Agent-based modeling of urban society and interactions

Andrew Imm
TJHSST Computer Systems Research Lab
2009-2010

June 16, 2010

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 environment — 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 unique and individualistic "personalities," this program attempts to create this type of urban model and use the system to run a number of epidemiological studies. Once the virtual society is established within its routines, and a network of social relationships has developed within the city's inhabitants, the simulation will reach an autonomous running state where it can develop and grow on its own. At this point, various scenarios can be implemented to draw conclusions about human populations in urban environments. In one such scenario, the introduction of a simulated influenza virus will help determine how an urban population reacts to such an infection, and how the disease is likely to spread through the city.

Keywords: Agent-based, urban simulation, social networks, urban society, interactions, disease, epidemiology, modeling, simulation tools

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. Although guided by their own rules, the actions which the agents take in response to their environment and each other lead to an emergent behavior in the overall society. Such a system is individual-driven, because agent actions are not globally controlled by a single method, but rather by the preferences and traits of individual agents. The system is also designed to be extremely extensible and modifiable — effectively, it can be used to test the effect of any representable 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 analyzed 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 susceptibility 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.

To fill this gap in epidemiology research, one of the main goals of this project was to develop a testing environment where urban scenarios can be implemented, and virtual populations can be experimented upon. The simulation environment provides a series of tools which can be used by scientists and researchers to customize and create their own simulations; the simulation system can be modified and manipulated as the user sees fit, from the drawing of maps to the creation of agent schedules and custom interaction methods. An epidemiologist studying viral spread can write interaction methods that cause a virus to spread from one agent to another, while a sociologist studying the proliferation of rumors can explore them through methods that transmit knowledge between agents. Items as intangible as thoughts, and as tangible as money can be transferred from one agent to another with the ability to customize the simulation at will.

Beyond the ability for customization, the in-depth qualities of realism in this simulation tool will provide an effective testing environment for epidemiology studies. Because the simulation is designed with a focus on individual interactions, the program works well for simulating the spread of disease from one individual to another. Using agent-based models to analyze the spread of disease is something that has been explored before by a few scientists, but the fact is that it is not a mainstream method of epidemiology modeling. Dr. Stephen Eubank from Virginia Polytechnic Institute is one of the leaders in the field of agent-based epidemiology modeling, and his projects explore the spread of many diseases in a variety of environments. For instance, his "Modelling Disease Outbreaks in Realistic Urban Social Networks" takes on a similar problem as my project does — exploring the spread of disease through a social network. However, his program does not take into account all of the environmental aspects of simulating a city that my project does. With the extra features found in my simulation model, I hope to establish a platform that can accurately assess the quality of various quarantine methods when dealing with infectious diseases.

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 system; 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 expedite 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. At run time, it is provided with the location of a simulation file, which tells the program where to look for each component of the simulation. The data needed to run the model is divided into various files: a file that defines the world, a file that defines agents and their schedules, and a file which contains the code for custom interaction methods. Once all of these program files have been located, the simulation loads the map file in order to properly construct the city environment. With the empty world now loaded into memory, the program then loads an agent file which tells the computer how to configure the virtual city's populous. 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. Using built-in methods, they can be ordered to travel to different buildings or areas of the map, and are able to find their own space to inhabit in each building they visit. 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.
The simulation is initialized with a map that is formatted in the following way:

```
[width]
[height]
buildings:
[Building]=[x1],[y1],[x2],[y2]
map:
0000000000000...
0000111111111...
0000111111111...
................
................
```

The first lines contain [width] and [height], which are the width and height (respectively) of the map in terms of grid squares. The next line contains the header "buildings:", which denotes that building definitions will follow. These building definitions are structured as seen, where [Building] is the internal name of the building that is used to refer to the specific area of the map, and [x1],[y1] and [x2],[y2] are the coordinates for the top-left and bottom-right corners (respectively of the building area. After all building definitions have been listed, the next line contains the header "map:", which denotes that the actual map data will follow. From that point, each line of the file represents a row on the map, where each individual character is a number representing the terrain type at that data point.
The agents are then loaded using a file that is formatted in the following way:

```
[Agent Name]
([x],[y])
{
-[time] [location]
................
```

```
. . . . . .
}
[characteristic]=[value]
. . . . . .
. . . . . .
.
.
.
```

The first line contains the agent's name, which is enclosed by square brackets ("[" and "]"). The next line contains the agent's initialization coordinates, where [x] is the x-coordinate and [y] is the y-coordinate of the point on the map where the agent begins its life. The next few lines, between the "{" and the "}" contain the agent's schedule. Each line of the schedule contains the [time] at which the agent needs to navigate to the [location]. The final lines after the schedule contain various characteristics that can be assigned to the agent, where [characteristic] is simply the name of the characteristic, and [value] is a floating-point number from 0.0 to 1.0.

Agents navigate the map according to their schedule, stepping through it to check whether they should be moving to a new location with each time increment. Their navigation method is a standard A* search based on the grid of the map, where horizontal and vertical movements of one square constitute a cost of 10, and diagonal movements of one square constitute a cost of 14 (10 times $\sqrt{2}$). Since agents are constantly moving, they are not treated as obstacles by the navigation method. However, multiple agents cannot occupy the same square on the map, so agents who are directly neighboring the currently-navigating agents are temporarily seen as obstacles. This behavior allows many agents to attempt to reach a single point in an effective realistic way: as the crowd gathers around the goal point, agents fill in the gaps in the crowd to form an approximately circle-shaped mass of agents around the point.

When agents find themselves neighboring each other, they always take notice of each other. They also record how often they have encountered certain individuals, so as to remember acquaintances and how well they know each other. If they see another agent whom they know well, they are given the opportunity to interact with that agent. Such interaction can involve a variety of actions, depending on the structure and application of the simulation. Agents might wish to share specialized knowledge with each other, or they

might unknowingly spread a virus through interaction. The specifics of these interactions, such as how they're actually carried out with program variables, are all defined in one of the files imported at the beginning of the simulation. These interactions are enabled by agent preferences; preferences with names that match the names of methods specify how likely an agent is to perform those actions when it encounters another agent. Through these custom methods and characteristics, the shape of interactions within the simulation evolves.

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. The program can also be used to create buildings on the map, and such buildings are used by the simulation in order to determine where to send agents. The map builder also features a variety of other features that can be useful for development, including distance calculations and map-printing abilities. These programs are external from the final simulation program, but they are necessary for producing the components of any simulation, and therefore are a part of the suite of tools which have been produced to develop urban simulation models.

3.3 Tests

In order to improve the efficiency of the program and determine the optimal scale of the simulation, various tests have been used to analyze the program's internal algorithms. These tests imported methods from the simulation and ran them on large sets of data to determine their practical limits. One algorithm that was very important to test was the path-finding method. This is one of the most frequently-called methods in the simulation, and it needed to be tested to determine how many times it could be run per program cycle before a noticeable lag occurred. Testing it again and again with large sets of data will helped to determine this number. It was also through these tests

that serious flaws were found in the path-finding algorithm that prevented agents from effectively reaching their goals and navigating efficiently. Agents eventually ended up in traffic jams that slowed city movement to a standstill, ruining the simulation. Analysis from these tests eventually determined that agents needed to avoid treating each other as obstacles unless such categorization were absolutely necessary. In later development stages, tests were used to determine inefficient segments of code. To improve memory and processor use, variables were reused to avoid duplication of efforts and to conserve the amount of memory consumed by the program. Tests enabled the program to run at top speed, and led to improvements in every stage of the program's main loop.

4 Discussion

The simulation, now completed, is able to load a simulation file which dictates where all of the variable pieces of the simulation may be found. The specifications laid out in the map file give a variety of details as to world terrain and the placement of named "buildings," which are used to refer to contiguous areas of the map. The program creates the agents as they are specified within the agent file, each with its own name, schedule, and personality attributes. When the simulation begins, the program keeps track of virtual time, and uses this clock to time and control the actions of agents. As it is, the agents within the simulation can continue to navigate the map indefinitely, moving to the various destinations indicated in their schedule. At they follow their daily schedules, they have the opportunity to interact with neighboring agents, potentially transmitting items through this interaction. Tests have been performed which involved the transfer of knowledge through this medium, and it has been used in larger experiments to examine the spread of disease. The other large piece of code is the map builder, which creates maps with a variety of useful features that are used in the simulation. The map builder features a graphical user interface that makes creation of the map much easier than editing a text file by hand. Dialogs for creating buildings on the map, as well as defining new, custom types of terrain allow for a feature-filled map creation environment.

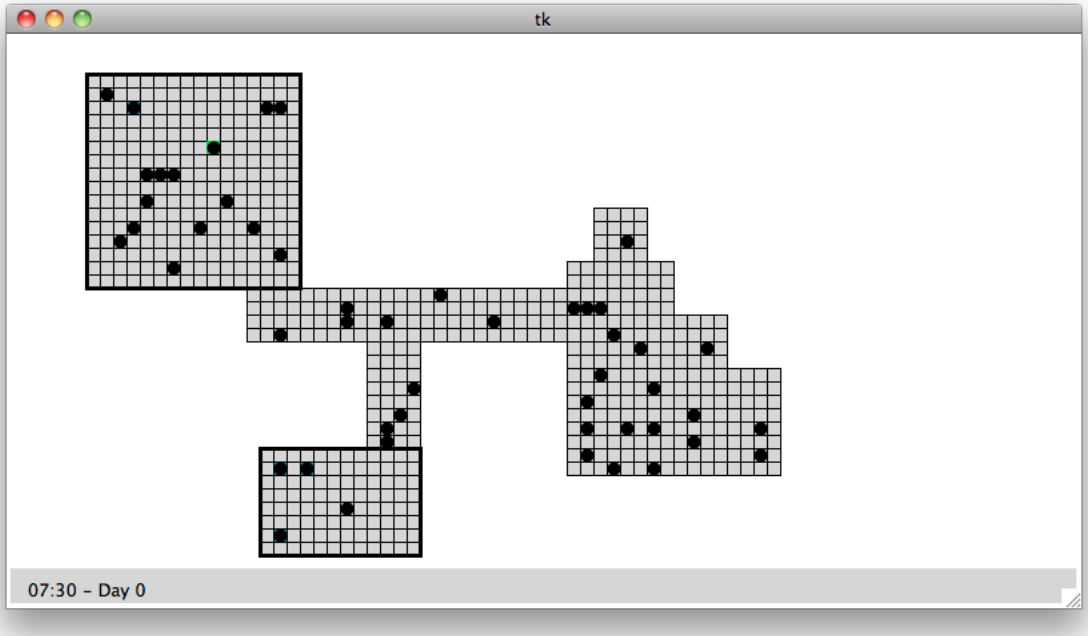

Figure 1: Agents navigating in a large map environment

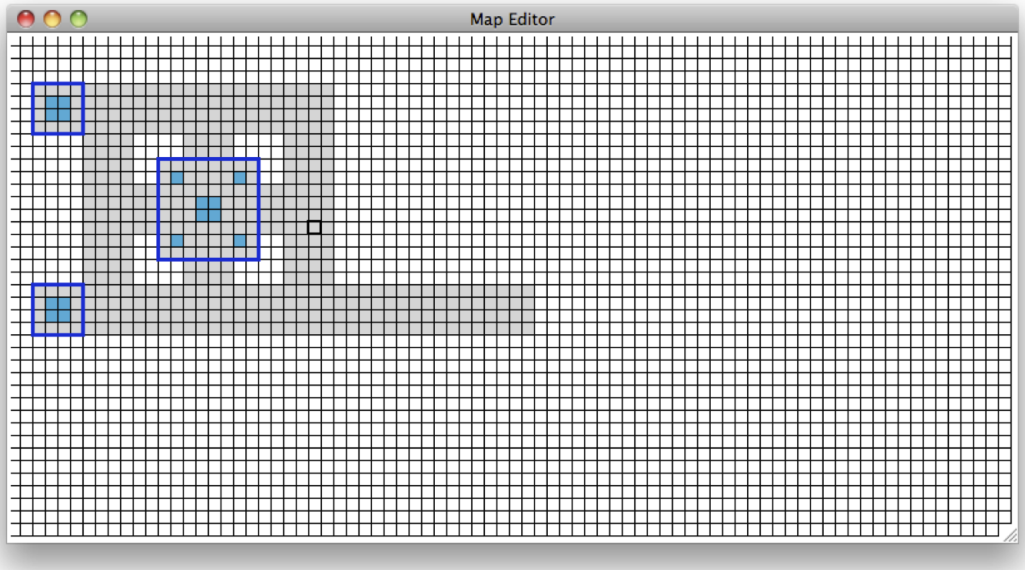

Figure 2: The map builder user interface

5 Tests and Results

Once the simulation system was developed, its ability to run a number of useful simulations was tested. Starting as any third-party user would, a simulation was begun with the creation of a map and a set of agents with certain properties. For each of the tests, special interaction methods were written which defined how the simulation ran. In the knowledge test, agents were told to share knowledge with each other. In more complicated tests (such as the one examining the spread of a virus), agents were given a variety of interaction choices. Each of these methods was written, and the final simulation file was created, including its pointers to each of the various component files. The processes involved in two of the major simulations implemented here are listed below. It should be noted that the purpose of this project was to create a system that could be used for simulations, and these tests served to tests the viability of the project in this context.

5.1 The Knowledge-Transfer Test

This first test examined the transfer of knowledge between agents, such as in the modeling of the spread of a rumor. Knowledge is shared easily, and is not lost after it has been obtained. As such, the entire population should eventually grow to obtain the knowledge — the question is, by what point in time will they do so? This simulation gave agents only one interaction method: the transfer of specific knowledge, which in this case was the location of a secret building. Agents were all told in their schedules to navigate to this building at noon, but initially no agents knew where the building was. One agent was initialized with the knowledge of the secret building, and was given the ability to share it with everyone else in the world. Each day, more and more agents would travel to the building at noon, because more and more of them were learning where the building was located. It took a while for agents with nocturnal schedules to gain that knowledge, but after 100 runs of the simulation, it was shown that it only took around 4-5 days for every agent to obtain the knowledge. The model ran successfully, and without any modifications to the original program. Every customized aspect was coded using the user-created external files that were imported at runtime.

5.2 The Viral-Spread Test

This later model explored how agents behave in an urban society where there is a virus in the population. Taking advantage of custom actions and interactions, this simulation gave agents many options when it came to encountering others. Each agent had a unique probability of performing each action, and the many actions served to both spread the virus and improve relationships between individuals. For instance, talking would raise the level of familiarity a little bit, while telling a joke would increase the level much further. This test explored how two different quarantine methods worked: one where agents relied on their own sense of self-preservation to avoid contact with others (and possible contraction of the virus). The other method placed a curfew on the city, limiting travel hours to certain times of day. It was found that the two methods were equally effective: it took approximately 22-23 days for every agent in the world to contract the virus in both scenarios. Again, the simulation program demonstrated its ability to be used for a variety of modeling purposes.

6 Conclusions

This project initially began as an examination into the spread of a virus through the implementation of a realistic urban simulation environment. As it progressed though, the focus changed to the creation of such a system that was dynamic and modular enough to be used for a variety of diverse modeling purposes. It was recognized that a system that can be easily changed and manipulated holds far more value in the field of agent-based modeling and simulation than a single-purpose program ever would. Through the year, the program was developed and expanded, making improvements in efficiency, realism, and the ability to customize simulations in a variety of ways. The final project was put to the test, creating two different models and exploring how they behaved; in the end, it was found that the simulation environment worked well for its newfound purpose, and was an effective way to implement models with widely differing properties.

Appendix A. Code Samples

This code contains the definition of the Agent class, which includes all of the complex pathfinding code.

```
def __init__(self,id,x,y,size,canv,map,attr):
    print 'Creating Agent',id
    self.id = id
    self.x = x
    self.y = y
    self.size = size
    self.canvas = canv
    self.map = map
    self.attribs = attr
    self.steplist = []
    self.disp = canv.create_oval(x*size,y*size,(x+1)*size,(y+1)*size,width=0,fill='#000000')
    self.waitcount = 0
    self.waitmax = 3
    self.tempchange = {}

def step(self):
    #print self.id
    #raw_input()
    if self.steplist:
        ss = self.steplist.pop()
        if self.map[ss] == None:
            c,d = ss.split(".")
            self.goto(int(c),int(d))
        else:
            self.steplist.append(ss)
            self.waitcount += 1
            if self.waitcount >= self.waitmax:
                self.waitcount = 0
                if len(self.steplist) > 1:
                    nbl = self.getmoves(self.x,self.y)
                    for nbi in nbl:
                        nb = nbi[0]
                        if self.map[nb] != None:
                            self.tempchange[nb] = self.map[nb]
                            self.map[nb] = -1
                            #print nb,'is INVALID'
                    #print self.steplist
                    destx,desty = self.steplist[0].split(".")
                    #print self.id,'is RENAVIGATING. Destination is',destx,desty
                    self.navigate(int(destx),int(desty))
                else:
                    self.steplist = []
                for tnb in self.tempchange:
                    self.map[tnb] = self.tempchange[tnb]
                self.tempchange = {}
                #print self.id,'is DONE renavigating'

def goto(self,c,d): #physically moves the agent to the specified coordinates
    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)

    def navbuilding(self,buildingname): #navigates to a named building
        if buildingname in positions:
            plist = []
            for p in positions[buildingname]: plist.append(p)
            while len(plist) > 0:
                ri = int(random()*len(plist))
                keystr = str(plist[ri][0])+"."+str(plist[ri][1])
                if self.map[keystr] == None:
                    a

    def navigate(self,c,d): #navigates to the specified coordinates using A*
        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()

    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] != -1:
                        f = 10
                        if xx != 0 and yy != 0:
                            f = 14
                        movelist.append([keystr,f])
        return movelist

    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
```

This code contains the map- and agent-loading code used by the simulation to read those system components from external files.

```
def loadagents(path):
    global lastagent
    print 'loading agent from file:',path
    file = open(path).read().split('\n')
    info = []
    k = 0
    while k < len(file):
        print k, '-',file[k]
        if file[k] == '':
            print 'CREATE_AGENT',lastagent
            genagent(info,lastagent)
            lastagent += 1
            info = []
        else:
            info.append(file[k])
        k += 1
```

```
def genagent(info,count):
    if len(info) == 0 or info[0][0] != '[':
        return
    name = info[0][1:-1]
    c = info[1][1:-1].split(',')
    coords = [int(c[0]),int(c[1])]
    attribs = {}
    k = 2
    while k < len(info):
        line = info[k].split('=')
        attribs[line[0].strip()] = float((line[1].strip()))
        k += 1
    newagent = Agent(count,coords[0],coords[1],grid_size,canvas,map,attribs)
    keystr = c[0]+'.'+c[1]
    agents.append(newagent)
    map[keystr] = newagent

        def loadmap(filename):
            global w,h
            global time
            print 'loading map:',filename
            file = open(filename).read().split('\n')
            data = []
            w = int(file[0])
            h = int(file[1])
            canvas.config(width=grid_size*w,height=grid_size*h+24)
            k = 2
            while k < len(file) and file[k].lower() != 'buildings:':
                k+=1
            if k >= len(file): return
            k += 1
            while k < len(file) and file[k].lower() != 'map:':
                try:
                    print file[k]
                    a = file[k].split('=')
                    bdefn = a[1].split(',')
                    buildings[a[0]] = []
                    for x in bdefn:
                        buildings[a[0]].append(int(x))
                except:
                    print 'Failed to read building...'
                k += 1
            print buildings
            k += 1
            l = 0
            while k < len(file):
                j = 0
                for x in file[k]:
                    data.append(int(x))
                    if int(x) != 0:
                        keystr = str(j)+'.'+str(l)
                        map[keystr] = None
                        if int(x) == 2:
                            for bd in buildings:
                                if within_rectangle(j,l,buildings[bd]):
                                    if bd in positions: positions[bd].append((j,l))
```

```
                                else: positions[bd] = [(j,l)]
                                break
                        canvas.create_rectangle(j*grid_size,(l)*grid_size,(j+1)*grid_size,(l+1)*gri
                    j += 1
                k += 1
                l += 1
            print positions
            for bd in buildings:
                b = buildings[bd]
                canvas.create_rectangle(b[0]*grid_size,b[1]*grid_size,(b[2]+1)*grid_size,(b[3]+1)*g
            time = [0,0,0]

        def within_rectangle(x,y,rect):
            return (x >= rect[0] and x <= rect[2] and y >= rect[1] and y <= rect[3])
```

References

[1] Conte, R. *Agent-Based Modeling for Understanding Social Intelligence.* Proceedings of the National Academy of Sciences of the United States of America, 2002.

[2] Eubank, Stephen. *"Modelling Disease Outbreaks in Realistic Urban Social Networks."* Nature 13 May 2004: 180-184.

[3] Jiang, Bin. *Agent-Based Approach to Modelling Environmental and Urban Systems Within GIS* University of Gavle, Sweden. Department of Geomatics.

[4] Kretzschmar, M, and Morris, M. *Measures of concurrency in networks and the spread of infectious disease.* Math Biosci. 133(2): 165-95.

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

[6] Makowski, Michael D. *An Agent-Based Model of Mortality Shocks, Intergenerational Effects, and Urban Crime.* George Mason University, Department of Economics. 11 Nov 2005.