AN EFFICIENT ROBOT PLANNER WHICH GENERATES ITS OWN PROCEDURES

by

L. Siklossy and J. Dreussi

1973 TR-10

To appear in: Proc. Third International Joint Conference on Artificial Intelligence, Stanford, CA, August 1973.

TR-10

AN EFFICIENT ROBOT PLANNER WHICH GENERATES ITS OWN PROCEDURES*

L. Siklóssy and J. Dreussi

Department of Computer Sciences

University of Texas, Austin, U.S.A.

Abstract

LAWALY is a LISP program which solves robot planning problems. Given an axiomatic description of its capabilities in some world, she generates her own procedures to embody these capabilities. She executes these procedures to solve specific tasks in the world. Hierarchies of subtasks guide the search for a solution. In sufficiently large worlds, LAWALY has routinely solved tasks requiring several hundred steps without needing to learn from previous tasks. The times to solution grow usually about linearly with the number of steps in the solution.

LAWALY is extensively compared to another robot planner based on a theorem prover.
Key words follow references.

1. Introduction: Approaches to Robot Planning.

A robot planning program, or robot planner, attempts to find a path from an initial robot world to a final robot world. The path consists of a sequence of elementary operations that are considered primitive to the system. A solution to a task could be the basis of a corresponding sequence of physical actions in the physical world.

In late 1971 and early 1972, two main approaches to robot planning were in use. One approach, typified by the STRIPS family of programs[1,2,3] at the Stanford Research Institute, is to have a fairly general robot planner which can solve tasks in a great variety of worlds. The second approach is to select a specific robot world, and for that world to write a specific program to solve tasks.[4,5] The program-writing can be made easier by the use of programming languages specifically designed for programming robot-like tasks, as, for example, PLANNER, MICROPLANNER, or QA4,[6,7,8] An example of a specialized robot is Winograd's,[4] operating in a world of blocks.

The first approach has lacked power, in the sense that some problems requiring only six steps to be solved push the problem-solver to its practical limitations. The second approach lacks generality, in that a new set of programs must be written for each world. On the other hand, for a given world, the second approach results in robot planners which can solve tasks well beyond the capabilities of the general robot planners of the first kind, while, on the set of commonly solved problems, the specialized systems are several hundred times faster!

To obtain a system that maintains generality, while having power and good performance, we have designed a robot planner which builds a specialized set of programs for each world it is given. Therefore, our system--named LAWALY--duplicates (to some extent) what a human being does when programming a specialized robot planner. LAWALY builds a procedure for each operator in the world, and she links these procedures with an overall monitor. She reorders subtasks so that she can most efficiently solve them, avoiding dead ends--which would necessitate backtracking--as much as possible.

The results have been up to our expectations. LAWALY is general: she has been applied to more than twenty worlds. She exhibits good performance: programmed in LISP on the CDC 6600 and run interpretively, she averages from 0.4 to about 2 seconds per node in her solution space, depending on the amount of backtracking and the size of the changing world. Finally, she has power: she routinely solves tasks that require several hundred steps. Moreover, the time taken to solve a task is about linear with the number of steps in the solution.

The self-generation of procedures by LAWALY, her use of a hierarchy of subtasks, and the results of her performance form the main parts of this article. For the impatient reader, Tables 1, 2 and 3 compare the performance of STRIPS with LAWALY for the tasks described graphically in Figures 1, 2 and 3 respectively.

2. Procedure Generation for an Operator.

As in [1], we consider worlds consisting of sets of predicates such as AT(BOX1 A1). The world can be changed by applying to it an operator. An operator can be applied to a world only if the world satisfies the preconditions of the operator. The changes of the world as a result of the application of the operator result from deleting from the world the delete set of the operator, then adding to the resultant world the add set of the operator. Two typical examples of operators, which will be used below, are:
push(object1 object2), meaning: robot pushes object1 next to object2.
Preconditions: PUSHABLE(object1) ONFLOOR NEXTTO(ROBOT object1) INROOM(object2 rm) ARMSEMPTY.
Delete set: AT(ROBOT $ $) NEXTTO(ROBOT $) AT(object1 $ $) NEXTTO(object1 $) NEXTTO($ ROBOT) NEXTTO($ object1).
Add set: NEXTTO(object1 object2) NEXTTO(ROBOT object1) NEXTTO(object2 object1).
gonextobj(object), meaning: robot goes next to object.
Preconditions: INROOM(ROBOT rm) INROOM(object rm) ONFLOOR.
Delete set: AT(ROBOT $ $) NEXTTO(ROBOT $) NEXTTO($ ROBOT).
Add set: NEXTTO(ROBOT object) NEXTTO(object ROBOT).
Note: This is only one of several possible axiomatizations for these operators. Since our emphasis here is on the solution of tasks, and not on how they should best be axiomatized, we make no claims as to the "correctness" of any of the operator axiomatizations chosen!

We shall now attempt to describe the procedure generation used by LAWALY. The types of procedures that are obtained resemble those that have been hand-coded by various individuals. If it is sometimes difficult to explain programs that manipulate some world, it is even more difficult to explain (and to program!) programs that generate programs that manipulate some world. To avoid being submerged in coding details, our description must remain sketchy.

LAWALY is coded in LISP, and for each operator a LISP EXPR is generated. Taking gonextobj as an example, the LISP function will be:
(GONEXTOBJ (LAMBDA (ALIS OBJECT) (PROG (RM) ...))).
The body of the PROG includes parts that perform the selection of bindings and the interaction with the monitor.

2.1 Selection of Bindings.

Bindings must be found for all the LAMBDA-variables (excepting ALIS) and PROG-variables. Variables may be bound from the call of the function, for example if we happen to want to execute gonextobj(BOX1) then OBJECT is bound to BOX1; or if the ALIS is not NIL it contains alternate bindings (this is used in backtracking.) All unbound variables are passed, together with the preconditions of the function, to a binding procedure. This

*Work partially supported by grant GJ-34736 from the National Science Foundation.

procedure attempts to bind these variables for the smallest cost of satisfying the preconditions. The cost of a choice of bindings is inversely proportional to the number of predicates of the precondition of the operator that are satisfied in the world. If, for example, RM is bound to the room ROOMC which satisfies (INROOM BOX1 ROOMC), then the precondition (INROOM OBJECT RM) will be satisfied immediately. The various alternatives for bindings are computed, and they will eventually all be tried. Alternate bindings are kept on the ALIS.

2.2 Interaction with the Monitor.

The preconditions that the operator must satisfy are reordered according to the hierarchy of subtasks (see section 3 below), and sent to the monitor. For each condition in turn, the monitor checks whether the condition holds in the world. If it does, the monitor passes to the next condition. If it does not hold, the monitor selects one (of possibly several) operator(s) which might change the world into one in which the desired condition holds. Before this operator is evaluated, using the LISP function EVAL, a node is created in the search space of operators. An example will clarify the process.

Assume an initial world W1 of three boxes and the robot in one room ROOM. An axiomatic description would be: (AT ROBOT A) (AT BOX1 A1) (AT BOX2 A2) (AT BOX3 A3). The BOXes are assumed PUSHABLE, the robot's arms are empty and she is on the floor. The desired final state is (NEXTTO BOX1 BOX2) (NEXTTO BOX2 BOX3). Monitor starts working to satisfy (NEXTTO BOX1 BOX2), since the other subtask of the goal has the same rank (see section 3.) To achieve this subtask, either (PUSH BOX1 BOX2) or (PUSH BOX2 BOX1) can be tried. The first choice is attempted, and the second saved for backtracking. A node G1 is created for the move (PUSH BOX1 BOX2) in the operator tree, and the choice kept there. (See Figure 4.)

We now EVALuate (PUSH BOX1 BOX2). Precondition (NEXTTO ROBOT BOX1) is found not to hold, control returns to the monitor which finds that (GONEXTOBJ BOX1) might lead to a world satisfying (NEXTTO ROBOT BOX1). Node G2 is created, LAWALY EVALuates (NEXTTO ROBOT BOX1), and we obtain a new world W2, (NEXTTO ROBOT BOX1) (AT BOXi Ai), i=1,2,3. At that point, EVALuation of (PUSH BOX1 BOX2) can be completed, to give a new world: W3: (NEXTTO ROBOT BOX1) (NEXTTO BOX1 BOX2) (NEXTTO BOX2 BOX1) (AT BOXi Ai), i=2,3. Monitor turns attention to the second subtask of the final goal, (NEXTTO BOX2 BOX3). To realize this subtask, either (PUSH BOX2 BOX3) or (PUSH BOX3 BOX2) can be tried. Starting with the first (node G3), the subgoal (NEXTTO ROBOT BOX2) is generated, with operator (GONEXTOBJ BOX2) as the way to obtain the goal (node G4). Operator G4 applied to world W3 gives world W4: (NEXTTO BOX1 BOX2) (NEXTTO ROBOT BOX2) (AT BOXi Ai), i=2,3. At that point operator G3 has all its preconditions satisfied, so it could be applied. But communication between monitor and G3 indicates that a previously satisfied subgoal, (NEXTTO BOX1 BOX2) would be deleted if G3 were applied. Backtracking descends to node G4, where no alternate operator is found. Backtracking then ascends to G3, where the alternate operator (PUSH BOX3 BOX2)--node G3'--is selected. To apply this operator, its precondition (NEXTTO ROBOT BOX3) must be satisfied. Hence, node G5 with operator (GONEXTOBJ BOX3) is created. G5 can be, and is, applied to world W3, to yield world W5. G3' is applied to W5 to yield W6, our final solution.

It is seen that the operator tree is created in preorder, while the successive states of the world correspond to the applications of the operators as the operator tree is traversed in endorder. Backtracking occurs in the operator tree in reverse preorder, but no easy relationship exists between reverse preorder and endorder; hence operator nodes in the operator tree point to the state of the world which is current when the node is created.

During backtracking, alternate paths are taken most frequently on the choice of operators that might realize a subtask, as has just been exemplified. The second most frequent mode of alternate paths makes use of the alternate bindings in the ALIS variable of the operator-procedure, as discussed in section 2.1. The least frequent backtracking mode consists of permuting the subtasks in a hierarchical group, as will be discussed in section 3. (Another ordering of the three backtracking modes might have been chosen.)

3. Hierarchies of Subtasks.

In [1], the robot can turn on a lightswitch if it is on some box, BOX1, which is close to the lightswitch. So two of the preconditions of the operator turnonlight would be: (ON ROBOT BOX1) (NEXTTO BOX1 LIGHTSWITCH). It is obvious that the second precondition should be satisfied first, then the other one. In this way, we arrive at the concept of hierarchies of subtasks: if several subtasks must be accomplished, it is safe to do some before others. By safe we mean that if subtasks are solved in the order of their hierarchy, then the task can be solved. There may also be solutions which violate the hierarchy, and some of these may be "better" --for example, requiring fewer steps--but our aim here is to obtain a solution in a reasonable time. An optimizing post-processor which tries to improve on an already existing solution is nearing completion.

The hierarchy of subtasks is also connected with the intuitive idea of freedom: if the robot first pushes a box to some place, it can be presumed that she subsequently can, i.e., still has the freedom to, climb on whatever (or whomever) she wants. On the other hand, if she first climbs on the box, she usually has no freedom left to move the box. Similarly, since turning on the light requires moving a box, the subtask (STATUS lightswitch ON) will have a higher rank in the hierarchy than (NEXTTO box something). If we assume that boxes remain boxes, i.e., cannot be burnt, then unchangeable subtasks, such as (TYPE BOX1 BOX), have the highest rank. Typically, the robot's position has the lowest rank.

In all the worlds that we considered, a static hierarchy could be found; i.e. subtasks had a hierarchical rank independent of the current and desired state of the world. In sufficiently complex worlds, it might not be feasible to find a static hierarchy for the possible subtasks.

As an example, the hierarchy of subtasks for the world of [1] (see Table 1 and Figure 1 for the results) would be:

Rank 0: (ON x y) (ONFLOOR).
Rank 1: (ATROBOT x) (NEXTTO ROBOT x).
Rank 2: (INROOM ROBOT x).
Rank 3: (NEXTTO x y) (AT x y).
Rank 4: (STATUS x ON) (STATUS x OFF).
Rank 5: all the unchangeable subtasks.

Presently, we are perfecting a heuristic program which will derive the hierarchy of subtasks from the operators of the robot. The same algorithm also discovers which operators are relevant to achieve some subtasks: for instance (PUSH OBJ1 OBJ2) and (PUSH OBJ2 OBJ1) to achieve (NEXTTO OBJ1 OBJ2) in the example of section 2.2. However, to allow additional experimentation (see section 7) the information on hierarchies and on relevant operators were input to the system. If the above mentioned heuristic program holds up to its promises, the only input to LAWALY beyond the description of the world, operators and tasks is a maze-running algorithm. This algorithm is a bi-directional search procedure which finds a (shortest) path between two

points in a maze (of rooms, doors, elevators, etc.). Without her maze-running capability, LAWALY has no sense of direction; with it, she has at least some indications on which way to go.

4. The Use of Hierarchies.

When a task is given to LAWALY, the subtasks which specify the goal are partitioned in sets of tasks having the same hierarchical rank. These sets are named hierarchical groups. For example, in task f, Table and Figure 1, three of these sets are obtained, which are in decreasing rank:
highest rank: (STATUS LIGHTSWITCH1 ON).
next rank: (NEXTTO BOX1 DOOR1) (NEXTTO BOX2 DOOR1) (NEXTTO BOX3 LIGHTSWITCH1).
lowest rank: (ATROBOT G).
LAWALY will first try to solve the tasks in the highest ranked hierarchical group, then in the next highest, etc. Once the subtasks in a hierarchical group are solved, the operator tree and the list of worlds are erased, thereby reclaiming memory. No backtracking occurs from one hierarchical group into one ranked higher, since, by definition, it is assumed that a lower ranked task can be accomplished (at least in some way) without disturbing a higher ranked task. Within a hierarchical group, we do not know in which order to try the subtasks, and if necessary all permutations of the subtasks are tried. We shall see an example in section 4.1.

4.1 Backtracking within a Hierarchical Group.

In the world of [1], as well as in the condensed version given in section 2.2, the goal state: (NEXTTO BOX1 BOX2) (NEXTTO BOX2 BOX3) (NEXTTO BOX3 BOX1), which is a more symmetric description of the state "the three boxes are next to each other", is not achievable. A disproof of this goal, i.e. a proof that there is no possible sequence of operators which leads from the initial to the goal state, necessitates much additional machinery[9] and is beyond the scope of this paper. LAWALY does determine, in 47.8 seconds (see task d, Table 1) that she cannot find a solution. Her failure does not mean that the goal is indeed unachievable, although it hints this. (See section 8 for a solvable task which LAWALY fails to solve.)

To illustrate the use of permutations of subtasks in a hierarchical group, we turn to LAWALY's attempt at solving the goal of the boxes symmetrically next to each other. We abbreviate (NEXTTO BOXi BOXj) as NEXTij. LAWALY first tries to solve the task in the order NEXT12 NEXT23 NEXT31. She does obtain NEXT12 NEXT23 as before (section 2.2). To obtain NEXT31, she can do (PUSH BOX3 BOX1) or (PUSH BOX1 BOX3), but either operation would delete one of the already achieved subtasks. Hence backtracking occurs: there is none possible with the ALIS, and backtracking on the choice of operators eventually fails too. Backtracking is non-destructive, and when it reaches the initial state of the world, the next alternate permutation of the hierarchical group is selected: NEXT12 NEXT31 NEXT23. (Successive permutations are selected so that the right-most parts of the permutation change most often.) Instead of trying the whole new permutation from the initial state of the world, LAWALY notices that she has backed from an unsuccessful solution which does however achieve the first subtask of the new permutation: NEXT12. So processing hops to the state in which NEXT12 is satisfied, thereby saving some computation. As the solution is continued, the old paths are destroyed. In effect, LAWALY learns from her failures. More than just knowing that she fails, she keeps the information contained in the failure, and, as this example shows, can reuse large parts of it in further problem-solving.

4.2 Observations on Solution Times.

With no backtracking, the time needed by LAWALY for a solution will grow about linearly with the number of steps in the solution. The exact time per step will depend on the size of the dynamic world (see section 6.3), the size of the set of preconditions, etc. When backtracking occurs (as in problems b, d, e, m), the overhead per node increases. Since backtracking is limited by the use of hierarchies (see section 4), total solution time often grows only about linearly with the length in steps of the solution.

5. Storage Structure of the World.

The processes described so far are independent of the storage structures used for the worlds. The results in the next two sections were obtained with the following storage structure:
- the static world (i.e. the parts of the world which are unchangeable) is stored using property lists. As a result, significantly larger static worlds barely affect LAWALY's performance, as shown in section 6.3.
- the dynamic world is stored in a list. During the processing of a hierarchical group, each dynamic world is kept separately. It is seen that this storage structure is not very efficient, and that much processing (additions, deletions, membership tests, etc.) will be slow. Average processing time by node does indeed increase with the size of the dynamic world. Our main objective has been to work on the more serious problems of procedure generation and search. Various alternative storage structures are presently investigated to improve the efficiency of the system in space and time.

6. Comparison with the STRIPS Programs.

LAWALY was asked to solve all the tasks that STRIPS solved (as available from all the documents to which we had access) and we threw in some others in the same worlds. The same version of LAWALY was used for all the runs (and for those of the next section too), while at least two versions of STRIPS were used: one of these uses MACROPs, and the other one does not. A MACROP (macro-operator) is the generalization of a task, so that a single new macro-operator replaces several original elementary operators. To use MACROPs, a sequence of related consecutive tasks must be given to STRIPS.

The times given for the STRIPS solutions are in partially compiled LISP on the PDP-10, excluding garbage collection. LAWALY's times are in interpreted LISP on the CDC-6600 and include garbage collection. The 6600 is estimated to be about 8 times faster than the PDP-10[10]; however the gain in speed due to compilation and the exclusion of garbage collection times make the two sets of times about directly comparable as given.

6.1 Comparison with STRIPS of [1].

Figure 1 shows the tasks; the performances are summarized in Table 1. Of the 24 predicates in the initial world, 16 are static. Seven operators are used. The additional node in the search tree of LAWALY corresponds to a call to the maze solving routine. (The same comment holds for some of the other examples.) Task b involves some backtracking, hence the longer average time per node in the search tree.

Tasks d, e and f were additional tasks given to LAWALY. Task d is the impossible task (in the axiomatization) of the three boxes being symmetrically next to each other. Task e attains a physically impossible goal --the robot being in two different places at the same time--which can be reached in the axiomatization of [1]. STRIPS would not be able to solve this problem due to some built-in heuristic (R. Fikes, personal communication). Task f requires 15 steps: no task requires a longer solution in this world.

6.2 Comparison with STRIPS of [2].

Figure 2 shows the tasks; the performances are summa-

rized in Table 2. Of the 66 predicates in the initial world, 47 are static. Seven operators are used. Task i is another physically impossible task which has a solution in the axiomatization of [2].

6.3 Comparison with STRIPS of [3].

Figure 3 shows the tasks; the performances are summarized in Table 3. Of the 100 predicates in the initial world, 75 are static. Nine operators are used. Results for both versions of STRIPS, with or without MACROPs, are given. STRIPS alone cannot solve task n. Tasks requiring six steps (k and m) appear to be in the upper range of its capabilities.

The world in [3] included an additional 67 static predicates, none of which were needed by LAWALY. With the entire 167 predicates, LAWALY's solution time increases by an insignificant 0.26%. Further augmenting the size of the world to a total of 528 predicates, increases LAWALY's original time by 7.2%; however, at least 50% of this increase can be attributed to one additional garbage collection. Hence we can conclude that LAWALY's performance is only marginally affected by the size of the static world.

We note that in all the solvable tasks in Tables 1, 2 and 3, LAWALY always found a shortest solution. In general, LAWALY does not always find a shortest solution.

7. Some Harder Problems Solved by LAWALY.

The tasks described in section 6 and its subsections are no challenge to LAWALY. Since the world of [1] had no tasks requiring more than 15 steps, and since the other two worlds were not much larger, we have built up larger worlds in which fairly hard tasks can be given. LAWALY was made to operate as a robot-janitor; she dries terraria with hot winds, or waters them with a pail that she must fill at a faucet; she empties trash baskets, sweeps floors with brooms and uses a dustbin in the process; she can carry objects, but must put them down to close doors. Moreover, she turns lights on (and also off), blocks doors with boxes and climbs onto boxes. There are 120 predicates in the initial world, of which 75 are static, and 26 operators. Subtasks are divided into nine priority ranks, and the ranks have been changed to study the effect of changes in the hierarchy of subtasks.

Figure 5a shows the initial "superworld". Figure 5b shows the first final state considered. The task was run with two different hierarchies. When the rank of a door being blocked is smaller than something being in a room, the task is solved in 198 steps. When the ranks are reversed for these two subtasks, all else unchanged, the task is solved in 209 steps. The tasks required 348 and 372 seconds, respectively, including 24 and 25 garbage collections. The time to generate the procedures was about 20 seconds, and is not included in the above times, since procedure generation is performed only once for the given set of operators.

With the same initial world, but Figure 5c as the final world, and the first of the hierarchies mentioned above, the solution found had 275 steps, and took 433 seconds. Upper limits on LAWALY's capabilities would be caused by memory limits, since, as mentioned, the internal representation of the world is inefficient, and ... economic considerations!

8. Example of a Failure by LAWALY.

We shall illustrate a case of a problem which is solvable, yet for which LAWALY does not find a solution. The task is illustrated in figure 6. The initial state could be axiomatized as INROOM(ROBOT A) CLOSED(DOOR) INROOM(BOX B), while the final state is: CLOSED(DOOR) NEXTTO(ROBOT BOX). LAWALY may decide to work first on the CLOSED(DOOR) condition, or first on the NEXTTO(ROBOT BOX) condition.

Consider the first case. LAWALY finds the door already closed in the initial state; hence she wants to obtain the NEXTTO BOX condition. To do that, she must enter Room B, thereby going through the DOOR. But that would mean opening the DOOR, and hence undoing what she has already achieved --CLOSED(DOOR)-- and so she decides to try to permute the goal subtasks. To be NEXTTO(ROBOT BOX), she goes to DOOR, opens it, goes through it, and then goes NEXTTO BOX. At that point, she realizes that she must still CLOSE the DOOR. However, that would make her undo something she wanted and had already achieved, namely NEXTTO(ROBOT BOX), so she quits, having failed.

The reason for LAWALY's failure is apparent: once she has focused attention on one subtask, she does not switch to another one until she either succeeds or fails to achieve the subtask. Her stubbornness is the cause of her downfall.

Perhaps LAWALY should not be blamed too much! She solves the task without difficulties if the specification of the final state includes INROOM(ROBOT B), or if this further specification is added by some (rather trivial) "transitivity of location" program.

9. The Advantages of Procedure Generation.

The goal-oriented programming languages[6,7,8] were designed, in part, to facilitate the writing of robot planners. We can expect that, for a long time, programs such as LAWALY that write procedures will often be less versatile than human beings. On the other hand, when available, programs such as LAWALY offer a measure of consistency and a lack of errors which is missing from programs produced by humans. Human programmers will often be tempted to take shortcuts, and may introduce bugs in their interpretations of the world.

A typical example of what happens--selected because it is the only documented case at our disposal--is a set of programs in QA4[7,8] to solve the robot problems in [1]. A shortcut is taken by treating the parameterless predicate (ONFLOOR) as a boolean flag. As a result, unintentional consequences creep into the QA4 programs: the robot cannot push two boxes next to each other if it starts on a box, since it never "thinks" of climbing off the box. The introduction of bugs is illustrated by the turnonlight operator: in [1], the robot must climb on BOX1 next to the lightswitch to turn the light on; in [7,8] the robot must have a box next to the lightswitch (not necessarily BOX1) and then climb onto BOX1, independently of the location of that box.

10. Learning.

As in [3], our robot system might improve her speed by building macro operators that combine a fixed sequence of operators. Such macros must be selected with care, since an indiscriminate generation of new operators could only lead to a cluttering of memory, additional choices to achieve a subgoal, and probably to costly reorganizations of the code generated for the operators.

Our approach to robot planning can be viewed as a form of learning: the robot studies her own environment and capabilities, and learns to interact efficiently with it. We can call this type of learning: procedural learning, and contrast it with statistical learning --where improvement in performance results from changes in parameters--and structural learning--where improvement in performance results from the building and modification of structures--. Many works in pattern recognition and the checker-playing program of Samuel are examples of statistical learning. The generation of MACROPs may be considered a form of structural learning; other examples are [11] and [12].

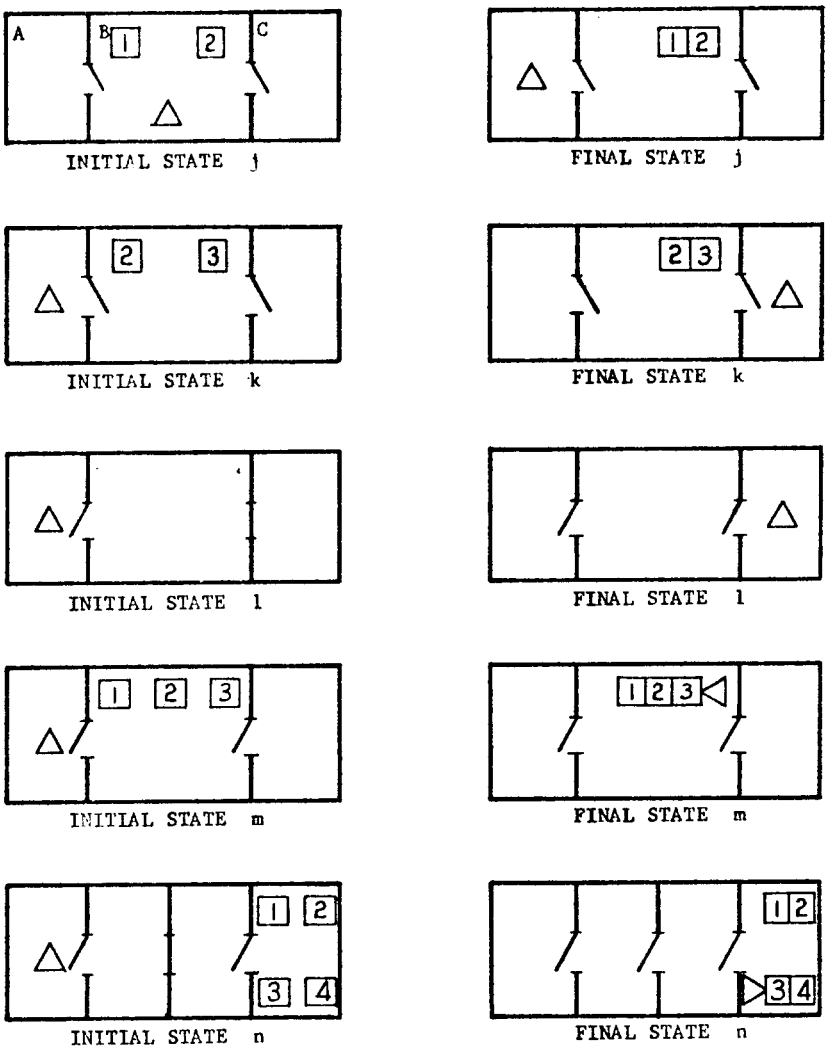


Figure 3.

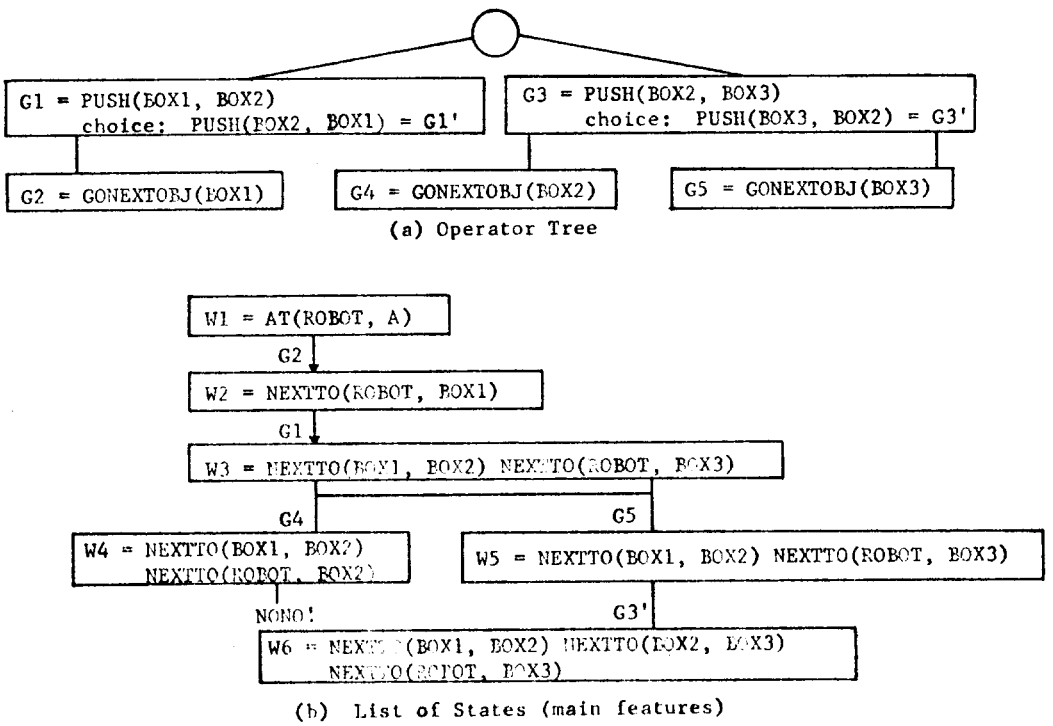
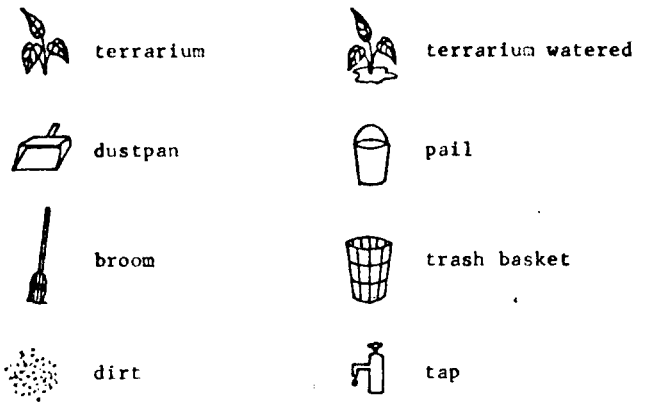
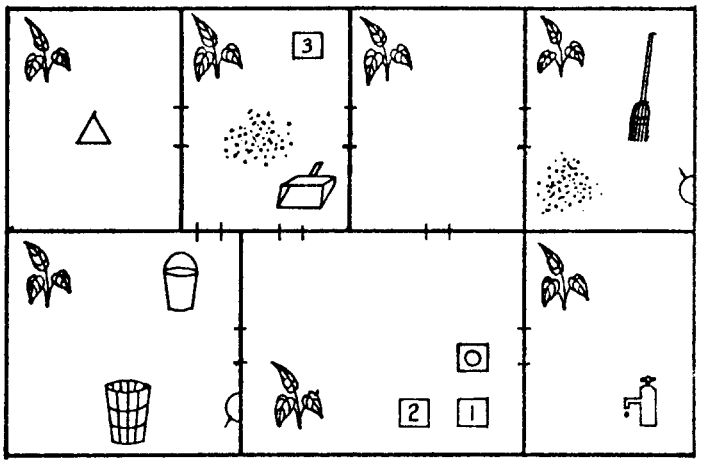


Figure 4. Proof of NEXTTO(BOX1, BOX2) ∧ NEXTTO(BOX2, BOX3).

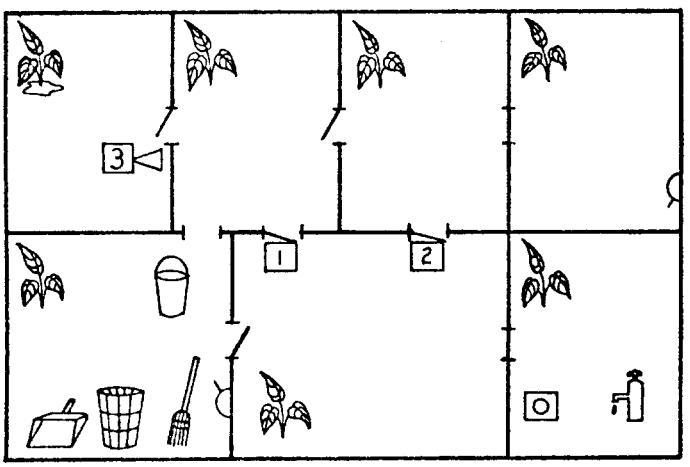


LEGEND



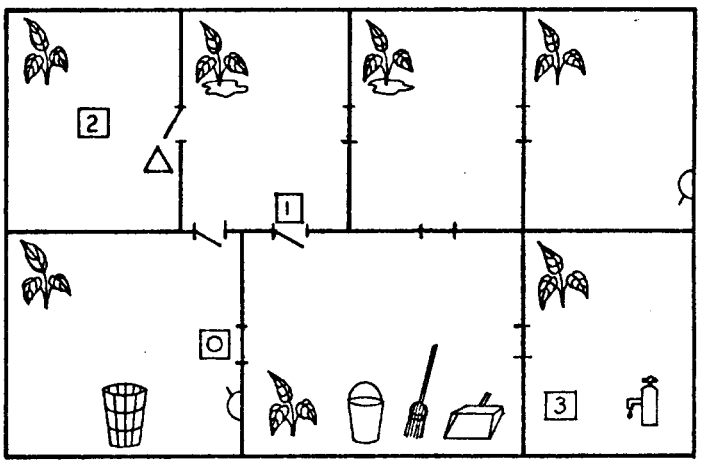
INITIAL STATE - SUPER WORLD

Figure 5a.



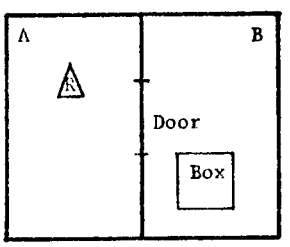
FINAL STATE - SUPER PROBLEM #1

Figure 5b.

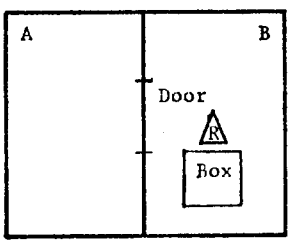


FINAL STATE - SUPER PROBLEM #2

Figure 5c.



INITIAL



FINAL

Figure 6.

	TASK	Theorem proving time in seconds		Number of operator applications				Time per node in the search tree in seconds		Number of nodes in LAWALY's initial world
				On solution path		In search tree				
		STRIPS	LAWALY	STRIPS	LAWALY	STRIPS	LAWALY	STRIPS	LAWALY	
a	Turn on the lightswitch	46.5	1.627	6	4	6	4	7.75	0.407	24
b	Push the three boxes together	92.5	4.095	4	4	6	5	15.42	0.819	24
c	Go to a location in farthest room	103.0	2.625	5	5	5	6	20.6	0.438	24
	Some other problems given to LAWALY in the same world.									
d	Push the three boxes together symmetrically (Impossible task in model)	--	47.796	--	no soln. found	--	50	--	0.956	24
e	Go next to box 1 and box 2 without pushing them together (physically impossible)	--	3.114	--	3	--	3	--	1.038	24
f	Turn on the lightswitch, push box 1 and box 2 next to door 1, push box 3 next to the lightswitch and go to a location in another room	--	10.268	--	15	--	16	--	0.642	24

Table 1. Some tasks in the robot world of (1).

g	Block door FG in room F with box 1	not known	6.228	not known	5	5	6	not known	1.038	66
h	Unblock door FG in room F	not known	5.546	not known	5	5*	6	not known	0.924	66
	Another problem given to LAWALY in the same world.									
i	Block doors FG and BF with box 1 in room F (physically impossible)	--	6.871	--	5	--	6	--	1.145	66

Table 2. Some tasks in the robot world of (2).

j	Push box 1 next to box 2 and go into room A	125.0 125.0*	7.366	4 4*	4	10 10*	5	12.5 12.5*	1.473	100
k	Push box 2 next to box 3 and go to room C	494.0 142.0*	10.589	6 6*	6	33 9*	8	14.97 15.78*	1.324	100
l	Go into room C	352.0 318.0*	7.710	5 5*	5	22 14*	6	16.00 22.71*	1.285	100
m	Push three boxes together	746.0 180.0*	13.859	7 6*	6	51 9*	8	14.63 20.00*	1.732	100
n	Push box 1 next to box 2 and push box 3 next to box 4	fails 349.0*	18.60	fails 11*	11	fails 14*	12	fails 24.93*	1.55	100

Notes: Unstarred items are for STRIPS without MACROPs.
Starred items are for STRIPS with MACROPs.

Table 3. Some tasks in the robot world of (3).