CMU-CS-85-130

A Game-Playing Program that Learns by Analyzing Examples

Steven Minton
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

22 May 1985

Abstract

This paper describes a game-playing program that learns tactical combinations. The program, after losing a game, examines the opponent's moves in order to identify how the opponent forced the win. By analyzing why this sequence of moves won the game, a generalized description of the winning combination can be produced. The combination can then be used by the program in later games to force a win or to block an opponent's threat. This technique is applicable for a wide class of games including tic-tac-toe, go-moku and chess.

This research was sponsored in part by an AT&T Bell Laboratories Phd. Scholarship and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

Table of Contents

1. Introduction

The process of "learning by examples", or *concept acquisition*, has been intensively studied by researchers in machine learning [4]. In the concept acquisition paradigm, a program is presented with a training set of positive and negative instances of a concept represented in some description language. The program's task is to find a generalization in this language that describes all of the positive instances and none of the negative instances.

A limitation of many existing concept acquisition programs is that large training sets may be required to learn certain concepts. This limitation highlights an important difference between these programs and humans: humans have the ability to make useful generalizations on the basis of just one example. For instance, a single demonstration of a forced win in tic-tac-toe suffices to teach a novice player how to execute that maneuver in similar situations. It is the student's understanding of *how* and *why* the example works which motivates the generalization he makes.

In this paper we explore a technique for reasoning from single examples in which generalizations are deduced from an analysis of why a training instance is classified as positive. A program has been implemented that uses this technique to learn winning combinations in games such as tic-tac-toe, go-moku and chess. In each case, learning occurs after the program loses a game. The program traces out the causal chain responsible for its loss, and by analyzing the constraints inherent in the causal chain, can describe and generalize the conditions that allowed the opponent to win. The generalized conditions are incorporated into a new rule that may be used in later games to force a win or to block an opponent's threat.

The first part of this paper introduces the learning technique embodied in the game-playing program. In section 4, the program itself is described, and then an example from the game go-moku is given which illustrates the learning process. The remainder of the paper describes when and why the learning algorithm works, and relates this approach to previous research in other domains. Two appendices are given: Appendix I describes the subroutines used by the learning algorithm and Appendix II discusses implementation issues.

A high-level description of the program has been presented elsewhere [17]. One of the purposes of this report is to present the details omitted from the earlier publication. The reader is cautioned that this work is part of a continuing project and that the implemented program has not yet been extended to its full generality. Appendix II describes where the discrepancies between the design and implementation presently exist.

2. Learning And Game-playing

More than any other game, chess has attracted the attention of researchers in Artificial Intelligence, and it is often regarded as the standard domain for game-playing research. Experience has shown that the depth to which a chess program can search is a key factor in its performance [3]. Therefore, much of the recent work in this area has focused on improving methods for quickly searching game-trees. Significantly less effort has been devoted towards building knowledge-intensive chess programs. This disparity of effort has been compounded by the poor performance of knowledge-intensive programs compared to to their "dumber" cousins.

The dominance of brute-force search programs over knowledge-based programs is somewhat surprising since it appears that human chess masters rely heavily on knowledge in order to focus their search [6, 7]. Factors that contribute to the relative success of brute-force programs include the following:

- Unlike the human brain, present-day general-purpose computers are ill-suited to the pattern matching operations required for a knowledge-based approach.
- Intelligent game-playing requires a surprisingly large amount of both common-sense and domain specific knowledge. Programming this knowledge into a computer is time-consuming and difficult. Furthermore, if any details are left out the program makes obvious mistakes. In a game such as chess, a single "stupid" mistake can be fatal.(See [33] and [2] for a programs that adopt a knowledge intensive approach to chess)

This latter point is often overlooked. Even if fast pattern-matching computers existed, the right knowledge would still have to be provided in order for such machines to be useful. In fact, the knowledge-intensive approach may never be successful until the prerequisite domain-specific knowledge can be acquired automatically by machine, rather than programmed by humans.

One type of knowledge that is useful for playing two-person games is strategic knowledge about how to trap one's opponent. In game-playing terminology, a *tactical combination* is a plan for achieving a goal where each of the opponent's intervening responses is forced. The diagram below, figure 2-1, illustrates a simple chess combination called a "skewer". The black bishop has the white king in check. Therefore, after the king moves out of check, as it must, the bishop can take the queen.

A student (human or otherwise) who falls prey to this trap should be less likely to succumb to it in later games. Much can be learned by analyzing what went wrong. For instance, in retrospect it should be obvious that the position of the queen "behind" the king is essential to the trap's success. The pawns, on the other hand, are irrelevant. By analyzing why the trap works, a set of general preconditions for this combination can be found. This knowledge can be used to guard against similar occurrences in future games, and perhaps catch unwary opponents as well.

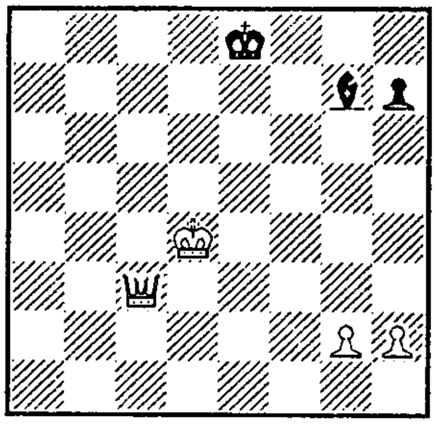

Figure 2-1: A Skewer

The learning technique that we are proposing is based on this type of retrospective analysis. It can be divided into three phases:

1. Recognize that the opponent achieved a specific goal.
2. Trace out the chain of events which was responsible for the realization of the goal.
3. Derive a general set of preconditions for achieving the goal on the basis of the constraints inherent in the causal chain.

3. Terminology

Before proceeding any further, it is necessary to introduce some terminology so that we may be precise in our analysis. All of the games we will be analyzing are two-person games. We will refer to the players as P_1 and P_2. Each of these games can be formally described as a 6-tuple $\langle S,\delta,M,s_0,W_1,W_2\rangle$ where:

- S is a finite set of game states. Each state in S represents a configuration of the board with a particular player to move.
- M is a finite set of moves such that for each $s \in S$, $m \in M$, $\delta(s,m)$ specifies the single state that can be reached from state s by move m. If m is an illegal move in state s, then $\delta(s,m)$ is undefined.
- $s_0 \in S$ is the initial state of the game.
- W_1 is the subset of S which are winning states for P_1.
- W_2 is the subset of S which are winning states for P_2. W_1 and W_2 are disjoint.

P_1 and P_2 alternate turns. Both players have complete information regarding the current state of the game.[1]

Features are predicates that are used for describing states. For example, in tic-tac-toe we might specify that **is-empty(square1)** is true in all states in which square1 is free. A *description* is a conjunction of features that describes a set of states. The description **is-empty(square1) & is-empty(<y>)**, for instance, describes all states in which there are two free squares, one of them being square1. (Angle brackets are used to denote variables, as in <x>). Specifically, a description D describes a state s iff each conjunct in D is matched by a distinct feature in s. The function σ maps descriptions into sets of states, that is, $\sigma(D)$ is the subset of S described by D.

A description can be *generalized* by dropping conjuncts and/or substituting new variables for constrained terms.[2] Accordingly, a description D_1 is said to be a generalization of D_2, denoted $D_1 > D_2$, if D_1 can be derived by generalizing D_2. A description D is *maximally-generalized* with respect to some predicate P over descriptions if P(D) is true and there exists no description $D_g > D$ such that P(D_g) is true. A description D_1 is a *maximally-specified* description if D_1 describes a single state s and there is no other description D_2 describing s such that $D_1 > D_2$.

4. Learning go-moku Combinations

In this section, we will discuss a program that learns winning combinations for a game called go-moku. Later we will discuss how this same program can be extended to a wide class of other games, including chess.

Go-moku is similar to tic-tac-toe, except that it is is played on a 19x19 board and the object of the game is to get 5 in a row, either vertically, horizontally or diagonally. Figure 4-1 illustrates a winning combination in go-moku. The last four states in a hypothetical game are shown. The first state shown, State-11, depicts the relevant section of the board prior X's move (the 11th in the game). If, in state-11, player X takes the square labeled A, then player O can block at B or C but either way X will win. If O had realized this prior to X's move, he could have pre-empted the threat by taking either square A or C.

A winning combination is a line of play that cannot be countered defensively - it is guaranteed to result in a win unless the opponent has a quicker way to win. A *forcing configuration* is a position on the board from which a winning combination can be executed. In go-moku, a forcing configuration exists if one can win in a single move, or create two or more independent ways to win. This latter method is commonly called a "fork",

[1]Note that we do not require that the rules of the game be identical for P_1 and P_2. However, as we will see, if they are identical then the game-playing program will be able to learn much more than it would otherwise

[2]There are two types of constrained terms: variables that appear more than once in a description, and constants.

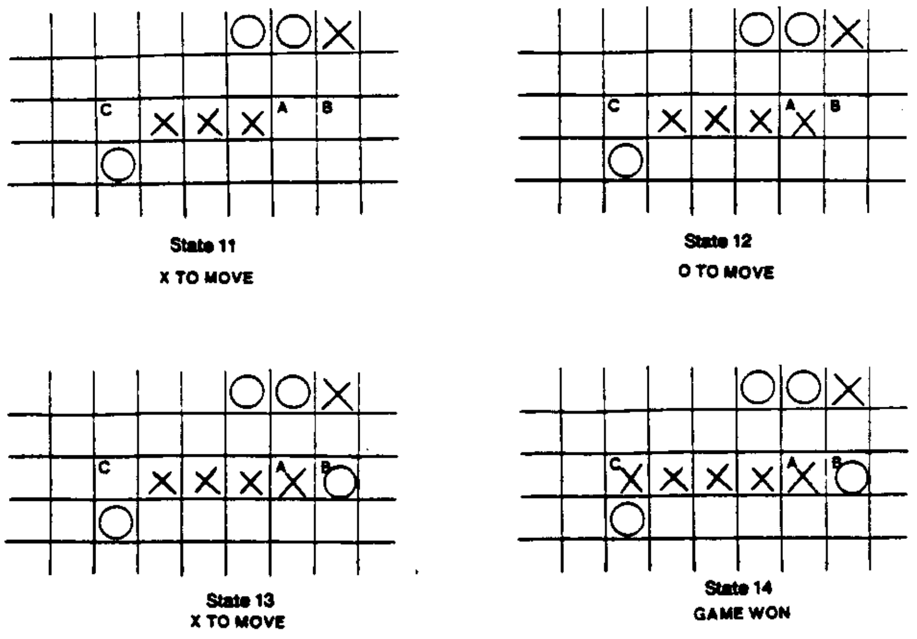

Figure 4-1: A Winning Combination in Go-moku

and the notion is central to many other games besides go-moku.

In the example above, a forcing configuration exists in state-11. This configuration, called an open-3, consists of three squares in a row, with at least two empty squares on one side and an empty square on the other. Whenever an open-3 exists for the player whose turn it is to move, then that player can win, provided the opponent has no quicker way to win. While an open-3 is a very simple forcing configuration, many people lose in precisely this way the first time they play go-moku. It appears that the open-3 concept is one of the first things that people learn in go-moku, since they rarely lose this way twice. The open-3 pattern a powerful building block from which more complex patterns can be built.

The go-moku program learns descriptions of forcing configurations and the appropriate offensive move for each configuration. The following subsections discuss the program in detail. Following this, in Section 5, we will return to the example Fig. 4-1 to demonstrate how the open-3 configuration is learned.

4.1. System Overview

The organization of the system is outlined in Figure 4-2. A *Top-Level* module interacts with the human player and queries the *Decision* module for the computer's moves. *Game-State* is a data structure listing the features which are true in the current state of the game. The knowledge necessary to play the game is partitioned into two sets of rules:

- A set of *State-Update* Rules provided by the programmer for adding and deleting features from

Game-State after each turn.

- A set of *Recognition* rules employed by the Decision module to detect forcing configurations. Initially this set is empty. The *Learning* module produces more recognition rules whenever the program loses.

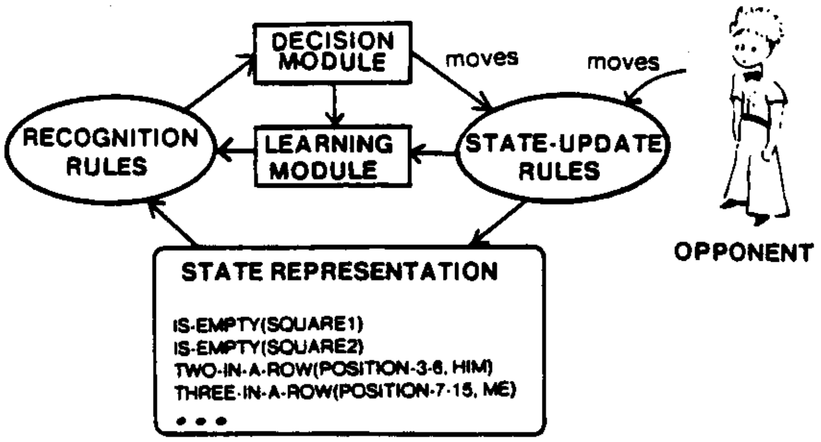

Figure 4-2: Overview of System Organization

4.2. The State-Update System

After each move in the game, the State-Update rules are invoked to update the current Game-State. The left-hand side (LHS) or "IF part" of each State-Update rule is a description. A rule is activated whenever its LHS description is matched by Game-State[3]. The right-hand side (RHS) of a rule consists of an add-list and a delete-list specifying features to be added and deleted from Game-State.

After each turn, the player's move is communicated to the State-Update system by adding a special **Input-Move** feature to Game-State. An **Input-move** feature lists the player that moved and the square that he has moved to. The State-Update system is then allowed to run until no more rules are applicable, at which point Game-State should accurately reflect the new board configuration. Once a player wins, the State-Update system adds the feature **Won(<p>)** to Game-State, where <p> is the name of the winning player. Figure 4-3 shows some State-Update rules that were written for go-moku.[4]

The State-Update system is essentially a mechanism for computing the function δ (defined in Section 3)

[3]In the present implementation, we require that the Left-hand sides of the State-Update rules describe disjoint sets of states, so that only one rule can be applicable at any time. Therefore no conflict resolution mechanism is necessary.

[4]**Four-in-a-row(<positionX>,<playerY>)** is true when there are four consecutive squares taken by <playerY> at <positionX> on the board. **Extends(<positionX>,<squareY>)** is true when <squareY> is adjacent to, and in the same line as, the sequence of squares at <positionX>. **Composes(<positionX>,<squareY>,<positionZ>)** is true when <squareY> and <positionZ> can be joined to form <positionX>. In rule Create-Four-In-A-Row, composes is used to bind the new position so that it can be referenced in the RHS

```
RULE Create-win1
IF input-move(<square>, <player>)
  is-empty(<square>)
  four-in-a-row(<4position>,<player>)
  extends(<4position>, <square>)
THEN
  ADD won(<player>)
```

```
RULE Create four-in-a-row
IF input-move(<square>,<player>)
  is-empty(<square>)
  three-in-a-row(<3position>,<player>)
  extends(<3position>, <square>)
  composes(<newposition>,<square>,<3position>)
THEN
  DELETE three-in-a-row(<3position>,<player>)
         input-move(<square>, <player>)
  ADD four-in-a-row(<newposition>, <player>)
```

Figure 4-3: Some State-Update rules for Go-moku

which maps a state and a move into a new state. In light of this, we will broaden our notation as follows. Let $<r_1,r_2,...r_n>$ be a sequence of State-Update rules. We will say that $\delta(D_1,I)=D_2$ *via* $<r_1,r_2,...r_n>$ iff when the state-update system is started in a state described by D_1 and given an input-move described by I, the rules $r_1,r_2,...r_n$ fire and the system halts in a state described by D_2.

```
RECOGNITION-RULE Recog-four
IF four-in-row(<position>, <player>)
  is-empty(<square>)
  extends(<position>, <square>)
THEN
  RECOMMENDED-MOVE
  input-move(<square>,<player>)
```

```
RECOGNITION-RULE Recog-open3
IF three-in-row(<3position>, <player>)
  is-empty(<square1>)
  extends(<3position>,<square1>)
  composes(<4position>,<square1>,<3position>)
  is-empty(<square2>)
  extends(<4position>,<square2>)
  extends(<4position>,<square3>)
  is-empty(<square3>)
THEN
  RECOMMENDED-MOVE
   input-move(<square1>,<player>)
```

Figure 4-4: Recognition Rules Learned in Go-moku

4.3. The Recognition Rules

The Decision module relies on a set of recognition rules to identify forcing configurations. Figure 4-4 shows some representative recognition rules. The right-hand side of each recognition rule specifies the appropriate move to initiate the combination.[5] During the game, if a recognition rule indicates that the opponent is threatening to win, then the computer will block the threat (unless it can win before the opponent). The blocking move is said to be *forced* since the computer will lose if it does not block the threat. Whenever a move is forced, the system must record the "reason" for the force. In the go-moku implementation, the name of the recognition rule and the features it matched constitute the "reason", and

[5] Instead of listing all the subsequent moves in the combination, a separate recognition rule exists for each step.

these are recorded for later inspection by the learning module. If more than one threat is active, and there is no way to block them all, the computer blocks the one which leads to the quickest win and records the corresponding recognition rule.

The Decision module also contains a simple procedure for deciding on the best move if no recognition rule is applicable. For example, in go-moku the program merely picks the move which extends its longest row.

4.4. The Learning Module

Whenever the computer loses a game the learning module is invoked to analyze why the loss occurred. By reasoning backwards, the features which were critical for the opponent's winning play can be isolated. From these features new recognition rules are composed.

The backward-analysis process involves examining the sequence of forced moves that the computer made at the end of the game. In each of the states preceding the computer's forced moves a forcing configuration must have been present, because the computer was subsequently forced into the loss. The critical features in these states are identified by analyzing a record of the rules activated during the game which were responsible for adding the Won feature to game-state.

Input: A game that the computer lost. The game is recorded as a sequence of Turns $<turn_1, turn_2....turn_n>$ where each $turn_i$ is associated with a $Player_i$, a maximally-specialized description of the state d_i which existed at the beginning of the turn, a $move_i$, and a $State\text{-}Update\text{-}Trace_i$. In addition, each of the computer's turns are associated with a $Decision\text{-}Trace_i$, which indicates whether or not $move_i$ was forced, and if so, lists the $forcing\text{-}conditions_i$ for that move.

Side-effect: Creates new recognition rules.

Description:

```
1. Begin
2. i←n                       -- n is the index of the last turn in the game
3. G_{i+1}← Won(<Player>)  -- describes the final game-state
4. Loop
5.     Begin
6.     G_i,I_i ← Back-Up(State-Update-Trace_i,G_{i+1})
                         -- Back-up over opponent's turn
7.     New-rule ← Build-Recog-Rule(G_i, I_i)
8.     If no equivalent rule already exists, add New-Rule to recog-rules
9.     If i=1 then Exit Loop
10.    i ← i-1
11.    G_tmp,I_i ← Back-Up(State-Update-Trace_i, G_i)
                         -- Back-up over computer's turn
12.    If move_i was not forced then Exit Loop
13.    G_i ← Combine-Conditions(G_tmp, forcing-conditions_i, d_i)
14.    If i=1 then Exit Loop
15.    i ← i-1
16.    end Loop
17. end
```

Figure 4-5: Main Loop of Learning Module

The main loop of the learning algorithm is shown in Figure 4-5. Three procedures, Back-Up, Combine-Conditions, and Build-Recognition-Rule are called; these procedures are described below and their specifications given in Appendix I.

The most significant of these procedures is Back-Up, as it accomplishes the backward reasoning necessary for identifying why the computer lost. Back-Up is given a State-Update-Trace generated during the game. This trace lists the sequence of rule activations $<r_1,r_2...r_k>$ which converted a state, s_1, into some other state, s_2, after some move m was made. A description G_2 is also provided, where G_2 is a generalized description of s_2. Back-Up's task is to find a maximally-generalized description G_1 and an Input-move description I, such that $\delta(G_1,I) = G_2$ *via* $<r_1,r_2....r_k>$. This is shown pictorially in Figure 4-6.

Back-Up functions by analyzing the effects of the rule sequence $<r_1,r_2...r_k>$ in reverse order. The inverse operator for each rule is successively applied to G_2. The operation performed by Back-Up is an instance of

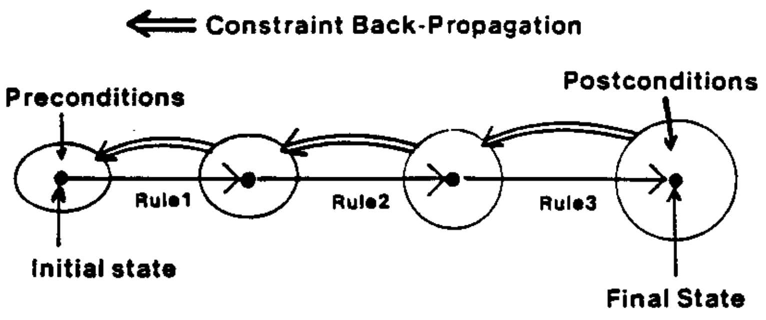

Figure 4-6: Constraint Back-Propagation (adapted from Utgoff [31])

constraint back-propagation, discussed by Utgoff [31].[6]

The procedure Combine-Conditions merges two descriptions into a single more specific description. Specifically, Combine-Conditions combines G_{tmp}, a description of the threat that the computer succumbed to, with the forcing-conditions. The resulting description includes all the features responsible for the computer's loss. In the go-moku implementation, G_{tmp} and the forcing-conditions both describe threats which exist on the board prior to the computer's move, and the description returned by Combine-Conditions describes the fork in terms of a conjunction of these two threats. See Appendix I for a more complete description of this procedure.

The third subroutine, Build-Recognition-Rule, takes a description *G*, and an input-move description *I* and creates a new recognition rule, where *G* is the Left-hand-side of the new rule and *I* the Recommended-Move. The rule is inserted into the data-base of recognition rules provided no equivalent recognition rule already exists.

5. An Example

In order to illustrate the learning algorithm, we will show how Recog-open3 (Figure 4-4) was acquired after the computer lost the position shown in Figure 4-1. Recog-open3 indicates that an "open-three" is a forcing configuration; Recall that an open-three was defined to be a three-in-a-row with two free squares on one side and one free square on the other side. In order for Recog-open3 to be learned, the computer must have previously learned the rule Recog-four (see Fig. 4-4) that states that a four-in-a-row with an adjacent open square constitutes a forcing configuration.

[6] This method of reasoning backwards is quite similar to the well-known predicate-transformer method for proving program correctness [10]. Furthermore, the operation called "regression" by Nilsson [23] is also closely related.

The game proceeds from state-11 as follows:

- **Move-11:** The opponent (player X) takes square A. The State-Update system is then invoked to reflect the change in state. A four-in-a-row feature for the opponent is added to Game-State, and the component three-in-a-row is deleted.

- **Move-12:** The computer (player O) finds two instantiations of Recog-four in state-12 (one for each way X can win). Since only one of these can be blocked, the computer arbitrarily chooses to block the threat involving square B. A record is made indicating that the move was forced by the instantiation of Recog-four. The State-Update system is then invoked and it adds to Game-State a feature indicating that square B is taken.

- **Move-13:** Finally, the opponent takes square C. The State-Update system is invoked, and it adds the feature **WON(opponent)** to Game-State. The learning module is then called upon to analyze why the game was lost.

The learning module begins by retrieving the State-Update Trace associated with the opponent's last turn. This trace indicates that a single rule, Create-Win1 (see Figure 4-3), was activated following the opponent's move and that this activation introduced the feature **Won(opponent)**. The subroutine Back-Up is called to describe the preconditions under which Create-Win1 will produce a state matching **Won(<player>)**. Back-Up returns a description G_{13} and an input-move I_{13}:

```
G13= four-in-row(<4position>,<player>)
     & is-empty(<squareA>)
     & extends(<4position>, <squareA>)

I13=  input-move(<squareA>)
```

G_{13} is a generalized description of State13. For any state described by G_{13}, if <player> moves to <squareA>, Create-win1 will be activated and a win will result. Therefore all states described by G_{13} are forcing states. Continuing, Analyze-Loss calls Build-Recognition-Rule(G_{13}, I_{13}). However, because the new rule is equivalent to Recog-four (which already exists in the database), it is discarded.

Now the State-Update-Trace associated with the transition from State-12 to State-13 is retrieved. Back-up then creates G_{tmp}, a description of the states for which some move will result in a state described by G_{13}. In this example, G_{tmp} is identical to G_{13}, since the configuration described by G_{13} was unaffected by Move-12.

Next the Decision-Trace associated with the computer's move is retrieved. This Decision-Trace specifies that Move-12 was forced by the threat recognized by Recog-four. Therefore the forcing conditions are given by the Left-hand-side of Recog-four. Combine-Conditions is called; G_{tmp} and the forcing-conditions describe the two threats present in State-12. Combine-Conditions returns G_{12}, a generalized description of the fork in State-12:

```
G12 = four-in-row(<4position>, <player>)
      & is-empty(<squareA>)
      & is-empty(<squareB>)
      & extends(<4position>, <squareA>)
      & extends(<4position>, <squareB>)
```

Now control returns to the beginning of the loop, with Move-11 under consideration. The associated State-Update-Trace indicates that transition from State-11 to State-12 was accomplished via the State-Update rule called Create-four-in-a-row. Back-Up applies the inverse of this rule to create G_{11} and I_{11}:

```
G11 = three-in-row(<3position>, <player>)
      & is-empty(<squareC>)
      & extends(<3position>, <squareC>)
      & composes(<4position>, <squareC>, <3position>)
      & is-empty(<squareA>)
      & is-empty(<squareB>)
      & extends(<4position>, <squareA>)
      & extends(<4position>, <squareB>)

I11 = input-move(<squareC>, <player>)
```

At this point Recog-open3 is built from G_{11} and I_{11}, and added to the existing set of recognition rules.

No other recognition rules will be built. The next time around the loop the exit branch at line 12 will be taken since the computer's previous move was not forced.

6. Correctness of the Learning Algorithm

A recognition rule is correct iff for every state *s* in which the rule recommends a move *i* for person *P*, a forcing configuration exists in *s*, and *i* is the appropriate move for *P* to initiate the force. The correctness of the rules that are learned by the program is an important consideration. If it is known that the rules are correct, then they can be used during play with full confidence and no search through the game-tree will be required.

To prove that the learning algorithm given in Figure 4-5 only generates correct recognition rules (assuming that the subroutines meet their specifications as given in Appendix I) it suffices to establish the following loop invariant:

Loop invariant: Every time control passes through line 7, $G_i \geq d_i$, G_i describes a set of forcing states, and I_i specifies the appropriate input-move to initiate the force.

This loop invarient can be verified by induction over the number of times the main loop is executed. The details of proof are are straightforward but tedious, and so instead we will endeavor to explain more informally why the learned rules are correct. While our discussion will be confined to go-moku, it should become obvious that there is little in the analysis that is specific to this particular game. Later on, in Section 8, we describe under what conditions the program can learn rules for other games.

In the last state preceding the opponent's win, state s_n, a forcing configuration must exist (by definition). As we have seen, Back-up takes a trace of the state-update-rules which fired after the computer's move and returns G_n and I_n. G_n describes states in which these same state-update rules will fire (and result in a win) provided that a move consistent with I_n is made. Therefore the recognition rule built from G_n and I_n is correct.

Next G_{tmp} and I_{n-1} are found by Back-up, such that for all states described by G_{tmp}, if a move consistent with I_{n-1} is made then a forcing state (described by G_n) will result for the opponent. (In this sense, G_{tmp} describes a threat that exists on the board in state s_{n-1}.) The forcing-conditions retrieved on line 7 describe the conditions under which the computer was "forced" to make move$_{n-1}$. G_{tmp} is then combined with the forcing-conditions to form G_{n-1}. All states described by G_{n-1} must be losing states, since the "reason" to make move I_{n-1} exists in G_{n-1}, and I_{n-1} results in a win for the opponent.

Now the loop is repeated, and G_{n-2} and I_{n-2} are found by Back-up. In any state described by G_{n-2}, if a move specified by I_{n-2} is made then the same state-update rules that fired during the game will again be activated, resulting in a state described by G_{n-1} (a losing state). Therefore all states described by G_{n-2} are forcing states (ie. have forcing configurations). The recognition rule that is built from G_{n-2} (and assuming no equivalent rule already exists, added to the pool of recognition rules) is therefore correct.

At this point we have gone more than once around the loop. The same arguments for the correctness of the learned recognition rules will continue to be applicable on successive iterations.

The learning algorithm terminates as soon as a non-forced move made by the computer is encountered. No state prior to this move can be shown to be a winning state without resorting to a different method of analysis. (The only other reason for the learning algorithm to terminate is that the first turn in the game is reached. This will only happen in those games where it is possible to force a win from the initial state.) In a sense, the concept of a forced move not only gives us a powerful tool for learning, but also inherently provides an effective way to limit the extent of the backwards analysis.

The strategy employed by the learning algorithm for discovering forcing states is not necessarily the most efficient. It particular, it is not always necessary to begin backing up from the end of the game in order to learn new recognition rules. For example, Recog-open3 can be built during the game as soon as the program recognizes the two independent threats in State 12. The learning algorithm presented in Figure 4-5 is not capable of performing this abbreviated analysis, but after some extra effort will arrive at the same result. The advantage of the less efficient algorithm is that it can learn forcing states in situations where independent threats can only be identified retrospectively. While we believe that it should be straightforward to enhance the program so that it can perform both types of analysis, this has not been implemented.

7. Discussion

7.1. Evaluation of the Go-moku Program

There have been a number of previous efforts to build game-playing programs that learn from single examples. Murray and Elcock [22] presented a go-moku program that learned forcing configurations by analyzing games that it had lost. A similar program by Koffman [15] learned forcing states for a variety of games. Pitrat [26] describes a program that learned chess combinations by analyzing single examples. In each of these programs, generalizations were produced either by explicit instruction, or through the use of a representation that only captured specific information. The approach outlined in this paper is similar in spirit to these earlier programs, but more powerful, since generalizations are deduced from a declarative set of domain-specific rules.

After being taught approximately fifteen examples, the program plays go-moku at a level that is better than novice, but not expert. Based on the performance of Elcock and Murray's go-moku learning program, it seems likely that the program could be brought to expert level by teaching it perhaps fifteen more examples. However, as more complex rules are learned the program slows down dramatically, despite the use of a fast pattern matcher (a version of the rete algorithm [12]). The problem is that the complexity of each new rule, in terms of number of features in its LHS, grows rapidly as the depth of the analysis is extended. In order to overcome this, the complex LHS descriptions should be converted into patterns that can be efficiently matched in the given domain. This has not yet been implemented, and whether or not it will be successful remains to be seen.

Another difficulty with the present implementation is that it plays a strong defensive game, but a comparatively weak offensive game. For defensive purposes, the recognition rules are quite powerful, since they enable the program to block the opponent before he can execute a combination. On the other hand, an effective offensive requires more than just recognizing fortuitous forcing configurations - it is necessary to create such configurations by active planning. At the present time, the program employs game-specific heuristics for deciding on the next move when no forcing configuration is found. While the performance of these heuristics is crucial to the overall performance of the game-playing program, this aspect of the program has little to do with its learning capabilities, and we have generally ignored it. Obviously this is a weak point in the program and it would be better to have a general method of planning in order to create known forcing configurations.

7.2. When is Learning Possible?

With many learning programs, it is necessary to find some "reasonable" set of features before learning can occur. What makes a set of features "reasonable" is rarely defined. An important aspect of this program is that we can specify exactly when a set of features is adequate for learning. The rule is very simple. If a state-update system can be built using only those features, then they are adequate. It is the existence of a state-update system that enables the learning algorithm to work.

What constitutes an appropriate set of state-update rules? The following three requirements answer this question. As long as the state-update rules satisfy these requirements, then rule sequences can be analyzed (by the procedure Back-Up) in order to back-propagate winning conditions.

1. Format Requirement: the State-Update rules must conform to the format specified in section 4.2. Specifically, the left-hand side of each rule must be a description, and the right-hand side must consist of an add list and/or a delete list indicating the changes to Game-State. Each move must be communicated to the system by adding a special **Input-move** feature to Game-State.

2. Legality Requirement: The Update-System must only accept legal moves. If an illegal move is made the system should halt in an error state.

3. Applicability Requirement: The State-Update rules must indicate when the game has been won by adding a **Won** feature to Game-State.

The format requirement is necessary in order for the procedure Back-Up to find the preconditions of a sequence of rules. As described in Appendix I, the method used by Back-Up is closely tied to the formalism for representing rules.

The legality requirement guarantees that only legal recommended-moves will be found. Remember that Back-up is called to describe the circumstances in which a given sequence of *legal* rules applications will fire and produce some interesting result. That is, Back-up returns a description D_1 and an input-move specification I such that $\delta(G_1,I)=G_2$ *via* $\langle r_1,r_2....r_k\rangle$ for a given legal rule sequence $\langle r_1,r_2....r_k\rangle$ and a given state G_2. If there was no legality requirement, then Back-up might return an I describing illegal moves in some of the states described by G_1.

Finally, the applicability requirement guarantees that the set of winning states can be specified in the description language. In addition, this requirement together with the legality requirement, insures that board configurations which are not isomorphic to each other[7] are represented by different Game-States.

[7] Two board configurations are isomorphic iff every sequence of input-moves results in the same outcome (win, lose, draw, or illegal state) from both configurations.

Any State-Update system which meets these requirements correctly models the game. While there will exist many State-Update Systems that meet these three requirements for any particular game, with any such system the learning algorithm can learn patterns describing winning states. However, the particular choice of features and rules will influence the generality of the learned patterns. The more general the State-Update rules are, the more general the learned patterns will be. In the previous section a recognition rule for Go-moku was learned; The generality of this rule was directly attributable to the level of generality in the State-Update rules. If instead, a large set of very specific State-Update rules was provided (eg. listing all 1020 ways to win) a much less general recognition rule would be learned from the exact same example.

While it is often desirable to learn generalized recognition rules, it is also the case that very general rules can be expensive to use. The pattern matcher may be able to operate more efficiently given a set of specific rules than it would with a single more general rule. This depends on the algorithm used by the pattern matcher, as well as properties of the domain and the representation. One possible improvement to the learning module would be to augment it with the ability to choose between alternative representations of a rule.

As we have seen, this section was described the properties that a set of State-Update rules must have in order for the game-playing program *to be able* to learn. In addition, we have seen that it is more difficult to describe what makes one state-update system better than another with regard to learning. For this we must appeal to the efficiency of the acquired rules. Given a state-update-system that is adequate for learning, we have no guarantee that it will be particularly useful, since, for example, the rules that are learned may not be particularly general. However, in practice the person who writes the state-update rules should find it most natural to write rules capturing the generalities that are immediately apparent to him (symmetries, etc). The result is that program will learn recognition rules that seem reasonable to that person. Therefore, in a practical sense the learning module can be considered a tool for acquiring complex recognition rules for forcing configurations.

8. Extending the Program

8.1. Other games

While the examples in previous sections have been taken exclusively from go-moku, there is little in our discussion that is actually specific to this game. Indeed, the program described in this paper can learn correct recognition rules for any 2-person game that meets the requirements set out in section 3, provided that the concept of a forcing state applies. We do not guarantee that the program will be able to learn rules for *all* forcing states for each game, only that the rules it does learn will be correct. In order to apply the program to a new game, it is merely necessary to construct a state-update system for that game. (In addition, a few game-specific parts of the program must also be modified. These are discussed in Section II.1).

In addition to go-moku, the program has learned recognition rules for tic-tac-toe and, to a limited extent, chess. Tic-tac-toe is obviously similar to go-moku, and the program quickly learned rules that allowed it to play tic-tac-toe without losing. The chess application was intended to demonstrate the generality of the technique rather than to provide practical means for actually playing chess. In this respect it was a success, but at the same time, certain limitations became obvious. Eventually this spurred us to consider further extensions to the program.

In chess, the idea of a winning combination corresponds to a forced check-mate. In other words, a forcing state exists if the offensive player can checkmate the other player in a finite number of moves (assuming the opponent does not win first). Therefore the recognition rules learned by the program detect opportunities for checkmating the opponent. Unfortunately, as mentioned above, the program is not guaranteed to be able learn recognition rules for all forcing states. Intuitively speaking, the program can only learn recognition rules that involve simultaneous threats, such as forks. Therefore the program can only learn a subset of all possible checkmates (although the rules that it does learn are correct). The problem is that the only type of "force" recognized by this algorithm is a threat by the opponent (as arises in a fork). However, in chess, a common way to force a move by the opponent is to limit the number of legal moves that he can make. For example, one way to do this is by putting the opponent in check. If one can create a state in which the opponent's only legal move is into a position where he can then be checkmated, then this first state is a forcing state, although this cannot be learned by the program we have described.

The solution we are adopting is to change the learning algorithm so that it more closely models the reasoning process given in Section 2. After each forced move, the program records the reason why the move was forced. In the original program, a reason could only consist of the the LHS of a recognition rule. Now, arbitrary forcing conditions can be listed as reasons. While we have only begun preliminary experimentation with this augmented version of the game-playing program, it appears to be a straightforward extension of the original program.

8.2. Learning Non-Winning Combinations

The program described in the previous sections can only learn forcing configurations that are guaranteed to result in a win. However, it is relatively simple to extend the learning algorithm so that recognition rules for other events besides forced wins can be learned, provided that such events are describable given the features provided by the State-Update system. For example, the program can learn to capture pieces in chess if it is given a state-update system (for chess) in which all captures can be expressed as conjunctions of features.

In order for the program to learn recognition rules for arbitrary events, the concept of a forced move must be extended so that the program has some criteria for determining how far to back up. We can define state *s* to be a *forcing state with respect to an event E* if the player to move is guaranteed to be able to produce an event that is at least as good as *E*. Of course, introducing recognition rules for arbitrary events may cause more harm than good if they are used indiscriminately. The computer may be able to force *E*, but in the process lose other advantages, eventually leading to a losing position. Relying on such recognition rules without performing any search would result in very myopic play. Instead, these rules could be used to focus search through the game-tree.

9. Comparing Programs that Learn from One Example

Within the past 2 years, a considerable amount of research has been presented on programs that learn from single examples [21, 36, 29, 8]. In addition, there exists some older work that is closely related [11]. Each of these programs is tailored to a particular domain: natural language understanding [8], visual recognition of objects [36], mathematical problem solving [21, 29] and planning [11]. However, it seems that these programs, along with the game-playing program described in this paper, share a common approach to learning. Below we have tried to distill the essential characteristics of this approach, which we have previously [17] referred to as as Constraint-based Generalization:[8]

Input: *A set of rules which can be used to classify an instance as either positive or negative* AND *a positive instance.*

Generalization Procedure:
Identify a sequence of rules that can be used to classify the instance as positive. Find the weakest preconditions of this sequence of rules such that a positive classification will result. Restate the preconditions in the description language.

Each of the programs alluded to earlier, as well as the game-playing program described here, can be viewed as using a form of Constraint-based Generalization despite their differing description languages and formats for expressing the rules and examples. In order to substantiate this claim, we will show how two well-known programs fit into this view.

Winston, Binford, Katz and Lowry [36] describe a system that takes a functional description of an object and a physical example and finds a physical description of the object. In their system, the rules are embedded in precedents. Figure 9-1 shows some precedents, a functional description of a cup, and a description of a particular physical cup. (The system converts natural language and visual input into semantic nets.) The

[8]Paul O'Rourke has recently suggested [private communication] that the term "explanation-based" is more appropriate than "constraint-based". This is probably true.

physical example is used to identify the relevant rules (precedents), from which a set of preconditions is established. The system uses the preconditions to build a new rule as shown in Fig. 9-2.

FUNCTIONAL DESCRIPTION OF A CUP: A cup is a stable liftable open-vessel.

PHYSICAL EXAMPLE OF A CUP: E is a red object. The objects body is small. Its bottom is flat. The object has a handle and an upward-pointing concavity.

PRECEDENTS:

- A Brick: The brick is stable because its bottom is flat. The brick is hard.
- A Suitcase: The suitcase is liftable because it is graspable and because it is light. The suitcase is useful because it is a portable container for clothes.
- A bowl: The bowl is an open-vessel because it has an upward pointing concavity. The bowl contains tomato soup.

Figure 9-1: Functional Description, Example, Precedents

```
IF [object-9 is light] & [object-9 has concavity-7]
 & [object-9 has handle-4] & [object-9 has bottom-7]
 & [concavity-7 is upward-pointing] & [bottom-7 is flat]
THEN [object-9 isa Cup]
UNLESS [[object-9 isa open-vessel] is FALSE]
    or [[object-9 is liftable] is FALSE]
    or [[object-9 is graspable] is FALSE]
    or [[object-9 is stable] is FALSE]
```

Figure 9-2: New Physical Description, in Rule Format

The LEX system learns heuristics for solving symbolic integration problems. Mitchell, Utgoff and Banerji [21] describe a technique that allows LEX to generalize a solution after being shown a single example. Each operator is represented as a rule for transforming the problem-state (Fig. 9-3). The problem-solving goal is to arrive at a state which contains no integrals; A solution is a sequence of problem-solving operators that transforms the initial problem state into a goal state. In this system, the example serves to identify a sequence of operators that can be used to solve a particular problem. The system then back-propagates the constraints through the operator sequence to arrive at a description of the problems that can be solved by applying this operator sequence. Below is a problem and a solution sequence provided to LEX:

$$\int 7(x^2)\,dx \overset{OP1}{====>} 7\int x^2 \overset{OP3}{====>} 7\ x^3/3$$

The precondition for applying op3 is that the expression be of the form $\int x^b$ where $b \neq 1$. When back-propagation is continued past op1, it is established that the expression must match $\int a(x^b)$ with $b \neq 1$ in order for this sequence of operators to be applicable. This information is used to refine the version space of plausible heuristics for OP1, the first operator in the sequence.

OP1: $\int r\ f(x)\ dx \Longrightarrow r\int f(x)\ dx$

OP2: $\int \sin(x)\ dx \Longrightarrow -\cos(x) + C$

OP3: $\int x^{r\neq 1}\ dx \Longrightarrow x^{r+1}/(r+1) + C$

Figure 9-3: Some Operators Used by LEX

10. Concluding Remarks

As stated in the last section, the game-playing program can be considered an application of constraint-based generalization, a method for collapsing useful sequences of domain rules into individual recognition rules. The input example, a played game, serves to identify a sequence of rules that is of particular interest. The constraints satisfied by the example can then be identified and combined.

The power of this approach results from analyzing rule sequences that have been shown to be useful. Many concept acquisition programs cannot take advantage of analytically derived knowledge, with the result that very large numbers of examples are required to learn complex concepts. On the other extreme, it is also possible to learn by analysis alone, however, with a large hypothesis space this may not be practical. For example, it is difficult to discover winning combinations for a game simply by examining the rules of the game. Constraint-based generalization combines these two extremes into a powerful learning technique that is applicable in many domains.

One problem with the technique is that the descriptions generated by combining constraints tend to grow quickly as the depth of analysis increases. In our game-playing program, this problem resulted in recognition rules whose left-hand sides' were very large, and expensive to compute, as pointed out in section 7.1. It may be that this difficulty can be overcome by employing more flexible methods of combining constraints. For example, rather than back-propagating all relevant constraints in order to find a recognition rule that is guaranteed to be correct, it might be more valuable to choose which constraints to back-propagate, ignoring those that appear to be of minor importance. The resulting recognition rule could be used as source of suggestions for focusing search rather than as a macro-operator that is guaranteed to work.

Throughout this paper we have identified many areas for future research. Improvements to the game-playing program need to be made. As we have just discussed, limitations of the general approach remain to be overcome. Constraint-based generalization appears to be a promising technique for generalizing from single examples, however, it is far too early to make any conclusive judgement.

11. Acknowledgements

Many of the ideas presented here were suggested by research on the Lex project by Tom Mitchell and his colleagues, especially Paul Utgoff and Richard Keller. In addition, I was also motivated by the work done by Gerald DeJong and by Pat Winston on analytic learning.

I thank Murray Campbell for his creative suggestions, Jaime Carbonell and Jack Mostow for their guidance and insight, and Hans Berliner and Pat Langley for their helpful comments.

I. Procedure Descriptions

Four procedures are described here: **Back-Up**, **Combine-Conditions**, **Build-Recognition-Rule** and **Merge**. The first three procedures are called from the main loop of the learning algorithm. **Merge** is called by both **Back-Up** and **Combine-Conditions**. **Unify**, a unification pattern-matching routine, is also used by these procedures; A description of unification pattern-matching can be found in [23]

Procedure **Back-Up**(state-update-trace, G_2)

Input:	A State-Update-Trace and a description G_2. The State-Update-Trace lists two maximally-specialized descriptions d_1 and d_2, a move m and an activation sequence $\langle r_1,r_2...r_n\rangle$ where $\delta(d_1,m)=d_2$ via $\langle r_1,r_2...r_n\rangle$. It must be the case that $G_2 > d_2$.
Output:	A pair (G_1,I_1) where G_1 is a maximally-generalized description such that $\delta(G_1,m)=G_2$ via $\langle r_1,r_2...r_n\rangle$ and $G_1 \geq d_1$.
Description:	The basic operation performed by Back-up is similar to Utgoff's constraint back-propagation [31] and Nilsson's regression [23]. A good introduction to regression can be found in Nilsson's book.

There are three major phases in Back-Up:

Phase 1: Incrementally specialize G_2 until no no state-update rule is applicable in any state described by G_2. The process of specializing G_2 is accomplished by gradually transforming it into d_2 first by instantiating the variables in G_2, then by adding conjuncts. Note that no state-update rule can be applicable in d_2 since it is the final state given by the State-Update Trace.

Phase 2: Set D_{post} equal to G_2. Then for each rule R in $\langle r_1,r_2...r_n\rangle$, starting with rule r_n, do the following:

1. Let LHS_R be the left-hand-hand side of R. For each literal l_{add} in R's add-list, attempt to find a literal l_{post} in D_{post} that can be unified with l_{add}. If such an l_{post} can be found, take the bindings which will transform l_{add} into l_{post}, and perform the substitutions on LHS_R and D_{post}. (If there exists more than one l_{post} that will unify with l_{add}, choose the l_{post} that will result in a maximally general LHS_R consistent with the state prior to the activation of R.) Remove l_{post} from D_{post}.

2. Find d_{pre}, a maximally-specialized description of the features in Game-State immediately preceding the activation of R. This is easily accomplished by examining the bindings that rule R was activated with.

3. $D_{post} \leftarrow$ Merge(LHS_r, G, d_{pre})

Phase 3: Let $G_1 \leftarrow D_{post}$. Let I be the input-move feature in G_1. Delete I from G_1. Return (G_1, I).

Procedure Combine-Conditions(D_1,D_2,d_s)

Input: Two descriptions D_1 and D_2, and a maximally specialized description d_s, where $D_1 \geq d_s$ and $D_2 \geq d_s$ It must be the case that for any state s described by D_1 or D_2 there exists a move that results in a forcing configuration for the opponent.

Output: A description D_{out} such that

- $D_{out} \geq d_s$
- $D_1 \geq D_{out}$
- $D_2 \geq D_{out}$

and there is no legal move in any state in $\sigma(D_{out})$ that does not result in a forcing configuration.

Description:

There are two aspects to Combine-Conditions. First, D_1 and D_2 must be merged into a single description consistent with d_s. This is accomplished by setting D_{out} equal to the result of Merge(D_1, D_2, d_s). (The description of Merge is given on the following page). Secondly, Combine-Conditions must insure that D_{out} only describes losing states - states in which every move results a forcing configuration for the opponent. This is done by generating two input-move descriptions I_1 and I_2, such that I_1 describes the moves forced by D_1 (the moves that do not result in a forcing configuration for the opponent) and I_2 describes the moves forced by D_2. Now Combine-Conditions can insure that there is no move consistent with I_1 and I_2 in any state described by D_{out} by examining whether it is possible to unify I_1 and I_2 without violating D_{out}. (A unifier is inconsistent with D_{out} if two variables are united that cannot possibly be equal if D_{out} is to be a legal description). If this is impossible, D_{out} is returned as is. If there does exist a unifier consistent with D_{out}, then D_{out} is conjoined with a predicate, not-equals(A,B), where A and B are bound to a term from I_1 and I_2 (respectively) so that I_1 and I_2 cannot now be unified in a manner consistent with D_{out}. This will insure that D_{out} only describes states where there is no way to simultaneously avoid the pitfalls described by D_1 and D_2.

Note: in the present implementation of the learning module, I_1 and I_2 are in fact accepted as arguments to Combine-Conditions. Since D_1 is bound to G_{temp}, I_1 is set equal to the Input-move description returned by Back-up on line 11 of learning algorithm). And since D_2 is bound to the forcing-conditions, I_2 describes the move that was forced - the recommended move of the recognition rule that detected the force.

Procedure Merge(D_1, D_2,d_s)

Input: Two descriptions D_1 and D_2, and a maximally-specialized description d_s, with $D_1 \geq d_s$ and $D_2 \geq d_s$

Output: A maximally-generalized description D_{out} such that $D_{out} \geq d_s$ and $D_1 \geq D_{out}$ and $D_2 \geq D_{out}$

Description:

Merge combines two descriptions, D_1 and D_2, to form a new less-general description D_{out}. The state given by d_s must be described by d_{out}.

Merge operates by finding the possible ways D_1 and D_2 can match the ground instance d_s. Initially D_{out} is the null expression. If a literal in the ground instance is matched by a single literal in D_1 or D_2, then that literal from D_1 or D_2 is conjoined with D_{out}. If a literal from D_1 and a literal from D_2 correspond (ie. can match the same literal in in the ground instance) then they are unified. If the resulting substitution is consistent with previous substitutions then the resulting literal can be conjoined with D_{out}. There may be different combinations of corresponding literals, since there may be many possible ways to match D_1 and D_2 with the ground instance. Therefore there may be more than one legal D_{out}. Merge must return a D_{out} that is at least as general as any other D_{out}.

II. Implementation Issues

II.1. Where is the Knowledge?

In section 8 we claimed that the game-playing program can learn winning combinations for a wide variety of games. We also noted that the current program is not capable of learning some types of winning combinations. It is our intent in this subsection to specify which parts of the program are game-specific, so that the reader will have a clear idea of what the program "knows" about game-playing in general, versus what it "knows" about each particular game it plays.

As indicated earlier, the state-update rules are the program's primary source of game-specific knowledge. The learning module is designed so that no other game-specific information is necessary. (Note, however, that the current implementation does not quite live up to this promise. See Implementation Notes below.)

The decision module, which picks the computer's next move, invokes a game-specific procedure to find an acceptable move if no forcing configuration is found on the board. Of course, this has little to do with the learning capabilities of the program. Nevertheless, we hope to be able to improve the program by giving it a more general method of picking moves when no recognition rules are activated.

Additionally, whenever it is forced to make a move, the decision module must record the "reason" for the force. In the initial implementation, the forcing conditions were taken from the left-hand-sides of recognition rules. This was sufficient to enable the program to learn winning combinations for go-moku, as well as any other game where winning depends on creating a fork. In order to expand the number of forcing states the program can recognize in chess, the implementation was changed to allow these forcing-conditions to be arbitrary conditions. As such they presently have to be specifically coded for each set of state-update rules used. Again, this is another aspect of the program we are trying to improve by making it game-independent.

In summary, the game-specific knowledge necessary for learning is found in the state-update rules and in the procedures that record forcing conditions. The rest of the program is game-independent, in the sense that it is applicable to any game meeting the requirements set forth in section 3.

II.2. Implementation Notes

The game-playing-system has been implemented in Franz Lisp on a Vax 780. The program was originally developed to play go-moku, and then generalized. Because of this development history, some aspects of the implementation still fall short of the design reported in this paper.

The following is a list of the more important differences that have not yet been rectified:

- As discussed in section 8, the present system is in the process of being modified for chess. As it is, the system can only detect forcing states that result from multiple simultaneous threats.

- The Combine-conditions procedure is not implemented in its full generality. It presently requires two extra arguments describing the moves that are forced by D_1 and D_2. This problem as discussed in Appendix I.

- The decision module recognizes that a move is forced by checking to see whether any recognition rules detect forcing states for the opponent. Actually, the system should look ahead one move, and if recognition rule fires then back-propagate the left-hand side of the rule in order to record the forcing-conditions. The present strategy will not work for some state-update systems.

- In order to make the system run faster, some of the state-update rules are simulated rather than matched against Game-State.

References

1. Banerji, R.. *Artificial Intelligence.* North Holland, 1980.

2. Berliner, H. *Chess as Problem Solving: The development of a Tactics Analyzer.* Ph.D. Th., CMU Dept. of Computer Science, 1974.

3. Berliner, H. An Examination of Brute Force Intelligence. Proceedings of the 7th International Joint Conference on Artificial Intelligence, 1981.

4. Carbonell, J., Michalski, R. and Mitchell, T. An Overview of Machine Learning. In *Machine Learning,* Carbonell, J., Michalski, R. and Mitchell, T., Ed., Tioga Publishing Co., 1983.

5. Carbonell, Jaime G. Derivational Analogy and its Role in Problem Solving. Proceedings of the National Conference on Artificial Intelligence, 1983.

6. Chase, W. and Simon H. "Perception in Chess". *Cognitive Psychology 4,* 1 (1973).

7. De Groot, Adriaan D.. *Thought and Choice in Chess.* Mouton & Co., 1965.

8. DeJong,G. An Approach to Learning by Observation. Proceedings, International Machine Learning Workshop, 1983.

9. DeJong,G. Acquiring Schemata through Understanding and Generalizing Plans. Proceedings of the 8th International Joint Conference on Artificial Intelligence, 1983.

10. Dijkstra, E.. *A Discipline of Programming.* Prentice Hall, 1976.

11. Fikes, R., Hart, P. and Nilsson, N. "Learning and Executing Generalized Robot Plans". *Artificial Intelligence 3,* 4 (1972).

12. Forgy, C. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem". *Artificial Intelligence 19,* 1 (1982).

13. Keller, R. A Survey of Research in Strategy Acquisition. DCS-TR-115, Computer Science Dept., Rutgers University, 1982.

14. Keller, R. Learning by Re-expressing Concepts for Efficient Recognition. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1983.

15. Koffman, E. "Learning Through Pattern Recognition Applied to a Class of Games". *IEEE Trans. Sys. Sciences and Cybernetics SSC-4,* 1 (1968).

16. Langley, P., Bradshaw, G. and Simon, H. Rediscovering Chemistry with the Bacon System. In *Machine Learning,* Carbonell, J., Michalski, R. and Mitchell, T., Ed., Tioga Publishing Co., 1983.

17. Minton, S. Constraint-Based Generalization. Proceedings of the National Conference on Artificial Intelligence, 1984.

18. Minton, S. Selectively Generalizing Plans for Problem Solving. International Joint Conference on Artificial Intelligence, 1985.

19. Mitchell, T. The Need for Biases in Learning Generalizations. CBM-TR-117, Computer Science Dept., Rutgers University, 1980.

20. Mitchell, T. "Generalization as Search". *Artificial Intelligence 18*, 2 (1982).

21. Mitchell, T., Utgoff, P. and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In *Machine Learning*, Carbonell, J., Michalski, R. and Mitchell, T., Ed., Tioga Publishing Co., 1983.

22. Murray, A. and Elcock, E. Automatic Description and Recognition of Board Patterns in Go-Moku. In *Machine Intelligence 2*, Dale, E. and Michie, D., Ed., Elsevier, 1968.

23. Nilsson, N.J.. *Principles of Artificial Intelligence*. Tioga, 1980.

24. O'Rorke, Paul. Generalization for Explanation-based Schema Acquisition. Proceedings of the National Conference on Artificial Intelligence, 1984.

25. Pitrat, J. A Program for Learning to Play Chess. In *Pattern Recognition and Artificial Intelligence*, Chen, Ed., Academic Press, 1976.

26. Pitrat, J. Realization of a Program Learning to Find Combinations at Chess. In *Computer Oriented Learning Processes*, Simon, J., Ed., Noordhoff, 1976.

27. Porter, B. and Kibler, D. Learning Operator Transformations. Proceedings of the National Conference on Artificial Intelligence, 1984.

28. Salzberg, S. Generating Hypotheses to Explain Prediction Failures. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1983.

29. Silver, B. Learning Equation Solving Methods from Worked Examples. Proceedings of the International Machine Learning Workshop, 1983.

30. Utgoff, P. Acquisition of Appropriate Bias for Inductive Concept Learning. LCSR-TM-2, Computer Science Dept., Rutgers University, 1982.

31. Utgoff, P. Adjusting Bias in Concept Learning. Proceedings International Machine Learning Workshop, 1983.

32. Waterman, D. "Generalization Learning Techniques for Automating the Learning of Heuristics". *Artificial Intelligence 1* (1970).

33. Wilkins, D. "Using Patterns and Plans in Chess". *Artificial Intelligence 14* (1980).

34. Winston, P. Learning Structural Descriptions from Examples. In *The Psychology of Computer Vision*, Winston, P., Ed., McGraw Hill, 1975.

35. Winston, P. "Learning New Principles from Precedents and Examples". *Artificial Intelligence 19*, 3 (1982).

36. Winston, P., Binford, T., Katz, B. and Lowry, M. Learning Physical Descriptions from Functional Definitions, Examples and Precedents. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1983.