

# Mouse in a Maze

Michael Huynh

*AP Computer Science, Upper Darby High School*

June 2004

This paper introduces a model of a mouse moving through a maze in the Java programming language and seeks to improve upon mouse movement and decision making by having the mouse make decisions like a human. These improvements include consistent rules in movement, location marking, and location mapping. By giving some intelligence and memory to the mouse, it will solve mazes more efficiently.

## 1. INTRODUCTION

Moving a mouse through a maze is a classic computer science problem that deals with concepts of matrix manipulation and indexing, object states, grid checking, and artificial intelligence.

In this study, the traditional mouse in a maze problem is solved in the Java programming language with the resources of a graphical user interface. Also, the mouse will include elements of artificial intelligence to increase its efficiency in finding the solution to the maze.

## 2. RESOURCES

Simulating a mouse moving through a maze has two major programming focuses. The first is the algorithm or move area, and the second is the aesthetics of the program. Aesthetics concern the graphical representation and display of the moving mouse as it solved the maze. Although important for debugging the code and verifying the results, the display is less important than the actual code that teaches a mouse object how to move. Because of this, the aesthetics of the program were outsourced to pre-written graphical display solutions. It was therefore decided that the programming solution would incorporate the Marine Biology Case Study.

The Marine Biology Case Study (MBCS) by the College Board for Advanced Placement Computer Science includes a set of Java classes that simulate fish moving in an aqueous environment<sup>1</sup>. Because the case study already includes a graphical user interface with graphical representation of the environment and fish objects, it would be very useful in simulating a mouse moving through a maze. Consequently, the MBCS was programmed for a certain degree of flexibility; it definitely could be modified and extended for a new purpose.

## 3. GENERAL SOLUTION

To solve the problem of finding the exit or solution in a maze, imagine being dropped in an unknown maze. What would a human do to solve it? A human might realize that randomly choosing paths in a maze could lead to disorientation and the possibility of never solving the maze. Therefore, the human might adopt a strategy of placing a hand on a wall and following it through the maze. Using this strategy, if there is a solution, the human will eventually find it. Mirroring this strategy in code was simply using a set of if statements that always checked to move east relative to the facing direction before moving in other directions.

Moving through a maze without a reference, a human could easily be lost and become trapped in the maze for a long time. To determine position, the easily strategy would be to leave a mark or an object in each area so that the human could identify if a particular area has already been visited. In the mouse maze program, an object called BreadCrumb was used for this purpose.

Eventually, the human might further his or her sense of position by recording all locations on a map relative to the starting point. By doing this, the human can avoid backtracking and can view the whole known area from a glimpse. When the human finds the exit, he or she will now have a method of finding the exit quicker next time if placed back into the maze. The memory of the mouse was created in code through a class called MemoryEnv. The MemoryEnv class kept track of the locations of the breadcrumb and could be saved and reloaded for each run.

These three concepts provided the mouse with the ability to solve mazes.

## 4. OBJECTS IN-DEPTH

The main objects in the mouse maze program are: BreadCrumb, Cheese, MazeEnv, MemoryEnv, Mouse, and Wall. These objects will be examined in further detail.

### **A. BreadCrumb, Cheese, and Wall**

All three of these classes inherit from the MBCS Fish class. The MBCS was intended for Fish objects in an Environment object. The core files of the MBCS could be theoretically modified to implement these three new objects. However, one objective of the programming solution was to keep from modifying any of the core MBCS classes. In addition, BreadCrumb, Cheese, and Wall are all immovable objects. Therefore, they can just be represented with Fish objects that have the move functionality disabled. All three classes extend Fish and override the act() method of the fish class to make it a stationary object. Also, each class has a hard-coded id (in contrast to the dynamically generated id in the Fish class). The static id would be used to identify the particular object by other classes. Internally, the Cheese and Wall class are almost exactly identical. They do not have any functionality except for mapping a spot in the maze with an id. By checking for the Wall id, the Mouse object can avoid moving there. Furthermore, by checking for the Cheese id, the Mouse object can determine when it has found the solution.

BreadCrumb, however, has a major difference from the other two classes. BreadCrumb includes an attribute called timesMovedHere which keeps track of how many times the Mouse has visited that location. This attribute is extremely important in ensuring that the Mouse does not continually visit a location over and over again while allowing the Mouse a certain degree of freedom to backtrack when necessary.

### **B. MazeEnv**

The MazeEnv object inherits from BoundedEnv and encapsulates the grid in which the Mouse object moves. The MazeEnv adds to the BoundedEnv by providing automatic random Wall generation and random Mouse and Cheese placements. By using a random seed each time MazeEnv is constructed, a new maze is ready for the Mouse to solve. However, the seed can also be set constant for a consistent maze.

### **C. MemoryEnv**

The MemoryEnv object also inherits from BoundedEnv. The MemoryEnv is constructed by the Mouse object and has the same dimensions as the MemoryEnv. MemoryEnv has one unique method called registerSpot(Locatable inObject). This method registers the current position of the Mouse with an Object. This Object can be Wall, Cheese, or a BreadCrumb. All registered Objects

are saved on file and then reloaded into MemoryEnv during the next Mouse run.

### **D. Mouse**

The Mouse object inherits from the Fish class so that it can be easily moved in the MazeEnv through existing MBCS classes. However, the Mouse object also contains a great deal of modification to the Fish move code.

When the Mouse object is constructed, method init() is called which loads an existing MemoryEnv if possible or creates a new MemoryEnv. During each simulation step, the act() method in Mouse is called. In act(), the Mouse checks its surroundings for the Cheese and if the Cheese is not found, the Mouse calls move() to traverse the maze in search for Cheese. move() checks the surroundings of the Mouse and registers any Objects it encounters with MemoryEnv. Then, emptyNeighbors() is called which calculates the next viable move position. emptyNeighbors() also checks for BreadCrumbs and adjusts the move based on the results.

The BreadCrumb check includes a variable called threshold which keeps track of the amount of times the Mouse had no possible moves without moving over an existing BreadCrumb. By keeping track of a threshold, the Mouse is encouraged to move to BreadCrumbs with a timesMovedHere count lower than the threshold. This allows the Mouse to retrace its steps when trapped.

## **5. OBSTACLES**

Using the MBCS was not as easy as originally thought. Although the MBCS was designed with flexibility in mind, there are many classes in the MBCS that are “blackboxed,” that is, no source code was provided because the developers did not anticipate other programmers using them.

When approaching the task of saving and loading MemoryEnv, this blackboxing led to the initial idea of serializing the MemoryEnv object and saving the serialized data to a file which could then be reloaded an unserialized. In theory, this approach sounded very good. The state of MemoryEnv would be saved and all of its registered objects could also be saved. In practice, the MemoryEnv and all of the classes it inherited from or registered had to be serializable—even the core MBCS class files. Additionally, although the classes were finally serialized and saved, loading of the serialized data resulted in numerous exceptions which rendered the idea unfeasible.

The next approach was to reverse engineer the black box classes of the MBCS to learn about its hidden classes and methods. The black box classes were housed in mbsbb.jar and mbsgui.jar. The jar

files were easy to extract, but the files were all compiled bytecode (class files). Within mbsbb.jar, a likely candidate, MBSDataFileHandler, was found that could possibly aid in handling Environment objects.

Bytecode can be reversed and a utility called Jad was used to decompile the files in mbsbb.jar and mbsgui.jar. Now having the source code to the black box classes, the methods of MBSDataFileHandler were revealed and were used in the Mouse class to save and load the MemoryEnv and its registered Objects.

However, the flaw with using MBSDataFileHandler was that individual Object states were not saved, only the Object's name and location in the Environment were saved. However, the workaround to this problem would be identifying the location of the Cheese in the MemoryEnv and directing the Mouse to that location without relying on the BreadCrumb objects too much since the solution location is already known.

#### **4. VALIDATION**

Initial testing by observing Mouse movement indicated that the Mouse would find a solution 100% of the time if the solution was solvable (i.e. That the Cheese was not surrounded completely by walls). However, upon further examination with larger grid sizes, the Mouse exhibited flaws in BreadCrumb checking as it fell into an infinite loop traversing the outer limits of the maze. The BreadCrumb checking code in Mouse should be reanalyzed.

Also, the MemoryEnv is still weak in its design. It currently does not provide a greater improvement than regular searching. While humans can look at a map and easily discern a direct route from their location to the solution, a program is limited in its linear processing so that it cannot "see" the whole grid at once. Further work must be done on MemoryEnv to have a more efficient memorization and "smart" moving system.

#### **5. SUMMARY**

This model of a mouse moving through a maze sought to improve on traditional models of mouse mazes by introducing methods in which the mouse makes decisions like a human. Approaches such as consistent rules in movement, location marking, and location mapping were used to aid the mouse in finding the solution and then remembering how it found the solution. Although the final solution has a few programming flaws, the concept can be

expanded upon to model a "smart" mouse capable of solving any maze.

#### **REFERENCES**

1. "Marine Biology Case Study." The College Board. 9 June 2004. < [http://www.collegeboard.com/student/testing/ap/compsi\\_ab/case.html](http://www.collegeboard.com/student/testing/ap/compsi_ab/case.html) >.
2. Pavel Kouznetsov. "Jad – The fast JAVa Decompiler." Pavel Kouznetsov. 9 June 2004. < <http://kpdus.tripod.com/jad.html> >.