Low Cost Experiments in Cognitive Robotics for Planning in Hostile Environments with Incomplete Information

Felipe W. Trevizan *, Leliane N. de Barros, and Flávio S. Corrêa da Silva

Institute of Mathematics, University of São Paulo
Rua do Matão, 1010, Cidade Universitária – 05508-090 São Paulo, SP, Brasil
{trevisan, leliane, fcs}@ime.usp.br

Abstract. Cognitive robotics is the research field at the confluence of Artificial Intelligence and robotics. Its goal is to program robotic agents using explicitly only high-level actions and relations among actions characterized as formal logical statements. The advantages of cognitive robotics when compared with traditional robot programming are (i) the possibility of formal verification of expected properties and behaviors of robotic agents, and (ii) the capability to program robots for highly complex tasks in information-rich environments, including tasks and environments that require communication, coordination and cooperation among robots.
A common barrier to work with cognitive robotics – and autonomous robots in general – is the high cost of robots. Low cost platforms such as Lego® MindStorms™ are usually considered useful only for teaching applications, and not for research prototyping and experimentation. In the present article we show how a Lego® MindStorms™ robot can be used to implement and run an experiment in which several Artificial Intelligence techniques are required.

1 Introduction

The design and implementation of agents for dynamic environments with incomplete information and non-deterministic actions is among the most interesting challenges for autonomous robots. Many practical applications can be modelled this way, e.g. search and rescue problems.

One of the most promising approach to solve this class of problems is the cognitive robotics [1,2,3] – the research field at the confluence of Artificial Intelligence and robotics. Its goal is to program robotics agents using explicitly only high-level actions and relations among actions characterized as formal logical statements. The advantages of cognitive robotics when compared with traditional robot programming are the possibility of formal verification of expected properties and behaviors of autonomous robots, and the capability to program robots for highly complex tasks in information-rich environments, including tasks with incomplete information, non-deterministic actions and environments that require communication, coordination and cooperation among robots.

* Funded by Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP)

Many applications in cognitive robotics use Golog [1], a language based on the Situation Calculus [4] which can be easily implemented as a Prolog meta-program and can be used for the specification of control programs for robotic agents. Golog has been successfully used to experiment with novel logical reasoning theories [2,5], but the obtained results have rarely been tested or used for robotic agents in complex physical environments [3,2].

A common barrier to work with cognitive robotics – and autonomous robots in general – is the high cost of robots. Low cost platforms such as Lego® MindStorms™ are usually considered useful only for teaching applications, and not for research prototyping and experimentation. In the present article we show how a robotic agent specified in Golog can be implemented to control a Lego® MindStorms™ robot. The implementation is done in Legolog [3], which extends Golog specifically for the Lego® MindStorms™ robots. Since Lego® MindStorms™ robots have limited computational resources, we must build a controlled and simplified physical world if we want to use these robots for experiments. In our case, we analyze the use of Legolog to program Lego® MindStorms™ robots for search and rescue operations. Essentially, search and rescue requires that a robotic agent navigates along a hostile environment with access only to incomplete information about the world.

Our simplified world for search and rescue operations conforms with the Wumpus World [6], which requires that agents are programmed using a composition of several Artificial Intelligence techniques, such as reactive planning, goal based planning, task execution and monitoring, reasoning with incomplete information, generation and discrimination of hypotheses about the world.

In order to make this article self-contained, in Section 2 we briefly describe the language Golog and the Situation Calculus, and in Section 3 we introduce the Lego® MindStorms™ robots , as well as the language Legolog. In Section 4 we present the construction phases of a Legolog agent for the Wumpus World: (1) Situation Calculus specification of the reasoning features of the agent; (2) Golog implementation; (3) Legolog implementation in the Lego® MindStorms™ robot; and (4) Physical representation of the Wumpus World.

2 Golog: A Language for Cognitive Robotics

Planning can be defined as the problem of finding a sequence of actions to achieve a desired state of the world (*goal state*) or behaviors (*goal task*). This usually amounts to computationally intractable problems, since the search space is proportional to $n_a^{|\text{plan}|}$, in which n_a is the number of possible actions and |plan| is the length of the smallest sequence of actions that archive the goal state from the initial state (solution plan). In order to make this search space smaller, classical planning algorithms employ (1) conflict resolution techniques for actions, which typically transform state space search into plan state search [7]; (2) heuristic methods [8,9] to guide the state space search; or (3) compound tasks, which define constraints on actions compositions, *through task networks*, also called *hierarchical task network planning* (HTN) [10].

Golog [1] is a programming language for intelligent agents through which we can specify constraints on actions compositions, thus pruning the search space. The constraints are specified in a *high-level program*, which can be defined as a program comprised by (1) primitive instructions, which are actions the agent can execute in the environment, described as Situation Calculus statements [4]; (2) tests, which make use of domain dependent predicates that are affected by actions; (3) procedures, which correspond to compound actions as in HTN [11]; and (4) non-deterministic choices, which allow lookahead in sets of actions to select what to add to the solution plan.

Instead of looking for a sequence of actions to achieve a goal, in Golog we look for a sequence of actions to generate a valid execution of the high-level program that implements the agent, i.e., a valid decomposition of the high level program, resulting in the desired behavior of the robot. The main difference between Golog and HTN planning is that in Golog it is still possible to reason about actions and the effects of actions.

2.1 The Situation Calculus

Golog is based on the Situation Calculus[4]: a logical formalism based on First Order Predicate Logics (FOPL). Its ontology includes *situations*, which are snapshots of the world; *fluents*, which represent world properties; and *actions*, which are capable of altering the truth value of fluents . In the Situation Calculus, the constant s_0 denotes the *initial situation*; the function $do(\alpha, \sigma)$ denotes the *resulting situation* after performing the action α in situation σ; the predicate $poss(\alpha, \sigma)$ represents that action α can be executed in situation σ; and the predicate $holds(\phi, \sigma)$ represents that fluent ϕ is true in situation σ. Section 4.2 shows an example of Situation Calculus axioms for the Wumpus World.

Given a specification of a planning domain as a Situation Calculus axiomatization, the solution plan can be found through theorem proving in FOPL. Let $\mathcal{A}$ be the set of axioms that characterize the actions of an agent, $\mathcal{I}$ the set of axioms that characterize the initial situation and $\mathcal{G}$ a logical statement that characterizes the agent's goal. The constructive proof of

$$\mathcal{A} \wedge \mathcal{I} \models (\exists S).legal(S) \wedge \mathcal{G}(S), \qquad \text{where}$$
$$legal(S) \equiv poss(\alpha_1, s_0) \wedge \ldots \wedge poss(\alpha_n, do(\alpha_{n-1}, do(\ldots, do(\alpha_1, s_0))\ldots)\ldots),$$

generates an instance of the variable S as the term $do(\alpha_n, do(\ldots, do(\alpha_1, s_0))\ldots)$, which corresponds to the sequence of actions $\langle \alpha_1, \ldots, \alpha_n \rangle$, that when executed by the agent from the initial situation s_0, takes it to the goal situation.

2.2 The Golog Meta-interpreter

Golog programs are executed by a specialized theorem prover in FOPL [1]. The user must provide an axiomatization $\mathcal{A}$, describing the actions of an agent (*declarative knowledge*), and a control program c, specifying the desired behavior of the agent (*procedural knowledge*). The execution of the Golog program corresponds to the proof that $\mathcal{A} \models exec(c, s_0, \sigma)$, where $exec(c, s_0, \sigma) \equiv (\exists \sigma).\sigma \wedge legal(\sigma)$ and $\sigma = do(\alpha_n, do(\ldots, do(\alpha_1, s_0))\ldots)$ is a decomposition of c.

Another characteristic of Golog is that it performs *off-line* planning, that is, Golog searches for a sequence of actions that is a valid execution of a high-level program *before* any action is actually executed by the agent. In order to solve problems that require the execution of actions *during* planning, namely *on-line* planning, the language Indigolog [12] was created. Using Indigolog we can specify programs that perform sensing and execution of actions, while search for a solution plan. However, Indigolog is not capable to combine the *off-line* and *on-line* planning, and a solution to this problem is partially done in section 4.3.

3 The Lego® MindStorms™ Robot

The Lego® MindStorms™ robot main component is the **RCX brick** (RCX stands for *Robotic Commander Explorer*). It contains a Hitachi H8/3297 16 bits microprocessor, capable of controlling up to three actuators and three sensors simultaneously. The actuators are motors with possible selection of five rotation speeds for both directions, and the sensors can be light sensors, and touch sensors. The RCX also has an infrared port that can communicate with an *infrared tower*, which can be connected to a desktop's serial port, enabling the communication between the robot and the computer. All of these features can be programmed by the robot designer, e.g., in the language NQC (*not-quite C*), which was used in this work.

3.1 Legolog

Legolog [3] is a software package that includes the Indigolog meta-interpreter, and the implementation of a communication protocol between the RCX brick and the desktop. This protocol enables the exchange of messages during the execution of a program stored in the RCX brick.

Legolog can be used to model client-server applications, in which the RCX is the server of actuators and sensors, and the desktop (executing the Indigolog meta-interpreter) is the client. In Figure 1 we show how a client/server architecture breaks a Legolog agent in modules, where (1) **Client** is the module executed in the desktop. It contains the Indigolog meta-interpreter and the *agent reasoning program*; (2) **Server** is the module executed in the RCX brick that contains the *actions execution program*.

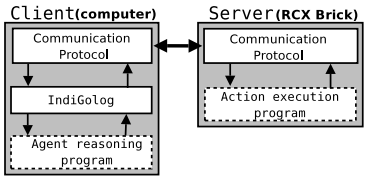

Fig. 1. Client/server model for the Legolog package, in which dashed rectangles are defined by the programmer.

A program to control a robot in Legolog, therefore, is comprised by two main parts (dashed rectangles in Figure 1):

Agent reasoning program. When executed by the Indigolog meta-interpreter, in the desktop, it performs incremental *on-line* generation of plans composed of primitive actions, taking into account the robot perceptions.

Actions execution program. Implemented in NQC and stored in the RCX. It specifies how primitive actions and perceptions are executed in the robot.

Using this model, the robot designer can define the appropriate level of abstraction of actions delegated to the RCX, i.e. the degree of autonomy of the robot. The RCX can be programmed simply to control inputs and outputs; or it can be programmed to implement more complex actions, e.g. follow a line, find an object, or even to perform a more complex task detailed in Section 4.5.

4 Case Study: A Legolog Agent for the Wumpus World

In this section we present the main steps to build a Legolog agent for the Lego® MindStorms™ robot. The concrete example we pick is the Wumpus World problem [6], which is a static domain with incomplete information that requires planning and execution of actions, as well as the formulation of hypotheses about the world. The Wumpus World is not only a theoretical exercise. As will become clear when we summarize the description of this problem, it can be envisaged as a simplified model of a search and rescue scenario.

4.1 The Wumpus World: an agent searches for a treasure in a hostile environment

The Wumpus World problem contains an agent that must explore a square grid, having information only about the neighborhood of the square in which it is located. The goal of the agent is to explore the grid, which is surrounded by walls, collecting the highest possible score on the way. The agent can increase its score by collecting coins, which are spread in various squares, performing the least number of movements. The agent must also avoid pits, and a wandering agent-devouring monster called Wumpus.

In Figure 2 we show a 4×4 instance of the Wumpus World in which the grid has already been successfully explored. The solid line represents the path to explore the environment and the dashed lines represent planned paths to kill the Wumpus or to get out of the grid. Squares containing horizontal straight lines represent *breeze*, and vertical curved lines represent the *smell* of Wumpus. This map characterizes an instance of the problem with high difficulty for the agent, requiring 41 actions to reach the solution state. The arrow indicates that the Wumpus was killed.

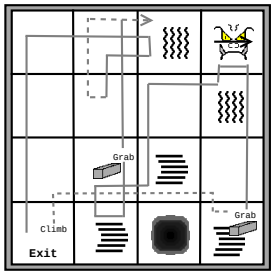

Fig. 2. A solved instance of the Wumpus World.

An agent for the Wumpus World has incomplete information about the world, since it can only sense the Wumpus (by feeling its smell) or a pit (by sensing a breeze) when it is in a neighboring square to the Wumpus or a pit. Therefore, the agent must perform a hypothetical reasoning about the world while exploring the environment, in order to classify the squares as safe or dangerous. The agent can also kill the Wumpus using an arrow.

The Wumpus World is used in introductory courses of Artificial Intelligence. Nevertheless, the design, programming and implementation of a complete program for an agent in the Wumpus World is no trivial task. In general, the Wumpus World problem is not completely solved in introductory courses. In [6], for

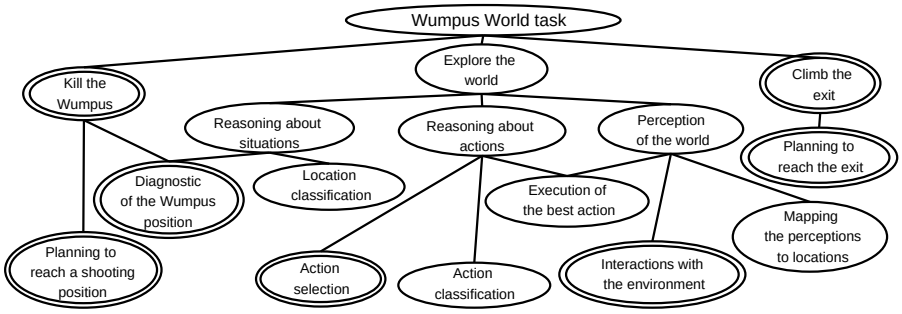

Fig. 3. Task decomposition for an agent in the Wumpus World.

example, only a few suggestions are presented on how to model the set of tasks that the agent executes (ovals in Figure 3). Moreover, that book only hints that the logical specification of the solution of the Wumpus World problem can be implemented as a Prolog program. The construction of such program, however, can be quite complicated without resorting to a language like Indigolog.

4.2 Logical specification of the agent

The specification in Situation Calculus of the agent for the Wumpus World was based initially on [6]. Figure 3 shows the task decomposition for the agent implemented in this world. Double ovals represent sub-tasks modelled and implemented specifically in this project.

The fluents in the Wumpus World are: **smelly(L)** or **breeze(L)**, which mean that position L is smelly and breezy; **atAgent(L)**, means that the agent is at position L; **agentDirection(D)**, means that D is the current direction of the agent; **visited(L)** and **secure(L)**, which denote that position L has been visited or is safe; and **holding(O)**, denotes that the agent is holding object O.

Using these fluents, we can write a set of axioms in the Situation Calculus to specify the agent. These axioms are divided into: (1) *initial state axioms*, which describe the initial state of the world; (2) *successor state axioms*, which represent how the fluents are changed or remain unchanged after the actions. For example, the successor state axiom for the fluent **smelly(L)** is:

```
holds(smelly(L), do(A, S)) :- holds(smelly(L), S);
    A==forward, stench(do(A, S)), holds(atAgent(L), do(A, S)).
```

It defines a position **L** as *smelly* in situation **do(A,S)** if **L** was already *smelly* in situation **S** or if the agent sensed the smell of Wumpus when reaching it using action **A**. Similarly, additional axioms must be defined describing the effects of all remaining agent's actions, such as: **turn**,**forward**, **grab**, **shoot**, and **climb**.

4.3 Procedures in Indigolog: the agent reasoning programming

Besides the above axioms, we must also specify the following compound tasks:

Actions classification The set of (primitive or composite) actions that can be executed from a given situation are classified as **Great**, **Good**, **Medium**, **Risky** and **Deadly**. This classification is implemented by procedures in Indigolog and its modification implies in different behaviors of the agent. For example:

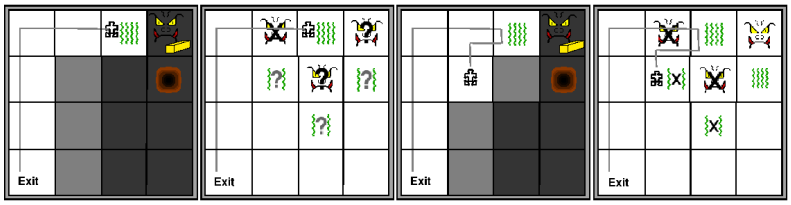

Fig. 4. Four maps representing two situations in the Wumpus World. The first and the third maps show what the agent already knows about the environment: white squares represent visited positions; black squares represent not visited positions; and grey squares represent not visited positions which were inferred to be safe. The second map shows the agent's hypothetical reasoning about the first map, and the fourth map shows the same about the third map. Squares labelled with "?" denote hypothesis; squares labelled with "X" denote a false hypothesis.

```
proc(greatAction,
  if(holding(gold), planning([0,0], [climb | []]),
       [sense(glitter), grab] # [tryToKillWumpus])).
```

This defines an agent whose action with highest priority (**greatAction**) is to plan the way out of the grid if it has already found coins. Otherwise, the agent must collect coins whenever it senses its shine. Finally, if none of the previous actions is possible, the agent must try to kill the Wumpus.

Diagnosis of the Wumpus position An interesting feature of the Wumpus World problem is that it requires the determination of the Wumpus position based on incomplete information. This task, illustrated in Figure 4, is named diagnosis since the identification of the Wumpus position can *explain* the observations (smelly positions) of the agent in previous situations. Knowing the position of the Wumpus gives to the agent the capability of killing it using an arrow. The Golog procedure **tryToKillWumpus**:

```
proc(tryToKillWumpus,
  [?(holding(arrow)), consultKB(smellyPositions(SmellyPos)),
   startSet(Wpos), findWumpus(SmellyPos, Wpos), planKillWumpus(Wpos)]).
```

specifies that in order to kill the Wumpus the agent must consult its knowledge base (**consultKB**) and retrieve a list of positions in which the smell of the Wumpus was sensed. Then the procedure **findWumpus** is triggered to generate and discriminate a list of hypotheses about the possible positions of the Wumpus (**Wpos**). If the list contains a single position, then the procedure **planKillWumpus(Wpos)** leads the agent to a plan to reach an adequate position to shoot an arrow and kill the Wumpus. The agent has only one arrow, and should not risk to waste it before knowing for sure where the Wumpus is.

The Golog procedure **findWumpus** below shows how the generation and discrimination of hypotheses about the Wumpus position is done.

```
proc(findWumpus([[SmeX, SmeY] | SmePos], Wpos),
  [abductWumpusAt([SmeX, SmeY+1], south, Wpos),
   abductWumpusAt([SmeX, SmeY-1], north, Wpos),
   abductWumpusAt([SmeX+1, SmeY], west, Wpos),
   abductWumpusAt([SmeX-1, SmeY], east, Wpos),
   if(SmePos = [], [cut(findWumpus(SmePos, Wpos))], [endSet(Wpos)])]).
```

In this procedure, for each position **[SmeX, SmeY]** in which the smell of the Wumpus was sensed, the four adjacent positions are considered hypotheses of localizations of the Wumpus, since the Wumpus can be sensed only when it is adjacent the agent. The generated hypotheses are then discriminated by the procedure **abductWumpusAt**:

```
proc(abductWumpusAt([WumX, WumY], IgnDir, Wpos), [
  if(secure([WumX, WumY]), [], if(wall([WumX, WumY]), [], [
    possibleWumpusPos([WumX, WumY], IgnDir), addToSet(Wpos, [WumX,WumY])] # [?(inCave)])])).
```

Initially, it is checked whether the position **[WumpusX, WumpusY]** has already been classified as safe or wall, otherwise the procedure **possibleWumpusPos** analyzes the three adjacent positions, excluding the position in the already visited direction *IgnDir*. This is done by verifying the hypothesis of the Wumpus being at position **[WumX, WumY]** through the analysis of the past perceptions in order to detect conflicts , i.e., if an adjacent position to the previously position **[WumX, WumY]** was visited and has not been labelled as *smelly*.

Planning to find the way out and the Wumpus The agent must, in certain situations, plan to reach a goal state crossing only safe positions, without sensing the world (*off-line* planning). This occurs in two occasions: when the agent decides to (1) **kill the Wumpus** and (2) to **climb the exit**. For (1) it must find the closest position in the direction of the Wumpus, and for decision (2) it must find the shortest path to the exit. An example of *off-line* planning occurs in the procedure described in Section 4.3, in which the command **planning([0,0], [climb | []])** is executed (decomposed) by the Indigolog meta-interpreter.

The planning algorithm was implemented in Prolog and performs an iterative deepening search in the state space [13] when the Prolog query "`plan(S),exec(S),holds(agentAt[Goal_Pos],S)`" is done, where **plan(S)** and **exec(S)** are defined as:

```
exec(s0).
exec(do(A,S)) :- poss(A,S), exec(S).
plan(s0).
plan(do(A,S)) :- plan(S).
```

This algorithm is used with the axioms of the Situation Calculus, to infer that $(\exists s).plan(s) \wedge exec(s) \wedge G$, in which G is a goal state. It is interesting to notice that this algorithm, despite its simplicity, is a concise way to find a solution plan s.

4.4 Description of the representation of the Wumpus World

In order to build a physical model, we had to find a way to represent the environment considering the sensors available for the Lego® MindStorms™. In our case, we only used a light sensor. The squares in the grid were identified by tags with different light emitting properties (opaque dark tags, shining silver tags, and so on).

In Figure 5 the dashed lines delimit each position in the Wumpus World, and the solid lines represent the possible paths for the robot to move.

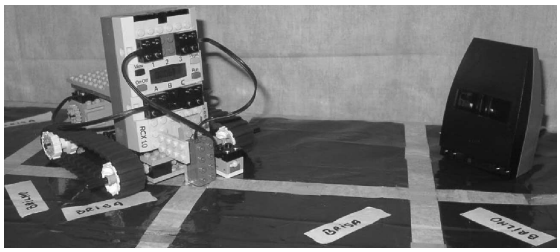

Fig. 6. Lego® MindStorms™ robot and the physical model of the Wumpus World.

The **perception tags** were created to represent the three possible perceptions for the Wumpus World: **glitter**, **stench** and **breezy**. All positions in the environment were divided in quadrants. Each quadrant is used to represent one type of perception (Figure 5). Another type of tag is the **rotation tag**, used by the perception algorithm to collect perception in the quadrants in P. A physical model to represent the Wumpus World can be seen in Figure 6.

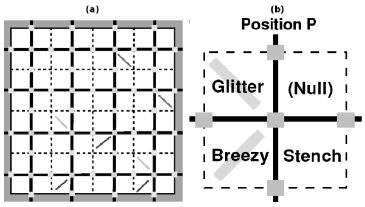

Fig. 5. (a) Physical representation of Figure 2 and (b) zoom of the position (4,4).

4.5 Primitive actions for the RCX brick

The perceptions collected in a position for **glitter**, **stench** and **breezy**, is implemented by the primitive action **percept** in the RCX brick which makes the robot to rotate 360° and returning a vector with the four perceptions. Since the robot can reach a position coming from four alternative directions (N, S, E or W), hence the reasoning program of the agent uses the present orientation of the robot to determine the amount of circular *shifts* that must be applied to the vector to recover the correct perception.

The following primitive actions were also implemented in the RCX brick: **turn_clockwise**, **turn_anti_clockwise**, **forward**, **grab**, **shoot** and **climb**. For the actions **grab**, **shoot** and **climb**, the robot emits different sounds indicating that they have been executed and waiting for a manual update of the environment.

5 Conclusions

Usually, researchers in cognitive robotics make use of simulations to test their theories, due to the difficulty to find affordable robots that are simple to program, assemble and use. The Lego® MindStorms™ robots present all these features. Moreover, the language Legolog permits the implementation of reasoning capabilities in robots without requiring skills about the robot's hardware. In [3] an implementation of Legolog was done to show the use of this language for a simple task: to make the robot follow a line recognizing and reacting to labelling tags. We could not find, however, in the literature, results showing the actual use of Lego® MindStorms™ robots to implement higher level cognitive actions.

The present work introduces a Legolog application that solves a much more challenging task than that presented in [3] – namely, the solution of the Wumpus World problem, which can be envisaged as a simplified model of a search and rescue scenario. This high-level behavior was implemented in Legolog as a concise and elegant program, making good use of the power of the Golog meta-interpreter.

Instance number	1	2	3
Wumpus position	[3,3]	[0,2]	[3,1]
Coin position	[1,1]	[1,2]	[3,0]
Pits positions	[2,0]	[3,1], [3,3]	
Number of steps of the executed plan	22	25	18
Average time for *on-line* action selection	0.01s	0.42s	0.05s
Standard deviation of time for *on-line* action selection	0.03s	1.68s	0.03s
Average time for plan generation (*off-line*)	0.03s	1.35s	0.13s
Standard deviation time for plan generation (*off-line*)	0.01s	1.30s	0.10s
Total time spent by the agent reasoning program	0.38s	13.25s	9.16s

Table 1. Statistic results for 3 instances of the Wumpus World with 1 coin position and 0.2 probability of pits.

Table 1 shows three instances of the Wumpus problem, with the Wumpus position, coin position and pits positions specified in lines 2, 3 and 4, respectively. In all experiments the robot has correctly detected safe and dangerous positions and also found optimal planning solutions to determine the approximation of the Wumpus and the way out. These features can be also formally verified from the agent's logical specification. The experiments show that for a medium size plan (22, 25 and 18 steps), the average time spent for *on-line* action selection was less then 0.5 seconds, while the average time for (*off-line* planning was less then 1.4 seconds. Since for this experiments we have used a planning method based on a logical formalism, with no use of heuristics, these results give us a good idea of how this approach can be seriously used for demonstrations and investigations of formal theories for robotics.

References

1. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. JLP **31** (1997) 59–84
2. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Workshop on Decision-Theoretic Planning, Proc. KR-00. (2000)
3. Levesque, H., Pagnucco, M.: Legolog: Inexpensive experiments in cognitive robotics. In: Proc. of the 2nd International Cognitive Robotics Workshop. (2000)
4. Mccarthy, J.: Situations, actions and causal laws. MIT Press (1963)
5. Reiter, R.: Sequential, temporal golog. In: Principles of Knowledge Representation and Reasoning: Proc. KR-98. (1998) 547–556
6. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd. edn. Prentice-Hall, Inc. (2003)
7. Kambhampati, S., Knoblock, C.A., Yang, Q.: Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. Artificial Intelligence **76** (1995) 167–238
8. Bonet, B., Geffner, H.: HSP: Heuristic Search Planner. In: Proc. of AIPS. (1998)
9. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. J. of Artificial Intelligence Research **14** (2001) 253–302
10. Erol, K., Hendler, J.A., Nau, D.S.: UMCP: A sound and complete procedure for hierarchical task-network planning. In: Proc. AIPS. (1994) 249–254
11. Barros, L.N., Iamamoto, E.: Planejamento de tarefas em golog. In: SBAI. (2003)
12. Lespérance, Y., Ng, H.: Integrating planning into reactive high-level robot programs. In: Proc. of the 2nd International Cognitive Robotics Workshop. (2000)
13. Pereira, S.L., Barros, L.N.: Formalizing planning algorithms: a logical framework for the research on extending the classical planning approach. In: Proc. of the ICAPS Workshop: Connecting Planning Theory with Practice. (2004)