there are at least four possible approaches for generating narrative within ABMs:

1. an individual agent with actions that are independent of their previous actions (i.e. they have no memory);
2. an individual agent with memory of its past actions that influence later behavior; and
3. an omniscient "journalist" reporting on logical groups of agents (e.g. by geography, industry); and
4. an overall narrative about the behavior of the model, describing model-wide trends and possibly the actions of a few notable agents.

This thesis gave an example of the first approach. Another model that would meet the criteria for the first approach is the Schelling (1971) Segregation Model, which has memoryless agents, but also has more interaction with other agents. The Schelling Segregation Model has agents that move or stay in place on a grid depending on the number of agents on adjacent grid squares who are similar to them.

A relatively simple model which would be a good candidate for demonstrating the second approach is the El Farol Bar Model (Arthur, 1994; Rand & Stonedahl, 2007), which has

agents using their memory of previous visits to a bar along with various strategies to determine whether to go to the bar on a particular night and risk the place being too crowded. An excellent candidate for exploring the other approaches is the Sugarscape ABM (Epstein & Axtell, 1996).

Sugarscape Model Narrative

The Sugarscape ABM was one of the earliest agent-based models developed and published about in book form. It is notable because it takes an initially simplistic world consisting of two "hills" of a resource ("sugar") and simple agents, then methodically increases the complexity of the world (e.g. adding a second resource, "spice"), agents, and their behaviors (e.g. trade, warfare, disease).

The agents in Sugarscape have a much broader range of behaviors than the ones in ZI Traders. Sugarscape agents can move around a varied landscape, interact with other agents, and, in variations of the model, reproduce through "sex", trade, fight, and more. This range of behaviors and actions on a landscape that can change will require a complex set of programming rules to generate narratives that cover all of these eventualities.

For the journalist approach, there could be three different "journalists" covering three geographic regions of the model as shown in Figure 9 below using a screenshot from the

NetLogo immediate growback version of Sugarscape (Li & Wilensky, 2009) from the NetLogo model library (Wilensky 1999).

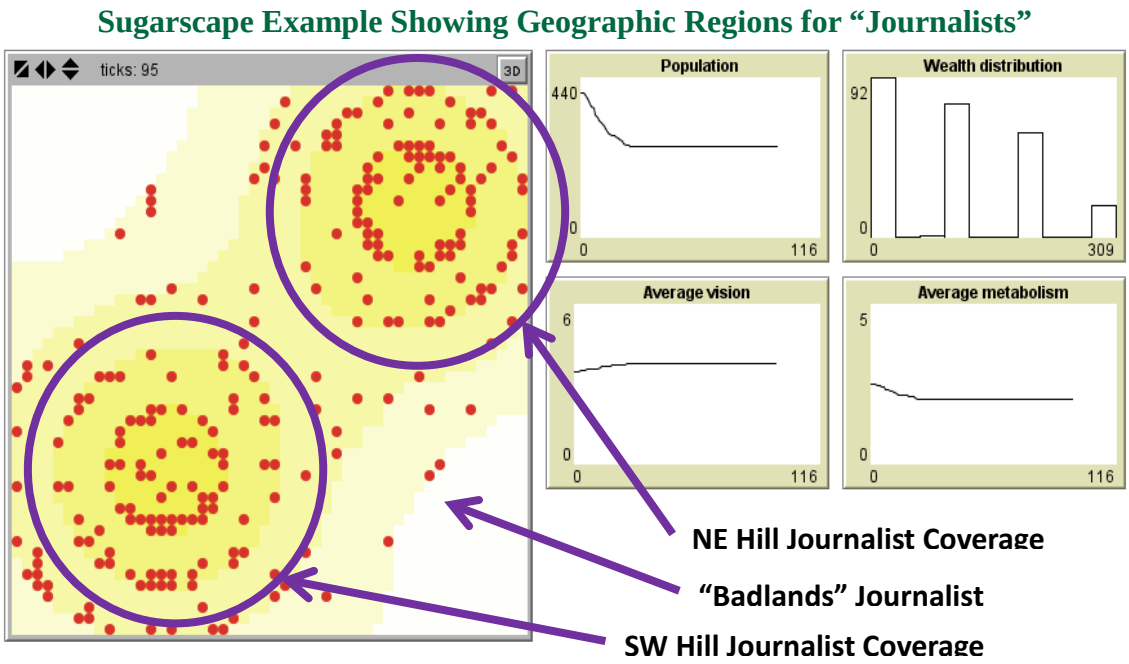

Figure 9: Sugarscape Example Showing Geographic Regions for "Journalists" (the model is from the NetLogo Model Library (Wilensky 2009, Li & Wilensky 2009))

One journalist will report on the behavior of agents and landscape for the sugar hill in the top right (northeast) corner of the model. A second will report on the other hill in the bottom left (southwest) corner. The third journalist will report on what happens everywhere else. This geographic division of journalists into these three regions provides a logical division of the agents, and we would expect different behaviors between the badlands and the hills. What will be interesting is to see if we get different narratives from the two hill journalists due to qualitative differences in the behaviors of agents each of on those hills.

The fourth approach could describe the trends in consumption of sugar over time, the movement of agents from one hill to the other and back again, and the wealth attained by the most successful agents. This is most similar to what a human would do when asked to quickly summarize a model run.

Summary

This thesis has shown that it is possible to generate useful narrative from simple agents, and that the narrative can be useful in gaining a sense of the agent-level behavior of the model. Most analysis of agent-based models focuses on aggregate behavior and looks for phase shifts or instances of radically different behavior given different parameters. However, one of the key "selling points" of agent-based modeling is that it has the information about individuals – not that it is aggregated – and methods such as reporting agent narratives allow that individual behavior to be extracted and highlighted.

There is certainly a lot more that can be done as discussed above, in this area, and I hope that this thesis provides a good starting point for me and others.

Appendix: Model Code

ZITraders.java

```
/*
 * ZITraders.java
 *
 * CSS 610
 * George Mason University
 *
 * v0 Spring 2005: basic implementation
 * v1 Summer 2009: cleaned-up and multithreaded
 *
 * Rob Axtell
 *
 * Modified by Brent Auble
 * December 2011
 * Added reporting options (agent narratives and complete transaction
history)
 *
 * October 2014 to April 2015
 * Added ability to identify most interesting agents
 *
 */

import java.util.Calendar;
import java.util.Date;
import java.util.Random;
import java.util.Arrays;
import java.text.DecimalFormat;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.text.Format;
import java.text.SimpleDateFormat;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class ZITraders {

    final double version = 4.0;

    final boolean verboseOutput = false;
    final boolean showRankingSummaryOutput = false;
```

```
    // Should the final transaction info for all buyers and sellers be
written to files?
    final boolean saveTransactions = true;
    // Should the final narratives for all buyers and sellers be
written to files?
    final boolean saveNarratives = true;
    // Where to save the output files on the computer
C:\Users\bauble\Workspace\ZITradersMultiQ
    final static String saveLocation =
"C:\\Users\\bauble\\Workspace\\ZITradersMultiQ\\output\\";

    final int numberOfBuyers = 5000;
    final int numberOfSellers = 5000;

    final static double maxBuyerValue = 100.0;
    final static double maxSellerCost = 100.0;

    final int maxNumberOfTrades = 10000;

    final boolean threadedTrades = true;
    // 850 seems to be nearly optimal for 10^6 agents total with a 10
ms sleep duration
    final int numberOfThreads = 1;
    final int sleepDuration = 10;

    /**  The number of features to be tracked for whether the agent is
"interesting",
         meaning the narrative should be looked at by a person  */
    final int numberOfInterestingFeatures = 10;
    String interestingFeatureDescriptions[] = new
String[numberOfInterestingFeatures];
    final int topN = 10; // The (minimum) number of agents to be
identified as "interesting" for each feature

    static DecimalFormat twoDigits = new DecimalFormat("0.00");

    AgentPopulation people;
    /** A common date/time stamp to identify files generated by this
run of the model */
    String dateTimeStamp;
    Data Transactions[] = new Data[numberOfThreads];
    public static double averageTransactionPrice = 0; // The average
price of all successful transactions, calculated at the end of the run

    /* Data structures and variables to store rankings of agents */
    // Counts of the number of trade attempts at the end of the model
run, used for ranking agents
    public static TreeMap<Integer, Integer> buyerTradeAttemptRanks =
new TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Integer, Integer> sellerTradeAttemptRanks =
new TreeMap<>(Collections.reverseOrder());
    public static Map<Integer, Integer> buyerTradeAttemptCountRanks =
new HashMap<>();
```

```
    public static Map<Integer, Integer> sellerTradeAttemptCountRanks =
new HashMap<>();
    public static int buyerTradeAttemptRankTopN; // The minimum number
of trade attempts that fall into the Top N for buyers
    public static int sellerTradeAttemptRankTopN; // The minimum number
of trade attempts that fall into the Top N for sellers

    // Same trade attempt rankings for those who aren't successful at
making a trade
    public static TreeMap<Integer, Integer>
buyerTradeAttemptRanksUnsucc = new
TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Integer, Integer>
sellerTradeAttemptRanksUnsucc = new
TreeMap<>(Collections.reverseOrder());
    public static Map<Integer, Integer>
buyerTradeAttemptCountRanksUnsucc = new HashMap<>();
    public static Map<Integer, Integer>
sellerTradeAttemptCountRanksUnsucc = new HashMap<>();
    public static int buyerTradeAttemptRankTopNUnsucc; // The minimum
number of trade attempts that fall into the Top N for buyers
    public static int sellerTradeAttemptRankTopNUnsucc; // The minimum
number of trade attempts that fall into the Top N for sellers

    // Successful trade prices
    public static TreeMap<Double, Integer> buyerPriceRanks = new
TreeMap<>();  // In ascending order because cheaper is better for a
buyer
    public static TreeMap<Double, Integer> sellerPriceRanks = new
TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Double, Integer> buyerPriceCountRanks = new
TreeMap<>();
    public static TreeMap<Double, Integer> sellerPriceCountRanks = new
TreeMap<>();
    public static Double buyerPriceRankTopN;
    public static Double sellerPriceRankTopN;

    // Successful offer prices
    public static TreeMap<Double, Integer> buyerOfferPriceRanks = new
TreeMap<>();  // In ascending order because cheaper is better for a
buyer
    public static TreeMap<Double, Integer> sellerOfferPriceRanks = new
TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Double, Integer> buyerOfferPriceCountRanks =
new TreeMap<>();
    public static TreeMap<Double, Integer> sellerOfferPriceCountRanks =
new TreeMap<>();
    public static Double buyerOfferPriceRankTopN;
    public static Double sellerOfferPriceRankTopN;

    // Successful trade maximum buyer and minimum seller prices
    public static TreeMap<Double, Integer> buyerMaxPriceRanks = new
TreeMap<>();  // In ascending order because cheaper is better for a
buyer
```

```
    public static TreeMap<Double, Integer> sellerMinPriceRanks = new
TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Double, Integer> buyerMaxPriceCountRanks =
new TreeMap<>();
    public static TreeMap<Double, Integer> sellerMinPriceCountRanks =
new TreeMap<>();
    public static Double buyerMaxPriceRankTopN;
    public static Double sellerMinPriceRankTopN;

    // Smallest gap between maximum buyer and minimum seller prices for
successful trades
    public static TreeMap<Double, Integer> minMaxPriceRanks = new
TreeMap<>();  // In ascending order because cheaper is better for a
buyer
    public static TreeMap<Double, Integer> minMaxPriceCountRanks = new
TreeMap<>();
    public static Double minMaxPriceRankTopN;

    // Smallest gap between buyer and seller offer prices for
successful trades
    public static TreeMap<Double, Integer> offerPriceGapRanks = new
TreeMap<>();  // In ascending order because cheaper is better for a
buyer
    public static TreeMap<Double, Integer> offerPriceGapCountRanks =
new TreeMap<>();
    public static Double offerPriceGapRankTopN;

    // Successful trade prices closest to the average price
    public static TreeMap<Double, Integer> averagePriceRanks = new
TreeMap<>();
    public static TreeMap<Double, Integer> averagePriceCountRanks = new
TreeMap<>();
    public static Double averagePriceRankTopN;

    // Successful trade maximum buyer and minimum seller prices
    public static TreeMap<Double, Integer> buyerOfferRangeRanks = new
TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Double, Integer> sellerOfferRangeRanks = new
TreeMap<>(Collections.reverseOrder());
    public static TreeMap<Double, Integer> buyerOfferRangeCountRanks =
new TreeMap<>();
    public static TreeMap<Double, Integer> sellerOfferRangeCountRanks =
new TreeMap<>();
    public static Double buyerOfferRangeRankTopN;
    public static Double sellerOfferRangeRankTopN;



    private ZITraders() {
        try {
            // Get the current date & time to append to the file names
            //String dateTimeStamp = String.format(arg0, arg1);
            Format formatter = new SimpleDateFormat("yyyy-MM-dd HHmm-
ss");
```

```
            dateTimeStamp = formatter.format(new Date());

            // Define the interesting feature descriptions
            interestingFeatureDescriptions[0] = "Most trade attempts
but ultimately successful";
            interestingFeatureDescriptions[1] = "Most trade attempts
without being successful";
            interestingFeatureDescriptions[2] = "Highest seller minimum
price that was successful";
            interestingFeatureDescriptions[3] = "Highest seller offer
price that was successful";
            interestingFeatureDescriptions[4] = "Lowest buyer maximum
price that was successful";
            interestingFeatureDescriptions[5] = "Lowest buyer offer
price that was successful";
            interestingFeatureDescriptions[6] = "Largest gap between
buyer and seller";
            interestingFeatureDescriptions[7] = "Smallest gap between
buyer and seller (most improbable trade)";
            interestingFeatureDescriptions[8] = "Trades closest to the
average/median/expected value";
            interestingFeatureDescriptions[9] = "Broadest range of
offers within a buyer or seller";

            people = new AgentPopulation(numberOfBuyers, maxBuyerValue,
                        numberOfSellers, maxSellerCost, dateTimeStamp);

            for (int i = 0; i < numberOfThreads; i++) {
                Transactions[i] = new Data();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void DoSingleThreadedTrades() {
        DoTrades tradingRun;

        tradingRun = new DoTrades(0, maxNumberOfTrades, 1,
numberOfBuyers, 1,
                    numberOfSellers);
        tradingRun.run();

    }

    public void DoMultiThreadedTrades() {

        Thread t[] = new Thread[numberOfThreads];
        boolean done = false;

        int tradesPerThread = maxNumberOfTrades / numberOfThreads;
        int buyersPerThread = numberOfBuyers / numberOfThreads;
        int sellersPerThread = numberOfSellers / numberOfThreads;
```

```
        for (int i = 0; i < numberOfThreads; i++) {
            t[i] = new Thread(new DoTrades(i, tradesPerThread, i *
buyersPerThread + 1,
                        (i + 1) * buyersPerThread, i * sellersPerThread
+ 1,
                        (i + 1) * sellersPerThread));
            t[i].start();
        }

        while (!done) {
            done = true;
            for (int i = 0; i < numberOfThreads; i++) {
                if (t[i].isAlive()) {
                    done = false;
                }
            }
            try {
                  //  This is here to keep the action on the threads!
                Thread.sleep(sleepDuration);
            } catch (InterruptedException ie) {
            }
        }
    }

    public class DoTrades implements Runnable {

        int threadID;
        int maxTrades;
        int beginBuyers;
        int endBuyers;
        int beginSellers;
        int endSellers;
        Random randomStream;

        public DoTrades(int threadNumber, int maxExchanges, int
startBuyerIndex, int endBuyerIndex, int startSellerIndex, int
endSellerIndex) {

            threadID = threadNumber;
            maxTrades = maxExchanges;
            beginBuyers = startBuyerIndex;
            endBuyers = endBuyerIndex;
            beginSellers = startSellerIndex;
            endSellers = endSellerIndex;

            randomStream = new Random();
        }

        public void run() {

            int buyerIndex, sellerIndex;
            double bidPrice, askPrice, transactionPrice;

            for (int counter = 1; counter <= maxTrades; counter++) {
```

```
                //
                // Pick a buyer at random who is still looking to
buy...
                //
                do {
                    buyerIndex = beginBuyers +
randomStream.nextInt(endBuyers - beginBuyers + 1) - 1;
                } while (people.buyers[buyerIndex].hasTraded());

                bidPrice = people.buyers[buyerIndex].formBidPrice();

                //
                // Pick a seller at random who is still looking to
sell...
                //
                do {
                    sellerIndex = beginSellers +
randomStream.nextInt(endSellers - beginSellers + 1) - 1;
                } while (people.sellers[sellerIndex].hasTraded());

                askPrice = people.sellers[sellerIndex].formAskPrice();

                double buyerValue =
people.buyers[buyerIndex].getValue();
                double sellerCost =
people.sellers[sellerIndex].getCost();

                if (bidPrice > askPrice) {
                    transactionPrice = askPrice +
randomStream.nextDouble() *
                                (bidPrice - askPrice);
                    people.buyers[buyerIndex].setTraded(true);
                    people.buyers[buyerIndex].price = transactionPrice;
                    people.sellers[sellerIndex].setTraded(true);
                    people.sellers[sellerIndex].price =
transactionPrice;
                    Transactions[threadID].AddDatuum(transactionPrice);

                    // Add record of completed trade to the buyer and
seller agents

      people.buyers[buyerIndex].addTradeAttempt(sellerIndex, bidPrice,
                                sellerCost, askPrice, transactionPrice,
true);

      people.sellers[sellerIndex].addTradeAttempt(buyerIndex, bidPrice,
                                buyerValue, askPrice, transactionPrice,
true);

                   if (verboseOutput) {
                        System.out.print(threadID +
                                    "Found two agents willing to trade:
");
                        System.out.println("Price of " +
```

```
                                    twoDigits.format(transactionPrice)
+
                                  " with buyer's bid of " +
twoDigits.format(bidPrice) +
                                  " and seller asking price of " +
                                  twoDigits.format(askPrice));
                    }
                } else {
                  // Add record of failed trade to buyer and seller
agents

      people.buyers[buyerIndex].addTradeAttempt(sellerIndex, bidPrice,
                              sellerCost, askPrice, 0, false);

      people.sellers[sellerIndex].addTradeAttempt(buyerIndex, bidPrice,
                              buyerValue, askPrice, 0, false);

                  if (verboseOutput) {
                    System.out.println(threadID + "Could not trade: " +
                              "; buyer's bid was " +
twoDigits.format(bidPrice) +
                             " while seller's asking price was " +
                              twoDigits.format(askPrice));
                  }
                }
            }
        }
    }

    private void OpenTrading() {

        System.out.println("\nZero-Intelligence traders, version " +
version +
                  ": Rob Axtell, George Mason University, with
reporting additions " +
                  "by Brent Auble");
        System.out.println("Number of Threads: " + numberOfThreads);

        long startTime = Calendar.getInstance().getTimeInMillis();

        if (!threadedTrades) {
            DoSingleThreadedTrades();
        } else {
            DoMultiThreadedTrades();
        }

        for (int i = 1; i < numberOfThreads; i++) {
            Transactions[0].CombineWithDifferentData(Transactions[i]);
        }
        averageTransactionPrice = Transactions[0].GetAverage();

        // How long the model takes to run without narrative
compilation or saving data
        long modelTime = Calendar.getInstance().getTimeInMillis();
```

```
        // Rank the agents on various characteristics in order to
identify the "most interesting" ones
        rankAgents();

        // Write out the final transactions
        if (saveTransactions) {
            saveFinalTransactions();
        }

        // Write out the final narratives
        if (saveNarratives) {
            saveFinalNarratives();
            saveInterestingNarratives();
        }

        // Count the number of buyers who had one or more interesting
features
        int interestingBuyerCount = 0;
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].getTopNRankingCount() > 0)
interestingBuyerCount++;
        }

        // Count the number of sellers who had one or more interesting
features
        int interestingSellerCount = 0;
        for (int i = 0; i < numberOfSellers; i++) {
            if (people.sellers[i].getTopNRankingCount() > 0)
interestingSellerCount++;
        }

        long finishTime = Calendar.getInstance().getTimeInMillis();

        System.out.println("\nFinal stats: " + Transactions[0].GetN() +
                  " transactions at " +
                twoDigits.format(Transactions[0].GetAverage()) +
                " average price; " +
                twoDigits.format(Transactions[0].GetStdDev()) +
                " standard deviation.\n");

        System.out.println("There were " +
                interestingBuyerCount + " interesting buyers and " +
                interestingSellerCount + " interesting sellers.\n");

        System.out.println("The model ran in " + (modelTime -
startTime) +
                  " milliseconds.");
        System.out.println("Generating and saving narrative files took
" +
                (finishTime - modelTime) + " milliseconds.");
        System.out.println("Total elapsed time: " + (finishTime -
startTime) +
                  " milliseconds\n");
```

```
    }

    // Loop through all the buyers and sellers and write out their
narratives
    private void saveFinalNarratives() {
        // Create file to store buyer agent final narratives
        FileWriter buyerNarrativeFile;
            try {
                  // Create a file to save the information about the
buyers
                  buyerNarrativeFile = new FileWriter(saveLocation +
"Buyer Narratives " +
                              dateTimeStamp + ".txt", true);
                  BufferedWriter buyerNarrativeWriter = new
BufferedWriter(buyerNarrativeFile);
                  buyerNarrativeWriter.write("Buyer Narratives");
                  buyerNarrativeWriter.newLine();
                  buyerNarrativeWriter.write("-----------------");
                  buyerNarrativeWriter.newLine();
                  buyerNarrativeWriter.newLine();

              for (int i = 0; i < numberOfBuyers; i++) {

      buyerNarrativeWriter.write(people.buyers[i].getCompleteNarrative(
));
                        buyerNarrativeWriter.newLine();
                        buyerNarrativeWriter.newLine();

      buyerNarrativeWriter.write("**********************************
***");
                        buyerNarrativeWriter.newLine();
                        buyerNarrativeWriter.newLine();
              }

              buyerNarrativeWriter.close();

            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

        // Create file to store seller agent final narratives
        FileWriter sellerNarrativeFile;
            try {
                  // Create a file to save the information about the
sellers
                  sellerNarrativeFile = new FileWriter(saveLocation +
"Seller Narratives "
                              + dateTimeStamp + ".txt", true);
                  BufferedWriter sellerNarrativeWriter = new
BufferedWriter(sellerNarrativeFile);
                  sellerNarrativeWriter.write("Seller Narratives");
                  sellerNarrativeWriter.newLine();
                  sellerNarrativeWriter.write("------------------");
```

```
                    sellerNarrativeWriter.newLine();
                    sellerNarrativeWriter.newLine();

                for (int i = 0; i < numberOfSellers; i++) {

      sellerNarrativeWriter.write(people.sellers[i].getCompleteNarrativ
e());
                        sellerNarrativeWriter.newLine();
                        sellerNarrativeWriter.newLine();

      sellerNarrativeWriter.write("**********************************
****");
                        sellerNarrativeWriter.newLine();
                        sellerNarrativeWriter.newLine();
                }

                sellerNarrativeWriter.close();

            } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
    }

    /* Loop through the buyers and sellers who have at least one Top N
     * "interesting" feature and write out their narratives */
    private void saveInterestingNarratives() {
        // Create file to store buyer agent interesting narratives
        FileWriter buyerNarrativeFile;
            try {
                    // Create a file to save the information about the
buyers
                    buyerNarrativeFile = new FileWriter(saveLocation +
"Interesting Buyer Narratives " +
                                dateTimeStamp + ".txt", true);
                    BufferedWriter buyerNarrativeWriter = new
BufferedWriter(buyerNarrativeFile);
                    buyerNarrativeWriter.write("Interesting Buyer
Narratives");
                    buyerNarrativeWriter.newLine();
                    buyerNarrativeWriter.write("----------------------
---");
                    buyerNarrativeWriter.newLine();
                    buyerNarrativeWriter.newLine();

            // Sort the buyers so the ones with the most interesting
features are first
            Arrays.sort(people.buyers, new
BuyerAgent.OrderByTopNRankings());

              for (int i = 0; i < numberOfBuyers; i++) {
                if (people.buyers[i].getTopNRankingCount() > 0) {

buyerNarrativeWriter.write(people.buyers[i].getCompleteNarrative());
```

```
                    buyerNarrativeWriter.newLine();
                    buyerNarrativeWriter.newLine();

buyerNarrativeWriter.write("**************************************");
                    buyerNarrativeWriter.newLine();
                    buyerNarrativeWriter.newLine();
                }
            }

            buyerNarrativeWriter.close();

        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

      // Create file to store seller agent interesting narratives
      FileWriter sellerNarrativeFile;
          try {
                // Create a file to save the information about the
sellers
                sellerNarrativeFile = new FileWriter(saveLocation +
"Interesting Seller Narratives "
                            + dateTimeStamp + ".txt", true);
                BufferedWriter sellerNarrativeWriter = new
BufferedWriter(sellerNarrativeFile);
                sellerNarrativeWriter.write("Interesting Seller
Narratives");
                sellerNarrativeWriter.newLine();
                sellerNarrativeWriter.write("----------------------
-----");
                sellerNarrativeWriter.newLine();
                sellerNarrativeWriter.newLine();

          // Sort the sellers so the ones with the most interesting
features are first
          Arrays.sort(people.sellers, new
SellerAgent.OrderByTopNRankings());

            for (int i = 0; i < numberOfSellers; i++) {
              if (people.sellers[i].getTopNRankingCount() > 0) {

sellerNarrativeWriter.write(people.sellers[i].getCompleteNarrative());
                    sellerNarrativeWriter.newLine();
                    sellerNarrativeWriter.newLine();

sellerNarrativeWriter.write("**************************************")
;
                    sellerNarrativeWriter.newLine();
                    sellerNarrativeWriter.newLine();
                }
            }

            sellerNarrativeWriter.close();
```

```
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
    }

    // Loop through all the buyers and sellers and write out all the
transactions
    private void saveFinalTransactions() {
        // Create file to store buyer agent transactions
        FileWriter buyerTransactionFile;
            try {
                  // Create a file to save the information about the
buyers
                  buyerTransactionFile = new FileWriter(saveLocation +
"Buyer Transactions "
                              + dateTimeStamp + ".csv", true);
                  BufferedWriter buyerTransWriter = new
BufferedWriter(buyerTransactionFile);

      buyerTransWriter.write("buyerID,attemptNum,sellerID,buyerValue,bu
yerOffer,sellerValue,sellerOffer,salePrice,tradeCompleted,timeStamp,nar
rative");
                  buyerTransWriter.newLine();

              for (int i = 0; i < numberOfBuyers; i++) {

      buyerTransWriter.write(people.buyers[i].getCompleteTransactionHis
toryCSV(true));
                }

                buyerTransWriter.close();

            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

            // Create file to store seller agent transactions
        FileWriter sellerTransactionFile;
            try {
                  // Create a file to save the information about the
sellers
                  sellerTransactionFile = new FileWriter(saveLocation +
"Seller Transactions "
                              + dateTimeStamp + ".csv", true);
                  BufferedWriter sellerTransWriter = new
BufferedWriter(sellerTransactionFile);

      sellerTransWriter.write("sellerID,attemptNum,buyerID,buyerValue,b
uyerOffer,sellerValue,sellerOffer,salePrice,tradeCompleted,timeStamp,na
rrative");
                  sellerTransWriter.newLine();
```

```
                for (int i = 0; i < numberOfBuyers; i++) {

      sellerTransWriter.write(people.sellers[i].getCompleteTransactionH
istoryCSV());
                }

                sellerTransWriter.close();

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        // Create file to store completed transactions
        FileWriter completedTransactionFile;
            try {
                // Create a file to save the information about
completed transactions
                completedTransactionFile = new
FileWriter(saveLocation + "Completed Transactions "
                            + dateTimeStamp + ".csv", true);
                BufferedWriter completedTransWriter = new
BufferedWriter(completedTransactionFile);

      completedTransWriter.write("buyerID,sellerID,buyerValue,buyerOffe
r,sellerValue,sellerOffer,salePrice,tradeCompleted,timeStamp");
                completedTransWriter.newLine();

                for (int i = 0; i < numberOfBuyers; i++) {
                if (people.buyers[i].hasTraded()) {

      completedTransWriter.write(people.buyers[i].getCompletedTransacti
onHistoryCSV());
                }
                }

                completedTransWriter.close();

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        if (false) {  // Not used right now, but here as an example
        // Create file to store completed transactions
        FileWriter fileBuyerTradeAttemptSorted;
            try {
                // Create a file to save the information about
completed transactions
                fileBuyerTradeAttemptSorted = new
FileWriter(saveLocation + "Buyer Transaction Attempt Ranking "
                            + dateTimeStamp + ".csv", true);
```

```
                BufferedWriter btaWriter = new
BufferedWriter(fileBuyerTradeAttemptSorted);
                btaWriter.write("buyerID,numAttempts,btaOrder,i");
                btaWriter.newLine();

            Arrays.sort(people.buyers, new
BuyerAgent.OrderByTradeAttempts());

            int n = Transactions[0].GetN();
              for (int i = 0; i < numberOfBuyers; i++) {
                  if (people.buyers[i].hasTraded()) {
                    n--;
                    StringBuffer sb = new StringBuffer();

sb.append(people.buyers[i].getBuyerID()).append(",");

sb.append(people.buyers[i].getTradeAttemptCount()).append(",");
                    sb.append(n).append(",");
                    sb.append(i);
                        btaWriter.write(sb.toString());
                    btaWriter.newLine();
                  }
              }

              btaWriter.close();

            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
        }

    }

    /* Save the transaction narratives of the agents most worth
reviewing manually to separate files
     * Only run at the completion of the model since it doesn't handle
threads
     *
     * Identifying Interesting Agents in ZI Traders:

                0) Most trade attempts but ultimately successful
                1) Most trade attempts without being successful
                2) Highest seller minimum price that was successful
                3) Highest seller offer price that was successful
                4) Lowest buyer maximum price that was successful
                5) Lowest buyer offer price that was successful
                6) Largest gap between buyer and seller
                7) Smallest gap between buyer and seller (most
improbable trade)
                8) Trades closest to the average/median/expected
value
                9) Broadest range of offers within a buyer or seller
     *
```

```
     */
    public void saveMostInterestingAgents() {

    }

    public void rankAgents() {
        // For buyers, count the number of agents for each number of
trade attempts
        for (int i = 0; i < numberOfBuyers; i++) {
            int nAttempts = people.buyers[i].getTradeAttemptCount();
            if (people.buyers[i].hasTraded()) {
                if (buyerTradeAttemptRanks.containsKey(nAttempts)) {
                    buyerTradeAttemptRanks.put(nAttempts,
                            buyerTradeAttemptRanks.get(nAttempts)+1);
                } else {
                    buyerTradeAttemptRanks.put(nAttempts, 1);
                }
            } else {
                if
(buyerTradeAttemptRanksUnsucc.containsKey(nAttempts)) {
                    buyerTradeAttemptRanksUnsucc.put(nAttempts,

buyerTradeAttemptRanksUnsucc.get(nAttempts)+1);
                } else {
                    buyerTradeAttemptRanksUnsucc.put(nAttempts, 1);
                }
            }
        }

        // Rank the counts of trade attempts from most to least
        int buyerAttemptCount = 1; //buyerTradeAttemptRanks.size();  //
Originally used before I sorted the TreeMap in reverse
        int buyerTopNTotal = 0;
        for (int key : buyerTradeAttemptRanks.keySet()) {
            buyerTradeAttemptCountRanks.put(key, buyerAttemptCount);
            buyerAttemptCount++; //buyerAttemptCount--;

            // Figure out which trade attempt count falls into the top
N responses
            if (buyerTopNTotal == 0) { // First time through only
                buyerTopNTotal = buyerTradeAttemptRanks.get(key);
                buyerTradeAttemptRankTopN = key;
                //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
            }  else {
                if (buyerTopNTotal < topN) {
                    buyerTopNTotal = buyerTopNTotal +
buyerTradeAttemptRanks.get(key);
                    buyerTradeAttemptRankTopN = key;
                    //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
                }
            }
        }
```

```
        // Rank the counts of unsuccessful trade attempts from most to
least
        buyerAttemptCount = 1; // Reset for unsuccessful trade attempts
        int buyerTopNTotalUnsucc = 0;
        for (int key : buyerTradeAttemptRanksUnsucc.keySet()) {
            buyerTradeAttemptCountRanksUnsucc.put(key,
buyerAttemptCount);
            buyerAttemptCount++; //buyerAttemptCount--;

            // Figure out which trade attempt count falls into the top
N responses
            if (buyerTopNTotalUnsucc == 0) { // First time through only
                buyerTopNTotalUnsucc =
buyerTradeAttemptRanksUnsucc.get(key);
                buyerTradeAttemptRankTopNUnsucc = key;
                //System.out.println("TEST Buyer Top N Unsucc: " +
buyerTradeAttemptRankTopNUnsucc + " (" + buyerTopNTotalUnsucc + ")");
            }  else {
                if (buyerTopNTotalUnsucc < topN) {
                    buyerTopNTotalUnsucc = buyerTopNTotalUnsucc +
buyerTradeAttemptRanksUnsucc.get(key);
                    buyerTradeAttemptRankTopNUnsucc = key;
                    //System.out.println("TEST Buyer Top N Unsucc: " +
buyerTradeAttemptRankTopNUnsucc + " (" + buyerTopNTotalUnsucc + ")");
                }
            }
        }

        if (showRankingSummaryOutput) {
            System.out.println();
            System.out.println("Numbers of Buyer Trade Attempts");
            for (int i: buyerTradeAttemptRanks.keySet()) {
                System.out.println(i + ":" +
buyerTradeAttemptRanks.get(i));
            }

            System.out.println();
            System.out.println("Buyer Top N: " +
buyerTradeAttemptRankTopN
                    + " (" + buyerTopNTotal + ")");

            System.out.println();
            System.out.println("Numbers of Buyer Trade Attempts Rank
Order");
            for (int i: buyerTradeAttemptCountRanks.keySet()) {
                System.out.println(i + ":" +
buyerTradeAttemptCountRanks.get(i));
            }

            System.out.println();
            System.out.println("Numbers of Unsuccessful Buyer Trade
Attempts");
            for (int i: buyerTradeAttemptRanksUnsucc.keySet()) {
```

```
                System.out.println(i + ":" +
buyerTradeAttemptRanksUnsucc.get(i));
            }

            System.out.println();
            System.out.println("Unsuccessful Buyer Top N: " +
buyerTradeAttemptRankTopNUnsucc
                    + " (" + buyerTopNTotalUnsucc + ")");

            System.out.println();
            System.out.println("Numbers of Unsuccessful Buyer Trade
Attempts Rank Order");
            for (int i: buyerTradeAttemptCountRanksUnsucc.keySet()) {
                System.out.println(i + ":" +
buyerTradeAttemptCountRanksUnsucc.get(i));
            }
        }

        // rank sellers with the most trade attempts
        for (int i = 0; i < numberOfSellers; i++) {
            int nAttempts = people.sellers[i].getTradeAttemptCount();
            if (people.sellers[i].hasTraded()) {
                if (sellerTradeAttemptRanks.containsKey(nAttempts)) {
                    sellerTradeAttemptRanks.put(nAttempts,
                        sellerTradeAttemptRanks.get(nAttempts)+1);
                } else {
                    sellerTradeAttemptRanks.put(nAttempts, 1);
                }
            } else {
                if
(sellerTradeAttemptRanksUnsucc.containsKey(nAttempts)) {
                    sellerTradeAttemptRanksUnsucc.put(nAttempts,

sellerTradeAttemptRanksUnsucc.get(nAttempts)+1);
                } else {
                    sellerTradeAttemptRanksUnsucc.put(nAttempts, 1);
                }
            }
        }

        // Rank the counts of trade attempts from most to least
        int sellerAttemptCount = 1; //sellerTradeAttemptRanks.size();
        int sellerTopNTotal = 0;
        for (int key : sellerTradeAttemptRanks.keySet()) {
            sellerTradeAttemptCountRanks.put(key, sellerAttemptCount);
            sellerAttemptCount++; //--;

            // Figure out which trade attempt count falls into the top
N responses
            if (sellerTopNTotal == 0) { // First time through only
                sellerTopNTotal = sellerTradeAttemptRanks.get(key);
                sellerTradeAttemptRankTopN = key;
                //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
```

```
            }  else {
                if (sellerTopNTotal < topN) {
                    sellerTopNTotal = sellerTopNTotal +
sellerTradeAttemptRanks.get(key);
                    sellerTradeAttemptRankTopN = key;
                    //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
                }
            }
        }

        // Rank the counts of unsuccessful trade attempts from most to
least
        sellerAttemptCount = 1; // Reset for unsuccessful trade
attempts
        int sellerTopNTotalUnsucc = 0;
        for (int key : sellerTradeAttemptRanksUnsucc.keySet()) {
            sellerTradeAttemptCountRanksUnsucc.put(key,
sellerAttemptCount);
            sellerAttemptCount++; //sellerAttemptCount--;

            // Figure out which trade attempt count falls into the top
N responses
            if (sellerTopNTotalUnsucc == 0) { // First time through
only
                sellerTopNTotalUnsucc =
sellerTradeAttemptRanksUnsucc.get(key);
                sellerTradeAttemptRankTopNUnsucc = key;
                //System.out.println("TEST Buyer Top N Unsucc: " +
sellerTradeAttemptRankTopNUnsucc + " (" + sellerTopNTotalUnsucc + ")");
            }  else {
                if (sellerTopNTotalUnsucc < topN) {
                    sellerTopNTotalUnsucc = sellerTopNTotalUnsucc +
sellerTradeAttemptRanksUnsucc.get(key);
                    sellerTradeAttemptRankTopNUnsucc = key;
                    //System.out.println("TEST Buyer Top N Unsucc: " +
sellerTradeAttemptRankTopNUnsucc + " (" + sellerTopNTotalUnsucc + ")");
                }
            }
        }

        if (showRankingSummaryOutput) {
            System.out.println();
            System.out.println("Numbers of Seller Trade Attempts");
            for (int i: sellerTradeAttemptRanks.keySet()) {
                System.out.println(i + ":" +
sellerTradeAttemptRanks.get(i));
            }

            System.out.println();
            System.out.println("Seller Top N: " +
sellerTradeAttemptRankTopN
                    + " (" + sellerTopNTotal + ")");
```

```
            System.out.println();
            System.out.println("Numbers of Seller Trade Attempts Rank
Order");
            for (int i: sellerTradeAttemptCountRanks.keySet()) {
                System.out.println(i + ":" +
sellerTradeAttemptCountRanks.get(i));
            }

            System.out.println();
            System.out.println("Numbers of Unsuccessful Seller Trade
Attempts");
            for (int i: sellerTradeAttemptRanksUnsucc.keySet()) {
                System.out.println(i + ":" +
sellerTradeAttemptRanksUnsucc.get(i));
            }

            System.out.println();
            System.out.println("Unsuccessful Seller Top N: " +
sellerTradeAttemptRankTopNUnsucc
                    + " (" + sellerTopNTotalUnsucc + ")");

            System.out.println();
            System.out.println("Numbers of Unsuccessful Seller Trade
Attempts Rank Order");
            for (int i: sellerTradeAttemptCountRanksUnsucc.keySet()) {
                System.out.println(i + ":" +
sellerTradeAttemptCountRanksUnsucc.get(i));
            }
        }

        // Rank Successful Trade Prices
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                int nAttempts =
people.buyers[i].getTradeAttemptCount();
                Double tradePrice =
people.buyers[i].tradeRecord.get(nAttempts - 1).getSalePrice();
                if (buyerPriceRanks.containsKey(tradePrice)) {
                    buyerPriceRanks.put(tradePrice,
                            buyerPriceRanks.get(tradePrice)+1);
                } else {
                    buyerPriceRanks.put(tradePrice, 1);
                }
            }
        }

        // Rank the counts of completed trade prices from least to most
        int buyerRankCount = 1;
        buyerTopNTotal = 0;
        for (Double key : buyerPriceRanks.keySet()) {
            buyerPriceCountRanks.put(key, buyerRankCount);
            buyerRankCount++;
```

```
            // Figure out which trade attempt count falls into the top
N responses
            if (buyerTopNTotal == 0) { // First time through only
                buyerTopNTotal = buyerPriceRanks.get(key);
                buyerPriceRankTopN = key;
                //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
            }  else {
                if (buyerTopNTotal < topN) {
                    buyerTopNTotal = buyerTopNTotal +
buyerPriceRanks.get(key);
                    buyerPriceRankTopN = key;
                    //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
                }
            }
        }

        if (showRankingSummaryOutput) {
            System.out.println();
            System.out.println("Buyer Successful Prices");
            int nn = 0;
            for (Double i: buyerPriceRanks.keySet()) {
                System.out.println(i + ":" + buyerPriceRanks.get(i));

                nn++;
                if (nn == 20) break;
            }

            System.out.println();
            System.out.println("Buyer Top N Successful Price: " +
buyerPriceRankTopN
                    + " (" + buyerTopNTotal + ")");

            System.out.println();
            System.out.println("Buyer Successful Prices Rank Order");
            nn = 0;
            for (Double i: buyerPriceCountRanks.keySet()) {
                System.out.println(i + ":" +
buyerPriceCountRanks.get(i));

                nn++;
                if (nn == 20) break;
            }
        }

        /* Rank Successful Trade Prices for Sellers
         * (NOTE: this is identical to the one for Buyers, but I'm
keeping it
         * in for easier understandability) */
        for (int i = 0; i < numberOfSellers; i++) {
            if (people.sellers[i].hasTraded()) {
                int nAttempts =
people.sellers[i].getTradeAttemptCount();
```

```
                Double tradePrice =
people.sellers[i].tradeRecord.get(nAttempts - 1).getSalePrice();
                if (sellerPriceRanks.containsKey(tradePrice)) {
                    sellerPriceRanks.put(tradePrice,
                            sellerPriceRanks.get(tradePrice)+1);
                } else {
                    sellerPriceRanks.put(tradePrice, 1);
                }
            }
        }

        // Rank the counts of completed trade prices from most to least
        int sellerRankCount = 1;
        sellerTopNTotal = 0;
        for (Double key : sellerPriceRanks.keySet()) {
            sellerPriceCountRanks.put(key, sellerRankCount);
            sellerRankCount++;

            // Figure out which trade attempt count falls into the top
N responses
            if (sellerTopNTotal == 0) { // First time through only
                sellerTopNTotal = sellerPriceRanks.get(key);
                sellerPriceRankTopN = key;
                //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
            }  else {
                if (sellerTopNTotal < topN) {
                    sellerTopNTotal = sellerTopNTotal +
sellerPriceRanks.get(key);
                    sellerPriceRankTopN = key;
                    //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
                }
            }
        }

        // Rank Successful Trade Buyer Offer Prices
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                int nAttempts =
people.buyers[i].getTradeAttemptCount();
                Double tradePrice =
people.buyers[i].tradeRecord.get(nAttempts - 1).getBuyerOffer();
                if (buyerOfferPriceRanks.containsKey(tradePrice)) {
                    buyerOfferPriceRanks.put(tradePrice,
                            buyerOfferPriceRanks.get(tradePrice)+1);
                } else {
                    buyerOfferPriceRanks.put(tradePrice, 1);
                }
            }
        }

        // Rank the counts of completed trade buyer offer prices from
least to most
```

```
        buyerRankCount = 1;
        buyerTopNTotal = 0;
        for (Double key : buyerOfferPriceRanks.keySet()) {
            buyerOfferPriceCountRanks.put(key, buyerRankCount);
            buyerRankCount++;

            // Figure out which trade attempt count falls into the top
N responses
            if (buyerTopNTotal == 0) { // First time through only
                buyerTopNTotal = buyerOfferPriceRanks.get(key);
                buyerOfferPriceRankTopN = key;
                //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
            }  else {
                if (buyerTopNTotal < topN) {
                    buyerTopNTotal = buyerTopNTotal +
buyerOfferPriceRanks.get(key);
                    buyerOfferPriceRankTopN = key;
                    //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
                }
            }
        }

        // Rank Successful Trade Seller Offer Prices
        for (int i = 0; i < numberOfSellers; i++) {
            if (people.sellers[i].hasTraded()) {
                int nAttempts =
people.sellers[i].getTradeAttemptCount();
                Double tradePrice =
people.sellers[i].tradeRecord.get(nAttempts - 1).getSellerOffer();
                if (sellerOfferPriceRanks.containsKey(tradePrice)) {
                    sellerOfferPriceRanks.put(tradePrice,
                            sellerOfferPriceRanks.get(tradePrice)+1);
                } else {
                    sellerOfferPriceRanks.put(tradePrice, 1);
                }
            }
        }

        // Rank the counts of completed trade seller offer prices from
most to least
        sellerRankCount = 1;
        sellerTopNTotal = 0;
        for (Double key : sellerOfferPriceRanks.keySet()) {
            sellerOfferPriceCountRanks.put(key, sellerRankCount);
            sellerRankCount++;

            // Figure out which trade attempt count falls into the top
N responses
            if (sellerTopNTotal == 0) { // First time through only
                sellerTopNTotal = sellerOfferPriceRanks.get(key);
                sellerOfferPriceRankTopN = key;
```

```
                //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
            }  else {
                if (sellerTopNTotal < topN) {
                    sellerTopNTotal = sellerTopNTotal +
sellerOfferPriceRanks.get(key);
                    sellerOfferPriceRankTopN = key;
                    //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
                }
            }
        }

        // Rank Successful Trade Buyer Maximum Offer Prices (Value)
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                Double tradePrice = people.buyers[i].getValue();
                if (buyerMaxPriceRanks.containsKey(tradePrice)) {
                    buyerMaxPriceRanks.put(tradePrice,
                            buyerMaxPriceRanks.get(tradePrice)+1);
                } else {
                    buyerMaxPriceRanks.put(tradePrice, 1);
                }
            }
        }

        // Rank the counts of completed trade buyer offer prices from
least to most
        buyerRankCount = 1;
        buyerTopNTotal = 0;
        for (Double key : buyerMaxPriceRanks.keySet()) {
            buyerMaxPriceCountRanks.put(key, buyerRankCount);
            buyerRankCount++;

            // Figure out which trade attempt count falls into the top
N responses
            if (buyerTopNTotal == 0) { // First time through only
                buyerTopNTotal = buyerMaxPriceRanks.get(key);
                buyerMaxPriceRankTopN = key;
                //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
            }  else {
                if (buyerTopNTotal < topN) {
                    buyerTopNTotal = buyerTopNTotal +
buyerMaxPriceRanks.get(key);
                    buyerMaxPriceRankTopN = key;
                    //System.out.println("TEST Buyer Top N: " +
buyerTradeAttemptRankTopN + " (" + buyerTopNTotal + ")");
                }
            }
        }

        // Rank Successful Trade Seller Minimum Offer Prices (Cost)
        for (int i = 0; i < numberOfSellers; i++) {
```

```
            if (people.sellers[i].hasTraded()) {
                Double tradePrice = people.sellers[i].getCost();
                if (sellerMinPriceRanks.containsKey(tradePrice)) {
                    sellerMinPriceRanks.put(tradePrice,
                            sellerMinPriceRanks.get(tradePrice)+1);
                } else {
                    sellerMinPriceRanks.put(tradePrice, 1);
                }
            }
        }

        // Rank the counts of completed trade seller offer prices from
most to least
        sellerRankCount = 1;
        sellerTopNTotal = 0;
        for (Double key : sellerMinPriceRanks.keySet()) {
            sellerMinPriceCountRanks.put(key, sellerRankCount);
            sellerRankCount++;

            // Figure out which trade attempt count falls into the top
N responses
            if (sellerTopNTotal == 0) { // First time through only
                sellerTopNTotal = sellerMinPriceRanks.get(key);
                sellerMinPriceRankTopN = key;
                //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
            }  else {
                if (sellerTopNTotal < topN) {
                    sellerTopNTotal = sellerTopNTotal +
sellerMinPriceRanks.get(key);
                    sellerMinPriceRankTopN = key;
                    //System.out.println("TEST Seller Top N: " +
sellerTradeAttemptRankTopN + " (" + sellerTopNTotal + ")");
                }
            }
        }

        /* For Successful Trades, Rank the Difference Between Buyer
Maximum
         * Offer Price (Value) and Seller Minimum Offer Price (Cost),
with
         * the smallest gaps being the most interesting */
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                int nAttempts =
people.buyers[i].getTradeAttemptCount();
                int sellerID =
people.buyers[i].tradeRecord.get(nAttempts - 1).getSellerID();
                Double sellerCost = people.sellers[sellerID].getCost();
                Double tradePriceDifference =
people.buyers[i].getValue() - sellerCost;
                if (minMaxPriceRanks.containsKey(tradePriceDifference))
{
                    minMaxPriceRanks.put(tradePriceDifference,
```

```
minMaxPriceRanks.get(tradePriceDifference)+1);
                } else {
                    minMaxPriceRanks.put(tradePriceDifference, 1);
                }
            }
        }

        /* Rank the counts of completed trade buyer max offer price and
         * seller min offer price differences from least to most */
        int rankCount = 1;
        int topNTotal = 0;
        for (Double key : minMaxPriceRanks.keySet()) {
            minMaxPriceCountRanks.put(key, rankCount);
            rankCount++;

            // Figure out which trade buyer max and seller min
difference falls into the top N responses
            if (topNTotal == 0) { // First time through only
                topNTotal = minMaxPriceRanks.get(key);
                minMaxPriceRankTopN = key;
                //System.out.println("TEST Min Max Diff Top N: " +
minMaxPriceRankTopN + " (" + topNTotal + ")");
            }  else {
                if (topNTotal < topN) {
                    topNTotal = topNTotal + minMaxPriceRanks.get(key);
                    minMaxPriceRankTopN = key;
                    //System.out.println("TEST Min Max Diff Top N: " +
minMaxPriceRankTopN + " (" + topNTotal + ")");
                }
            }
        }

        /* For Successful Trades, Rank the Difference Between Buyer
Offer
         * Price and Seller Offer Price, with the smallest gaps being
the
         * most interesting */
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                int nAttempts =
people.buyers[i].getTradeAttemptCount();
                Double buyerOffer =
people.buyers[i].tradeRecord.get(nAttempts - 1).getBuyerOffer();
                Double sellerOffer =
people.buyers[i].tradeRecord.get(nAttempts - 1).getSellerOffer();
                Double tradePriceDifference = buyerOffer - sellerOffer;
                if
(offerPriceGapRanks.containsKey(tradePriceDifference)) {
                    offerPriceGapRanks.put(tradePriceDifference,

offerPriceGapRanks.get(tradePriceDifference)+1);
                } else {
                    offerPriceGapRanks.put(tradePriceDifference, 1);
```

```
                }
            }
        }

        /* Rank the counts of completed trade buyer max offer price and
         * seller min offer price differences from least to most */
        rankCount = 1;
        topNTotal = 0;
        for (Double key : offerPriceGapRanks.keySet()) {
            offerPriceGapCountRanks.put(key, rankCount);
            rankCount++;

            // Figure out which trade offer difference falls into the
top N responses
            if (topNTotal == 0) { // First time through only
                topNTotal = offerPriceGapRanks.get(key);
                offerPriceGapRankTopN = key;
                //System.out.println("TEST Offer Diff Top N: " +
offerPriceGapRankTopN + " (" + topNTotal + ")");
            }  else {
                if (topNTotal < topN) {
                    topNTotal = topNTotal +
offerPriceGapRanks.get(key);
                    offerPriceGapRankTopN = key;
                    //System.out.println("TEST Offer Diff Top N: " +
offerPriceGapRankTopN + " (" + topNTotal + ")");
                }
            }
        }

        if (showRankingSummaryOutput) {
            System.out.println();
            System.out.println("Buyer and Seller Offer Price Gaps");
            int nn = 0;
            for (Double i: offerPriceGapRanks.keySet()) {
                System.out.println(i + ":" +
offerPriceGapRanks.get(i));

                nn++;
                if (nn == 20) break;
            }

            System.out.println();
            System.out.println("Buyer and Seller Offer Price Gaps Top N
Price: " + offerPriceGapRankTopN
                    + " (" + topNTotal + ")");

            System.out.println();
            System.out.println("Buyer and Seller Offer Price Gaps Rank
Order");
            nn = 0;
            for (Double i: offerPriceGapCountRanks.keySet()) {
                System.out.println(i + ":" +
offerPriceGapCountRanks.get(i));
```

```
                nn++;
                if (nn == 20) break;
            }
        }

        //averagePrice
        /* For Successful Trades, rank the trade prices closest to the
         * average, with the closest price being the most interesting
*/
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                int nAttempts =
people.buyers[i].getTradeAttemptCount();
                Double tradePrice =
people.buyers[i].tradeRecord.get(nAttempts - 1).getSalePrice();
                Double tradePriceDifference =
Math.abs(averageTransactionPrice - tradePrice);
                if
(averagePriceRanks.containsKey(tradePriceDifference)) {
                    averagePriceRanks.put(tradePriceDifference,

averagePriceRanks.get(tradePriceDifference)+1);
                } else {
                    averagePriceRanks.put(tradePriceDifference, 1);
                }
            }
        }

        /* Rank the trades whose trade price is closest to the average
*/
        rankCount = 1;
        topNTotal = 0;
        for (Double key : averagePriceRanks.keySet()) {
            averagePriceCountRanks.put(key, rankCount);
            rankCount++;

            // Figure out which trade offer difference falls into the
top N responses
            if (topNTotal == 0) { // First time through only
                topNTotal = averagePriceRanks.get(key);
                averagePriceRankTopN = key;
                //System.out.println("TEST Average Price Diff Top N: "
+ averagePriceRankTopN + " (" + topNTotal + ")");
            }  else {
                if (topNTotal < topN) {
                    topNTotal = topNTotal + averagePriceRanks.get(key);
                    averagePriceRankTopN = key;
                    //System.out.println("TEST Average Price Diff Top
N: " + averagePriceRankTopN + " (" + topNTotal + ")");
                }
            }
        }
```

```
        if (showRankingSummaryOutput) {
            System.out.println();
            System.out.println("Average Price Closest Difference");
            int nn = 0;
            for (Double i: averagePriceRanks.keySet()) {
                System.out.println(i + ":" + averagePriceRanks.get(i));

                nn++;
                if (nn == 20) break;
            }

            System.out.println();
            System.out.println("Average Price Difference Top N: " +
averagePriceRankTopN
                    + " (" + topNTotal + ")");

            System.out.println();
            System.out.println("Average Price Difference Rank Order");
            nn = 0;
            for (Double i: averagePriceCountRanks.keySet()) {
                System.out.println(i + ":" +
averagePriceCountRanks.get(i));

                nn++;
                if (nn == 20) break;
            }
        }

        //buyerOfferRangeRanks
        /* For Buyers and Sellers, rank the range of prices they
offered in
         * trade attempts, with the largest price range being the most
         * interesting */
        // Rank Successful Trade Buyer Offer Ranges
        for (int i = 0; i < numberOfBuyers; i++) {
            if (people.buyers[i].hasTraded()) {
                Double tradeRange = people.buyers[i].getOfferRange();
                if (buyerOfferRangeRanks.containsKey(tradeRange)) {
                    buyerOfferRangeRanks.put(tradeRange,
                            buyerOfferRangeRanks.get(tradeRange)+1);
                } else {
                    buyerOfferRangeRanks.put(tradeRange, 1);
                }
            }
        }

        // Rank the counts of buyer offer ranges from most to least
        buyerRankCount = 1;
        buyerTopNTotal = 0;
        for (Double key : buyerOfferRangeRanks.keySet()) {
            buyerOfferRangeCountRanks.put(key, buyerRankCount);
            buyerRankCount++;
```

```
            // Figure out which range count falls into the top N
responses
            if (buyerTopNTotal == 0) { // First time through only
                buyerTopNTotal = buyerOfferRangeRanks.get(key);
                buyerOfferRangeRankTopN = key;
                //System.out.println("TEST Buyer Top N: " +
buyerOfferRangeRankTopN + " (" + buyerTopNTotal + ")");
            }  else {
                if (buyerTopNTotal < topN) {
                    buyerTopNTotal = buyerTopNTotal +
buyerOfferRangeRanks.get(key);
                    buyerOfferRangeRankTopN = key;
                    //System.out.println("TEST Buyer Top N: " +
buyerOfferRangeRankTopN + " (" + buyerTopNTotal + ")");
                }
            }
        }

        // Rank Seller Offer Ranges
        for (int i = 0; i < numberOfSellers; i++) {
            if (people.sellers[i].hasTraded()) {
                Double tradeRange = people.sellers[i].getOfferRange();
                if (sellerOfferRangeRanks.containsKey(tradeRange)) {
                    sellerOfferRangeRanks.put(tradeRange,
                            sellerOfferRangeRanks.get(tradeRange)+1);
                } else {
                    sellerOfferRangeRanks.put(tradeRange, 1);
                }
            }
        }

        // Rank the counts of seller offer ranges from most to least
        sellerRankCount = 1;
        sellerTopNTotal = 0;
        for (Double key : sellerOfferRangeRanks.keySet()) {
            sellerOfferRangeCountRanks.put(key, sellerRankCount);
            sellerRankCount++;

            // Figure out which range count falls into the top N
responses
            if (sellerTopNTotal == 0) { // First time through only
                sellerTopNTotal = sellerOfferRangeRanks.get(key);
                sellerOfferRangeRankTopN = key;
                //System.out.println("TEST Seller Top N: " +
sellerOfferRangeRankTopN + " (" + sellerTopNTotal + ")");
            }  else {
                if (sellerTopNTotal < topN) {
                    sellerTopNTotal = sellerTopNTotal +
sellerOfferRangeRanks.get(key);
                    sellerOfferRangeRankTopN = key;
                    //System.out.println("TEST Seller Top N: " +
sellerOfferRangeRankTopN + " (" + sellerTopNTotal + ")");
                }
            }
```

```
        }


    }

    public static String getOrdinalNumberSuffix(int theNumber) {
        // Adapted from
http://www.javalobby.org/java/forums/t16906.html
        int hundredRemainder = theNumber % 100;
        int tenRemainder = theNumber % 10;
        if ((hundredRemainder - tenRemainder) == 10) {  // Teens
            return "th";
        }

        switch (tenRemainder) {
            case 1:
                return "st";
            case 2:
                return "nd";
            case 3:
                return "rd";
            default:
                return "th";
        }
    }

    // Main entry point
    //
    public static void main(String[] args) {

        ZITraders Market = new ZITraders();

        Market.OpenTrading();

    }
}
```

BuyerAgent.java

```
/*
 * BuyerAgent class
 *
 * CSS 610
 * George Mason University
 * Spring 2005
 *
 * Rob Axtell
 *
 * With modifications added by Brent Auble, February 2010
 *   - added tradable and sellerList variables
 *   - added getValue() and buyPossible()
 *
 * Additional modifications by Brent Auble,
 * December 2011, October 2014 to April 2015
 *   -
 *
 */

import java.util.ArrayList;
import java.util.Comparator;

public class BuyerAgent implements Comparable<BuyerAgent> {

    private int buyerID;
    private double value; // The maximum price this buyer is willing to
buy for
    public boolean traded; // Has this agent successfully made a trade?
    public boolean tradable; // Can anyone still sell to this buyer?
    public double price; // The final price this buyer traded at
    private double averageBid; // The average of bid prices offered
    public ArrayList<Integer> sellerList;
    // Values for each of the interesting features for this agent,
calculated after each trade is attempted
    public double[] interestingFeatures;
    // How each agent ranks on each interesting feature
    public int[] interestingFeatureRanks;
    public ArrayList<TradeAttempt> tradeRecord;
    private int topNRankingCount; // The number of interesting features
this agent ranks in the top N compared to other buyers

    public BuyerAgent(double maxBuyerValue) {
        value = Math.random() * maxBuyerValue;
        traded = false;
        tradable = true; // Assume that a newly created buyer can be
traded with
        sellerList = new ArrayList<Integer>();
        tradeRecord = new ArrayList<TradeAttempt>();
        averageBid = 0;
        interestingFeatures = new double[10];  // Default to 10
features if a value isn't passed in
```

```
        topNRankingCount = 0;
    }

    public BuyerAgent(double maxBuyerValue, int buyerID) {
        value = Math.random() * maxBuyerValue;
        this.buyerID = buyerID;
        traded = false;
        tradable = true; // Assume that a newly created buyer can be
traded with
        sellerList = new ArrayList<Integer>();
        tradeRecord = new ArrayList<TradeAttempt>();
        averageBid = 0;
        interestingFeatures = new double[10];  // Default to 10
features if a value isn't passed in
        topNRankingCount = 0;
   }

    public BuyerAgent(double maxBuyerValue, int buyerID, int
numberOfInterestingFeatures) {
        value = Math.random() * maxBuyerValue;
        this.buyerID = buyerID;
        traded = false;
        tradable = true; // Assume that a newly created buyer can be
traded with
        sellerList = new ArrayList<Integer>();
        tradeRecord = new ArrayList<TradeAttempt>();
        averageBid = 0;
        interestingFeatures = new double[numberOfInterestingFeatures];
        topNRankingCount = 0;
   }

    public int getBuyerID() {
        return buyerID;
    }

    @SuppressWarnings("unused")
    private void setBuyerID(int buyerID) {
        this.buyerID = buyerID;
    }

    public void setTraded(boolean t) {
        traded = t;
    }

    public boolean hasTraded() {
        return (traded);
    }

    public int getTopNRankingCount() {
        return topNRankingCount;
    }

    public void setTopNRankingCount(int topNRankingCount) {
        this.topNRankingCount = topNRankingCount;
```

```
    }

    public double formBidPrice() {
        return Math.random() * value;
    }

    // Added a getter for value in order to test what happens when a
buyer
    // and seller don't attempt to trade unless the buyer's value is at
least
    // the same as or less than the seller's maxCost -- i.e. the don't
    // attempt to trade unless a trade is theoretically possible
    public double getValue(){
        return value;
    }

    // This method just determines whether a trade is possible given
the minimum price
    // a seller is willing to sell for -- this is redundant with the
SellerAgent's
    // sellPossible method, but is included here to allow either to be
used
    public boolean buyPossible(double sellerMinimum){
        if (sellerMinimum <= value) {
            return true;
        } else {
            return false;
        }
    }

    /*
     * getNarrative returns a narrative explanation of the agent's
history
     *
     * NOTE: Buyers offer a price somewhere between 0 and their
"value",
     * which is the maximum amount they're willing to pay.  The
narrative
     * reflects how low a price the agent ends up paying for a
successful
     * trade.
     */
    public String getNarrative(int sellerID, double buyerOffer, double
sellerOffer,
            double salePrice, boolean tradeCompleted) {
        // The +1 below is necessary because the attempt hasn't been
added yet
        int numTradeAttempts = getTradeAttemptCount()+1;
        String s = "";
        //StringBuffer sb = new StringBuffer();  // This is much more
efficient, but will need to be done later
        double percentUnderMaximum = (value - buyerOffer) / value;
        // s1 is the basic offer narrative, usable for both successful
and un- trades
```

```
        String s1 = "I offered seller " + sellerID + " "
                + ZITraders.twoDigits.format(buyerOffer) + ", which is
"
                + ZITraders.twoDigits.format(percentUnderMaximum * 100)
                + "% under my maximum";
//        sb.append("I offered seller ");
//        sb.append(sellerID);
//        sb.append(" ");
//        sb.append(buyerOffer);
//        sb.append(", which is ");
//        sb.append(ZITraders.twoDigits.format(percentUnderMaximum));
//            sb.append("% under my maximum");

        if (tradeCompleted) {
            // Handle a successful trade narrative
            double percentDiff = (value - salePrice) / value;
            String t = getSuccessfulTradeReaction(percentDiff,
salePrice);

            // Handle the narrative based on the number of trade
attempts
            switch (numTradeAttempts) {
            case 1:
//                sb.insert(0, "My first trade attempt was successful!
");
//                sb.append(".  ").append(t);
                s = "My first trade attempt was successful!  " + s1 +
".  " + t;
                break;
            case 2:
//                sb.insert(0, "My second trade attempt was successful.
");
//                sb.append(".  ").append(t);
                s = "My second trade attempt was successful.  " + s1 +
".  " + t;
                break;
            case 3:
//                sb.insert(0, "The third time was the charm!  ");
//                sb.append(".  ").append(t);
                s = "The third time was the charm!  " + s1 + ".  " + t;
                break;
            default:
//                sb.insert(0, " attempts, I was finally able to make a
trade.  ");
//                sb.insert(0, numTradeAttempts);
//                sb.insert(0, "After ");
//                sb.append(".  ").append(t);
                s = "After " + numTradeAttempts + " attempts, I was
finally able to " +
                    "make a trade.  " + s1 + ".  " + t;
            }


        } else {
```

```
            // Handle the narrative for an unsuccessful trade attempt

            // s2 is the buyer's reaction to the amount it offered
under its max value
            String s2 = pctUnderMaxReaction(percentUnderMaximum);
            // Get the narrative about the average offers
            String s3 = averagePctUnderMaxReaction(buyerOffer);
            // Merging a common set of strings for unsuccessful
attempts
            String s4 = s1 + " -- " + s2 + "  " + s3;

            switch (numTradeAttempts) {
            case 0:
                // With the +1 above, this will never be executed, but
I may
                // handle it another way, so I'll leave it here
                s = "No trades attempted yet.  Hopefully an opportunity
will come up soon.";
                break;
            case 1:
                s = "My first trade attempt wasn't successful.  " + s4;
                break;
            case 2:  // Bargaining
                s = "Two failed attempts.  Third time is the charm...
hopefully.  "
                        + s4;
                break;
            case 3:  // Denial
                s = "The third time was not the charm.  I think I've
been reasonable.  "
                        + "This time, " + s4;
                break;
            case 4:  // Anger
                s = "What the heck, is nobody willing to trade with
me?!  This was my "
                        + "fourth failed attempt. " + s4;
                break;
            case 5:  // Depression
                s = "My fifth attempt failed.  This is getting
depressing.  " + s4;
                break;
            case 6:
            case 7:
            case 8:
            case 9:
            case 10:  // Acceptance
                s = "My " + numTradeAttempts + "th try and still no
luck.  I'll just "
                        + "keep trying.  " + s4;
                break;
            default:
                s = "Unsuccessful trade attempt number " +
numTradeAttempts +
```

```
                    ".   Will this ever end with a successful trade?  "
+ s4;
            }
        }

        return s;
    }

    private String getSuccessfulTradeReaction(double percentDiff,
double salePrice) {
      // Handle the narrative based on the percent under the maximum
amount
      // the buyer was willing to pay compared to what the trade was
      // completed for
      StringBuffer sb = new StringBuffer();
      if (percentDiff < 0.1) {
            // Not too bad, less than 10% under my maximum.  Wonder if
it was
            // possible to do better.
            sb.append("This wasn't too bad of a trade.  I paid ");
            sb.append(ZITraders.twoDigits.format(percentDiff * 100));
            sb.append("% under my maximum.  The sale price was ");
            sb.append(ZITraders.twoDigits.format(salePrice));
            sb.append(". I wonder if I could've gotten a better
deal?");
      } else if (percentDiff <= 0.3) {
            // Decent trade
            sb.append("This was a good trade.  I paid ");
            sb.append(ZITraders.twoDigits.format(percentDiff * 100));
            sb.append("% under my maximum.  The sale price was ");
            sb.append(ZITraders.twoDigits.format(salePrice));
            sb.append(".  I would be surprised if I could've gotten a
better deal.");
      } else {
            // Great! Got it for more than 30% less than I was willing
to pay
            sb.append("This was a great trade!  I paid an unbelievable
");
            sb.append(ZITraders.twoDigits.format(percentDiff * 100));
            sb.append("% under my maximum.  The sale price was ");
            sb.append(ZITraders.twoDigits.format(salePrice));
            sb.append(".  I almost feel guilty about the deal I got
from the seller.");
            sb.append("..  Almost.");
      }
      return sb.toString();
    }

    private String pctUnderMaxReaction(double pctUnderMax) {
        // Generate a sentence indicating how the agent reacts to an
unsuccessful offer
        String s = "";
        if (pctUnderMax < 0.1) {
            s = "I couldn't go much higher!";
```

```
        } else if (pctUnderMax < 0.3) {
            s = "I think that was reasonable.";
        } else {
            s = "I wonder if it was too little?";
        }

        return s;
    }

    private String averagePctUnderMaxReaction(double buyerOffer) {
        // Generate the sentence describing how this bid compares to
its average so far
        // Calculate a temporary averageBid because the main one hasn't
been updated yet
        int numTrades = getTradeAttemptCount() + 1;
        if (numTrades == 1) {
            return "";
        } else {
            double tempAverageBid = ((averageBid * (numTrades-1)) +
                    buyerOffer)/numTrades;
            StringBuffer sb = new StringBuffer();
            sb.append("After ");
            sb.append(numTrades);
            sb.append(" trade attempts, my average offer was ");
            sb.append(ZITraders.twoDigits.format(tempAverageBid));
            sb.append(", which is ");
            sb.append(ZITraders.twoDigits.format(((value -
tempAverageBid)/value) * 100));
            sb.append("% under my maximum.");

            return sb.toString();
        }
    }

    public double getAverageBid() {
        return averageBid;
    }

    //int buyerID, int sellerID, double buyerOffer, double sellerOffer,
double salePrice, boolean tradeCompleted, String narrative
    public void addTradeAttempt(int sellerID, double buyerOffer, double
sellerCost,
            double sellerOffer, double salePrice, boolean
tradeCompleted) {
        tradeRecord.add(new TradeAttempt(buyerID, sellerID, value,
buyerOffer,
            sellerCost, sellerOffer, salePrice, tradeCompleted,
            getNarrative(sellerID, buyerOffer, sellerOffer, salePrice,
tradeCompleted)));
        int tradeAttempts = getTradeAttemptCount();
        averageBid = ((averageBid * (tradeAttempts-1)) +
                buyerOffer)/tradeAttempts;
        // Update interesting features
        /*
```

```
         *  6) Largest gap between buyer and seller
            7) Smallest gap between buyer and seller (most improbable
trade)
            8) Trades closest to the average/median/expected value
            9) Broadest range of offers within a buyer or seller

         */
        if (tradeCompleted) {
            interestingFeatures[0] = tradeAttempts;  // Number of trade
attempts when ultimately successful
            interestingFeatures[4] = price; // 4) Lowest buyer maximum
price that was successful
            interestingFeatures[5] = buyerOffer; // 5) Lowest buyer
offer price that was successful
            interestingFeatures[6] = sellerOffer - buyerOffer; // 6)
Largest gap between buyer and seller
            interestingFeatures[7] = interestingFeatures[6]; // 7)
Smallest gap between buyer and seller (most improbable trade)

        } else {
            interestingFeatures[1] = tradeAttempts;  // Number of trade
attempts without being successful

        }
    }

    public int getTradeAttemptCount() {
        return tradeRecord.size();
    }

    /**
     *
     * @return A single String with the complete set of narratives
     */
    public String getCompleteNarrative () {
      StringBuffer sb = new StringBuffer();
            sb.append("Narrative for buyer ");
        sb.append(buyerID);
        sb.append(", who has a maximum purchase price of ");
        sb.append(value);
        sb.append(": \n\n");

      int n = getTradeAttemptCount();
      if (n==0) {
            sb.append("No trades have been attempted yet.  ");
      } else if (n==1) {
            sb.append("1 trade was attempted.  ");
      } else {
            sb.append(n);
            sb.append(" trades have been attempted.  ");
            sb.append("The range of offers made by this buyer was ");
            sb.append(getOfferRange());
            sb.append(".  ");
      }
```

```
      if (traded) {
            sb.append(" A successful trade was completed with seller
");
            sb.append(tradeRecord.get(n - 1).getSellerID());
            sb.append(" for ");
            sb.append(tradeRecord.get(n - 1).getSalePrice());
            sb.append(".");
          } else {
            if (n==1) {
                  sb.append("It was not successful.");
            } else if (n==2) {
                  sb.append("Neither was successful.");
            } else {
                  sb.append("None have been successful.");
            }
      }

      sb.append("\n\n");
        sb.append(getTopRankingNarrative());
      sb.append("\n");

      for (int i = 0; i < n; i++) {
            sb.append(tradeRecord.get(i).toString());
            if (i != n-1) { // Only add newlines after the non-final
narrative
                  sb.append("\n\n");
            }
      }

      return sb.toString();
    }

    private String getTopRankingNarrative () {
        StringBuffer sb = new StringBuffer();

        int n = getTradeAttemptCount();
      if (traded) {
            // Check to see if this agent is in the Top N for number of
Trade attempts
            if (n >= ZITraders.buyerTradeAttemptRankTopN) {
                topNRankingCount++;
                int attemptRank =
ZITraders.buyerTradeAttemptCountRanks.get(n);
                sb.append("* This agent ranks in ");
                sb.append(attemptRank);

sb.append(ZITraders.getOrdinalNumberSuffix(attemptRank));
                sb.append(" place for number of attempts before
successfully making a trade");
                int attemptRankTies =
ZITraders.buyerTradeAttemptRanks.get(n);
                if (attemptRankTies > 1) {
                    sb.append(" (tied with ");
```

```
                    sb.append(attemptRankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            // Check to see if this agent is in the Top N for trade
price
            Double tradePrice = tradeRecord.get(n - 1).getSalePrice();
            if (tradePrice <= ZITraders.buyerPriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.buyerPriceCountRanks.get(tradePrice);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest successful trade price
among buyers");
                int rankTies =
ZITraders.buyerPriceRanks.get(tradePrice);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            // Check to see if this agent is in the Top N for
successful trade offer price
            Double offerPrice = tradeRecord.get(n - 1).getBuyerOffer();
            if (offerPrice <= ZITraders.buyerOfferPriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.buyerOfferPriceCountRanks.get(offerPrice);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest successful offer price
among buyers");
                int rankTies =
ZITraders.buyerOfferPriceRanks.get(offerPrice);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            // Check to see if this agent is in the Top N for
successful trade with lowest value
            if (value <= ZITraders.buyerMaxPriceRankTopN) {
                topNRankingCount++;
```

```
                int rank =
ZITraders.buyerMaxPriceCountRanks.get(value);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest maximum purchase price
among buyers who made a successful trade");
                int rankTies = ZITraders.buyerMaxPriceRanks.get(value);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            /* Check to see if this agent is in the Top N for
             * successful trade with smallest difference between
             * buyer's max price and seller's min price */
            Double valueDiff = value - tradeRecord.get(n -
1).getSellerValue();
            if (valueDiff <= ZITraders.minMaxPriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.minMaxPriceCountRanks.get(valueDiff);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest difference between the
buyer's maximum purchase price and the seller's minimum sale price
among buyers who made a successful trade");
                int rankTies =
ZITraders.minMaxPriceRanks.get(valueDiff);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(", with a difference of ");
                sb.append(valueDiff);
                sb.append(".\n");
            }

            /* Check to see if this agent is in the Top N for
             * successful trade with smallest difference between
             * buyer's offer price and seller's offer price */
            Double offerDiff = tradeRecord.get(n - 1).getBuyerOffer() -
                               tradeRecord.get(n - 1).getSellerOffer();
            if (offerDiff <= ZITraders.offerPriceGapRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.offerPriceGapCountRanks.get(offerDiff);
                sb.append("* This agent ranks in ");
                sb.append(rank);
```

```
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest difference between the
buyer's offer price and the seller's offer price among buyers who made
a successful trade");
                int rankTies =
ZITraders.offerPriceGapRanks.get(offerDiff);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(", with a difference of ");
                sb.append(offerDiff);
                sb.append(".\n");
            }

            /* Check to see if this agent is in the Top N for
             * successful trade with smallest difference between
             * average transaction price and final trade price */
            Double averageDiff =
Math.abs(ZITraders.averageTransactionPrice -
                                    tradeRecord.get(n -
1).getSalePrice());
            if (averageDiff <= ZITraders.averagePriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.averagePriceCountRanks.get(averageDiff);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest difference between the
final trade price and the overall average transaction price (");
                sb.append(ZITraders.averageTransactionPrice);
                sb.append(")");
                int rankTies =
ZITraders.averagePriceRanks.get(averageDiff);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(", with a difference of ");
                sb.append(averageDiff);
                sb.append(".\n");
            }

            // Range of prices offered in trade attempts, successful
only
            double offerRange = getOfferRange();
            if (offerRange >= ZITraders.buyerOfferRangeRankTopN) {
                topNRankingCount++;
                int attemptRank =
ZITraders.buyerOfferRangeCountRanks.get(offerRange);
                sb.append("* This agent ranks in ");
```

```
                sb.append(attemptRank);

sb.append(ZITraders.getOrdinalNumberSuffix(attemptRank));
                sb.append(" place for the range of prices offered (");
                sb.append(offerRange);
                sb.append(" from max to min) in trade attempts");
                int attemptRankTies =
ZITraders.buyerOfferRangeRanks.get(offerRange);
                if (attemptRankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(attemptRankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

          } else {
            if (n >= ZITraders.buyerTradeAttemptRankTopNUnsucc) {
                topNRankingCount++;
                int attemptRank =
ZITraders.buyerTradeAttemptCountRanksUnsucc.get(n);
                sb.append("* This agent ranks in ");
                sb.append(attemptRank);

sb.append(ZITraders.getOrdinalNumberSuffix(attemptRank));
                sb.append(" place for number of unsuccessful trade
attempts without making a trade");
                int attemptRankTies =
ZITraders.buyerTradeAttemptRanksUnsucc.get(n);
                if (attemptRankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(attemptRankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }
      }

         return sb.toString();
    }

    /**
     * Loop through all the transactions this agent has attempted and
     * return a string with all the information about those
     * transactions in a format suitable for a comma separated values
     * (CSV) file
     *
     * @return String
     */
    public String getCompleteTransactionHistoryCSV(boolean
includeNarrative) {
      StringBuffer sb = new StringBuffer();
      int n = getTradeAttemptCount();
```

```
      if (n==0) {
            sb.append(buyerID).append(",,,,,,,,,\"No trade
attempts\"\n");
      } else {

      //"buyerID,attemptNum,sellerID,buyerValue,buyerOffer,sellerValue,
sellerOffer,salePrice,tradeCompleted,timeStamp,narrative"
            for (int i = 0; i < n; i++) {
                  sb.append(buyerID).append(",");
                  sb.append(i).append(",");

      sb.append(tradeRecord.get(i).getSellerID()).append(",");

      sb.append(tradeRecord.get(i).getBuyerValue()).append(",");

      sb.append(tradeRecord.get(i).getBuyerOffer()).append(",");

      sb.append(tradeRecord.get(i).getSellerValue()).append(",");

      sb.append(tradeRecord.get(i).getSellerOffer()).append(",");

      sb.append(tradeRecord.get(i).getSalePrice()).append(",");

      sb.append(tradeRecord.get(i).isTradeCompleted()).append(",");
                  if (includeNarrative) {

      sb.append(tradeRecord.get(i).getTimeStamp()).append(",");
                        // Surround the narrative with quotes

      sb.append("\"").append(tradeRecord.get(i).getNarrative()).append(
"\"\n");
                  } else {

      sb.append(tradeRecord.get(i).getTimeStamp()).append("\n");
                  }
            }
      }

      return sb.toString();
    }

    /**
     * Loop through all the transactions this agent has attempted and
     * return a string with the information about only the transaction
     * that successfully completed in a format suitable for a comma
     * separated values (CSV) file.  Note that this automatically
     * excludes the narrative.
     *
     * @return String
     */
    public String getCompletedTransactionHistoryCSV() {
      StringBuffer sb = new StringBuffer();
      int n = getTradeAttemptCount();
```

```
      if (n==0) {
            sb.append(buyerID).append(",,,,,,,,,\"No trade
attempts\"\n");
      } else {

      //"buyerID,sellerID,buyerValue,buyerOffer,sellerValue,sellerOffer
,salePrice,tradeCompleted,timeStamp"
            for (int i = 0; i < n; i++) {
                  if (tradeRecord.get(i).isTradeCompleted()) {
                        sb.append(buyerID).append(",");

      sb.append(tradeRecord.get(i).getSellerID()).append(",");

      sb.append(tradeRecord.get(i).getBuyerValue()).append(",");

      sb.append(tradeRecord.get(i).getBuyerOffer()).append(",");

      sb.append(tradeRecord.get(i).getSellerValue()).append(",");

      sb.append(tradeRecord.get(i).getSellerOffer()).append(",");

      sb.append(tradeRecord.get(i).getSalePrice()).append(",");

      sb.append(tradeRecord.get(i).isTradeCompleted()).append(",");

      sb.append(tradeRecord.get(i).getTimeStamp()).append("\n");
                  }
            }
      }

      return sb.toString();
    }

    // Return the range (max - min) of offer prices on all transaction
attempts by this agent
    public double getOfferRange() {
        int n = getTradeAttemptCount();
        if (n == 0) {
            return 0.0;
        } else {
            double minTransPrice = ZITraders.maxBuyerValue + 1.0;
            double maxTransPrice = -1.0;
            for (int i = 0; i < n; i++) {
                  double transPrice =
tradeRecord.get(i).getBuyerOffer();
                if (transPrice < minTransPrice) minTransPrice =
transPrice;
                if (transPrice > maxTransPrice) maxTransPrice =
transPrice;
            }
            return maxTransPrice - minTransPrice;
        }
    }
```

```
    // Perform all the actions necessary at the agent level to complete
a successful trade
    public void completeTrade(int sellerID, double buyerOffer, double
sellerCost, double sellerOffer,
                double salePrice, boolean tradeCompleted) {
        traded = true;
        price = salePrice;
        addTradeAttempt(sellerID, buyerOffer, sellerCost, sellerOffer,
salePrice, tradeCompleted);
        interestingFeatures[0] = getTradeAttemptCount();  // Most trade
attempts but ultimately successful
    }

    @Override
    public int hashCode() {
        int hash = 5;
        hash = 37 * hash + this.buyerID;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final BuyerAgent other = (BuyerAgent) obj;
        if (this.buyerID != other.buyerID) {
            return false;
        }
        return true;
    }

    @Override
    public int compareTo(BuyerAgent otherAgent) {
        return this.buyerID > otherAgent.getBuyerID() ? 1 :
                (this.buyerID < otherAgent.getBuyerID() ? -1 : 0);
    }

    public static class OrderByPrice implements Comparator<BuyerAgent>{

        @Override
        public int compare(BuyerAgent t, BuyerAgent t1) {
            double tPrice = t.getValue();
            double t1Price = t1.getValue();

            return tPrice > t1Price ? 1 :
                    (tPrice < t1Price ? -1 : 0);
        }
    }
```

```
    public static class OrderByTradeAttempts implements
Comparator<BuyerAgent>{

        @Override
        public int compare(BuyerAgent t, BuyerAgent t1) {
            double t_val = t.getTradeAttemptCount();
            double t1_val = t1.getTradeAttemptCount();

            return t_val > t1_val ? 1 :
                    (t_val < t1_val ? -1 : 0);
        }
    }

    public static class OrderByTopNRankings implements
Comparator<BuyerAgent>{
        // Sort agents by the number of Top N rankings they have,
descending
        @Override
        public int compare(BuyerAgent t, BuyerAgent t1) {
            int t_val = t.getTopNRankingCount();
            double t1_val = t1.getTopNRankingCount();

            return t_val < t1_val ? 1 :
                    (t_val > t1_val ? -1 : 0);
        }
    }

}
```

SellerAgent.java

```
/*
 * SellerAgent class
 *
 * CSS 610
 * George Mason University
 * Spring 2005
 *
 * Rob Axtell
 *
 * With modifications added by Brent Auble, February 2010
 *   - added tradable and buyerList variables
 *   - added getCost() and sellPossible()
 *
 * Additional modifications by Brent Auble,
 * December 2011, October 2014 to April 2015
 *   -
 *
 */

import java.util.ArrayList;
import java.util.Comparator;

public class SellerAgent implements Comparable<SellerAgent>{

    private int sellerID;
      private double cost; // The minimum price this seller is willing
to sell for
    private double maxCost; // The maximum possible seller cost in the
model, used to form ask prices by finding the gap between cost and
maxCost
    public boolean traded; // Has this agent successfully made a trade?
    public boolean tradable; // Can anyone still buy from this seller?
    public double price; // The final price this seller traded at
    private double averageBid; // The average of bid prices offered
    public ArrayList<Integer> buyerList;
    // Values for each of the interesting features for this agent,
calculated after each trade is attempted
    public double[] interestingFeatures;
    // How each agent ranks on each interesting feature
    public int[] interestingFeatureRanks;
    public ArrayList<TradeAttempt> tradeRecord;
    private int topNRankingCount; // The number of interesting features
this agent ranks in the top N compared to other sellers

    public SellerAgent(double maxSellerCost) {
        cost = Math.random() * maxSellerCost;
        maxCost = maxSellerCost;
        traded = false;
        tradable = true; // Assume that a newly created seller can be
traded with
```

```
        buyerList = new ArrayList<Integer>();
        tradeRecord = new ArrayList<TradeAttempt>();
        averageBid = 0;
        topNRankingCount = 0;
    }

    public SellerAgent(double maxSellerCost, int sellerID) {
        cost = Math.random() * maxSellerCost;
        this.sellerID = sellerID;
        maxCost = maxSellerCost;
        traded = false;
        tradable = true; // Assume that a newly created seller can be
traded with
        buyerList = new ArrayList<Integer>();
        tradeRecord = new ArrayList<TradeAttempt>();
        averageBid = 0;
        topNRankingCount = 0;
    }

    public int getSellerID() {
          return sellerID;
      }

      @SuppressWarnings("unused")
      private void setSellerID(int sellerID) {
          this.sellerID = sellerID;
      }

      public void setTraded(boolean t) {
        traded = t;
    }

    public boolean hasTraded() {
        return (traded);
    }

    public int getTopNRankingCount() {
        return topNRankingCount;
    }

    public void setTopNRankingCount(int topNRankingCount) {
        this.topNRankingCount = topNRankingCount;
    }

    public double formAskPrice() {
        return cost + (Math.random() * (maxCost - cost));
    }

    /** Added a getter for cost in order to test what happens when a
buyer
     * and seller don't attempt to trade unless the buyer's value is at
least
     * the same as or less than the seller's maxCost -- i.e. they don't
     * attempt to trade unless a trade is theoretically possible
```

```
     */
    public double getCost(){
        return cost;
    }

    /** This method just determines whether a trade is possible given
the
     * maximum price a buyer is willing to pay -- this is redundant
with the
     * BuyerAgent's buyPossible method, but is included here to allow
either to be used
     * @param buyerMaximum The Buyer agent's maximum purchase price
     * @return Whether it's possible for this seller to trade with the
selected buyer
     */
    public boolean sellPossible(double buyerMaximum){
        if (buyerMaximum >= cost) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * getNarrative returns a narrative explanation of the agent's
     * history
     *
     * NOTE: Sellers offer a price somewhere between their "cost",
     * which is the minimum amount they're willing to sell for, and the
     * maxCost, which is the maximum allowed cost in the model
     * (although not a number the seller explicitly "knows").  The
     * seller wants to sell for as high a price as possible, and the
     * narrative reflects how well it did.
     */
    public String getNarrative(int buyerID, double buyerOffer, double
sellerOffer,
            double salePrice, boolean tradeCompleted) {
        // The +1 below is necessary because the attempt hasn't been
added yet
        int numTradeAttempts = getTradeAttemptCount()+1;
        String s = "";
        double percentOverMinimum = (sellerOffer - cost) / cost;
        // s1 is the basic offer narrative, usable for both successful
and un- trades
        String s1 = "I offered buyer " + buyerID + " "
                + ZITraders.twoDigits.format(sellerOffer) + ", which is
"
                + ZITraders.twoDigits.format(percentOverMinimum * 100)
                + "% over my minimum sell price";

        if (tradeCompleted) {
            // Handle a successful trade narrative
            double percentDiff = (salePrice - cost) / cost;
```

```
            String t = getSuccessfulTradeReaction(percentDiff,
salePrice);

            // Handle the narrative based on the number of trade
attempts
            switch (numTradeAttempts) {
            case 1:
                s = "My first trade attempt was successful!  " + s1 +
".  " + t;
                break;
            case 2:
                s = "My second trade attempt was successful.  " + s1 +
".  " + t;
                break;
            case 3:
                s = "The third time was the charm!  " + s1 + ".  " + t;
                break;
            default:
                s = "After " + numTradeAttempts + " attempts, I was
finally able to "
                        + "make a trade.  " + s1 + ".  " + t;
            }


        } else {
            // Handle the narrative for an unsuccessful trade attempt

            // s2 is the seller's reaction to the amount it offered
over its min value
            String s2 = pctOverMinReaction(percentOverMinimum);
            // Get the narrative about the average offers
            String s3 = averagePctOverMinReaction(sellerOffer);
            // Merging a common set of strings for unsuccessful
attempts
            String s4 = s1 + " -- " + s2 + "  " + s3;

            switch (numTradeAttempts) {
            case 0:
                // With the +1 above, this will never be executed, but
I may handle
                // it another way, so I'll leave it here
                s = "No trades attempted yet.  Hopefully an opportunity
will "
                        + "come up soon.";
                break;
            case 1:
                s = "My first trade attempt wasn't successful.  " + s4;
                break;
            case 2:  // Bargaining
                s = "Two failed attempts.  Third time is the charm...
hopefully.  "
                        + s4;
                break;
            case 3:  // Denial
```

```
                s = "The third time was not the charm.  I think I've
been "
                        + "reasonable.  This time, " + s4;
                break;
            case 4:  // Anger
                s = "What the heck, is nobody willing to trade with
me?!  "
                        + "This was my fourth failed attempt. " + s4;
                break;
            case 5:  // Depression
                s = "My fifth attempt failed.  This is getting
depressing.  " + s4;
                break;
            case 6:
            case 7:
            case 8:
            case 9:
            case 10:  // Acceptance
                s = "My " + numTradeAttempts + "th try and still no
luck.  "
                        + "I'll just keep trying.  " + s4;
                break;
            default:
                s = "Unsuccessful trade attempt number " +
numTradeAttempts +
                ".   Will this ever end with a successful trade?  " +
s4;
            }
        }

        return s;
    }

    /**
     * Handle the narrative based on the percent over the minimum
     * amount the seller was willing to sell for compared to what the
     * trade was completed for
     *
     * @param percentDiff
     * @param salePrice
     * @return String of a sentence describing the seller's reaction to
the trade
     */
    private String getSuccessfulTradeReaction(double percentDiff,
double salePrice) {
        StringBuffer sb = new StringBuffer();
        if (percentDiff < 0.1) {
            // Not too bad, less than 10% over my minimum.  Wonder if
            // it was possible to do better.
            sb.append("This wasn't too bad of a trade.  I got ");
            sb.append(ZITraders.twoDigits.format(percentDiff * 100));
            sb.append("% over my minimum.  The sale price was ");
            sb.append(ZITraders.twoDigits.format(salePrice));
```

```
            sb.append(".  I wonder if I could've gotten a better
deal?");
        } else if (percentDiff <= 0.3) {
            // Decent trade
            sb.append("This was a good trade.  I got ");
            sb.append(ZITraders.twoDigits.format(percentDiff * 100));
            sb.append("% over my minimum.  The sale price was ");
            sb.append(ZITraders.twoDigits.format(salePrice));
            sb.append(".  I would be surprised if I could've gotten a
better deal.");
        } else {
            // Great! Sold it for more than 30% more than I was willing
to take
            sb.append("This was a great trade!  I received an
unbelievable ");
            sb.append(ZITraders.twoDigits.format(percentDiff * 100));
            sb.append("% over my minimum.  The sale price was ");
            sb.append(ZITraders.twoDigits.format(salePrice));
            sb.append(".  I almost feel guilty about the deal I got
from the buyer.");
            sb.append("..  Almost.");
        }
        return sb.toString();
    }

    /**
     * Generate the sentence indicating how the agent reacts to an
     * unsuccessful offer
     *
     * @param pctOverMin
     * @return
     */
    private String pctOverMinReaction(double pctOverMin) {
        String s = "";
        if (pctOverMin < 0.1) {
            s = "I couldn't go much lower!";
        } else if (pctOverMin < 0.3) {
            s = "I think that was reasonable.";
        } else {
            s = "I wonder if it was too much?";
        }

        return s;
    }

    private String averagePctOverMinReaction(double sellerOffer) {
        // Generate the sentence describing how this bid compares to
its average so far
        // Calculate a temporary averageBid because the main one hasn't
been updated yet
        int numTrades = getTradeAttemptCount() + 1;
        if (numTrades == 1) {
            return "";
        } else {
```

```
            double tempAverageBid = ((averageBid * (numTrades-1)) +
sellerOffer)/numTrades;
            StringBuffer sb = new StringBuffer();
            sb.append("My average offer was ");
            sb.append(ZITraders.twoDigits.format(tempAverageBid));
            sb.append(", which is ");
            sb.append(ZITraders.twoDigits.format(((tempAverageBid -
cost)/cost) * 100));
            sb.append("% over my minimum.");

            return sb.toString();
        }
    }

    public double getAverageBid() {
        return averageBid;
    }

    /**
     *
     * @return A single String with the complete set of narratives
     */
    public String getCompleteNarrative () {
        StringBuffer sb = new StringBuffer();
        sb.append("Narrative for seller ");
        sb.append(sellerID);
        sb.append(", who has a minimum sale price of ");
        sb.append(cost);
        sb.append(": \n\n");

        int n = getTradeAttemptCount();
        if (n==0) {
            sb.append("No trades have been attempted yet.  ");
        } else if (n==1) {
            sb.append("1 trade was attempted.  ");
        } else {
            sb.append(n);
            sb.append(" trades have been attempted.  ");
            sb.append("The range of offers made by this seller was ");
            sb.append(getOfferRange());
            sb.append(".  ");
        }

        if (traded) {
            sb.append("A successful trade was completed with buyer ");
            sb.append(tradeRecord.get(n -1).getBuyerID());
            sb.append(" for ");
            sb.append(tradeRecord.get(n -1).getSalePrice());
            sb.append(".");
        } else {
            if (n==1) {
                sb.append("It was not successful.");
            } else if (n==2) {
                sb.append("Neither was successful.");
```

```
            } else {
                sb.append("None have been successful.");
            }
        }

      sb.append("\n\n");
        sb.append(getTopRankingNarrative());
      sb.append("\n");

        for (int i = 0; i < n; i++) {
            sb.append(tradeRecord.get(i).toString());
            if (i != n-1) { // Only add newlines after the non-final
narrative
                sb.append("\n\n");
            }
        }

        return sb.toString();
    }

    // Generate narrative of rankings of how this agent did on various
metrics:
    private String getTopRankingNarrative () {
        StringBuffer sb = new StringBuffer();

        int n = getTradeAttemptCount();
      if (traded) {
            // Check to see if this agent is in the Top N for number of
Trade attempts
            if (n >= ZITraders.sellerTradeAttemptRankTopN) {
                topNRankingCount++;
                int attemptRank =
ZITraders.sellerTradeAttemptCountRanks.get(n);
                sb.append("* This agent ranks in ");
                sb.append(attemptRank);

sb.append(ZITraders.getOrdinalNumberSuffix(attemptRank));
                sb.append(" place for number of attempts before
successfully making a trade");
                int attemptRankTies =
ZITraders.sellerTradeAttemptRanks.get(n);
                if (attemptRankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(attemptRankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            // Check to see if this agent is in the Top N for trade
price
            Double tradePrice = tradeRecord.get(n - 1).getSalePrice();
            if (tradePrice >= ZITraders.sellerPriceRankTopN) {
                topNRankingCount++;
```

```
                int rank =
ZITraders.sellerPriceCountRanks.get(tradePrice);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for highest successful trade price
among sellers");
                int rankTies =
ZITraders.sellerPriceRanks.get(tradePrice);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            // Check to see if this agent is in the Top N for
successful trade offer price
            Double offerPrice = tradeRecord.get(n -
1).getSellerOffer();
            if (offerPrice >= ZITraders.sellerOfferPriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.sellerOfferPriceCountRanks.get(offerPrice);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for highest successful offer price
among sellers");
                int rankTies =
ZITraders.sellerOfferPriceRanks.get(offerPrice);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            // Check to see if this agent is in the Top N for
successful trade with highest value
            if (cost >= ZITraders.sellerMinPriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.sellerMinPriceCountRanks.get(cost);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for highest minimum sale price among
sellers who made a successful trade");
                int rankTies = ZITraders.sellerMinPriceRanks.get(cost);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
```

```
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

            /* Check to see if this agent is in the Top N for
             * successful trade with smallest difference between
             * seller's min price and buyer's max price */
            Double valueDiff = tradeRecord.get(n - 1).getBuyerValue() -
cost;
            if (valueDiff <= ZITraders.minMaxPriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.minMaxPriceCountRanks.get(valueDiff);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest difference between the
seller's minimum sale price and the buyer's maximum purchase price
among sellers who made a successful trade");
                int rankTies =
ZITraders.minMaxPriceRanks.get(valueDiff);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(", with a difference of ");
                sb.append(valueDiff);
                sb.append(".\n");
            }

            /* Check to see if this agent is in the Top N for
             * successful trade with smallest difference between
             * seller's offer price and buyer's offer price */
            Double offerDiff = tradeRecord.get(n - 1).getBuyerOffer() -
                               tradeRecord.get(n - 1).getSellerOffer();
            if (offerDiff <= ZITraders.offerPriceGapRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.offerPriceGapCountRanks.get(offerDiff);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest difference between the
seller's offer price and the buyer's offer price among sellers who made
a successful trade");
                int rankTies =
ZITraders.offerPriceGapRanks.get(offerDiff);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
```

```
                }
                sb.append(", with a difference of ");
                sb.append(offerDiff);
                sb.append(".\n");
            }

            /* Check to see if this agent is in the Top N for
             * successful trade with smallest difference between
             * average transaction price and final trade price */
            Double averageDiff =
Math.abs(ZITraders.averageTransactionPrice -
                                    tradeRecord.get(n -
1).getSalePrice());
            if (averageDiff <= ZITraders.averagePriceRankTopN) {
                topNRankingCount++;
                int rank =
ZITraders.averagePriceCountRanks.get(averageDiff);
                sb.append("* This agent ranks in ");
                sb.append(rank);
                sb.append(ZITraders.getOrdinalNumberSuffix(rank));
                sb.append(" place for lowest difference between the
final trade price and the overall average transaction price (");
                sb.append(ZITraders.averageTransactionPrice);
                sb.append(")");
                int rankTies =
ZITraders.averagePriceRanks.get(averageDiff);
                if (rankTies > 1) {
                    sb.append(" (tied with ");
                    sb.append(rankTies);
                    sb.append(" others)");
                }
                sb.append(", with a difference of ");
                sb.append(averageDiff);
                sb.append(".\n");
            }

              // Range of prices offered in trade attempts, successful
only
                double offerRange = getOfferRange();
                if (offerRange >= ZITraders.sellerOfferRangeRankTopN) {
                    topNRankingCount++;
                    int attemptRank =
ZITraders.sellerOfferRangeCountRanks.get(offerRange);
                    sb.append("* This agent ranks in ");
                    sb.append(attemptRank);

sb.append(ZITraders.getOrdinalNumberSuffix(attemptRank));
                    sb.append(" place for the range of prices offered
(");
                    sb.append(offerRange);
                    sb.append(" from max to min) in trade attempts");
                    int attemptRankTies =
ZITraders.sellerOfferRangeRanks.get(offerRange);
                    if (attemptRankTies > 1) {
```

```
                    sb.append(" (tied with ");
                    sb.append(attemptRankTies);
                    sb.append(" others)");
                }
                sb.append(".\n");
            }

        } else {
          if (n >= ZITraders.sellerTradeAttemptRankTopNUnsucc) {
              topNRankingCount++;
              int attemptRank =
ZITraders.sellerTradeAttemptCountRanksUnsucc.get(n);
              sb.append("* This agent ranks in ");
              sb.append(attemptRank);

sb.append(ZITraders.getOrdinalNumberSuffix(attemptRank));
              sb.append(" place for number of unsuccessful trade
attempts without making a trade");
              int attemptRankTies =
ZITraders.sellerTradeAttemptRanksUnsucc.get(n);
              if (attemptRankTies > 1) {
                  sb.append(" (tied with ");
                  sb.append(attemptRankTies);
                  sb.append(" others)");
              }
              sb.append(".\n");
          }
      }

        return sb.toString();
    }

    //int buyerID, int sellerID, double buyerOffer, double sellerOffer,
double salePrice, boolean tradeCompleted, String narrative
    public void addTradeAttempt(int buyerID, double buyerOffer, double
buyerCost,
            double sellerOffer, double salePrice, boolean
tradeCompleted) {
        tradeRecord.add(new TradeAttempt(buyerID, sellerID, buyerCost,
buyerOffer,
            cost, sellerOffer, salePrice, tradeCompleted,
            getNarrative(sellerID, buyerOffer, sellerOffer, salePrice,
tradeCompleted)));
        averageBid = ((averageBid * (getTradeAttemptCount()-1)) +
            sellerOffer)/getTradeAttemptCount();
   }

    public int getTradeAttemptCount() {
        return tradeRecord.size();
    }

    /**
     * Loop through all the transactions this agent has attempted and
     * return a string with all the information about those
```

```
     * transactions in a format suitable for a comma separated values
     * (CSV) file
     *
     * @return String
     */
    public String getCompleteTransactionHistoryCSV() {
        StringBuffer sb = new StringBuffer();
        int n = getTradeAttemptCount();

        if (n==0) {
            sb.append(",").append(sellerID).append(",,,,,,,,\"No trade
attempts\"\n");
        } else {

//"sellerID,attemptNum,buyerID,buyerValue,buyerOffer,sellerValue,seller
Offer,salePrice,tradeCompleted,timeStamp,narrative"
            for (int i = 0; i < n; i++) {
                sb.append(sellerID).append(",");
                sb.append(i).append(",");
                sb.append(tradeRecord.get(i).getBuyerID()).append(",");

sb.append(tradeRecord.get(i).getBuyerValue()).append(",");

sb.append(tradeRecord.get(i).getBuyerOffer()).append(",");

sb.append(tradeRecord.get(i).getSellerValue()).append(",");

sb.append(tradeRecord.get(i).getSellerOffer()).append(",");

sb.append(tradeRecord.get(i).getSalePrice()).append(",");

sb.append(tradeRecord.get(i).isTradeCompleted()).append(",");

sb.append(tradeRecord.get(i).getTimeStamp()).append(",");
                // Surround the narrative with single quotes

sb.append("\"").append(tradeRecord.get(i).getNarrative()).append("\"\n"
);
            }
        }

        return sb.toString();
    }

    // Return the range (max - min) of offer prices on all transaction
attempts by this agent
    public double getOfferRange() {
        int n = getTradeAttemptCount();
        if (n == 0) {
            return 0.0;
        } else {
            double minTransPrice = ZITraders.maxSellerCost + 1.0;
            double maxTransPrice = -1.0;
            for (int i = 0; i < n; i++) {
```

```
                double transPrice =
tradeRecord.get(i).getSellerOffer();
                if (transPrice < minTransPrice) minTransPrice =
transPrice;
                if (transPrice > maxTransPrice) maxTransPrice =
transPrice;
            }
            return maxTransPrice - minTransPrice;
        }
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 89 * hash + this.sellerID;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final SellerAgent other = (SellerAgent) obj;
        if (this.sellerID != other.sellerID) {
            return false;
        }
        return true;
    }

    @Override
    public int compareTo(SellerAgent otherAgent) {
        return this.sellerID > otherAgent.getSellerID() ? 1 :
                (this.sellerID < otherAgent.getSellerID() ? -1 : 0);
    }

    public static class OrderByCost implements Comparator<SellerAgent>{

        @Override
        public int compare(SellerAgent t, SellerAgent t1) {
            double tCost = t.getCost();
            double t1Cost = t1.getCost();

            return tCost > t1Cost ? 1 :
                    (tCost < t1Cost ? -1 : 0);
        }
    }

    public static class OrderByTradeAttempts implements
Comparator<SellerAgent>{
```

```
        @Override
        public int compare(SellerAgent t, SellerAgent t1) {
            double t_val = t.getTradeAttemptCount();
            double t1_val = t1.getTradeAttemptCount();

            return t_val > t1_val ? 1 :
                    (t_val < t1_val ? -1 : 0);
        }
    }

    public static class OrderByTopNRankings implements
Comparator<SellerAgent>{
        // Sort agents by the number of Top N rankings they have,
descending
        @Override
        public int compare(SellerAgent t, SellerAgent t1) {
            int t_val = t.getTopNRankingCount();
            double t1_val = t1.getTopNRankingCount();

            return t_val < t1_val ? 1 :
                    (t_val > t1_val ? -1 : 0);
        }
    }

}
```

AgentPopulation.java

```
/*
 * AgentPopulation class
 *
 * CSS 610
 * George Mason University
 * Summer 2009
 *
 * Rob Axtell
 *
 * Modified by Brent Auble
 * December 2011
 *
 */
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class AgentPopulation {

    public BuyerAgent[] buyers;
    public SellerAgent[] sellers;
    int nBuyers;
    int nSellers;

    public AgentPopulation(final int numBuyers, final double maxValue,
            final int numSellers, final double maxCost, String
dateTimeStamp) {

        nBuyers = numBuyers;
        nSellers = numSellers;

        buyers = new BuyerAgent[nBuyers];
        sellers = new SellerAgent[nSellers];

        for (int i = 0; i < numBuyers; i++) {
            buyers[i] = new BuyerAgent(maxValue, i);
        }

            for (int i = 0; i < numSellers; i++) {
            sellers[i] = new SellerAgent(maxCost, i);
        }

            // Save initial info about the buyer and seller agents to
files
            writeBuyerInfo(dateTimeStamp);
            writeSellerInfo(dateTimeStamp);

    }

    public int getActiveBuyers() {
```

```
        int count = 0;

        for (int i = 0; i < nBuyers; i++) {
            if (buyers[i].hasTraded()) {
                count++;
            }
        }
        return (count);
    }

    public int getActiveSellers() {

        int count = 0;

        for (int i = 0; i < nSellers; i++) {
            if (sellers[i].hasTraded()) {
                count++;
            }
        }
        return (count);
    }

    public void writeBuyerInfo(String dateTimeStamp) {
        // Create file to store buyer agent info
        FileWriter buyerFile;
            try {
                  // Create a file to save the information about the
buyers
                  buyerFile = new FileWriter(ZITraders.saveLocation +
                              "Buyers " + dateTimeStamp + ".csv",
true);
                  BufferedWriter buyerWriter = new
BufferedWriter(buyerFile);
                  buyerWriter.write("buyerID,MaxPurchasePrice");
                  buyerWriter.newLine();

              for (int i = 0; i < nBuyers; i++) {
                  buyerWriter.write(i + "," + buyers[i].getValue());
                        buyerWriter.newLine();
                  }

                  buyerWriter.close();

            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

    }

    public void writeSellerInfo(String dateTimeStamp) {
        // Create file to store seller agent info
        FileWriter sellerFile;
```

```
            try {
                  // Create a file to save the information about the
sellers
                  sellerFile = new FileWriter(ZITraders.saveLocation +
                              "Sellers " + dateTimeStamp + ".csv",
true);
                  BufferedWriter sellerWriter = new
BufferedWriter(sellerFile);
                  sellerWriter.write("sellerID,MinSalePrice");
                  sellerWriter.newLine();

                for (int i = 0; i < nSellers; i++) {
                  sellerWriter.write(i + "," + sellers[i].getCost());
                  sellerWriter.newLine();
                }

                sellerWriter.close();

            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }

    }

}
```

TradeAttempt.java

```
/**
 * This class records trade attempts for ZITraders
 */

/**
 * @author Brent
 * December, 2011
 */

import java.util.Date;

public class TradeAttempt {

	private int buyerID;
	private int sellerID;
	private double buyerValue; // The buyer's maximum purchase price
	private double buyerOffer; // The amount the buyer was actually
willing to pay for this trade
	private double sellerValue; // The seller's minimum sales price
	private double sellerOffer; // The amount the seller actually
offered
	private double salePrice; // The completed sale price (or 0 if
the trade failed)
	private boolean tradeCompleted; // Whether the trade was
successful or not
	private Date timeStamp;  // When was the trade completed
	private String narrative; // The agent's narrative description of
the trade attempt

	/**
	 * Build a record of a trade attempt with complete information,
including the
	 * buyer and seller values (their actual maximum/minimum prices,
respectively)
	 *
	 * @param buyerID
	 * @param sellerID
	 * @param buyerValue
	 * @param buyerOffer
	 * @param sellerValue
	 * @param sellerOffer
	 * @param salePrice
	 * @param tradeCompleted
     * @param narrative
	 */
	public TradeAttempt(int buyerID, int sellerID, double buyerValue,
			double buyerOffer, double sellerValue, double
sellerOffer,
			double salePrice, boolean tradeCompleted, String
narrative) {
```

```
            super();
            this.buyerID = buyerID;
            this.sellerID = sellerID;
            this.buyerValue = buyerValue;
            this.buyerOffer = buyerOffer;
            this.sellerValue = sellerValue;
            this.sellerOffer = sellerOffer;
            this.setSalePrice(salePrice);
            this.tradeCompleted = tradeCompleted;
            this.narrative = narrative;
            setTimeStamp(new Date());
      }

      /**
       * Build a record of a trade attempt with all required
information
       *
       * @param buyerID
       * @param sellerID
       * @param buyerOffer
       * @param sellerOffer
       * @param salePrice
       * @param tradeCompleted
     * @param narrative
       */
      public TradeAttempt(int buyerID, int sellerID, double buyerOffer,
                  double sellerOffer, double salePrice, boolean
tradeCompleted,
                  String narrative) {
            super();
            this.buyerID = buyerID;
            this.sellerID = sellerID;
            this.buyerOffer = buyerOffer;
            this.sellerOffer = sellerOffer;
            this.setSalePrice(salePrice);
            this.tradeCompleted = tradeCompleted;
            this.narrative = narrative;
            setTimeStamp(new Date());
      }

      /**
       * Build a record of a completed trade attempt
       *
       * @param buyerID
       * @param sellerID
       * @param buyerOffer
       * @param sellerOffer
       * @param salePrice
       * @param tradeCompleted
       */
      public void TradeAttemptCompleted(int buyerID, int sellerID,
double buyerOffer,
                  double sellerOffer, double salePrice) {
            this.buyerID = buyerID;
```

```
            this.sellerID = sellerID;
            this.buyerOffer = buyerOffer;
            this.sellerOffer = sellerOffer;
            this.setSalePrice(salePrice);
            this.tradeCompleted = true;
      }

      /**
       * Build a record of a failed trade attempt
       *
       * @param buyerID
       * @param sellerID
       * @param buyerOffer
       * @param sellerOffer
       * @param salePrice
       * @param tradeCompleted
       */
      public void TradeAttemptFailed(int buyerID, int sellerID, double
buyerOffer,
                  double sellerOffer, double salePrice) {
            this.buyerID = buyerID;
            this.sellerID = sellerID;
            this.buyerOffer = buyerOffer;
            this.sellerOffer = sellerOffer;
            this.setSalePrice(0); // a failed attempt would have a sale
price of 0
            this.tradeCompleted = false;
      }

      public int getBuyerID() {
            return buyerID;
      }

      public void setBuyerID(int buyerID) {
            this.buyerID = buyerID;
      }

      public int getSellerID() {
            return sellerID;
      }

      public void setSellerID(int sellerID) {
            this.sellerID = sellerID;
      }

      public double getBuyerValue() {
            return buyerValue;
      }

      public void setBuyerValue(double buyerValue) {
            this.buyerValue = buyerValue;
      }

      public double getBuyerOffer() {
```

```
        return buyerOffer;
    }

    public void setBuyerOffer(double buyerOffer) {
        this.buyerOffer = buyerOffer;
    }

    public double getSellerValue() {
        return sellerValue;
    }

    public void setSellerValue(double sellerValue) {
        this.sellerValue = sellerValue;
    }

    public double getSellerOffer() {
        return sellerOffer;
    }

    public void setSellerOffer(double sellerOffer) {
        this.sellerOffer = sellerOffer;
    }

    public double getSalePrice() {
        return salePrice;
    }

    public void setSalePrice(double salePrice) {
        this.salePrice = salePrice;
    }

    public boolean isTradeCompleted() {
        return tradeCompleted;
    }

    public void setTradeCompleted(boolean tradeCompleted) {
        this.tradeCompleted = tradeCompleted;
    }

    public Date getTimeStamp() {
        return timeStamp;
    }

    public void setTimeStamp(Date timeStamp) {
        this.timeStamp = timeStamp;
    }

    public String getNarrative() {
        return narrative;
    }

    public void setNarrative(String narrative) {
        this.narrative = narrative;
    }
```

```
    @Override
      public String toString() {
            return timeStamp + ":  " + narrative;
      }

}
```

Data.java

```
/*
 * Data class
 *
 * CSS 610
 * George Mason University
 * Spring 2005
 *
 * Rob Axtell
 *
 */

public class Data {

    private int N;
    private double min, max, sum, sum2;

    public Data() {
        N = 0;
        min = 1000000000.0;
        max = 0.0;
        sum = 0.0;
        sum2 = 0.0;
    }

    public void AddDatuum(double Datuum) {
        N++;
        if (Datuum < min) {
            min = Datuum;
        }
        if (Datuum > max) {
            max = Datuum;
        }
        sum += Datuum;
        sum2 += Datuum * Datuum;
    }

    public int GetN() {
        return N;
    }

    public double GetMin() {
        return min;
    }

    public double GetMax() {
        return max;
    }

    public double GetAverage() {
        if (N > 0) {
```

```
            return sum / N;
        } else {
            return 0.0;
        }
    }

    public double GetVariance() {

        double avg, arg;

        if (N > 1) {
            avg = GetAverage();
            arg = sum2 - N * avg * avg;
            return arg / (N - 1);
        } else {
            return 0.0;
        }
    }

    public double GetStdDev() {
        return Math.sqrt(GetVariance());
    }

    public void CombineWithDifferentData(Data otherData) {
        N += otherData.GetN();
        min = Math.min(min, otherData.GetMin());
        max = Math.max(max, otherData.GetMax());
        sum += otherData.GetN() * otherData.GetAverage();
        sum2 += otherData.GetVariance() * (otherData.GetN() - 1) +
otherData.GetN() * otherData.GetAverage() * otherData.GetAverage();
    }
}
```

References

Anscombe, F. J. (1973). Graphs in Statistical Analysis. The American Statistician, 27 (1): 17-21.

Arthur, W. B. (1994). Inductive Reasoning and Bounded Rationality. The American Economic Review, 84 (2): 406-411.

Axelrod, R. (1997). Advancing the Art of Simulation in the Social Sciences. Complexity, 3(2), 16–22.

Balci, O. (1998). Verification, Validation, and Testing. In J. Banks (Ed.), Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice (pp. 335–393). Hoboken, NJ, USA: John Wiley & Sons, Inc.

Bigbee, A., Cioffi-Revilla, C., & Luke, S. 2007. Replication of Sugarscape Using MASON. In Agent-Based Approaches in Economic and Social Complex Systems IV: Post-Proceedings of The AESCS International Workshop 2005, edited by T. Terano, H. Kita, H. Deguchi and K. Kijima. Tokyo: Springer.

Cioffi-Revilla, C., Honeychurch, W., & Rogers, J. D. (2013). MASON Hierarchies: A Long-Range Agent Model of Power, Conflict, and Environment in Inner Asia. In: Bemmann, J. and Brosseder, U., The Complexity of Interaction Along the Eurasian Steppe Zone in the First Millennium AD: Empires, Cities, Nomads and Farmers. Bonn: Bonn University Press.

Dautenhahn, K. & Coles, S. J. (2001). Narrative Intelligence from the Bottom Up: A Computational Framework for the Study of Story-Telling in Autonomous Agents. Journal of Artificial Societies and Social Simulation 4 (1) 1. http:/jasss.soc.surrey.ac.uk/4/1/1.html.

Epstein, J. M., and Axtell, R. L. (1996). Growing Artificial Societies: Social Science from the Bottom Up. Cambridge, MA: The MIT Press.

Gilbert, N. (2008). Agent-Based Models (Quantitative Applications in the Social Sciences). SAGE Publications.

Gilbert, N. & Troitzsch, K. G. (2005). Simulation for the Social Scientist, 2nd Ed. Open University Press, London.

Gode, D. K. & Sunder, S. (1993). Allocative efficiency of markets with zero intelligence traders: markets as a partial substitute for individual rationality. Journal of Political Economy, 101: 119-137.

Gode, D. K. & Sunder, S. (1997). What makes markets allocationally efficient? The Quarterly Journal of Economics, 112 (2): 603–630.

Grimm, V. (2002). Visual Debugging: A Way of Analyzing, Understanding and Communicating Bottom-up Simulation Models in Ecology. Natural Resource Modeling 15 (1): 23–38.

Guerin, F. & Pitt, J. (2000). A Semantic Framework for Specifying Agent Communication Languages. Fourth International Conference on Multi-Agent Systems (ICMAS '00), pp.0395.

Kirke, A., & Miranda, E. (2015). A Multi-Agent Emotional Society Whose Melodies Represent its Emergent Social Hierarchy and Are Generated by Agent Communications. Journal of Artificial Societies and Social Simulation, 18(2), 16. http://jasss.soc.surrey.ac.uk/18/2/16.html

Kornhauser, D., Wilensky, U., & Rand, W. (2009). Design Guidelines for Agent Based Model Visualization. Journal of Artificial Societies and Social Simulation 12(2)1. http://jasss.soc.surrey.ac.uk/12/2/1.html

Kübler-Ross, E. (1969). On Death & Dying. Routledge. (Based loosely on the descriptions from http://en.wikipedia.org/wiki/K%C3%BCbler-Ross_model and http://www.ekrfoundation.org/five-stages-of-grief).

Li, J. & Wilensky, U. (2009). NetLogo Sugarscape 1 Immediate Growback model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/models/Sugarscape1ImmediateGrowback

Madden, N. (2009) Collaborative narrative generation in persistent virtual environments. PhD thesis, University of Nottingham, U.K.

Millington, J. D., O'Sullivan, D., & Perry, G. L. (2012). Model histories: Narrative explanation in generative simulation modelling. Geoforum, 43(6), 1025-1034.

North, M. J. & Macal, C. M. (2007). Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation. Oxford University Press, USA.

Railsback, S. F. & Grimm, V. (2011). Agent-Based and Individual-Based Modeling: A Practical Introduction. Princeton University Press, Princeton, NJ.

Rand, W. M. (2005). Controlled Observations of the Genetic Algorithm in a Changing Environment: Case Studies Using the Shaky Ladder Hyperplane-Defined Functions. Doctoral dissertation, The University of Michigan.

Rand, W. M. & Stonedahl, F. (2007). The El Farol Bar Problem and Computational Effort: Why People Fail to Use Bars Efficiently. In Proceedings of Agent 2007 on Complex Interaction and Social Emergence. Chicago, IL.

Schelling, T. (1971). Dynamic models of segregation. The Journal of Mathematical Sociology, 1:2, 143-186

Schelling, T. (1978). Micromotives and Macrobehavior. New York: Norton.

Walras, L. (1926). Eléments d'Economie Politique Pure, Paris et Lausanne: translated by W. Jaffé (1954), Elements of Pure Economics. Homewood: Richard D. Irwin, Inc.

Wilensky, U. (1997). NetLogo Segregation model. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/models/Segregation.

Wilensky, U. (1999). NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/.

Wilensky, U., & Rand, W. M. (2015). An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo. MIT Press.

Biography

Brent D. Auble is a Ph.D. student in Computational Social Science at George Mason University (GMU). He received a Bachelor of Science in Computer Science from Lafayette College in Easton, PA, and a Graduate Certificate in Computational Social Science from GMU. He works full-time as a consultant with LMI, assisting government agencies with improving how they manage their organization and data.