Constraint-based Generalization

Learning Game-Playing Plans from Single Examples

Steven Minton

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Constraint-based Generalization is a technique for deducing generalizations from a single example. We show how this technique can be used for learning tactical combinations in games and discuss an implementation which learns forced wins in tic-tac-toe, go-moku, and chess.[1]

1 Introduction

During the last decade "learning by examples", or *concept acquisition*, has been intensively studied by researchers in machine learning [1]. In this paradigm, the learner induces a description of a concept after being shown positive (and often negative) instances of the concept.

A limitation of many existing concept acquisition systems is that numerous examples may be required to teach a concept. People, on the other hand, can make accurate generalizations on the basis of just one example. For instance, a novice chess player who is shown just one example of a "skewer" (Figure 1) will later be able to recognize various forms of the skewer principle in different situations. Understanding *how* and *why* a particular example works allows him to arrive at a generalized concept description. However, most existing concept acquisition systems are completely data-driven; They operate in relative isolation without the benefit of any domain knowledge.

Constraint-based Generalization is 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 forced wins in 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, find a description of the general conditions under which this same sequence of events will occur. This description is then incorporated into a new rule which can be used in later games to force a win or to block an opponent's threat. Following a discussion of this implementation, a domain-independent formulation of Constraint-based Generalization will be introduced.

2 Learning Plans for Game-playing

In game-playing terminology, a tactical combination is a plan for a achieving a goal where each of the opponent's moves is forced. Figure 1 illustrates a simple chess combination, called a "skewer". The black bishop has the white king in check. After the king moves out of check, as it must, the bishop can take the queen.

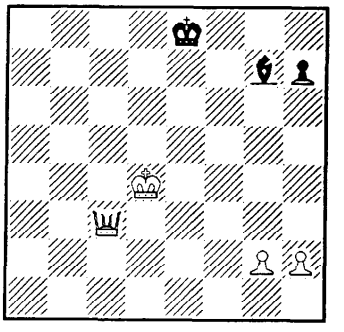

Figure 1: A Skewer

A student who has had this particular instance demonstrated to him can find an appropriate generalization by analyzing why the instance worked. Such an analysis can establish that while the pawns are irrelevant in this situation, the queen must be "behind" the king for the plan to succeed. Ultimately, a generalized set of preconditions for applying this combination can be found. In future games this knowledge can be used to the student's advantage. Presumably, he will be less likely to fall into such a trap, and may be able to apply it against his opponent.

The learning algorithm we propose models this reasoning process. There are three stages:

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

3 The Game Playing System

This section describes a game-playing system that has learned winning combinations in tic-tac-toe, go-moku and chess. A *forcing state* is a configuration for which there exists a winning combination - an an offensive line of play that is guaranteed to win. Figure 2 illustrates a winning combination in go-moku, a game played on a 19x19 board. The rules are similar to tic-tac-toe except that the object of the game is to get 5 in a row, either vertically, horizontally or diagonally. If, in state A, player X takes the square labeled 2, then player O can block at 1 or 6, 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 2 or 6. The game-playing system learns descriptions of forcing states and the the appropriate offensive move to make in each such state by analyzing games that it has lost.

[1]This research was supported in part by the Defense Advanced Projects Agency (DOD) Arpa Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551, and in part by a Bell Laboratories Ph.D Scholarship.

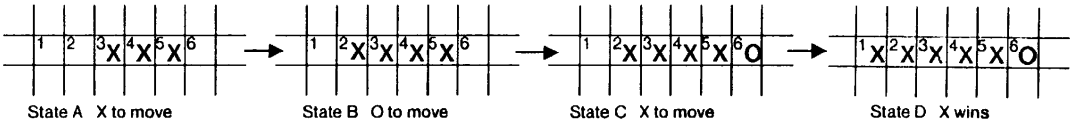

Figure 2: A Winning Combination in Go-moku

The game-playing system is organized into several modules, including a *Top-Level* module that interacts with the human player and a *Decision* module that chooses the computer's moves. A set of features which describes the current board configuration is kept in a data structure called *Game-State*. Most of the system's game-specific knowledge 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 *Recognition* rules employed by the Decision module to detect forcing states. Initially this set is empty. The *Learning* module produces more recognition rules whenever the program loses.

Features in Game-State are represented by predicates. For example, in tic-tac-toe **is-empty(square1)** might be used to indicate that square1 is free. The State-Update-Rules form a production system that updates the Game-State as the game progresses. The IF-part or left-hand side of each rule is a *description*: a conjunction of features possibly containing variables. (Angle brackets are used to denote variables, eg. <x>). The right-hand side of a rule consists of an add-list and a delete-list specifying features to be added and deleted from Game-State when the rule is activated. Figure 3 shows some State-Update rules that were used for go-moku.[2]

In the present implementation, only one State-Update rule can be applicable at any time, so no conflict resolution mechanism is necessary. Whenever a rule fires, it leaves behind a State-Update-Trace, indicating the features it matched in Game-State.

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

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

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

An **INPUT-MOVE** feature is added to Game-State after each player moves (see Figure 3). The State-Update system is then allowed to run until no rules can fire, at which point Game-State should accurately reflect the new board configuration. When a player <p> wins, the State-Update system adds a feature **WON(<p>)** to Game-State.

The Decision module relies on the set of Recognition rules to identify forcing states. (See Figure 4 for some representative recognition rules). The right-hand side of each recognition rule indicates the appropriate move to initiate the combination[3]. When a recognition rule indicates that the opponent is threatening to win, then the computer blocks the threat (unless it can win before the opponent). The blocking move is classified as *forced*, and the name of the recognition rule and the features it matched are recorded in a data-structure called DECISION-TRACE. 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.

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

```
RECOG-RULE Recog-Open-Three
IF three-in-row(<3position>,<player>)
  is-empty(<squareC>)
  extends(<3position>,<squareC>)
  composes(<4position>,<squareC>,
        <3position>)
  is-empty(<squareB>)
  extends(<4position>,<squareB>)
  is-empty(<squareA>)
  extends(<4position>,<squareA>)
RECOMMENDED-MOVE
  input-move(<squareC>,<player>)
```

Figure 4: Recognition Rules Learned in Go-moku

Initially, the system has no recognition rules. Whenever it loses a game the learning module analyzes why the loss occurred; A new recognition rule can be introduced if the state occurring after the computer's last non-forced move - the critical state - can be shown to be a forcing state. If this analysis is successful, a new rule will be built so that this state, and others like it, can be recognized as forcing states in future games.

The learning module must identify the features in the critical state that allowed the opponent to win. It accomplishes this by examining the sequence of state-update rules and recognition rules which fired between each of the opponents moves and which eventually added the the feature **WON(opponent** to Game-State. Assuming that the threats recognized by the computer were independent[4] then the critical state must have been a forcing state. Indeed, any state in which this same sequence of rules will fire, resulting in a win, must be a forcing state. To build the new recognition rule, the learning module finds a generalized description of the critical state such that the constraints defined by the sequence of rules are satisfied.

A procedure named Back-Up accomplishes this by reasoning backward through the rule sequence. In order to traverse rules in the backward direction, Back-Up takes a description of a set of post-features, and finds the most general set of pre-features such that if the pre-features are true before the rules fire, the post-features will be true afterwards. This operation is an instance of "constraint back-propagation" [12]: Dijkstra formalizes this method of reasoning backwards in his discussion of weakest preconditions for proving program correctness [3].

In order to illustrate how Back-Up operates, we will consider how Recog-Open-Three (Figure 4) is acquired after the computer loses the position shown in Figure 2. Recog-Open-Three recognizes an "open three", which consists of a "three-in-a-row" with two free squares on one side and one free square on the other. In order for Recog-Open-Three to be learned, the computer must have previously learned Recog-Four which states that a four-in-a-row with an adjacent open square constitutes a forcing state. After the opponent (player X) takes square 2, the computer (player O) finds two instantiations of Recog-Four in state B (one for each way for X to win). Since only one of these

[2] 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>.

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

[4] Threats are independent if there is no way to block them simultaneously.

can be blocked, the computer arbitrarily moves to square 6, recording that the move was forced by the particular instantiation of Recog-Four. Then the opponent proceeds to win by taking the fifth adjacent square on the other side.

The learning module is then invoked to build a new recognition rule. By examining the State-Update-Trace, the program finds that an instantiation of Rule Create-Win1 (Figure 3) was responsible for adding **Won(opponent)** to Game-State after the opponent made his last move. Back-propagation identifies the pre-features necessary for this rule to produce a post-feature matching Won(<player>):

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

Deleting the **input-move** feature gives a generalized description of state C, the forcing state existing prior to the opponent's last move. Since the computer's move previous to this (from State B to State C) was in response to the independent threat identified by Recog-Four, the system continues backing-up. The left-hand side of Recog-Four is combined with the preconditions above to arrive at a generalized description of state B. This is a state with two independent threats:

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

Continuing, Back-Up finds that the opponent's move (into square X) caused rule Create-four-in-a-row to fire, producing the **four-in-a-row** feature in this description. Back-propagating across this rule allows us to restate the pre-conditions as show in Recog-Open-Three (Figure 4). The Recommended-Move is the input-move precondition corresponding to X's move from state A to State B. The left-hand side of Recog-Open-Three describes the relevant features in state A which allowed X to force a win.

4 Discussion

Murray and Elcock [9] present a go-moku program that learned patterns for forcing states by analyzing games that it had lost. A similar program by Koffman [6] learned forcing states for a class of games. Pitrat [10] 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 15 examples, the system plays go-moku at a level that is better than novice, but not expert. Based the performance of Elcock and Murray's go-moku learning program, it seems likely that the system could be brought to expert level by teaching it perhaps 15 more examples. However, as more complex rules are learned the system slows down dramatically, despite the use of a fast pattern matcher (a version of the rete algorithm [5]). The problem is that the complexity of each new rule, in terms of the number of features in its left-hand side, grows rapidly as the depth of the analysis is extended. In order to overcome this, the complex left-hand side descriptions should be converted into domain-specific patterns that can be efficiently matched. This has not been implemented.

In addition to learning combinations for winning tic-tac-toe and go-moku, the system (with modifications to the decision module) has learned patterns for forced matings in chess. While we believe that this implementation demonstrates the generality of the learning technique, it does not provide a practical means for actually playing chess. The patterns learned are inefficient and represent only a fraction of the knowledge required to play chess [13].

5 Requirements for Learning

With many learning systems, it is necessary to find some "good" set of features before learning can occur. An important aspect of this system is that we can specify exactly what is necessary for the system to be able to learn. In particular, if a State-Update system can be written that satisfies the following requirements, it can be shown that correct recognition rules will be acquired for tic-tac-toe, go-moku, or any other game in which the concept of a forcing state can be appropriately formalized [7].

1. FORMAT REQUIREMENT: the State-Update rules must conform to the format specified in section 3.
2. APPLICABILITY REQUIREMENT: The State-Update rules must indicate when the game has been lost by adding a **Won** feature to Game-State.
3. LEGALITY REQUIREMENT: The Update-System must only accept legal moves.

Informally speaking, the FORMAT requirement guarantees that back-propagation can be used to find the preconditions of a sequence of rules; The APPLICABILITY requirement guarantees that the system can identify when to begin backing up; The LEGALITY requirement guarantees that only legal Recommended-moves will be found.

While there will exist many Update-Systems that meet these requirements for any particular game, with any such system the learning algorithm can learn patterns describing forcing 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.

It is possible to extend the system so that preconditions for other events besides forced wins can be learned, provided that such events are describable given the features used by the State-Update system. For example, learning to capture pieces in checkers is only possible if one is able to describe a capture in the description language. In order to learn recognition rules for arbitrary events, the definition of a forcing state must be modified. We define a state S to be a forcing state for player P *with respect to event E* iff P can make a move in S that is guaranteed to produce an event at least as good as E. Unfortunately, recognition rules for arbitrary events may cause more harm than good if they are used indiscriminately. A player may be able to force E, but then find himself in a situation where he is worse off in other respects.

6 Comparing Constraint-based Generalization systems

Within the past 2 years, a considerable amount of research has been presented on systems that learn from single examples [8, 14, 11, 2]. In addition, there exists an older body of related work [4, 6, 9]. Each of these systems is tailored to a particular domain: game playing [6, 9], natural language understanding [2], visual recognition of objects [14], mathematical problem solving [8, 11] and planning [4]. In order to characterize what these systems have in common, we present the following domain-independent description of Constraint-based Generalization:

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. Employ backward reasoning to 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 systems alluded to earlier can be viewed as using a form of Constraint-based Generalization although they differ in their description languages, formats for expressing the rules and examples, and criteria for how far to back-propagate the preconditions. In order to substantiate this claim, we will show how two well-known systems fit into this view.

Winston, Binford, Katz and Lowry [14] 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 5 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 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. 6.

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 5: Functional Description, Example, Precedents

IF [object9 is light] & [object9 has concavity7]
& [object9 has handle4] & [object9 has bottom7]
& [concavity7 is upwardpointing] & [bottom7 is flat]
THEN [object9 isa Cup]
UNLESS [[object9 isa openvessel] is FALSE]
or [[object9 is liftable] is FALSE]
or [[object9 is graspable] is FALSE]
or [[object9 is stable] is FALSE]

Figure 6: New Physical Description, in Rule Format

The LEX system learns heuristics for solving symbolic integration problems. Mitchell, Utgoff and Banerji [8] describe a technique that allows LEX to generalize a solution after being shown a single example. A solution is a sequence of problem-solving operators that is applied to the initial problem state. (Fig. 1). 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 generalized description of the problems that can be solved by applying this operator sequence. Below is a problem and a solution sequence provided to LEX:

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

Back-propagation establishes that the initial expression must match $\int a(x^{b\neq 1})$ in order for this sequence of operators to be applicable.

OP1: $\int r\,f(x)\,dx \Rightarrow r\int f(x)\,dx$
OP2: $\int \sin(x)\,dx \Rightarrow -\cos(x) + C$
OP3: $\int x^{r\neq 1}\,dx \Rightarrow x^{r+1}/(r+1) + C$

Table 1: Some Operators Used by LEX

7 Conclusions

Constraint-based generalization is a form of meta-reasoning in which generalizations are deduced from a single example. The example serves to isolate a sequence of rules that identify positive instances. By finding the weakest preconditions of these rules that produce a positive classification, a generalization can be made. The power of this technique stems from the focus that the example provides for the analysis process.

8 Acknowledgements

Tom Mitchell and his colleagues' research on LEX suggested many of the ideas presented here. I thank Murray Campbell, Jaime Carbonell, Hans Berliner and Pat Langley for their suggestions.

References

1. 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.
2. DeJong,G. An Approach to Learning by Observation. Proceedings, International Machine Learning Workshop, , 1983.
3. Dijkstra, E.. *A Discipline of Programming.* Prentice Hall, 1976.
4. Fikes, R., Hart, P. and Nilsson, N. "Learning and Executing Generalized Robot Plans." *Artificial Intelligence 3*, 4 (1972).
5. Forgy, C. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem." *Artificial Intelligence 19*, 1 (1982).
6. Koffman, E. "Learning Through Pattern Recognition Applied to a Class of Games." *IEEE Trans. Sys. Sciences and Cybernetics SSC-4*, 1 (1968).
7. Minton, S. A Game-Playing Program that Learns by Analyzing Examples. Tech Report, Computer Science Dept., Carnegie Mellon University, forthcoming
8. 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.
9. 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.
10. Pitrat, J. Realization of a Program Learning to Find Combinations at Chess. In *Computer Oriented Learning Processes*, Simon, J., Ed.,Noordhoff, 1976.
11. Silver, B. Learning Equation Solving Methods from Worked Examples. Proceedings of the International Machine Learning Workshop, , 1983.
12. Utgoff, P. Adjusting Bias in Concept Learning. Proceedings International Machine Learning Workshop, , 1983.
13. Wilkins, D. "Using Patterns and Plans in Chess." *Artificial Intelligence 14* (1980).
14. 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.