Navigating Wumpus World Environments Using Theorem Proving Software

Michael Anderson

December 2, 2010

Abstract

In this brief report I investigate the feasibility of using theorem proving software alongside domain-specific routines to implement an agent that, as successfully as possible, navigates toy Wumpus World environments. For brevity, it is assumed that the reader is already familiar with the Wumpus World.

The challenges of this project included the modeling of the Wumpus World and its various ideosyncracies in code, implementing an appropriate version of A* search that allowed the agent to move from one part of the world to another as quickly as possible, and the conversion of the ideosyncracies of the Wumpus World to first-order logic for the theorem prover.

Although the Prover-9 package turned out to be far too slow in practice to be incorporated into an agent for this world, I learned much about first-order logic and theorem proving software that will be useful in future projects that require such tools.

1 Implementation

1.1 High Level Overview

The implemented system contains several interacting parts, some of which will be explained in further detail. All of the code was written in Python 2.4.3.

The WumpusWorld class holds the 4×4 grid, along with the locations of the pits, the wumpus, and the gold. It is also responsible for displaying the environment to the user, and it is capable of generating random Wumpus Worlds that conform to the constraints of the domain.

The HumanAgent class gives the user the ability to take actions within a particular wumpus world, i.e. the user is the agent. The main purpose of this class is for testing, to ensure that the system responds appropriately as actions are taken by an agent.

The ComputerAgent class is the meat of the project. This class is the implementation of an agent that can be given percepts as input and asked for actions as output, and its chief responsibility is to find the most reasonable action given what it has so far perceived about a wumpus world environment.

Finally, the Simulation class acts as the controller. It iteratively gives percepts to a given agent (either human or computer), prompts it for actions, makes changes to the environment and the position of the agent based on those actions, and outputs information to the console.

1.2 Computer Agent

To interact successfully with a WW environment, the agent keeps several data members in memory. The agent is aware of its current location within the grid, the number of actions so far taken (the current time), the direction it is facing at any given moment, and the queue of actions that should be taken next if there are any (the "plan"), and the knowledge base.

KB holds a list of all of the information that the agent has accumulated about the environment it is occupying, in a first-order logic format that is consistent with Prover-9's lexicon. When the agent is created, the KB is initialized with all of the atemporal "physics" of the environment, such as:

Definition of a visited square:

$$Visited(x) \leftrightarrow \exists y \; At(Agent, x, y)$$

A square is breezy iff there is a pit that is adjacent to it:

$$Breezy(x) \leftrightarrow \exists y \; Adjacent(x, y) \wedge Pit(y)$$

A square is safe iff it contains no wumpus and no pit:

$$OK(x) \leftrightarrow \neg Pit(x) \wedge \neg At(Wumpus, x, current\_time)$$

As the agent explores the WW environment, it receives percepts that give it additional information that it can use to make inferences, such as where breezes and stenches are, whether the wumpus is still alive, and whether the agent still has an arrow to fire. All of this information is appended into the KB as it is received.

The ComputerAgent class also contains several member functions. The important ones are plan_route(), plan_shot(), ask(), tell(), and get_action().

plan_route() is responsible for finding the set of actions that move the agent from one square in the world to another. This is done by using a domain-specific implementation of A*, with the following pseudocode:

```
plan_route(self, current, facing, goals, allowed):
    root_node <- A new node with the current square and facing the current direction
    root_node.g <- 0
    root_node.f <- root_node.g + calculate_heuristic(root_node.square, goals)
    root_node.action <- NULL
    root_node.parent <- NULL
    frontier <- [root_node]
    explored <- [ ]

    loop:
        if the frontier is empty:
            return False
        n = frontier.pop()
        if n.square is one of the goals:
            solution <- [ ]
            while n has a parent:
                solution <- solution + [n.action]
                n <- n.parent
            return solution, n.square, n.facing
        explored <- explored + [n.square]
        for each action that can be taken from n:
            child <- A new node that is the same as n
            if action is forward:
                child.square <- child.square with new row/column, dependant on facing
            elif action is turn_right:
                child.facing <- child.facing turned 90 degrees clockwise
            elif action is turn_left:
                child.facing <- child.facing turned 90 degrees counter-clockwise
            if child.square is not in allowed:
                continue
            child.g <- n.g + 1
            child.f <- child.g + calculate_heuristic(child.square, goals)
            child.parent <- n
            child.action <- action
```

```
        if child.square is not in either explored or frontier:
            add child to the frontier queue, keep the queue sorted by f
        for each frontier_node in frontier:
            if the child is in the frontier with a lower f cost:
                remove the more expensive node from frontier
                add child to the frontier queue, keep the queue sorted by f
```

The heuristic used in this A* algorithm is the minimum of the Manhattan Distances to the goal squares.

plan_shot is used when the agent wants to fire the arrow at where it thinks the wumpus might be. The agent must travel to the closest that lines up a shot with where the wumpus might be, and then plan to shoot in the appropriate direction. Pseudocode for it is:

```
plan_shot(self, cur_sq, cur_facing, possible_wumpus, safe):
    goals <- the possible_wumpus squares that are also safe
    if goals is empty then return
    actions, goal_square, goal_facing = self.plan_route(cur_sq, cur_facing, goals, safe)
    actions <- actions + [some turns to face the agent in the direction of the wumpus]
    actions <- actions + [shoot]
    return actions
```

ask() takes a query, and returns true if the query can be inferred from the current KB, and false if it cannot. ask() constructs an input file with all of the current KB given as assumptions and the query given as the goal to be proved, in the format and lexicon that Prover-9 expects. ask() then executes Prover-9, and reads the contents of an output file to see if it was successful in finding a proof for the query.

tell() is a one-liner that simply adds a given list of facts to the KB.

get_action() is the top-level function of ComputerAgent, that calls all of the others. It takes a percept, and returns the best action in the given situation. It is very similar to the hybrid-wumpus-agent function given on p. 270 of Russell & Norvig. Its pseudocode is:

```
get_action(self, percept):
    tell(make_percept_sentence(percept))
    safe <- squares which ask(ok(square)) is true
    if percept.glitter
        plan <- [grab] + plan_route(current_sq, facing, starting_sq, safe) + [climb]
    if plan is empty
        unvisited <- squares such that ask(visited(square)) is false
        plan <- plan_route(cur_sq, cur_facing, squares in both unvisited and safe, safe)
    if plan is empty and ask(haveArrow,t)
        possible_wumpus <- squares for which ask(-at(Wumpus, square, cur_time)) is false
        plan <- plan_shot(cur_sq, cur_facing, possible_wumpus, safe)
    if plan is empty
```

```
        not_unsafe <- squares for which ask(-ok(square)) is false
        plan <- plan_route(cur_sq, cur_facing, sqs in unvisited and not_unsafe, safe)
    if plan is empty
        plan <- plan_route(cur_sq, cur_facing, [1,1], safe) + ['climb']
    action <- plan.pop()
    time <- time + 1
    return action
```

1.3 Simulation

The Simulation class is the top level controller of the system. Unlike the agent, it knows the entire state of the world. It takes an agent (either human or computer), feeds it percepts and then asks it for actions. Although it is actually implemented as a class, it could have easily been written as an equivalent stand-alone function with the following pseudocode:

```
simulation(WumpusWorld ww, Agent a):
    ww.display()

    loop
        percept <- the condition of the world at the agent's current time and location
        output percept to the console

        action <- a.get_action(percept)

        if action is forward
            if the agent is not facing a wall
                move the agent forward
            else:
                make next percept have a bump
            if the agent has moves into a pit or wumpus then end the simulation
        elif action is turn left
            turn the agent 90 degrees counter-clockwise
        elif action is turn right
            turn the agent 90 degrees clockwise
        elif action is grab
            agent gets the gold if there is any
        elif action is shoot
            if the agent has an arrow
                if the shot is lined up with the wumpus
                    remove the wumpus from the world
                    make next percept have a scream
        elif action is climb
            if agent.location is [1,1] then end the simulation
        else:
            output 'INVALID ACTION TAKEN' to the console
```

```
    end simulation

output last action taken to the console
pause until the user presses a key
```

2 Experimental Setup

To experiment with various WW environments, it was also necessary to create a couple more minor stand-alone functions. generate_worlds() was responsible for creating 100 random WW environments that conformed to the Wumpus World constraints, and then outputting them to a worlds.txt file. In this way, various experiments could be run on a single set of unchanging wumpus worlds, to reduce the effects of random chance. read_worlds() is called at the beginning of every run of the program, and this function opens world.txt, and reads in its contents into an array of WumpusWorld instances. Once that occurs, the Simulation class can be instantiated and given any of those instances to work in.

3 Results

Unfortunately, the results of running the simulation on various wumpus worlds was not very successful. Each time the simulation gets an action from the ComputerAgent instance, ask() must be called dozens of times. However, even the initial KB is so large that Prover-9 cannot work quickly enough to make this feasible. After several minutes of working on even a single query, my computer was not able to complete the search for a proof. I do know that Prover-9 was actually working, and not just glitching out because I entered information incorrectly, because it outputs its progress as it performs its search, and it displays an error if it receives a statement in a format it cannot understand. Furthermore, Prover-9 worked quickly and correctly with simpler test problems.

One inefficiency in my knowledge representation was that each square had its own constant variable, as did each time. This is because Prover-9 seemed to be unable to handle integer addition and subtraction except under highly restricted circumstances. This made the representation of adjacency rather large, because each pair of adjacent squares needed their own individual statement. If squares could have been represented by one variable instead of sixteen, perhaps the theorem prover would have run much faster. The number of statements to be processed seemed to be simply much too large.

I also attempted to use the Vampire theorem prover, but was frustrated by the lack of documentation available online. Ultimately, the few examples that were given were not sufficient to demonstrate the functionality I wished to use.

4 Conclusion

Although the experiment failed, I believe that the system was generally well-designed. The Simulation and WumpusWorld classes worked correctly and the environment was accurately represented, confirmed by testing with a Human-Agent instance. The hybrid-wumpus-agent algorithm in the book made sense and it was fairly straightforward to implement my own version of it. Representing the characteristics of the wumpus world with first-order logic took some thought, the main concern was making sure that my representation was complete and gave the agent the knowledge base needed to make all of the inferences possible in a given situation. Once I decided on the knowledge that needed to be represented, converting that knowledge to first-order logic statements was not too difficult.

As can often be expected, the most difficult part of the project was learning how to use new pieces of software, i.e. the theorem provers. With more time, it is possible I would have figured out how to fully use Vampire and had more luck with it, or found some sort of workaround within the mess that is Prover-9's documentation to the problem of not being able to use integer addition and subtraction. There also may have been other theorem provers out there that were worth trying.