CONCEPTS AND METHODS FOR HEURISTIC SEARCH

Erik J Sandewall
Computer Sciences Department
Uppsala University
Uppsala, Sweden

0. Summary*

The transformation or derivation problem treated by most "problem-solving" programs is expressed in a formal notation, and various methods for "problem-solving" are reviewed. The conventional search tree is generalized into a search lattice which can accomodate multiple-input operators, e.g. resolution. The paper argues that descriptions of heuristic methods can be significantly compacted if a higher degree of formalization is used. This point is illustrated with two practical examples.

1. Introduction

This paper is an attempt at a survey and synthesis of past work on heuristic search methods. Following Feigenbaum and Feldman (1963), we define a heuristic method as a device which drastically limits search for solutions in large problem spaces. As can be seen from the list of references, much work on heuristic methods has been performed during the last few years.

Two approaches have been competing in this work. In his report on SIN**, Moses characterizes them as emphasizing generality and expertise, respectively. In the generality approach, one tries to write a general program which can solve all kinds of problems, provided only that (adequately phrased) information about the particular "problem environment" of each problem is provided. The "General Problem Solver" (sic!), DEDUCOM, and the Graph Traverser are examples of this approach.

In the approach that stresses expertise, one concentrates instead on writing a good program for solving problems in one given problem environment. SIN itself is a typical example of that approach, as are game-playing programs (Samuel's, Greenblatt's), and some programs which, according to rumour, are being used for industrial purposes.

* This research was supported in part by the Swedish Natural Science Research Council (contracts Dnr 2711-5, 2711-6) and by the Swedish Research Institute of National Defense (beställn. 714257, 715411).

** We shall refer to previous work in heuristics by its acronym and/or the author's name. For exact references, use tables at the end of the paper.

The advantages and disadvantages of each approach are obvious: generality has to be paid for by a decrease in program efficiency. An advantage with the generality approach is that one single heuristic method can quickly be put to use in a variety of problem domains.

It would seem, however, that methods which have been developed in one "expertise" program can be carried over to another problem environment and another program. The only problem is to pull out the abstract heuristic methods from the program descriptions, which are often quite technical and detailed.

One example of this will suffice. The SIN program contains an important heuristic, which Moses describes as follows: "The Edge heuristic is based on the Liouville theory of integration. In this theory it is shown that if a function is integrable in closed form, then the form of the integral can be deduced up to certain coefficients. A program which employs the Edge heuristic, called Edge, uses a simple analysis to guess at the form of the integral and then it attempts to obtain coefficients." (page 8). The Edge heuristic is further described on seventeen pages in chapter 5.

Unfortunately, the author fails to formulate this important heuristic method in abstract terms. Such an abstract formulation could e.g. run as follows: The purpose of the integration program is to start from a given, initial object, and to apply the right operators (from a given set of operators) in the right order, until the given object has been transformed into a given target set (i.e. the set of all expressions where the integral sign(s) have been eliminated). The Edge heuristic relies on information which is local to this particular problem environment, and which makes it possible to say, during the search of the solution tree, where in the target set we will eventually land. The Edge program utilizes this information to get a better estimate of the remaining "distance" to the target set from each node. - With such a description, it becomes clear that the same heuristic may well be applicable to other problem environments, in other expertise-oriented programs.

Abstract method descriptions, as outlined here, can of course not serve as substitutes for conventional ones. A concrete description, like the one Moses has given for SIN, will always be needed by the user of the program, or the researche who attempts to improve on previous work. By contrast, the abstract description is useful for the man who wants to carry over methods to other problem environments, and (of course) for the theoretician who, some time in the future, will attempt to build a mathematical theory of heuristics.

The morale is, therefore, that we need an abstract frame of reference, a set of concepts for describing and analysing heuristic methods. Such concepts would help in the dissemination of know-how; they would also make it possible to compare the efficiency of various methods and programs, expertise-oriented as well as generality-oriented.

In this report, we shall attempt to set up such a "frame of reference". In section 2, we formulate a general "transformation problem", and discuss some of its cases. In sections 3-4, various commonly used heuristic techniques are formulated and discussed. Since we argued, in section 2, that one-input and multiple-input operators must be carefully distinguished, we use section 5 to extend the conventional search tree into a search lattice. Our stock of concepts is tested in sections 6-7, where abstract descriptions of some well-known programs and heuristic methods are given.

2. Heuristic search: rules of the game

The problem environments for heuristic search methods always include a set P of objects and a set Q of operators on these objects. The following problem has often been studied, (see e.g. {Newell 1960c} and {Doran 1967a}), and has sometimes been referred to as the problem-solving problem:

Basic transformation problem.

Given an initial set $R \subset P$, a target set $M \subseteq P$, determine r in R and $q_1, q_2, \ldots q_k$ in Q such that

$$q_k(q_{k-1}(\ldots q_2(q_1(r)) \ldots))$$

exists and is a member of the target set M. We call this a transformation problem from R to M.

A method for solving basic transformation problems is called a heuristic search method if it searches the tree(s) of all possible operator applications, and the order in which the nodes of this tree are inspected, is governed in some ways by properties of the nodes which have already been created. Heuristic methods require, therefor, that the objects in P are known as symbolic expressions or otherwise have a non-trivial information content. They cannot simply be non-informative tokens of the form "p_i". The following variations to the basic transformation problem occur frequently:

Operators with several outputs.

The problem specification is changed as follows. Application of an operator can return a set of objects, rather than a single object. In the transformation process, each output of the operator must then be transformed into the target set.

Example: In analytic integration, the target set M consists of the set of all formulae where the integration sign does not occur. The rule

$$\int A + B\, dt = \int A\, dt + \int B\, dt$$

can be used as an operator q defined by

$$q(\int A + B\, dt) = \{ \int A\, dt, \int B\, dt\}$$

In other words, q tells us to integrate $A + B$ by integrating A and B separately. (The final task of joining together the solutions to those two integration problems with a + sign is a trivial matter).

Operators with several input.

The problem specification is changed as follows. Initially, each member of R is considered available. At each cycle of the solution process, one selects one operator q_j^i which requires i arguments, and i available objects

$$p_1, p_2, \ldots p_i \ . \quad \text{If} \quad q_j^i(p_1, p_2, \ldots p_i)$$

is defined, it is included among the available objects. Problem: find some available object which is also a member of M .

Exemple: This variation frequently occurs in "forward" logical inference, e.g. in the resulution logic environment. It has been common practice in heuristic research to consider the cases of several inputs or several outputs as trivial extensions of the one-input/one-output case. For example, the General Problem Solver is formulated in terms of one input operators, and then immediately applied to a problem environment where a two-input operator (Modus Ponens in forward proof) is essential. Similarly, Slagle's group have attempted to use their MULTIPLE program (which is designed for one-input, multiple-output operators) to the resolution logic environment, where the most important operator has two inputs and one output.

The fact that an operator requires several inputs can be "hidden" in various ways. In the case of Modus Ponens, which takes A and $A \supset B$ as inputs, one can say that the operator "essentially" takes $A \supset B$ as input, so that the merit of an $A \supset B$ formula determines whether the operator shall be applied or not. If the system decides to apply Modus Ponens to a formula $A \supset B$, it checks whether the formula A is available. If it is not, the output is "failure". - Another, and more general way of hiding multiple inputs is to consider the set of all available objects as a "higher level" object. Similarly, the operators are redefined to accept one higher level object as input, and to emit an incremented object as output. The disadvantage of all such tricks is that important information gets lost to the system. For example, with the introduction of "higher level" objects and operators, one will have

$$q(q'(p)) = q'(q(p))$$

(except when $q'(p) - p$ is essential for the application of q, or $q(p) - p$ is essential for the application of q'). It is hard to make traditional tree-search routines "aware" of such commutativity. In our opinion, one should instead face the fact that some operators take multiple inputs, and study then separately.

Thus the failure to recognize multiple-input operators has led to inefficient programs. It has also led to a regrettable lack of communication: techniques which have been designed for handling multiple-input operators (e.g. the various "strategies" for the resolution method) have not been recognized as heuristic methods. People seem to think that they are technical details for handling resolution, whereas in fact they are examples of quite general heuristic principles. One can make a parallell with the "Edge" heuristic discussed in section 1: general principles have gone unnoticed for lack of abstract concepts to phrase them in.

As a first step to remedy this situation, let us introduce separate names for the various kinds of operators. The following terms are believed to be illustrative:

number of inputs	number of outputs	name
one	one	perporator
one	multiple	diporator
multiple	one	conporator
multiple	multiple	fociporator

Our second step is to introduce a formalism and a vocabulary which enables us to deal with these different kinds of operators. The formalism is based on lattice theory, and requires a section (section 5) of its own.

Our third step will be to illustrate these general concepts and principles by re-interpreting some current heuristic methods (including the unit preference strategy in resolution). This is done in sections 6 and 7.

Some other complications which may occur in the basic transformation problem, are:

Operators with or-connected outputs.

One often encounters operators which, like diporators, yield a set of objects of outputs, but which merely require that one of the outputs is to be transformed to the target set. Such or-connections may occur

(a) intrinsically, e.g.* "in order to prove $a \vee b$, prove a, or prove b"

(b) because the operator is ambigous, e.g. in resolution logic, where the resolution operator takes two clauses as input and gives one clause as output. Each of the two clauses is a set of literals, and the operator "annihilates" (in a certain sense) two literals, one from each input. The operator has one output for each combination of literals in the two inputs, and is therefore ambiguous.*

(c) because the operator requires a parameter, which may or may not be in the set of objects. For example, in order to prove B in conventional predicate calculus, it is sufficient to prove A and $A \supset B$, where A is arbitrary.*

We shall refer to all operators which yield or-connected outputs, as ambiguous. Thus (a) exemplifies an ambigous perporator, (b) an ambiguous conporator, and (c) an ambiguous diporator.

Still another complication is

Operators with restricted domain, i.e. a domain which is a proper subset of the set P. Some possible ways of dealing with this complication are discussed in section 3.

Example: In integration, the partial integration operator is not always applicable.

A final complication is

No back-up.

In typical problem-solving, application of an operator is never irrevocable: we are always permitted to back up in the solution tree and try some other operator on a previously used object. In some situations (e.g. the Edinburgh studies of heuristic automata), one encounters similar problems where back-up is not permitted. The transformation problem the boils down to the problem of selecting the best operator in each step.

Sometimes, e.g. in planning, a back-up problem can be transformed to a no-back-up problem, or vice versa. We therefore consider both kinds as variants of the same basic problem.

Summing up, transformation problems can be characterized by a couple of features, i.e.

(1) what kinds of operators? (per-, con-, di-, foci-porators)

(2) are operators ambiguous?

(3) are there restrictions to the domain of operators?

(4) is back-up permitted?

3. Approaches to heuristic search

In this section, we shall attempt to classify and name some methods of heuristic search. Our classification will be put to use in the next few sections, where some previously published methods for heuristic search are reviewed.

---

* In example (b), we assume forward proof, and in (a) and (c) backward proof.

In each cycle of the heuristic search process, the program should select one operator to use, and one object (viz. set of objects) to use it on. Object selection seems to be performed in most cases by either of the following two methods:

(A1) Labyrinthic methods proceed doøn the search tree, and have an explicit mechanism for deciding direction in the tree.* This mechanism tells the program "this is a good branch, go on the same direction", or "this is a bad branch, back up -- steps and select another branch".

(A2) Best bud methods use an evaluation function which assigns a priority or merit to each growth direction (bud) in the tree. At each cycle, the program takes a global look at all the buds, selects the best one, sprouts it, and iterates the cycle. In the new cycle, the best bud from last cycle is no longer a candidate, but it has yielded several new buds. All other buds from last cycle are candidates anew. Back-up occurs automatically if the new buds are unable to compete with the stand-by buds from last cycle.

Methods (A1) and (A2) have been formulated for perporators. It is easy to extend them to diporators. For conporators, it is sometimes a good idea to select one input to the operator according to a labyrinthic or best-bud method, and then to select "best companions" to the selected first input. We consider this the generalization of (A1) and (A2) to multiple-input operators. A third method cathegory for them would be

(A3) Best bud bundle methods, which use an evaluation function which assigns a priority to each combination ("bundle") of "buds", and selects the best one in each step.

GPS and SIN use labyrinthic methods, whereas SAINT, the Graph traverser, MULTIPLE, and PPS use best-bud methods. The unit preference heuristics (strategy) in resolution is an example of a best bud bundle method.

Another (and at least in principle, independent) basis of classification is how the program selects the operator in each cycle. The following methods have often been used in practice:

(B1) Object(s) first, one operator afterwards method: First select the most promising object(s) to work upon, according to a labyrinthic or best-bud method. After that, find a good operator to apply to it (them).

(B2) Exhaustive method: Select object(s) like in (B1) and apply all operators to it.

(B3) Object(s) first, a few operators afterwards method: A compromise between (B1) and (B2): a few (but not all) operators are selected and applied to the object(s).

(B4) Object and operator together method: Consider all possible object-operator combinations and select one of them, using a priority function. (This is in other words a best-bud method, where each object-operator combination is considered as a "bud".)

The MULTIPLE program is an example of (B2), GPS and SAINT are examples of (B3), whereas unit preference and PPS are examples of (B4). The version of the Graph Traverser described in {Doran 1966a} is an example of (B2), whereas the later version described in {Michie 1967a} is of type (B1).

In methods (B2) and (B3), object selection in one cycle is effectively a choice of operator in the previous cycle. Therefore, they can be considered as special cases of (B1), with a very careful and timeconsuming method for operator selection.

The four cases above are clearly not exhaustive, as it is in principle quite possible to run an operator first, object afterward method. Also, labyrinthic instead of best-bud selection of operators is possible (one would keep using the same operator until a "back-up" or "change operator" criterion is satisfied). However, these possibilities are probably useless for practical problems.

If the number of operators is very large, or if some operators are ambiguous with a large number of alternatives, then it is not possible to search through all possible cases. This excludes (B2) and (B4) methods. One must first select the proper object(s), and then use a function which selects one or a few operators (and ways of applying them, if ambiguous). Usually, this function recognizes features in the given object, features which determine what operators may be suitable.

In many practical problem environments, one encounters operators which are only defined on a subset of the set P of objects. This restriction has been dealt with in at least two ways, which provides us with a classification in still another dimension:

(C1) Consider as failure. If we have heuristically selected an object and an operator, and it turns out that the object is not in the domain of the operator, then give up this branch and try something else.

(C2) Solve sub-problem. Let M' be the domain of the operator. Solve the transformation problem from the given object to M', and apply the given operator to the result. Formally, we extend the definition of our operators, so that $q(p) = q(p_1)$, where p_1 is the (possibly ambiguous!) solution to the transformation problem from p to the domain of q.

* As we shall see later, we sometimes have a lattice rather than a simple tree.

SAINT uses a type (C1) method, whereas GPS and PPS use type (C2) methods.

In conclusion, we have pointed out three features in heuristic methods. These features can be used to classify and characterize the methods. They are:

(A) Mode of object selection

(B) Mode of operator selection

(C) Way of handling restricted domains for operators.

4. Some frequent techniques in heuristics.

In this section, we shall discuss the use of "merit orderings", plans, and feature vectors ("images") in heuristic methods.

Use of merit orderings.

Definitionwise, best-bud methods require that there exists a way of selecting the "best" one from a set of buds. In all best-bud-type methods known to the author, this selection is based on an (explicit or implicit) partial ordering $>$ on the set P of objects. Some maximal bud according to $>$ (i.e. some bud b^* such that no other bud satisfies $b > b^*$) is then selected as "best bud", and is sprouted.

In some, but not in all cases, the merit ordering $>$ is implemented as an explicit merit assignment function e, i.e. a mapping from P to the set of real numbers. $>$ is then defined in an obvious manner through

$$p_1 > p_2 \equiv e(p_1) > e(p_2)$$

The problem of finding a suitable merit ordering for a given problem environment is of course crucial. Often, it is thought about as an estimate of distance. One attempts to define a function d, where $d(p_1,p_2)$ is a rough estimate of the work (the number of operator applications) required to transform p_1 into p_2. Similarly, one attempts to compute

$$D(p,B) = \min_{b \in B} d(p,b)$$

for reasonable sets B. The merit function e is then defined e.g. as $e(p) = - D(p,M)$.

The use of merit orderings is not restricted to best-bud methods. In labyrinthic methods, the criterion for abandoning a path and trying another may be that $q(p) < p$ by some merit ordering. The GPS utilizes exactly this heuristics.*

* The name "General Problem Solver" has sometimes been criticized as being too uninformative. It is natural to call a heuristic method goal-directed if its merit function is defined through D. The variant of GPS described in {Newell 1961a} can then be characterized as a Goal-directed Perporator Search method.

At first sight, the idea of using a merit ordering has much appeal. On closer scrutiny, it turns out to be less than obvious. It all depends on what kind of economy we desire.

Suppose we are solving a transformation problem for perporators, and that we have already searched part of the tree. Then which of the following quantities do we want to minimize in our next step:

(D1) The number of steps (i.e. operator applications) in the "solution path" from the initial set R to the target set M ?

(D2) The remaining number of steps in the "solution path" from the selected bud to a member of the target set M ?

(D3) The (remaining) number of steps, including steps that are performed in blind alleys (i.e. the total number of arcs in the solution tree the way it looks when we have reached M)?

(D4) The quantity mentioned in (D3), except that if a path is trodden, abandoned through back-up, and then resumed, the steps which are trodden several times shall be counted as multiple steps?

If the path to the solution of the transformation problem is to be used as a plan for a more expensive activity in another environment, then (D1) is of course the correct criterion. On the other hand, if we are interested in a member of M, rather than in the path to this member (e.g. if we are searching for a solution to an integration problem), then (D3) or (D4) would be the correct quantity to minimize. (D3) should be used if the entire search tree is stored in memory, and (D4) should be used if the search tree is stored implicitly on the push-down-list, so that abandoned paths are garbage-collected and all work there has to be re-performed. (D2) is sound in no-back-up situations, like Doran's heuristic automaton.

If criterion (D1), (D3), or (D4) is to be used, then the "merit" of a bud is not simply a function of that bud and the target set, but instead a function of the whole "stump" of the solution tree that has been searched up to now. For example, if the criterion (D3) is used, then the remaining work from a bud is affected if there exists some other bud which has almost as much merit, and which in the future may attract the problem-solver's attention for blind-alley work. It follows that the idea of a merit ordering is sound only if we want to use criterion (D2).

Although theoretically shaky, the use of merit orderings seems to be the only available technique today. If criteria (D3) or (D4) are relevant (which is usually the case), then the use of a distance estimate as a merit function is even more questionable. We shall treat this question in a later paper. But again, the distance estimate seems to be the only technique we have.

Use of plans.

Let P, Q, R, and M define a transformation problem for which a solution is known, and let P', Q' = Q, R', and M' define a transformation problem which is to be solved. Assume also that there exists some mapping h which maps P' onto P, R' onto R, etc. in such a way that if p and q(p) are steps in the known solution, and if p = h(p'), then q(p) = h(q(p')). In other words, the function h maps solutions in P' onto solutions in P. Then we can clearly find a solution in P' by just retracing the solution in P *. The solution in P will be referred to as a plan for the solution in P'.

This ideal situation probably never exists, except when h is the identity function. However, it may be the case that the requirement q(p') = h(q(p)) often (though not always) holds. Then it can still be a good strategy to try to follow the plan. If it does not work, we have to take resort in another plan, or in the object-operator selection methods mentioned above. (In other words, use of plans may be considered as yet another method, (B5), of operator selection).

Plans can be generated in several ways, e.g. by memorization of previous, successful solutions (Doran's heuristic automaton), by human advice, or by "look-ahead": solution of an analogous problem in an auxiliary problem space (e.g. in the Planner system and the PPS).

When the problem environment is predicate calculus, the "abstraction function" h can e.g. be selected so as to throw away everything except the variables in the formulas (planning GPS) or to throw away everyting except the boolean connectives (Planner).

A third technique is

Use of images.

By an image, we mean an item which expresses some, but not all the information of an object in the set P. The image may be for example, a vector of features in the object, or (in the case of a LISP-type formula), the top-level structure of the object, with lowerlevel sub-expressions being replaced by asterisks. Although they rarely talk about it in abstract terms, many creators of heuristic programs do in fact use such images. Images are used for several purposes, including:

(1) as a basis for merit functions (a numerical value is assigned to each feature, and merit is computed as a weighted average of the feature values) or distance functions (computed as a wighted average of "distance" between features);

(2) as objects in an auxiliary problem space used for planning;

(3) in methods of type (B3), for the selection of operators that should be applied to a given object.

* To insure that we have a solution, we must assume that only members of M' are mapped into M, i.e.

$$h(p') \in M \supset p' \in M'$$

Moreover, it is essential that R' is mapped onto (rather than into) R, and that M' is mapped onto M.

Examples: (1) game-playing programs and (with certain modifications) Doran's heuristic automaton; (2) planning GPS, Planner, PPS; (3) GPS.

In this section, we have described and classified general heuristic techniques, and given references from each technique to actual programs which utilizes it. In sections 6-7, we shall build an inverse system of references. Each section will review one heuristic program in terms of the classification and concepts above.

5. Lattice instead of trees.

Heuristic search is often referred to as tree search. However, the tree model is only applicable to cases where all operators are perporators or (with some extra conventions) diporators. With conporators, the need for a more general structure arises. In this section, we shall suggest one possible way of performing the generalization.

Instead of a solution tree, we shall introduce a solution lattice. For perporators, but not for diporators, the solution lattice degenerates into a tree as usually drawn. - For a good introduction to lattice theory, see {Rutherford 1965a}.

First some general notation. Let q be an unambiguous operator which is defined with one set $P' \subseteq P$ as inputs, and which yields $P'' \subseteq P$ as outputs. We then write $P'' = q(P')$. For the moment, we forget about ambiguous operators.

The ordered k-tuple whose elements are $a_1, a_2, \ldots a_k$ will be written $<a_1, a_2, \ldots a_k>$. a and $<a>$ are considered as distinct items*. If b is a k-tuple, the last element of b is written $\omega(b)$. If B is a set of tuples, the set of last elements of members of B is written $\Omega(B)$.

We now define the set S (the solution lattice) as follows:

(1) if p is an object, then $\langle p \rangle$ is a member of S;

(2) if s and t are members of S , then $s \cup t$ and $s \cap t$ are also members of S;

(3) see below.

Following Rutherford, we define $x \subseteq y$ to mean $x = x \cap y$. Also, we assume commutative, associative, and absorptive laws for $\cup$ and $\cap$. Distributive and idempotent ($x \cup x = x$ etc.) laws for $\cup$ and $\cap$ follow easily. Also, we find that the $\subseteq$ relation is transitive, and that

$$x \subseteq y \wedge y \subseteq x \leftrightarrow x = y$$

* Instead, we shall frequently write a when we mean {a} .

In diagrams, we shall illustrate members of S as nodes, and relations $\subseteq$ as arcs. Arrowheads will usually not be indicated; instead, we always position the left-hand argument of $\subseteq$ below the right-hand argument. For example, $x \subseteq y$ is illustrated as

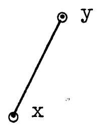


Figure 1.

If $S' = \{s_1, s_2, \dots s_k\}$ is a subset of S, we write $\bigcap S'$ for $s_1 \cap s_2 \cap \dots \cap s_k$, and similarly for $\bigcup S'$.

The last part of the recursive definition of S can now be given:

(3) Let S' be a subset of S for which $P' = \Omega(S')$ and $P'' = q(P')$ for some operator q are defined. Then the triple $< \bigcup S', q, p''>$ is a member of S for each p" in P".

The members of Q are defined as operators on S as follows: If $P'' = q(P')$, $P' = \Omega(S')$, $S'' = \{< \bigcup S', q, p''> \mid p'' \in P''\}$, then $\bigcap S'' = q(\bigcup S')$.

We now have the formal apparatus needed to express the given transformation problem in lattice terms. First, we define the initial set $\bar{R}$ and the target set $\bar{M}$ in S:

$$\bar{R} = \{ <r> \mid r \in R \}$$

$$\bar{M} = \{ s \mid s \in S \wedge \omega(s) \in M \}$$

Thereafter, we make it an axiom that

$$(\forall s \in S) \quad q(s) \subseteq s$$

when q(s) is defined. The given transformation problem then is equivalent to

Lattice transformation problem.

Prove that $\bigcap \bar{M} \subseteq \bigcup \bar{R}$.

The general method to solve this problem is to inspect nodes in the (usually infinite) set S, until one has identified a finite set $S' \subseteq S$ which forms a "bridge" between $\bigcap \bar{M}$ and $\bigcup \bar{R}$. Successive nodes on this bridge are to satisfy a $\subseteq$ relationship according to the above axiom, and the desired result $\bigcap \bar{M} \subseteq \bigcup \bar{R}$ then follows by the transitivity of $\subseteq$.

At each step in the search for this "bridge", a finite subset of S has been inspected. This subset is not a lattice, since it does not satisfy assumption (2) on page 1?. As it is still a partially ordered set (under $\subseteq$), we call it the search (solution) poset.

We shall not attempt to prove the formal equivalence between the general transformation problem and the lattice representation, but only illustrate the idea through a couple of examples.

Example 1. Only perporators considered. The initial solution is illustrated in figure 2. Let us now select a node r in $\bar{R}$ and an operator q, and extend the graph with q(r). The node r has the form $<p>$, and the single new node has the form $<r,q,q(p)>$, where q(p) is of course another member of P. This situation is illustrated in figure 3. After a few more operators have been applied to various nodes, we may obtain the situation in figure 4. Clearly, as soon as some node s in the growing tree has a member of M as its last component, it is a member of $\bar{M}$, and then it has been established that $\bigcap \bar{M} \subseteq \bigcup \bar{R}$, which was our goal. For the pure perporator case, the search poset provided by the lattice formulations is clearly identical to the conventional search tree.

Example 2. A diporator q yields two outputs. Let $q(p) = \{p_1, p_2\}$, and assume that a node s such that $\omega(s) = p$, is already in the lattice. By our assumptions, the two nodes $s_1 = \langle s,q,p_1\rangle$ and $s_2 = \langle s,q,p_2\rangle$ are also members of the lattice, and we have

$$s_1 \cap s_2 = q(s)$$

as well as

$$q(s) \subseteq s$$

It is easily proved that

$$\bigcap \bar{M} \subseteq s_1 \wedge \bigcap \bar{M} \subseteq s_2 \equiv \bigcap \bar{M} \subseteq s_1 \cap s_2$$

The right-hand side of the equivalence implies that $\bigcap \bar{M} \subseteq s$. In other words, transforming both s_1 and s_2 to $\bar{M}$ is sufficient for transforming s to $\bar{M}$. For an illustration, see figure 5.

Example 3. A conporator q takes two inputs, when applied to members of P. Let $q(r_1 \cup r_2) = s$, where r_1 and r_2 are members of $\bar{R}$. The graph is given in figure 6. To prove

$$\bigcap \bar{M} \subseteq \bigcup \bar{R}$$

it is clearly sufficient to prove

$$\bigcap \bar{M} \subseteq q(r_1 \cup r_2).$$

A similar example, where the arguments of q are not members of $\bar{R}$, is illustrated in figure 7. The relation

$$s_1 \cup s_2 \subseteq r_1 \cup r_2$$

(indicating by alternating dots and lines) has been inferred from the others, and is crucial.

Let us finally turn to the case of ambiguous operators. Suppose, in example 2 (figure 5) that q is an ambiguous perporator, and that it is sufficient to transform either p_1 or p_2 to target set M. We then simply redefine $q(s)$ as $s_1 \cup s_2$, with unchanged notation otherwise. See figure 8, and compare figure 5.

If desired, the notation can of course be further extended to arbitrarily complex and/or structures:

Example 4. After applying operator q to object p, we find that it is sufficient to transform

$$p_2 \text{ or } (p_3 \text{ and } p_4 \text{ and } (p_5 \text{ or } p_6)) \text{ or } p_7$$

to the target set M. With s_i as before, we then simply define

$$q(s) = s_2 \cup (s_3 \cap s_4 \cap (s_5 \cup s_6)) \cup s_7$$

Using the obvious distributive etc. laws, this expression can be reduced to the canonical form of an ambiguous diporator.

In summary of this section, we have suggested a formal and pictorial representation of the search "trees" for arbitrary operators. Our search lattice S is the set of all possible nodes in search space. In the search for a solution, we gradually extend the searched poset, which is a subset of S, until it has been proved that

$$\bigcap \overline{M} \subseteq \bigcup \overline{R}$$

6. Heuristics in the SAINT program

In sections 1-5, some aspects of heuristic programs have been discussed. As an exercise in the use of these concepts, we shall now give a description of Slagle's program SAINT. We wish to demonstrate that, with the concepts that have been introduced, the description can be more abstract and involve less programming details than before.

Problem environment.

The set P of objects consists of all formulas built from real numbers, variables, various arithmetic functions *, and one functional: the integration operator. The target set M consists of all objects which do not use the integration operator. The initial set R consists of one single object, which is given to the program on each occasion of use.

The set Q consists of 44 operators. All are perporators, except for one diporator, the formula for the integral of a sum. Some of the perporators (e.g. the substitution operator) are ambiguous and governed by a parameter. Most operators have a restricted domain.

Discussion of heuristic method.

It is natural to sort up the operators in Q into the following disjoint cathegories:

a. Standard forms (26 operators). These are perporators whose output is always in the target set M (if the input contains only one occurrence of the integral operator). An example of such a perporator is

$$\int c^v \, dv = c^v / \ln c$$

Remark: the possibility to single out those operators which land in the target set is particular for this problem environment, and does not occur in e.g. logical inference.

b. Algorithm-like transformations (8 operators). These are operators which, if applicable, are usually appropriate. The diporator is one of them.

c. Heuristic transformations (10 operators). These are operators which may or may not be appropriate. Substitution is one of them.

Let us call these sets Q1, Q2, and Q3, respectively, and define:

P1 the set of all objects in P which are in the domain of some operator in Q1;

P2 the set of all objects in P-P1 which are in the domain of some operator in Q2;

P3 = P - P1 - P2.

Objects in P1 have a solution just around the corner, and should of course be given top priority. For objects in P2, we know which operator should be applied (it turns out that there is never more than one), so such objects are given higher priority than objects in P3. For objects in P3, several operators may be applicable, so a heuristic search has to be performed.

Each object p stands for an expression built with functions. The "maximum depth" of this expression is significant for the following reasons: (1) the members of P1 (usually) have small maximum depth; (2) operators often perform only a small change (one or a few units) in the maximum depth of their input. Under such conditions, it is reasonable to use the depth of an expression as a gross measure of its "distance" to the target set, and (therefore ?) to use it as a merit function.

With this background, the heuristic method used by SAINT can be outlined.

* addition, subtraction, multiplication, power function, logarithmic, trigonometric, and inverse trigonometric functions.

Images.

The SAINT program uses images = feature vectors with eleven components. Maximum depth of expression is one of them. Images are used for three purposes:

(a) selection of best bud (only maximum depth component used);

(b) selection of appropriate operators for a given object in P3;

(c) selection of parameters for ambiguous operators.

Handling of restricted domain.

If a selected operator is not applicable to a selected object, SAINT just gives up. It does not try to solve a sub-problem.

Object and operator selection.

Abstractly speaking, the SAINT program uses an "object first, a few operators afterwards" selection system, where objects are selected with a best bud method based on a merit ordering. However, there are certain complications to this simple scheme.

The following merit ordering is used:

p > p' iff p is a member of P1 and p' is not,

or p is a member of P2, and p' is a member of P3,

or both p and p' are members of P3, but p has less maximum depth than p' has.

In each step, SAINT selects some maximal bud in the search tree according to this partial order, and applies suitable operators to it. The operators are selected according to the following table:

if object is in	then select operator(s) from
P1	Q1
P2	Q2
P3	Q3

In P2, only one operator is usually applicable; in P3, the object's image determines which operators shall be selected. Notice in particular that if object is in P2, then an operator from Q3 is never selected, even if the object is in its domain. The reason is that an object in P2 can be transformed one or more steps by operators in Q2, and then the desired operator in Q3 can be applied to the result. This is sufficient (and is in fact a good pruning technique), since operators in Q2 only effect trivial modifications on the objects.

Programming

Since only one operator is applied to objects in P1 and P2, these objects and operators can be given a separate and "algorithmic" treatment. The heuristic search need only span objects in P3 and operators in Q3.

Like most heuristic programs, SAINT maintains a bud list, i.e. a list of objects to which no operator has yet been applied. This list contains members of P3 ordered according to (the merit order) > .

Somewhat idealized, the cycle in the SAINT program runs as follows:

(1) Take the first object on the bud list. (This is a maximum bud in P3).

(2) Select suitable operators for this object. Apply them. The set or results is called P".

(3) For each member of P", check if some member of Q1 or Q2 is applicable. If so, apply it. If it was a member of Q1, terminate. Otherwise, include the result in P" and reperform (3) on it.

(4) Let P^+ be the modified P" after all Q1 or Q2 operators have been applied. By hypothesis, P^+ is a subset of P3. Merge P^+ into the bud list according to > .

The cycling starts in step 3 with the bud list empty, and with P" = the given, initial object ("the given integration problem").

The occurrence of a diporator in the problem environment is a complication. To handle this, the program maintains a "goal tree", which is equivalent to the search poset of last section, but utilizes a slightly different notation. On discovery of a member of Q1 (step (3) in the routine) SAINT does not actually terminate, but utilizes instead the "goal tree" to remove from the bud list those buds that need no longer be transformed to the target set. In lattice terms, if SAINT has proved for a node t is the search poset that $\bigcap \overline{M} \subseteq t$, then it removes from the bud list all nodes t' such that $t' \subseteq t$. Also, and for the same reason, such members of P" (P^+) are thrown away. SAINT then continues the above cycle (starting in step (1)) as long as there is anything left on the bud list.

Remarks

This terminates our description of the SAINT program. It is based on a rather short summary of the work on SAINT, {Slagle 1963a}, rather than the full thesis. There may therefore be mistakes in details of our description. However, let us repeat that the intention with this section was to demonstrate how exactly the same material may be described in completely different terms when it is to be used for another purpose.

To facilitate comparison, let us finally give a short dictionary that translates between Slagle's terminology and ours:

Slagle	here
heuristic goal list	bud list
(temporary) goal list	P", P^+
character	image
characteristic	feature

7. The unit preference heuristics in resolution

The purpose of this section is the same as that of section 6, i.e. to demonstrate the usefulness of abstract heuristic concepts. In addition, we shall try to show that the so-called strategies used in resolution are in fact heuristic methods, and amenable to the same treatment as other such methods *. Therefore, we have selected to make a description of the unit preference strategy for resolution.

Problem environment

Each object in the set P is a set of literals, a literal being a symbolic expression (NOT (R_i ... ...)) or (R_i ... ...) . The target set M has one member: the null set (i.e. the set of no literals). The initial set R consists of a relatively small number of objects and is given to the program on each occasion of its use.

Notice that in this case, R is given as input to the program, and M is fixed. In the case of SAINT, we had the opposite situation.

The set Q consists of a two-input conporator ("resolution") and a perporator ("factoring"). Both have a restricted domain, and both are ambiguous. The ambiguities are moderate: the number of alternatives is finite and so small that all can be tried.

Images

Unit preference uses images for object-operator selection. The image of an object is an integer, viz. the number of literals in the object. Operators can be extended to images in the following manner: if the inputs to the resolution operator have images j and k , then the output (if it exists) has image j+k-2 . Similarly, if the input to the factoring operator has image j , then the output, if it exists, has j-1 as image**.

Discussion of heuristic method.

Since the target object has image zero, and the operators effect a relatively small change on images, it is reasonable to take the image of an object as a crude estimate of its merit in the search towards the target, with small images having a higher merit. Therefore, operations which decrease the image can be expected to bring us closer to a solution. This gives us a preference for factoring, and for resolution when one input has image 1.

A trivial strategy would be to reduce the image to zero through successive factorings. However, we run into problems with the restricted domains of the operators: factoring when the image of the input is 1 (i.e. the last step) is never possible, and in all reasonable problems we would fail far before that.

Resolution when the partner's image is 1 ("unit resolution") seems to be a better strategy, adn is what our heuristics prefers as first choice. When it cannot be had, we perform other resolutions or factoring a couple of steps, in the hope of achieving unit resolution later.

Handling of restricted domain.

If a desired operator is not applicable, the unit preference method just gives up.

Object and operator selection.

Unit preference utilizes a best bud bundle method, where a suitable operator and its input(s) are selected together. The system makes implicit use of a merit ordering > defined as follows on $I \cup I^2$:

numerical relationship (< means "less than")	merit ordering (> means "better than")	
$j < m$	$\rightarrow \{1,j\} > \{1,m\}$	... (1)
$k \neq 1,\ m \neq 1$	$\rightarrow \{1,j\} > \{k,m\}$	... (2)
$k \neq 1$	$\rightarrow \{1,j\} > k$	... (3)

if i,j,k,m all are $\neq 1$, we have:

$i + j < k + m$ or $i+j = k+m$, $\min(i,j) < \min(k,m)$	$\rightarrow \{i,j\} > \{k,m\}$	... (4)
$i + j \leq k$	$\rightarrow \{i,j\} > k$	... (5)
$k < i + j$	$\rightarrow k > \{i,j\}$	... (6)

For example, we have

$$\{1,2\} > \{1,4\} > 2 > 3 > \{2,2\} > 4 > \{3,2\} > \ldots$$

$$\ldots > 7 > \{6,2\} > \{5,3\} > \{4,4\} > 8 > \ldots$$

The relation > is extended to $P \cup P^2$ in the obvious way.

* Feigenbaum, in his IFIP 68 paper {Feigenbaum 1968a}, argues a similar standpoint.

** It may accidentally happen that the image of the output is less than (but never greater than) j+k-2 viz. j-1 . Such accidents are rare and do not affect the heuristics.

In each cycle, unit preference uses $>$ to select one maximal object or object-pair and applies the correct operator (factoring in the case of an object, resolution in the case of an object-pair) to it. In case of ambiguity, all alternatives are treated with the same priority. If operator application in some alternative is successful, and the output has higher priority than the input (and therefore, higher priority than the other alternatives processed together with this one), then the higher priority is honored immediately.

Programming.

Although our reference says little about the actual program that performs the unit preference heuristics, the following are some suggestions for such a program.

The program utilizes lists L_1, L_2, ... L_j, ... , where L_j contains all generated objects with image j, together with the following information for each object:

(1) has factoring been attempted on this object?

(2) if factoring is ambiguous, for which cases has it been attempted?

(3) with what other objects has resolution been attempted?

(4) if resolution is ambiguous, for which cases has it been attempted?

The answers to these questions can be represented as follows:

(1) for each list L_j, where $j \leq 2$, a pointer indicate how far down the list factoring has proceeded;

(2) for the pointed-at element of each list L_j, the attempted alternatives are listed. (For all other alternatives of L_j, either none or all alternatives have been attempted);

and similarly, for each object p_{jm} on each list L_j,

(3) for each list L_k, where $k \leq j$, a pointer indicates how far down L_k resolution with p_{jm} has been attempted;

(4) for the pointed-at element of each list L_k, the attempted alternatives are listed.

With these conventions, programming is straightforward.

Remark.

The images used by the unit preference method have a noteworthy property: the image of the output of an operator is a function of the image(s) of the input(s), if the operator is applicable; but the image does not contain enough information to determine applicability. This "semi-deterministic" property has otherwise been characteristic of planning methods, notably planning GPS, and PLANNER. As a result of some present work, we believe that semi-deterministic images have interesting theoretic properties.

Pruning criteria.

The unit preference heuristics should only be used in combination with various pruning criteria, such as:

(1) Restriction on search depth. The depth of an object is the number of resolutions that was required to construct it. Objects of depth $\geq k_0$ (where k_0 is a fixed parameter) are rejected;

(2) Set of support strategy. A subset T of R is singled out as "essential initial objects", and nodes p in the search poset which satisfy

$$p \subseteq \bigcup(\overline{R}-\overline{T})$$

are given zero merit;

(3) Rejection by pattern. Objects p which conform to certain patterns (e.g. contain two literals of the form A viz. (NOT A)) are rejected.

We have then made a distinction between heuristics (i.e. rules which govern the order in which the solution lattice is searched) and pruning criteria (which are extreme cases of heuristic rules since they cut off some "branches" altogether). In the resolution literature, both heuristics and pruning criteria are called strategies.

Pruning criteria can formally be treated as a further restriction on the domains of operators. The first two pruning criteria above can (alternatively) be implemented by using images $<j,d,s>$, defined as follows:

If p is a member of the initial set R , p's image is $<j,d,s>$, where

j is the number of literals in p ;

d is zero;

s is the truth-value of $p \in T$

If p was derived through resolution, and the images of the inputs were $<j_1,d_1,s_1>$ and $<j_2,d_2,s_2>$, then the image of p is $<j,d,s>$, where

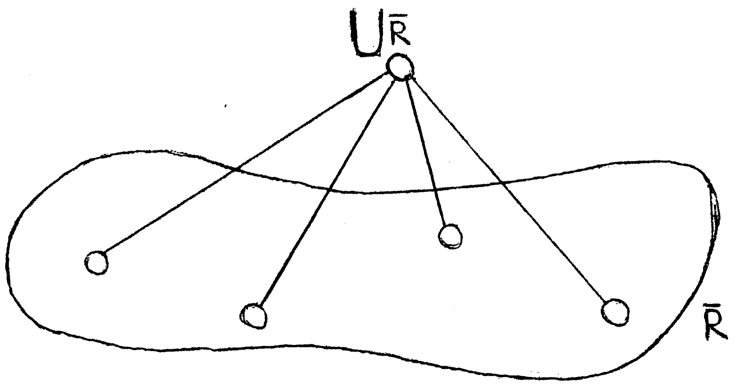
$j = j_1 + j_2 - 2$ (the number of literals in p)

$d = \max(d_1,d_2) + 1$

$s = s_1 \vee s_2$

Finally, if p was derived through factoring, and the image of the input was $<j,d,s>$, then the

Figure 2.



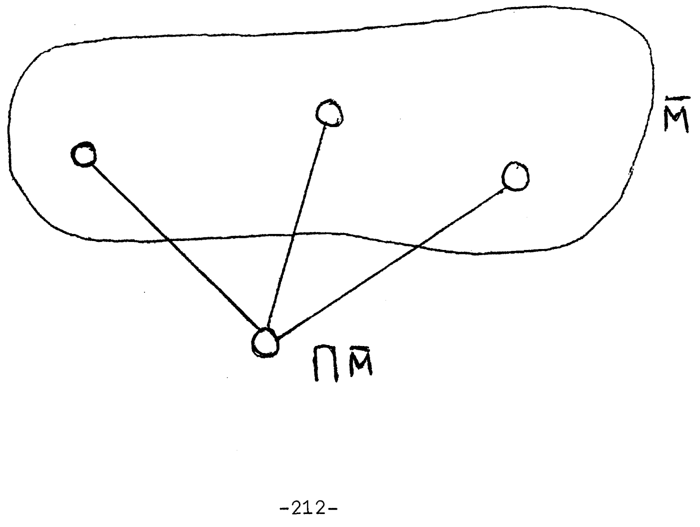
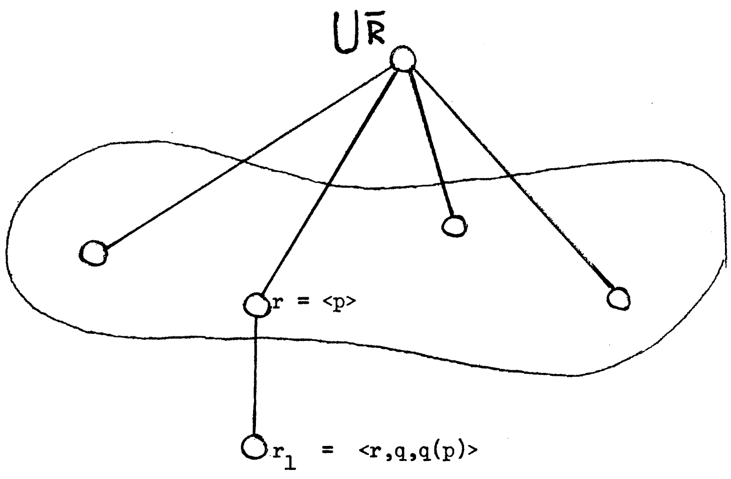
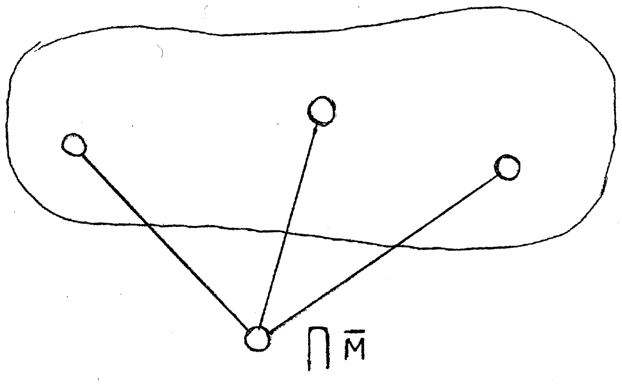


Figure 3.

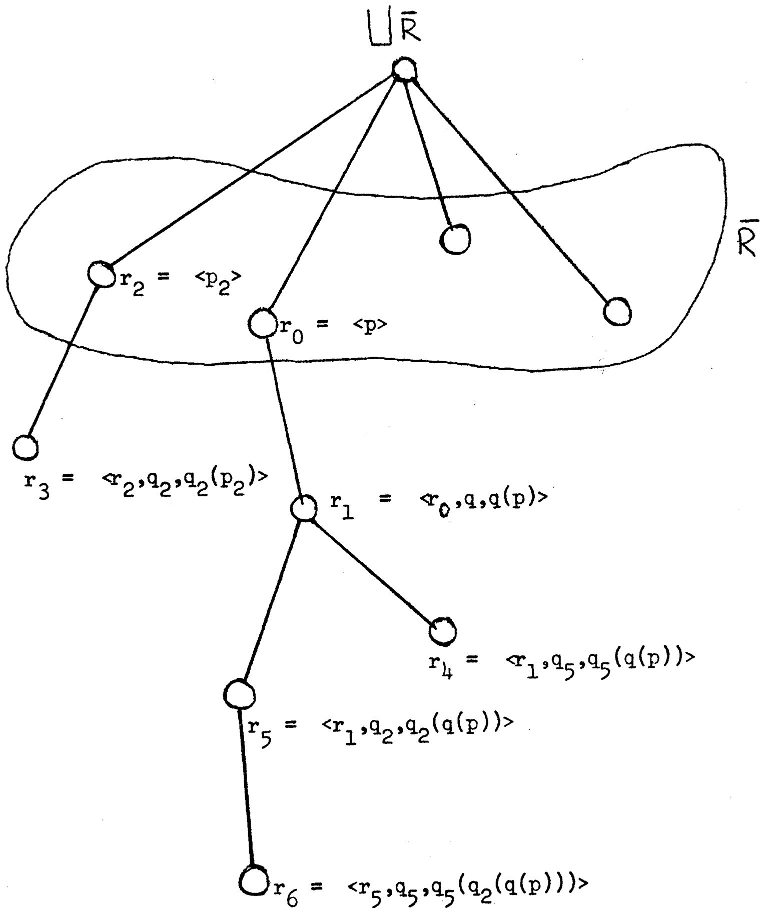


$\bigcup \bar{R}$
$r = \langle p \rangle$
$r_1 = \langle r, q, q(p) \rangle$

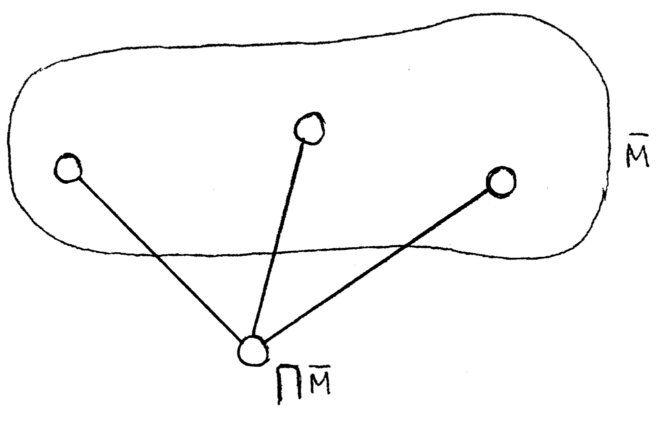


$\bigcap \bar{M}$

Figure 4.

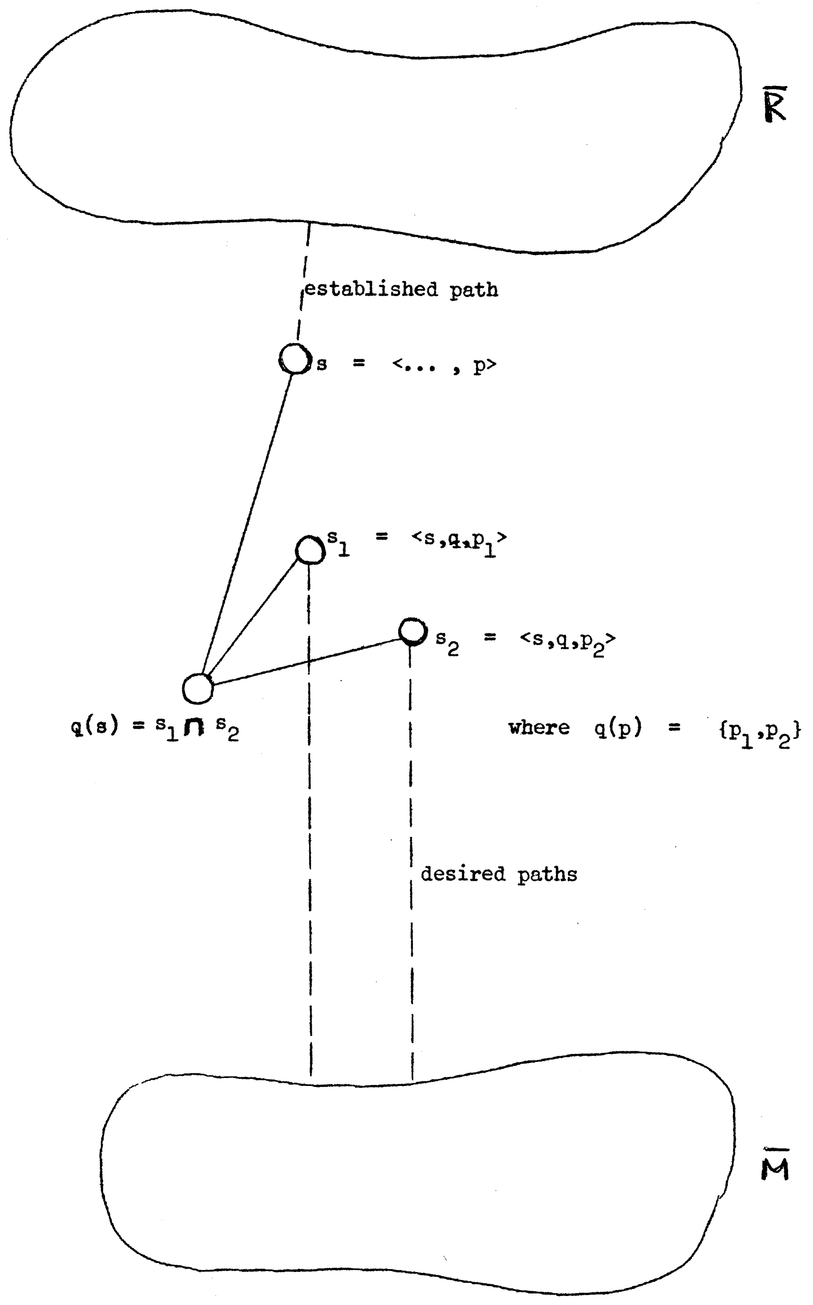


$\bigsqcup \bar{R}$
$\bar{R}$
$r_2 = \langle p_2 \rangle$
$r_0 = \langle p \rangle$
$r_3 = \langle r_2, q_2, q_2(p_2) \rangle$
$r_1 = \langle r_0, q, q(p) \rangle$
$r_4 = \langle r_1, q_5, q_5(q(p)) \rangle$
$r_5 = \langle r_1, q_2, q_2(q(p)) \rangle$
$r_6 = \langle r_5, q_5, q_5(q_2(q(p))) \rangle$



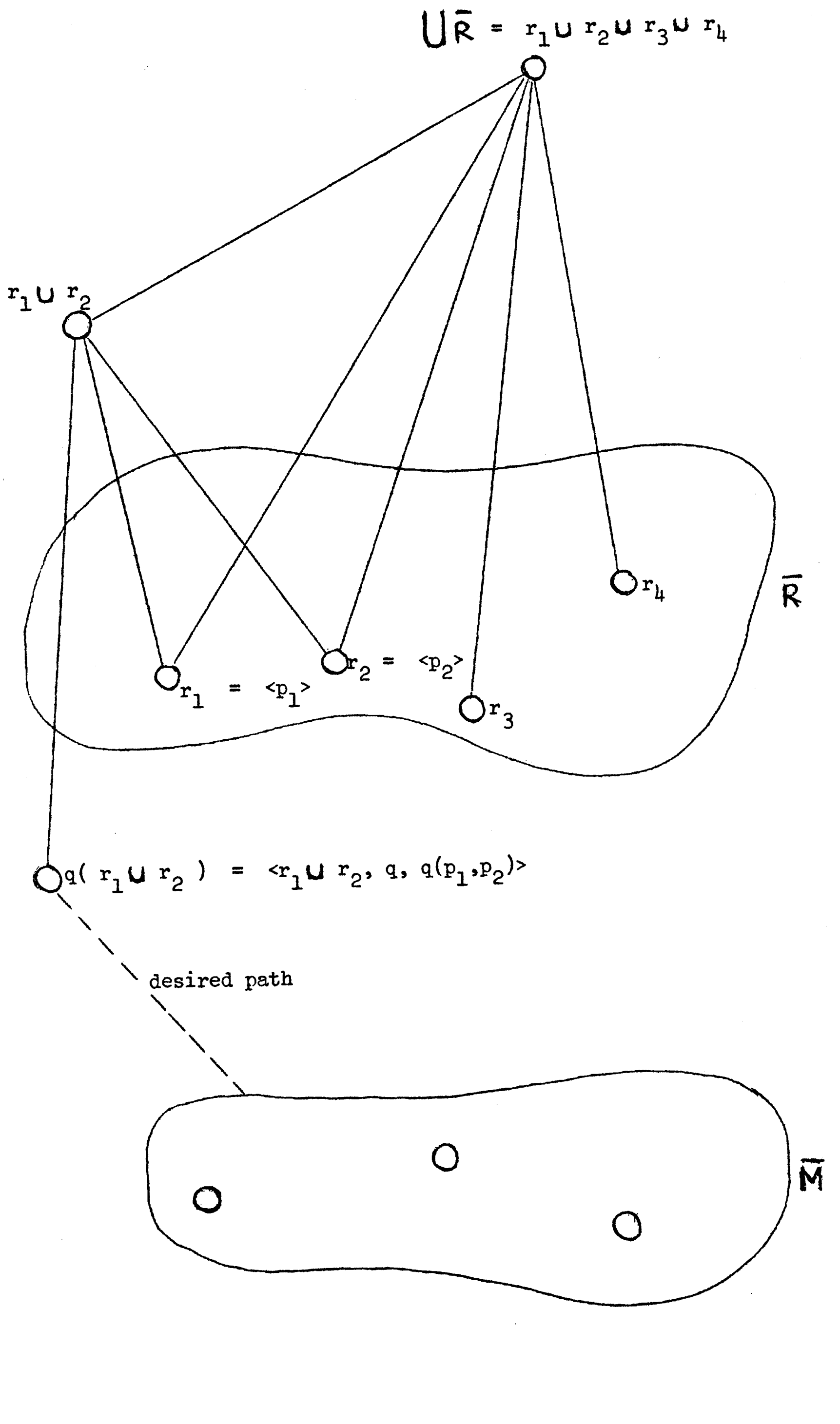
$\bar{M}$
$\prod \bar{M}$

Figure 5.



$\bar{R}$
established path
$s = \langle \ldots , p \rangle$
$s_1 = \langle s,q,p_1 \rangle$
$s_2 = \langle s,q,p_2 \rangle$
$q(s) = s_1 \cap s_2$
where $q(p) = \{p_1,p_2\}$
desired paths
$\bar{M}$

Figure 6.



$\bigcup \bar{R} = r_1 \cup r_2 \cup r_3 \cup r_4$
$r_1 \cup r_2$
$r_1 = \langle p_1 \rangle$
$r_2 = \langle p_2 \rangle$
r_3
r_4
$\bar{R}$
$q(r_1 \cup r_2) = \langle r_1 \cup r_2, q, q(p_1, p_2) \rangle$
desired path
$\bar{M}$

Figure 7.

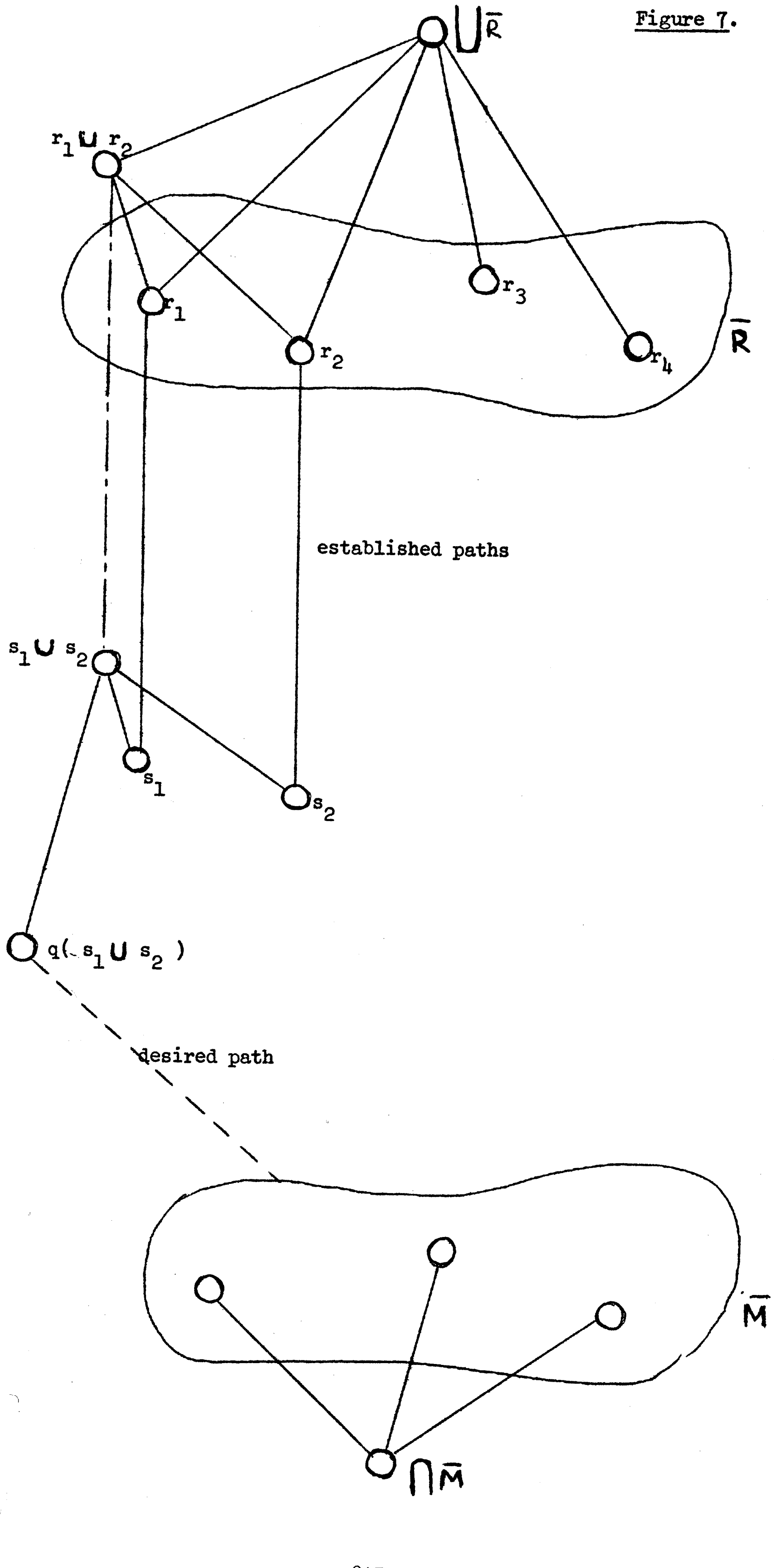
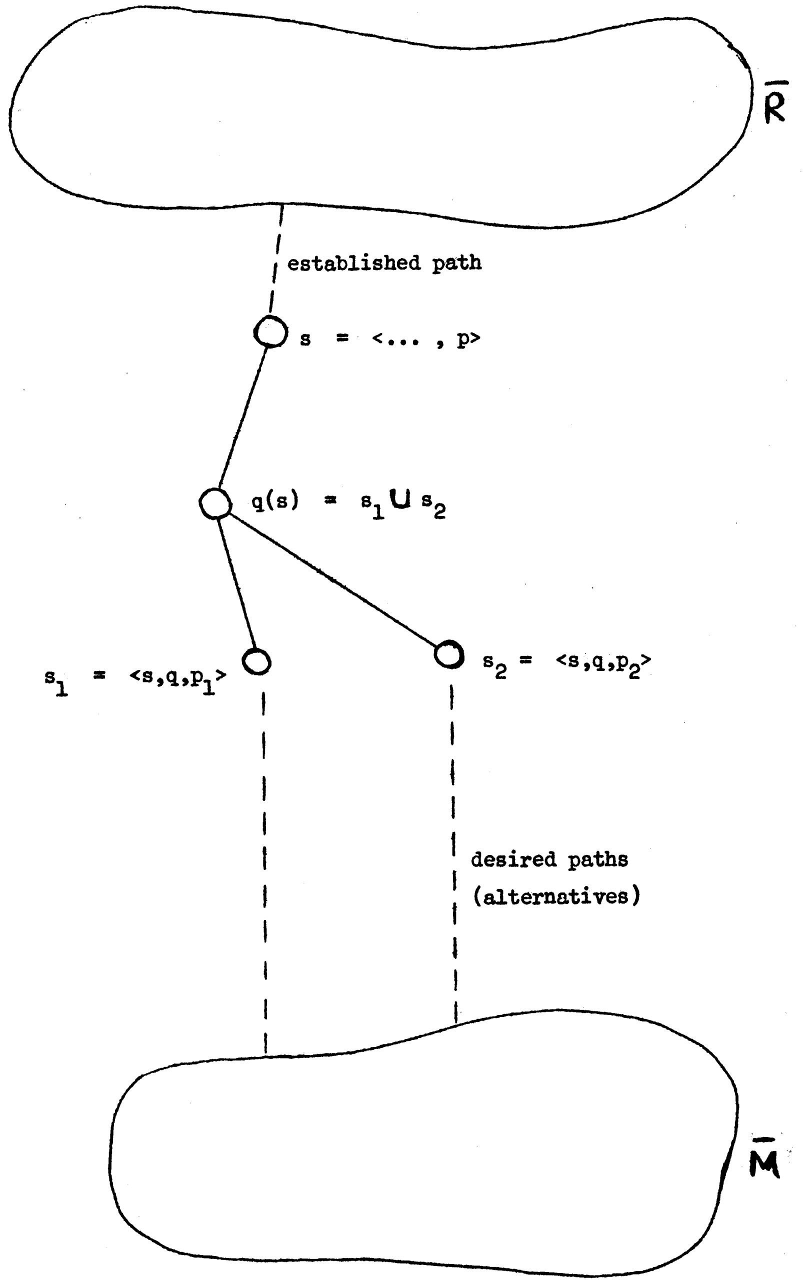

$\bigcup\bar{R}$
$r_1 \cup r_2$
r_1
r_2
r_3
r_4
$\bar{R}$
established paths
$s_1 \cup s_2$
s_1
s_2
$q(s_1 \cup s_2)$
desired path
$\bar{M}$
$\bigcap\bar{M}$

Figure 8.


$\bar{R}$
established path
$s = <\ldots , p>$
$q(s) = s_1 \cup s_2$
$s_1 = <s,q,p_1>$
$s_2 = <s,q,p_2>$
desired paths
(alternatives)
$\bar{M}$