Collaborators: Zhiling Hu and Sabrina Accime
Professor Lisa Ballesteros
Artificial Intelligence
Final Lab: Robot Maze-Navigation
May 8, 2017

<u>Design Document</u>

Section 1: Project Description

For our project, we have developed an autonomous navigating Lejos NXT bot, called *Goblin*. When given a maze navigation problem, *Goblin*, who lacks information about its environment other than start location and goal location, must gathered information while navigating through a maze from the starting point to the goal/destination and then make its way directly back to the start without making a wrong turn. Once *Goblin* has reached its destination, it acknowledges this by playing a simple song.

The implementation of our project is heavily influenced by the robotic architecture called Subsumption architecture. Subsumption architecture is a form of robot control in which the control is divided into layers corresponding to levels of behavior. The idea of subsumption is that not only do more complex layers depend on lower, more reactive levels, but that they could also influence their behavior. Within subsumption architecture, the controlling structure is an arbitrator. The arbitrator looks through a list of behaviors, and depending on the current conditions, will fire off a certain behavior.

Behavior generally refers to the actions or reactions of an object or organism, usually in relation to the environment or surrounding world of stimuli. In our case, we are going to want to have behaviors that fire depending on information that we receive in from our sensors. Therefore, we can break down our behaviors into two parts, the conditions that determine whether or not to fire a certain behavior, and the actions to take if those conditions are satisfied.

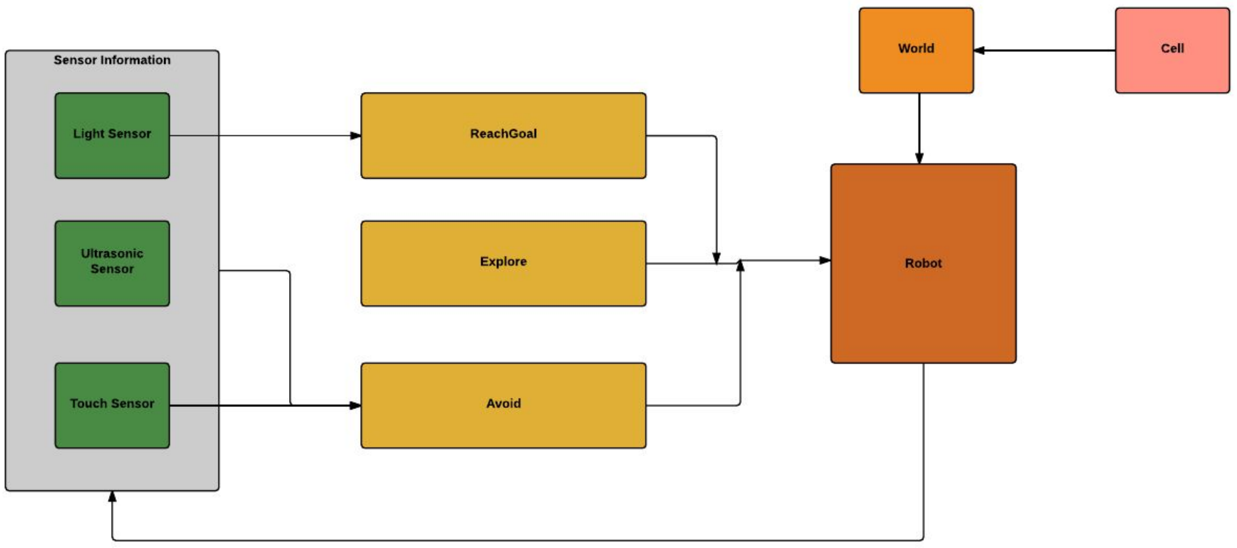
Section 2: Search Algorithm

Our search algorithm lived in one of our behavior classes called **Explore**. We decided to use Depth First Search as the algorithm that assisted the *Goblin* in navigating its environment. Depth First Search always expands the *deepest* node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.

```
DFS(G,v)   ( v is the vertex where the search starts )
    Stack S := {};   ( start with an empty stack )
    for each vertex u, set visited[u] := false;
    push S, v;
    while (S is not empty) do
      u := pop S;
      if (not visited[u]) then
        visited[u] := true;
        for each unvisited neighbour w of u
          push S, w;
      end if
    end while
  END DFS()
```

The advantages of this algorithm it can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node. The disadvantages of using this algorithm is that it may settle for non-optimal solution and that it may explore a single path for a long time. The running time of this algorithm is O(b^m) where b is the branching factor and m is the maximum depth of the any cell.

Section 3: High Level Design Overview



Section 4 Low Level Design Overview

Robot

It holds the main. It will initialize all the sensors being used. It creates an Arbitrator object that regulates when each of the behaviors will become active. It takes an array of behaviors as a parameter. The order of the behaviors in the array indicates the priority level of the behaviors. The default behavior being that with the lowest index, which is Explore.

Instances it holds:

- Sensors
  - TouchSensor
  - LightSensor
  - Ultrasonic Sensor
- Behaviors
  - Explore
  - Avoid
  - ReachGoal
- Information for the maze
  - Start: (1, 1)
  - Goal: (5, 8)
  - Size: 7 * 10

Avoid

Avoid implements Behavior interface. It takes Touch Sensor and Ultrasonic Sensor. It will be called on once an obstacle is detected. It will update the World by setting the cell value -1. It has three built in methods for Behavior class.

- takeControl()
    - If Touch Sensor is pressed or Ultrasonic Sensor detects an obstacle one cell ahead
- action()
    - Flag down
    - Update the current cell value to be -1 in the **World**
    - Update the World by marking the cell with the obstacle as visited
    - Travel backwards to previous cell
- suppress()
    - Set the flag back

Explore

Explore implements Behavior interface. It's the main behavior for *Goblin*. It will walk through the maze, mark the cell in the **World** if an obstacle is detected by Avoid, and walk the way back if reach goal, which is flagged by **ReachGoal**.

The constructor takes Differential Pilot robot and **World** as parameters. It navigates from the starting cell (1,1) to the goal cel (5,8). It defines and initiates the action to perform in different cases when walking through the maze. It holds two ArrayList types, one is called toCheck, working as a stack to store all cells to be explored, the other one is called path.

- If Goal Reached
    - Call on World to create a reverse path
    - Walk the way back
- Else
    - Decides on the state of current cell
        - If it is an obstacle (flagged by **Avoid)**
            - Backtracking to the previous cell
    - Explores adjacent cells

        - Call checkAround method to check all adjacent cells of current cell
    - Picks next cell
        - Pops up one from toCheck
        - Chooses orientation for the robot to turn depends on the relationship between previous cell, current cell, and next cell (will be shown in the example below)
    - Updates Cells for the next round
        - Previous = current
        - Current = next

<u>Example for picking orientation for robot</u>

P - previous;
C - current;
T - temp (next);

P	C	T

If P and C are in the same row, C and T are in the same row:

```
robot.travel(cellDistance);
```

	T	
P	C	

If P and C are in the same row, T is in the upper right corner in the perspective of P:

```
robot.rotateLeft();
robot.travel(cellDistance);
```

	C	T
	P	

If P and C are in the same col, T is in the upper right corner in the perspective of P:

```
    robot.rotateRight();
    robot.travel(cellDistance);
```

ReachGoal

ReachGoal implements Behavior interface. It takes Light Sensor. It will be called on once it gets light reflected from the white cell. It has three built in methods for Behavior class. Plus, it has a method for walking back to the starting cell, and a method for playing a song.

- takeControl()
    - If Light Sensor detects a white cell
- action()
    - Play the song
    - Call on **World** to generate a path back to starting cell
    - Walk back
- suppress()
    - Stop the robot
- play()
    - Plays the tone stored in the note array
- walkBack()
    - Walk back to the starting cell

Cell

The cell holds information for coordinates regarding its location in the maze. It will also be assigned value that indicates the count of emerging paths from it.
It contains many helper methods.

- setValue
    - set the cell value to a specific integer
- setVisited
    - flag the boolean to mark the cell visited
- setObstacle
    - set the cell value to be -1
- setDeadEnd
    - set the cell value to be 1
- removeAPath
    - cut off the cell value by 1
- isObstacle
    - returns true if the cell value is -1
- isDeadEnd
    - returns true if the cell value is 1
- printPos
    - print position s

World

The World holds the grid and knowledge, built from beginning or gained during exploration, and DFS algorithm for generating a path. And all helpers methods for behaviors to update quickly.

- Start: (0, 0)
- Goal: (5, 8)
- Initialize 2D array for the maze

Row: i; Col: j;
#rows = 7; #cols = 10;

	0	1	2	3	4	5	6	7	8	9
6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	2	3	3	3	3	3	3	2	-1
4	-1	3	4	4	4	4	4	4	3	-1
3	-1	3	4	4	4	4	4	4	3	-1

2	-1	3	4	4	4	4	4	4	3	-1
1	-1	2	3	3	3	3	3	3	2	-1
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

1. Corner Cells (Shaded in orange) - set value to be 2:
    a. (1,1)
    b. (1,8) = (1, #cols-2)
    c. (5,1) = (#rows-2, 1)
    d. (5,8) = (#rows-2, #cols-2)

2. Border Cells (Shaded in yellow) - set value to be -1:
    a. for j = 0 to 9 (#cols-1)
        i. i = 0,
        ii. i = 6 (#rows-1),
    b. for i = 0 to 6 (#rows-1)
        i. j = 0,
        ii. j = 9 (#cols-1)

3. Other Border Cells (Not shaded) - set value 4
    a. for j = 2 to 7 (#cols-3)
        i. i = 1,
        ii. i = 5 (#rows-2),
    b. for i = 2 to 4 (#rows-3)
        i. j = 1,
        ii. j = 8 (#cols-2)

4. Inner Cells (Shaded in green) - set value 4

    ```
    for i = 2 to 4 (#rows-3)
        for j = 2 to 7 (#cols-3)
    ```

- Depth First Search Algorithm
    - Examine all visited cells
    - Be called by Explore to generate a path
- Reverse a path for **ReachGoal** to walk back to the starting cell
- Helper methods
    - isVisited(int row, int col)

    - Checks if the cell is visited before
- isValidMove(int row, int col)
    - Identifies whether a cell is a legal move
    - by checking the cell value != -1 or 1
- obstacleDetected()
    - Called by Explore once an obstacle is detected two cells away
    - Finds the cell in the maze and marks it visited
- obstacleAround(int row, int col)
    - Called consequently by obstacleDected()
    - Once an obstacle is detected.
    - Inform all adjacent cells should remove a path.
- setVisited(int row, int col)
    - Finds the cell in the maze and marks it visited
- setCurrObstacle()
    - Marks the current cell as an obstacle
- getCurrCell()
    - Returns the current cell
- getCurrOrient()
    - Compares previous cell and current cell to get an orientation
- getPrev()
    - Getter for the previous cell
- updateCurrCell(Cell cell)
    - Resets the current cell to the cell passed in by updating the coordinates
- commandForExplore()
    - called by Explore behavior
    - find a possible move
    - Identify the orientation for that move if exists
- nextCell()
    - Finds the succeeding step to be taken
- checkAdjancent()
    - It will check through adjacent neighbors to find all legal states succeeding the current cell
- orientation(Cell)
    - Identifies the orientation in order to make the next move
- getDFSPath()
    - Sets start state
- getReveresedPath()
    - Reverses a path

        - Returns a path
    - dfs()
        - Hold the Depth First Search Algorithm

Section 5: Benefits/Assumptions/Challenges

Benefits:
- We rely on sensors to help us build up knowledge for the world.
- The Assumption Architecture we are using helps up not only do more complex layers depend on lower, more reactive levels, but that they could also influence their behavior.
- It can switch between modes of behavior in response to a variety of stimuli detected by sensors.
- Default behavior will not be overwriting memory.
- Other behavior can come interrupt an ongoing behavior if the conditional is met.

Assumptions:
- We assume that our sensors will always detect an obstacle when that's not always the case
- We assume that a higher-level behavior may suppress a lower-level behavior
- We assume that the arbitrator class takes an array of Behavior objects as the argument to its constructor, and provides the higher-level control for the various behaviors
- We assume a continuous action such as our Explore Behavior class won't stop until it's suppressed
- We assume that priority index of the arbitrator is appropriate given the existing behavior properties

Challenges:
- Hard to identify the robot's orientation
- Robot doesn't always make a full turn as we want.
- It's hard to set a threshold for Light Sensor. If is low, it will always call on the goalReach behavior. If is high, it's hard to be activated.
- The TouchSensor doesn't work for most of time.
- It's hard to figure out if the robot is recalling the information or knowledge gathered during the search.
- You can only use print out to debug.

Section 6: Next Steps

If we were given more time for this project, we would try to figure out out the bug in our code and probably try a different search algorithm, possibly breadth-first search.

https://github.com/hu-zhiling-540/Robot-Maze-Navigation/tree/LASTTRY

Bibliography

1. http://www.lejos.org/nxt/nxj/tutorial/LCD_Sensors/LCD_Sensors.htm#5