THE UNIVERSITY *of* EDINBURGH

A Computational Model of Learning

by

Timothy J. Radford

for the degree of

Master of Philosophy

University of Edinburgh

1979

Acknowledgements

My sincere thanks to all of the following:

For supervision and encouragement,

Dr. Gordon Plotkin

Dr. Jim Howe

For reading the draft, and more encouragement,

Dr. Ben du Boulay

For financial support,

The Scarbrow Bursary Trust

The South African Council for Scientific and Industrial Research

For the use of "Diamond" word-processing equipment,

Data Recall Ltd

And to all of those at the Department of Artificial Intelligence who helped create an exciting and stimulating environment during the course of this research.

Contents

Chapter 1

Introduction

1.1 Objectives

The program described learns to improve its performance in the playing of a game, from experience. The main objectives of the project are that the system should observe the following principles:

1) The program should not rely on any special evaluation functions, which would embody domain-specific information.

2) Initial knowledge of the domain should be minimal, and further knowledge gained should be assimilated in terms of prior knowledge

3) The system of representation employed should as far as possible be independent of the domain, again avoiding the incorporation of domain-specific information.

In customary Artificial Intelligence terms, the program is referred to as existing in a domain or environment. The model has a goal within this domain and has available certain actions which it may take in order to achieve its goal. The goal is represented as a Structure. This term will be used throughout to denote a set of objects from the domain, constrained by various domain-pertinent

relationships. The actions, goals and objects are the initial known facts of the environment. The program has an innate ability to plan simple sequences of actions to achieve its goals. Inevitably, these plans do not take into account enough of the nature of the domain and prove inadequate. In such events the descriptive abilities of the program are invoked to correct the deficiency, and the program's model of its environment is enriched.

The chosen domain is that of a two-person game, namely GoMoku (see Appendix A). This is played on the vertices of a 19 by 19 board. Each player has a set of uniform playing pieces, called stones. One set is white and the other black. Players alternately place single stones on the board, the object being to form a line of five adjacent stones of one colour. The player to do so first wins. Strictly, the line formed must contain no more than five adjacent stones of one colour, but this rule has been relaxed for present purposes.

The fundamental assumption underlying the model is that anything that is learnt must always be assimilated in terms of what is already known. The only knowledge initially posessed by the program is the description of its domain and of its goal. What the system learns can only be related to the domain and the goals, and it is therefore necessary that it should be possible to represent new information in an identical manner to that in which the initial information is described. The model

therefore hinges on a system of description which can be employed to represent all the elements of the domain and can be used by the program to create its own descriptions. Two methods of forming new descriptions from old are used: The first is by the modification of the constraints on the variables of the old description, while the second involves combining more than one old description to form a compound description.

A secondary aspect of the work presented is an extension of the notion of pattern directed invocation. Conventionally, a procedure which is to be invoked in this manner has an associated pattern, which consists of a string of words or values and variables. A procedure is invoked if its pattern matches a target pattern, according to some matching algorithm. Matching a value in the target string to a variable assigns the value to the variable for the duration of the invocation. The present program incorporates a logical extension of this process, whereby patterns associated with procedures may be represented using the full powers of the descriptive system.

The pattern of a procedure may either be a representation of the outcome of the application of the procedure, or a description of conditions under which it might be appropriate to apply the procedure. In goal directed languages such as PLANNER [Hewitt, 1972], the term "procedure" is usually reserved for the former of these alternatives, and the term "demon" is used for the latter. The present

model regards the pattern as a description of a position to which the program may have to respond, i.e. in the "demon" sense. The term "response" more obviously describes this function, and is thus used hereafter.

The program is implemented in POP-2 (see Appendix B). The main feature of this language that is employed is the facility to describe and manipulate data-structures, and the ease with which programs can be treated as objects and associated with such structures.

Diagrams depicting GoMoku positions occur throughout this text. In these figures, white stones are shown as "o" and black stones as "x". The symbol "+" is used to highlight unoccupied vertices that are significant in some other way.

1.2 The Program in Action

The program's approach to the game is initially simple-minded. It randomly selects a line of five vertices and proceeds to place a stone on each. Figure 1.1 shows the position arrived at after it has made four moves. The program has just placed the fourth white stone and is planning to place the fifth at position G8. The opponent naturally places his fourth black stone at G8 and the program finds its own plan thwarted. To avoid similar events in the future, the program must be able to anticipate them. By comparing the position arrived at in the planned line with the description of its goal, and taking

into account the previous move, it is able to describe the situation immediately prior to the move.

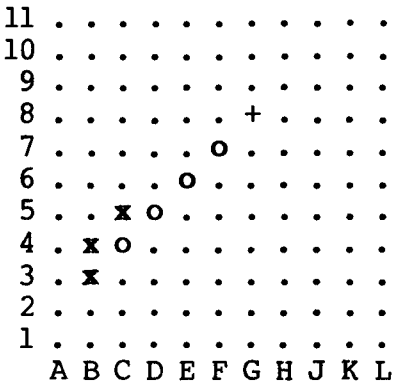


Figure 1.1 - Learning about blocking

The goal description is independent of the actual positions of the vertices, the direction of the line and the colour of the stones, save that they are of the same colour. This is also true of the new description just formed. The effect of "undoing" the last move is to descibe the vertex at the end of the line as unoccupied. The description is therefore of the form:

s s s s +

where the occupied vertices are represented by "s" and the unoccupied vertex by "+". In forming this description, the system employs the technique of modifying the constraints within the original description.

The game continues and the newly-gained knowledge is soon put to use. In figure 1.2, the program is constructing a new line on row 10, in its simplistic manner. Its plan necessitates playing next on vertex E10. It notices,

however, that its opponent has constructed a configuration that matches the description recently acquired, on column B. As the number of moves required to complete the opponent's line is less than that required to complete its own, it diverts from the plan and plays at position B5. Without the initial experience the program would have proceeded with its own plan and made no attempt at blocking.

The system does assume the domain to be the same from the point of view of either player. If this were not the case, it could not generalise from being blocked to blocking.

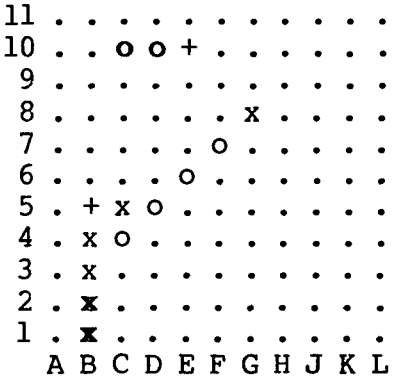


Figure 1.2 - Blocking

Having been successful in blocking the opponent's line, the program continues with its line on row 10. At the position reached in figure 1.3 it is about to play on vertex F10, when it is confronted by two simultaneous instances of the familiar structure. One runs from B4 to F8, and the other from E7 to A3. At this point the

program finds that it is impossible to block both of the lines at once and determines that it has lost.

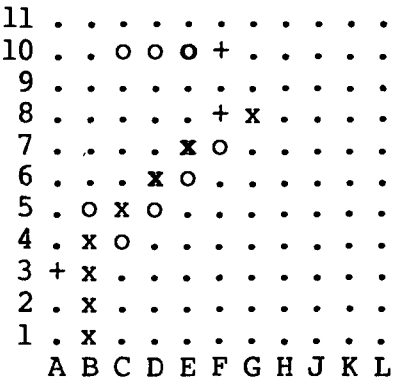


Figure 1.3 - Learning "Open-fours"

The two instances of the structure are now treated in the manner described above, in the light of the previous move. Two new, distinct structures result, each having three stones and two blanks. The line beginning at B4 results in a structure of the form:

s s s + +

while that starting at E7 produces one of the form:

+ s s s +

The program employs the technique of combining the two descriptions by pairing the description-variables which represent the same objects. This results in a compound structure which describes the overall configuration, thus:

+ s s s + +

This fully describes the line stretching from A3 to F8, with the black stone on E7 "unplayed". The program is able to recognise any occurrences of patterns which match this structure. In any new game the program will now

always try to prevent the opponent from creating an "open-four". If this becomes impossible, it will learn to anticipate the new conditions under which it became impossible and to avoid them likewise.

1.3 Commentary

The program is a model of learning by trial and error. The planning component proposes a sequence of actions which is intended to produce an instance of the goal structure. This plan is initially naive, but even when the performer becomes more sophisticated it can always fall back on the planner. In the example in the section above, the model decides on a set of five vertices which form a line as required, and proposes a sequence of actions which will place five stones on the vertices, thus forming an instance of the goal structure. Such a plan naturally overlooks the interactive nature of the game and will inevitably be defeated.

There are two ways in which the plan can fail. The first is by interference: Some intended action may be prevented by another event. In the current domain all other events are perpetrated by an opponent. One of the actions may, for instance, be prevented by an action of the opponent, or in game-playing terms, may be "blocked". In the example, the program's planned line of five is quite easily blocked. A generalised description is formed of the situation immediately prior to the blocking move, together with a description of the blocking move and its outcome,

namely the failure of the planned line. Only the objects related to the goal description are considered, so that the order of the opponent's moves and any other moves he might have made are irrelevant at this point.

Because the game is symmetric, in that the goals and possible actions of the opponent are the same as those of the performer, such a description can be used as a blocking response. The second way in which a plan can fail is by the instantiation of the counter-goal. In a game, this usually means that the opponent has achieved his goal. In the example, this happens when the opponent creates a line of five first. In this case the performer needs to find a way of anticipating and preventing the reoccurrence of such an event.

Generalisation of the former of the situations above results in descriptions of blocking responses, while the latter situation leads to responses as long as it is possible to see a way in which the event could have been prevented. It is also sometimes possible to anticipate the opponent's success, if it has actually become inevitable (or un-blockable), and such situations will also give rise to responses. The GoMoku example in section 1.2 ends in this way, when the black player forms an "open-four". The description of the prelimenery state is given to the pattern matcher as a standard situation to look out for. If such a situation is later detected, the blocking

response is recommended to the performer, who may then decide whether or not it is to be applied.

1.4 Thesis Outline

The following chapter will give a brief background to the computational modelling of learning and relate some details of an earlier program which considered the game of GoMoku. The third chapter describes the structure of the present program and is followed by three chapters detailing some aspects of the system, namely, Representation, Pattern Matching and the use of Responses.

Chapter 7 provides a "worked example", illustrating the operation of these components of the system over the period of a few moves, and is followed by a chapter suggesting some general conclusions. Two appendices provide a brief background to the game of GoMoku itself, and to the POP-2 programming language.

Chapter 2

Background

2.1 Learning

Early programs which were intended to learn, or improve their performance with experience, were naturally based on prevailing general problem solving techniques. The most significant example of this is the adaption of graph-traversing or tree-searching techniques to learning. "Trees" arise in problems such as game-playing and puzzle-solving, where each move or step gives rise to a branching point. In all but the simplest of games and puzzles the trees are too large to be searched exhaustively. It is often possible to assign a "value" to any particular node of a tree, in terms of the likelihood of the goal being achieved from the node. Various techniques may then be used to "prune" the tree, by considering only those paths which lead to "good" nodes. A program using such an evaluation function can be made adaptive by enabling it to modify the evaluation function.

One of the best known examples of the use of this technique is a Draughts-playing program [Samuel, 1959]. Here, the evaluation function is divided into a set of functions, each sensitive to different aspects of positions in the game. These functions are weighted to produce the final evaluation. The weights may be adjusted after a game depending on whether it was won or lost, and on how the states which were traversed were evaluated. The

general problem of adjusting the weights associated with the component functions may be abstracted: In the topological space spanned by the functions, coefficients must be determined to define a surface which optimally separates the "desirable" states from the "undesirable". This general problem is pursued in the book, "Perceptrons" [Minsky & Papert, 1969], which chiefly considers it in the context of learning to recognise patterns.

The evaluation-function approach is avoided in the present work, as the components of the evaluation function embody domain-specific knowledge and at the same time inhibit the acquisition of any concepts not thus embodied.

The use of production systems as a way of describing a program is another technique that has lent itself to the modelling of learning. Production rules were initially used for the formal description of Grammers [Chomsky, 1965]. In its simplest form, a production is a pair of strings of symbols. The interpretation is that the string on the left hand side gives rise to the one on the right. Production systems are now often used to describe formally the syntax of programming languages. Examples are ALGOL [Van Wijngaarden et al, 1976] and POP-2 [Burstall et al, 1971]. In heuristic programs employing a "state vector" representation of the domain, each element of the state vector will fall in a given range. The heuristics themselves can be represented as sets of productions by

allowing the symbols used in the production rules to denote subsets of these ranges.

Conventional programming languages have too complex a syntax to make it feasible for a program to modify its own behaviour by editing its own program text. The simple, uniform syntax of production systems does, however, facilitate program self-modification. One method is to represent the heuristics as described above and permit the addition of further productions and the modification of the subranges specified in existing productions. This is essentially the technique used in Waterman's poker-playing program [Waterman, 1970], which has two main classes of production. The first of these, his "action rules", indicate what should be done in a situation of a specific type. The second, the "backward form" rules, determine what constitutes such a situation. The program learns both by the acquisition of new productions and the modification of existing ones.

The "state-vector" representation of a domain is highly specific to the domain. The composition of the state-vector depends on the programmer's knowledge of the game. One objective of the current system is to employ a scheme of representation that is independent of the domain.

Learning can reasonably be defined as the acquisition of information or skill. For the present purposes, information and skill are superficially distinguished by their

respective rôles: Information is thought of as passive and subdivides into facts and definitions, while skill is active, dividing into procedures and responses. The term "Knowledge" will be used to embrace all of these catagories.

Facts are simple statements or assertions, generally qualifying some object. For example:

"Roses are red."

Definitions are sets of descriptions which apply to all instances of whatever it is being defined. For example:

"Rose: A beautiful and usually fragrant flower which grows upon a shrub of the genus rosa, usually of a red, white or yellow colour." [Shorter Oxford English Dictionary]

Procedures embody how something may be accomplished:

"To ring alarm, break glass."

Responses are embodiments of how to behave in specific situations:

"When in doubt, scream and shout."

These distinctions can become rather blurred, as they depend on context. The statement of a procedure, for example, might be regarded as a fact. Indeed, all four of these divisions have the same structure, being associative pairs. Facts associate an object with a qualification; Definitions associate a label with a set of descriptions;

Procedures associate a goal state with a set of actions, and Responses associate a stimulus with a set of actions.

2.2 Models of Learning

The four categories of knowledge each have their analogue in computational terms. Facts are the simplest. A file containing a list of names and addresses is a collection of facts, as is any simple data-base.

Definitions find their analogue in logical predicates. These are a series of tests, on the basis of which an object may qualify as a member of a set. Definitions may take the form of "templates" which the object must fit to qualify, and a routine to match objects to templates must be provided.

Procedures correspond to subroutines. In the language PLANNER and its derivatives, procedures may be invoked according to an associated pattern.

Responses have their counterparts in these same languages, in the form of antecedent theorems, or "demons". These are subroutines which are invoked when an assertion is made that matches their associated pattern. Programs have been devised which make extensive use of demons. Their use is described in a model of story comprehension [Charniak, 1972] where they are employed as "facts" waiting for an occasion on which to be useful.

It is suggested above that learning may be regarded as the acquisition of facts, definitions, procedures and responses. The first of these is relatively trivial: The addition of a fact to a data-base requires merely its integration into whatever structure has been imposed on the data-base. For instance, if a file of names and addresses is ordered with respect to surnames, the addition of a new datum is accomplished by inserting it so that the order is maintained. While fact acquisition is commonplace in computing applications, this cannot be said of the remaining forms of learning.

Definitions are set-inclusion rules. The term "concept" has sometimes been used in AI literature to describe the principle of learning to determine whether an object fits the description of elements of a set [e.g. Church, 1956]. An early attempt to model definition acquisition, or concept formation, took the form of a program to simulate the responses of human subjects of psychological experiments, in which the subjects had to formulate rules to describe sets of objects. The model included consideration of whether the objects were presented one by one, or all at once [Hunt and Hovland, 1963]. A more recent model design, related to computer vision, derived structural descriptions of such things as "arches" by generalising an example and modifying the generalisation in the light of further examples and counter-examples [Winston, 1970].

The acquisition of procedures is the essence of the dreamed-of self programming computer. While the reality of such machines is still on the distant horizon, some progress towards a model of this process has been made. The Macro-operations (MACROPS) used in STRIPS [Fikes, Hart and Nilsson, 1971 & 1972] are simple examples of such programs. STRIPS is a problem solver which generates robot plans to achieve goals specified as well formed formulae in the predicate calculus. The STRIPS world is similarly described in terms of well formed formulae. Having solved a problem, the system saves the solution in a generalised form for future use. Generalisation is achieved by replacing constants in the plans with variables but ensuring that these variables coincide, where this is necessary to the original plan. The representation additionally permits each subplan in the plan to be retrieved. If a plan is a subplan of one developed later, then it is replaced by that plan and in this way long plans tend to supercede shorter ones.

The LISP language [McCarthy et al, 1962] was intended to facilitate programs which could manipulate programs. The language basically manipulates lists, and programs themselves are represented by lists. However, it was not until the advent of extensions to LISP such as PLANNER [Hewitt, 1972] and CONNIVER [Sussman, McDermott, 1972] that this became effective. These languages provided a generalised procedure-calling mechanism, whereby procedures could readily be added to or removed from the set

of procedures eligible for calling at a given point in a program. This facility was combined with an associative database. HACKER [Sussman, 1973] was probably the first program to model the process of programming and debugging. Programming was accomplished using knowledge of procedures which could be called to achieve certain goals. The debugger used specific information given by the domain on what had gone wrong when the program was run, and the programming component could then be invoked to correct the program.

Limitations on the patterns used in these languages render them unsuited to the description of the structures employed in the present program.

Waterman's poker program is essentially a model of response acquisition. An action rule indicates exactly what action is to be taken when the state vector matches its left hand side. The program is able to create or modifiy production rules, thereby modifying its responses

The present work concentrates on skill acquisition. Procedures and responses are closely related in form. A procedure may contain a statement of a precondition which must hold if it is to be applied. It contains a set of actions, the procedure body, and a statement of what it is used for, its "pattern". Responses contain a description of the conditions under which they should be applied. This is their "pattern". Naturally they also have their

sets of actions which are usually initiated when a match is made to their pattern. Finally, they may contain a description of their outcome. Thus, in their fullest forms, procedures and responses are both of the following composition:

STATE -> (ACTIONS) -> STATE

They differ, however, by being accessed through their "goal-state" on the one hand and by their "stimulus-state" on the other. It is conceivable that a system could be devised that used the same objects in both senses.

2.3 Generalisation

The essence of skill learning is generalisation. A lesson is learnt in a specific situation. To be useful in similar situations it must be assimilated in terms of what the situations have in common. Generalisation of a specific event may result in one of a whole spectrum of descriptions, from the near-specific to the totally abstract. In computational terms, generalisation can be seen as the replacement of constants by constrained variables, and of constrained variables by less constrained variables. A constant may belong to a whole sequence of nested sets, each less constrained than its subsets and therefore an abstraction of its subsets. For example, a particular tree may be primarily a pine-tree, then a conifer, a tree, and finally a vegetable. Naturally, generalisation cannot take place if the successive catagories are not known, or if the constraints are not explicit. A frequent problem is to know how far to proceed

with the process of generalising. If a concept is not general enough, it may not be accessed in some situations where it is pertinent. If it is too general, it may be accessed too often, in situations where it is irrelevant. To describe a tree as a "thing" (a member of the universal set) is not often useful! This problem arises particularly when a generalisation is made from a single example. The approach in the program hypothesised by Winston, mentioned above, is to generalise as little as possible from single instances, and to modify such generalisations in the light of successive examples and counterexamples.

Another approach is to have a "teacher" provide the program with the correct generalisations. This technique is employed in Waterman's poker playing program, which has a facility for entering "training" information at any stage of a game. The information consists of a good decision to make in the situation, the elements of the "state vector" which are relevant to the decision and the reason for the decision in terms of these elements. The training information amounts to a new production rule, although it may be used to modify other productions rather than simply be added to the production system. An advice-taking, chess program [Zorbrist and Carlson, 1973] also permits an "expert" to provide the program with generalised patterns which represent aspects of the chess position, together with "weights" to apply when incorporating the detection of a pattern instance into the evaluation of a position.

This evaluation is then used in a normal tree pruning heuristic.

The present program does generalise from single examples. However, the aspect of the state that it is describing always relates to an instance of its goal, of which it naturally has a generalised description. The level of generalisation is "borrowed" from this description in a manner which will be described later.

2.4 GoMoku and Learning

The game of GoMoku has been used in earlier experimental learning programs. One example in particular can be compared with the present work [Elcock and Murray, 1967]. The system centred around a technique referred to as "Backtrack Analysis". An attempt was made to be independent of the system of formal description employed, except at the interfacing level. However, it was assumed that the description would provide a ranking of the possible moves. The system accumulated a list of descriptions which it regarded as subgoals. These were assumed to represent positions from which a win might always be forced. The list was ordered according to the number of moves required to reach a winning position, the Level of a subgoal. When the program was to move, it considered all possible moves, generating a description of each resultant position. These were compared with the descriptions in the subgoal list. The best move was that which matched

one of the subgoals and had the least number of moves to play to win.

The descriptions were acquired by the Backtrack Analysis Component of the program. This was activated at the end of a game which had been lost by the program. The moves of the game were "unplayed" in reverse order, until the point was reached where the opponent had created a position which was not on the subgoal list. The description of this position became a new subgoal. In adding it to the subgoal list, the program took care to remove any descriptions of the same level which included the new subgoal, thereby ensuring a tendancy towards "minimal" descriptions of subgoals.

An example of the form the descriptive language may take is given.

> A 7-Pattern is a number, N: Consider a line of seven vertices. If it is not possible to construct a line of five stones of the same colour on the line, then the value of the 7-Pattern is 0. Otherwise the value is the number of stones already in place in the line.
>
> A Line-Pattern is a pair, (n,r): Consider a set of colinear 7-Patterns going through a given vertex. The first value, n, is the highest value in the set. The second, r, is the number of elements in the set with the value n.

An Ordering is defined on Line-Patterns by ordering first on n, and then on r.

A descriptive system which considers only the "best" Line-Pattern through each vertex would be capable of describing open-fours, which are essentially one-dimensional. To describe two-dimensional patterns such as intersecting-threes would require the inclusion of the two best Line-Patterns through a vertex. It was acknowledged that the descriptive language might be inadequate to describe certain complex situations and some consideration was given to the detection of inadequate descriptions.

An important difference between their system and the present one is that the descriptions they derived were used as subgoals. Their program considered all possible moves to determine whether any generated an instance of a description. As only a one move look-ahead was employed, the effect is equivalent to the present scheme where the description has the effect of a move "undone" and the match is always to the current state of the domain, with the look-ahead obviated. If it were not for the burden that the present representation places on the pattern matcher, its advantage over such a look-ahead system would be conclusive. The major difference between the two systems is the attitude towards descriptive language. The philosophy behind the Elcock and Murray program is summerised in their words:

> "If the formal language has been suitably designed, then this description will automatically generalise the abstract board situation."

Their descriptive language was designed in anticipation of the concepts which emerge from the game, and was therefore in terms of exactly those features which generalise any particular position. Features such as the number of stones of a colour in a line of seven vertices might be primitive to the language! The philosophy behind the descriptive medium of the present program is simply that the goals and domain elements are represented in a uniform and malleable way, and that it is up to the program what else is to be described. This approach shows itself to be the more general by starting with a weak descriptive language (relative to the specific domain) and still arriving at the necessary concepts.

Elcock and Murray followed up their work with a program in which the descriptive language was further refined to broaden the scope of its representation and to improve its powers of generalisation [Murray and Elcock, 1968]. This development moved away from the approach advocated in the present work. They provided a catalogue of structures with which their program was able to play "expertly". A few of these are shown in Appendix A. In principle, all of these can be represented in the more general structural terms of the present system.

The examples of learning programs mentioned above are but a few of many. They serve to illustrate the main techniques that have been employed, and some comparison has been drawn between these techniques and those adopted in the present system.

Chapter 3

System Overview

3.1 Model Requisites

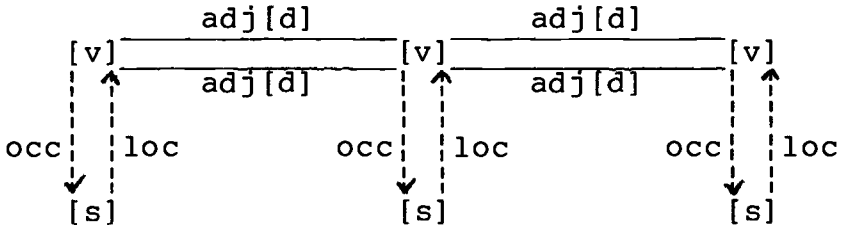
It is usual in Artificial Intelligence to consider a model as comprising two components. The first of these is the domain, sometimes referred to as the environment, context or world. The second part is the performer, often given a name and personified, or else referred to as "the program" or "the model". While the emphasis is usually on the behavior of the performer, there are invariably assumptions made about the domain: If the domain is not a good model of the relevant aspects of the real world it may not be possible to draw any conclusions about the validity of the performer model. The game of GoMoku, which forms the domain of the present model, has already been introduced.

Typically, to describe a domain one needs to represent objects, the relationships which may pertain between objects, sets of objects, the state of the world, the changes which can take place and the performer's goals. The performer would comprise such components as a planning mechanism, a pattern matcher and a description generator.

In the present model the domain contains objects and relationships. The changes which may take place are described in terms of actions which may be taken. The goal is described as a structure. A structure is a collection of variables, each constrained to contain only specific types

of object. The variables are further constrained by relationships which must exist between objects assigned to them.

In the GoMoku domain the objects are the stones and the board. The latter may be regarded as a collection of vertices. The relationships which may exist between these objects are occupancy and adjacency. Occupancy pertains between a stone and a vertex and is in fact described as the complementary pair of relationships, "occupant" and "location". Adjacency pertains between a pair of vertices and a direction of adjacency has to be specified. This is in order to describe the fact that the goal requires the five stones to be in a straight line. These objects and relationships allow the goals to be described as a set of typed variables. Figure 3.1 is a diagram of such a description.



Key

[v] - variable of type vertex
[s] - variable of type stone
[d] - variable of type direction
adj - Adjacent
occ - Occupant
loc - Location

Figure 3.1 - "Line-of-three" structure

Direction could, of course, have been a separate relationship, but as it is only applicable to adjacent

vertices, and essentially distinguishes between the adjuncts to a vertex, it was combined with the adjacency relationship. If the program had provision for more general descriptions of sets, it might have been possible simply to apply the colinearity requirement to the set of stones or occupied vertices.

The performer models a learning process through which both procedures and responses are acquired. The elements of this model are an elementary planning mechanism, a description generator and a pattern matcher. The planning mechanism is completely ad hoc, as planning is not central to the model. The planner is able to describe a hypothetical instance of the goal structure and propose a sequence of actions which would lead to its achievement. The description generator is able to form generalised descriptions of situations which arise during the course of the performer's activity. The pattern matcher is an extensive mechanism which continually monitors the domain for instances of certain structures, reporting if they arise.

The performer must attempt to manipulate objects until valid, one-to-one assignments can be made to all the variables in the goal structure. Such an assignment will be referred to as an Instance of the structure. There is a similarly described counter-goal, of which an instance must at no time exist.

Actions are represented in terms of those relationships which the program can alter. The only action in GoMoku is the placement of stones. This can be described as changing the occupant of a vertex from being empty (undefined) to being a stone. The location of the stone is simultaneously changed from being undefined to being the vertex. Explicit in this representation are the preconditions that the vertex may not have any other occupant, nor may the stone be at any other location. If the performer tries to act contrary to these conditions the action fails and the performer can then see which of the preconditions was violated, and perhaps learn something from the situation.

3.2 Control Sequence

Figure 3.2 depicts the main processes within the system and the data structures through which they communicate. The arrows depict the direction of flow of information between the processes and the data structures. The remainder of this section will describe these functions.

The Action-Sequencer operates from the Action-List, which contains scheduled actions, devised either to achieve its own goal, or to thwart its opponent. Moves from this list are alternated with moves by the opponent. If the list is empty, the primitive Planning-Mechanism is invoked to provide an appropriate set of actions. Usually this is only necessary at the beginning of a game.

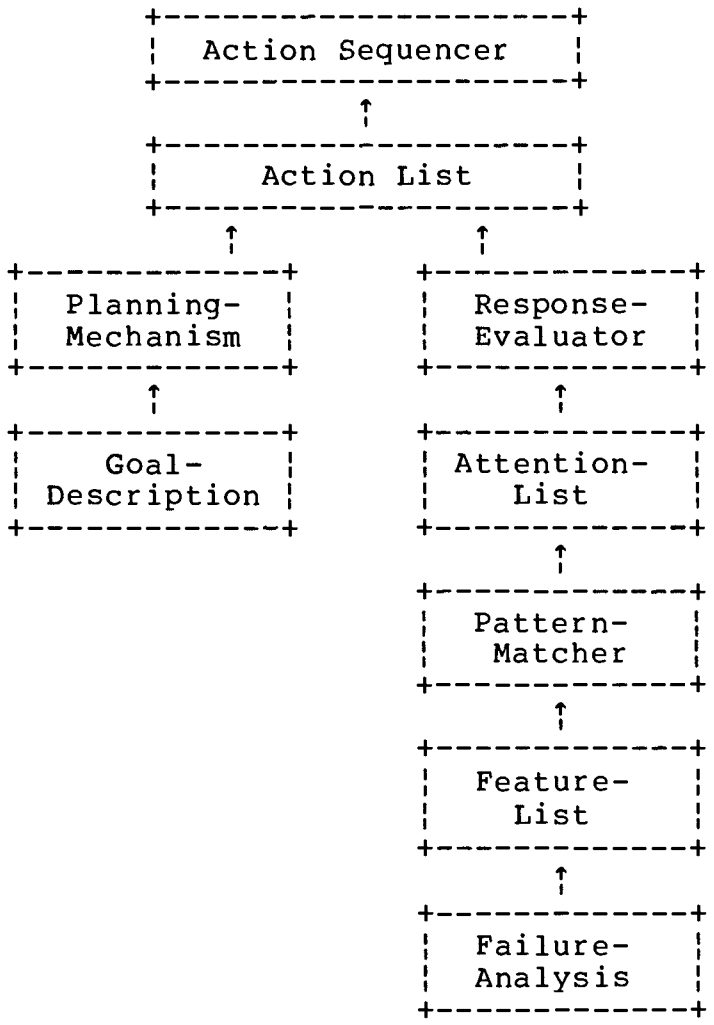


Figure 3.2 Chief Processes and Associated Data

A side effect of making a move is to invoke the Pattern-Matcher. This acts on the Feature-list, containing structural descriptions which the program has previously acquired. The objects involved in the most recent action are regarded as candidates for assignment to variables in these descriptions. There may also be suspended partial matches associated with the objects by previous operations of the Pattern-Matcher, and an attempt is made to resume these. Any completed matches are placed in another list, the Attention-List.

After an attempted move by the Action-Sequencer, it considers whether or not the move was completed. If it was, the Move-Sequencer also checks further to determine whether it has completed its goal, announcing its victory if it has. If the action was in some way inhibited, by being blocked, for example, the Failure-Analysis component is invoked to try to establish what happened. The remaining actions associated with the plan are discarded, as failure of the action implies failure of the plan. The Failure-Analysis, in this case, determines why the move could not be made and which event in the game brought about the circumstances preventing the move. By reversing the effect of that event in the context of the generalised goal structure, a description of the unblocked position is obtained. This is then included in the Feature-List for the purposes of the Pattern-Matcher.

After a move by the opponent, the Attention-List is examined by the Response-Evaluator. If the opponent has won, or if his victory may be anticipated, the Failure-Analysis component is also invoked, this time to describe the position immediately before the move just completed. The result is also added to the Feature-List. On the other hand, if a threat is present, this is considered in case the current plan has to be suspended. Other constructive moves may also be proposed, and if one of these is better

than that scheduled, in terms of plan length, the current plan may be suspended anyway. Suspension of a current plan entails placing the suggested actions ahead of it in the Action-List. Failed plans are removed from the Action-List, but suspended plans remain for the program to fall back on.

3.3 Size and Speed

The size of the program is variable, due to the manner in which records are created during a game and released afterwards. The figures below describe this behaviour. Estimated sizes are expressed in "K" (thousands of memory words).

After initial compilation:	- 9 K
At the end of a game, having created new descriptions and including information produced by the Pattern-Matcher:	- 90 K
Before another game, having abandoned all information specific to the previous game, but retaining the descriptions gained:	- 11 K

The program's speed is likewise variable, decreasing with the increasing number of descriptions with which the Pattern-Matcher has to deal. Speed was not an objective

of the project: Even so, the results are disappointing! Estimates are given of typical times taken before each of its moves, at different stages of development.

No patterns	-	2 sec
Line-of-Four	-	10 sec
Open-Four	-	2 min
Intersecting-Threes	-	7 min

Chapter 4

Representation

4.1 Records

The system employs a set of Records to represent the various items that it manipulates. Although the records used all have the same essential structure, they have different functions, designated by their Use. The main uses of the records are:

Object-Definitions - Used to associate information with different types of object, and to restrict the type of object that a variable may represent.

Objects - The basic manipulatable entities of the domain.

Variables - Used in structural descriptions to stand for objects that are not specifically determined.

Relationships - Used to impose constraints, other than type, on variables.

Structures - Collections of related variables, representing configurations of objects.

Action-Definitions - Descriptions of the changes that the system may make within the domain

Actions - Proposed or implemented changes to the domain.

POP-2 facilitates the definition of arbitrary compound items with accessible component items. A single such data-structure is defined in the program to permit the consistent representation of all aspects of the present model. The basic structure is a triple, comprising three components. Two of these components form an associative pair, while the third always contains another triple. This allows the triples to be "chained" together. The term "record" is used throughout to refer to such chains.

The first triple in a record is always the header, used to identify the record use. The first component of a header is a word, such as OBJ, DFN or ACT, designating an Object-Record, a Definition-Record or an Action-Record respectively. The second header component is simply a number, unique to the record, with the sole purpose of identifying the record in reports from the program. In all such printing, the first two components of the header triple are printed between double angle-brackets, thus:

<<OBJ 27>>

The triples following the header triple are name/value pairs. The name component is always a word, for example

OCCUPANT or RANGE. The value component may in general be any POP-2 item, such as a word, list, record or function. When records are printed in full, as in the figures below, the names and associated values appear in two columns below the record header.

4.2 Object-Definitions

It is useful to be able to associate information with different types of object, and this is the function of the Object-Definition records. The property names of these records are used to access individual pieces of information. An Object-Definition record has the folowing properties:

Name — This is not used by the system itself, but is helpful when the program's reports are examined.

Props — The properties of the object that are pertinent to the various structure processing functions are listed.

Genfn — An object "generator" is also provided, for use in planning and playing. A stone generator, for example, would correspond to the pile of stones at the player's disposal.

Acts — A list of actions which may be used to manipulate the objects is also provided.

In the example in figure 4.1, a VERTEX is "defined". The pertinent property of a vertex is its OCCUPANT. A function called GENVTX will "randomly" generate vertices, and the only action involving vertices is that described by the Action-Definition record, <<ACD 13>>.

```
<<DFN 1>>
        NAME        VERTEX
        PROPS       [OCCUPANT]
        GENFN       <function>GENVTX
        ACTS        [<<ACD 13>>]
```

Figure 4.1 - An Object-Definition Record

4.3 Objects

A distinct record represents each distinct object. Each Object record has a property called TYPE. The value of this property is an Object-Definition, denoting the object type. Apart from the TYPE property, objects have a group of properties which are important to the description of structures. As has been seen, the definition record has a list of these properties. For vertices this list has only one property, namely OCCUPANT. However, stones have two such properties, LOCATION and COLOUR. These properties are used, for example, in planning and in pattern matching. The property lists are also used as a generally convenient place to hang an assortment of "system" information in the course of running the program. This can be seen in figure 4.2 below, and also in the discussions of some of the program components.

```
<<OBJ 27>>
        TYPE        <<DFN  1>>
        OCCUPANT    UNDEF
        XCO         17
        YCO         9
```

Figure 4.2 - An Object-Record

The Object represented in the figure is a vertex, which is indicated by the TYPE property. The OCCUPANT property indicates that the vertex is unoccupied. The remaining properties give the vertex coordinates, which are only used indirectly through certain system functions.

Objects may be related to each other directly through their property lists. For example, a particular stone may be related to a given vertex through the OCCUPANT property of the vertex. The vertex would likewise be related to the stone through the LOCATION property of the stone. Thus, when a stone has been placed on a vertex, the respective records would refer to each other as shown in figure 4.3.

```
<<OBJ 28>>
        TYPE        <<DFN  2>>
        LOCATION    <<OBJ 27>>
        COLOUR      <<OBJ  3>>

<<OBJ 27>>
        TYPE        <<DFN  1>>
        OCCUPANT    <<OBJ 28>>
        XCO         17
        YCO         9
```

Figure 4.3 - A Stone on a Vertex

The second Object-Definition record in the figure is that of stones. Colours are also represented as objects, for consistency in the pattern matching process.

4.4 Variables

To describe a structure in a general way, the use of variables is required. In the present system, the function of variable is filled by so-called "Ghost" records. These too have property lists, including a TYPE property, by which the Ghost record is constrained to represent a particular type of Object. In a structural description, Ghost records take the place of the actual objects which might occur in an instance of the structure.

In an instance of the structure, various relationships pertain between the Objects involved. It is necessary to describe these relationships as existing between the variables in the generalised structural descriptions. The property lists of the Ghost records describe the relationships which must exist between corresponding Objects in an instance of the structure. Property-list relationships between Objects are implied simply by the presence of a property of the appropriate name in the Ghost record, the associated value being the related Ghost.

Figure 4.4 is a general representation of a stone on a vertex. The Ghost representing the vertex has the Ghost representing the stone as its OCCUPANT value, and the reciprocal LOCATION of the Ghost stone is similarly filled

by the Ghost vertex. In the figure, <<DFN 0>> defines colour Objects.

```
<<GHO 23>>
        TYPE        <<DFN  1>>
        OCCUPANT    <<GHO 24>>

<<GHO 24>>
        TYPE        <<DFN  2>>
        LOCATION    <<GHO 23>>
        COLOUR      <<GHO 20>>

<<GHO 20>>
        TYPE        <<DFN  0>>
```

Figure 4.4 - Abstraction of a Stone on a Vertex

4.5 Relationships

The relationships which may exist between two objects fall into two categories. The first of these occurs when two objects are related directly through their property lists, and does not involve the use of any other records. This relationship has been denoted "property" relationship, and has been dealt with in the two previous sections. This section deals with "functional" relationships.

Functional relationships relate two objects to each other in such a way as to associate a value with the relationship. For example, two vertices may be adjacent to each other in one of eight directions. There are two ways in which functional relationships are employed: In planning it is desirable to be able to generate an object from another object, given a relationship and an associated value; In matching, it is necessary to be able to

determine the value of the relationship between two objects, sometimes simply for verification.

```
<<REL 12>>
        RELFN     <function>ADJFUN
        RELGEN    <function>ADJGEN
        RANGE     [ - list of all eight
                              directions - ]
```

Figure 4.5 - The Adjacency Relationship

The only example of a functional relationship in the GoMoku domain is the adjacency relationship. Two adjacent vertices have an associated direction. As this is an unchanging property of the board on which the game is played, it is unnecessary to mention it in the property lists of the objects themselves. Instead, two functions are provided: One determines whether two vertices are adjacent, supplying the direction if they are. The other generates a vertex adjacent to another, given vertex, in a specified direction. These are held together in a relationship-record, as in figure 4.5. The range of the function is supplied for the use of the planner.

Functional relationships are associated with Ghosts through CONSTRAINT properties. The value of this property is a list of relationship/Ghost/value combinations. The value referred to may itself be a Ghost. For example, two adjacent vertices are represented in figure 4.6. Note that <<REL 12>> is the adjacency relationship described earlier. Ghosts 30 and 31 both represent directions,

their opposition being a consequence of the relationship functions.

```
<<GHO 23>>
        TYPE        <<DFN  1>>
        CONSTR      [ [<<REL 12>> <<GHO 25>>
                                  <<GHO 30>>] ]

<<GHO 25>>
        TYPE        <<DFN  1>>
        CONSTR      [ [<<REL 12>> <<GHO 23>>
                                  <<GHO 31>>] ]
```

Figure 4.6 - Abstraction of Adjacent Vertices

4.6 Structures

The goals of the system are represented as structures. There are three kinds of structure used: Basic-Structures are sets of typed variables together with a set of relationships obtaining between the variables. A Reduced-Structure is a basic structure together with a set of additional constraints (relationships) on its variables. A Compound-Structure is a set of reduced structures or compound structures and a set of correspondences between their variables.

A basic structure such as a line of five is represented by a set of Ghosts, all interrelated by both property and functional relationships. In the course of its experience, however, the program will need to describe certain situations in terms of such structures. One of the ways in which this is done results in a reduced structure. Reduced structures use the same pairing representation as the other records but instead of associating name with

value, they associate the Ghosts of the basic structure with new property lists, thereby imposing additional constraints on the Ghosts. Any property appearing in this new list is regarded as superceding the corresponding property of the original Ghost. In this way the Ghosts keep all their previous properties except for those explicitly overridden.

For example, the reduced structure describing a line of four stones refers to the original basic structure representing a line of five stones. An additional property list is associated with one of the Ghost vertices, specifying that it is to have no occupant.

The second way of describing new structures in terms of old is as compound structures. If two lines of four intersect, at least one of the vertices in the first line corresponds to a vertex in the second. There is a similar correspondance between the occupants of the corresponding vertices. A compound structure refers to two previously defined structures and further specifies a correspondence between their Ghosts. This means that any object assigned to a Ghost in one structure must be assigned to the corresponding Ghost in the other.

In the example of two intersecting lines of four, at least one of the Ghost vertices of the first line has a corresponding Ghost vertex in the second. This correspondence represents the point of intersection of the two lines. If

the vertex is to be occupied, there will also be a pair of corresponding Ghost stones.

Compound Structures have their own sets of Ghosts through which the correspondences between the Ghosts belonging to the sub-structures is established. A compound structure is therefore represented as a list of structures. Associated with each structure is a list which pairs the Ghosts of the structure with those of the compound structure. For example, if S1 and S2 are structures, the Ghosts G1 belonging to S1 and G4 belonging to S2 are shown to correspond in figure 4.7.

```
S1:    [ G1:G1' G2:G2' ]
S2:    [ G3:G3' G4:G1' ]
```

Figure 4.7 - Correspondance in a Compound Structure

The Ghosts G1', G2' and G3' are those belonging to the compound structure and imply the correspondences between the Ghosts of the component structure.

4.7 Action Definitions

As with Objects, each kind of Action has a single defining record to which all actual Actions of that kind point. The Action-Definition contains a list of property names effected by the Action and a description of the effect on the properties. The example from the GoMoku domain is the placement of a stone. The properties effected are OCCUPANT and LOCATION. The Action changes the occupant of

a vertex from being UNDEF (empty) to being a stone, and the location of the stone from UNDEF to the vertex. The Action refers to two Ghosts to represent the vertex and the stone. Figure 4.8 shows the Action-Definition for the PLACE action.

```
<<ACD  7>>
       PROPS      [OCCUPANT LOCATION]
       OCCUPANT   [<<GHO 12>>  UNDEF  <<GHO 13>> ]
       LOCATION   [<<GHO 13>>  UNDEF  <<GHO 12>> ]
```

Figure 4.8 - An Action-Definition Record

The second property line can be read as: "The occupant of Ghost 12 (a vertex) is changed from UNDEF to Ghost 13 (a stone). The next line describes the reciprocal effect.

4.8 Actions

The Ghosts in the Action-Definition are again variables. An actual Action is a record which has a TYPE, which indicates the Action-Definition, and a binding list which associates actual Objects with the Ghosts of the Action-Definition. Such records enable reference to be made to Actions both before and after they have been taken. A plan is essentially a list of proposed Actions. When an Action is successfully completed the specified changes are actually made. The stone and vertex object-records, for example, have their property lists amended so that the stone becomes the OCCUPANT value of the vertex and the vertex becomes the LOCATION value of the stone.

Chapter 5

Pattern Matching

5.1 Assignment to Variables

The task of the Pattern-Matcher is to identify instances of structures as they arise in the model domain. The process of matching a structural description to a part of the domain is equivalent to making assignments to each variable in the description in such a way that none of the specified relationships are violated. The structures in question arise as a result of the occurrence of some situation in which the program's current plan has been thwarted, by being blocked or by being beaten. Thus, when the pattern matcher actually detects a structure instance, it signifies that a potentially dangerous situation has arisen, and may provide information about the threat that can be used to avoid the danger. The matcher must find all possible matches after each action, as any given position need not necessarily be completely described by a single structure. Once all the matches have been made, the position must be assessed in order to determine whether or not to suspend the original plan in order to deal with the situation. This function is the task of the Response-Evaluator, to which the completed matches are passed.

To complete a match to a structure, each ghost-record in the description must be paired with an actual object in the domain, which must be of the same type as the ghost.

Futhermore, relationships described in the ghost-structure must hold between the corresponding objects thus assigned. For example, if an actual vertex is to be assigned to a ghost vertex, and the ghost vertex has a ghost stone as its OCCUPANT, then the assignment is only possible if the vertex is itself occupied by a stone which can in turn be assigned to the ghost stone. The same principle applies to functional relationships between two ghost-records. In this case, if the function value has already been determined, the objects assigned to the ghosts must both obey the same functional relationships and have the same function value. The generator function associated with the relationship may be used to derive the second object, given the first object and the function value. For instance, a structure might describe a ghost vertex with a second ghost vertex adjacent and above it. If an actual vertex is assigned to the first ghost, the vertex above it can be determined. It must be possible to assign this vertex to the second ghost. If the function value is not already assigned, it is necessary to consider the entire range of the relationship function. In the case of the adjacency relationship, this would amount to examining all adjuncts to a vertex.

```
<<MATCH 89>>
        STR         <<STR 32>>
        BINDLST     <ptriple>
```

Figure 5.1 - A Match-Record

To record the progress of each application of the pattern matcher, a Match-Record is created and maintained. The structure of these records is shown in figure 5.1. The value of the STR property is the structure-record to which the matcher is being applied. The BINDLST value is a chain of triples, pairing ghosts from the structure with objects from the domain. This binding list gets extended as each successful assignment is made.

5.2 Commencing a Match

It would be undesirable and impractical to have the pattern matcher continually and arbitrarily processing the domain, attempting to match up all of its target structures. Opportunity for complete matches can only arise after some change has taken place, and any consequent match must involve the objects which participated in the change. The pattern matcher need only therefore be concerned after every successful action taken, and only the objects affected need be considered as "starting-points" for the matching process. Moreover, only structures which refer to objects of the same type as those affected need be regarded as candidates for the process.

When the pattern matcher attempts to find an instance of a particular structure, it may assume that if it is successful, some of the objects involved in the most recent action will be included in the match. It therefore commences its task with at least one candidate object for assignment to a ghost in the structure. If there is more

than one ghost of the same type as this object, however, it cannot say which of these should be paired with the object. To avoid an unneccessary expansion in the amount of work the program has to do, it makes the assumption that the first ghost it comes across which has the right type and satisfies some preliminary checks, can be paired with the object. If the assumption is incorrect, or if there is more than one valid assignment, leading to different matches, then a seperate attempt at matching the same structure, "starting" with a different object, will contain the correct pairing of this object with a ghost.

Thus, when a stone is placed on a vertex, all structures which include occupied vertices may be matched. Each structure is likely to have more than one occupied vertex, and therefore more than one way in which the stone and vertex may be assigned within it. Instead of investigating all such possible assignments, the pattern matcher examines only the first to arise. If, for example, the structure describes two occupied ghost vertices, then the actual vertex involved in the latest action may be assigned to either the first or the second of these. Assignment to the second implies some other particular vertex assigned to the first. If a later event makes this assignment, the Pattern-Matcher will derive the previously ignored match.

5.3 Continuing a Match

The assignment of a given object to a particular ghost in

a structure inevitably implies further assignments of objects to ghosts. As an assignment only succeeds if these implicit assignments succeed too, and so on, the matching process is naturally multiply recursive. Because the structures are by nature networks, a given assignment may eventually depend on itself. As long as a set of assignments remains self-consistent, it is regarded as valid.

Different processes are involved in propagating the matching activity through each of the two kinds of relationship. Confirming that the property relationships match is relatively straightforward. For each pertinent property of the object and ghost being paired, the associated values are obtained from both the object and the ghost. This should yield another object and ghost pair which are in turn processed in the same way. However, to confirm that the functional relationships match, there may be further problems. These arise when the function value is not yet determined, implying that the actual object referred to in the relationship is likewise undetermined. For the match to continue, the entire range of the relationship function has to be considered and the matching process subdivides into as many branches as there are objects obtained in this way. Most of these branches would be expected to terminate quite soon by failing, but if this is not the case, severe combinatorial problems arise, particularly if there is more than one branching point. In the present domain, this problem is not severe,

but in another domain, the strategy may have to be revised. If a match is completed, the system is informed of the fact.

5.4 Suspending a Match

If, during the matching process, an object and a ghost fail to match due to conflicting property constraints, two alternatives arise: It may or may not be possible that, in some future state of the domain, the property of the object may assume a value which does match. In the context of the GoMoku domain, if a vertex is required to have a white stone on it but it is in fact empty, it may at some later time in fact have a white stone placed upon it. If, however, it has a black stone occupying it, there is no way that it can later support a white stone. In the former case it is desirable to be able to "suspend" the matcher so as to avoid repetition of the work already done on that match. In fact the scheme of only starting one match to each structure with a given object demands that matches reaching this state should be later continuable, as the particular match information would otherwise never be recovered.

The program adopts a relatively simplistic approach to the problem of deciding whether to suspend or terminate a match. In a case where an object and a ghost would match but for a property failing to match, all the actions possible in the world that can affect the object concerned are examined. The representation of actions is such that

it is easy to determine which properties of each object type an action may affect, and the conditions under which it may be taken. If the program can determine any action which can alter the property value of the offending object, it assumes that the match should be suspended. Naturally it will sometimes suspend matches when they should be terminated, as there may still not be any sequence of actions which would cause the match to eventually succeed, but it will never terminate a match which it should have suspended. In the GoMoku domain there is only one action, that of placing a stone, so it is always correct in its assumption. Thus, when it comes across an unoccupied vertex while looking for one with a white stone, it finds that the PLACE action will change the OCCUPANT of a stone from UNDEF, and suspends the match. If however, the vertex had had a black stone on it, the system would find no action capable of changing the situation. Note that this would not be true in, say, GO, where capture is permitted, although in this case it would be far from straightforward to prove whether or not a match should be suspended.

To be able to resume a suspended match to a structure, the ability to describe the intermediate results is required. The match-record mentioned above keeps a list of the satisfactory pairings which have already been made, associated with the structure being matched. Until all assignments have been made it therefore represents a Partial-Match. The matching process should be continued

when an event occurs which involves one of the objects already assigned in the structure, as this implies that properties of the object may have changed. The suspended matches are therefore associated directly with all such objects which may yet be involved in actions. This allows the pattern matcher immediate access to all the pertinent partial matches after any particular action, and, more to the point, only the pertinent ones.

There are two possible strategies for deciding when a match should be suspended. The first is to suspend it immediately a "temporary" failure is observed. The alternative is to continue to match as much of the structure as possible before suspending. The former has the advantage of being quicker in the short term, and while not reducing the number of partial matches created, does imply that they occupy slightly less space, both by containing less information and by being referred to less often. The advantage of the latter strategy is that, while reason to suspend a match may be met at one point, evidence for killing it may be found a little later and in this way there may be less "dead wood" lying around in the form of suspended partial matches which will never be completed and may never be killed. In the event, this latter scheme was adopted, although no practical comparison was made to determine preference in terms of speed and memory requirements. In the present domain, the branching at functional relationships is not severe, there being only eight directions possible and essentially only one instance of

the relationship in the goal structure. There are in fact two different references to the relationship, but the second instance is completely determined by the first. In other domains where branching is more critical and compounded, the strategy of immediate suspension is almost certainly the better.

5.5 Matching Compound Structures

While compound structures arise under slightly different circumstances to reduced structures, they serve the same purpose and instances must be detected as soon as they occur. One approach to making matches to compound structures would be to follow the same procedures as for reduced structures, but treating the "points of intersection" specially. This has a disadvantage arising from the increased combinatorial problem associated with larger structures, and ignores the underlying nature of the compound structures.

The building blocks of compound structures are reduced structures, just as the building blocks of reduced structures are the primitive objects (or their ghosts). The only "relationship" pertaining between the component reduced structures is the notion of "intersection", or the correspondence between ghosts. The matching scheme used is in some ways analogous to that used with the reduced structures. Whereas reduced structures are associated with the definition-records of the objects they comprise, compound structures are associated with the reduced structures of

which they are composed. A simple event (action) involving an object will stimulate the reduced structure matching process, while the compound structure matching process is initiated when a successful match is made to a reduced structure. Finally, partial matches to compound structures are recorded and associated with the component structures which have not yet been matched, which is analogous to partial matches to reduced structures being associated with the objects which have held up the match.

As mentioned above, the compound structure matcher is initiated by the reduced structure matcher. Before a completed match to a reduced structure is reported to the system, each compound structure associated with the structure involved is examined for the purposes of compound matching. This always results in the formation of a partial match. Any other partial matches already associated with the compound structure are examined to attempt their continuation. Unlike the reduced structure matches, there is no simple way of telling whether a partial match to a compound structure may be killed, except when all of its component matches have been killed. This unfortunately implies that these may continue to be examined in situations where they are no longer relevent.

The results of the Pattern-Matcher, the completed matches, are entered into the Attention-List, through which it communicates with the Response-Evaluator.

Chapter 6

Responses

6.1 Matched Patterns

When the Pattern-Matcher sees any instances of structures, the completed match-records are given to the executive section of the program to be assessed in the light of the current plan. The following alternatives are possible:

1) The structure may be an instance of the opponent's goal, the program's counter-goal. The program must learn to anticipate such events.

2) The structure may warn the program of impending defeat. The program must decide whether it is necessary to block the attack.

3) Simultaneous warnings of doom may imply that defence is impossible. The program must again learn to anticipate the event.

Cases (1) and (3) result in the acquisition of responses. Reduced structures arise from (1) and Compound Structures from (3). Case (2) allows the program to make use of what it has learned. These aspects will now be treated in the enumerated order, as each provides the context for the next.

6.2 Generalisation by Reduction

The system derives reduced structures from structure instances. A structure instance is represented by a Match-Record, which contains a pairing of ghosts from the structure with objects from the domain. The nature of the situation in which the generalisation is made is such that it is not sufficient simply to recognise its reoccurrence. It has to be possible to anticipate it. For example, the present program may be thwarted when its opponent forms a line of five. It is vital that the program should be able to anticipate such an event before it happens again, and respond to the anticipation by attempting to prevent its repetition. It is therefore necessary to be able to describe the situation prior to whatever action created the undesirable position. In the case of this example, the structure matched is the line-of-five, with the stones the "wrong" colour. The program needs to describe the configuration immediately prior to the last action, which completed the line. To do this it needs to be able to take the generalised description of the line, which is part of its original knowledge, and "reverse" the effect of the last action on this description. The order of the Opponent's moves and any other moves he may have made are irrelevant at this point, as only the matched pattern is considered.

The ability to achieve the required description relies on the manner in which the system's actions are described.

The representation of actions would make it quite easy to reverse the effect of an action if this was desired. All the information needed to determine the generalised effect of undoing an action is available. The match-record provides the binding between the objects and the structure's ghosts, while the action-record describes the effect the action had on the objects. The generalised effect can be inferred through this correspondence. Instead of copying the structure description and then actually modifying the appropriate ghosts to arrive at the desired result, a reduced structure record is created. This refers to the original structure, but has additional sets of property lists associated with some of the ghosts. The new structure is interpreted as being identical to the old except where a new property list exists to override the old.

The GoMoku Line-of-Five is described as a set of ghosts, each with a property list describing its relationships with other ghosts. To "reduce" this description to that of a line of four, with one end vertex unoccupied, all that is needed is to associate a single property list with the end ghost vertex, with the OCCUPANT property explicitly undefined. The associated ghost stone has also to be marked as absent. In matching to this new description, the ghosts would all be regarded as having their original properties, except in the explicitly superceded cases.

6.3 Responding to Patterns

The procedure described above also provides further information. If the undesirable state is to be avoided it is likely to be possible to do so by preventing the action which created the state. By determining such a blocking action and associating it with the reduced structure, an appropriate response is represented. The program must be able to "notice" when instances of such structures arise. The reduced structure is entered into the Feature-List, containing all the structures which have to be noticed, and this list is used by the pattern matching component of the system. When the pattern-matcher detects any of the features with which it has been provided it supplies the playing section with the match-records which it has completed. It may be recalled that the program is busy trying to carry out a plan, and it must now compare the response indicated by the detected feature with the actions previously proposed.

The criterion used is simple. If the current plan can be completed before the threat can be realised, then the threat is disregarded. If the threat is real, however, then the program must play to prevent its fulfilment. The matched structure has a suitable action associated with it and the program thus deviates from its plan to make this blocking move.

6.4 Generalisation by Composition

The program may find itself confronted by two threats at

the same time. The example of the open-four has been mentioned, where two lines of four coincide, each requiring just one more stone to be placed to complete a line of five. If two or more different actions are needed at the same time they cannot both be taken and the undesirable outcome of one of the described situations is inevitable.

Once more it is necessary to be able to anticipate this sort of occurrence. Again, to do so, the situation prior to the action which created it must be described. There may be several such positions, and each has to be learned separately. The structures matched necessarily have objects in common: For matches to be made to the two structures at the same time they must clearly both refer to objects involved in the action which led to the completion of the matches. The description of such "intersecting" structures is accommodated by the compound structure mechanism.

Anticipation of these compound events breaks down into two more simple activities, namely the anticipation of the component events and determination of their intersection. The component events are easily foreseen by the technique described above. The already generalised reduced structure which was matched is further reduced and added to the feature list. To represent the compound structure, all that remains is to describe the intersection between the structures. The pattern matcher has provided a binding list for each structure and together these imply the

correspondence. Ghosts which have the same object assigned to them must coincide. The required description is easily obtained by copying these binding lists, but in place of the actual objects, further ghosts are used. A unique ghost replaces each individual object and the new correspondence-lists thereby imply identity between ghosts in the two new structures. The compound structure record which is created associates a correspondence-list with each component sub-structure. As the pattern-matcher need only consider the possibility of a compound structure arising once a relevant reduced structure has been matched, the record is kept in the component structures. As with reduced structures, the action which should be taken in order to avoid the consequences is also represented so that whether an instance of a reduced structure or of a compound structure is detected, the treatment is uniform.

Chapter 7

A Worked Example

7.1 Setting the Scene

Chapter 3 described the main components of the system and how they relate to each other. The main communication channels were also mentioned. This chapter is intended to illustrate these functions by presenting an example of their action over the course of a few moves of an actual game.

To recapitulate, the Action-Sequencer alternates moves from the Action-List with those of the opponent. The Planning-Mechanism initialises the Action-List in accordance with the Goal-Description. The Pattern-Matcher constantly awaits the instantiation of positions described in the Feature-List, entering any which do arise in the Attention-List. Before any move by the program, the Attention-List is examined by the Response-Evaluator, which may choose to modify the Action-List. Any violation of the program's expectation is considered by the Failure-Analysis component, which places any new descriptions in the Feature-List.

The example presented supposes that the program has already learned the "Line of Four" pattern, but has yet to learn the "Open Four" pattern. The program is playing black, shown as "x", and the opponent is white, shown as "o". The program always operates from a plan, and in this

example has planned to place stones successively on vertices F7, G6, H5, I4 and J3. The commentary starts after the players have each made two moves and the position is as shown in figure 7.1. The opponent is about to play.

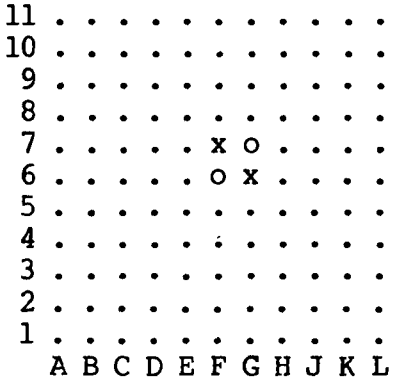


Figure 7.1 - Position after two turns

7.2 Partial Matches

The pieces already played will each have been considered by the pattern matcher in relation to the Feature-List. The assumption will be made that the only pattern in the Feature-List is that of the Line-of-Four. This pattern describes a line of five vertices, the first four of which are occupied by stones of a colour. Clearly no matches to this pattern can have been completed yet, but many matches will have been attempted, for both the black and the white stones on the board. To simplify, only the matches to the white stones will be considered: The treatment of the black stones is similar.

Partial matches become associated with the objects that are assigned to their variables but do not completely

satisfy the matcher. Thus the white stone placed on vertex F6 will result in partial matches to the Line-of Four pattern being associated with all the vertices marked "+" in figure 7.2. The partial matches radiate out from the white stone, except where a black stone causes the match to fail entirely. The partial match passing through the second stone will have two assignments made to its ghost-stones, while the others will only have one. A similar pattern may be derived for the other white stone.

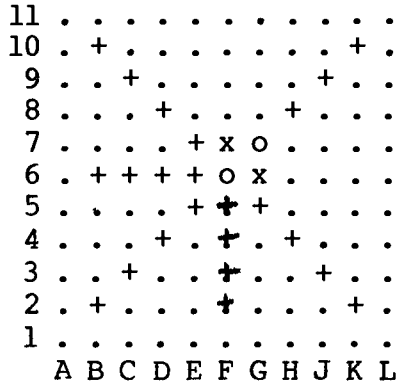


Figure 7.2 - Partial-Match Associations

7.3 A Move and Reply

The Opponent places his stone on vertex E5. This action results in a record representing the stone becoming the "occupant" value of the record representing the vertex E5. The vertex also becomes the "location" value of the stone. Once these associations have been made, the Pattern-Matcher is invoked.

The stone and the vertex involved in the move are matched to a ghost stone and vertex in the description of a Line-

of-Four. A Match-Record is created for each of the eight directions in which the line may lie. In each case, all of the vertices are matched, but only in the case of the line passing through the other white stones are any more stones matched. Each suspended partial match becomes associated with the vertices that have been bound to its ghost vertices.

The vertex E5 already has two partial matches associated with it, originating when the other white stones were placed. These matches are resumed and in each case the stone just placed becomes bound to a ghost stone, before the match is again suspended. No completed matches are produced by this application of the Pattern-Matcher and so the Attention-List remains empty.

As there is nothing in the Attention-List, the Response-Evaluator remains idle. The Action-Sequencer therefore continues with the planned move and places a black stone on vertex H5. The pattern-Matcher goes about its task in much the same way as before. The vertex H5 is one of those with which a suspended match was associated on the previous move. The Pattern-Matcher tries to resume this match and finds that the stone now occupying the vertex cannot be matched to the required ghost vertex because of the colour difference. This particular partial match is therefore terminated, and all references to the Match-Record are removed from the relevant vertices. Again no matches have been completed and the Feature-List remains

empty. The Action-Sequencer is ready for the Opponent's next move.

7.4 The Open-Four

As one might expect, the Opponent now plays on vertex D4. The position reached is shown in figure 7.3. The Pattern-Matcher is invoked as before and this time the line passing through the other white stones and terminating on the empty vertex H8, matches the description in the Feature-List fully. The completed Match-Record is placed in the Attention-List. Next the suspended matches associated with the vertex D4 are resumed, and this time one of them is also fully matched. This time, the line is that starting on vertex G7 and ending at C3. This Match-Record is also placed in the Attention-List.

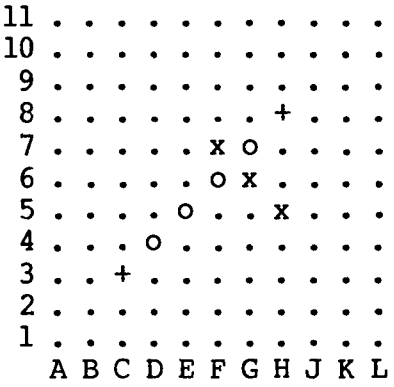


Figure 7.3 - The Open-Four

The Response-Evaluator is now applied to a no longer empty Attention-List. The structures referred to by the Match-Record both indicate that a single action is needed for the Opponent to complete a Line-of-Five. On the other

hand, two actions are required to complete the program's planned line, and the blocking moves must therefore take precedence. As two distinct blocking moves have to be made, the Response-Evaluator concludes that the game is lost.

The Failure-Analysis component is invoked to provide the Pattern-Matcher with a new Feature. To allow the system to anticipate the position, it describes the two matched patterns as they were before the Opponent's last move. This is done by taking each of the structures which were matched to (both the same in this case), and forming new reduced-structure descriptions that impose additional constraints on their variables.

The actual constraints to be imposed are determined by the binding-list of the Match-Record and the action. The action refers to two objects, a stone and a vertex. The binding-list associates a variable with each of these objects. The reverse of the action is applied to each of these particular variables, which causes the ghost-stone to cease to be the "occupant" of the ghost-vertex, and the ghost-vertex to cease to be the "location" of the ghost-stone. The ghost-stone is thereby "cast adrift" from the structure representation. Although the structure was the same in each instance, the two reduced-structures result from different binding-lists, and therefore different variables are affected. Figure 7.4 depicts: (a) the Line-of-Four structure; (b) the result of reducing the

D4-H8 match; and (c) the result of reducing the G7-C3 match. The stone on vertex D4 is the one "unplayed.

(a) s s s s .

(b) . s s s .

(c) s s s . .

Figure 7.4 - Structure Reductions

The final stage in describing the position to anticipate is to combine the two new reduced structures in a compound structure. This is a matter of describing the correspondences between their ghosts. Again the two binding-lists are used. Variables which were bound to the same objects have to correspond. The compound structure record is formed, containing a reference to the two reduced structures and a Correspondence-List. The reduced structures are placed in the Feature-List, each having the compound structure associated with them. The system is now equipped to recognise and respond to the threat of an Open-Four.

Chapter 8

Conclusions

8.1 Objectives

The main object of the project was to produce a learning program which did not rely on any purpose-built evaluation functions or move ranking scheme. The intention was that knowledge should be assimilated in terms of what is already known and that the initial knowledge provided the system should be kept to a minimum. In addition, it was intended that the system of representation employed should be as far as possible independent of the domain of the model.

The first objective, that the program's means of evaluating a situation should not be based on built-in, domain specific considerations, is in contrast with most previous examples of learning programs. Programs based on tree pruning techniques, such as Samuel's Draughts program, were provided with evaluation functions which were devised in the light of knowledge of the domain, which knowledge was thus implicit in the functions. Other systems, such as Waterman's poker program and Elcock and Murray's GoMoku program, incorporated domain specific knowledge in their descriptive language.

Assimilation of new knowledge in terms of old is not really relevant to the tree pruning systems. Essentially, these do not acquire any new concepts of their domains in

the course of their experience. Instead the significance attributed to any particular feature is modified. Programs such as ABSTRIPS, which plan to achieve their goals and then retain generalised representations of their plans, however, are acquiring knowledge and representing it in terms of previous knowledge.

The central theme to the project is that of description and detection. The nature of patterns is a limiting factor in the PLANNER gamut of languages. It would be possible to describe the sort of structures employed here as the logical conjunction of numerous simple assertions, but matching would always have to be from scratch as the PLANNER matcher makes no provision for "partial" matches, nor for any of the complexities inherent in compound structure description. The pattern of a procedure or demon is a description of some possible arrangement of elements of the domain. As such, the representation of the pattern has to relate to the representation of the domain. While the PLANNER languages satisfy this requirement, the overall representation is too cumbersome for any complex domain.

8.2 Achievements

The project has largely met with success in each of the three main objectives. The system built does not rely on evaluation functions and ranks plans on a simple criterion of length. It initially has a minimum of knowledge of its

domain, upon which it builds, and it employs a uniform, non-specific representation scheme.

The system departs from the conventional use of position evaluation and move ranking by employing a system of expectation and response, instead of look-ahead. Moves are usually made as part of a planned sequence, designed by the program to instantiate its goal. While this plan is adhered to, there is no need for the program to consider other possible moves. Positions in which the program may need to depart from its scheduled moves are recognised. When this happens, the program need only consider the moves that it has planned and those indicated by the position. The comparison here is admittedly based on the lengths of the respective sequences of actions, which is decidedly à built-in criterion. However, this is a fairly broad-based principle which is usually valid except in domains where the program's moves do not bear a symmetric relationship to the other events in the domain.

The program is initially provided with a representation of its domain, its goals and the actions it may take. The responses it acquires are all entirely in terms of these primitives. New positions are described in terms of the goals and then in terms of each other. This ability rests on the structural nature of the descriptions and on techniques for deriving generalised descriptions from instances of other generalised descriptions. The descriptive system depends on the use of strongly typed variables, the

ability to impose constraints on sets of variables and the technique of combining descriptions into more complex descriptions.

The objective of the representation scheme employed in the current program is to represent everything explicitly as items which can be manipulated by different parts of the system. Items are permitted to be associated directly with other items, so that anything that the system is likely to require in connection with an item is immediately accessible. When the pattern matcher needs to know, say, what actions may affect a given type of object, it finds all such actions associated with the definition-record for that object. The descriptive system provides general facilities for the representation of a class of domains. The primitive aspects of domain description supported are objects, single- and multi-valued relationships, and actions. Goals are represented in terms of sets of related variables.

The program does behave as intended in most other respects. The scenario in section 2.5 is a real example of its behaviour. On being blocked or beaten, the program learns to recognise and block Lines-of-Four. When this strategy proves to be inadequate, it learns about Open-Fours. From Open-Fours the program proceeds to describe Intersecting-Threes. In principle, it can arrive at the more complicated forcing patterns discussed by Elcock and Murray, such as those depicted in Appendix A. Having

described Intersecting-Threes, the Pattern-Matcher becomes so prohibitively slow that the program has not been taken beyond the point of their description and recognition. The creation and matching of all structure types has been tested by this point, however. The Line-of-Four is a simple Reduced Structure, The Open-Four is a Compound Structure composed of Reduced Structures, and the Intersecting-Threes configuration is a Compound Structure composed of Compound Structures. The VDU to which the user-interface was tailored permitted a certain amount of dynamic display, so that it was possible to monitor the progress of the pattern matcher after each move, thereby confirming its expected behaviour.

8.3 Limitations of Representation

The current scheme of representation is only adequate for the task. The most fundamental limitation is the inability to describe sets of arbitrary size. The GoMoku domain is particularly simple in that the inherent structures can have their components enumerated: A Line-of-Five can be represented by five variables representing stones and another five representing vertices, and their various relationships. The game of Go, on the other hand, requires the representation of structures which are inherently sets of unspecified size. The incorporation of such sets would impose further complications on the Pattern-Matcher and would necessitate a refinement of the notions of reduction and composition.

Another limitation is in the level of abstraction provided. Care is taken in the program that the structures already known are not relearnt. Newly derived structures are compared with existing structures and duplicates are discarded. Reductions of the same structure, with the same additional constraints are easily detected. Compound structures composed of the same substructures, with equivalent correspondences, are also similar.

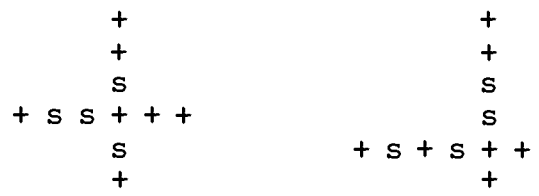


Figure 8.1 Functionally Equivalent Structures

There is also an indirect form of equivalence between structures, which depends on the level of description employed. Figure 8.1 shows instances of two differing compound structures. However, in Murray and Elcock, these would both be described by:

> "There exists a node which is a constituent of two possible 5-patterns, with two pieces played, on each of two lines through the node."

Although the two patterns are not structurally equivalent, they are functionally equivalent, in that if a further stone is played on the point of intersection of the two lines, an unbeatable pattern is created. While the present program will determine both of these patterns

under appropriate circumstances, it is unable to discover this equivalence, and would be unable to represent it even if it did. This emphasises the shortcomings of the purely structural approach employed. While it does not in principle prevent the program from learning all that it needs to know about the domain, the further generalisation has a greater intuitive appeal. Apart from this, the more compact representation would in practice reduce the number of descriptions needed and thus reduce the load on the pattern matcher.

The further generalisation represents quite a conceptual jump from the simple similarity of structures, yet it is an automatic feature of the Murray and Elcock system. This is not to say that their system realises the equivalence of the two structures and describes it: Rather, it would be unable to describe the two as different structures. The description of the one automatically incorporates the other. A more satisfactory model would arrive at the two structures independently and determine their functional equivalence in terms of their respective paths towards the goal. It would then describe what the two structures have in common, perhaps using the sort of processes proposed in Winston's hypothesised structural learning program. It may be that a different level of representation would be needed for this description, but this may imply as many descriptive techniques as levels of generality. It would be preferable to devise a uniform scheme of representation which could be employed

throughout. Representation is as central to generalisation as generalisation is to learning, and learning is the essence of intelligence.

8.4 Limitations of the Pattern-Matcher

It cannot be claimed that playing against the program is a satisfactory experience. In its naive state its performance is trivial and as it improves it becomes significantly slow. The more patterns of which it becomes aware, the longer the Pattern-Matcher takes to examine the domain for instances. The pattern matching procedures form the single most complex component of the entire system. The present scheme represents the third attempt at producing a viable process and there is probably much room for refinement. The current approach is satisfactory while only a few structures are to be matched, but the response of the system as a whole does degenerate once a useful number of structures have been recorded. The amount of work for the Pattern-Matcher compounds with the spawning of partial matches and for the system's response to be satisfactory, further reductions should be made to the workload.

There are several ways in which it might be possible to speed up the matching process, but as these have not been attempted it is not possible to comment on how significant an improvement might be obtained. One possible scheme would involve trying to match only the most reduced structures. Less reduced structures would be considered when one of these matches fails without actually

contradicting the basic underlying structure. This approach would reduce the workload to the examination of the terminal nodes on the graph of possible reductions, instead of all the nodes. The present system is able to determine the equivalance of reductions, so the reductions do form a graph rather than a tree.

8.5 Contributions

The present work, as an evaluation of a particular technique, makes three contributions to the field of Artificial Intelligence.

1) The nature of "pattern", in the context of pattern-directed-invocation, is clarified and extended to describe complex occurrences in the domain. This technique is then used to enable a program to respond to new situations.

2) A scheme of representation is presented that readily facilitates the description of such occurrences and permits initial descriptions to be modified or combined to describe further, related situations.

3) Ideas about the recognition of positions described in this manner are put forward in the form of a Pattern-Matcher that retains information about incomplete matches for later reassessment, employing a highly associative data-structure.

These points all stem from the initial objectives of the project, defining the form that the program's learning should take.

Appendix A

GoMoku

GoMoku is a simple, two-player game, in some ways similar to the familiar game of Noughts-and-Crosses. Each player tries to build an uninterrupted, straight line of five "stones" on a board comprising the vertices of a rectangular grid. The line may be diagonal or orthogonal. While the game is notionally played on an infinite board, in practice a 19 by 19 board, as used for the game "Go", is usually employed. Smaller boards can also be used, and the examples throughout this work are all on an 11 by 11 board. Each player has a set of uniform pieces, the two sets being distinguished by colour, usually black and white. The symbols "x" and "o" are used in the examples.

The concepts of blocking and forcing are familiar to any game player, and have their place in GoMoku. In figure A.1, the placing of a black stone at vertex F5 constitutes a block, as white is prevented from completing a Line-of-Five by playing on the same vertex. The blocking move is at the same time a forced move, as failure to block in this instance leads to immediate defeat. This example illustrates the most primitive of the concepts employed in GoMoku.

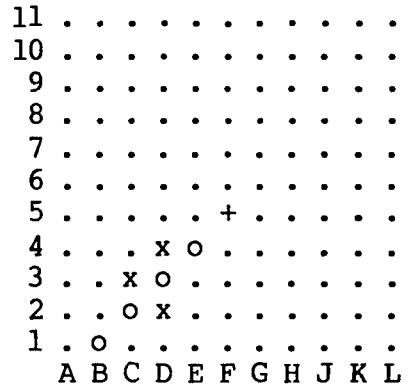


Figure A.1 - Blocking and Forcing

Figure A.2 shows an almost identical situation, except for the position of the stones relative to the edge of the board. This difference is vital, however. In this case it is not possible to prevent white from completing a line, as he may play at either A1 or F6 to do so. This is an example of an "Open-Four", comprising a line of four stones of the same colour, with an empty vertex at each end.

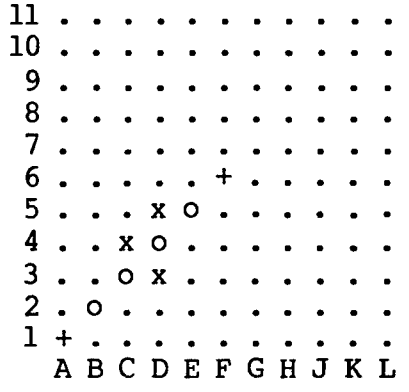


Figure A.2 - "Open-Four"

A player will not get far in the game without being able to anticipate and prevent the formation of Open-Fours. An Open-Four is an example of a position from which a Line-

of-Five is inevitable. It is in fact a particular and common case of the simultaneous occurrence of two lines of four, where the two lines partially coincide. Figure A.3 shows the more general case. Here white may play at either D8 or H4 to complete a Line-of-Five.

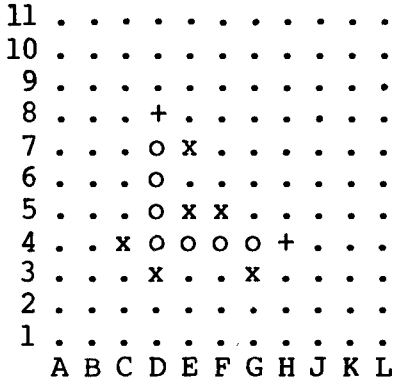


Figure A.3 - Simultaneous Fours

Having realised the significance of Open-Fours, it becomes imperative to prevent their formation by the opponent. In figures A.4 (a) and (b) it can be seen that white may play at positions D4 and E5 respectively, in order to prevent the threatened Open-Four.

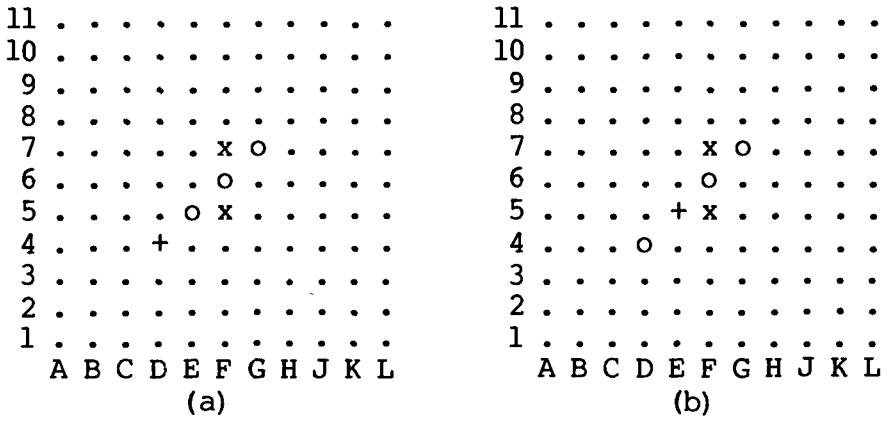


Figure A.4 - Threatened "Open-Fours"

Just as an Open-Four is a configuration which inevitably leads to a Line-of-Five, there are positions from which an Open-Four can always be created. The position shown in figure A.5 is usually referred to as "Intersecting-Threes". A black stone must be played at, say, either F7 or G5 to prevent the formation of an Open-Four. In either case white may play at the alternative vertex and an Open-Four thus completed, followed by a Line-of-Five.

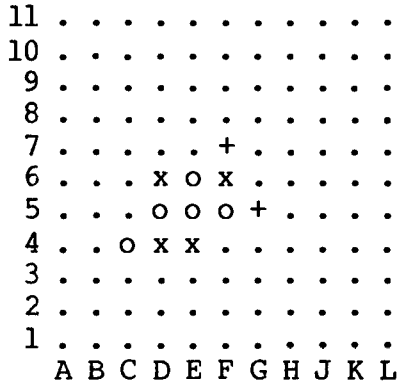


Figure A.5 - "Intersecting-Threes"

A further class of winning patterns has thus to be recognised by the player. Furthermore, there are positions which predestine Intersecting-Threes, and so on. However, the more remote the forced position, in terms of the number of pieces still to be played, the harder it is to recognise. The experienced player will seldom think in terms of the Line-of-Five as a goal, as it is too easy to spot and prevent. Instead, the less obvious forcing

positions are generally aimed at. Murray and Elcock [1968] give a large selection of forcing positions determined by their program, not all of which are obvious to the human player. Some of these are shown below.

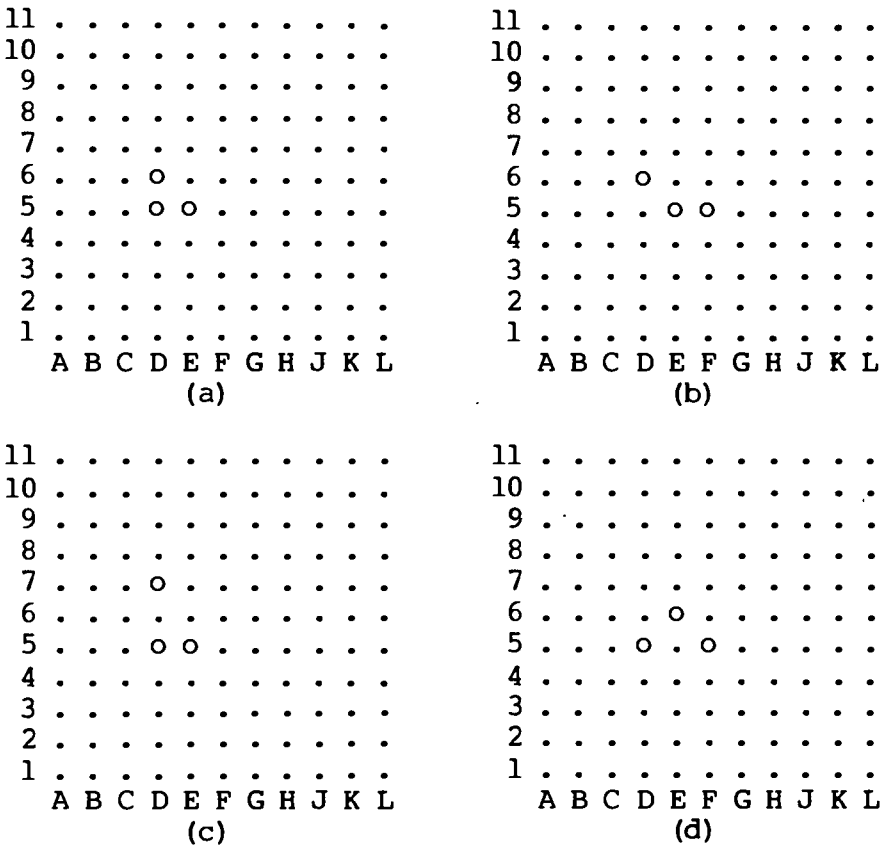


Figure A.6 - Forcing Patterns

In each of the four examples in figure A.6, it is possible for white to play a sequence of forcing moves until a Line-of-Five is completed. These positions all require most of the areas surrounding the configuration to be empty, and consequently do not often arise. The reader may like to determine some of the winning sequences.

Appendix B

POP-2

The POP-2 programming language was designed by R. M. Burstall and R. J. Popplestone [Burstall et al, 1971]. The language has three important attributes which make it eminently suited to programming in the field of Artificial Intelligence: It was designed with a view to handling non-numeric information, it is conversational, and it is extensible.

The non-numeric aspect of the language derives from the facilities provided to define and manipulate arbitrary data-structures, and from the primitive data-structures inherent to the language. Amongst the primitive data-structures are character-strings and lists, and the facilities to create and manipulate them. Arbitrary data-structures called Records can readily be defined, along with the functions to utilise them. This feature is extensively used by the program presented. Garbage collection is automatic, that is to say that the memory-space occupied by parts of the data-structures which are no longer referenced is automatically retrieved by the system.

The language strongly emphasises the notion of the Item as the basic manipulatable entity. This is taken to the extent that even a function, the basic unit of executable program, is an Item, and thus able to be the object of an

assignment, the argument to the application of another function, and even the result of a function application. In this way functions can be contained in lists or referenced in any user-defined Record. This is a facility particularly used in the present program.

The conversational aspect of the language makes it particularly easy to ascertain how a program is behaving. At any stage during its execution, a program may be interrupted and the data-structures it employs may be inspected or modified. Other programs may be applied to manipulate the data if this is desired and even the functions used within the program may be modified. The interrupted program may then be resumed in the modified environment. Used carefully, these capabilities readily facilitate program development and debugging.

The Macro facilities offered by the POP-2 system make the language extensible, in that the user may provide himself with new syntactic forms in order to make certain program constructs less verbose and more legible. POP-2 Macros are actually functions which are applied at compilation time, and can access and modify the program source. The current model, for example, prints its "ptriple" record-chains between double angle-brackets. Figure 4.1 is reproduced here as figure B.1. A similar syntax is used within the program text to create the records. Figure B.2 shows the program text that results in the record depicted in figure B.1.

```
<<DFN 1>>
        NAME        VERTEX
        PROPS       [OCCUPANT]
        GENFN       <function>GENVTX
        ACTS        [<<ACD 13>>]
```

Figure B.1 - An Object-Definition Record

```
<<DFN
      &NAME       "VERTEX",
      &PROPS      [OCCUPANT],
      &GENFN      GENVTX,
      &ACTS       ACTLIST
>>
```

Figure B.2 - Program Source

The differences between the two figures are largely due to the fact that in the creation of the record, its components are dynamically evaluated.

References

BURSTALL, R.M., COLLINS, J.S., POPPLESTONE, R.J. (1971): "Programming in POP-2." Edinburgh University Press. Revised and republished, 1977. Edinburgh University.

CHARNIAC, E. (1972): "Towards a Model of Childrens Story Comprehension." Ph.D. Thesis; Artificial Intelligence Laboratory, M.I.T., Cambridge, Mass.

CHOMSKY, N. (1965): "Aspects of the Theory of Syntax." M.I.T. Press: Cambridge, Mass.

CHURCH, A. (1956): "Introduction to Mathematical Logic 1."Princeton University Press, Princeton N.J.

ELCOCK, E.W. and MURRAY, A.M. (1967): "Experiments with a learning component in a a GoMoku playing program." Machine Intelligence 1, 87-103.

FIKES, R.E., HART, P.E. and NILSSON, N.J. (1972): "Learning and Executing Generalised Robot Plans." Artificial Intelligence 3, 251-288.

FIKES, R.E. and NILSSON, N.J. (1971): "STRIPS" A new approach to the application of theorem proving to problem solving." Artificial Intelligence 2, 189-208.

HEWITT, C. (1972): "Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot." Ph.D. Thesis; Dept. of Mathematics, M.I.T., Cambridge, Mass.

HUNT, E.B. and HOVLAND, C.I. (1963): "Programming a Model of Human Concept Formation." Computers and Thought (Eds Feigenbaum and Feldman) McGraw-Hill

McCARTHY, J., et al (1972): "LISP 1.5 Programmers Manual." M.I.T., Cambridge, Mass.

MINSKY, M. and PAPERT, S. (1969): "Perceptrons." M.I.T. Press: Cambridge, Mass.

MURRAY, A.M. and ELCOCK, E.W. (1968): "Automatic descriptions and recognition of board patterns in Go-Moku." Machine intelligence 2, 75-88.

SACERDOTI, E.D. (1974): "Planning in a hierarchy of abstraction spaces." Artificial Intelligence 5, 115-135.

SAMUEL, A.L. (1959): "Some studies in machine learning using the game of checkers." IBM J.3, 210-229.

SUSSMAN, G.J. (1973): "A computational model of skill acquisition." Ph.D. Thesis, M.I.T., Cambridge, Mass.

SUSSMAN, G.J. and McDERMOTT, D. (1972): "Why Conniving is better than planning." A.I. Memo 225A, M.I.T., Cambridge, Mass.

VAN WIJNGAARDEN, A., et al (1976): "Revised Report on the International Algorithmic Language - ALGOL 68." Springer Verlag.

WATERMAN, D.A. (1970): "Generalising Learning Techniques for Automating the Learning of Heuristics." Artificial Intelligence 1 (1970) 121-170

WINSTON, P.H. (1970): "Learning Structural Descriptions from Examples." Ph.D. Thesis, M.I.T., Cambridge, Mass.

ZORBRIST, A.L. and CARLSON, F.H. (1973): "An Advice-Taking Chess-Playing Computer." Scientific American, June 1973, 92-105.