Wumpus World
Planning in an Unknown Environment

Tylar Hoag

Contents

Abstract

The Wumpus World is a famous problem in computer science, specifically artificial intelligence. My project models a subset of this toy problem using answer set programming. This can be viewed as a puzzle solving problem and therefore also a plan generation problem. The interesting aspect of this problem is that it involves learning and reasoning. The program observes certain pieces of knowledge and infers new ones based on them. The two biggest challenges of this problem are how to handle a knowledge base that can change and ensuring that knowledge is coherent, that is the knowledge base does not give contradicting information. This problem could serve as the foundation for expansion to cover properties like risk, dynamic environments, and possibly even non-deterministic environments. A successful implementation of this soulution would have implications for the contributions ASP could make in the field of artificial intelligence.

Specification

The Wumpus World

The classic wumpus world problem includes the challenges of reasoning in dynamic, nondeterministic environments. The environment is a grid that can be traveled by moving in one of the four non-diagonal directions. There is a hero which is the agent whose goal is to find the square that contains the goal, a treasure. The obstacles are pits and the wumpus. None of the squares are known by the agent initially, they must be explored. The pit squares and the wumpus must be avoided. The only way the agent can avoid these is that the squares adjacent to obstacles have characteristics like being “breezy” or having a “smell”. Thus, reasoning is necessary to succeed. In the classic problem another possible goal could be to kill the wumpus using an arrow when its location has been determined.

My Implementation of the Wumpus Problem

My wumpus problem will be a subset of the classic problem that will limit it to being a static environment. The wumpus will not be able to move, and there is no implementation for the wumpus to be killed. The wumpus has essentially become a pit, but program just implements the wumpus and no pits. The focus will be more on the acquisition of knowledge and representing inference in ASP. The actions will simply be moving in a non-diagonal direction and the affected fluent will be the hero's location at a given time. The components of the knowledge base will be three pieces of knowledge. Safe and nearDanger will be static once they are acquired through sensing. Danger will be the dynamic knowledge that is gained through inference and can change with the addition of new pieces of knowledge. There will be one precondition which is that the player must start on a safe square, otherwise no inference could occur.

Significance

To solve the full problem would be very difficult and significant, but this sub-problem still addresses many issues. The agent in the program will be performing actions and learning, two major aspects of artificial intelligence. If the environment were dynamic the agent would have to re-plan after every step based on the newly gained knowledge. My implementation does not require this. Instead it generates a plan that models stepwise knowledge acquisition, but assumes that everything in the environment remains constant. This could be useful for programming an agent to explore an environment that is static and make basic decisions based on new findings. The program could even be used when things were dynamic if it were simply restarted after the environment changed. Although, the drawback is that none of the previous knowledge would be retained. This is not a complete system, but it does serve as an intermediary point for one.

Milestones

Board Generation

The first milestone was the completion of the board program that was used as a helper for the wumpus program. Although this was not a very difficult milestone it laid a lot of the framework for the rest of the coding. Also, it allowed for a hands-on elaboration into the problem. The board's code contains a set of constants that determine the board's size and other initial conditions. The other conditions that are set are the player's position, the location of the treasure (goal), and the location of the wumpus. All of the aforementioned items are encouraged to be set by someone who wants to test this code. Multiple wumpus and treasure positions are allowed, but only one initial position of the hero is allowed.

The following items were part of the board milestone but were later moved into the main program. The reason was so that the board was simply an input and not a helper program necessary for functionality. The domains of the variables defining the board are declared and the board is then generated. Also, the at and –at fluents are generated. Next, the adjacency predicate is defined which behaves like a boolean function. When it is invoked you give it two pairs of coordinates and if they are adjacent it will evaluate to true. Adjacency includes all squares surrounding a square and the square itself. Also, there is a define predicate for smell which uses the wumpus' position and adjacency.

Finally, a few simple constraints were added to ensure the player is only at one location initially and the precondition that the location is a safe square.

Planning in a Known Environment

The next milestone was plan generation in a known environment. At this point the program handled all of the actions and the related fluent of “at”. First, the domains for time and direction are declared. Then, there is the generation rule which creates all the possible moves and a constraint that limits the hero to one move per time unit. Next, all of the consequences of moves are defined. This includes both setting the hero’s new position with the at predicate and unsetting his old position, the –at predicate. Each consequence needs two rules per move, one for setting and one for unsetting, which leads to eight consequence rules. Next, is the frame control which is needed to illustrate the commonsense law that things will not change on their own. After the actions there were two constraints: the player can’t move out of the dungeon and the player won’t move onto a square it perceives as possibly dangerous. At that time no squares were perceived as such, that was for later reasoning. Finally, there are two more rules. One is a define predicate fAILEDaT to tag if and when the hero moved onto a wumpus square. I chose that format instead of eliminating the answer set so the output would show a failure in the set. The other rule is the goal definition.

Planning in an Unknown Environment

This phase was when all of the knowledge was handled. First the sensing of static knowledge was implemented. This was simple; a square is safe if it does not smell and nearDanger otherwise. Static knowledge also needed a time component to signify that the knowledge was not always there and had to be gained at some point. The interesting part was the dynamic knowledge of danger. Known danger knowledge was acquired by inferring that every square adjacent to a nearDanger square was dangerous, but this knowledge was overridden by the inference that if a square was adjacent to a safe square it could not be dangerous. Frame control was added for all pieces of knowledge, including those that were static so it was preserved throughout time. In the final product the answer sets returned correspond to a plan that can be executed in a given time and the final knowledge of danger.

Challenges

Representation of Danger

Knowledge representation and its use were difficult to model in this program. This is a partially observable environment which makes the problem much more complex. As stated earlier, the nearDanger and safe predicates were not as difficult to handle because they are static, but the danger predicate is not as simple. There are three instances of the danger knowledge for a given position at a given time. These are danger, strong negation danger, and negation danger; meaning known dangerous, known not dangerous, and unknown dangerous. One approach to handling this is what I would describe as cautious. All squares would be assumed to be dangerous until knowledge is gained to infer otherwise. Since the precondition states that the agent must start on a safe square then all adjacent squares will be updated as not dangerous and he will be able to begin exploring the environment. It would be nice to change this schema to include unknown dangerous as this would be a more realistic interpretation. I was successful in representing all three forms of the piece of danger knowledge, but it did give rise to coherency problems.

Coherency

The most difficult aspect of this project was handling incoherency. It was not acceptable to have two inferences that ever contradicted each other at a given time. Earlier it was stated that one would simply override another, but it was not that simple. Instead, careful steps had to be taken to ensure that inferences were only made when a contradiction would not be derived. This involved changing the safe inference, and partially introduced the time handling problem. The reason this problem gave me the greatest level of difficulty is because debugging it was exceedingly difficult. This is because Smodels automatically removes incoherent sets so it was initially difficult to determine what was causing the problem. After some clever manipulation to a few predicates I was able to determine exactly where the incoherency arose and develop a solution.

Time Handling

The next big challenge was time handling. This problem was tied strongly to incoherency. Basically, you would have situations where incoherency would occur because of the structure of time in the program. The problem was that time was in discrete units where it was difficult to separate the period in which information was sensed/inferred and when it was acted upon. Eventually a solution was found, but it is not as elegant as I would have liked because it gives the impression that the agent is given the ability to "see the future". I determined this is not the case because if it were a small change would change the answer sets, which it does not. That change would be that visiting a treasure square would infer that square is safe. Then, in a board in which the goal cannot be safely attained it would be reached (see appendix D). In retrospect, it would have been more convincing to implement this program as having each perceived time unit actually contain two separate time periods: sensing/inference time and action time.

Proving Actions Based on Acquired Knowledge

Another challenge I have had is how to prove that the environment is indeed only partially observable. This is not a client-server system so all knowledge of the environment is actually within the memory of the agent program. Originally, I planned on having unknown and known position predicates, but determined this information would be redundant and useless. My only insight into this problem was that none of the actions performed by the hero are determined by the wumpus or treasure predicates, but I'm reluctant to say that this is sufficient. Now that the program is complete I can offer this pseudo-proof by contradiction. Suppose the environment board in appendix A was fully observable. Then, in four time units there should be five paths to the goal state, but there are only three. Therefore we have a contradiction and the environment is shown to not be fully observable.

Future Work

Risk

Currently the system does not handle risk in any way. It only returns plans that can be executed without any chance of moving onto an obstacle. This presents a problem, because there are situations where the goal can be reached without moving onto an obstacle, but the program will return no possible plan. An example of this is anytime the goal is encapsulated within "smell" squares, like the board in appendix D. The drawback would be that sometimes the agent would fail and move onto a wumpus square. Regardless, it would be ideal to have the program be capable of handling risk.

One possible way to handle risk would be to use weights and weak constraints. In this implementation moving onto squares considered safe would have a cost of one. Moving onto a danger square would become a weak constraint and have a greater cost. Extra levels of inference could even be added to reason that some squares are more dangerous than others. For example, the more “smell” squares that are adjacent to a given square, the more likely it is to be the wumpus square. The advantage of this implementation is that the system could still be represented in just ASP. Another advantage would be that this would automatically give you plans with the minimal amount of time necessary, instead of specifying a time in a stepwise manner until the minimal one was found.

Another possibility for risk handling is a stepwise wrapper program that calls the ASP planner. This program would take a partial plan up to time x and the knowledge base at time x and if there were no more possible safe moves it would act. The knowledge base would be modified to make the agent believe that there is another open square. The main advantage to this implementation is that it offers greater expandability since a wrapper program would probably be utilized in a dynamic environment.

Dynamic Environment

As mentioned earlier the classic wumpus world problem involves a dynamic environment. One change could be that the player could kill the wumpus. If its location could be determined then the agent could use a shoot action and kill it. Remember there can be multiple wumpuses and the agent would probably only have arrows = wumpuses.

This would be significant because now all aspects of the knowledge base would be dynamic. Safe and nearDanger could change with time and handling all of those interactions while avoiding incoherency would be a challenge. Another significant aspect would be the arrows themselves and the shooting action, because this would be an action that relies on a limited resource.

Another change could be that the wumpus actually moves, or a goal state that moves. If the environment were still deterministic it wouldn't be terribly difficult to implement. For instance, the wumpus might patrol back and forth in a predictable manner. This alteration would have the same effect as the previous; it would make all aspects of the knowledge base dynamic. Again, this would be a significant change, but the more interesting effect would be that to avoid the wumpus the agent would need to undertake a whole new level of inference. The agent would be reasoning based on inferred knowledge and time. At this point it might become necessary to have a wrapper program. I would theorize that the best way to handle this would be to write a whole different program that makes these complex inferences while the main program remains a simple explorer. A wrapper would then mediate the knowledge through them and possibly apply algorithms that could not be implemented in logic programming.

Non-Deterministic Environment

The ultimate accomplishment with respect to this problem would be having an agent that could reason and plan in a non-deterministic environment. For example, the wumpus could move randomly around the board, or not even move at all during certain time points. The details of this implementation would be terribly complex and almost definitely require a controlling program. It would call the ASP program multiple times to re-plan after each move. At this point the program would become a game playing agent. This would be a rather complex and logical game which would be interesting to implement it in a logic language. This has already been done with classical planning, but it would be interesting to do it with conditional planning where the agent has only partial information.

Final Remarks

I have been successful in implementing a static version of the dynamic wumpus world problem. This involved issues of inference, knowledge representation, planning, and coherence. The final product is nothing amazing, but it has the potential to be expanded into a much more significant system. The system could be re-structured to handle risk and a dynamic environment. This would involve planning with uncertainty and possibly limited resources. Eventually, an agent that plans in a non-deterministic environment could likely be designed. All of the previous expansions of the program would be like cobblestones that pave the road to achieving this goal. If the goal could be reached it would be an important contribution to artificial intelligence.

References

Acknowledgement: Special thanks to Joohyung Lee. I appreciate your guidance on this project and answer set programming in general.

Tran Cao Son, Phan Huy Tu, and Chitta Baral. Planning with Sensing Actions and Incomplete Information using Logic Programming. Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, 261-274.

Appendix A: Board Code

```
%%%%%%%%%%Board basics(initial state)
%%%%%%%%%%Set these values

 %Initial board size
#const size = 3.

%Initial position of the hero, don't change 0
:- not at(1,1,0).

%initial position of treasure
treasure(3,3).

%initial position of wumpus(es), can add multiple wumpus
wumpus(3,1).
```

Appendix B: Wumpus Program Code

```
directions(right;down;left;up).
time(0..atime-1).
#domain directions(D;D2), time(T;T2).
#domain board(X;Y;I;I2;J;J2).

%%%%%%%%%%Board Characteristics%%%%%%%%%%
%Generates and defines board
board(1..size).
space(X,Y).

%Cannot be at the same square and off it at the same time
1{at(X,Y,0), -at(X,Y,0)}1.
%Cannot start in more than one square
:- at(X,Y,0), at(I,J,0), X != I | Y != J.
%Precondition: player must initially start on a safe square
:- at(X,Y,0), smells(X,Y).

%Defines adjacency on the board
adjacent(X,Y,I,J) :- abs(X-I) == 1, abs(Y-J) == 1 | abs(Y-J) == 0.
adjacent(X,Y,I,J) :- abs(X-I) == 0, abs(Y-J) == 1 | abs(Y-J) == 0.

%All squares surrounded by wumpus smell
smells(X,Y) :- space(X,Y), wumpus(I,J), adjacent(X,Y,I,J).

%%%%%%%%%%%Actions%%%%%%%%%%
%Generate action choices
```

```
{move(D,T)}.
:- move(D,T), move(D2,T), D != D2.

%Action definitions
-at(X,Y,T+1) :- -at(X+1,Y,T), at(X,Y,T), move(right,T).
-at(X,Y,T+1) :- -at(X,Y+1,T), at(X,Y,T), move(down,T).
-at(X,Y,T+1) :- -at(X-1,Y,T), at(X,Y,T), move(left,T).
-at(X,Y,T+1) :- -at(X,Y-1,T), at(X,Y,T), move(up,T).
at(X+1,Y,T+1) :- at(X,Y,T), -at(X+1,Y,T), move(right,T).
at(X,Y+1,T+1) :- at(X,Y,T), -at(X,Y+1,T), move(down,T).
at(X-1,Y,T+1) :- at(X,Y,T), -at(X-1,Y,T), move(left,T).
at(X,Y-1,T+1) :- at(X,Y,T), -at(X,Y-1,T), move(up,T).

%Frame control
at(X,Y,T+1) :- at(X,Y,T), not -at(X,Y,T+1).
-at(X,Y,T+1) :- -at(X,Y,T), not at(X,Y,T+1).

%%%%%%%%%%Constraints%%%%%%%%%%
%: can't move into a wall
:- move(right,T), at(X,Y,T), not space(X+1,Y).
:- move(down,T), at(X,Y,T), not space(X,Y+1).
:- move(left,T), at(X,Y,T), not space(X-1,Y).
:- move(up,T), at(X,Y,T), not space(X,Y-1).

%can't move onto a danger square
:- move(right,T), at(X,Y,T), danger(X+1,Y,T+1).
:- move(down,T), at(X,Y,T), danger(X,Y+1,T+1).
:- move(left,T), at(X,Y,T), danger(X-1,Y,T+1).
:- move(up,T), at(X,Y,T), danger(X,Y-1,T+1).

%%%%%%%%%%Knowledge acquisition of environment%%%%%%%%%%
%Static knowledge
safe(X,Y,T) :- at(X,Y,T), not smells(X,Y).
nearDanger(X,Y,T) :- at(X,Y,T), smells(X,Y).

%Frame Control
safe(X,Y,T+1) :- safe(X,Y,T).
nearDanger(X,Y,T+1) :- nearDanger(X,Y,T).

%%%%%%%%%%Dynamic knowledge(Danger)
%Inference
danger(X,Y,T) :- at(I,J,T), nearDanger(I,J,T), adjacent(X,Y,I,J),
        safe(I2,J2,T), not adjacent(X,Y,I2,J2).
-danger(X,Y,T) :- safe(I,J,T), adjacent(X,Y,I,J), at(I,J,T).%, adjacent(I2,J2,X,Y).

%Frame control
```

```
danger(X,Y,T+1) :- danger(X,Y,T), not -danger(X,Y,T+1).
-danger(X,Y,T+1) :- -danger(X,Y,T), not danger(X,Y,T+1).

%%%%%%%%%%%Goal definition%%%%%%%%%%%
:- not at(X,Y,atime), treasure(X,Y).
fAILEDaT(X,Y,T) :- at (X,Y,T), wumpus(X,Y).

%%%%%%%%%%%Hide and shows%%%%%%%%%%%%
hide.
show move(_,_).
fdanger(X,Y,atime) :- danger(X,Y,atime).
show fdanger(_,_,_).
show fAILEDaT(_,_,_).
```

Appendix C: Example Input/Output

```
C:\Tylar\Logic>lparse --true-negation -c atime=4 board wumpus | smodels 0

smodels version 2.26. Reading...done
Answer: 1
Stable Model: move(down,0) move(down,1) move(right,2) move(right,3)
Answer: 2
Stable Model: move(down,1) move(down,2) move(right,0) move(right,3) fdanger(3,1,
4)
Answer: 3
Stable Model: move(down,0) move(down,2) move(right,1) move(right,3) fdanger(3,1,
4)
False
Duration 0.328
Number of choice points: 2
Number of wrong choices: 2
Number of atoms: 549
Number of rules: 1949
Number of picked atoms: 127
Number of forced atoms: 24
Number of truth assignments: 2576
Size of searchspace (removed): 6 (21)
```

Appendix D: Unreachable Goal Board Code

```
%%%%%%%%%%Board basics(initial state)
%%%%%%%%%%Set these values

 %Initial board size
#const size = 3.
```

```
%Initial position of the hero, don't change 0
:- not at(1,1,0).

%initial position of treasure
treasure(3,3).

%initial position of wumpus(es), can add multiple wumpus
wumpus(2,3).
```