Programming Tools for Artificial Intelligence Classes

A Senior Project presented to

the Faculty of the Computer Engineering Department

California Polytechnic State University, San Luis Obispo

In Partial Fullfillment

of the Requirements for the Degree

Bachelor of Science

By Christopher McKee

June 2013

1. Brief Summary

This project was begun in Winter of 2013 at California Polytechnic State University by myself, Christopher McKee, and was advised by Foaad Khosmood. The main goal of this project was to create an alternative to a current program that was being used in the Artificial Intelligence course on campus. The previous program used Java to create a Graphical User Interface (GUI) for students to use when learning how to create the various intelligent agents. In the versions created within the scope of this project, one of the environments used Java as the programming language, while the other two environments leveraged JavaScript for all programming.

2. Background

When implementing these environments, two separate libraries were used during the process. The first library used was the Greenfoot library, which was used in the aforementioned Greenfoot section of the project. The second library used was the EnchantJS library, which was used for the two versions that were implemented using JavaScript.

- Greenfoot is a package developed by the University of Kent in Canterbury, UK. This is a simple Java based Integrated Development Environment (IDE) that allows students, as well as other developers, create game like projects with ease. This ease of use is what made Greenfoot an ideal package for use in this senior project. When used, the Greenfoot IDE allows graphical access to the various classes in the environment, which are all extensions of the libraries Super classes: Actor and World. Through the extension of these two Super classes, the programmer is able to create fully animated worlds and characters with only a few methods.
- EnchantJS is an Application Programming Interface that allows a programmer to abstract calls that are needed when created projects that use Canvas. Canvas is a new element in the HTML5 standard. The Canvas element allows programmers to access to a 2 dimensional virtual canvas. Within that canvas, content creators are able to draw using primitives, such as arcs, lines, and

squares. This allows for easily managed 2 dimensional games. The EnchantJS library works to make 2D content creation even easier, with the aid of their internal Game and Sprite classes. The Game class of EnchantJS is the overarching container for all content within Enchant, containing all of your Sprites, as well as keeping a score internally, managing your frames drawn per second, and controlling the start and stop of the animation/action of the game. The Sprites are the basic player class, which contain a foreground image that can be changed via a frame selection from a sprite sheet.

The main aim of this project was to develop an environment where student would be able to program a logical agent, which is often associated with what is known as the Wumpus World. Within the Wumpus problem, it is up to the logical agent to use his senses to safely find a gold bar. The gold bar's location is never known to the agent at the beginning of execution, and so they are required to safely navigate throughout the entire world, eventually finding the gold using deduction. Deduction is actually the main skill that the logical agent is required to use. When navigating, they are only able to sense a breeze, a stench, a glitter, a scream, or a bump. These informative details are used to determine the safest path throughout the world. The breeze is indicative of close proximity to a pit, which if fallen into will kill the player. The stench tells the agent that they are within close proximity of a Wumpus, which is a creature that if collided with will eat the agent. The glitter allows the player to know that there is gold within their current location, which is the overall goal of the player. The only defense that the agent has to the vicious Wumpus is a bow and arrow, which the agent will need to aim at the creature and fire. In standard implementations, the agent is reduced to having a single shot, while for this implementation that value can vary based on the instructional use. When fired, the player will know of its success based on the Wumpus letting out a scream in pain when shot. The final sense of bumping is what the player must use when walking forward. They are essentially blind, with the exception of noticing glitter, which means that when walking forward, it is common that they will walk into a wall.

3. Project Details

As this project was created with the intention of being used within the Artificial Intelligence class on campus, I was also charged with creating assignments that would accompany the new environments. These assignments were largely adapted from the previous assignments used by Prof. Franz Kurfess, with introductions and environment specific implementations provided by myself. These assignments were created with the intention that the student, having had only the required pre-requisite course work completed, would be able to complete them given the knowledge of the course. This meant that, since we do not necessarily learn JavaScript within our course work, the assignments that leveraged EnchantJS all included a section detailing the necessary knowledge pertaining to JavaScript. The implementations were all created to be merely demoes of the API that I created within the environment for the students. Due to the fact that I have not taken the applicable AI coursework, the implementations created were elementary, which was beneficial in the end as this meant that there would not be any plagiarism regarding the provided source code.

This project began with the Greenfoot version, as this was the most similar to the prior environment used within the course. The first obstacle with this implementation was generating the world. When dealing with Greenfoot, since all drawn objects have to be an extension of the actor class, I was required to make the sense feedback items all actors. To get around having so many unnecessary classes, I instead made a single Other class that I then set visually based on the constructor. This allowed for the Greenfoot project to look a lot more organized, while still maintaining the visual appeal of the project. Another hurdle that I had to overcome was the abstracted movement calls. To help solve this I needed to save the current heading of the agent. This was done using an enumeration method, that I could then use in all future switch statements. This allowed me to correctly move in the faced direction when the movement within Greenfoot is based on the x and y position.

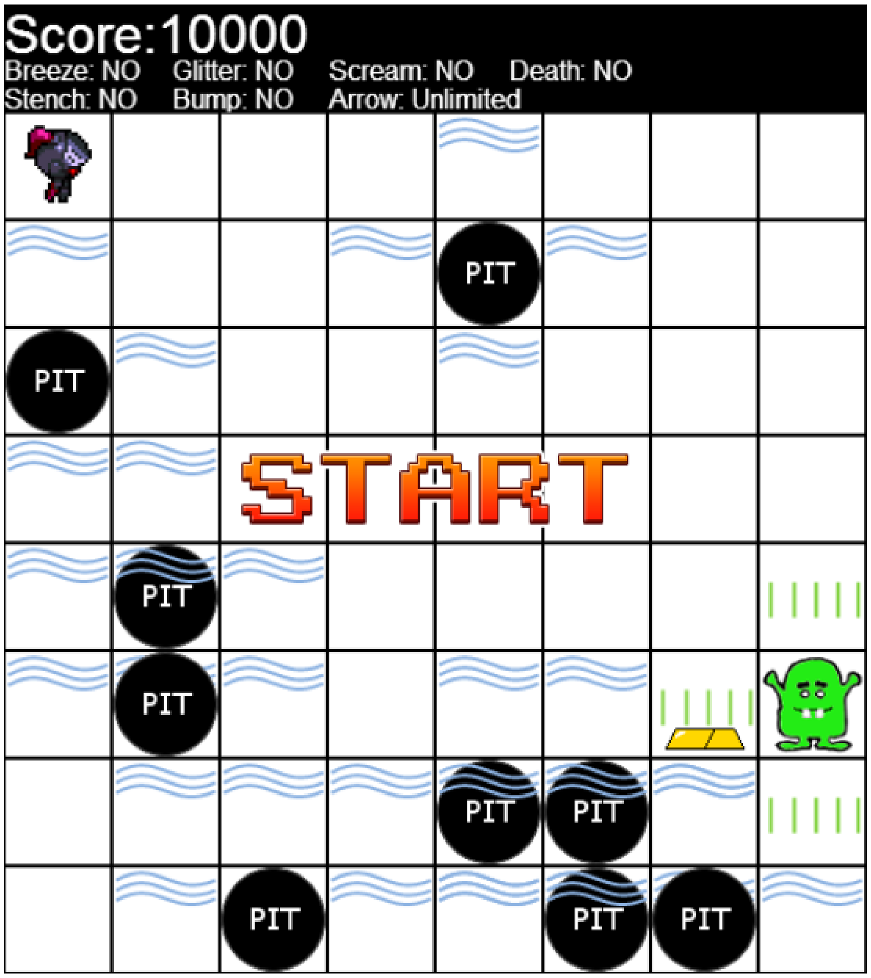


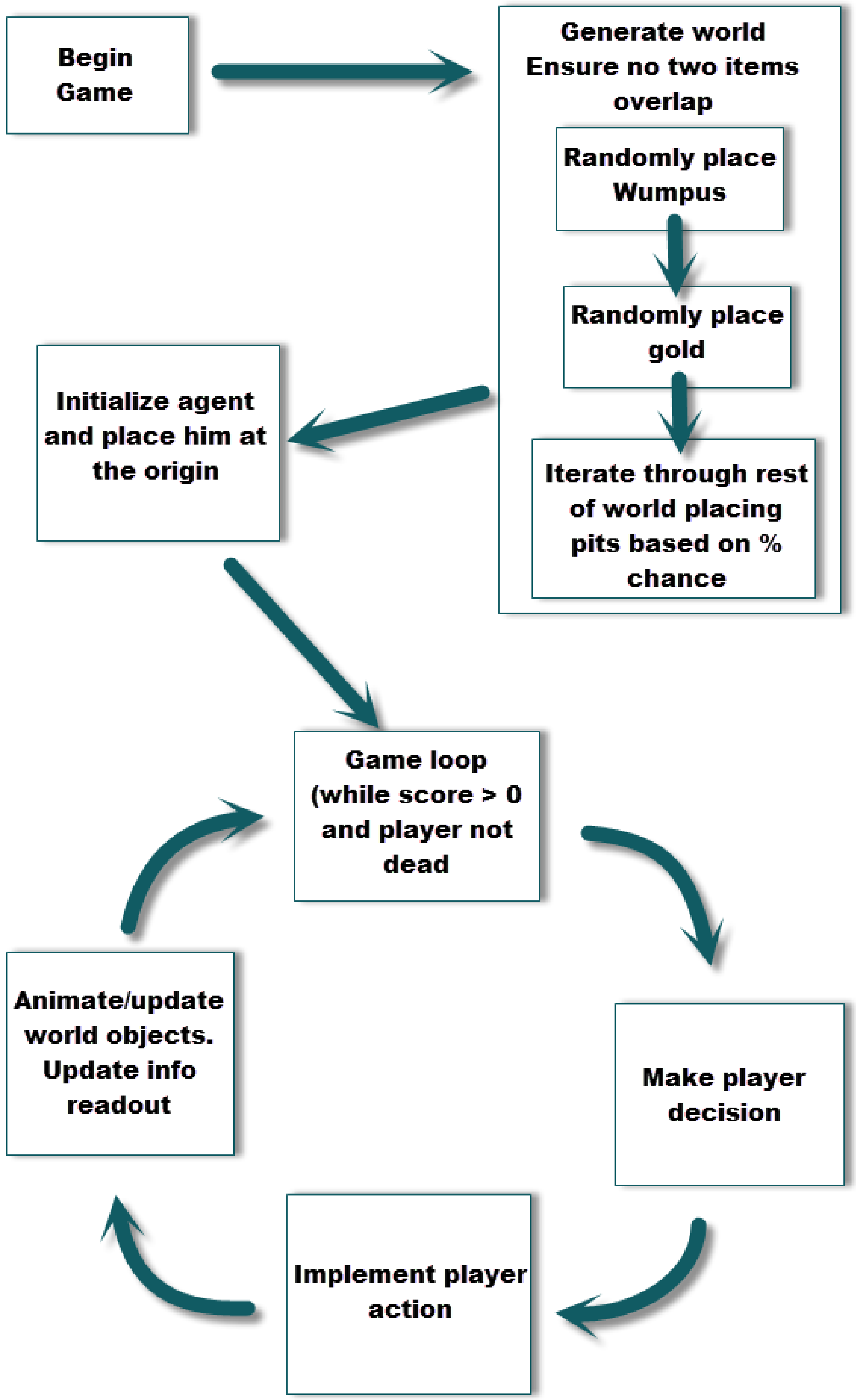
Figure 1: Image of the EnchantJS environment

The next section of the project turned towards Canvas and EnchantJS. When implementing the Wumpus World in EnchantJS, I was able to use the prior planning and lessons learned to help make this version better. When implementing the abstracted movement, I was able to leverage the data type looseness of JavaScript. Rather than needing the enumerated values that I used in Java, I was able to easily do comparisons based on the string values. This allowed me to be more direct when implementing the functions. In all, the frame based animation of the Sprite class made for a much more visual project. This was why the main production for this project was the Canvas components. When compared, though the Java is more familiar, the JavaScript iterations were much better designed and more easily implemented.

When it comes to the use for the classes, each game loop, be it the Act method within Greenfoot, which is run each turn automatically, or the onEnterFrame function of the EnchantJS version, each environment has a decision making function that the students will need to fill in. These functions are all based within the Agent(Player) classes, which limits the students to using only the information made available to that class. This made it so that I built only those sense based functions in, which although some call up to the world, or game, that they exist within, each of the functions will only return to the user the information that is relevant to the internal structure of the class. The student will be prompted to only edit this method, which will allow the grading to be simple as this means that in order to prevent cheating the instructor would only need to copy this method into a clean copy of the editable source.

The sample implementations for the Wumpus world environments were made to be as simple as possible, while still showing valid use of the functions. My sample implementation was one that would simply iterate through each of the senses, first glitter, followed by the stench and breeze senses. Each one of these senses would be set within conditionals statements, which if true would react accordingly, such as firing the arrow when you sensed a stench. In my testing, the AI was only able to win approximately 10% of games, which leaves ample room for improvement from the students. As all of these functions, when called, will decrement the player base score, on top of creating more effective AI, the students will also have room to make the most efficient AI as all of that will be clearly visible, based on the informative readouts within the environments.

4. Flow Diagrams


Begin Game
Generate world Ensure no two items overlap
Randomly place Wumpus
Randomly place gold
Iterate through rest of world placing pits based on % chance
Initialize agent and place him at the origin
Game loop (while score > 0 and player not dead
Make player decision
Implement player action
Animate/update world objects. Update info readout

5. Appendices

A. Methods within the Wumpus World that the students will use

turnLeft(): This function will cause the player to rotate their heading 90° counter clockwise. This function returns nothing and costs the student 10 points to use.

turnRight(): This function will cause the player to rotate their heading 90° clockwise. This function returns nothing and costs the student 10 points to use.

move(): This function will cause the student to travel one space forward in their current heading. This returns true when successful, and false when it fails, which is an indication of a bump. Invoking this function will cost the student 20 points.

senseBreeze(): This function tests to see if the player is currently standing on a breeze. This returns true when on a breeze, and false otherwise. Invoking this function will cost the player 1 point.

senseStench(): Similar to senseBreeze(), but instead is a sense for the stench.

senseGlitter(): Similar to senseBreeze(), but instead tests to see if the player us standing on the gold.

takeGold(): Grabs the gold from the plot of land where the agent currently is. This will end the game if successful, which in turn adds 10,000 points to the players score. Invoking this function will also invoke the senseGlitter() function, which applies the cost of said function when invoking takeGold() as well.

shoot(): This function will fire an arrow in the direction that the agent is currently facing. This function is dependent on how many arrows the agent is still holding. If the player exhausts all of their arrows, this function will return false and no penalty will be deducted. If the player does have arrows, the player will lose 20 points when the arrow fires. If the Wumpus is detected within the path of the arrow, the function will return true, otherwise it will return false.

B. Assignments Created

a. Assignment created for the Greenfoot environment

Wumpus World in Greenfoot

The Wumpus World with Greenfoot uses Java programming within the Greenfoot environment to control your logical agent. The Greenfoot environment uses two main classes to control the world, which are the World, and Actor classes. In this implementation, the main class of concern to the student is the Agent class, which extends the actor. What you need to know about this implementation is that at each refresh, the Agent will execute whatever is in its act method, which is your makeNextMove method. This means that within this method, you need to use your sensing data to determine your next move of action.

Programming the hunter

The goal for this task is to build an agent that can navigate the Wumpus World successfully: It should find a safe path to the gold (if there exists one). Use the characteristics for the Wumpus World as discussed in the class and described in the textbook.

You can make the following assumptions:

- the world is a rectangular grid, and your agent knows this;

- your agent does not know the dimensions and configuration of the environment;
- a square may contain only one entity (e.g. pit, wumpus, gold); however, a square may contain multiple percepts (a breeze and a smell, for example);
- there is only one Wumpus in the world, and (at least in this assignment) the wumpus doesn't move;
- your agent has a large (practically unlimited) supply of arrows, but each shot has a cost.

An important aspect of this task is the representation of the knowledge the agent has (e.g. the state of the Wumpus World), and the reasoning the agent performs in order to decide about the safety or risk of squares. It is also advisable for your agent to indicate its internal world model to the user. This could be an additional display showing the agent's knowledge of the Wumpus World at a particular point, or a text-based printout of the current status.

There is a modified version of the BotEnvironment for the Wumpus World task; please check Blackboard for the latest version. Please note that this version provides an infinite supply of arrows to your agent, in contrast to the limit of one arrow in the original description.

This task is to be performed in two parts. In the first part, you determine the methods and techniques you use for the representation of knowledge about the Wumpus World and for reasoning, e.g. about the properties of the squares, or the location of the wumpus. The second part is the actual implementation.

Part 1: Knowledge Representation and Reasoning

Describe the knowledge representation and reasoning methods that you are planning to use for this assignment. Knowledge representation relates directly to the data structures you use. To determine a suitable set of data structures, you need to take into consideration the main entities, their properties, what the agent needs to know about them, and how the actions of the agents affect the environment. The reasoning mechanism is to some degree dependent on the data structures chosen for knowledge representation. Here you should consider the combination of existing knowledge into new knowledge (e.g. properties of locations that your agent can deduce from known properties of other locations), the progress of the agent towards a goal through sequences of actions, and the making of decisions on which action to choose.

Part 1 Grading

We will use the following grading guidelines for Part 1:

- 5 points: Knowledge representation
- 5 points: Reasoning abilities of the agent

Part 2: Going for the Gold

Based on your first part, the agent now has to find a path to the square with the gold, and pick up the gold. Your agent needs to determine the properties of a given square, based on the direct information received from the environment, and on previously gathered information. So for each of the adjacent squares of the current square the agent should determine the following conditions:

is safe (empty) [E]

contains an obstacle (including walls) [X]

may contain a pit (possibly with a risk factor)	[P? 0.9]
is known to contain a pit	[P!]
may contain the Wumpus (possibly with a risk factor)	[W? 0.9]
is known to contain the Wumpus	[W!]
contains the gold	[G]

Agent Performance

Note: Some specific aspects of the calculations in this section may have to be adapted to fit the WumpusEnvironment.

The performance of the agent is scored as follows:

P(A) = Life-Points - Search-Cost - Path-Cost + Gold-Value

The agent has 10,000 life points to start with. If the agent encounters a Wumpus minion, it loses 1,000 life points. Finding the gold adds 10,000 points to the agent's performance score. The overall score of the agent at the end of the game will be used to judge its performance.

The life span of an agent is affected by the following events:

- If the agent's life goes down to zero, it dies.
- If the agent falls into a pit, it dies.
- If the agent enters a square occupied by the Wumpus, the agent dies.
- If the agent dies or finds the gold, the game is over.

The search costs for the agent are as follows:

- 1 for each interaction with the manager (e.g. for each query about a sense)

The path costs for the agent are:

- 20 for a move from one location to the location in front of the agent
- 10 for a turn action

Other costs accrued are:

- 20 for each arrow fired

In this task, the agent needs to perform its activities in an online mode: It can only detect the properties of a location once it is there. While it would be nice for the agent to explore the cave from the safety of its home base by sending out the fairy, and then simply taking the safe route to the gold (if there is one), it would diminish the purpose of the game. This means that the agent has to perform all the search and movement actions itself, and it has to rely on the current and past percepts.

The configuration for the Wumpus world with the location of the gold, pits, wumpus, and obstacles will be provided through the environment. Your agent is expected to work not only with sample environments, but with any environment that adheres to the Wumpus World specification as provided in the textbook and in the description of this assignment.

Part 2 Grading

We will use the following grading guidelines for Part 2:

- 10 points: Successful completion of the task
- 10 points: Performance of the agent
- 3 points for the README file and documentation of the source code

Part 2 Sample

As part of the assignment, you will need to complete the makeNextMove() method in the Agent class. This function is called automatically by the player each turn, so you will not need to worry about calling this manually.

The functions available to the agent are:

- senseBreeze(): This will return true if the player is currently standing in a breeze, false otherwise.
- senseStench(): This will return true if the player is currently standing in a stench, false otherwise.
- senseGlitter(): This will return true if the player is currently standing on the gold, false otherwise.
- move(): This will return true if the player was able to move forward. If this function returns false, this is an indication of a bump to the player, meaning they are facing a wall.
- turnLeft(): This rotates the agent 90 degrees to their left.
- turnRight(): This rotates the agent 90 degrees to their right.
- shoot(): This will fire an arrow along its current path. If this returns true then the player was able to shoot and hit the wumpus. Note: When firing the player will get instant feedback for an impending hit, but the arrow will travel in real time until it makes the actual collision. Because of this, if the player begins walking on the same turn as the player shoots the arrow, they have the possibility of walking into the wumpus before the arrow has a chance to kill it. This means that the shoot and move command cannot be performed on the same turn.

Some other relevant properties of the world/player:

- bumped: This is a Boolean value that is set from the return of the move method. This property should only be read, and not overwritten. This indicates whether or not the player was able to move forward on the last execution of the move function.
- heading: This is a String that indicates the direction that the player is currently facing. The possible values are EAST, SOUTH, WEST, and NORTH.

A sample implementation is:

```
private void makeNextMove()
{
    //senseGlitter detects if you are on the gold
```

```
        if(this.senseGlitter() == true)
        {
            world.takeGold();
        }
        //senseStench detects if you are currently adjacent to the wumpus
        else if(this.senseStench() == true)
        {
            //shoot fires the arrow along your current heading, returning true on a hit
            if(this.shoot() == false)
                this.turnLeft();
        }
        //senseBreeze detects if you are adjacent to a pit
        //bumped indicates if you previously failed to move forward
        //turning indicates if you preciously rotated your agent
        else if((this.senseBreeze() == true || this.bumped == true) && this.turning == false)
        {
            bumped = false;
            if(Greenfoot.getRandomNumber(1000) < 500)
                this.turnLeft();
            else
                this.turnRight();

            turning = true;
        }
        else
        {
            //player moves forward one space, if this
            if(this.move() == false)
                bumped = true;
            else
                bumped = false;

            turning = false;
        }
}
```

This function will either shoot, turn left or right, or move forward each turn. As noted in the shoot info above, the shoot and move commands are intentionally made to be separate conditional blocks. On each execution, the player will first sense for a stench. If found, the player will shoot, turning to its left if the shot is unsuccessful in killing the wumpus. This is used to have a guaranteed kill of the wumpus. If there is no stench, the player checks for a breeze or a bump. I have added an additional var to the player to track whether or not I turned the player on my previous turn. This third condition is to stop the player from spinning endlessly. If the player is either bumped or in a breeze, and not already turning, then I will randomly turn. Note that the Greenfoot has its own random function, that will return a number between 0 and the parameter value. My third main condition is the fallback, meaning that if I have no senses, implying a safe square, I will move forward. Also, each turn the agent will check for the gold in the current location. This can either be done at the beginning or end of each turn. The choice of when is up to your own implementation.

Running the code

The provided code includes the full Greenfoot project. You will need to download the IDE from Greenfoot.org, and then open the ,project file. The Act button will allow you to step through the frames(turns) one at a time, while the run button will simply run until the game ends. The speed selector can be used to increase or decrease your run time, so as to help you debug.

b. Assignment created for the Wumpus Canvas environment

Wumpus World in Enchant JS

The Wumpus World with EnchantJS uses JavaScript to control your logical agent. The first hurdle then, for any of you unfamiliar, is to use JavaScript for programming the agent. As a quick intro, I will cover some JavaScript principles that you will find necessary to complete this assignment.

First off, when using JavaScript all variables are simply a var type. You will not need to distinguish between say an int or string.
Ex.

```
   var sample = 7;
   sample = “A different value”;
```

These two lines are valid assignments in JavaScript.

Secondly, JavaScript is an object oriented language. All of the objects will have properties, which can be accessed using dot notation.
Ex.

```
   obj = new Object();
   obj.runFunc();
   obj.value = 10;
```

In this example, the object above has a function called runFunc(), and a property named value.

Aside from this, JavaScript is an easy enough language, and almost all questions regarding further syntax or usage are available through Google.

Programming the hunter

The goal for this task is to build an agent that can navigate the Wumpus World successfully: It should find a safe path to the gold (if there exists one). Use the characteristics for the Wumpus World as discussed in the class and described in the textbook.

You can make the following assumptions:

- the world is a rectangular grid, and your agent knows this;
- your agent does not know the dimensions and configuration of the environment;
- a square may contain only one entity (e.g. pit, wumpus, gold); however, a square may contain multiple percepts (a breeze and a smell, for example);
- there is only one Wumpus in the world, and (at least in this assignment) the wumpus doesn’t move;

- your agent has a large (practically unlimited) supply of arrows, but each shot has a cost.

An important aspect of this task is the representation of the knowledge the agent has (e.g. the state of the Wumpus World), and the reasoning the agent performs in order to decide about the safety or risk of squares. It is also advisable for your agent to indicate its internal world model to the user. This could be an additional display showing the agent's knowledge of the Wumpus World at a particular point, or a text-based printout of the current status.

There is a modified version of the BotEnvironment for the Wumpus World task; please check Blackboard for the latest version. Please note that this version provides an infinite supply of arrows to your agent, in contrast to the limit of one arrow in the original description.

This task is to be performed in two parts. In the first part, you determine the methods and techniques you use for the representation of knowledge about the Wumpus World and for reasoning, e.g. about the properties of the squares, or the location of the wumpus. The second part is the actual implementation.

Part 1: Knowledge Representation and Reasoning

Describe the knowledge representation and reasoning methods that you are planning to use for this assignment. Knowledge representation relates directly to the data structures you use. To determine a suitable set of data structures, you need to take into consideration the main entities, their properties, what the agent needs to know about them, and how the actions of the agents affect the environment. The reasoning mechanism is to some degree dependent on the data structures chosen for knowledge representation. Here you should consider the combination of existing knowledge into new knowledge (e.g. properties of locations that your agent can deduce from known properties of other locations), the progress of the agent towards a goal through sequences of actions, and the making of decisions on which action to choose.

Part 1 Grading

We will use the following grading guidelines for Part 1:

- 5 points: Knowledge representation
- 5 points: Reasoning abilities of the agent

Part 2: Going for the Gold

Based on your first part, the agent now has to find a path to the square with the gold, and pick up the gold. Your agent needs to determine the properties of a given square, based on the direct information received from the environment, and on previously gathered information. So for each of the adjacent squares of the current square the agent should determine the following conditions:

is safe (empty)	[E]
contains an obstacle (including walls)	[X]
may contain a pit (possibly with a risk factor)	[P? 0.9]
is known to contain a pit	[P!]
may contain the Wumpus (possibly with a risk factor)	[W? 0.9]

is known to contain the Wumpus	[W!]
contains the gold	[G]

Agent Performance

Note: Some specific aspects of the calculations in this section may have to be adapted to fit the WumpusEnvironment.

The performance of the agent is scored as follows:

P(A) = Life-Points - Search-Cost - Path-Cost + Gold-Value

The agent has 10,000 life points to start with. If the agent encounters a Wumpus minion, it loses 1,000 life points. Finding the gold adds 10,000 points to the agent's performance score. The overall score of the agent at the end of the game will be used to judge its performance.

The life span of an agent is affected by the following events:

- If the agent's life goes down to zero, it dies.
- If the agent falls into a pit, it dies.
- If the agent enters a square occupied by the Wumpus, the agent dies.
- If the agent dies or finds the gold, the game is over.

The search costs for the agent are as follows:

- 1 for each interaction with the manager (e.g. for each query about a sense)

The path costs for the agent are:

- 20 for a move from one location to the location in front of the agent
- 10 for a turn action

Other costs accrued are:

- 20 for each arrow fired

In this task, the agent needs to perform its activities in an online mode: It can only detect the properties of a location once it is there. While it would be nice for the agent to explore the cave from the safety of its home base by sending out the fairy, and then simply taking the safe route to the gold (if there is one), it would diminish the purpose of the game. This means that the agent has to perform all the search and movement actions itself, and it has to rely on the current and past percepts.

The configuration for the Wumpus world with the location of the gold, pits, wumpus, and obstacles will be provided through the environment. Your agent is expected to work not only with sample environments, but with any environment that adheres to the Wumpus World specification as provided in the textbook and in the description of this assignment.

Part 2 Grading

We will use the following grading guidelines for Part 2:

- 10 points: Successful completion of the task
- 10 points: Performance of the agent
- 3 points for the README file and documentation of the source code

Part 2 Sample

As part of the assignment, you will need to complete the makeDecision() function in the Player class. This function is called automatically by the player each turn, so you will not need to worry about calling this manually.

The functions available to the agent are:

- senseBreeze(): This will return true if the player is currently standing in a breeze, false otherwise.
- senseStench(): This will return true if the player is currently standing in a stench, false otherwise.
- senseGlitter(): This will return true if the player is currently standing on the gold, false otherwise.
- move(): This will return true if the player was able to move forward. If this function returns false, this is an indication of a bump to the player, meaning they are facing a wall.
- turnLeft(): This rotates the agent 90 degrees to their left.
- turnRight(): This rotates the agent 90 degrees to their right.
- shoot(): This will fire an arrow along its current path. If this returns true then the player was able to shoot and hit the wumpus. Note: When firing the player will get instead feedback for an impending hit, but the arrow will travel in real time until it makes the actual collision. Because of this, if the player begins walking on the same turn as the player shoots the arrow, they have the possibility of walking into the wumpus before the arrow has a chance to kill it. This means that the shoot and move command cannot be performed on the same turn.

Some other relevant properties of the world/player:

- player.bump: This is a Boolean value that is set within the move function. This property should only be read, and not overwritten. This indicates whether or not the player was able to move forward on the last execution of the move function.
- worldWidth: This is an integer value that is the set width of the playing field. As in all CS applications, the integer value indicated is one more than the highest index in the player coordinates, ie. If the worldWidth = 8, then the player location will be within 0 and 7.
- Heading: This is an integer value that indicates the direction that the player is currently facing. The possible values are 0-3, indicating East, South, West, and North in that order.

A sample implementation is:

```
makeDecision: function() {
  if(this.senseGlitter() == true)
  {
    this.takeGold();
    return;
  }

  if(this.senseStench() == true)
  {
    this.shoot();
    this.turnLeft();
  }
  else if((this.senseBreeze() == true || this.bump == true) && this.turning == false)
  {
    if(Math.random() < 0.5)
      this.turnLeft();
    else
      this.turnRight();
    this.turning = true;
  }
  else
  {
    this.move();
    this.turning = false;
  }
}
```

This function will either shoot, turn left or right, or move forward each turn. As noted in the shoot info above, the shoot and move commands are intentionally made to be separate conditional blocks. On each execution, the player will first sense for a stench. If found, the player will shoot, turning to its left if the shot is unsuccessful in killing the wumpus. This is used to have a guaranteed kill of the wumpus. If there is no stench, the player checks for a breeze or a bump. I have added an additional var to the player to track whether or not I turned the player on my previous turn. This third condition is to stop the player from spinning endlessly. If the player is either bumped or in a breeze, and not already turning, then I will randomly turn. Note that the random function in JavaScript returns a value from 0 to 1, so I will turn half of the time in either direction. My third main condition is the fallback, meaning that if I have no senses, implying a safe square, I will move forward. Also, each turn the agent will check for the gold in the current location. This can either be done at the beginning or end of each turn. The choice of when is up to your own implementation. Note that if you check at the beginning of the turn, you will want a return from the function on true so that you do not run through the other senses and unnecessarily reduce your score/performance.

Running the code

The provided code includes 4 JavaScript files, a number of image assets, and one html files. The file of your interest will be main.js. Here is where you will find the wumpus code, as well as the makeDecision function. Using an editor of your choosing, edit this main.js file. The html file is already linked to this file, so you will just need to make sure that the four files are all in the same directory when running the html file. To run the html, open it using a web browser, such as Chrome or Firefox.

c. Assignment created for the A* Canvas environment

A* in Enchant JS

EnchantJS uses JavaScript to control the game state. This means that you will need to implement JavaScript code in order to navigate the world with your agent. As a quick intro, I will cover some JavaScript principles that you will find necessary to complete this assignment.

First off, when using JavaScript all variables are simply a var type. You will not need to distinguish between say an int or string.
Ex.

```
  var sample = 7;
  sample = “A different value”;
```

These two lines are valid assignments in JavaScript.

Secondly, JavaScript is an object oriented language. All of the objects will have properties, which can be accessed using dot notation.
Ex.

```
  obj = new Object();
  obj.runFunc();
  obj.value = 10;
```

In this example, the object above has a function called runFunc(), and a property named value.

A final concept that you will need in order to complete this assignment is arrays in JS. Arrays in JavaScript are similar to an object. They are created by making a “new Array()” for the var. They can then be indexed similar to a standard array by using bracket notation. Another key detail is that arrays can contain varied data, ie. they are not limited to a single data type.
Ex.

```
  arr = new Array();
  arr[0] = “some string”;
  arr[1] = 7;
```

Aside from this, JavaScript is an easy enough language, and almost all questions regarding further syntax or usage are available through Google.

Assignment

In this assignment, you will explore various aspects of search methods. Your task is to program an agent that systematically investigates its environment by using some of the search methods discussed in class.

Search Methods

In this assignment, you have to implement the following search methods:

- uniform-cost (lowest-cost-first) search
- greedy best-first search
- A* search

The goal of this lab is to implement an A* search algorithm for the Wumpus agent. Since the A* algorithm is a combination of two more basic ones, uniform-cost and best-first search, you will implement these methods first, and then combine them into the A* method. For all three methods, the agent uses an off-line search: First, it constructs a path to the goal, visualized through the "fairy". Second, the agent carries out this path. Once the agent reaches the goal, print out the search cost and path cost to the logging window.

The task of the agent is as follows: It starts from its initial position (default is the tile in the upper left corner) and has to reach a certain tile identified as the goal position. On its path to the goal, it may have to clean tiles, and the amount of dirt on a tile affects the path cost of the agent. For some of the search methods, the agent needs additional information to estimate the cost from the current node to the goal node. In this case, the agent can ask the field manager for a hint. The manager may provide some hints, such as:

The distance between the current and the goal node.

Off-Line vs. On-Line Search

In an artificial environment, it is frequently easy to perform an off-line search by first examining the search space, determining the best (or a "good enough") solution, and then executing the respective actions.

In a real-world setting, the agent might have to perform an on-line search, interleaving the computations to expand the search space with the execution of the corresponding steps in the real world.

In this assignment, you implement an off-line search. After the agent has identified the goal, it should determine the best possible solution among the ones it explored, perform the actions to achieve the goal, and print it to the display and the log file.

This approach has an important consequence for the agent. When an off-line agent encounters a dead end during its investigation (i.e. there are no uninspected reachable squares from the current square), it can continue its investigation with another square on the fringe easily by "mentally" jumping to that square. An on-line agent, however, has to back up and retrace its steps in order to reach the next square on the fringe, thus increasing the path cost considerably. When you calculate the cost, make sure to differentiate between thesearch cost and the path cost.

Implementation Hints

The agents must keep track of the tiles it has already examined, and for some search methods, it must be able to backtrack (retrace its steps). The agent should build a data structure that tracks the path it has taken so far, allowing it to retrace its steps. The agent can also construct its own internal map, which may make navigation considerably easier. Please note that the agent does not know all relevant aspects of the environment in advance, such as the dimension of the playground, or the placement of obstacles.

An easy way to store the evaluator functions "f" for these nodes is to use arrays of constant dimensions.

Recall that your searchStep() function should implement only the search part of finding a path to the goal. The movementStep() function should carry out this path.

Your program must provide the following information:

search method used

number of search steps performed

current node in the search tree (position of the "fairy")

path cost to the current node g(n)

estimated cost from the current node to the goal (heuristic) h(n), if applicable

estimated total cost f(n) = g(n) + h(n)

the configuration of the field (size and name of the map)

This information should be updated as it changes, and written to the console using console.log() for later analysis.

Assignment Submission

This assignment must be submitted electronically via PolyLearn by the deadline specified in the class schedule. Please submit the following material, preferably in an archive (.zip, .gz, .rar or .tar):

a plain text file (not a MS Word document) named README.txt with your name, a brief explanation of your program, and instructions for running your program

the JavaScript source code for your agents

screen shots for the "Search Map"; the screen shots should show the agent view, and the final score; use names that indicate the respective map (e.g. fkurfess-Challenge-results.png)

log files that capture the performance of your agent on the maps provided; please save them as plain text files, and use names that indicate the respective map (e.g. fkurfess-Challenge-results.txt)

Naming Conventions

Please use your Cal Poly login as the first part of your agent's name, and an indication of the search method as the second. For example, my agents would be fkurfess-greedy, fkurfess-uniform, and fkurfess-astar. This will allow us to keep all agents in the Agents directory, without having to edit your files or moving files back and forth when we do the grading.

Collaboration

This is an individual assignment. If you’re using code fragments from the related lab exercise with the breadth/depth-first agent done in a team, please indicate this in the README file. It is fine with me to discuss general aspects of this lab with others (e.g. general aspects of the different search strategies).

Questions about the Assignment

If you have general questions or comments concerning the programming aspects of the homework, post them on the PolyLearn Discussion Forum. The grader and I will check that forum on a regular basis, and try to answer your questions. If you know the answer to a support or clarification question posted by somebody else, feel free to answer it; this will count as extra participation credit.**Part 2 Sample**

As part of the assignment, you will need to complete the makeDecision() function in the Player class. This function is called automatically by the player each turn, so you will not need to worry about calling this manually.

The functions available to the agent are:

- move(): This will return true if the player was able to move forward. If this function returns false, this is an indication of a bump to the player, meaning they are facing a wall.
- turnLeft(): This rotates the agent 90 degrees to their left.
- turnRight(): This rotates the agent 90 degrees to their right.
- checkIfSafe(xpos, ypos): This returns true if the square is safe, or false if there is a barrier/pit within that square. This will allow you to test whether you can navigate to the square in your path towards the goal.
- distanceToGoal(): This will return the number of tiles between the agent and the gold. This does not take into account the safe path, only the absolute path between the two.
- getGoalX(), getGoalY(): These can be used to get the position of the gold. This can be useful when finding which direction the gold is in relation to the agent.

A sample implementation is:

```
makeDecision: function() {
        if(this.move() === false) //Checks to see if you ran into a wall
                this.turnRight();
        if(this.distanceToGoal() > this.previousDistance)
                this.turnRight();

        this.previousDistance = this.distanceToGoal();
}
```

This is a simple solution that will navigate the agent in an attempt to get closer to the gold.

Running the code

The provided code includes 4 JavaScript files, a number of image assets, and one html file. The file of your interest will be main.js. Here is where you will find the player code, as well as the makeDecision function. Using an editor of your choosing, edit this main.js file. The html file is already linked to this file, so you will just need to make sure that the four files are all in the same directory when running the html file. To run the html, open it using a web browser, such as Chrome or Firefox.

C. References Used

http://wise9.github.io/enchant.js/doc/plugins/en/index.html http://www.raywenderlich.com/23370/how-to-make-a-simple-html5-game-with-enchant-js

http://users.csc.calpoly.edu/~foaad/enchant/

http://www.greenfoot.org/files/javadoc/