Week 4: LinkedList and Agent-based Simulations

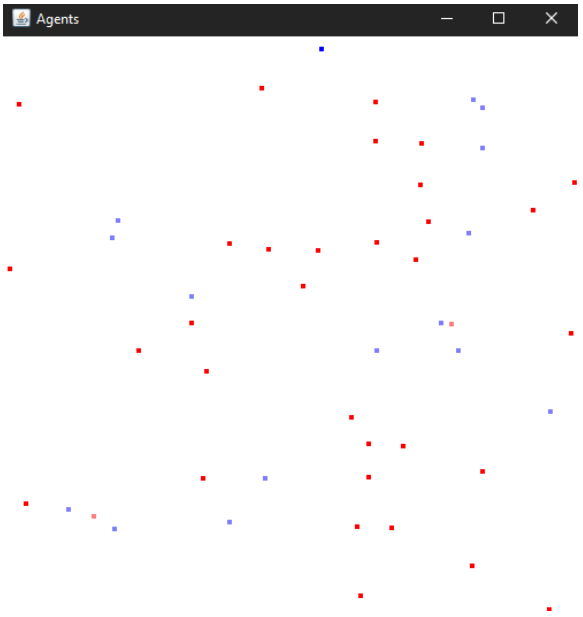
Abstract

This week's project focuses on the new LinkedList data structure and its use in the implementation of agent-based simulations. The core project explores the time it would take for a simulation to reach equilibrium/stable state, with two agents that is the SocialAgent and the AntiSocialAgent. Each agent has multiple parameters that can be varied to find interesting results. To measure the metrics that is time to equilibrium, the explorations focuses on 1. varying the number of Agents in the Landscape and 2. alternative rules for the Agent.

Results

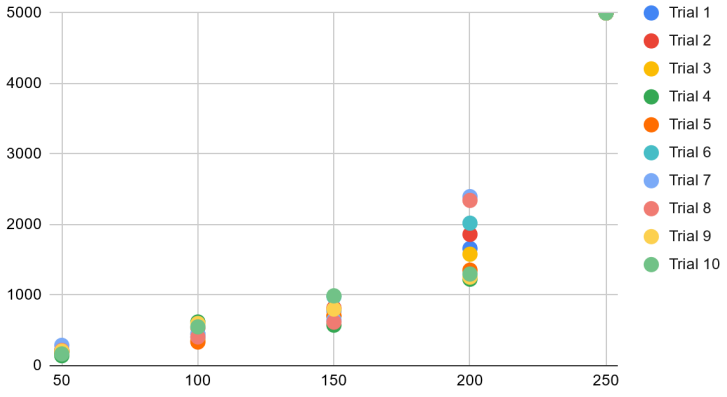
Exploration 1: Varying the number of Agents

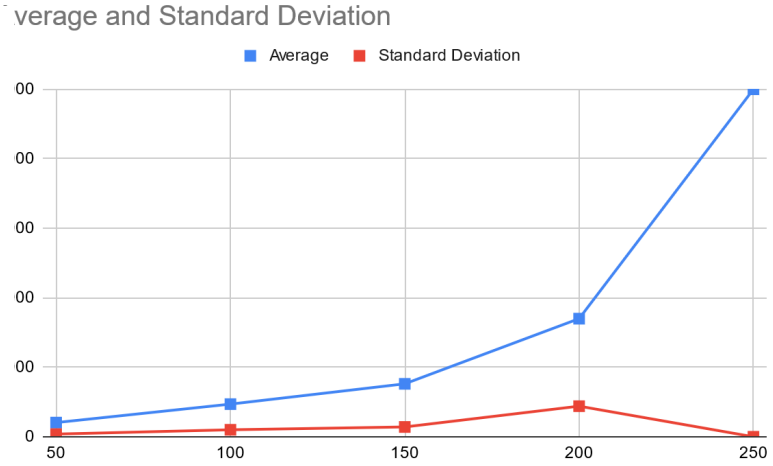
The number of agents is set to 50, 100, 150, 200 and 250 and we observe the time it takes for the simulation to stop. Data is collected from the `SocialAgentSimulation` class. To collect data simply compile and use `java SocialAgentSimulation`. To (!un)/visualize the simulation (un)comment the `LandscapeDisplay display` object and `Thread.sleep()` line. Here, the agents are ran with default settings of radius = 25



As we increase the number of agents the grid gets more crowded, the increasingly anti-social population in the same fixed size grid then finds it harder to settle down to a place. Therefore, it is predicted that the time for the simulation to stop will increases. And such is proved by the table and graph below:

N	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average	Standard Deviation
50	219	198	203	141	187	216	288	216	207	166	204.1	38.46629116
100	402	539	353	619	335	447	448	407	598	550	469.8	100.7006565
150	778	818	703	574	707	979	649	617	802	990	761.7	141.3066406
200	1661	1861	1580	1226	1354	2020	2396	2343	1255	1299	1699.5	440.3688479
250	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000	0

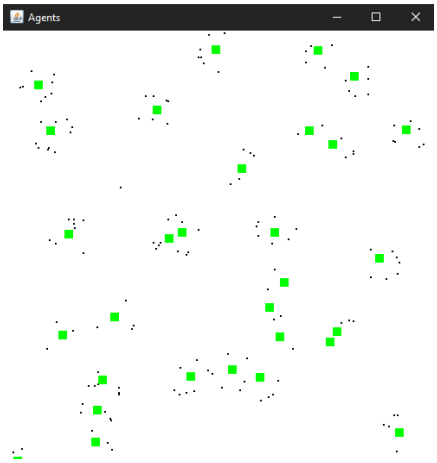




Exploration 2: Red really likes Blue

In this version: There are the Dog and Ticks simulation. The Dog moves if there are too many Ticks around them. The Tick moves if there are too many Ticks or too few Dogs around them. There are far fewer Dogs than Ticks. I varied the number of Ticks and Dogs to measure how long it would take for the simulation to stop. I created a copy of the `SocialAgentSimulation` class and modified it to the `DogTicksSimulation` class. To run simply compile and use

java DogTicksSimulation



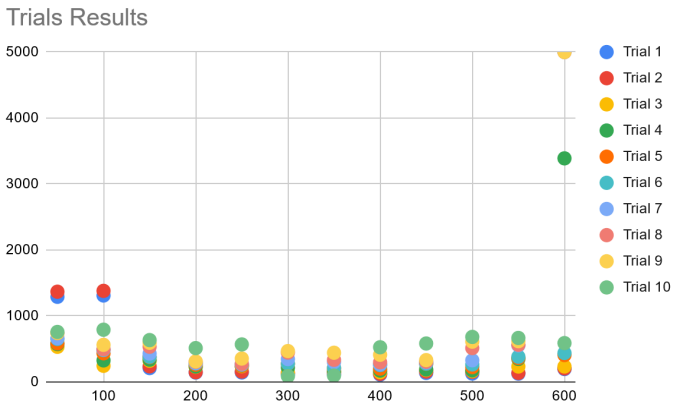
As above, to (!un)/visualize the simulation (un)comment the `LandscapeDisplay display` object and `Thread.sleep()` line.

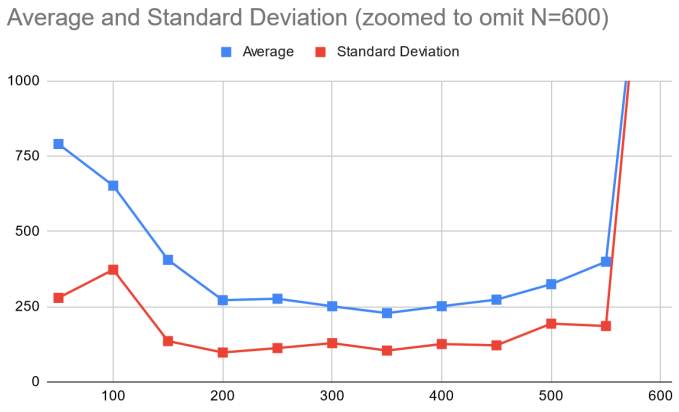
Both Dogs and Ticks have a search radius of 25, Both Dogs and Ticks will run away if there are more than 10 ticks around it. The ticks the ticks can jump up to ±20 in each axis while the Dog can only do ±10. Ticks are always 5 times the number of Dogs.

Running the simulation for 50-600 Ticks with intervals of 50 (10-120 Dogs) the obtained results is obtained in the table and graphs below.

We see an interesting dip at around 350 ticks, implying an ideal amount/ratio for Ticks to find match with Dogs. At the lower end, we observe that since the landscape is too scarse the Ticks have a hard time finding Dogs. At the higher end, the Ticks either settle very quickly (because there are a lot of dogs around) but find it harder too find a place with few ticks. Therefore at the higher end the results accross trials can be drastically different, judging from the high standard deviation.

N	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average	Standard Deviation
50	1291	1368	533	574	585	652	662	742	742	758	790.7	279.4405
100	1311	1381	245	327	434	483	491	496	562	792	652.2	372.9662
150	211	237	321	338	376	382	430	534	594	635	405.8	135.888
200	143	149	231	235	262	282	286	305	311	513	271.7	97.93166
250	144	158	238	246	248	264	267	277	355	569	276.6	112.6004
300	118	137	138	219	275	278	347	446	468	89	251.5	128.9785
350	113	138	159	164	207	214	301	336	443	97	228.7778	104.3326
400	113	123	147	173	202	256	266	299	412	526	251.7	126.0254
450	137	157	182	190	275	277	296	307	331	583	273.5	121.7738
500	128	154	164	176	225	266	328	514	611	683	324.9	193.7377
550	126	137	234	349	363	382	555	564	618	667	399.5	185.9641
600	196	206	234	3389	408	439	5000	5000	5000	589	2046.1	2132.177





Extension: Supply and Demand Simulation Using Agent-Based Modeling

Abstract

In my extension I decided to use the agent-based simulation to abstract a market economy implementing supply and demand dynamics, inspired by the Principles of Microeconomics course. I attempted to demonstrate microeconomics theory: how the interactions between individual buyers and sellers as autonomous agents who interact and adapt based on successful or not exchanges come to determine the equilibrium price and quantity exchanged of the market as a whole. That is, autonomous action determines collective results. The simulation focuses on the price and histogram of participants in the market but uses little numerical metrics and instead rely on visualization for qualitative discussion of the results. The extension implements OOP extensively and modifies heavily on the core project code, the details of which is within the comments of the code.

Usage

Simply compile every class and run `java LandscapeDisplay`. This automatically runs a simulation of 800x400 size with 100 buyers and 200 sellers. Modify the scape in LandscapeDisplay's initialization to adjust parameters of the simulation.

Design and Implementation

Key Components

1. **Agent Architecture** Based on the core project code, the simulation uses an abstract `Agent` class as the foundation, with specialized `Buyer` and `Seller` subclasses

   Each agent maintains a price point (`p`) and adapts it based on market interactions. The Agent no longer have the x and y coordinates as field because now the task of interpreting the `p` to coordinates on the screen is delegated to the `Landscape` and `AgentList` class. This is because the result depends on the height and width and the many ways to intepret this `p` of the `Landscape` class. In fact, we don't even see the actual Agents but only the analyses and visualized collect tive results of their choices.

2. **AgentList.java** An extended `ArrayList` called `AgentList<E>` manages lists of agents. I decided to *not* use `LinkedList` in the extension due to 1. not needing individual modification (add/remove at the middle) of the list 2. new needs from new custom methods makes it helpful to have O(1) access to elements.

   Custom methods in `AgentList` handle interpretation of the Agent's prices, including mapping the price to histogram and cumulative histogram forms, and translate coordinates to Java Graphics-ready integers.

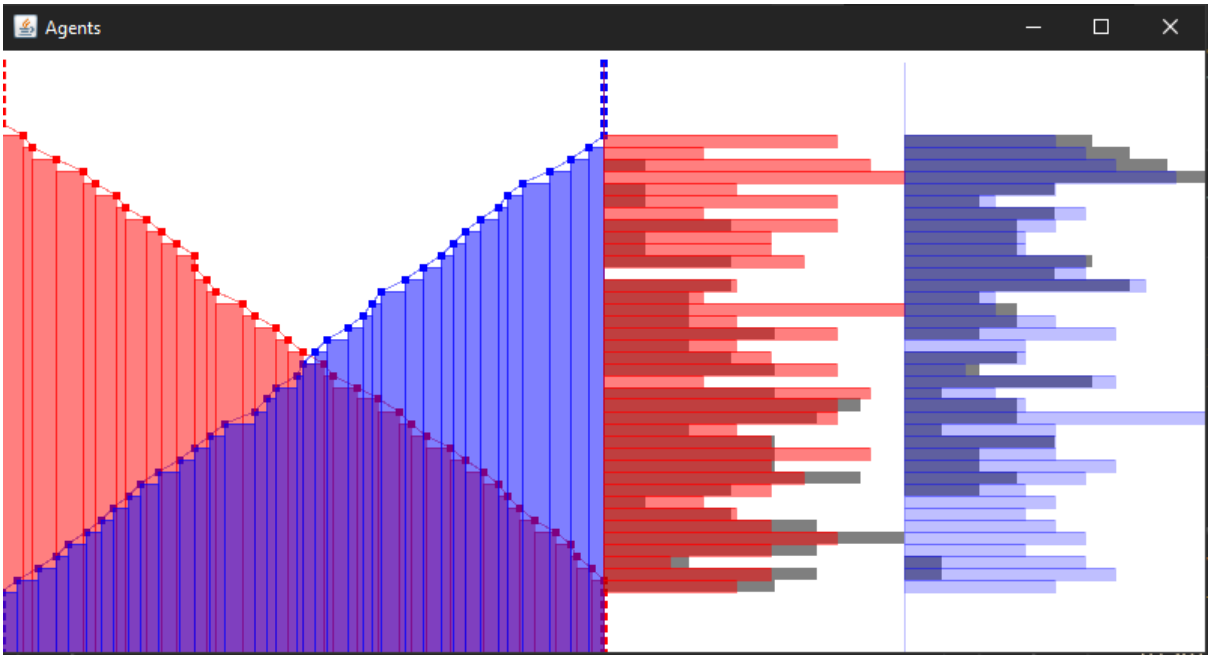
   More notable and confusing is the cumulative histogram calculations, here I used the `accumulate` and `accumulateR` methods to interpret the seller and the buyer respectively's function in the market. In particular, `accumulate` is used to calculate the seller's cumulative histogram function in the market because the seller will sell at their expected price or higher so the direction of accumulation is with the direction of the price. Similar but oppositely, `accumulateR` is used to calculate the buyer's accumulative function in the market because the buyer will buy at their expected price or lower so the direction of accumulation is against the direction of the price.

3. **Market Simulation Algorithm** The `Landscape` class orchestrates the market simulation. Each iteration, called by the `updateAgent()` method shuffles the market and assign a random partner to each participant in the market when possible. The two partners compares prices and decide if the want to follow through with the exchange or not and update their prices accordingly. We also assume that the goods exchanged at hand is an absolute necessity. That is, agents are always willing to adjust their prices in order to exchange the goods. This is one of the many simplification made because the simulation cannot encapsulate completely the real world. Refer to `Agent.java`, `Buyer.java`, `Seller.java` and the `matchAgents()` method in `Landscape.java`.

Visualization Design

The simulation implements two types of graphs:

1. **histogram Graph**: Displays the frequency distribution of prices on intervals
2. **cumulative histogram Graph**: Shows the cumulative histogram of prices, that is, the typical Demand and Supply graph one would find in a microeconomics course. The extension aims to make clear how we arrive to this graph.



On the screen, from left to right, is the

1. cumulative histogram graph, represents the market
2. histogram graph of the buyers
3. histogram graph of the sellers

The consumer/buyer is always color coded in red, and the supplier/seller is always color coded in blue. Participants who did not get to exchange is color coded in gray/black. This is important to discuss the results of the extension later.

For the sake of simplicity, in this market it is assumed that the price is a double ranging from in [0;1.0], each Agent exchanges one item/time and therefore quantity supplied and demanded is equavalence to the number of participants willing to supply or demand.

That is, the y-axis is always the price and x-axis shows the cumulative histogram/histogram number of sellers/buyers willing to participate at each price point.

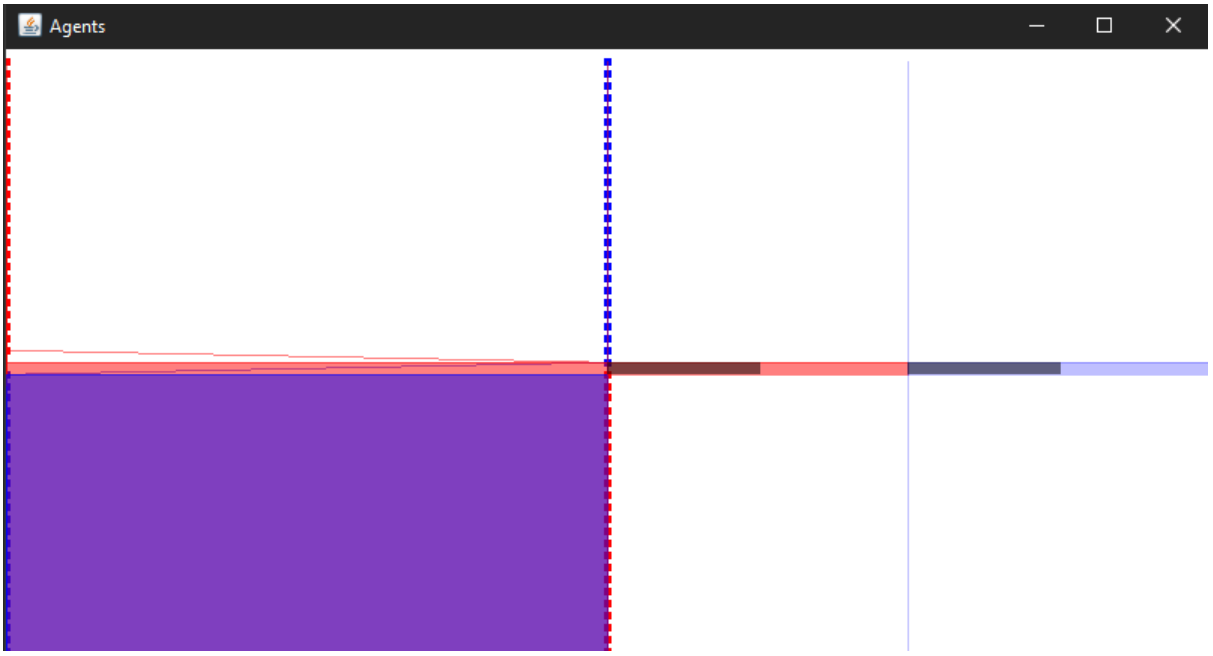
The graph subclasses inside Landscape were helpful to drawing the graphs where and how they were needed. The details are explained in the code comments but I must admit most of it is arithmetics and jargon.

Results and Discussion

The simulation demonstrates several key economic principles:

Price convergence

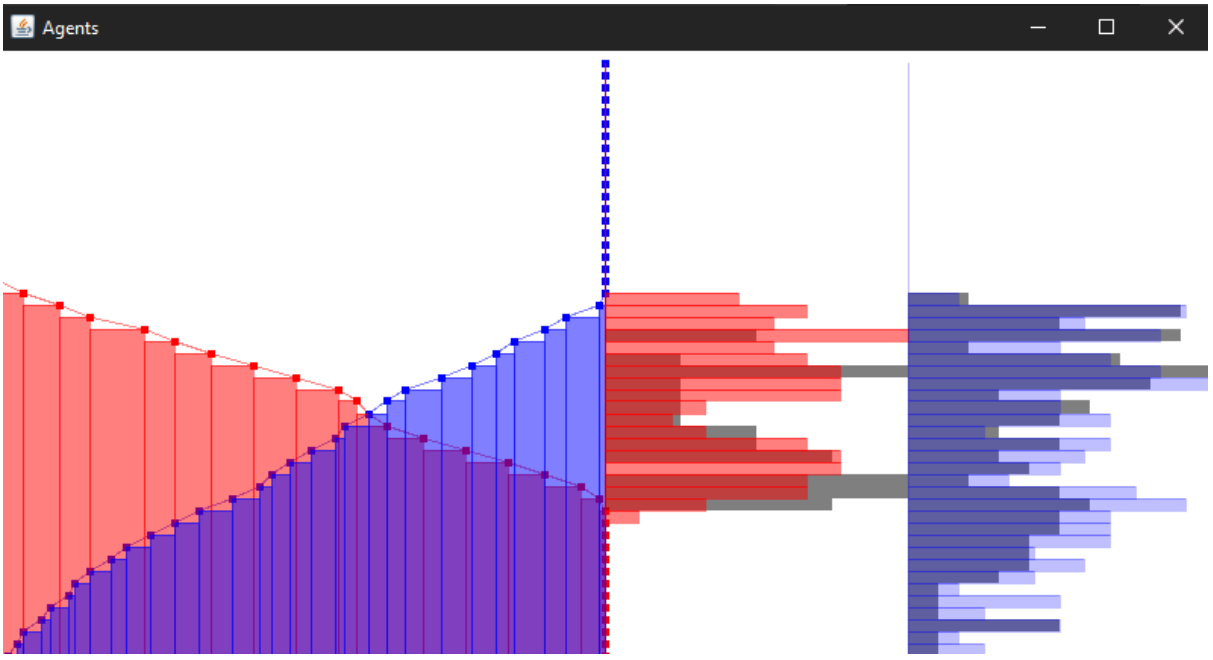
Over time, buyer and seller price points tend to converge toward an equilibrium. We observe this as the “collapse” of the two graphs towards one price point. This is shown as the participants grow to trade each time closer and closer to the equilibrium price point. The histogram graphs of both buyer and seller shows this when the market reaches equilibrium. Although it is notable that this simulation focused heavily on the price, and not the quantity exchanged in the market, due to its simplification.

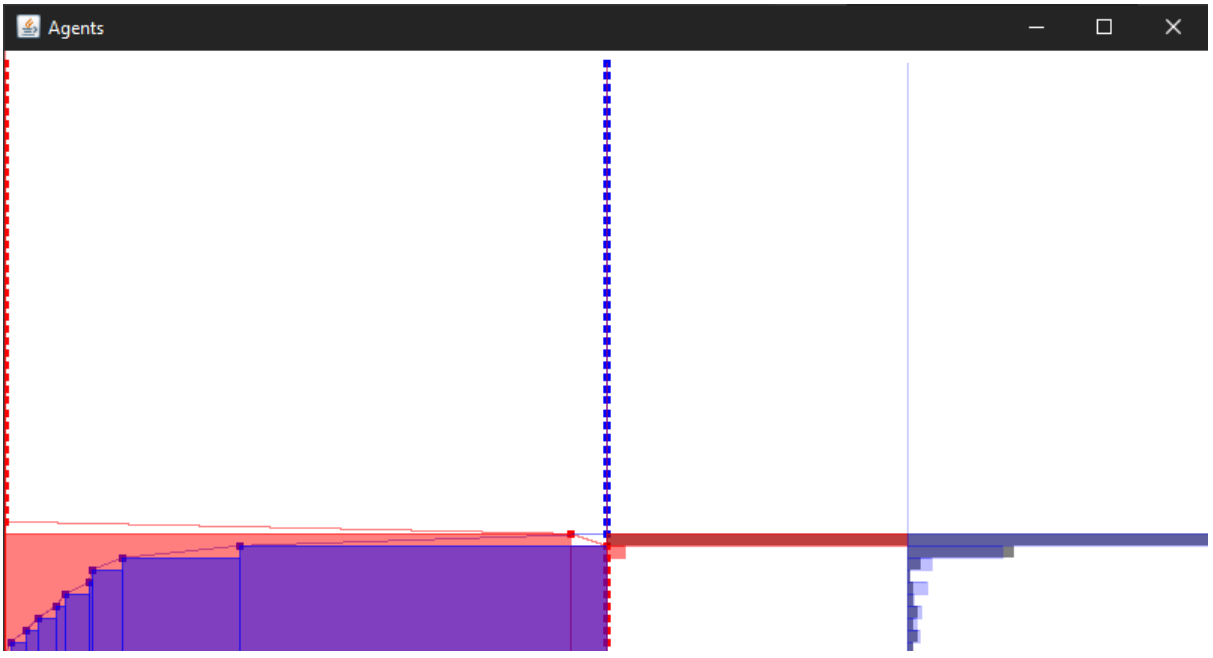


A graph showing a market where everyone has “collapsed” to the equilibrium price. That is, all buyers and sellers trade at this price point.

Shifts of the market

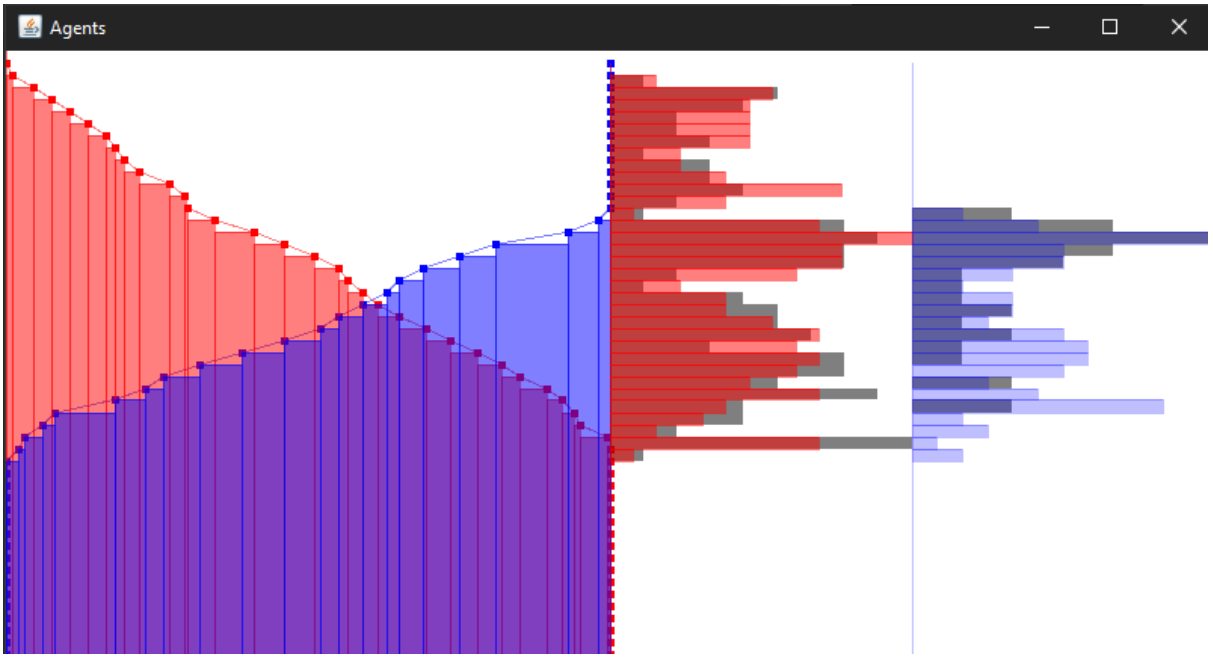
The simulation should also somewhat demonstrate shifts of supply and demand. Here, it is exemplified by the different result from the diffrent initial condition of the simulation. That is, if the market has significantly less buyers than sellers, the buyers has an advantage and therefore equilibrium price tends to be lower to accomodate to the few buyer’s want. This comes from the 100 buyers each iteration that have no partner to trade with (in black in the histogram graph), therefore more willing to raise their price. In the simulation ran below, there are 100 buyers and 200 sellers. We can see in the seller’s distribution graph that they are willing to sell at lower prices.

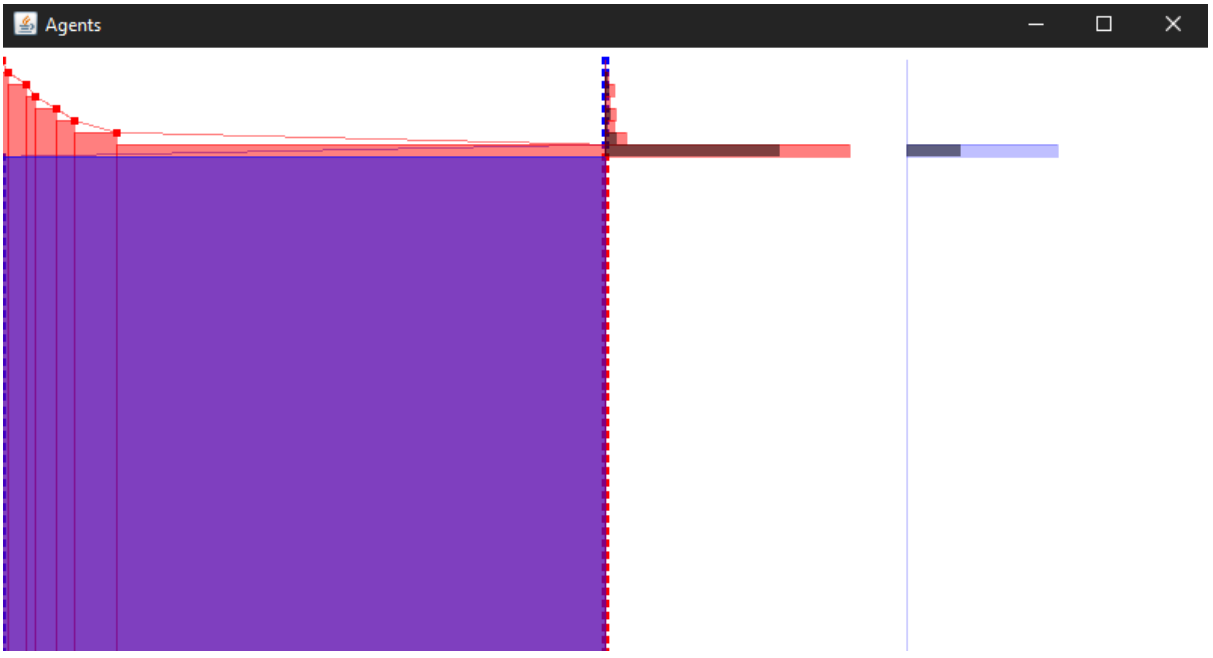




The opposite is true. In the example below, there are 200 buyers and 200 sellers instead. We can see that the price converges to a higher point, with many buyers willing to pay high prices for the item because there are so few sellers. We can also observe a higher rate of unsuccessful purchases in black in the histogram graph.

This is a market where are many more buyers (200) than sellers(100) , gradually collapsing towards equilibrium





The market in its total equilibrium at a higher price. We see many buyers still desperate to obtain the good and trade even at a price higher than the equilibrium.

Acknowledgements

In this extension I was heavily inspired by my EC133 course at Colby. My idea of how to abstract the problem also came from the Primer Youtube channel. I also attended Prof. Bender's office hour for advice on organizing my code to OOP. Beyond that, I only relied on Java documentation and myself to write these code.

https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html
https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html
https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
https://docs.oracle.com/javase/8/docs/api/java/util/Random.html