FSU Computer Science Department

Design and Implementation of a Goal-and Knowledge-Based Agent Traversing Through Wumpus World



COSC 455 Section 001

Artificial Intelligence

December 16, 2013

By John Coleman and Travis O’Donnell

Abstract:

As a group, we set out to achieve certain goals. Our goals dealt with making a goal and knowledge-based agent capable of traversing through simulated versions of Wumpus World, finding a goal, all while avoiding all obstacles. Implying previous and newfound methods, our group was able to make an agent capable of such a task. Our agent performed as we intended it to during the final stages of testing by finding its goal. If given the opportunity to attempt this project again, we would make changes to our used colors and how the color sensor responds.

Table of Contents

[Table of Figures 3](#_Toc375002287)

[Introduction 4](#_Toc375002288)

[Background 4](#_Toc375002289)

[Approach 5](#_Toc375002290)

[Results 8](#_Toc375002291)

[Conclusion 10](#_Toc375002292)

[References 11](#_Toc375002293)

[Appendix 12](#_Toc375002294)

## Table of Figures

[Figure 1: Colors and Their Representations 6](#_Toc375002295)

[Figure 2: Agent Without Middle Motor (rear) 7](#_Toc375002296)

[Figure 3: Agent Without Middle Motor (side) 7](#_Toc375002297)

[Figure 4: Eight Possible Testing Environments 8](#_Toc375002298)

[Figure 5: Agent In Maze Before Test 9](#_Toc375002299)

[Figure 6: Agent After Finding Gold 10](#_Toc375002300)

## Introduction

For a final project, our group decided to create an agent capable of traversing through a maze in search of a goal. This maze would represent a fictional world entitled "Wumpus World." The goal our agent would attempt to reach would be a room within the world, which contains "gold." To do this, our group needed to design an agent capable to traversing through such a maze, as well as be able to get a constant reading of its environment. On top of that, our agent needed to compare its environmental readings with items stored within its knowledge base.

Based upon the knowledge base programmed into memory, our agent needed to be capable of making decisions quickly, and without error. One slight error and our agent could potentially fail the test entirely. Based upon these circumstances, our group decided that this scenario would make an appropriate final project. This project would test our previous knowledge working with our agent, force us to combine parts of past objects together, as well as challenge us to implement ideas we had never used before.

## Background

"The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold [1]."

To traverse such an environment, our group needed an agent that was goal-based, as well as one that is knowledge-based. To start, a goal-based agent is one that's movements are based upon its search for a goal. On top of this, our agent needed to be a knowledge-based agent. "Knowledge-based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge [2]. With the combination of these two types of agents in mind, our group need to construct such an agent.

To build such an agent, our group decided to use a L EGO® Mindstorm kit. By definition, LEGO® MINDSTORMS® is a programmable robotics construction series that gives you the power to build and program your own LEGO robots [3]. To program these robots, we used a programming language called leJOS, similar to the Java programming language. LeJOS offers the following:

* Object oriented language (Java)
* Preemptive threads (tasks)
* Arrays, including multi-dimensional
* Recursion
* Synchronization
* Exceptions
* Java types including float, long, and String
* Most of the java.lang, java.util and java.io classes
* A Well-documented Robotics API [4]

Due to our group’s past experience with Java, and given that leJOS is so similar to Java, using it was an obvious choice.

While still in the planning phase of this project, our group decided to implement the PEAS method to help us design our agent. The PEAS method (being Performance, Environment, Actuators, Sensors) [5] helped our group simplify this project, and helped us get a better grasp on what we wanted to achieve, as well as how we would achieve our goals. In terms of performance, our agent needed to be capable of traversing wumpus world, as well as stay within the boundaries of the world, and continue to navigate until a goal is found. Our environment would be pre-determined by our group, but would appear like a completely new maze to our agent each time we tested. This environment would be constructed from a large sheet of white paper and formulated into a grid-like format. Within this grid would be different colored rooms, with a different color to represent each scenario. For Actuators, our group needed an agent capable of accelerating, braking, and reversing. In terms of Sensors, our agent needed to interact with its environment, as well as understand the differences between the different colors in the environment.

## Approach

In order to test our agent, our group needed to construct a testing environment. Within the environment, our group needed the option of interchanging the colors within the rooms. By interchanging the colors within the rooms, our groups could test out numerous situations within just one testing environment. To do this, our group first constructed a rather large white sheet. From this white sheet, our group then divided the sheet into a 4 x 4 square grid, with each room of the grid being 12 inches by 12 inches. After our grid was constructed, our group made use of 12 inch by 12 inch colored paper to overlap the necessary room. For colors, our group used white, black, blue, red, green, yellow, orange, pink, purple. Each color corresponded to a different situation that our agent could encounter from within Wumpus World. Within the world, our agent could encounter situations such as a pit, the wumpus, or gold. Also, to indicate these situations are near, in the rooms adjacent to pits, the wumpus, and gold, there are also separate indicators. In the rooms surrounding a pit, a breeze is encountered. In the rooms surrounding the wumpus, there is a stench given off by the wumpus. Also, there can exist combinations of these indicators, such as a room containing a breeze, a stench, and gold. To differentiate between all of these possible scenarios, our group assigned the following colors to each scenario:

|  |  |
| --- | --- |
| **Color** | **Representation** |
| White | An empty room with no indicators |
| Black | A pit |
| Yellow | Gold |
| Purple | The Wumpus |
| Pink | Stench |
| Blue | Breeze |
| Green | Combination of Breeze and Stench |
| Red | Combination of Breeze and Gold |
| Orange | Combination of Stench and Gold |

Figure : Colors and Their Representations

Our group added these colors into our agent's knowledge base. By doing this, our agent could make evaluations about its environment based upon what color each room is. Besides this, the only other thing in our agent's knowledge base was that the whole world was a 4 foot by 4 foot grid.

Upon starting construction of our agent, our group decided to make improvements upon our previous builds. Our previous build, designed after the model in the manual that came with our Mindstorm kit, was too bulky for what we wished to achieve in this project. To simplify the agent, our group removed the middle motor (see Figures 2 and 3).



Figure : Agent Without Middle Motor (rear)

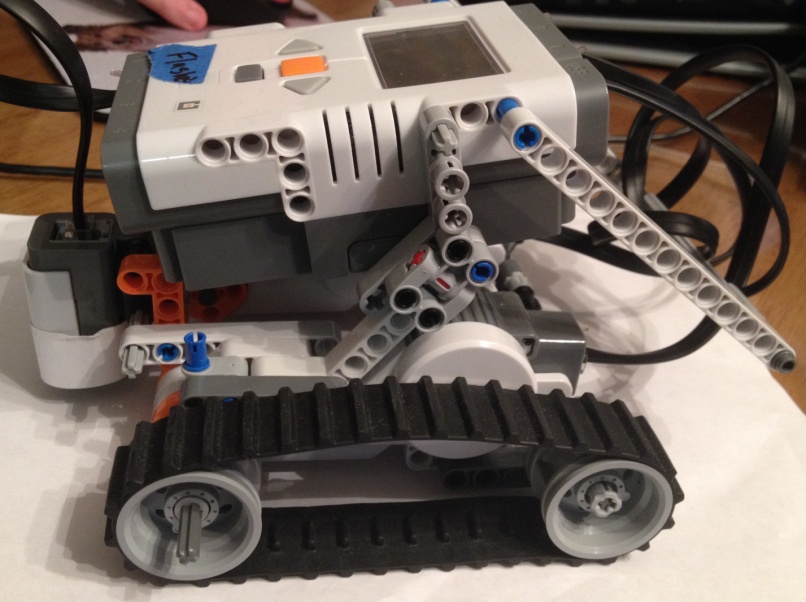


Figure : Agent Without Middle Motor (side)

The middle motor was never attached to anything, and our group deemed this as an unnecessary waste of size. Also, because we had removed the middle motor, our group could move our color sensor closer to our agent. With the color sensor moved closer to the body of our agent, the agent proved to be far more compact that it had been in any of our previous builds.

To test out possible testing environments, our group decided that, before we actually testing our agent, we should test out the logic of our world. To do this, our group wrote a program that would simulate a version of Wumpus World. In this program, all scenarios are generated at random. By making the scenarios generate at random, our group could detect errors within our logic much easier, all while giving us ideas of possible testing environments to put our agent through. Our group compiled eight such environments to test (see Figure 1).

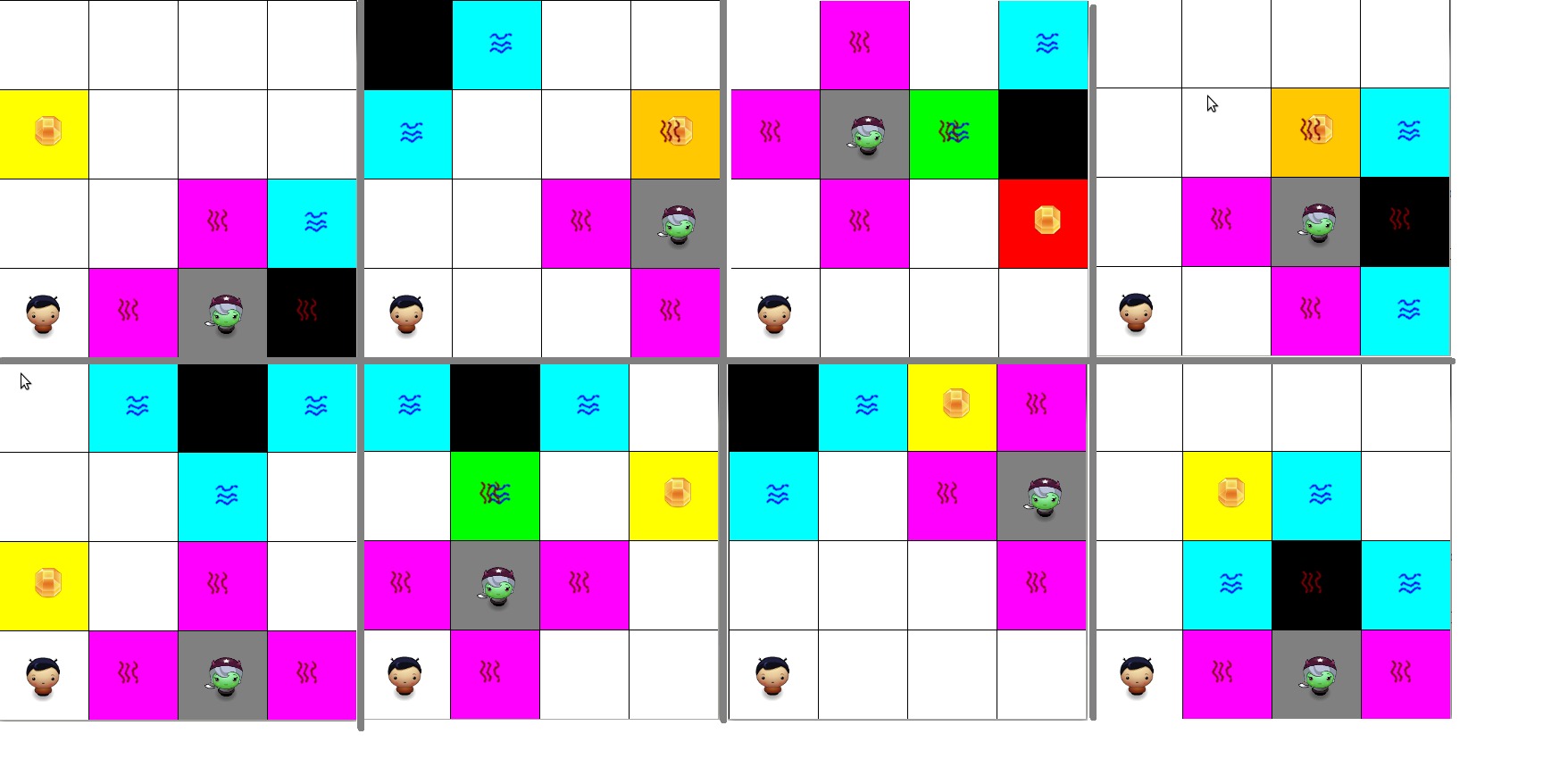


Figure : Eight Possible Testing Environments

After compiling these possible environments, our group was ready to move onto the next stage of our project, actually testing our agent's functionality.

## Results

When it came time to test, our group decided upon implementing the eight testing environments show in Figure4. To do this, we used our colored paper and placed them in the corresponding rooms. For example, our group tested the environment shown in Figure 5



Figure : Agent In Maze Before Test

The agent started in the normal starting room and turned right. The agent detected that it had encountered a stench and returned to the start room. The agent then turned right and detected another stench. At this point, the agent knew that the wumpus was in the room to the right of it. So, as it was designed, the agent turned right, shot the wumpus, confirmed that it killed the wumpus, turned left, and continued on its course. After reaching the next safe room, the agent turned right and found a green room. This indicated a breeze and a stench. As a result, the agent backed up, returned to start, and continued around the other edge of the maze. Eventually, the agent found its goal (see Figure 6).



Figure : Agent After Finding Gold

After testing all of our eight possible environments, our group was satisfied with our agent's results and concluded our testing.

## Conclusion

After evaluating our testing results, our group is very satisfied with what our agent's performance. In the beginning, our group knew that this project would not be easy. But, after numerous amounts of trial and error, our group was able to achieve our goal. This goal was to created an agent capable of finding its goal (the gold) while avoiding all obstacles (pits and the wumpus). Upon testing all of our pre-determined environments, our agent was able to achieve such a feat, and also managed to "kill" the Wumpus whenever it was necessary.

If we were to do this project over, although we are satisfied with our results, our group would change a few things. One of the key things we would change would involve our color sensor. Some testing environments would prove impossible for our current build. For example, our color sensor was never truly able to distinguish between pink and red. As a result, if a red and pink room were right next to each other, our agent would not perform as it is intended to.

## References

[1] S. Russell and P. Norvig, “Chapter 7 – Logical Agents,” in Artificial Intelligence: A Modern Approach, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010, ch. 7, sec.7.2, pp. 236-237.

[2] S. Russell and P. Norvig, “Chapter 7 – Logical Agents,” in Artificial Intelligence: A Modern Approach, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010, ch. 7, pp. 235.

[3] LEGO®. (2013, October 18. What is LEGO® MINDSTORMS®? (1st ed.) [Online]. Available:http://service.lego.com/en-us/helptopics/products/themes/mindstorms/general-questions/what-is-mindstorms

[4] J. Moral. (2013, October 18. What is leJOS? (1st ed.) [Online]. Available: http://www.lejos.org/nxj.php

[5] S. Russell and P. Norvig, “Chapter 2 – Intelligent Agents,” in Artificial Intelligence: A Modern Approach, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010, ch. 2, sec. 2.3.1, pp. 40

## Appendix

import java.io.IOException;

import java.util.ArrayList;

import java.util.Random;

import java.util.Stack;

import lejos.nxt.Button;

import lejos.nxt.ColorSensor;

import lejos.nxt.LCD;

import lejos.nxt.Motor;

import lejos.nxt.SensorPort;

import lejos.nxt.Sound;

import lejos.robotics.Color;

import lejos.robotics.RegulatedMotor;

import lejos.robotics.navigation.DifferentialPilot;

import lejos.robotics.navigation.MoveController;

import lejos.util.Delay;

/\*\*

\* Wumpus World

\* See lab report for description of Wumpus World

\* @author Travis ODonnell John Coleman

\*

\*/

public class WumpusWorld {

private static final RegulatedMotor RIGHT\_MOTOR = Motor.C;

private static final RegulatedMotor LEFT\_MOTOR = Motor.A;

private static final float TRACK\_WIDTH = 15.24f; // Needs to be measured

private static Thread tracker;

/\*\*

\* Byte values

\*

\* Visited - 00000001 Safe - 00000010 Breeze - 00000100 Stench - 00001000

\* Glitter - 00010000 Pit - 00100000 Wumpus - 01000000

\*/

private static final byte VISITED = 0x01, SAFE = 0x02, BREEZE = 0x04,

STENCH = 0x08, GLITTER = 0x10, PIT = 0x20, WUMPUS = 0x40;

private static CoordList stenchList, breezeList;

private static int location\_x, location\_y, wumpus\_x, wumpus\_y, pit\_x,

pit\_y, temp\_x, temp\_y;

private static int direction; // 0 - NORTH, 1 - EAST, 2 - SOUTH, 3 - WEST

private static int numArrows;

private static boolean isDead, hasGold, gameOver;

private static byte[][] world;

private static Random random;

private static final float FOOT\_IN\_CM = 42.1f;

private static DifferentialPilot pilot;

private static ColorTracker ct;

public static ColorSensor sensor;

/\*\*

\* Coordinate class just holds a R^(2) (x,y) value from the Coordinate plane

\* @author ODonnell

\*

\*/

static class Coordinate {

private int x, y;

public Coordinate(int x, int y) {

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

public Coordinate north() {

return new Coordinate(x, y + 1);

}

public Coordinate south() {

return new Coordinate(x, y - 1);

}

public Coordinate east() {

return new Coordinate(x + 1, y);

}

public Coordinate west() {

return new Coordinate(x - 1, y);

}

}

/\*\*

\* Holds an arraylist of type Coordinate (could have just used an arraylist though)

\* @author ODonnell

\*

\*/

static class CoordList {

private ArrayList<Coordinate> list;

private int size;

public CoordList() {

size = 0;

list = new ArrayList<Coordinate>();

}

public void addCoord(int x, int y) {

size++;

list.add(new Coordinate(x, y));

}

public int size() {

return size;

}

public Coordinate indexOf(int n) {

return list.get(n);

}

}

public static void main(String[] args) throws IOException,

InterruptedException {

initializeWorld();

initializeAgent();

setFlags();

while (!gameOver) {

LCD.drawString("" + ct.getCurrentWord(), 0, 1);

LCD.drawString("Direction " + direction, 0, 2);

LCD.drawString("X " + location\_x + " Y " + location\_y, 0, 3);

makeMove();

if (checkGold())

break;

}

LCD.clear();

if(checkGold())

LCD.drawString("Gold found!", 0, 0);

else

LCD.drawString("Game Over", 0, 0);

}

/\*\*

\* Initializes all used variables

\*/

private static void initializeWorld() {

world = new byte[4][4];

location\_x = 0;

location\_y = 3;

direction = 0;

numArrows = 1;

pit\_x = -1;

pit\_y = -1;

wumpus\_x = -1;

wumpus\_y = -1;

isDead = false;

hasGold = false;

gameOver = false;

random = new Random();

stenchList = new CoordList();

breezeList = new CoordList();

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++) {

world[i][j] = (byte) 0x00;

setFlag(SAFE, i, j);

}

}

/\*\*

\* Initializes the agent.

\* code below in this method is just setting up the DifferentialPilot class that came with LeJos

\* @throws IOException

\* @throws InterruptedException

\*/

private static void initializeAgent() throws IOException,

InterruptedException {

pilot = new DifferentialPilot(MoveController.WHEEL\_SIZE\_NXT2,

TRACK\_WIDTH, LEFT\_MOTOR, RIGHT\_MOTOR, false);

sensor = new ColorSensor(SensorPort.getInstance(0));

pilot.setTravelSpeed(25);

pilot.setRotateSpeed(60);

ct = new ColorTracker(sensor);

tracker = new Thread(ct);

tracker.start();

sensor.setFloodlight(Color.WHITE);

}

/\*\*

\* returns boolean based upon wether bit value is set or not at X, Y

\* @param VALUE

\* @return

\*/

private static boolean checkFlag(byte VALUE) {

return ((world[location\_x][location\_y] & VALUE) == VALUE);

}

/\*\*

\* deletes the specified bit value from the world at X, Y

\* @param VALUE

\* @param x

\* @param y

\*/

private static void delFlag(byte VALUE, int x, int y) {

world[x][y] = (byte) (world[x][y] & ~VALUE);

}

/\*\*

\* returns boolean based upon wether bit value is set or not at X, Y

\* @param VALUE

\* @return

\*/

private static boolean checkFlag(byte VALUE, int x, int y) {

return ((world[x][y] & VALUE) == VALUE);

}

/\*\*

\* Checks current color at position X, Y and sets corresponding bits

\* Black - Pit

\* BLUE - Breeze

\* Magenta - Wumpus

\* Yellow - Gold

\* Green - Pit and Stench

\* Orange - Gold and Stench

\* Red - stench

\*/

private static void setFlags() {

Delay.msDelay(150);

switch (ct.getCurrentColor()) {

case Color.BLACK:

setFlag(PIT);

break;

case Color.BLUE:

setFlag(BREEZE);

break;

case Color.MAGENTA:

setFlag(WUMPUS);

break;

case Color.GREEN:

setFlag(STENCH);

setFlag(PIT);

break;

case Color.YELLOW:

setFlag(GLITTER);

setGold(true);

break;

case Color.ORANGE:

setFlag(GLITTER);

setFlag(BREEZE);

setGold(true);

break;

case Color.RED:

setFlag(STENCH);

break;

case Color.WHITE:

break;

default:

break;

}

}

/\*\*

\* Sets specified bit at current X, Y

\* @param f

\*/

private static void setFlag(Byte f) {

world[location\_x][location\_y] |= f;

}

/\*\*

\* Sets specified bit at current X, Y

\* @param f

\*/

private static void setFlag(Byte f, int x, int y) {

world[x][y] |= f;

}

/\*\*

\* Returns boolean value based upon whether pit has been set or not

\* @return

\*/

private static boolean pitSet() {

if (pit\_x != -1)

return true;

return false;

}

/\*\*

\* Sets pit at X, Y position in the world

\* Deletes byte values for breeze, visited execpt for where pit is

\* Sets Safe value for all positions except for where pit is

\* @param x

\* @param y

\*/

private static void setPit(int x, int y) {

pit\_x = x;

pit\_y = y;

world[pit\_x][pit\_y] |= PIT;

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 4; j++)

if (i != pit\_x && j != pit\_y) {

delFlag(BREEZE, i, j);

setFlag(SAFE, i, j);

delFlag(VISITED, i, j);

}

}

}

/\*\*

\* Set true is gold has been found

\* @param b

\*/

private static void setGold(boolean b) {

gameOver = true;

hasGold = b;

}

/\*\*

\* Return boolean value on whether wumpus is dead

\* @return

\*/

private static boolean checkDead() {

return isDead;

}

/\*\*

\* Returns boolean value on whether gold has been found or not

\* @return

\*/

private static boolean checkGold() {

return hasGold;

}

/\*\*

\* Shoots 'arrow' at wumpus. Only done when 2 known stench values are (See checkStench())

\* @return

\*/

private static boolean shootArrow() {

LCD.drawString("SHOOOOOOOT", 0, 0);

if (numArrows > 0) {

numArrows--;

Sound.playTone(2000, 1250);

}

return false;

}

/\*\*

\* Checks to see if X and Y are within the array bounds

\* @param x

\* @param y

\* @return

\*/

private static boolean checkValid(int x, int y) {

if (x >= 0 && x < 4 && y >= 0 && y < 4)

return true;

return false;

}

/\*\*

\* Rotates the agent based upon current direction to the new specified direction

\* @param d

\*/

private static void changeDirection(int d) {

LCD.drawString("Direction", 0, 0);

switch (direction) {

case 0:

if (d == 1)

pilot.rotate(142);

if (d == 2)

pilot.rotate(142 \* 2);

if (d == 3)

pilot.rotate(-142);

break;

case 1:

if (d == 0)

pilot.rotate(-142);

if (d == 2)

pilot.rotate(142);

if (d == 3)

pilot.rotate(142 \* 2);

break;

case 2:

if (d == 0)

pilot.rotate(142 \* 2);

if (d == 1)

pilot.rotate(-142);

if (d == 3)

pilot.rotate(142);

break;

case 3:

if (d == 0)

pilot.rotate(142);

if (d == 1)

pilot.rotate(142 \* 2);

if (d == 2)

pilot.rotate(-142);

break;

default:

break;

}

direction = d;

}

/\*\*

\* Traverses to next position if position is safe to venture to, else try a new random direction

\* @return

\*/

private static boolean traverse() {

LCD.drawString("Traverse", 0, 0);

switch (direction) {

case 0:

if (location\_y + 1 < 4) {

if (checkFlag(SAFE, location\_x, location\_y + 1)

&& location\_x != wumpus\_x && location\_y + 1 != wumpus\_y) {

pilot.travel(FOOT\_IN\_CM);

location\_y++;

} else {

changeDirection((int) (Math.random() \* 4));

traverse();

}

return true;

}

return false;

case 2:

if (location\_y - 1 >= 0) {

if (checkFlag(SAFE, location\_x, location\_y - 1)

&& location\_x != wumpus\_x && location\_y - 1 != wumpus\_y) {

pilot.travel(FOOT\_IN\_CM);

location\_y--;

} else {

LCD.drawString("TURN", 0, 0);

changeDirection((int) (Math.random() \* 4));

traverse();

}

}

return false;

case 1:

if (location\_x + 1 < 4) {

if (checkFlag(SAFE, location\_x + 1, location\_y)

&& location\_x + 1 != wumpus\_x && location\_y != wumpus\_y) {

pilot.travel(FOOT\_IN\_CM);

location\_x++;

} else {

LCD.drawString("TURN", 0, 0);

changeDirection(((int) Math.random() \* 4));

traverse();

}

return true;

}

return false;

case 3:

if (location\_x - 1 >= 0) {

if (checkFlag(SAFE, location\_x - 1, location\_y)

&& location\_x - 1 != wumpus\_x && location\_y != wumpus\_y) {

pilot.travel(FOOT\_IN\_CM);

location\_x--;

} else {

LCD.drawString("TURN", 0, 0);

changeDirection(((int) Math.random() \* 4));

traverse();

}

return true;

}

return false;

default:

return false;

}

}

/\*\*

\* Checks to make sure the agent didn't make a mistake and end up in

\* dangerous room (wumpus, pit)

\*

\* @return true if game over

\*/

private static boolean checkForMistake() {

if (checkFlag(WUMPUS) || checkFlag(PIT)) {

gameOver = true;

return true;

}

return false;

}

/\*\*

\* Changes current direction to opposite using Mod operator

\*/

private static void getOppositeDirection() {

changeDirection((direction + 2) % 4);

}

/\*\*

\* Depending on which current direction of agent, sets unsafe byte values for

\* position ahead, right, left of agent

\*/

private static void setUnsafeFlags() {

switch (direction) {

case 0:

if (checkValid(location\_x, location\_y + 1))

delFlag(SAFE, location\_x, location\_y + 1);

if (checkValid(location\_x + 1, location\_y))

delFlag(SAFE, location\_x + 1, location\_y);

if (checkValid(location\_x - 1, location\_y))

delFlag(SAFE, location\_x - 1, location\_y);

break;

case 1:

if (checkValid(location\_x, location\_y + 1))

delFlag(SAFE, location\_x, location\_y + 1);

if (checkValid(location\_x + 1, location\_y))

delFlag(SAFE, location\_x + 1, location\_y);

if (checkValid(location\_x, location\_y - 1))

delFlag(SAFE, location\_x, location\_y - 1);

break;

case 2:

if (checkValid(location\_x, location\_y - 1))

delFlag(SAFE, location\_x, location\_y - 1);

if (checkValid(location\_x + 1, location\_y))

delFlag(SAFE, location\_x + 1, location\_y);

if (checkValid(location\_x - 1, location\_y))

delFlag(SAFE, location\_x - 1, location\_y);

break;

case 3:

if (checkValid(location\_x, location\_y + 1))

delFlag(SAFE, location\_x, location\_y + 1);

if (checkValid(location\_x, location\_y - 1))

delFlag(SAFE, location\_x, location\_y - 1);

if (checkValid(location\_x - 1, location\_y))

delFlag(SAFE, location\_x - 1, location\_y);

break;

}

}

/\*\*

\* Checks stench byte value at current location

\* If we already have stench recorded at current position, do nothing

\* Else Add position to StenchList, if it is first element in StenchList, rotate backwards and keep traversing

\* If it is the second stench value recorded, pinpoints exact location of wumpus, rotates agent towards wumpus and shoots arrow

\* @return

\*/

private static boolean checkStench() {

if (checkFlag(STENCH)) {

boolean checked = false;

for (int i = 0; i < stenchList.size(); i++) {

if ((location\_x == stenchList.indexOf(i).getX())

&& (location\_y == stenchList.indexOf(i).getY()))

checked = true;

}

if (checkFlag(STENCH) && !checked) {

stenchList.addCoord(location\_x, location\_y);

switch (stenchList.size()) {

case 1:

setUnsafeFlags();

getOppositeDirection();

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++)

delFlag(VISITED, i, j);

break;

case 2:

Coordinate t = stenchList.indexOf(0);

int d = calculateDirection(t);

changeDirection(d);

shootArrow();

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 4; j++)

if (i != pit\_x && j != pit\_y) {

setFlag(SAFE, i, j);

delFlag(VISITED, i, j);

delFlag(WUMPUS, i, j);

delFlag(STENCH, i, j);

}

}

break;

default: // We already found wumpus, so we don't need to do

// anything

// for 3 and 4

break;

}

return true;

}

return false;

}

return false;

}

/\*\*

\* Uses for shooting arrow

\* Based upon current direction, location and wumpus location returns direction it needs to look to correctly fire arrow

\* @param t

\* @return

\*/

private static int calculateDirection(Coordinate t) {

if (t.getX() == location\_x)

if (t.getY() > location\_y)

return 3;

else

return 1;

else if (t.getY() == location\_y)

if (t.getX() > location\_x)

return 2;

else

return 0;

else if ((t.getX() + 1 == location\_x))

return 3;

else if (t.getX() - 1 == location\_x)

return 1;

else if (t.getY() + 1 == location\_y)

return 0;

else if (t.getY() - 1 == location\_y)

return 2;

return -1;

}

/\*\*

\* Checks if agent has glitter

\* @return

\*/

private static boolean checkGlitter() {

if (checkFlag(GLITTER)) {

setGold(true); // Picked up gold

return true;

}

return false;

}

/\*\*

\* Checks breeze byte value at current location

\* If we already have breeze recorded at current position, do nothing

\* Else Add position to breezeList, if it is first element in breezeList, rotate backwards and keep traversing

\* If it is the second stench value recorded, pinpoints exact location of pit, sets unsafe and sets pit

\* @return

\*/

private static boolean checkBreeze() {

if (checkFlag(BREEZE)) {

boolean checked = false;

for (int i = 0; i < breezeList.size(); i++) {

if ((location\_x == breezeList.indexOf(i).getX())

&& (location\_y == breezeList.indexOf(i).getY()))

checked = true;

}

if (checkFlag(BREEZE) && !checked) {

breezeList.addCoord(location\_x, location\_y);

switch (breezeList.size()) {

case 1:

setUnsafeFlags();

getOppositeDirection();

break;

case 2:

Coordinate t = breezeList.indexOf(0);

pit\_x = (int) ((location\_x + t.getX()) / 2 + 1);

pit\_y = (int) ((location\_y + t.getY()) / 2 + 1);

setPit(pit\_x, pit\_y);

break;

default: // We already found pit, so we don't need to do

// anything

// for 3 and 4

break;

}

return true;

}

return false;

}

return false;

}

/\*\*

\* Adds neighbor coordinate to a coordlist for traverseHelper function

\* @return

\*/

private static CoordList neighborHelper() {

CoordList neighbors = new CoordList();

neighbors.addCoord(location\_x, location\_y + 1);

neighbors.addCoord(location\_x + 1, location\_y);

neighbors.addCoord(location\_x, location\_y - 1);

neighbors.addCoord(location\_x - 1, location\_y);

return neighbors;

}

/\*\*

\* Goes through neighbor coordinates, traverses based upon already visited or safe

\*/

private static void traverseHelper() {

LCD.drawString("TraverseHelper", 0, 5);

CoordList neighbors = neighborHelper();

int x, y;

for (int i = 0; i < 4; i++) {

x = neighbors.indexOf(i).getX();

y = neighbors.indexOf(i).getY();

if (x < 4 && y < 4 && x >= 0 && y >= 0) {

if (checkFlag(SAFE, x, y) && !checkFlag(VISITED, x, y)) {

changeDirection(i);

traverse();

return;

}

}

}

for (int i = 0; i < 4; i++)

for (int j = 0; j < 4; j++)

delFlag(VISITED, i, j);

}

/\*\*

\* Checks if agent made a mistake

\* If not, setsFlags for current position

\* Checks whether stench, gold or breeze is at current position

\* Goes to traverseHelper

\* @return

\* @throws InterruptedException

\*/

public static boolean makeMove() throws InterruptedException {

Delay.msDelay(1000);

if (!checkForMistake() || gameOver) {

setFlags();

setFlag(VISITED);

if (!checkDead())

checkStench();

if (!pitSet())

checkBreeze();

if (checkFlag(PIT)) {

gameOver = true;

return true;

}

if (checkGlitter()) {

setGold(true);

Sound.beepSequenceUp();

return true;

}

traverseHelper();

}

return false;

}

/\*

\* Ideas for this class came from ColorSensorTest.java, which is found

\* within the leJos sample code

\*/

static class ColorTracker implements Runnable {

public int currentColor; // Need to let Main.java see this value

String colorNames[] = { "None", "Red", "Green", "Blue", "Yellow",

"Megenta", "Orange", "White", "Black", "Pink", "Grey",

"Light Grey", "Dark Grey", "Cyan" };

private ColorSensor sensor;

private String currentColorName;

public ColorTracker(ColorSensor s) {

this.sensor = s;

}

public void run() {

while (true || Button.ESCAPE.isDown()) {

ColorSensor.Color value = sensor.getColor();

currentColor = value.getColor();

}

}

public String getCurrentWord() {

return colorNames[currentColor + 1];

}

public int getCurrentColor() {

return currentColor;

}

}

}