Agent-Based Modeling of Urban Society and Interactions

Andrew Imm
TJHSST Computer Systems Research Lab
Q1 2009-2010

October 27, 2009

Abstract

Current systems used to model the spread of disease treat populations as single entities, and neglect the actions of individuals. By developing an agent-based simulation focused upon the accurate modeling of social interactions seen in an urban environment, a testing bed that resembles a modern city arises. This testing bed — with its accurate modeling of day-to-day interactions within a city — provides a far better system to use when developing epidemiology simulations. Using an implementation of goal-oriented agents who are guided by a number of variables that make up their "personality," this program attempts to create this urban model and use the system to run a number of epidemiological studies.

Keywords: Agent-based, urban simulation, social networks, urban society, interactions

1 Introduction

By taking into account the needs and motivations of people, a realistic simulation of an urban society can be created. This project builds a system of agents who navigate their city according to individual schedules, interact with others to gather information and satisfy a need for socialization, and ultimately make their decisions through a complex system of algorithms that take into account the various aspects of an agent's personality. The system is also designed to be extensible — effectively, it can be used to testff the effect of a stimulus upon a bustling urban environment. With the simulation completed, this project will look into the implementation of epidemiology studies in this environment. These studies will follow the virtual citizens of the simulation after a virus is introduced into the city. The agents' interactions with each other provide a vector for viral transfer, providing a chance to study how the virus

spreads along social networks. Finally, the effectiveness of various quarantine methods can be analysed in order to create a contingency plan that can be implemented in the real world.

2 Background

In the field of epidemiology, most models used to predict the outcomes of plagues and epidemics are math-based. They treat the entire population of a region or nation as a single entity. This take on the problem of studying the spread of disease has one major downfall — it assumes that all members of the population have similar behaviors. If any stratification is done to divide the population into subgroups, these are generally only related to succeptibility to the disease in the study. In other words, the unique characteristics of individuals are lost. An agent-based model, while more processor-intensive than a strict mathematical model, brings into play this individuality. However, past models that took an agent-based approach were very simplistic. For instance, viral modeling has been popular in the TJHSST Computer Systems Research Lab for years, but nearly every project has involved agents moving randomly within a closed, featureless environment. Effectively, these simulations resembled nothing more than an experiment of specialized bacteria moving around in a petri dish — hardly an experiment that can be used to make generalizations or conclusions about a human population. For such conclusions, the agents in the model must act as humans do; this necessity provides the reason for developing an accurate simulation of an urban society.

3 Development

The development of this project has been divided into three different groupings of code. The first piece of code represents the actual simulation; it is this code that is used when the simulation is finally run. The second piece of code is composed of various tools and helper programs that are used to expedie the process of project development. The third and final piece of code includes any tests that are run in order to analyze the stability and efficiency of the simulation. Although only the first group of programs is used in the final simulation, the other groups ensure that the final product is developed as quickly and accurately as possible.

3.1 Simulation

The simulation makes up the majority of the code written for this project. Initially, it loads a map file that tells the simulation how to construct the city. It then loads an agent file which tells the program how to configure the virtual city's population. Each agent is assigned a name, a schedule, and a "personality" — a set of preferences that dictate how likely the agent is to perform various actions. Once the world and its inhabitants have been built, the program initializes its internal clock to 12:00 midnight

on day 0. As the model runs, the virtual clock updates, and eventually agents wake up. As time progresses in the simulation, the agents go about the daily routines dictated in their schedules, navigating the city using the simulation's path-finding algorithm. Inherently, the agents encounter others throughout the day, and begin to remember other agents whom they often see. These memories of acquaintances are the beginning of the agent's social network: a stored list of friends and colleagues that allows the agent to keep track of people it has already met. The agent's list of acquaintances also keeps track of how well the agent knows others; this data is used by the agent to decide whom to interact with. As the simulation ages, the virtual city begins to resemble its real counterpart. Agents become established in their routines, and have dependable networks of friends that keep them socially active. At this point, a range of tests can begin in the simulation. Manipulation or addition of variables — such as a virus — at this stage ensures that the results resemble a real-world reaction as best as possible.

3.2 Additional Programs

This project requires the creation of other programs that speed up the process of development. For instance, the simulation uses complex files to store maps, and the easiest way to create these maps is with a secondary program. The map builder allows the user to create maps with a graphical interface that displays the map as it will appear when the simulation is run. While such programs are not used in the final simulation, the products they create enable the experiments of this project, and these programs are therefore crucial to the completion of this project.

3.3 Tests

In order to improve the efficiency of the program and determine the optimal scale of the simulation, various tests will be used to analyze the program's internal algorithms. These tests will import methods from the simulation and run them on large sets of data to determine their practical limits. One algorithm that is very important to test is the path-finding method. This is one of the most frequently-called methods in the simulation, and it needs to be tested to determine how many times it can be run per program cycle before a noticeable lag occurs. Testing it again and again with large sets of data will help to determine this number. These tests are like the additional programs in that their code does not appear in the final project. Instead, they are used to develop and refine the simulation so that its final state is the optimal version.

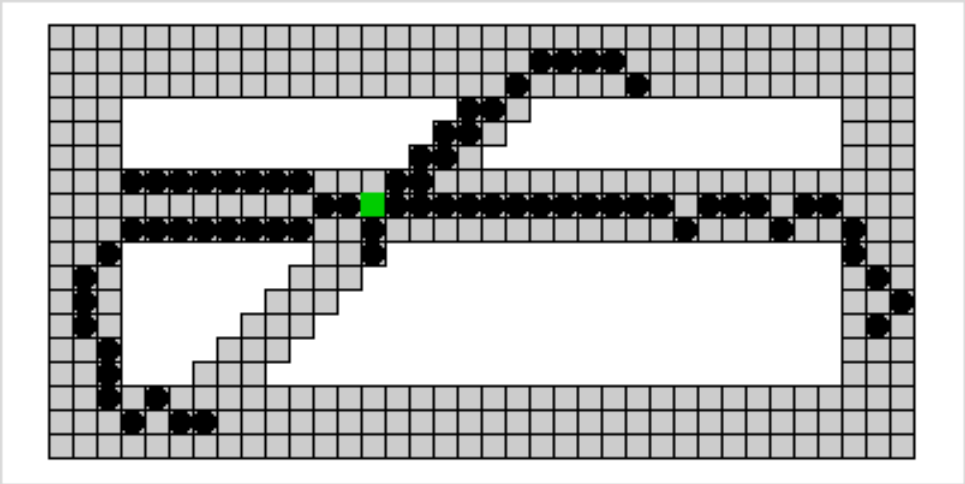

Figure 1: Agents navigating towards a green square in the center of the map

4 Discussion

Currently, the simulation is at a stage where it can load a map and populate the world with random agents. The agents then can be told to navigate to any square on the map by clicking. The simulation also keeps track of virtual time, although the internal clock is not currently used by the agents to guide their actions. The main purpose of the simulation at this point is to demonstrate the path-finding algorithm. The other large piece of code is the map builder, which currently creates maps with far more features than those that are used in the simulation at this point. The map builder features a graphical user interface that makes creation of the map much easier than editing a text file by hand.

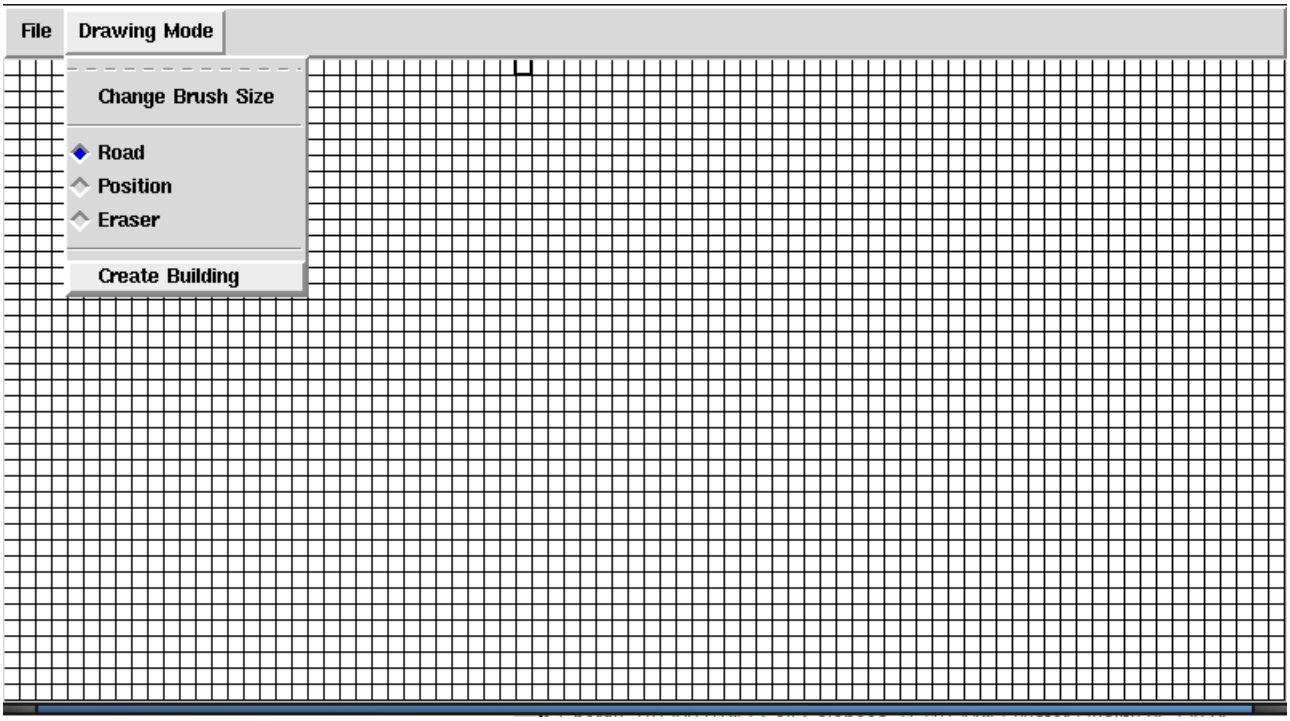

Figure 2: The map builder user interface

Appendix A. Code Samples

This code makes up the path-finding algorithm found in the Agent class.

```
# Moves an agent directly to the given coordinates
def goto(self,c,d):
        z = str(self.x)+"."+str(self.y)
        self.x = c
        self.y = d
        self.map[str(c)+"."+str(d)] = self.map[z]
        self.map[z] = None
        self.canvas.coords(self.disp,c*self.size,d*self.size,(c+1)*self.size,(d+1)*self.size)

#Tells the agent to begin the path-finding process in order to reach (c,d)
def navigate(self,c,d):
        if self.x == c and self.y == d:
        return
        self.steplist = []
        self.steplist = self.findpath(self.x,self.y,c,d)
        print self.steplist
        if self.steplist:
                self.steplist.pop()

#Returns a list of possible moves surrounding a given square on the map
def getmoves(self,a,b):
        if not (str(a)+"."+str(b) in self.map):
                return []
        movelist = []
        for xx in range(-1,2):
                for yy in range(-1,2):
                        if not (xx == 0 and yy == 0):
                                keystr = str(a+xx)+"."+str(b+yy)
                                if keystr in self.map and self.map[keystr] == None:
                                        f = 10
                                        if xx != 0 and yy != 0:
                                                f = 14
                                        movelist.append([keystr,f])
        return movelist

#Begins the A* Search used in path-finding
def findpath(self,a,b,c,d):
        open = {}
        closed = {}
        mystr = str(a)+"."+str(b)
        closed[mystr] = ["START",0]
        moves = self.getmoves(a,b)
        min = 999999999
        mindex = "-1"
        if not moves: return []
        for m in moves:
                j,k = m[0].split(".")
                j = int(j)
                k = int(k)
                open[m[0]] = [mystr,m[1]] #[parent,f-value] (we can calculate h at any time from f)
                md = self.mdist(j,k,c,d)
                if m[1]+md < min:
                        mindex = m[0]
```

```
                        min = m[1]+md
        return self.pathhelper(mindex,c,d,open,closed)

def mdist(self,a,b,c,d):
        return (math.fabs(a-c)+math.fabs(b-d))*10

def pathhelper(self,mystr,c,d,open,closed):
        a,b = mystr.split(".") # current square
        a = int(a)
        b = int(b)
        closed[mystr] = open[mystr]
        del(open[mystr])
        if a == c and b == d: return self.extractpath(mystr,closed)
        gg = closed[mystr][1]
        mm = self.getmoves(a,b)
        for m in mm:
                if not (m[0] in closed):
                        if not (m[0] in open):
                                open[m[0]] = [mystr,gg+m[1]]
                        elif gg+m[1] < open[m[0]][1]:
                                open[m[0]] = [mystr,gg+m[1]]
        min = 999999999
        mindex = "-1"
        if not open: return []
        for m in open:
                j,k = m.split(".")
                j = int(j)
                k = int(k)
                md = self.mdist(j,k,c,d)
                if open[m][1]+md < min:
                        mindex = m
                        min = open[m][1]+md
        return self.pathhelper(mindex,c,d,open,closed)

def extractpath(self,mystr,closed):
        str = mystr
        steps = []
        while str != "START":
                steps.append(str)
                str = closed[str][0]
        return steps
```

References

[1] Lester, Patrick. *A* Pathfinding for Beginners.* Jul. 18 2005. Web. Oct. 3 2009. ¡http://www.policyalmanac.org/games/aStarTutorial.htm¿.