THE ROLE OF PREPROCESSING IN PROBLEM SOLVING SYSTEMS

"An Ounce of Reflection is Worth a Pound of Backtracking"

Clive Dawson and Laurent Siklossy
Dept. of Computer Sciences
University of Texas
Austin, Texas 78712 USA

0» Abst ract

We describe REFLECT, a problem solver in simulated robot worlds. REFLECT has successfully solved a variety of non-linear problems deseribed *as a* conjunction of subtasks, including most of those found in the literature- First, during a preprocessing phase, the system explores its capabilities in the environment it is presented with. Various intrinsie properties of the given operators *are* inferred, such as the unattainabillty of certain conjunctive tasks. Macro operators are built, again considering only the given operators. REFLECT is then ready to accept problems, and proceeds by conducting backward heuristie search on a variation of a regular state-space graph termed a Goal-Kernel graph. Attention is not focused on any one subtask in particular, so that a global view of the problem is always maintained.

1 Int reduction

The literature on solving simulated robot problems, represented as a conjunction of subgoals G|, G_2, •••» G which must be simultaneously satisfied in the final state, has become extensive. A review can be found in [Sikl6ssy-75]. One of the principal difficulties in solving such problems has been called non-linearity, meaning that the problem cannot be solved by considering each subtask successively in some order $G_p/j_)$, $G_{p(2)}$, •••, $^{G}p(n)$ where P *is a* permutation of (l,«»*,n). As an extreme case, we can consider a solvable problem consisting of only two subgoals, Gj & $G_{2\#}$ A solution to the problem may be unattainable if the subgoals arc considered successively in the order Gj G_2 or in the order C_2 G^, such that once a subgoal is achieved it is protected (i.e. its attainment cannot be destroyed during subsequent problem solving).

One of the simplest examples, taken from [Sikl6ssy & Roach-73], will illustrate the difficulty. The world contains two rooms RMA and RMB which communicate through DOOR. DOOR, initially closed, can be opened or closed by the ROBOT, initially in RMA. To open or close a door, the ROBOT must be next to it. There is also a BOX, initially in RMB, which the ROBOT can go next to. Hence we have an INITIAL STATE of:

[(INROOM ROBOT RMA)(CLOSED DOOR)(INROOM BOX RMB)].

We wish to achieve a FINAL STATE of:

[(NEXTTO ROBOT BOX)(CLOSED DOOR)].

If we first achieve (NEXTTO ROBOT BOX), the DOOR will be left open. It must be closed to achieve (CLOSED DOOR). To close the door, the ROBOT must go next to DOOR, which would destroy the already achieved (NEXTTO ROBOT BOX). If, on the other hand, we first achieve the second subtask, (CLOSED DOOR), we need do nothing since that property is true in the initial state. However, protecting this subtask prevents us from achieving (NEXTTO ROBOT BOX).

In the next section, we review previous approaches to the design of problem solvers that attempt to solve non-linear tasks. All of these approaches seem to be characterized by the fact that the problem solver concentrates its efforts on certain subtasks in particular, thereby temporarily ignoring the other subtasks. Tt appears that these problem solvers wear "blinders", and have no good global view of the problem to be solved. By contrast, REFLECT maintains an overall view of the problem, favoring no subtask with its attention. At each step, the solution to any subtask can be advanced, irrespective of which subtask was advanced on the previous step, depending on what seems to advance the total solution the most. No achieved subgoal needs to be protected. REFLECT draws its power from the results of its preprocessing stage, which serves to analyze the inherent capabilities of the robot as specified in the operators.

During one phase of preprocessing, the system deduces that certain sets of states are unattainable, due to the simultaneous presence of incompatible assertions such as (OPEN DOOR) and (CLOSED DOOR) or (INROOM ROBOT RMA) and (INROOM ROBOT RMB). During another phase, macro operators are built. These are combinations of the given operators which naturally fit together, and allow one new macro-step to advance a solution by several original steps. We call these macro operators BIGOPs, not to be confused with MACROPs In [Fikes et.al.-72]. BIGOPs are based solely on the description of the original operators, whereas MACROPS are based on generalizations of particular solved problems. Therefore, BIGOPs represent the intrinsic properties of the problem domain, and do not depend on any judicious or accidental choice of problems.

Preprocessing is performed once for a domain (set of operators), and the results remain valid for any attempt at problem solving in that domain. Preprocessing is akin to what a good human problem solver would do when confronted with a new environment. Before solving a problem, he would take time to reflect upon the situation, examine the capabilities of his tools (operators), perhaps performing some small exploratory experiments.

REFLECT's operators are represented using a strictly declarative formalism. The results of

preprocessing, however, are used in a way similar to the strategic information found programmed into procedural representations of knowledge REFLECT, therefore, reduces the differences between the declarative and procedural approaches to problem solving by showing that declarative style operators are not quite as devoid of imperative content as many may think.

2. Previous Research

The conjunctive problem is easier if some permutation $G_D n)>$ ^n(2)·#*** $^{G}D(n)$ exists such that the problem can be solved left to right in the order given by the permutation of the subtasks. In [Fikes & Nilsson-71], some permutations of the initial ordering of the subtasks were attempted. In [Sikl6ssy & I)reu8si-73], a hierarchy (hand input or computer generated) re-ordered the subtasks so that after solving $G_{p(i)}$, there remained sufficient liberty to solve tasks $^{G}D(i+-i)$ f°r J>0* A similar hierarchy scheme was developed in [Sacerdoti-73] •

The problems become more difficult If Interferences between subtasks are such that no permutation as above exists. In [Sikl&ssy & Roach-74], collaboration between a problem solver [Sikl6ssy & Dreussi-73] and a disprover [Sikl6ssy & Roach-73] (which attempts to show that the problem has no solution) solves some of these problems. In [Tate-75, Sacerdoti-75, Waldinger-75], basically, independent solutions are obtained for each subtask. These independent solutions are then postprocessed, and an attempt is made to mesh them together. In [Warren-74], the problem is attempted left-to-right in the order given. When the next subtask G^ cannot be achieved without destroying a previously achieved subtask, and when no further way exists to achieve this previous subtask, an attempt is made to insert a solution for G^ in the sequence of operators that achieved G^, 0<i<k. Statements similar to our incompatible assertions (Sec. 3.2) are input by hand to limit, search space growth. In [Sussman-73], a procedure is built incrementally to solve a problem, and may include various modifications and patches on previous versions of the procedure. Non-linearity is also discussed in [Hewitt-75], where the applicability of backward search is noted.

The ubiquitous blocks world, made popular in [Winograd-71], has been used in most of these problem solvers. It should be mentioned that for simple stacking problems, such as those that have been considered, a simple hierarchy in the sense of [Sikl6ssy & Dreussi-73], exists: stack the blocks from bottom to top] More complicated block stacking problems, such as those solved by the specialized system described in [Fahlman-74], appear beyond the capabilities of any of the more general problem solvers mentioned.

In the next section, we choose a blocks problem to illustrate the behavior of REFLECT, mentioning en route the capabilities not directly touched on by the example.

3. A^ Detailed Example

In keeping with the tradition established by [7,10,17,18,19,20], we will first describe how REFLECT handles the 3-block problem. (See Sec. 4 for other sample problems.) The world consists of a table and three blocks. The initial configuration and the desired final state are as follows:

The following four operators are initially given to REFLECT:
(BP*Binding Preconditions, PR«Preconditions, DS-Delete Set, AS-Add Set)

```
OP: Pickup ($B)
BP: (TYPE BLOCK $B)
PR: (ONTABLE $B)(CLEAR $B)(HANDEMPTY)
DS: (ONTABLE $B)(CLEAR $B)(HANDEMPTY)
AS: (HOLDING $B)

OP: Putdown ($B)
BP: (TYPE BLOCK $B)
PR: (HOLDING $B)
DS: (HOLDING $)
AS: (ONTABLE $B)(CLEAR $B)(HANDEMPTY)

OP: Stack ($B1,$B2)
BP: (TYPE BLOCK $B1)(TYPE BLOCK $B2)
PR: (HOLDING $B1)(CLEAR $B2)
DS: (HOLDING $)(CLEAR $B2)
AS: (HANDEMPTY)(ON $B1 $B2)(CLEAR $B1)

OP: Unstack ($B1,$B2)
BP: (TYPE BLOCK $B1)(TYPE BLOCK $B2)(ON $B1 $B2)
PR: (HANDEMPTY)(CLEAR $BI)
DS: (ON $B1 $)(HANDEMPTY)(CLEAR $B1)
AS: (HOLDING $BI)(CLEAR $B1)
```

(Binding Preconditions are used to determine whether the operator is applicable by supplying the necessary bindings for the variables. They will never be set up as subgoals.)

The initial state of the world is described by the following set of predicates:

(TYPE BLOCK A)	(CLEAR B)	(ONTABLE A)
(TYPE BLOCK B)	(CLEAR C)	(ONTABLE B)
(TYPE BLOCK C)	(HANDEMPTY)	(ON C A)

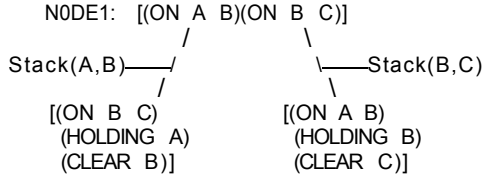
Our first attempt at the problem will be without the aid of BIGOPs, since this will serve to better illustrate the behavior of the problem solver itself. We will then reconsider the problem to illustrate the use of BIGOPs.

3.1 Backward Searching and Goal Kernels

Before considering the preprocessing stage, we will briefly jump ahead to the actual problem solving. As previously stated, REFLECT employs backward heuristic search on a goal-kernel graph. Unlike a state-space graph, in which each node

corresponds to a distinct state of the world, a goal-kernel graph is composed of nodes which correspond to sets of states. Each such set is characterized by the fact that it contains all states in which the goal associated with the node is satisfied. A node labeled [(ON A B)(ON B C)], for example, corresponds to the set of all states in which these two assertions hold. (ON A B) and (ON B C) are called elements of the goal kernel. This node will serve as a start node for the problem solver. The aim is to find a node (goal-kernel) whose every element is contained in the initial state of the world.

A node is expanded by applying operators in reverse. We reason this way: In order to arrive at some state in which (ON A B) and (ON B C) hold, we must have applied an operator such as Stack(A,B) or Stack(B,C). The application of irrelevant operators is automatically prevented, since they would not move us to a different goal-kernel. The goal-kernels for the new nodes can be generated by removing the assertions in the operator's ADD set, and adding the assertions in the operator's PRECONDITION set. We obtain the following graph:



The whole idea should be a little more, clear now. What we are saying is that given ANY state in which (ON B C), (HOLDING A), and (CLEAR B) are true, it Is always possible to apply the operator Stack(A,B) to achieve our final state in which (ON A B) and (ON B C) are true. Similarly, given ANY state in which (ON A B), (HOLDING B) and (CLEAR C) hold, we will always be able to apply Stack(B,C) to achieve our final state.

But wait! Is there even one state in which (ON A B), (HOLDING B) and (CLEAR C) can be simultaneously true? The answer is clearly no, and this is precisely what REFLECT has already deduced during preprocessing (Sec. 3.2). In this case, (HOLDING B) and (ON A B) form the offending pair of incompatible assertions (I.A.'s). When the node expander notices this, it refuses to consider Stack(B,C) as a possible operator. The result of the expansion, then, is a single new node, N0DE2, which makes our next choice very easy.

3.2 How I. A.'s are used in A. I.

We now back up to consider how preprocessing is able to examine the operators and deduce pairs of incompatible assertions (I.A.'S). (Our discussion will, for the moment, not touch on larger sets of I.A.'s, such as triplets or quadruplets.) Given the four operators for the blocks world, REFLECT considers the five changeable predicates which appear: HANDEMPTY, HOLDING, ONTABLE, ON, and CLEAR. (TYPE is static and therefore not considered.) Taking them one pair at a time, REFLECT conducts what could be termed "mini-experiments" to try and make both predicates true at the same time.

Some experiments are very easy. In particular, REFLECT assumes instant success if the two predicates appear together on the ADD set of some operator. Examples of this are (ONTABLE $X) and (CLEAP $X), (HANDEMPTY) and (ON $X $Y), as well as (HANDEMPTY) and (CLEAR $X). Other experiments are seen to fail rather easily. For example, (HANDEMPTY) and (HOLDING $X) can never be true simultaneously, since for every ADD set in which one appears, the other is found in the corresponding DELETE set.

These experiments are actually abstract problem solving sessions similar in structure to the example we are now considering. The main difference is that we do not have an initial world state, nor do we have bindings for the variables. Handling the variables can be tricky. The two-variable predicate (ON $X $Y), for example, is actually considered twice; first to deduce properties for variable $X and second to deduce them for variable $Y. Predicates are even paired with themselves. This allows us to deduce, for example, that (HOLDING A) and (HOLDING B) cannot be simultaneously true. We have developed a special notation to represent this, similar in spirit to the various prefix characters used by PLANNER, CONNIVER, and QAA. Given that $X represents a variable, #X denotes a variable which can accept any binding except the current binding of $X.

As REFLECT conducts the mini-experiments, it imposes a bound on the "depth" of the graph generated. If the bound is ever exceeded, the system abandons that particular pair and goes on to the next. In this way, the "obvious" I.A.'s are found quickly. These can then be used on subsequent passes to prove further pairs of incompatibilities. For example, in an experiment with PI and P2, the system may achieve PI; the achievement of P2, however, is found to require P3, which has previously been proven to be incompatible with PI. If a similar situation is found when trying the problem in reverse, then PI and P2 have been proved incompatible. When a pass produces no new results, the depth bound is increased, and REFLECT tries again. This process continues until all experiments have concluded or until some maximum depth is reached.

It is, of course, possible to construct a world in which it will take an arbitrarily large amount of processing to deduce all possible pairs of I.A.'s. If an I.A. pair is not detected, the search space may become larger due to less effective pruning, but this does not affect the attainability of the solution.

It is also possible to "sabotage" the deduction procedure by introducing what we term a "Garden of Eden" initial state. Consider a situation where REFLECT has just completed preprocessing, having proved that (HOLDING A) and (HOLDING B) can never be simultaneously true. It now smugly asks for its first problem, together with the initial state of the world. Sure enough, for some mysterious reason the initial state just as smugly asserts (HOLDING A) and (HOLDING B)!

This is an example of a "Garden of Eden" state, namely one which could not have been arrived at by any sequence of operators. In many cases, these states follow their historical counterpart and don't survive very long, (Note that any application of Putdown($B) will delete all HOLDING assertions, not just (HOLDING $B).)

REFLECT produces the following incompatible assertions for this version of the blocks world:

```
[(HANDEMPTY)(HOLDING $X)] [(ONTABLE $X)(ON $X $Y)]
[(HOLDING $X)(HOLDING #X)]  [(ON $X $Y)(ON $X #Y)]
[(HOLDING $X)(ON $X $Y)]    [(ON $X $Y)(CLEAR $Y)]
[(HOLDING $X)(ONTABLE $X)]  [(ON $X $Y)(ON #X $Y)]
[(HOLDING $X)(CLEAR $X)]    [(ON $X $Y)(ON $Y $X)]
```

3.3 Ancestors, troublemakers, and other details

We return now to the 3-block problem, where the next step is to expand N0DE2, whose goal kernel is [(HOLDING A) (CLEAR B) (ON B C)]. REFLECT first focuses on the subgoal (HOLDING A). N0DE3 is produced by reverse application of the operator Pickup(A). The goal kernel [(ONTABLE A) (CLEAR A) (HANDEMPTY) (CLEAR B) (ON B C)] is obtained by replacing (HOLDING A) with the preconditions of Pickup(A). When REFLECT considers the subgoal (ON B C), the assertion (HOLDING B) required by Stack(B C) is found to be incompatible with (HOLDING A), which is already in the goal kernel. The final subgoal in N0DE2, (CLEAR B), is already true in the initial state of the world, so it is not considered. (It Is not always possible to ignore true assertions in this manner, as we shall see.) The graph now looks like this:

```
          N0DE1: [(ON A B)(ON B C)]
Stack(A,B)--|
          N0DE2: [(HOLDING A)(CLEAR B)(ON B C)]
Pickup(A)---|
          N0DE3: [(ONTABLE A)(CLEAR A)(ON B C)
                  (HANDEMPTY)(CLEAR B)]
```

The expansion of N0DE3 will illustrate various interesting rules used by REFLECT. We will describe this expansion by presenting a trace of REFLECT's activity.

```
EXPANDING N0DE3...
 TASK: (CLEAR A)
  OPERATOR:  Putdown(A)
   REJECT-ANCESTOR: N0DE2                          [a]
  OPERATOR:  Unstack(C,A)
   REJECT-DELETE: (HANDEMPTY)                      [b]
   REJECT-I.A.: (CLEAR C)(ON B C)
 TASK: (ON B C)
  OPERATOR:  Stack(B,C)
   NEW NODE: N0DE4 [(HOLDING B)(CLEAR C)
                    (ONTABLE A)(CLEAR A)]
 TASK: (HANDEMPTY)
  OPERATOR:  Putdown(A)
   REJECT-ANCESTOR: N0DE2
  OPERATOR:  Putdown(B)
   REJECT-I.A.: (HOLDING B)(ON B C)
  OPERATOR:  Putdown(C)
   REJECT-I.A.: (HOLDING C)(ON B C)
```

On line [a], we notice how REFLECT has avoided a loop by Invoking the ancestor rule* This rule states that if the goal kernel under consideration is a superset of one of its ancestors, we should discard It, since any operator applicable in the new goal kernel would have been equally applicable in the ancestor.

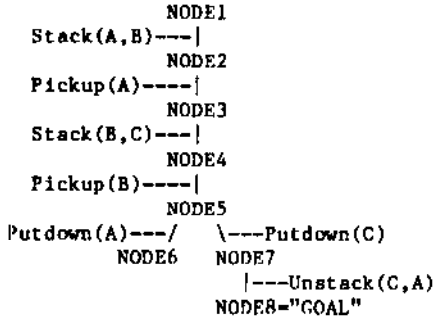
On line [b], REFLECT is trying to achieve (CLEAR A) using the operator Unstack(C,A). It rejects this, however, because even though Unstack(C,A) makes (CLEAR A) true, another element of the goal kernel, (HANDEMPTY), is made false! The applicable rule, known as the deletion rule, states that no operator can ever be applied whose DELETE set contains a member of the current goal kernel. This little failure has had an important consequence, however. Because of it, (HANDEMPTY) Is branded as "troublesome". This label is given to assertions which hold in the initial state of the world, but which may have to be temporarily violated since they are interfering with the achievement of another subgoal. A troublesome assertion will always be considered during expansion because, in our backward searching, this point might correspond to the time when the assertion (after having been temporarily false) needs to be reachieved. Even after Unstack(C,A) is rejected, REFLECT checks for I.A.'s, since this could point to other "troublesome" assertions. The system now considers the next subgoal, (ON B C), and does succeed in producing a new node, using the operator Stack(B,C). Finally, (HANDEMPTY) (already true but labeled "troublesome"), is considered but produces no new nodes.

Expansion of NODEA also produces a single new node when REFLECT applies Pickup(B) to achieve (HOLDING B). All other applicable operators lead to I.A.'s. When NODE5 is expanded, REFLECT generates two new nodes, as shown below:

```
                 .
                 .
                 |
          N0DE3:[(ONTABLE A)(CLEAR A)(ON B C)
                 |    (HANDEMPTY)(CLEAR B)]
Stack(B,C)-----|
          N0DE4:[(HOLDING B)(CLEAR C)(ONTABLE A)
                 |    (CLEAR A)]
Pickup(B)------|
          N0DE5:[(ONTABLE B)(CLEAR B)(HANDEMPTY)
                 |    (CLEAR C)(ONTABLE A)(CLEAR A)]
                / \
Putdown(A)--/    \--Putdown(C)
           /      \
     N0DE6:        N0DE7:[(CLEAR A)
 [(HOLDING A)              (HOLDING C)
  (ONTABLE B)              (ONTABLE B)
  (CLEAR B)                (CLEAR B)
  (CLEAR C)]               (ONTABLE A)]
```

REFLECT finally has a choice to make! Since only one of N0DE6's subgoals is not satisfied in the inital state, whereas two of N0DE7's are not, the heuristic evaluation function reports that N0DE6's goal kernel is "closer" to the initial state of the world than N0DE7's is. The system therefore chooses N0DE6 to expand next. The only

applicable operator, Pickup(A), is rejected by the ancestor rule, however. REFLECT moves immediately to N0DE7, which was the correct choice. When applying the operator Unstack(C,A), the goal kernel [(HANDEMPTY) (CLEAR C) (ONTABLE B) (CLEAR B) (ONTABLE A)] is produced. Problem solving halts, since all elements of this goal kernel are true in the initial state. The final graph for this solution of the 3 blocks problem is:



The final plan is obtained by simply travelling back up to the root: Unstack(C,A), Putdown(C), Pickup(B), Stack(B,C), Pickup(A), Stack(A,B).

3.4 BIGOPs

We shall now consider the 3-block problem again. This time, we will make use of the results of another preprocessing phase: the construction of BIGOPs. The generation of these combined operators is accomplished by analyzing the inherent structure of the original operators provided to the system. The basic aim is to construct new operators which are made up of two or more primitive operators. This allows one arc in the goal kernel graph to take the place of several arcs, and results in as much as an order of magnitude decrease in the size of the search space. After the problem is solved, the solution is expanded back in terms of the original operators. Since BIGOPs can be thought of as generalized operators, the expansion process often is able to leave out certain unnecessary steps. For example, assume that we have preprocessed the operators in the STRIPS world of [Fikes Nilsson-71] to obtain a BIGOP called "CLIMBOFF & GOTO & OPEN & GOTHRUDOOR" which gives the robot the ability to go into another room. The presence of each primitive operator is designed to achieve one of the preconditions which will ultimately be needed in order to go through a door. At expansion time, however, it may be found that one or more of the conditions are already true. For example, the ROBOT may already be on the floor, or the door may already be open. In this case, final expansion would leave out the corresponding operators, since they would be essentially no-ops.

Given the four initial operators in our version of the blocks world, REFLECT begins by considering which pairs of operators can be applied successively. Of the sixteen possible ordered pairs, it discovers that conflicting preconditions prevent eight of them from ever being adjacent in a plan: Pickup-Pickup, Pickup-Unstack, Putdown-Putdown, Putdown-Stack, Stack-Putdown, Stack-Stack, Unstack-Pickup, and Unstack-Unstack. These conflicts are all caused by the I.A. pair [(HOLDING $X) (HANDEMPTY)]. The remaining eight operator pairs are candidates for BIGOPs. The method of construction is simple. Let Pk, Dk,, and A^ denote the Precondition set, Delete set, and Add set of operator $0^{p}k$« Operators OPi and OP., may be combined into a BIGOP where

$$P_k = P_j + (P_i - A_i)$$
$$D_k = D_j + (D_i - A_j)$$
$$A_k = (A_i - D_j) + A_j$$

(+ and - are set union and difference).

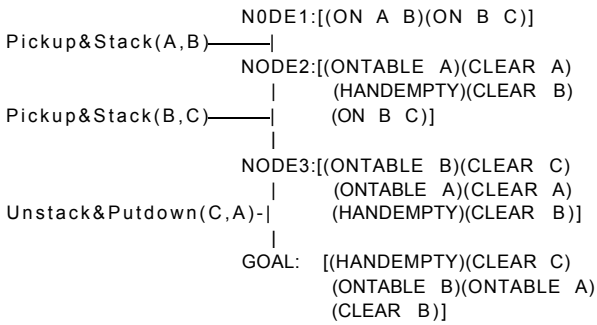
Even though the above operation is theoretically possible on all the remaining operator pairs, REFLECT uses some additional criteria in deciding whether the combination should exist or not. It is important that the variables correspond properly. In a Pickup&Stack operation, for example, it is clear that the block which is picked up is the same block which will be stacked. In a Stack&Unstack operation, it is not so clear. The blocks may be different or they may be one and the same. Finally, in a Putdown&Unstack, it is clear that they have to be different blocks. REFLECT requires that at least one variable in both operators correspond at all times. This has proved to be a good measure of the "naturalness" of combining two operators, and helps eliminate combinations of operators which are semantically unrelated. This criterion eliminates Putdown-Pickup, Putdown-Unstack, Stack-Pickup and Stack-Unstack. Finally, REFLECT checks for combinations which would essentially be no-ops, and manages to discard Pickup-Putdown. We are left with:

```
OP:  Pickup&Stack($Bl,$B2)
BP:  (TYPE BLOCK $B1)(TYPE BLOCK $B2)
PR:  (ONTABLE $B1)(CLEAR $B1)(HANDEMPTY)
     (CLEAR $B2)
DS:  (ONTABLE $B1)(CLEAR $B2)
AS:  (ON $B1 $B2)

OP:  Unstack&Stack($Bl,$B2,$B3)
BP:  (TYPE BLOCK $B1)(TYPE BLOCK $B2)
     (TYPE BLOCK $B3)(0N $B1 $B2)
PR:  (HANDEMPTY)(CLEAR $B1)(CLEAR $B3)
DS:  (ON $B1 $B2)(CLEAR $B3)
AS:  (ON $B1 $B3)(CLEAR $B2)

OP:  Unstack&Putdown($Bl,$B2)
BP:  (TYPE BLOCK $B1)(TYPE BLOCK $B2)(ON $B1 $B2)
PR:  (CLEAR $B1)(HANDEMPTY)
DS:  (ON $B1 $B2)
AS:  (CLEAR $B2)(ONTABLE $B1)
```

Let us now take another look at the 3-block problem given the presence of these BIGOPs. REFLECT is able to solve the problem with no wrong moves in less than one-third the time (1.7 8. vs. 5.9 s., interpreted LISP on a CDC6600) and produces the following graph:



4. Other sample problems

The figures at the end of this paper together with Table 1 show some typical linear and non-linear problems solved by REFLECT. Many have been chosen from the literature. The order in which the subtasks of a problem are given to REFLECT is irrelevant, while it may be crucial to other systems. Although we provide comparative execution times when available, comparisons on this basis are tenuous at best. As workers in this area know well, slight changes in the axiomatization of operators can cause large differences in the behavior of these systems.

5. Conclusions and Further Work

REFLECT has successfully solved problems typified by the samples given. These problems have often been solved only with difficulty by other problem solvers. The success of REFLECT can be attributed to a combination of preprocessing results and the goal-kernel backwards search, which has proved adequate for handling tasks up to several dozen steps long. However, the preprocessing heuristics currently in use are not sufficiently powerful to prevent a combinatorial explosion when REFLECT attempts tasks which are several hundred steps long (within the capabilities of LAWALY [12]). We can attribute this to several reasons. First, REFLECT has essentially put aside the linearity principle in order to maintain a global view during problem solving. If particular worlds are not strongly enough constrained to provide a good set of I.A.'s, then the lack of effective pruning becomes evident quickly. Linearity is, obviously, an extremely powerful heuristic when used appropriately. Rather than using other heuristics to overcome the problems of linearity, as almost all previous systems have done, it is our aim to develop heuristics that tell us when to take advantage of linearity. Along these lines, a hierarchy scheme implemented for backward search seems promising.

Another area which is being implemented is the extension of I.A. pairs to triplets, quadruplets, etc. An analysis of goal-kernels generated by the present system shows that I.A. triplets arise in substantial numbers, but go undetected. This means that the system is working on tasks which are doomed to failure. Detection of new I.A.'s during preprocessing could be carried as far as desired; there is presumably a point of diminishing returns, beyond which it would be more profitable to start the problem solving itself.

Finally, an area that needs a lot of work is the use of preprocessing techniques to generate good heuristic evaluation functions. REFLECT currently generates a very crude evaluation function during preprocessing by counting the average number of preconditions needed to achieve a given assertion. This could use substantial improvement.

6. Acknowledgements

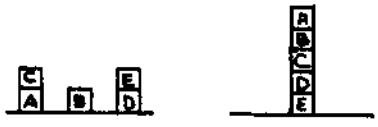
We would like to acknowledge the influence provided by many long and fruitful discussions on this subject with Daniel Chester, Joseph Dreussi, and John Roach.

7. References

1. S. Fahlman. "A Planning System for Robot Construction Tasks." Artificial Intelligence, 5, 1974.
2. R. E. Fikes. "Monitored Execution of Robot Plans produced by STRIPS." International Federation for Information Processing Congress 71, Ljubljana, 1971.
3. R. E. Fikes, P. E. Hart and N. J. Nilsson. "Learning and Executing Generalized Robot Plans." Artificial Intelligence, 3, 1972.
4. R. E. Fikes, P. E. Hart and N. J. Nilsson. "New Directions in Robot Problem Solving." In: D. Michie and B. Meltzer (Eds.) Machine Intelligence 7, Edinburgh University Press, Edinburgh, 1972.
5. R. E. Fikes and N. J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem-solving." Artificial Intelligence, 2, 189-208, 1971.
6. C. Green. "Application of Theorem-proving to Problem Solving." In: D. E. Walker and L. M. Norton (Eds.) Proc. International Joint Conference of Artificial Intelligence, 1969.
7. C. Hewitt. "How to Use What You Know." Fourth International Joint Conference on Artificial Intelligence. Tbilisi, 1975.
8. B. Raphael. "The Frame Problem in Problem-solving Systems." In: N. V. Findler and B. Meltzer (Eds.) Artificial Intelligence and Heuristic Programming, American Elsevier, New York, 1971.
9. E. Sacerdoti. "Planning in a Hierarchy of Abstraction Spaces." Third International Joint Conference on Artificial Intelligence, Palo Alto, 1973.
10. E. D. Sacerdoti. "The Non-Linear Nature of Plans." Fourth International Joint Conference on Artificial Intelligence, Tbilisi, 1975.
11. L. Siklóssy. "Some Issues in Problem-Solving in Modelled Worlds. Third International Congress of Cybernetics and Systems, World Organization of General Systems and Cybernetics, Bucharest, 1975 (printed 1977).
12. L. Siklóssy and J. Dreussi. "An Efficient Robot Planner which Generates its own Procedures." Third International Joint Conference on Artificial Intelligence, Palo Alto, California, 1973.

13. L. Siklóssy and J. Dreussi. "Simulation of Executing Robots in Uncertain Environments." Proc. 1974 National Computer Conference, Chicago, 1974.
14. L. Siklóssy and J. Roach. "Proving the Impossible is Impossible is Possible: Disproofs based on Hereditary Partitions." Third International Joint Conference on Artificial Intelligence, Palo Alto, 1973.
15. L. Siklóssy and J. Roach. "Collaborative Problem-solving between Optimistic and Pessimistic Problem Solvers." International Federation for Information Processing Congress 74, Stockholm, 1974.
16. L. Siklóssy and J. Roach. "Model Verification and Improvement using DISPROVER." Artificial Intelligence, 6, 1975.
17. G. J. Sussman. A Computational Model of Skill Acquisition. MIT AI-TR-297, Cambridge, Mass., 1973.
18. A. Tate. "Interacting Goals and Their Use." Fourth International Joint Conference on Artificial Intelligence, Tbilisi, 1975.
19. R. Waldinger. Achieving Several Goals Simultaneously. Artificial Intelligence Center, Technical Note 107, Stanford Research Inst., Menlo Park, 1975.
20. D. H. D. Warren. WARPLAN: A System for Generating Plans. Memo 76, Dept. of Computational Logic, U. of Edinburgh, 1974.
21. T. Winograd. Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. Project MAC TR-84, MIT, Cambridge, Mass., 1971.

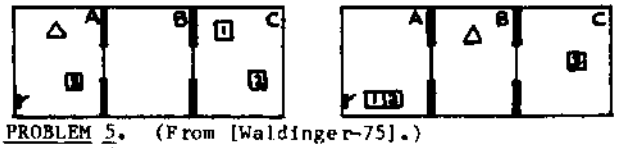
PROBLEM 1. (From [Warren-74].)
Goal: (ON A B)(ON B C)(ON C D)(ON D E)



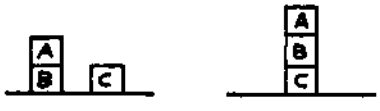
PROBLEM 2. (From [Siklóssy & Roach-73].)
Goal: (DOORCLOSED DOOR)(ON ROBOT BOX)

PROBLEM 3. (From [Siklóssy & Dreussi-73].)
Goal: (NEXTTO BOX1 BOX2)(NEXTTO BOX2 BOX3)
(NEXTTO BOX3 BOX1)
Note: This problem is impossible given the axiomatization.

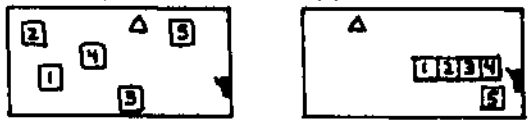
PROBLEM 4.
Goal: (INROOM BOX1 ROOMA)(INROOM BOX2 ROOMA)
(INROOM BOX3 ROOMC)(INROOM ROBOT ROOMB)
(NEXTTO BOX1 LITESW)(NEXTTO BOX1 BOX2)
(DOORCLOSED DOORAB)(DOORCLOSED DOORBC)



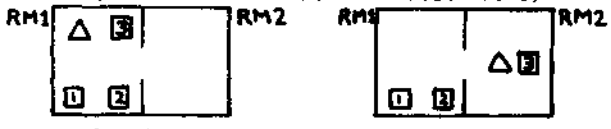
PROBLEM 5. (From [Waldinger-75].)
Goal: (ON A B)(ON B C)



PROBLEM 6.
Goal: (NEXTTO BOX1 BOX2)(NEXTTO BOX2 BOX3)(NEXTTO BOX3 BOX4)(NEXTTO BOX4 LSWIT)(NEXTTO BOX5 LSWIT)



PROBLEM 7. (From [Warren-74].)
Goal: (INROOM ROBOT RM2)(NEXTTO ROBOT BOX3)



PROBLEM 8. (From [Sacerdoti-75].)
Goal: (ON A B)(ON B C)(ON C D)

PROBLEM 9. (From [Sacerdoti-73].)
Goal: (NEXTTO BOX1 BOX2)(INROOM ROBOT RUN1)

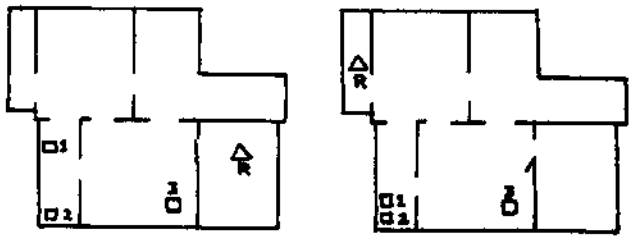


TABLE 1

Problem	1	2	3	4	5	6	7	8	9
Steps in solution	10	7[c]	0	34	6[e]	10	4	8	11
Nodes generated	8	5	1	41	5	7	5	7	12[g]
Nodes expanded	5	3	1	22	3	5	2	4	5
Time (seconds)[a]	7.93[b]	1.29	0.44[d]	86.87	1.56	9.99	1.14	4.85[f]	4.47[g]

[a] Interpreted LISP, CDC6600 (includes garbage collection).
[b] WARPLAN time: 52 sec. (PROLOG, IBM 360/67).
[c] LAWALY [12] failed to solve this problem.
[d] Time needed to determine impossibility. (LAWALY found no solution after 47 sec. DISPROVER [14] found a disproof in 7 sec.)
[e] [19] can solve as (ON B C)(ON A B) but not in opposite order.
[f] NOAH [10] time: 41 sec. (Compiled QLISP, DEC-10).
[g] ABSTRIPS [9] generated 60 nodes, using 328 sec. (Compiled LISP, DEC-10). STRIPS [5] time: >1640 sec.

THE CONTRACT NET: A FORMALISM FOR THE CONTROL OF DISTRIBUTED PROBLEM SOLVING*

Reid G. Smith
Heuristic Programming Project,
Department of Computer Science,
Stanford University, Stanford, California, 94305.

Distributed processing offers the potential for high speed, reliable computation, together with a means to handle applications that have a natural spatial distribution. In this paper, distributed processing is defined as processing that is characterized by physical decomposition of the processor into relatively independent processor nodes. Recent advances in LSI technology, expected to result in single silicon wafers with at least 10^6 active elements by 1981 [Noyce, 1976], indicate that it is reasonable to contemplate designs which incorporate large networks of single chip processor nodes.

In this paper we examine the control of problem solving in such an environment, where most information is local to a node, and relatively little information is shared by the complete network. Individual nodes correspond to "experts" which cooperate to complete a top level task (analogous to the "beings" of [Lenat, 1975]). The distributed processor is thus to be composed of a network of "loosely-coupled", asynchronous nodes, with distributed executive control, a flexible interconnection mechanism, and a minimum of centralized, shared resources. This puts the emphasis on "coarse grain" parallelism, in which individual nodes are primarily involved with computation (large kernel tasks), pausing only occasionally to communicate with other nodes.

The CONTRACT NET represents a formalism in which to express the control of problem solving in a distributed processor architecture. Individual tasks are dealt with as contracts. A node that generates a task broadcasts its existence to the other nodes in the net as a contract announcement, and acts as the contract manager for the duration of that contract. Bids are then received from potential contractors, which are simply other nodes in the net. An award is made to one node which assumes responsibility for the execution of that contract. Subcontracts may be let in turn as warranted by task size or a requirement for special expertise or data not in the possession of the contractor. When a contract has been executed, a report is transmitted to the contract manager.

Contracts may be announced via general broadcast, limited broadcast, or point-to-point communications mechanisms, depending on information about relevant contractors available to the contract manager. If, for example, a manager has knowledge about the location of particular data, then its contract announcement will be directed to the node(s) believed to possess that data, so that the complete network is not needlessly involved.

Contracting effectively distributes control throughout the network, thus allowing for flexibility and reliability. Decisions about what to do next are made as a result of relatively local considerations, between pairs of processors, although the nature of the announcement-bid-award sequence maintains an adequate global context; that is, the decision to bid on a particular contract is made on the basis of local knowledge (the task being processed in the node contemplating a bid), and global knowledge (other current contract announcements). The formalism also incorporates two way links between nodes that share responsibility for tasks (managers and contractors). The failure of a contractor is therefore not fatal, since a manager can re-announce a contract and recover from the failure.

A node in the CONTRACT NET is composed of a contract processor, management processor, communications interface, and local memory. The contract processor is responsible for the applications-related computation of the node. The management processor is responsible for network communications, contract management, bidding, and the management of the node itself. Individual nodes are not designated a priori as contract managers or contractors. Any node can take on either role, and during the course of problem solving, a particular node normally takes on both roles simultaneously for different contracts.

A contract is represented as a record structure with the following fields: NAME - the name of the contract, NODE - the name of another processor node associated with the contract, PRIORITY - a description of the "importance" of the contract, TASK - a description of the task to be performed, RESULTS - a description of the results obtained, and SUBCONTRACTS - a pointer to the list of subcontracts that have been generated from the contract.

Contracts are divided into two classes in a node: those for which the node acts as contractor, and those for which it acts as manager. The node field of a contract is filled accordingly - with the name of the contract manager in the first case, and with the name of the contractor in the second case. Subcontracts waiting for service are held at the node that generated them, with an empty node field.

The priority description is used by a management processor to establish a partial order over contracts to be announced, and by potential contractors to order contracts for the purpose of bidding. Similar descriptions are also used to order contractors for the purpose of awards. The concept of priority thus must be generalized over simple integer descriptions to include such (layered) descriptions of potentially arbitrary complexity, which include both applications-related and architecture-related information.

A task description also contains two types of information: the local context in which the task is to be executed, and the applications software required to execute it. This information is passed when a contract is awarded. Depending on the task, the required global context may be passed with the award, or further contracts may be let to obtain it. Software passed to a node for execution of a particular contract is retained for future use, and its presence has a favorable effect on the future bids of that node.

A SAIL simulation has been constructed to test the formalism. It accepts simple applications programs. and maps them onto a simulated distributed processor with a variable number of nodes. The simulation is being used to determine the costs associated with the formalism, in terms of both processor and communications overhead, and the decrease in computation time that can be expected for various applications. Distributed heuristic search is presently being examined in this way, and alternatives in distributed deduction will be examined in the near future.

* This work is supported by the Advanced Research Projects Agency under contract DAHC 15-73-C-0435. Computer facilities are provided by the SUMEX-AIM facility under National Institutes of Health grant RR-00785. The author is supported by the Research and Development Branch of the Department of National Defence of Canada. E. A. Feigenbaum, B. G. Buchanan, G. Wiederhold, K. G. Knutsen, and E. J. Gilbert provided a number of useful suggestions. F. K. Buelow was a valuable source of LSI technology data.

References

D. B. Lenat. "Beings: Knowledge As Interacting Experts," IJCAIU Proceedings, Tbilisi, USSR, September 1975, pp. 126-133.

R. N. Noyce, "From Relays to MPU's," Computer, Vol. 9, No. 12, December 1976, pp.26-29.