MODELLED EXPLORATION BY ROBOT

by

L. Siklóssy

April 1972 TR-1

Technical Report No. 1

Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Siklossy, L.

ABSTRACT

Techniques which permit a modelled mobile robot to explore its environment are described. The robot generates tasks for itself, generalizes solutions to these tasks, changes and improves the efficiency of its operators, and generates its own advice to speed its search techniques. Learning takes the form of modification and creation of programs that are used by the robot.

We discuss how a robot can attempt to test the adequacy of its world by checking whether global properties of the modelled universe, perhaps in the form of conservation laws, hold in the states that can be reached.

1. INTRODUCTION

The development of sophisticated robots is no longer a domain reserved to science fiction. Computer-controlled hand-eye systems have been built in the United States[1,2], Great Britain[3] and Japan[4]. An eye on wheels is operational in the United States[5]. The range of applications of robots is so wide that we can expect a growing interest in the field of robotics.

A robot's capabilities of solving tasks can be expected to be largely proportional to the amount of knowledge that it has of its environment and the ways it can interact with this environment; and of itself and the means of improving its performance, However much knowhow is initially given to a robot, it is desirable to investigate robots that can improve over this initial knowledge. We propose that the exploration of its environment is a method by which the robot can become more sophisticated.

We shall describe, and exemplify, techniques which model various aspects of exploration by robot. We may conceive (in an anthropomorphic way) that the robot, as it explores its environment, tries to answer the following questions:

1) What is the nature of my environment? Which objects surround me? What are their properties? In what ways can they be manipulated? (Section 2 gives some answers to these questions.)

2) What tasks can I perform? Which tasks are impossible, undesirable, dead-ends? Given some task that I can achieve, can I also achieve some more general task? Can I learn to perform the task more efficiently? (In section 3, we shall consider how the robot can introspect to determine its own capabilities.)

3) To which extent is my model of the world correct? Can I perform tasks that result in views of the world that are in contradiction

with other information that I have about the world? (In section 4, we shall see how the robot can check to some degree the validity of its own model.)

Our discussion pools the partial results of several experimental programs that simulate robot exploration. To focus the discussion, we shall be interested in a mobile robot in the SRI robot's[5] environment of rooms, connecting doors, boxes, etc. Choosing this environment saves us both from having to design our own robot, and from falling into the temptation of cheating by modelling a robot built specifically to illustrate our views of exploration. We hope that our ideas can be transferred to the exploration of other environments.

2. EXPLORATION OF THE ROBOT'S SURROUNDINGS.

A model of a robot, following STRIPS[5], is given in the Appendix. We shall call this model APWORLD. A state of the robot is represented as a conjunction of predicates giving properties of objects in the model. Operators can transform one state into another state. For an operator to be applicable to a state, the state must satisfy the preconditions of the operator. Predicates of the state that match the Delete list of the operator are deleted from the state, while the predicates of the Add list of the operator are added to the representation of the state.

Operators are applied in the robot's model of the world, not in the real world. For example, the operator climbonbox(box) does not make the robot climb the box. It performs a _gedanken_ move in the model world of the robot. If the robot so chooses, a sequence of operators can be translated by an executive program into real world actions. We assume that the executive receives feedback from sensory organs as the robot moves in the world, and that it is appropriately interrupted if the real world makes it impossible to accomplish certain actions that seemed possible in the model.

Since we are dealing with two worlds: the _real_ physical world, and the robot's model of it, a little terminology will help. A _real task_ attempts to answer a question (Is the robot near a box?), or solve a problem (Bring about a state where the robot is near a box.) in the _real_ physical world. A _modelled task_, or simply _task_, attempts to answer similar questions, or solve similar problems, based only on the model of the world. In general, the model of the world is much less complex than the real world. In this section, and in the next one, we shall assume that the solution of real tasks can usually be inferred from the solution of modelled tasks. In section 4, we shall lift this restriction.

2.1 Discovery of Objects.

In the real world, the discovery and recognition of objects can be a time-consuming activity. In a modelled exploration, it is possible to simulate the discovery of objects by incorporating these objects in the model, but hiding them from the robot (for example by setting a bit attached to the object.) If some circumstances are met (for example if the robot happens to "bump" into a box), the hidden object is discovered by the robot.

2.2 Discovery of Properties of Objects.

In a similar (and just as uninteresting) manner, we can simulate the discovery of properties of objects. For example, after some box has been visited by the robot, the fact that this box is pushable, which was preprogrammed but initially invisible, becomes evident to the robot. Similarly, the robot could "discover" that a lightswitch could be turned on.

2.3 Discovery of Relations among Objects.

As the robot goes poking around, he can accumulate more challenging*

* Information which was preprogrammed, but initially hidden, is not particularly challenging.

information about objects and their interrelationships. For example, an estimate of the distance separating two objects in two different rooms would depend on various paths, through other rooms, that link the two rooms. As new rooms are entered by the robot, the shortest known path between the two objects might be changed. Shortest path information is useful to solve tasks.

3. SOLVING TASKS.

After some preliminary exploration, the robot has some knowledge of its environment: of objects, their properties, and how they can be manipulated. The Appendix describes such a state of knowledge in the APWORLD model of the robot. The robot will have to solve tasks in its model, and will explore its capabilities of becoming a better problem solver.

A state in the model is a conjunction of predicates in the model.[*] APWORLD describes an initial state. A task is an ordered pair of states, ($state_1$, $state_2$). A task solution to this task is a sequence of operators op_1, op_2, ..., op_n, such that all the predicates of $state_2$ are contained in the state:

$$op_n\ (\ldots op_2\ (op_1\ (state_1))\ \ldots)^{**}$$

Difficult tasks will have long task solutions, i.e., will be made up of long sequences of operators. Solving a task becomes a problem of search, and two main approaches seem to have been used for this type of problem.

* We say nothing at this point about whether such a state models some realizable physical situation.

** We assume that the preconditions to all the operators are satisfied as necessary.

(a) Fairly general (hence weak) tools are given to the robot to solve tasks. For example, STRIPS[5] solves tasks by using means-ends analysis techniques.

(b) Much information about how to solve a large variety of tasks is given to the robot's task solver. Such an approach was used by Winograd.[6] The task solver, written in a dealect of Planner,[7] is given advice on how to manipulate a world of blocks.

If the world of the robot is sufficiently complex, we can assume that the advice provided in an approach similar to (b) above will not permit the task solver to solve all problems without search. In an approach similar to (a), few difficult tasks could be solved. In both approaches, the task solver must improve its performance. Perhaps the only difference is that, in the second case, improvements start at a higher level of proficiency.

We conceived of the task solver as being able to generate its own advice. This advice is used to prune the search tree and to direct the search. We now discuss some ways in which the robot can generate its own advice.

3.1 Non-Recoverable Properties.

In a state, some of the predicates may be <u>non-recoverable</u>: that is, if they have been deleted by some operator, there is no way that they can be added. For example, if our model included an operator BURN(box), the ashes that result from BURNing the box could not be transformed back into the box.

In APWORLD, the robot considers the Delete and Add lists of its operators, and finds several non-recoverable properties:* --it is impossible

* Here, as in the other examples derived from APWORLD, no attempt is made to be complete. Unless otherwise indicated, the statements about APWORLD also hold in the complete STRIPS[5] system.

to have any of the three boxes AT any place beyond their original location.

If a non-recoverable property must be present in the final state of a task, the robot generates the absolute advice that at no time should this property be deleted (from intermediate states).

3.2 Non-Changeable Properties.

Some of the predicates in the initial state are easily found to be non-changeable, since they are mentioned in neither Delete nor Add lists of any operator. For example, in APWORLD, CONNECTS(DOOR1, ROOM1, ROOM5) and TYPE(BOX1, BOX) are such predicates. These predicates are global to all states, and may be handled more efficiently by the task solver.

Other predicates are added by some operator, but cannot be deleted by any operator. In APWORLD, once STATUS(lswitch, ON) is true, it can never be deleted: the LIGHTSWITCH1 will always be ON.

If a non-changeable property is not present in the final state of the task, the robot generates the absolute advice that at no time should this property be added (to intermediary states).

3.3 Undesirable Properties.

The appearance, or disappearance, of certain predicates may be highly undesirable, and would generate strong, if not absolute, advice to the robot. The property that has appeared may be hard to delete if not wanted; or it may be very costly to rediscover a property that has disappeared. As an example of the former, if some BOX4 could be glued to the ground, rendering it unpushable (unless unglued, an expensive task), and if BOX4 must be moved, then it is undesirable to glue BOX4 to the ground.

An example of the latter is found in APWORLD. From the initial state, the sequence of operators:

goto2a(ROBOT,BOX1), pushtoa(BOX1,BOX2),goto2a(ROBOT,BOX2),pushtoa(BOX2,BOX3) deletes positional information about BOX1: i.e. in the model, BOX1 is neither AT some place, nor NEXTTO something. It is possible (if tricky) to lose all AT and NEXTTO information about all the boxes in APWORLD. If we think now of a task solution, containing such undesirable properties, as mapped back to a sequence of actions of a real task, it is clear that the robot must rediscover the position of all the boxes, a costly chore. (In the robot model, we must have information about such costs to be able to discover certain undesirable properties.)

3.4 Impossible Tasks.

In solving the task (state$_1$, state$_2$), it may happen that we hypothesize some intermediary state$_3$, and try to solve the two subtasks (state$_1$, state$_3$) and (state$_3$, state$_2$). If state$_3$ cannot be realized in the model, then the first subtask is not solvable. Knowledge about impossible tasks allows the task solver to avoid doomed solution paths. We shall see how some non-trivial impossible tasks can be discovered during task solution generalization.

3.5 Generalization of Task Solutions.

Once a solution to a task has been found, it is desirable that this solution be generalized to a larger class of tasks. We shall indicate various directions in which the generalization can be made.

3.5.1 Generalization over Objects.

If a task was solved for some specific objects, for example in APWORLD:

place BOX2 next to DOOR1 with LIGHTSWITCH1 ON

the task solver tries to generalize over the objects BOX2, DOOR1 and LIGHTSWITCH1. Each of the objects is replaced by a variable having as domain objects in candidate extension sets. In APWORLD, extension sets could be the whole universe; combinations of objects with certain properties:

objects of TYPE BOX; or objects that can replace some parameter in an operator: for example the variable 'door' in gothrudoor.

Certain extensions will lead to impossible tasks. For example if APWORLD is extended to include some other DOOR2 of ROOM5, then the task is no longer solvable since the boxes cannot leave ROOM1. If APWORLD were augmented to permit boxes to be pushed from one room to another, then the extension would include appropriate calls to path finding routines. Knowledge of shortest paths between objects (see section 2.3) would prove valuable.

3.5.2 Generalization over Number

If certain objects have been members of several extension sets, it is reasonable to try to generalize certain tasks over several of these objects simultaneously.

For example, in APWORLD, if it has been noticed that BOX1, BOX2 and BOX3 belonged to several extension sets, then from the solution of the task: (initial state; NEXTTO(BOX1, BOX2)) it is natural to generalize to a solution of:

(initial state; NEXTTO(BOX1, BOX2), NEXTTO(BOX1, BOX3))

which is solvable, and to generalize further to the task:

(initial state; NEXTTO(BOX1,BOX2), NEXTTO(BOX1,BOX3), NEXTTO(BOX2, BOX3))

which is not solvable in APWORLD!

3.5.3 Generalization over Preconditions

We shall see below (section 3.5.5) that every task solution can be viewed as a giant operator, with its preconditions, Delete list and Add list. We shall be interested in generalizing operators by weakening their preconditions.

To focus attention, consider the operator turnonlight in APWORLD. Its precondition is:

TYPE(lswitch,LIGHTSWITCH) ON(ROBOT,BOX1) NEXTTO(BOX1,lswitch)

This operator is generalized if we weaken its precondition. We could weaken the precondition in several ways:

1) elimination of one or more predicates:

state_1 = TYPE(lswitch, LIGHTSWITCH) NEXTTO(BOX1,lswitch)

2) weakening of a constant to a variable:

state_1 = TYPE(lswitch,LIGHTSWITCH) ON(ROBOT,x) NEXTTO(x,lswitch)

3) weakening of a coupling among parameters:

state_1 = TYPE(lswitch,LIGHTSWITCH) ON(ROBOT,x) NEXTTO(y,lswitch)

or a combination of these three methods.

For the generalization to be successful, we must solve the task: (state_1; original precondition of operator), where to state_1 we add the non-changeable initial properties of the model. All of the three generalizations mentioned are feasible. The greatest generalization of turnonlight reduces the precondition to TYPE(lswitch,LIGHTSWITCH). The code generated for this new operator includes the subtask: get ONFLOOR first, then if necessary:
goto2a(ROBOT,BOX1), pushtoa(BOX1, lswitch), climbonbox(BOX1).

3.5.4 Updating of Operators, and Improved Efficiency.

If an operator can be successfully generalized, the new version of the operator replaces the old one, since the old operator serves no real purpose.

It is also possible to change operators in a somewhat less radical way. The robot can use the technique of hereditary properties to determine that, in APWORLD, only one predicate of the form NEXTTO(ROBOT,x) can hold in any one state. Hence, whenever the particular form of this predicate is found by a precondition, as in the predicates pushtoa and gothrudoor, the delete list component NEXTTO(ROBOT,$) can be replaced by a precise NEXTTO(ROBOT,object), where the value of "object" has already been bound. In this way, some efficiency is gained in searching the state of the robot.

3.5.5 Macro-operators.

The preceding discussion was general if we can show that every task solution can be considered as an operator. It is enough to show that a task solution consisting of the product $op_2 \cdot op_1$ of two operators can be viewed as an operator. Let pre_i, del_i and add_i be the preconditions, Delete list and Add list of op_i. Notice that op_1 is applied before op_2.

Let
$$pre = pre_1 + (pre_2 - add_1)$$
$$del = del_2 + (del_1 - add_2)$$
$$add = (add_1 - del_2) + add_2$$

(where + and - are set union and difference, following appropriate binding of variables) be the precondition, Delete list and Add list of a new operator op. It is seen, by construction, that op has the same effect as $op_2 . op_1$.

Macro-operators for arbitrary task solutions can be generated automatically. However, they tend to have unwieldy preconditions, and should be followed by a generalization over preconditions.

3.6 Hierarchies in Subtasks.

A goal state is represented as the conjunction of several predicates. If we partition these predicates into disjoint subsets, each subset can be viewed as a subtask that must be solved. Usually, the subtasks are not independent, since they are coupled through common variables and constants. Nevertheless, it is desirable to find some reasonable ranking of subtasks. For example, it appears "obvious" that if we wish, in APWORLD, the robot to be next to the doorl, and the light to be on; then the robot should first turn the light on, then go to the door.

A good ordering of subtasks appears quite powerful. By taking the partition to include single predicates only, and reordering them in a fixed hierarchic order, it is possible to write a task solver for the STRIPS world[5] that seems capable of solving all the tasks in that world. Even the longest solutions (15 operators) take less than 0.6s (interpreted LISP on CDC-6600) while STRIPS requires 65.0s, 122.1s and 125.9s (partially compiled LISP on PDP-10, excluding garbage collection time) to solve problems which have solutions of length 4, 4 and 5 respectively.

While it is unlikely that, in a rich environment, algorithm-like solutions to tasks exist, a robot can explore its environment to estimate the relative priority that should be given to the solution of the atomic subtasks of a task.

It is sometimes possible to determine the <u>last</u> operator that must be applied. In APWORLD, a ON(ROBOT, object) in the final state means that climbonbox must be applied last. More generally, the robot explores the freedom it has to perform a subtask in the two cases when it has, or has not, accomplished another subtask.

The relative hierarchy of subtasks has been coupled with a measure of the distance between objects (in fairly complex scenes) to yield a distance function that approximates the distance between two states. The early results are very encouraging.

3.7 The Generation of Tasks

The tasks solver is given <u>ab initio</u> some tasks that are "interesting": those that can be solved by one application of an operator! Using various generalization techniques, and other considerations such as symmetry which we have not discussed here, it is possible to generate a large number of new "interesting" tasks. Beyond this point, a task generator may be necessary.

The generator could be an outside source of "interesting" tasks. It appears particularly challenging to design, as part of the robot, a task generator that would, in the purest scienctific spirit, challenge the model of the robot. We turn to a related topic in the next section.

4. MODEL TESTING

The model of the robot is only a projection of the real world. Just as a scientific theory is an incomplete explanation of physical phenomena, we may expect the model to desagree "badly" with the real world. We do expect that errors in measurement will place objects at incorrect places in the model: these are minor errors. The failure of the robot to recognize a door would be less minor. But a truly drastic error in the model would be the failure to satisfy, in some state, some higher-level properties of its world, such as conservation laws. In many worlds, we expect lightswitches to remain lightswitches, and boxes not to multiply.

We have found a violation of a conservation law in APWORLD. If the robot pushes a box next to itself (the robot) and then moves away next to another box, he will be in a state when he is sumultaneously next to two objects that could be far from each other. As the reader can guess, the robot did not notice this violation: it was not endowed with such sight!

As models become increasingly complicated, the designer can be less and less sure of their adequacy. Extensive exploratory testing of the model, followed by the generation of tasks that attempt to contradict conservation laws of the model, provide an added measure of adequacy.

5. CONCLUSIONS

Robot exploration is a tool that permits a task solver to generalize its capabilities, generate its own advice, modify its own capabilities and improve its performance. The task solver learns by generating and modifying the programs that it uses to solve tasks. Hence, learning takes the form of growth of structures, and their modifications through experience[8,9]. The robot can study part of its own model, although the fragments of the robots's programs that do this study are not themselves studied. In this way we might avoid some problems associated with multiple levels of models[10].

Finally, the robot can check the adequacy of its own model by resorting to a form of super-model that could be represented as conservation rules.

6. ACKNOWLEDGMENTS

The author owes an intellectual debt to the designers of STRIPS. Discussions with, and programs by, participants in a problem-solving seminar in the Fall of 1971, in particular R. Amsler, C. Dawson, J. Dreussi and G. Hendrix, helped exploring... robot exploration.

7. REFERENCES

1. Feldman, J. A. and Sproull, R. F., System Support for the Stanford Hand-Eye System. Second Int. Joint Conf. on Artificial Intelligence. 1971, 183-189.

2. Project MAC Progress Report VIII, July 1970 to July 1971. Massachusetts Institute of Technology, Cambridge, MA, 1971.

3. Barrow, H. G. and Popplestone, R. J., Relational Descriptions in Picture Processing, In: Meltzer, B. and Michie, D. (Eds.), Machine Intelligence 6, American Elsevier, New York, 1971, 377-396.

4. Ejiri, M., Uno,T., Yoda, H., Goto, T. and Takeyasu, K., An Intelligent Robot with Cognition and Decision-Making Ability. Second Int. Joint Conf. on Artificial Intelligence. 1971, 350-358.

5. Fikes, R. E. and Nilsson, N. J., STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Second Int. Joint Conf. on Artificial Intellegence. 1971, 608-620.

6. Winograd, T., Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. Report MAC TR-84, Massachusetts Institute of Technology, Cambridge, MA. 1971.

7. Hewitt, C., Procedural Embedding of Knowledge in Planner. Second Int. Joint Conf. on Artificial Intelligence. 1971, 167-182.

8. Siklóssy, L., A Language-Learning Heuristic Program. Cognitive Psychology, 2, 4, 1971, 479-495.

9. Winston, P. H., Learning Structural Descriptions from Examples. Report MAC TR-76, Massachusetts Institute of Technology, Cambridge, MA, 1970.

10. Minsky, M. L., Matter, Mind, and Models. In: Minsky, M. L. (Ed.) Semantic Information Processing, MIT Press, Cambridge, MA, 1968, 425-432.

APPENDIX

APWORLD

This description of the robot is obtained after making the following changes to the STRIPS robot[5]:

--only the contiguous ROOM1 and ROOM5 are kept.

--the STRIPS operators goto2 and pushto have been split, for convenience, into pairs of operators having conjunctive preconditions.

--with the elimination of location F in ROOM4, operator goto1 would serve no purpose and was removed.

Uppercase identifiers have been used for predicate names and constants. Mnemonic lower case identifiers are used for variables, to make the operator definitions easier to grasp. The short descriptions of operators are sometimes incomplete: again, they are given only to help the reader.

INITIAL STATE OF APWORLD

CONNECTS(DOOR1,ROOM1,ROOM5); CONNECTS(DOOR1,ROOM5,ROOM1); AT(LIGHTSWITCH1,D);

TYPE(BOX1,BOX); TYPE(BOX2,BOX); TYPE(BOX3,BOX); TYPE(LIGHTSWITCH1,LIGHTSWITCH);

INROOM(BOX1,ROOM1); INROOM(BOX2,ROOM1); INROOM(BOX3,ROOM1); INROOM (LIGHTSWITCH1,ROOM1);

PUSHABLE(BOX1); PUSHABLE(BOX2); PUSHABLE(BOX3);

AT(BOX1,A); AT(BOX2,B); AT(BOX3,C);

ATROBOT(E); INROOM(ROBOT,ROOM1); ONFLOOR; STATUS(LIGHTSWITCH1,OFF)

Operators in APWORLD

goto2a (object). Robot goes next to an object (typically a box) in the same room. Preconditions:

ONFLOOR $\wedge \exists$ place [INROOM(ROBOT,place) $\wedge$ INROOM(object,place)]

Delete list: ATROBOT($),NEXTTO(ROBOT,$)

Add list: NEXTTO(ROBOT,object)

goto2b (object). Robot goes next to connecting object (typically a door).

Preconditions:

ONFLOOR $\wedge \exists$ place1 $\exists$ place2 [INROOM(ROBOT,place1) $\wedge$ CONNECTS(object,place1,place2)]

Delete list and Add list: same as goto2a.

pushtoa (object1, object2). Robot pushes object1 (typically a box) next to object2 (typically a box).

Preconditions:

PUSHABLE(object1) $\wedge$ ONFLOOR $\wedge$ NEXTTO(ROBOT,object1) $\wedge \exists$ place [INROOM(object1,place) $\wedge$ INROOM(object2,place)] .

Delete list: ATROBOT($),AT(object1,$),NEXTTO(ROBOT,$),NEXTTO(object1,$), NEXTTO($,object1)

Add list: NEXTTO(object1,object2),NEXTTO(object2,object1),NEXTTO(ROBOT,object1) pushtob (object1, object2). Robot pushes object1 (typically a box) next to object2 (typically a door).

Preconditions:

PUSHABLE(object1)∧ONFLOOR∧NEXTTO(ROBOT,object1)∧∃place1 ∃place2 [INROOM(object1,place1)∧CONNECTS(object2,place1,place2)]

Delete list and Add list: same as pushtoa.

turnonlight (lswitch). Robot turns on lightswitch lswitch.

Preconditions:

TYPE(lswitch,LIGHTSWITCH)∧ON(ROBOT,BOX1)∧NEXTTO(BOX1,lswitch)

Delete list: STATUS(lswitch,OFF)

Add list: STATUS(lswitch,ON)

climbonbox (box). Robot climbs on box 'box'.

Preconditions: ONFLOOR∧TYPE(box, BOX)∧ NEXTTO(ROBOT,box)

Delete list: ATROBOT($),ONFLOOR

Add list: ON(ROBOT,box)

climboffbox (box). Robot climbs off box 'box'.

Preconditions: TYPE(box,BOX)∧ON(ROBOT,box)

Delete list: ON(ROBOT,box)

Add list: ONFLOOR

gothrudoor (door,fromroom,toroom). Robot goes through door 'door' from room 'fromroom' to room 'toroom'.

Preconditions:

NEXTTO(ROBOT,door)∧CONNECTS(door,fromroom,toroom)∧INROOM(ROBOT,fromroom)∧ONFLOOR

Delete list: ATROBOT($),NEXTTO(ROBOT,$),INROOM(ROBOT,$)

Add list: INROOM(ROBOT,toroom)