Module: Agent Architecture

Stephanie E. August
saugust@LMU.edu

Andrew Won
andyjwon@gmail.com

Eric Jaso
ericjaso@gmail.com

Michael Fraser
mfraser702@gmail.com

Miguel Vazquez
mvazque93@gmail.com

Poulomi Chatterjee
poulomi.chatterjee22@gmail.com

Joaquín Loustau
joaquinloustau@gmail.com

Liyang Hao
lynnhh36@gmail.com

Parisa Pouya
ppouya@lmu.edu

Gustavo Dias
guhdias@gmail.com

Elizabeth Shen
eshen1@lion.lmu.edu

James Yen
jyen2@lion.lmu.edu

Artificial Intelligence Laboratory

This document is a TAILS module

Loyola Marymount University
Contents

1 The Idea 4
  1.1 Purpose .................................................. 4
  1.2 Background ............................................... 4
    1.2.1 Agent Architecture .................................. 4
    1.2.2 Agent Architecture: Value Diminishing Agent .......... 5
    1.2.3 Environmental Traversal ............................... 5

2 Applications 7

3 Input-Process-Output 9

4 Design 15

5 Implementation-Specific Design 18

6 Test Suite and Drivers 26

7 Experiments 27
  7.1 Experiment 1 ............................................. 27
  7.2 Experiment 2 ............................................. 27
  7.3 Experiment 3 ............................................. 27
  7.4 Experiment 4 ............................................. 27
  7.5 Experiment 5 ............................................. 27
  7.6 Experiment 6 ............................................. 27
  7.7 Experiment 7 ............................................. 27

8 Source Code 28
  8.1 Agent Architecture Source Code .......................... 28
List of Figures

1  Agent reacting to different scenarios ................................. 4
2  Autonomous Robot ......................................................... 7
3  Principle user interface for agent module ............................. 9
4  Example of alert that will appear when agents have finished running 12
5  Options chosen for this example ....................................... 12
6  This is what the environment looks like shortly after pressing start 13
7  This is what the environment looks like halfway through the run process 13
8  This is what the environment looks like at the end ................... 14
9  Use Case for this module .................................................. 16
10 Breakdown of the bubbles within User Interface from last figure .......... 17
1 The Idea

1.1 Purpose

The purpose of this lab is to familiarize the experimenter with the environment and agent relationship:

This lab will represent the relationship between an environment and an agent. The module consists of an environment represented by a grid of tiles. The agents acting on this environment are represented by the agent images on the grid. The experimenter will have a wide variety of manners in which they can change the environment. They will also have the ability to chose how they want the agent to move around the environment. With these tools the experimenter will be able to see the relationship of a sensor to the agent and an agent to effector.

1.2 Background

1.2.1 Agent Architecture

There are five main types of intelligent agents. They are simple reflex agent, model-based agents, goal-based agents, utility, and learning agents. This module shows the simple reflex agent type. The agent has sensors which give information on the environment to the agent. The agent will take this information and based on the algorithm given, make a decision on what to do to the tile it is currently on and which direction it should go next. Once it has decided on the proper actions to take it will send this information to its effector which will execute the actions.

As can be seen the environment has the potential to change at all times and the agents will react accordingly on the environment as demonstrated by the figures below.

![Figure 1: Agent reacting to different scenarios](image)

Figure 1: Agent reacting to different scenarios
Agent Architecture is normally made up of three components. They are sensors, effectors, and the agent. These three components work together in order to act on the environment.

One common example of Agent Architecture is in the autonomous vacuum problems. The autonomous robots rely on the usage of their sensors and effectors to know which direction to take and to clean the environment respectively.

1.2.2 Agent Architecture: Value Diminishing Agent

Agents are beings with the ability to ”think”. This lab will use an agent and have it move about the environment with different mentalities. That is to say, with each mentality it will give priority to different tiles or behave in a different manner than the other thought types it will have.

1.2.3 Environmental Traversal

As was stated agent architecture consists of three different components. This lab uses this idea and has the program work with three different parts

The first of these parts is the sensor. This is the part of the program that acts as the agents ”senses.” The sensor gathers data about the environment such as whether the agent is currently located above a tile with a value or which direction seems to have the highest value. All this information is sent to the agent part of the architecture.

The agent is the part of the architecture where the ”thinking” happens. This part will get the data it receives from the sensors and use it to decide what to do. Based on the ”thought” algorithm that it has programmed into it, an action or set of actions will be decided and set into motion through the effectors.

The effectors are the final part of the architecture. They are the part that acts on the environment. In this case they are the part that diminishes the value of the tile it is located on and of the environment as a whole.

This module comes with four complete agent movement algorithms. The experimenter will need to program two of the movement types but will be provided with an example to see what their movement should look like. The movement types are as follows: Simple Agent, Value Sensing Agent, Value Chasing Agent, and Mini-Mapping Agent.

Simple Agent will have the agent ”scan” the environment in order to estimate the size of the room in a measurement of tiles. That is to say it will figure out how many tiles exist on the environment to be acted upon. It will move in a snaking pattern of left to right moving up and down only at the edges or at random points. This program will move in this manner until it has gone over as many tiles as it had estimated there would be and then stop even if there are still tiles that need to be acted upon.

Value Sensing Agent will have the agent moving in a direction depending on the value of that direction. However should it ever end up at a point in which there is no value in all four directions it will have a movement similar to Simple Agent. The main difference between
the Simple Agent and Value Sensing Agent is the Value Sensing Agent will not stop moving until it has caused the environment to have a value of zero (except if the remaining values are trapped under another agent).

Value Chasing Agent will have the agent giving priority to tiles with a higher value. In this case the agent will choose to move to the neighboring tile with the highest value. Should all of its neighboring tiles have a value of zero it will behave in a manner similar to Value Sensing Agent.

Mini-Mapping Agent is essentially the same as Value Chasing Agent except for the fact that the agent will have a map that it will update with each move it makes. The map can be seen below the environment and shows how the agent has moved within the environment.
2 Applications

Agent Architecture is a big part of artificial intelligence. It is used in autonomous vacuums but could also be applied with tsunami warning systems, and even in traffic lights.

![Automonomous Robot](image)

Figure 2: Autonomous Robot

Consider the autonomous vacuums out in the market. These vacuums are acting based on what they sense. When they initially start they will scan the room to estimate how big it is and decide on how many hours they should spend cleaning it. Once that is done it will start to move in a spiraling motion cleaning anything it may pass over until it hits an obstacle. It will then clean along the obstacle ensuring that it changes directions whenever it’s bumpers tell it of an obstacle blocking its progress. Should it pass over a particularly dirty location it will go over it multiple times until it reaches a level of cleanliness it is programmed to consider as good. It will continue to act in this manner until it has cleaned for however amount of time it predicted it will finish cleaning in or its battery runs out.

![Buoys and battery](image)

A system used to warn of possible tsunamis involves buoys which will float out in open water and gather data based on the waves that move it. Should these waves have a certain pattern it will send a report to one of the main stations. This station will end up taking the data and look at other data (such as recent earthquakes and magnitudes). Based on the
information from its sensors it will either choose to ignore it or send out an alert stating there is a possible tsunami. This alert could be checked further in depth by people but it could give an early warning for normal citizens to evacuate or at least prepare to evacuate thus minimizing risk.

Traffic lights are another example of agent architecture being used. As you may know there are sensors to tell the traffic light if there is a car present or if a pedestrian is trying to cross the street. This sensory input affects the time the traffic light stays in a certain color. If no data is received the light will follow a simple algorithm that should have been given to it as a default. That is to say this system uses information gathered from its sensors to decide on how to affect the environment by changing its color.
3 Input-Process-Output

1. Input

The principle user interface can be found in the top center of the page. It has a variety of options for the user to change the experiment and its process that are explained further in depth below.

![Figure 3: Principle user interface for agent module](image)

**Modifiers**
This is where you will find the environment modifiers. By selecting the checkbox you indicate that you wish the modifier to take effect. The effects are as follows:

- **Children at Play**: There is a percent chance that a random tile will gain a value level
  - (a) This is the only way that a tile can have a value of four
- **Murphy’s Law**: There is a percent chance the agent will fail to decrease the value of the tile and instead increase the value
- **Child Interference**: A child will pick up the vacuum and drop it off at a random location
  - (a) Visually the agent will disappear from the point it was on and reappear in random tile
- **Hyper Action**: The agent can act two or three times on a tile based upon the percentages

**Modifier Percent** The textboxes in this section are related to the modifier they are next to. The number placed in the box represents the chance out of 100 that the effect will take place. For example if Children At Play has 100 then after every move a random tile will increase in value, whereas if a 50 were placed a tile would increment in value every other move.

**Number of Agents** The user will select the number of agents they wish to have acting upon the environment during the experiment. No matter what the user selects there will always be four textboxes and dropdown menus located below it.
Start Locations  The user will type in a number which corresponds a tile on environment. The agent will begin in that location and move according to its movement algorithm from there.

(a) If the user inputs a number that does not exist in the environment then the agent will be placed in the lowest available tile

(b) If the user tries to place the agent on top of an obstacle then the agent will be placed in the lowest available tile

Number of Initial Bugs  The user will type a number inside the textbox. This is the number of times the program will try to add value to the floor.

(a) The tiles that increment in value are chosen at random.

(b) If a tile is chosen more than once then its value will increase accordingly except if it has a value of four

(c) If a tile with value of three is selected it will count as having attempted to add a value and decrement the number of times it will try to add a value. However if it chooses an obstacle it will not count towards the number of tries.

Width Length  This indicates the width and the length of the environment the agent(s) will be acting on. Any size selected should fit perfectly within the section

Agent Type  Indicates the type of movement the user would like the agent to execute.

• Simple Agent
  
  (a) Agent will move in a horizontal manner and will go up or down if it reaches an edge or at random.
  
  (b) It will scan the environment to estimate the number of tiles in the room. It will then proceed to scan that many tiles and will stop once it has done so
  
  (c) There is a chance that the agent will stop even if it has not completely cleaned the floor

• Value Sensing Agent
  
  (a) Agent will move towards the direction that has the highest value sum (it will pick the direction with the highest value) and follow that direction until it can no longer follow that direction.
  
  (b) If it finds that all direction have a value of zero it will have a movement algorithm similar to simple agent (horizontal with occasional vertical movements)
  
  (c) It knows if the floor has a value and will not stop until the value of the floor is zero (meaning value of environment is zero)
(d) Agent will stop if a simple agent is covering the last values on the environment

- **Value Chasing Agent**
  
  (a) Agent will move onto the neighboring tile that has the highest value
  (b) If all the neighbors have a value of zero then it will begin to move like Value Sensing Agent
  (c) It knows if the floor has a value and will not stop until the value of the floor is zero (meaning value of environment is zero)
  (d) Agent will stop if a simple agent is covering the last values on the environment

- **Mini-Mapping Agent**
  
  (a) Same as Value Chasing Agent except it will create a mini-map of the path it has traveled and updates it with every move
  (b) Any Agent using Mini-Mapping Agent will also update the master Mini-Map which will be shown next to the actual moment

**Obstacle**  If the checkbox is selected then the user states they would like to have the obstacle functionality on the environment.

**Obstacle Locations**  This is where the user can give a list of locations they wish to have the obstacles in. The list should have the values separated by commas and the list should end with a comma.

**Sensor Failure**  The checkbox will indicate whether sensors will fail or not. The slider underneath it will indicate the percentage of failure.

**onClick Functionality**  Along with the user interface shown in the figure the user can also interact by clicking on the tiles.

- A left click will augment the value of the tile. It will not increase it past a value of three
- A right click will decrement the value of a tile. If the tile has a value of zero when it is right clicked it will become an obstacle and be added to the list of obstacles
- A double click on an obstacle will turn it into a normal clean tile and remove it from the list of obstacles
- Any values on tiles will reset when the run button is pressed. Obstacles will remain unaffected however.

2. **Process:**
   Once the user has selected the desired options and press the run button the agent will
begin to traverse the environment based on the type of agent the user selected. The agent will go from tile to tile and act upon it based on sensory data given to it. The agent will continue to act on tiles until its algorithm tells it that it is done at which point an alert will appear.

3. **Output:**
   Once the agents have finished running, an alert will appear to give a performance report of how they did. The alert should look similar to the one shown in the alert below.

![Alert Example](image)

Figure 4: Example of alert that will appear when agents have finished running

4. **Example:**
   Using the options shown in the interface this program will be run to show an example of what to expect.

![Agent Module Options](image)

Figure 5: Options chosen for this example
Figure 6: This is what the environment looks like shortly after pressing start

Figure 7: This is what the environment looks like halfway through the run process
Figure 8: This is what the environment looks like at the end
4 Design

For this program most of the work necessary will be done with the user interface. In each of the modifier percent textboxes a number should typed in if they are selected. This number will be used and a random number between 0 and 100 will be chosen. Should the random number be smaller than the number in the textbox then that modifier will take effect. If not it will be ignored for that move.

The start location textboxes will take the value located within and check to make sure that it is a valid starting position. If it is the program will take the agent and place it there once the run button is pressed. If the value is not valid then the program will place it in the lowest possible tile it can find.

Number of initial bugs will be given a number of tries from the textbox. Every try it will pick a random number that exists within the environment. The number it picks will be a number that will be incremented in value unless it has a value of three or it is an obstacle. If it is an obstacle it will not count as a try. The program will repeat this process until it has used up the number of tries it was given by the user.

Width and height causes the creation of the vertices. This will help govern how many vertices will be created in every row and however many rows will be created. Once the number of vertices in row equals the width a new row will be created. This process will continue until the number of complete rows equals the length. The obstacles list will be used to go through every vertex and mark obstacles as obstacles. The sensor failure number will also represent the value a random number between 0 and 100 needs to be less than in order to take effect.

Once the run button is pressed the agent will be placed in the designated start location. From there the agent will move in accordance to the algorithm type that was chosen for it unless Child Interference is active. If it is then a random number will be chosen to see if it takes effect for that move, if it is a random number is chosen and that is the tile number the agent will be placed on. At every tile it will use its sensor to gather information and based on this information it will either do nothing or it will act on the tile. If it acts on the tile that means that it will lower the value of the tile which will cause the tile to change color and image. If hyper action is active a random number will be chosen and will act on the tile the same number of times as the number that is chosen due to the random number. If sensor failure is in effect there is a chance that the agent will get to the tile and think it has a value of zero and move. It may also approach a tile with value of zero and think it has another value and attempt to clean it which will cause nothing to happen. If Murphy’s Law is active there is a chance the agent will end up increasing the value of the tile instead of decreasing its value at this point. All the agents will do this and when they are done if Children At play is active random spot will be chosen to have its value incremented based on the percentage given by the modifier’s text box. This process will continue until the agents state that they

15
are done at which point the alert will appear to give a progress report of how the agents did.

Figure 9: Use Case for this module
Figure 10: Breakdown of the bubbles within User Interface from last figure
5 Implementation-Specific Design

The modifiers on the user interface have a checkbox which is used in an if statement to decide whether there should be a chance that the selected modifier take effect. If the box is checked then a random number will be chosen using JavaScript’s Math.random() command. If this number is less than the number in the modifier’s textbox then the modifier will take effect.

How The Modifiers Work

• **Children At Play**

  - Random number is selected using Math.random
  - Function checks if random number is less than selected percentage for modifier and if Children at Play checkbox is selected.
    * If false function end doing nothing
    * If true function will select another random number and get the vertex with the same value. It will then use the vertex’s childrenAtPlay function to color it in.

• **Murphy’s Law**

  - Effectors decreaseValue function is used
  - Function checks if random number is less than selected percentage for modifier and if Murphy’s Law checkbox is selected.
    * If false vertex of current location will use colorDecrease function to diminish value of vertex and environment
    * If true vertex of current location will use colorAsValue function to increment value of vertex and environment.

• **Child Interference**

  - Variable ”luck ” is given a random value
  - Function checks if random number is less than selected percentage for modifier and if Child Interference checkbox is selected.
    * If false function returns the vertex the agent was located at
    * If true while loop is initiated until valid location is chosen and function returns that vertex
• Hyper Action

- Variable "luck" is given a random value
- Function checks if Hyper Action checkbox is selected.
  * If false function returns the value one
  * If true function checks if luck is less than percentage of the x3 action
    · If false function continues
    · If true function returns the value three
  * Function checks if luck is less than percentage of the x2 action
    · If false function continues
    · If true function returns the value two

How the agent options work

The start locations give the program values that will be checked once the run button is pressed. It will check the tiles given by the value to make sure they are valid and change the value if they are not. From there the chosen agent type algorithm will come into play and cause the movement of the agent. This a description of the movement types

• Simple Agent

The agents have an array that will hold the vertex it will act upon. At the start of its movement the array will have a value popped off. That is the vertex that will be acted upon. At every tile the agent will use its sensors in order to decide whether it should act on the tile or not. Should the sensor state that the tile needs to be acted upon the agent will call upon its effectors which will clean the tile. From there the agent will call upon the bumpers which are a part of sensors. This information will call upon the direction method which uses the sensors to decide if it can go left or right or if it needs to move up. There is also a random number generator that will cause the agent to give priority to vertical movement instead of horizontal some percentage of the time. This method will return the value of difference between the tile the agent is currently on and the tile it should be going towards according to the bumpers. That new vertex will be pushed onto the stack. After all the agents have moved the program will animate their movement to the tile they will be acting on next.

- The node that the agent is currently on will be placed in variable currentNode
- Function will check to see the number of tiles it has scanned matches the number of tiles it is meant to scan
  * If false function will continue
  * If true function will go to end stop the agent on the vertex it is located and stop all actions, still Running variable will be false and leftOverValue will be updated
childInterference will be used and possibly change the current location of the agent.
Agent will check to see if current location has a value
  * If true agent will act on the location however many times hyperAction tells it to
  * If false program will continue to next step
direction function will be used to chose the appropriate direction to move agent towards based on the bumpers information from sensors
For loop will be initiated to push the the next vertex based on information given by sensors
Program checks to see if stack is empty
  * If true current vertex will be pushed onto stack
  * If false program will continue to next step
private variables are updated and program ends.

• Value Sensing Agent
This agent acts similar to Simple Agent in the manner that it obtains the current vertex and acts upon it. After it has acted upon the tile it will use its lookAheadDirection method which will use the sensor’s lookAhead method to count the values in every direction from the agent. These values will be used by lookAheadDirection and sorted from highest to lowest. The highest value will be chosen as the direction to be used and the method will return the difference in value between the current value and the next value in the appropriate direction. If the lookAheadDirection finds that all four directions have a value of zero it will use the direction method and move like Simple Agent until it senses that a direction has a value. The value returned by the method will be used to find the vertex that will be pushed onto the stack to be used for the next move. Then as with Simple Agent once all agents have acted they will be animated to the new vertex.

The node that the agent is currently on will be placed in variable currentNode
Function will check to see if the value of the floor is zero
  * If false program will continue
  * If true program will go to end stop the agent on the vertex it is located and stop all actions, still Running variable will be false and leftOverValue will be updated
childInterference will be used and possibly change the current location of the agent
Agent will check to see if current location has a value
  * If true agent will act on the location however many times hyperAction tells it to
  * If false program will continue to next step
− The function lookAheadDirection will be used to see how much dirt is in each direction. Based on which ever direction is dirtiest a number will be returned to calculate the next tile the agent should move to.
− For loop will be initiated to push the the next vertex based on information given by sensors
− Program checks to see if stack is empty
  ∗ If true current vertex will be pushed onto stack
  ∗ If false program will continue to next step
− private variables are updated and program ends.

• Value Chasing Agent
  This agent type will start by popping off the vertex it will be acting on from the stack. This vertex will have the sensors check it to see if there is need for any action. If the sensors state that there is need for action it will also tell them how many times to act upon the location. Once the agent has acted upon the location it will us its valueOfNeighbors method. This method will get the value information of the neighbors and return an array with a boolean stating there is a neighbor with a value and the number related to the highest valued neighbor. Should the boolean be true this value will be used to get the respective vertex and push it onto the stack to be used later. Should the boolean be false it will follow the movement algorithm from Value Sensing Agent until it has neighbors with a value. Once all the agents have moved they will be animated onto their new vetex and the process will begin anew.

− The node that the agent is currently on will be placed in variable currentNode
− Function will check to see if the value of the floor is zero
  ∗ If false program will continue
  ∗ If true program will go to end stop the agent on the vertex it is located and stop all actions, still Running variable will be false and leftOverValue will be updated
− childInterference will be used and possibly change the current location of the agent
− Agent will check to see if current location has a value
  ∗ If true agent will act on the location however many times hyperAction tells it to
  ∗ If false program will continue to next step
− The function valueOfNeighbors will be used to choose the highest valued neighbors as the next destination.
− Check if valueInformation[0] is true
  ∗ If true push valueInformation[1] which is highest valued neighbor onto stack and skips next two steps
The function lookAheadDirection will be used to see how much dirt is in each
direction. Based on which ever direction is dirtiest the difference in values between
the tile in the correct direction and current tile will be returned.

For loop will be initiated to push the the next vertex based on information given
by sensors

Program checks to see if stack is empty
  * If true current vertex will be pushed onto stack
  * If false program will continue to next step

private variables are updated and program ends.

• Mini-Mapping Agent
  This agent type is identical to Value Chasing Agent except for a step in between
  cleaning the vertex and selecting the next vertex to move to. In between these two steps
  it will use a graph variable that is unique to itself. It will use the setupMiniMap method
to draw the information of its current point onto a smaller map that can be seen below
the actual environment. It will also give the information of its neighbors so that the
map will have that included as well. This way the agent has an understanding of where
it is and where it has been. It also has a second mini-map known as the masterMiniMap
which is shared by all the agents. This will also have its setupMiniMap method used
and update a smaller map that will appear next to the main map. All agents will update
this masterMiniMap with every move they make to show the collective progress of the
agents.

  The node that the agent is currently on will be placed in variable currentNode

  Function will check to see if the value of the floor is zero
    * If false program will continue
    * If true program will go to end stop the agent on the vertex it is located and
      stop all actions, still Running variable will be false and leftOverValue will be
      updated

  childInterference will be used and possibly change the current location of the agent

  Agent will check to see if current location has a value
    * If true agent will act on the location however many times hyperAction tells
      it to
    * If false program will continue to next step

  miniMap for this agent is updated using updateMiniMap function
  masterMiniMap is updated using updateMiniMap function

  The function valueOfNeighbors will be used to choose the highest valued neighbors
  as the next destination.
– Check if valueInformation[0] is true
  * If true push valueInformation[1] which is highest valued neighbor onto stack and skips next two steps
  * If false continues to next step
– The function lookAheadDirection will be used to see how much dirt is in each direction. Based on which ever direction is dirtiest the difference in values between the tile in the correct direction and current tile will be returned.
– For loop will be initiated to push the the next vertex based on information given by sensors
– Program checks to see if stack is empty
  * If true current vertex will be pushed onto stack
  * If false program will continue to next step
– private variables are updated and program ends.

Explanation of each file

- Agent Architecture.html
  This is where the appearance of the module comes from and the interface. This creates the buttons, textboxes, checkboxes, and legend that appears on screen. This is where one would edit the text that appears as well as the names given to the html elements that are used in the JavaScript parts of this module.

- basic search.js
  This is where the graph variable is created and made to appear on the screen. This is where the run button gets its functionality and ability to run the program. This is one of the files that would require change if more agents are to be added.

- environment.js
  This is where the agents are created and set up with the appropriate information. This is also where the agents are ordered to run based on whatever agent type they were chosen to be. This part of the code also keeps track of any values that have been branded as untouchable to allow other agents to stop if there is nothing else they can do to decrease the value of the environment. This is where one can change how the agents are run as well as how many. It is also where one could change when and how the program should end

- search.js
  This contains various search methods that can be used by the agent. At the current moment no agent type uses it however. This contains methods to find the quickest path between two vertices. All it needs is a goal vertex and the vertex the agent is currently located on. This will end up giving the respective minigraph variable a string with the path that the agent should follow to reach its goal.
Note: At the current moment the search.js is not aware of obstacles and other agents, that being said use with caution.

- **minigraph.js**
  This is where the mini maps that appear besides and below the main map are made. This will take the current vertex and the values of its neighbors then color the respective vertices the appropriate color. This is where one could choose how to color in the mini-map.

- **effectors.js**
  This contains the method that is responsible for decreasing the value of a vertex. It has an if statement that decideds whether Murphy’s Law will take effect and increase the value of the vertex instead of decreasing it. This is where one can change how the agent affects the environment.

- **sensors.js**
  This is the object that gather information about the environment. This is where it is decided what information the agent will have to help it make decisions. If the user wants the agent to behave in a different manner and requires more environmental information this where the changes in information should be made. This also holds some memory of where the agent traveled from and will update it when necessary.

- **agent.js**
  This is where most of the ”thinking” takes place. This is what will take the information from the sensor and decide on what to do as an action. This is where the movement algorithms will be found as well as the orders to use the sensor and the effectors as well as update the mini-map. This is also where the child interference will take place as the agent will be ”picked up” and ”dropped off” somewhere else which causes the agent to realize it is in another location.

- **graph.js**
  This is the visual that the agents act upon. This holds the collection of vertices, it creates them and connects them as neighbors. This object also has the method createEntireGraph which will make and display the visual that will appear on the module. It is also what holds the value currently present on the environment. It will also create edges between the neighbors. At the current moment edges are of no use but they could be used with A* within the search.js file to make a more efficient path finding method.

- **edge.js**
  These are the connections between the vertices. These contain a heuristic value which gives some sort of significance to the connection. This would be used with A* as it requires the edge values and there are plans to make the value be the total dirt between the two neighbors so that when searching for a path between two points one could ask for the most valuable path instead of the shortest.
• vertex.js
These are the individual tiles shown on the visual. These objects know their index and value. Should a vertex have to be colored or have value changed this is where the appropriate method will be found. This is where one can change the color of each type of vertex and what value they have. One could also change how much information each vertex know here for other movement types.

• animate.js
This is in charge of moving the html element which holds the agent so that it seems to be animated and moving across the board.
6 Test Suite and Drivers

The experiments for this module can be found in the folder titled Experiments For Agent Architecture. The experiments should be completed in order as some of the experiments build off of one another. In order to run the module select Agent Architecture.html and select the run button
7 Experiments

7.1 Experiment 1

This experiment can be found in the Experiments For Agent Architecture folder. This should be completed before Experiment 2 and Experiment 4 are attempted.

7.2 Experiment 2

This experiment can be found in the Experiments For Agent Architecture folder. This should be completed after Experiment 1 and before Experiment 4 is attempted.

7.3 Experiment 3

This experiment can be found in the Experiments For Agent Architecture folder. This should be completed before Experiment 4 is attempted.

7.4 Experiment 4

This experiment can be found in the Experiments For Agent Architecture folder. This should be completed after Experiments 1, 2 and 3 are completed but before Experiment 6 is attempted.

7.5 Experiment 5

This experiment can be found in the Experiments For Agent Architecture folder. This experiment can be completed at any time as long as simpleAgent is not used.

7.6 Experiment 6

This experiment can be found in the Experiments For Agent Architecture folder. It is recommended that this be attempted after completing Experiment 4 but it could be done without it.

7.7 Experiment 7

This experiment can be found in the Experiments For Agent Architecture folder. It is recommended that this be attempted after completing all other experiments but it could be done without them.
8 Source Code

8.1 Agent Architecture Source Code

The source code for this module can be found in the folder titled Source Code in the Agent Architecture Module folder.