CSCI 446 Artificial Intelligence
Project 2 Design Report

Roy Smart Nevin Leh Brian Marsh

October 7, 2016

1 Introduction

Logic and the Wumpus World is an artificial intelligence problem first proposed by Michael Genesereth, and described in detail by his student Stuart Russel[1]. The problem involves navigating an environment known as the Wumpus World using logic to avoid dangers as the agent attempts to reach a goal. The environment is a square grid of tiles that can be empty or contain gold for the goal state, a pit that the agent will fall into, or a monster known as a wumpus. As the agent navigates the environment, a score is calculated from the various actions that the agent makes. The objective of the game is thus to maximize the score, which can be achieved by using logic to find the most efficient route.

2 Problem Statement

To solve the Wumpus World problem, we will develop a problem generator that creates the wumpus worlds in which the agent will navigate. We will then implement a logic system that uses unification and resolution on first-order rules that allow the agent to navigate. The reasoning system will use the following information: the stench when near a Wumpus, the breeze when near a pit, the scream of a wumpus as it is killed, and the wall when ran into. Additionally, a reactive agent will be created that makes decisions based upon random decisions regarding which cells are safe and which are not. We will test the logic system on wumpus worlds of sizes 5x5, 10x10,, 25x25. Our performance metrics are as follows: number of times the gold is found, number of wumpi killed, number of times the explorer falls into a pit, number of times the wumpus kills the explorer, and number of cells explored.

To solve the Wumpus World problem, we will develop a problem generator that creates the Wumpus worlds in which the agent will navigate. We will then implement a logic system that uses unification and resolution on first-order rules that allow the agent to navigate. The reasoning system will use the following information: the stench when near a Wumpus, the breeze when near a pit, the scream of a Wumpus as it is killed, and the wall when ran into. Additionally, a reactive agent will be created that makes decisions based upon random decisions regarding which cells are safe and which are not. We will test the logic system on Wumpus worlds of sizes 5x5, 10x10,, 25x25. Our performance metrics are as follows: number of times the gold is found, number of Wumpus killed, number of times the explorer falls into a pit, number of times the Wumpus kills the explorer, and number of cells explored.

3 Software Design

Our design will incorporate three main classes: `World`, `EnvironmentEngine`, and `Agent`. These classes have all of the important attributes and functions needed to implement this game. The `World` class is responsible for creating and maintaining worlds. The `EnvironmentEngine` is responsible for checking the move an agent selects and then telling the agent the result of the move. Finally, the `agent` is an abstract class that uses information provided by the `EnvironmentEngine` to decide what the next move is. The three implemented versions of `Agent` are `HumanAgent`, `ReactiveAgent` , and `ReasoningAgent`.

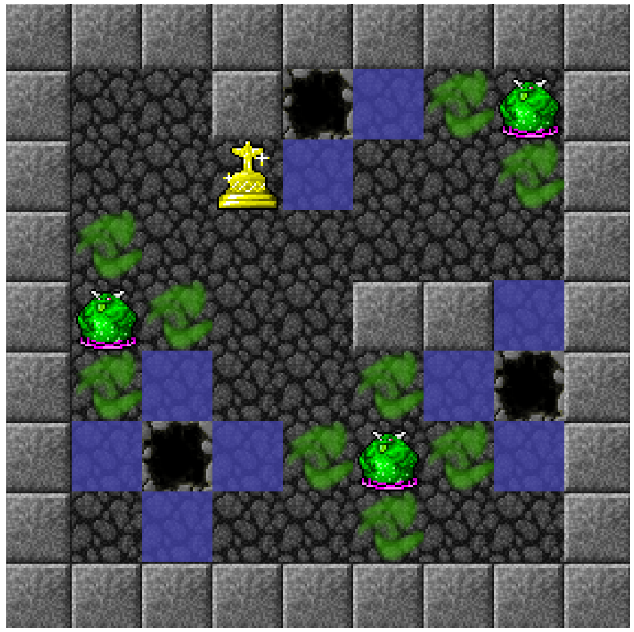

Figure 1: *An example of a wumpus world produced by our problem generator. The green haze represents stench and the blue tiles represent breezy squares.*

The implementation of the `Agent` class is very important. We chose to create an abstract class because we required three unique agents that had some of the same instance variable and methods. Each agent has an agent world variable that is blank except for the starting square initially. This world will be updated as each agent discovers things about the world and according to each agents abilities. The two methods that are shared across agents are `next_move` and `update_world`. `next_move` is used called to see the results of the previous move and returns its choice for the next move. Each agent implements this method in a unique way since each agent determines it's next move differently. The other method, `update_world`, is responsible for taking in the result of the previous move and adding it to the knowledge base. The agent can then use this updated world to make the best decision possible. Again, each agent updates the world differently and, therefore, the `update_world` method is different for each agent.

3.1 Wumpus World Generation

The `World` class is used to create both the fully populated "Master World" and the blank "Agent World". A world will consist of a vector of vectors that each hold an int. Each int represents one square in the world, with each bit representing the presence of a certain feature. We chose to represent the board this way because it seemed more simple than using a whole object for each tile. We decided to use a vector of vectors because they are much more forgiving to work with than two dimensional arrays in c++. The `World` class will also be responsible for keeping some important data such as number of wumpi and pits for general usage throughout the program.

3.2 Environment Engine

The `EnvironmnetEngine` is responsible for holding the master world object and telling the agents about the world as it is explored. It's main job is to operate a loop that constantly calls the `update_world` method in the agent. The idea is that the `update_world` returns the agents next move. The `EnvironmentEngine` then applies that move to the master world. The next time `update_world` is called the `EnvironmentEngine` includes the results of the attempted move. These results include the agents new position, the integer that

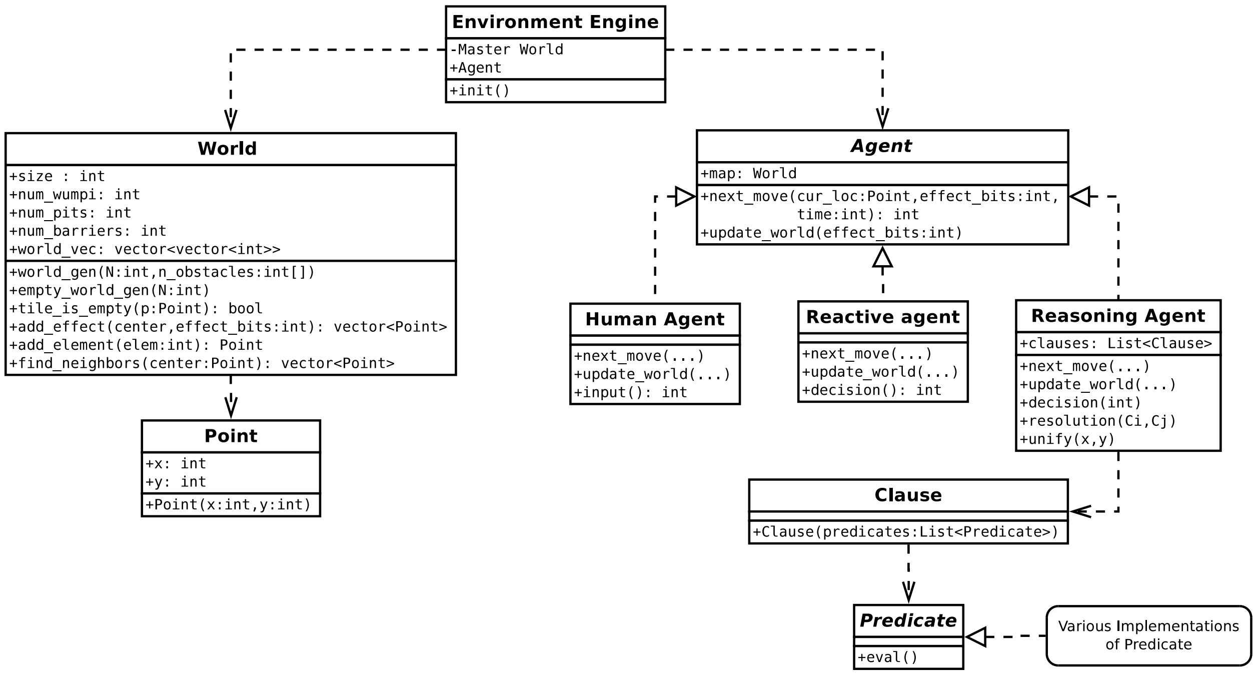

Figure 2: *A UML diagram describing the proposed layout for our program.*

represents the facts about that square, and miscellaneous facts such as whether a bump was detected. This system is a good option because the same `EnvironmnentEngine` can be used for all of the agents.

3.3 Reasoning Agent

Our reasoning agent will acquire knowledge about the world around it through its precepts, such as stench, breeze, and bump. These precepts will be inserted into the knowledge base, where the inference engine will use the precepts and logical implications to infer new information that could not be directly perceived (such as the location of the wumpus). The inference engine will only have the ability to answer specific questions, e.g. "Is there a wumpus at this square?" or "Is this square clear?", so our agent will need appropriate control logic to ensure that it queries the correct sequence of questions to the inference engine.

3.3.1 Inference Engine

Our inference engine will use the function PL-Resolution(KB, α) described by Russel and Norvig in Figure 7.12 [1] to perform *resolution*. Resolution allows the inference engine to use a knowledge base of facts to ascertain new details about the wumpus world. The function PL-Resolution accepts a knowledge base, KB and α, the query to be checked. Of course, PL-Resolution is based on propositional logic clauses, but according to Section 9.5.2 of Russel and Norvig, we can adapt it to operate using first order logic clauses if we we modify the function PL-Resolve(C_i, C_j) to find two variables complementary if one *unifies* with the negation of the other.

To check if two variables can be unified, we will use the function Unify(x, y, θ) provided in Figure 9.1 of Russel and Norvig. In the interest of efficiency, we will also implement a *predicate indexing*, described in Section 9.2.3 of Russel and Norvig[1]. The predicate index provides Unify() an efficient way to only attempt unification on sentences that are possible to be unified. We will accomplish this by constructing lists of all facts that use a certain predicate, e.g. all facts that contain the predicate $Adjacent(r, s)$ will be placed into a list. Then, if Unify() is asked to unify a sentence containing $Adjacent(u, v)$, it will only have to search the list described in the previous sentence instead of the entire knowledge base.

3.3.2 Knowledge Base

The knowledge base will be represented using an object-oriented approach. We will define a `Clause` object, that will contain a list of predicates that are assumed to be in a disjuction. The knowledge base will then be characterized by a list of `Clause`s that similarly are assumed to be in conjunction. The predicates will be represented by a class called `Predicate`, that will have an "abstract" function named `eval()`. Then, to make specific predicates, we will instantiate classes that inherit from `Predicate` and override the `eval()` function. This structure provides a way to represent logical facts in *conjunctive normal form*(CNF).

Since our knowledge base can only represent sentences in conjunctive normal form, we will need to ensure all of the rules for the wumpus world are expressed in this form. To accomplish this, we will create first-order logic rules for the wumpus world in the spirit of Section 8.3.4 in Russel and Norvig. We can then convert these rules to CNF using the process described in Section 9.5.1 in Russel and Norvig, or using *Mathematica* if it is acceptable for this project. Since all of our rules will be provided in CNF, there will be no reason to do further CNF conversion, as the resolution process produces clauses that are also in CNF.

Each call to the resolution algorithm will use the information in the knowledge base to find new facts about the wumpus world. These facts, in turn will be inserted into the knowledge base to be used in further resolution operations. As such, the knowledge base will increase in size as the explorer uncovers new information about a particular wumpus world. Great care should be taken to ensure that the knowledge base does not grow so large, or become so disorganized that it cannot be searched efficiently. A strategy known as predicate indexing has already been described in the second paragraph of Section 3.3.1 and is one example of how searching the knowledge base can be made more efficient. Another possible way to make searching the knowledge base more economical is to organize information by location. Many of the logical rules in the wumpus world use the $Adjacent(r, s)$ predicate defined in Section 8.3.4 of Russel and Norvig. This implies that the facts pertinent to the resolution process are very often in close spatial proximity, so it seems reasonable that the resolution algorithm should resolve clauses with similar spatial locations first.

3.3.3 Pathfinding

In the previous sections, we have discussed the role of the inference engine in determining whether a square is safe. We propose to also use the inference engine to inform the agent where to move next. We will provide the inference engine with a series of logical statements describing the preferences associated with each actuator e.g. "If there is an obstacle in the way, rotate to the right, otherwise move forward." Using these rules, the agent can ask questions such as "Is the best actuator move forward?", "Is the best actuator rotate to the left?", etc. In this way, our agent will be able to explore the map using only the inference engine and the knowledge base.

3.4 Reactive Agent

The next agent to be implemented is the `ReactiveAgent`. This agent can only act on the information provided by the cell the agent is currently in. In this case the method `update_world` will simply use the information returned by the `EnvironmentEngine` as the entirety of the world. This agent won't have a whole world. It only knows about the square it is currently in. The implementation of the `next_move` method will consist of some simple logical rules such as always choose a safe square if possible.

3.5 Human Agent

The human agent is the most simple and will be mostly used for debugging purposes. The `next_move` method will simply update the world and then use the result of user input to make moves. This Agent will be useful for testing our boards and making sure that everything is set up correctly. The `update_world` will apply the results of the previous move to the local agent world. This world is persistent in that old facts are remembered and it can be determined if the agent has been in a square before or not.

4 Experiment Design

To measure the performance of our agents, we will tally the following statistics: number of times the gold is found, number of wumpus killed, number of times the explorer fall into a pit, number of times the wumpus kills the explorer, and total number of cells explored. These statistics will be tracked independently, but they can also be combined into one statistic using the scoring system for the wumpus world described in Section 7.2 of Russel and Norvig. This scoring system provides point values to each of the statistics enumerated above, e.g. +1000 for finding the gold. This scoring system will provide a convenient way to measure the overall performance of our agents.

Using the performance measurement, we will then vary the parameters of the wumpus world such as the size, number of wumpi, number of pits, etc. and plot the relationship between the size of the world, number of obstacles, and the score in each of the categories described in the preceding paragraph. We will then use the curve-fitting procedures in *Mathematica* to provide an analytic function describing the performance vs size and number of obstacles. Finally, we will produce 3D plots displaying the data and the curve of best fit of performance vs. size and number of obstacles.

We would also like to develop a method to estimate the time-complexity of the inference procedure vs. size and number of obstacles. To quantify the time complexity of inference, we propose to track the number of times the resolution and unification algorithms are called. This will provide a good lower-bound on the amount of computational resources consumed by the inference process. As with the performance measurement, we will implement a curve fitting procedure in *Mathematica* to represent the time-complexity as an analytic function. This function will be plotted with the data on a 3D logarithmic plot for visualization.

References

[1] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 3rd edition, 2010.