CS531 Programming Assignment 4: Wumpus Agent

Michael Lam, Xu Hu
EECS, Oregon State University

November 28, 2012

Abstract

In this assignment we design, implement and evaluate an algorithm that uses first-order logic and A* search for an agent in order to solve Wumpus puzzles.

1 Introduction

The Wumpus world is a 4x4 grid containing pits, one Wumpus and one gold at various locations. The objective of the agent is to retrieve the gold without dying from the Wumpus or falling into a pit. Furthermore the agent can perceive its environment and infer the location of pits and the Wumpus due to the rules of constructing a Wumpus world. Therefore it makes sense to implement an algorithm involving logic and search to make intelligent decisions.

We used the existing Wumpus environment simulator for Python provided by Walker Orr. We designed an agent that uses first-order logic with a tell-ask interface to assert/query what it knows about the Wumpus world and A* search to plan routes around the Wumpus world. The algorithm is essentially the same as the one provided in the Russell-Norvig textbook (pg. 270, fig. 7.20). For answering logic queries, the program uses the Prover-9 program.

To evaluate our algorithm, we designed some experiments and collected statistics.

2 Approaches

In this section we describe our implementation of the agent and discuss our design decisions.

2.1 Simulator

We used the simulator for Python provided by Walker Orr. While the simulator represented the Wumpus world fairly well as described in the textbook, it was still incomplete. First of all, the simulator lacked the notion of facing directions as well as the percepts bump and scream. In addition, the simulator would declare reaching the gold a success, eliminating the need for the agent to travel back and climb at the initial square. This also differs from the textbook's formulation. Therefore, we modified the simulator to add these remaining percepts, fixed the simulator to keep track of facing direction and allowed the agent to return to the initial square to climb. The modified simulator should represent the Wumpus world as described in the textbook.

2.2 Knowledge Base

We designed our Knowledge Base as follows:

- Atemporal

  B(x,y) Breezy at location (x,y)

  P(x,y) Pit at location (x,y)

  S(x,y) Smelly at location (x,y)

  W(x,y) Wumpus at location (x,y)

- Temporal

  WumpusAlive(t) Wumpus is alive at time t

  HaveGold(t) Agent has gold at time t

  HaveArrow(t) Agent has arrow at time t

- Both

  Loc(x,y,t) Agent was at location (x,y) at time t

OK(x,y,t) Square (x,y) at time t was safe to visit

- Perception

  Breeze(t) Breeze perceived at time t

  Stench(t) Stench perceived at time t

  Glitter(t) Glitter perceived at time t

  Bump(t) Bump perceived at time t

  Scream(t) Scream perceived at time t

- Action

  Forward(t) Forward action at time t

  TurnLeft(t) TurnLeft action at time t

  TurnRight(t) TurnRight action at time t

  Shoot(t) Shoot action at time t

  Grab(t) Grab action at time t

  Climb(t) Climb action at time t

We wrote out some rules that Prover-9 could use to prove a query. Note that in our implementation, we explicitly fill for each x, y and t instead. We know the size of the world is 4x4 so we are able to enumerate all x and y. For each time step, we enumerate all x and y for time t. The reason for enumerating out all possibilities instead of using variables is to keep the logic relatively simple and avoid having to implement arithmetic in Prover-9.

- $B(x, y) \Leftrightarrow P(x1, y1) \vee P(x2, y2) \vee P(x3, y3) \vee P(x4, y4)$
- $S(x, y) \Leftrightarrow W(x1, y1) \vee W(x2, y2) \vee W(x3, y3) \vee W(x4, y4)$
- $Loc(x, y, t) \Rightarrow (Breeze(t) \Leftrightarrow B(x, y))$
- $Loc(x, y, t) \Rightarrow \neg P(x, y)$
- $Loc(x, y, t) \Rightarrow (Stench(t) \Leftrightarrow S(x, y))$
- $Loc(x, y, t) \Rightarrow (\neg W(x, y)) \vee (W(x, y) \wedge \neg WumpusAlive(t))$
- $OK(x, y, t) \Leftrightarrow \neg P(x, y) \wedge \neg (W(x, y) \wedge WumpusAlive(t))$

Other than these fundamental rules, the rest are essentially assertions of percepts, actions and derived atemporal facts of the Wumpus world at every time step. These information should be sufficient for the agent to reason about the world and make good decisions.

2.3 Algorithm

We implemented the algorithm in the textbook (pg. 270, fig. 7.20). It is mostly the same in terms of the decision-making process. There are a few technical differences regarding the Knowledge Base representation.

We asserted all the facts at the beginning of the algorithm. The Make-World-Logic-Sentences function constructs sentences that assert conditional sentences of the Wumpus world at the current time. This includes sentences relating "OK" to "Pit" and "Wumpus" as well as explicitly writing out the sentences relating adjacent squares such as relating "Breeze" to "Pit." It is important to note that some of these sentences are temporal, so we create them at every new time step. This is to avoid adding arithmetic logic to Prover-9. Another note is that the adjacent squares are written out explicitly. Again this is to avoid using arithmetic as a design choice.

One final technical note is that every call to Ask also caches the query if the query is proven true. This is to improve performance.

Algorithm 1 Hybrid-Wumpus-Agent

```
HYBRID-WUMPUS-AGENT(percept)
Inputs : "percept" list
Persistent : KnowledgeBase "KB", time "t", actionsequence "plan"
Returns : singlenext "action"
Tell(KB, Make-World-Logic-Sentences(t))
Tell(KB, Make-Percept-Sentence(percept, t))
Tell(KB, Make-Location-Safe-Sentence(current))
Tell(KB, Make-Have-Arrow-Sentence())
Tell(KB, Make-Have-Gold-Sentence())
safe := {(x,y) : Ask(KB, OK(x,y,t) = TRUE}
if Ask(KB, Glitter(t)) = TRUE then
   plan := [Grab] + Plan-Route(current, {(0,0)}, safe) + [Climb]
end if
if plan is empty then
   unvisited := {(x,y) : ASK(KB, Loc(x,y,t')) = FALSE for all t' <= t}
   plan := Plan-Route(current, unvisited ∩ safe, safe)
end if
if plan is empty AND Ask(KB, HaveArrow(t)) = TRUE then
   possible_wumpus := {(x,y) : Ask(KB, -W(x,y)) = FALSE}
   plan := Plan-Shot(current, possible_wumpus, safe)
end if
if plan is empty then
   not_unsafe := {(x,y) : Ask(KB, -OK(x,y,t)) = FALSE}
   plan := Plan-Route(current, unvisited ∩ not_unsafe, safe)
end if
if plan is empty then
   plan := Plan-Route(current, {(0,0)}, safe) + [Climb]
end if
action := Pop(plan)
Tell(KB, Make-Action-Sentence(action, t))
t := t+1
return action
```

2.4 A* Search

In the Plan-Route function we formulate an A* search problem given the agent's current location, goal locations and allowable squares. The A* algorithm is essentially the same as in the previous assignment with a different heuristic.

Algorithm 2 A* Search

```
exploredSet = ∅
frontier = [initialPath]
while number(explored) < NMAX do
  if frontier == ∅ then
    return FALSE
  end if
  path = frontier.pop()
  state = path[0]
  exploredSet.add(state)
  if state == goalState then
    return path
  end if
  for action in state.validActions() do
    for newState in action.results() do
      newPath = path + newState
      if ismember(frontier,newPath) == FALSE then
        frontier.push(newPath)
      end if
    end for
  end for
end while
```

3 Experiments

4 Discussion

Discussion here.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12

Table 1: Caption here.