CS531 Programming Assignment 4: Wumpus Agent

Michael Lam, Xu Hu
EECS, Oregon State University

November 29, 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 Hybird Agent

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 Route Planning

In the Plan-Route function we formulate an search problem given the agent's current location, goal locations and allowable squares. We implement the A* search and RBFS search which are essentially the same as in the previous assignment with a different heuristic. The state space is defined by position and direction, that is a tuple $(x, y, direction)$. Since the agent may have multiple goals, the heuristic is the minimum city block distance from current square to one of goals. When the path is found, we simply back track the squares on the path and return a sequence of actions, which will be combined with other actions in hybird agent aforementioned.

3 Experiments

We test out agent on 40 randomly generated maps and use A* search and RBFS search respectively. For the running time consideration, we only test with 4×4 maps. All other condtions are the same including heurisitc and logic engine. We record 5 targets: returning successfully, grabing gold, killing wumpus, number of steps and points of reward. For rewards, we set every action -1, there are 500 for grabing the gold, 100 for killing wumpus and 200 for return back and climb out the cave. The details for each experiment are displayed in table 1. Note that there are many unsolvable maps. For A* search, the agent finished 17 games, got 17 golds and killed the wumpus 27 times. The average steps and rewards for A* search are 119.75 and 210.25. The RBFS search couldn't finish some of searches because we set a relative small number of allowed expanded nodes. It completed 10 games, got 17 golds, killed wumpus 27 times. The average steps for RBFS search is 96.55, while the average reward is 198.45. From these, we can see that the performances of two search algorithms are similar. A* search is slightly better.

4 Discussion

Discussion here.

A*	Index	1	2	3	4	5	6	7	8	9	10
	Success	1	0	0	1	0	0	0	0	0	0
	Gold	1	0	0	1	0	0	0	0	0	0
	Kill	0	0	1	1	0	0	0	1	0	0
	Steps	241	12	14	251	329	315	230	60	175	192
	Reward	559	88	-14	449	-229	-215	-130	-60	-75	-92
RBFS	Index	1	2	3	4	5	6	7	8	9	10
	Success	0	0	0	0	1	1	0	1	0	1
	Gold	0	0	0	0	1	1	0	1	0	1
	Kill	0	1	1	1	1	1	1	1	0	1
	Steps	205	60	67	7	99	98	7	39	211	146
	Reward	-105	-60	-67	-7	601	602	-7	661	-111	554
A*	Index	11	12	13	14	15	16	17	18	19	20
	Success	0	0	1	1	1	0	0	1	1	1
	Gold	0	0	1	1	1	0	0	1	1	1
	Kill	0	1	1	1	1	1	1	1	1	1
	Steps	65	7	99	39	99	90	7	162	77	235
	Reward	35	-7	601	661	601	-90	-7	538	623	465
RBFS	Index	11	12	13	14	15	16	17	18	19	20
	Success	0	0	1	0	0	0	1	0	0	1
	Gold	0	1	1	0	1	0	1	0	0	1
	Kill	0	1	1	1	1	1	1	1	1	1
	Steps	19	168	130	14	174	7	122	60	161	99
	Reward	81	332	570	-14	326	-7	578	-60	-61	601
A*	Index	21	22	23	24	25	26	27	28	29	30
	Success	1	0	0	1	1	0	1	0	0	0
	Gold	1	0	0	1	1	0	1	0	0	0
	Kill	1	1	1	1	1	0	0	1	1	0
	Steps	99	7	7	175	122	19	271	67	7	206
	Reward	601	-7	-7	525	578	81	529	-67	-7	-106
RBFS	Index	21	22	23	24	25	26	27	28	29	30
	Success	0	0	0	0	1	0	0	0	0	1
	Gold	0	1	0	0	1	1	0	0	0	1
	Kill	0	1	0	1	1	1	0	1	0	1
	Steps	12	159	138	90	39	152	72	7	65	77
	Reward	88	341	-38	-90	661	348	28	-7	35	623
A*	Index	31	32	33	34	35	36	37	38	39	40
	Success	1	1	0	0	0	0	1	1	1	0
	Gold	1	1	0	0	0	0	1	1	1	0
	Kill	1	1	0	1	1	1	1	1	1	0
	Steps	235	98	72	60	7	60	39	174	294	72
	Reward	465	602	28	-60	-7	-60	661	526	406	28
RBFS	Index	31	32	33	34	35	36	37	38	39	40
	Success	0	0	1	0	0	0	0	0	0	0
	Gold	0	0	1	0	1	1	0	0	1	0
	Kill	0	0	1	1	1	0	1	0	0	1
	Steps	260	214	99	77	158	193	60	72	88	7
	Reward	-160	-114	601	-7	342	407	-60	28	512	-7

Table 1: Experiment Results.