Abstract

In this project, I practiced the implementation of LinkedList, which is a generic Node-based data structure that stores information using pointers/references. In the Lab, we made a LinkedList class, which contains a Node class which is served as a container of the information, as well as an Iterator class that can help us access the information along the LinkedList with higher efficiency.

This project used LinkedList to implement an agent-based simulation, where Agents are positioned in a continuous 2D space and become active/inactive according to their own class rules.

Task solutions:

Agent class: the Agent class serves as a parent class for the two types of Agents that I made later in this project. It contains the double fields x and y that store the positions of the agent, and basic methods such as returning and setting the position of the agent.

I tested the class by creating an Agent object in the main method and set its position to make sure the methods work. But before I started writing its child class (SocialAgent), I changed the Agent class to an abstract class to ensure that its child class will override its abstract methods – updateState() and draw().

(pic of abstract class)

SocialAgent class: this is the first child class of the Agent class. It has a private field that stores if the agent has moved in the update, and calls the parent constructor using super (x0, y0) and overrides the parent abstract methods draw and updateState (updated later). The draw method draws a circle at the Agent’s location and fills it with a blue if the moved field is false, and fills the circle with cyan if the moved field is true.

Landscape: the Landscape class has three fields: the width and height of the landscape and a LinkedList object that stores the agents. Apart from the basic methods that return or set the dimensions of the Landscape, the class uses the implementation of a LinkedList for the methods addAgent() and uses the Iterator in LinkedList to return loop through the agents stored in the LinkedList to get a list of neighbors at a position and call the draw method of all agents in the list.

The getNeighbor() method returns an array list of all the agents within a radius r from position (x0, y0). It loops through every agent in the list and calculate its distance from (x0, y0) using Math.sqrt(). If the value falls into the range of the radius, the agent is added into the arraylist neighbors.

Visualization: below is a screenshot of the first visualization of the agent landscape.

Then I added the updateState() method in SocialAgent class, which takes in two parameters, an Landscape object and a double r that decides the radius range of the neighbors. It first generates an ArrayList using the geNeightbors() method of the Landscape class. Because when the method generates the list of agents within a certain radius to a specific agent, it includes the same agent itself, so if the condition is that an agent has more than 3 neighbors, the size of the neighbor list should be larger than 4, as the snippet shown below.

(pic of neightbor.size() >4)

This algorithm will also include the agent exactly at location (x0, y0), but not agent that is used as the reference.

(to be continued)

The updateAgents() first returns a shuffled list of all the agents and loop over each agent in the list and calls their updateState() method in the SocialAgent class.

SocialAgentSimulation: this simulation is modeled after the LifeSimulation in a previous project. Its main method asks for three command line arguments that respectively set the dimension of the Landscape and the number agent initialized. Then it loops for 200 times to call the updateAgents() and repaint the landscape. Below is gif showing the changes in a 500\*500 landscape with 200 SocialAgents.

(gif of social simulation)

CategorizedSocialAgent: this is a child class that inherits the SocialAgent class, where its uniqueness is a field category that differentiates the categories of social agents. It also override the draw method that fills different category of agents with a different color.

The class implements a new update rule that can be coded using similar algorithm of the updateState() method in its parent class. It also first generates a list of the neighbors of the referencing agent, and then loops through it to count the number of agents in the same category. And use similar if statements to update the moved field of the agent.

CategorizedSocialAgentSimulation: I simulated the CategorizedSocialAgent with two types of agents (blue and gray).

Small radius (20):

Medium radius (40):

Large radius (80):

I found that the larger the radius is, the lesser agents will be active after 200 iterations because an agent will have more neighbors if the radius is larger and thus has less possibility to move.

In addition, the smaller the radius is, the more likely the agents are to form into clamps. On the left is the state of the simulation with a radius of 10, which shows that the inactive agents have formed into small clamps that consist of 3-4 agents; versus on the right where the radius is 40, in which the inactive agents are larger in number but scattered around the landscape (probably because there are too many inactive agents).

Extensions:

I first made a class CustomizedSimulation, which put each the separate simulations (SocialAgent and CategorizedAgent) into one class, and created JOptionpane to let the user control the which type of simulation they want to do, the radius of simulation, the number of agents initialized on to the landscape.

The main method of this class will first ask the user to enter the dimensions of the window, which is a square, and the show an option dialogue that provides the user with two simulation options, as show below.

If the user chooses “Social agent only”, the program will ask them for input of the number of social agents initialized onto the landscape and run the simulation by creating a SocialAgentSimulation object. Because the landscape field of the object is private, I had to create a public method getScape() which returns the landscape field of a SocialAgentSimulation object.

If the user chooses “Categorized Agents”, the program will ask them to input of the number of agent category that they want to use in the simulation (maximum 3), and the number of each category of agents, and the run the simulation. Below is the screenshot of a simulation with the 3 categories, each of which has 100 agents and a radius of 20.

In the class GenericArray, I used array to implement a similar list that has the same function as the LinkedList class.

I first created an array of Objects, which will be cast over the type of elements that I wish to add into the array.

The I added several basic methods that are in the LinkedList class such as addLast, addFirst, add (index, item), get, remove, clear.

Both addLast and addFirst first check if the list is full, and double its size if so by creating a new array that is twice of the size of the original array and copy over all the data in the old array.

addLast will then add the new element to the end of the list whereas addFirst will the shift all the elements down along the array to make space for the new element at the beginning of the list, as shown below.

The general add method takes in an index of a position in the array and an element. It also first checks if the array is full and starts to shift all the elements after the index position forward by 1, and then add the new element to the index position.

The clear method empties the list but because the size N is the only reference we use to access the number of elements in the list, I simply set N to 0 in this method.

Because the element of the array is generic, whenever a method returns a generic object, there was a warning for “unsafe operations” when I compiled the class. To suppress the warning, I added @SuppressWarnings(“unchecked”) in front of the remove and get method. I tested the GenericArray class, I created a main method that used its methods to make sure it works.

Then I tried to use the GenericArray class instead of LinkedList to hold the agents, so I created the ArrayLandscape class, which is also a child class of Landscape. The difference is that ArrayLandscape has a field of an GenericArray object instead of a LinkedList. It also overwrites its parent’s methods such as addAgent and getNeighbors, especially in the for each loop (as shown below) because I did not implement an iterator in the GenericArray class. To test its function, I created an ArrayLandscape object in the main method just like when I tested the Landscape class – adding several agents and let it prints out the neighbors. The result (as shown below) shows that the GenericArray list works the same as the LinkedList.

Conclusion:

In this project, I learnt how to create and use a LinkedList and use it simulation continuous movements of objects around the landscape. I also learnt how to create a generic array and make it to have to same function as an arraylist and LinkedList.

Credits:

Caleb

Prof. Maxwell and his lecture notes.