

A STRATEGIC METAGAME PLAYER FOR GENERAL CHESS-LIKE GAMES

BARNEY PELL

*Caelum Research Corporation
NASA Ames Research Center
AI Research Branch, Mail Stop: 269-2
Moffett Field, CA 94035-1000
pell@ptolemy.arc.nasa.gov*

This paper introduces METAGAMER, the first program designed within the paradigm of Meta-Game Playing (Metagame). This program plays games in the class of symmetric chess-like games, which includes chess, Chinese-chess, checkers, draughts, and Shogi. METAGAMER takes as input the *rules* of a specific game and analyses those rules to construct an efficient representation and an evaluation function for that game, which are used by a generic search engine. The strategic analysis performed by METAGAMER relates a set of general knowledge sources to the details of the particular game. Among other properties, this analysis determines the relative value of the different pieces in a given game. Although METAGAMER does not learn from experience, the values resulting from its analysis are qualitatively similar to values used by experts on known games, and are sufficient to produce competitive performance the first time METAGAMER plays a new game. Besides being the first Metagame-playing program, this is the first program to have derived useful piece values directly from analysis of the rules of different games. This paper describes the knowledge implemented in METAGAMER, illustrates the piece values METAGAMER derives for chess and checkers, and discusses experiments with METAGAMER on both existing and newly generated games.

Key words: games, metagame, heuristic search, evaluation-function learning, strategy, rule analysis, chess, knowledge representation, methodology, game generation

1. INTRODUCTION

Virtually all past research in computer game-playing has attempted to develop computer programs which could play existing games at a reasonable standard. While some researchers consider the development of a game-specific expert for some game to be a sufficient end in itself, many scientists in AI are motivated by a desire for generality. Their emphasis is not on achieving strong performance on a particular game, but rather on understanding the general ability to produce strength on a wider variety of games (or problems in general). Hence additional evaluation criteria are typically placed on the playing programs beyond mere performance in competition: criteria intended to ensure that methods used to achieve strength on a specific game will transfer also to new games. Such criteria include the use of learning and planning, and the ability to play more than one game.

However, even this generality-oriented research is subject to a potential methodological bias. The human researchers know at the time of program-development which specific game or games the program will be tested on, and therefore it is possible that they import the results of their own understanding of each game directly into their program. In this case, it is difficult to determine whether the subsequent performance of the program is due to the general theory it implements, or merely to the insightful observations of its developer about the characteristics necessary for strong performance on this particular game. An instance of this problem is the *fixed representation trick* (Flann & Dietterich 1989), in which many developers of learning systems spend much of their time finding a representation of the game which will allow their systems to learn how to play it well.

This problem is seen more easily when computer game-playing with known games

is viewed schematically, as in Figure 1. Here, the human researcher or programmer

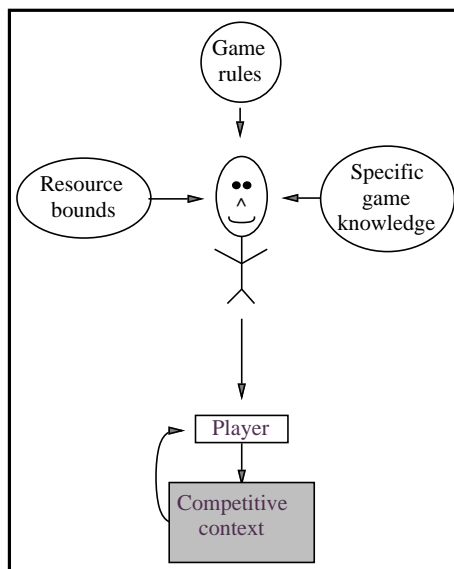


FIGURE 1. Computer Game-Playing with existing games: the human programmer mediates the relation between the program and the game it plays.

is aware of the rules and specific knowledge for the game to be programmed, as well as the resource bounds within which the program must play. Given this information, the human then constructs a playing program to play that game. The program then plays in competition, and is modified based on the outcome of this experience, either by the human, or perhaps by itself in the case of experience-based learning programs. In all cases, what is significant about this picture is that the human stands in the center, and mediates the relation between the program and the game it plays.

1.1. Metagame

Most AI games researchers, when pressed, will confess that their real interest is not in writing an expert-level chess program or checkers program. Their interest is in understanding general principles that are best tested by play at these well-understood games. This introduces the possibility of experimental bias. As discussed in the previous section, an experimenter can design a program that takes advantage of peculiar features of a single game, or use a tuned representation to permit the program to learn features of which the experimenter is well aware.

The concept of *Meta-Game Playing* (Pell 1992a), shown schematically in Figure 2, is an attempt to reduce the possibility of such bias. Rather than designing a program to play a game known in advance, we design a *metagamer* to play a large, well-defined *class* of games. A metagamer takes as input only the rules of a new game (as produced by an automatic game generator and encoded in a well-specified language), and then produces a specialized program (a *player*) to play that game. Different metagamers are evaluated in the context of a *Metagame-tournament*, in which a set

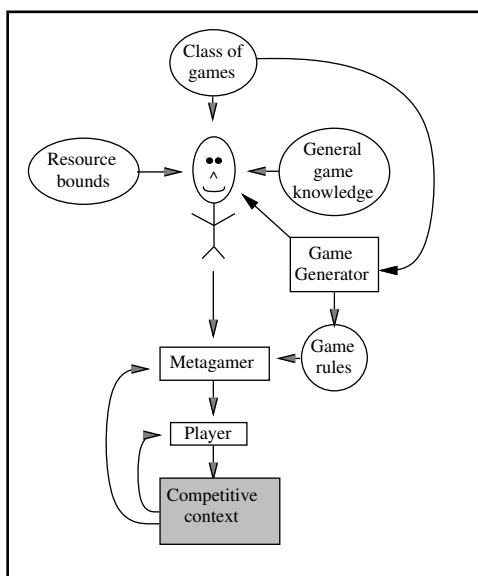


FIGURE 2. Metagame-playing with new games.

of new games are generated, the game rules are provided directly to the programs, and the programs then play the games against each other without human intervention. As only the *class* of games is known to the human developer in advance, metagamers are required to perform any game-specific optimizations without human assistance. The challenge is to produce the metagamer which receives the highest overall score across all game instances and opponents in the tournament. In contrast with the discussion on existing games above, the human no longer mediates the relation between the program and the games it plays; instead, she mediates the relation between the program and the *class* of games it plays. By making the class explicit, we are able to quantify the level of generality achieved. Moreover, we can begin with classes which represent only moderate generalizations over tasks we have already looked at, and gradually move to more general classes of problems as scientific understanding develops.

1.2. SCL-Metagame

SCL-Metagame (Pell 1992b) instantiates this concept in the class of symmetric chess-like games. The class includes the games of chess, checkers, tic tac toe, Chinese-chess, and Shogi (Japanese Chess). An implemented game generator produces new games in this class (some of which are objects of interest in their own right).

1.2.1. Class of Games. A *symmetric chess-like game* is a two-player game of perfect information, in which the two players move pieces along specified directions, across rectangular boards. Different pieces have different powers of movement, capture, and promotion, and interact with other pieces based on ownership and piece type. Goals involve eliminating certain types of pieces (*eradicate goals*), driving a

player out of moves (*stalemate goals*), or getting certain pieces to occupy specific squares (*arrival goals*). Most importantly, the games are *symmetric* between the two players, in that all the rules can be presented from the perspective of one player only, and the differences in goals and movements are solely determined by the direction from which the different players view the board. For a formal definition and analysis of this class of games, see Pell (1992b).

As an illustration of how chess-like games are defined in this class, Figure 3 presents a grammatical representation of the complete rules for American Checkers as a symmetric chess-like game. As shown in the figure, the game definition defines the game in terms of the board topology (an 8 by 8 square planar board), initial setup (*men* on the odd squares of the first three rows), goals (to run the opponent out of moves), constraints (a player must make a capture move whenever possible), and piece definitions. The pieces are defined in terms of moving, capturing, and promoting powers. The capturing power here means that pieces capture by hopping over a diagonally adjacent enemy piece, which is then captured. Both moving and capturing powers are composed out of movements (like diagonal hops and orthogonal lines), which are defined with explicit representation of symmetry. It is interesting to note that the only differences here between the definitions of a *king* and a *man* are that the king has a forward symmetry (i.e., it can move backward) and does not promote.

1.2.2. Game Generation. The goal of game generation is to produce a wide variety of games, all of which fall in the class of games as described by a formal grammar. We have implemented a statistical game generator for this class (Pell 1992b) and analyzed the complexity and variety of games it produces (Pell 1993b).

1.3. Structure of Paper

With the provision of the class definition and generator, Pell (1992b) formally defined the Metagame research problem of SCL-Metagame. This paper extends that work by constructing the first program to play games in this class, and by evaluating this program within the Metagame paradigm. The rest of this paper is organized as follows. Section 2 discusses the general architecture and the class-specific knowledge implemented in METAGAMER. Section 3 illustrates how METAGAMER to determines piece values from the rules of a game. Section 4 discusses experiments with METAGAMER on chess and checkers and on newly generated games. Section 5 compares METAGAMER to other work on general game-playing programs and on automatic methods for determining feature values in games. Section 6 discusses limitations and future work, and Section 7 concludes the paper.

2. CONSTRUCTING A METAGAME-PLAYER

The main intention of Metagame is to serve as a test-bed for learning and planning. One obvious use of a testbed is to compute lower bounds: how well do existing techniques perform on the challenges of the testbed. In this section, we engineer a metagamer using existing game-playing techniques.

```

GAME          american_checkers
GOALS         stalemate opponent
BOARD_SIZE   8 BY 8
BOARD_TYPE   planar
PROMOTE_RANK 8
SETUP man AT {(1, 1) (3, 1) (5, 1) (7, 1) (2, 2) (4, 2)
              (6, 2) (8, 2) (1, 3) (3, 3) (5, 3) (7, 3)}
CONSTRAINTS  must_capture

DEFINE man
MOVING
MOVEMENT
  LEAP (1, 1) SYMMETRY {side}
END MOVEMENT
END MOVING
CAPTURING
CAPTURE
  BY {hop}
  TYPE [{opponent} any_piece]
  EFFECT remove
MOVEMENT
  HOP BEFORE [X = 0]
      OVER [X = 1]
      AFTER [X = 0]
  HOP_OVER [{opponent} any_piece]
  (1, 1) SYMMETRY {side}
END MOVEMENT
END CAPTURE
END CAPTURING
PROMOTING
  PROMOTE_TO king
END PROMOTING
CONSTRAINTS continue_captures
END DEFINE

DEFINE king
MOVING
MOVEMENT
  LEAP (1, 1) SYMMETRY {forward side}
END MOVEMENT
END MOVING
CAPTURING
CAPTURE
  BY {hop}
  TYPE [{opponent} any_piece]
  EFFECT remove
MOVEMENT
  HOP BEFORE [X = 0]
      OVER [X = 1]
      AFTER [X = 0]
  HOP_OVER [{opponent} any_piece]
  (1, 1) SYMMETRY {forward side}
END MOVEMENT
END CAPTURE
END CAPTURING
CONSTRAINTS continue_captures
END DEFINE

END GAME.

```

FIGURE 3. Definition of *American Checkers* as a symmetric chess-like game.

2.1. Search Engine

To this end, we implemented a generic search engine which can be used across all games in the class. The search engine is similar to that provided by the Smart Game Board (Kierulf, Chen, & Nievergelt 1990). The game-specific inputs to the search engine are routines to produce the set of legal moves in a given position (called move generators), routines to detect the end of game, and an evaluation function which assigns numeric values to each non-terminal position encountered during a search.

The Metagame search engine incorporates many standard game-playing search techniques (see Levy & Newborn (1991)). It is based on the *minimax* algorithm with *alpha-beta pruning*, *iterative deepening*, and the *principal continuation heuristic*. More details of the Metagame search engine are given by Pell (1993b).

2.2. Automated Efficiency Optimization

This powerful search engine should allow a playing program to search deeply. However, search speed is linked to the speed with which the primitive search operations of move-generation and goal detection can be performed. For a game-specific program, these components can be easily hand-engineered, but for a program which is to play an entire class of games, the excess overhead of such generality initially caused the search to be unbearably slow. This problem was overcome by using a knowledge compilation approach. We represent the semantics of the class of games in an extremely general but inefficient manner, and after receiving the rules of a given game the program automatically partially evaluates the general theory with respect to the given game, thus compiling away both unnecessary conditionality and the overhead of interpretation (Pell 1993a). The result of this compilation process is that the general program (METAGAMER) can read in the rules of a specific game, like chess, and output two of the chess-specific components (move generators and goal detectors) needed by the search engine.

In addition to the partial evaluation techniques discussed by Pell (1993a), METAGAMER contains a set of pre-computation methods which produce *static-analysis tables*. These tables speed up position evaluation by caching compute-intensive properties, in some cases providing approximate versions of knowledge which would be too expensive to compute correctly at runtime (Pell 1993b).

2.3. Meta-Level Evaluation Function

We have so far discussed a generic search engine and a method to generate two of the game-specific components it requires. The final component required by the search engine is a game-specific evaluation function. This section provides some background on evaluation functions in computer game-playing and then describes the knowledge used by METAGAMER to arrive at a game-specific evaluation function for each game it plays.

A standard evaluation function has a set of component functions, called *features*, each of which maps a game position into a number. For example, chess programs have a separate *material* feature for each type of piece (like queen, rook, and bishop) which counts how many pieces of that type the players have on the board. Some programs also have a table of *positional* features, which are used to favor certain locations for different types of pieces. Two other typical examples are features that count the *mobility* (number of moves available) of each player in the current position, and features that recognize threats by either player to capture an opponent's piece. Each feature thus computes a number from a position, called the *value* of the feature in the current position, and the evaluation function combines the numbers to yield a final number, called the *evaluation* of the position. The most common functional form of the evaluation function is a weighted sum (i.e., a linear combination) of the values of all the features in the current position. The choice of relative *weight* given to each feature has dramatic implications on the performance of a playing program, and much human effort is spent in choosing a good set of weights.

The evaluation functions used in strong game-playing programs are largely *game-specific*, in the sense that the components of the function (the features and weights) only make sense in the context of the current game. For example, the feature *number-of-bishops* is only applicable to a game which has bishops, as opposed to the game-independent feature *number-of-moves*. In general, game-specific features refer to the

rules of the specific game, which include piece movement definitions (e.g., how a bishop moves), goal definitions (e.g., a player wins when the king is captured) and piece interactions (e.g., some pieces capture only certain enemy pieces). Similarly, the weights chosen for a set of game-specific features are obviously game-specific.

Since METAGAMER has to play new games entirely without human assistance, it is impossible to provide the program with a game-specific evaluation function for each game it plays.¹ Thus it must either make use of a game-independent evaluation function, which can be used for any game it might play, or construct its own evaluation function for each new game. The approach we have taken is actually a mixture of these two extremes. First, we provide METAGAMER with a set of game-independent features. Similar to the *advisors* in HOYLE (Epstein 1989), these measure properties of a position which can be determined using the move generator as a black box. Examples include immediate-mobility, which counts the legal moves in a position; threat, which recognizes a threat to remove an enemy piece; and piece-count, which counts the total number of pieces for each player.

These game-independent features were sufficient to produce a legal player, but led to only weak performance on test games (like chess and checkers). In an attempt to emulate the evaluation functions used in strong chess and checkers programs, we gave METAGAMER the ability to construct its own game-specific features and weights for each new game it plays. While the automatic construction of novel game-specific features for arbitrary logical problems is a difficult task ((de Grey 1985; Callan & Utgoff 1991; Fawcett & Utgoff 1992)), the feature-construction task is considerably simpler in our case. In particular, we have taken advantage of the fact that METAGAMER plays a well-defined class of games by giving METAGAMER methods to construct features which are commonly used in programs which play instances of the class (namely chess and checkers programs). Essentially, METAGAMER constructs a *material_X* feature for each piece *X* defined in the game rules. The *material_X* feature counts the number of pieces of type *X* present in the current position. Similarly, for each piece in the defined piece set and each square on the defined game board, METAGAMER makes a *positional_X* feature which registers 1 if piece *X* is on the corresponding square in the current position, and 0 otherwise. Applied to chess, for example, this process constructs a material feature for each of the 12 chess pieces types (6 white piece types and 6 black piece types) and a positional feature for each of those 12 piece types on each of the 64 squares, leading to 768 positional features.

While naming these game-specific features is straightforward, it is a much harder task to come up with a good set of weights for all of those features. One approach is to have the program learn these weights from experience playing the game, through a combination of self-play and competition. While this approach remains an open research area for Metagame-playing (see section 6), it has the disadvantage that it appears to require playing many games to reach a competitive standard (especially given the need to find weights for approximately 1000 features). Instead of using such a learning approach, we have provided METAGAMER with knowledge of how to choose its own weights for each of the game-specific features it constructs. This knowledge is encoded in a *feature evaluation function (FEF)*. An FEF takes as input a feature name (like the material value of a specific piece type), evaluates that feature with

¹For testing purposes it is possible to provide an evaluation function to the program. But in competition, the program must take in rules of generated games a human has never seen before and then play the game without human intervention.

respect to a set of functions called *subfeatures*, and combines them into a number which is used as the weight for that feature. Just as a feature in a state evaluation function measures some aspect of a position which is relevant to the overall value of the position (such as mobility, threats, and material), a subfeature in an FEF measures some aspect of the feature which is relevant to that feature's importance in a state evaluation function. For example, in the case of a material feature, the following considerations are among those which influence its weight:

- Which types of enemy pieces can it capture?
- How much mobility does that piece have in general?
- How many different squares can it reach in a given number of turns?
- Can it promote into other good pieces?
- Is it immune to capture by certain enemy pieces?
- Can it be used to achieve important goals in the game?

As in the case of state evaluation functions, there are several choices of functional form for combining the subfeature values into an overall weight for the feature. For purposes of simplicity, we have chosen to use a weighted sum in this case as well. This means that we need weights for the subfeatures, which will be combined to get the weight for the overall feature. In effect, then, we have replaced the need for game-specific feature weights with the need for game-independent feature and subfeature weights.

The use of the weighted sum (or linear combination) as functional form in both cases means that we must characterize the importance of features and subfeatures independent of the contributions of the other features. While this is a limitation in terms of expressiveness, it is also useful in that it facilitates a modular encoding of knowledge. Similar to Epstein's HOYLE (Epstein 1989) architecture, the knowledge represented in each feature and subfeature recognizes properties which should be significant about a position, other things being equal. Based on this similarity, we follow Epstein and refer to our game-independent knowledge sources (both features and subfeatures) as *advisors*. However, unlike HOYLE's advisors which can comment negatively about a position, we follow the treatment of goals by Wellman & Doyle (1991), in that we only represent in our advisors aspects of a position or feature which should be *favorable* to a player, other things being equal. Aspects of a position which are bad for a player are designed to be recognized as aspects which are good for the opponent, and so need not be recognized separately. In this way the advisors can be viewed as being always constructive: they never tell a player that some aspect is bad for the player, but they sometimes point out that an aspect is good for the opponent.

It is of course possible that a parameter-learning system would find some advisors to be negatively correlated with success, and that negating those weights could produce improved performance. This would suggest to us that other things are not, in fact, equal, and should motivate a program (or its designer) to search for an advisor which recognized the aspects which become favorable for the opponent as a player increases the value of the negatively-correlated advisor.

This concludes the general discussion of the knowledge used by METAGAMER to produce game-specific evaluation functions when provided with the rules of a new game. To summarize: METAGAMER constructs its own game-specific evaluation function (normally requiring a human decision concerning game-specific features and weights) from the following components:

- game-independent features (e.g., mobility, threats)
- game-independent feature generators (e.g., features for piece types and positional bonuses)
- game-independent subfeatures for each feature generator (e.g., average mobility of a piece, number of squares piece can reach from a given square), and
- weights for the game-independent features and subfeatures

The next two subsections provide details of the knowledge encoded in METAGAMER at present. Section 2.4 presents the advisors in detail, and Section 2.5 discusses the problem of assigning weights to game-independent advisors.

2.4. Game-Independent Advisors

This section briefly explains some of the advisors currently implemented for METAGAMER. The advisors (both features and subfeatures) were derived from an extensive human analysis of the class of games, performed by the author. This analysis took three forms, which we shall only briefly outline here. First, we generalized features used by programs which play existing games in the class to get game-independent features, such as mobility and capturing. Second, we made use of expert knowledge to find the subfeatures (considerations) used to produce the weights for certain game-specific features in existing games (like chess and checkers piece values). Finally, we observed the program's performance on generated games. When METAGAMER failed to make a distinction in a position which was obvious to a human observer, this suggested the addition of an advisor to capture this distinction.

It should be noted that this list is incomplete due to space limitations, the set is still growing, and there are several important general heuristics which are not yet incorporated (such as *distance* and *control* (Levinson & Snyder 1993)). Motivation and more detailed descriptions of all advisors are provided in (Pell 1993b). The advisors can be categorized into four groups, based on the general concepts from which they derive.

2.4.1. Mobility Advisors. The first group is concerned with different indicators of *mobility*. These advisors were inspired in part by Church & Church (1979) and Botvinnik (1970), and by generalizing features used for game-specific programs (Pell 1993b).

- **capturing-mobility**: counts the number of captures each piece can make in the current position.
- **dynamic-mobility**: counts the squares to which a piece can move directly from its current square on the current board, using a *moving* ability.²
- **static-mobility**: a static version of the above, this counts the squares to which a piece could move directly from its current square on an otherwise empty board.
- **eventual-mobility**: measures the total value of all squares to which a piece could move eventually from its current square on an otherwise empty board, using a *moving* ability. The value of each square decreases with the number of moves required for the piece to get there. Thus while a bishop has 32 eventual moves and a knight has 64 from any square, the bishop can reach most of its squares more quickly, a fact captured by this advisor.

²Pieces in this class (e.g., checkers pieces) may move and capture in different ways (see Section 1.2).

Note that static calculations such as **static-mobility** and **eventual-mobility**, which are based on analysis on an otherwise empty board, deliver no information about the capturing power of a piece. However, such information is considered by dynamic advisors, such as **capturing-mobility**.

2.4.2. Threats. The second group of advisors deals with threats and conditions enabling threats. The advisors come in two types, *local* and *global*. Local advisors assess each threat separately, and return the sum of the values of all threats in a position. Global advisors determine the most important of these threats, and return only the maximum value of the local advisors. To determine the value of a threat, these advisors make use of the other advisors to assess, for example, the contribution a threatened piece makes to the present position.

- **basic-threat**: for each target piece which a piece could capture in the current position, this measures the value of executing this threat (based on the other advisors), but reduces this value based on which player is to move. The rationale for this reduction is that a threat in a position where a player is to move is almost as valuable for that player as the value derived from executing it (i.e., he can capture if he likes), but if it is the opponent's move the threat is less valuable, as it is less likely to happen. In general, attacking an enemy piece while leaving one's own piece attacked (if the pieces are of equal value) is a losing proposition, but if the threatened enemy piece is worth much more this may be a good idea. The value of these threats are also augmented based on the *effect* of the capture.

- **potent-threat**: this extracts from the basic-threat analysis just those threats which are obviously potent. A threat is *potent* for the player on move if the target is either undefended or more valuable (based on the other advisors) than the threatening piece. A threat is potent for the non-moving player only if the attacker is less valuable than the target and the moving-player does not already have a potent threat against the attacker.

2.4.3. Goals and Step Functions. The third group of advisors is concerned with goals and regressed goals (indicators of potential goal achievement) for this class of games. For a review of the types of goals in symmetric chess-like games, see section 1.2.

- **vital**: Measures dynamic progress by both players on goals to eradicate sets of pieces.

- **arrival-distance**: This is a decreasing function of the abstract number of moves it would take a piece to *move* (i.e., without capturing) from its current square to a goal *destination* on an otherwise empty board. This abstract number is based on the minimum distance to the destination (if the path were empty) and the cost or difficulty of clearing the path. For example, it is easier to move our own pieces out of the way than to force enemy pieces out of the way. If we must force an enemy away, it is easier if we have a piece which can capture it. It is still easier to traverse a path with no obstacles at all.

- **promote-distance**: For each *target-piece* that a piece could promote into, this measures both the value and difficulty of achieving such a promotion.

- **possess**: This advisor handles games involving *placements*, in which a player can place a piece down on any of a set of squares. Examples of such placement games are Shogi and Nine-Men's Morris. The value in such a situation is a fraction of the

value the piece would have on the maximum available square, as determined using other advisors.³

For some of the advisors which anticipate goal-achievement, such as **promote-distance** and the threat advisors, we have placed an additional constraint on the weight they can take in an evaluation function. The reason is that they return some fraction (say F for a particular advisor) of the value which would be derived if the goal they anticipate were to be achieved. If such an advisor received a weight which was greater than the inverse of that fraction ($1/F$), the value of the threat could exceed the anticipated value of its execution, and the program would not in general choose to execute its threats. This type of constraint is commonly used when tuning weights for chess programs, as otherwise, for example, a pawn on the seventh rank may be seen as more valuable than the queen into which it could promote.⁴ The fraction F for each such advisor is explicitly declared when the advisor is defined, and this constraint is enforced for any setting of weights METAGAMER plays with.

2.4.4. Material Value. The final group of advisors are subfeatures used for assigning a fixed material value to each type of piece, which is later awarded to a player for each piece of that type he owns in a given position. This value is a weighted sum of the values returned by the advisors listed below, and does not depend on the position of the piece or of the other pieces on the board. Static-mobility and eventual-mobility are discussed with respect to Mobility advisors above.

- **max-static-mob**: The maximum static-mobility for this piece over all board squares.
- **avg-static-mob**: The average static-mobility for this piece over all board squares.
- **max-eventual-mob**: The maximum eventual-mobility for this piece over all board squares.
- **avg-eventual-mob**: The average eventual-mobility for this piece over all board squares.
- **victims**: Awards 1 point for each type of piece this piece has the ability to capture (i.e., the number of pieces matching one of its *capture-types*).⁵
- **eradicate**: Awards 1 point for each opponent goal to eradicate this piece, and minus one point for each player goal to eradicate this piece.
- **stalemate**: This views the goal to stalemate a player as if it were a goal to eradicate all of the player's pieces, and performs the same computation as *eradicate* above.
- **immunity**: Awards 1 point for each type of enemy piece that *cannot* capture this piece.
- **giveaway**: Awards 1 point for each type of friendly piece that *can* capture this piece. This may be important in games in which the goal is to eliminate some of your own pieces.
- **arrive**: This awards positive points to pieces which help a player achieve arrival

³This analysis would have to be improved substantially to capture the complexities of possessed pieces in games like Shogi.

⁴The chess-master Tartakover (quoted in (Hunvald 1972)) said, "A threat is more powerful than its execution." But he did not mean by this that a player should prefer to have threats against pieces than to have captured them for free.

⁵A more sophisticated version of this feature, not fully implemented yet, takes into account the value of each victim, as determined by other static advisors.

goals, and negative points to those which help the opponent.

- **promote**: This is computed in a separate pass after all the other material values. It awards a piece a fraction of the material value (computed so far) of each piece it can promote into. This advisor is not fully implemented yet, and was not used in the work discussed in this paper.

Section 3 provides concrete examples of the application of these advisors to the rules of different games discussed in this paper.

2.5. Weights for Advisors

This section discusses the problem of assigning weights for game-independent advisors. It must be emphasized that the problem of finding good weights for the advisors for use in a specific game is different from the problem of finding weights for game-specific features, as those weights are derived by METAGAMER automatically using the subfeature advisors (see section 2.3). Thus, a decision about advisor weights is a decision about the relative importance of short-term mobility, long-term mobility, and threats against the opponent (among other factors). It is not a decision directly about the relative values of pieces, for example, a knight and a bishop.

The use of subfeatures provides a significant constraint on the relative weights generated for game-specific features. For example, consider two pieces A and B such that A is defined to have all the powers of B plus some other powers which are recognized as useful by some subfeatures. A concrete example of this situation is where A and B are, respectively, the queen and rook in chess: the queen is defined to have all the powers of a rook plus diagonal moves, and the two pieces are interchangeable with respect to all other game properties (e.g., all pieces with power to capture a rook also have power to capture a queen, and conversely). So long as the subfeatures are all weighted positively (which is consistent with their design, as discussed in section 2.3), the weight generated for a queen will always be greater than the weight generated for a bishop.

As a consequence of this, it can be seen that even a random setting of weights to subfeature advisors should still result in a competitive advantage over a player that assigned random weights directly to the game-specific features (assuming that the general encoded knowledge is useful at all). For example, a random assignment of weights directly to queen and rook would lead to a rook being weighted higher than a queen half the time, whereas a random assignment of weights to the mobility and capturing-ability advisors would always leave a queen more highly weighted.

This discussion shows that subfeatures, which are combined to assign weights to features, can provide advantageous constraints on preferences even with a random or uniform weighting. The same point is true of features, which are combined to assign value to a position. So long as the features are positively related to success, a position A which is identical to a position B except that A is more favorable with respect to a feature F will always be more highly valued than B , even if a random set of feature weights is used. Abramson (1990) found such a result when learning weights for base-level features in chess: a program using a random assignment of weights for chess pieces performed significantly better than a program which assigned random values to positions as a whole.

Thus if only one feature is different between positions, or one subfeature is different between two pieces, then the distinguishing advisor in both cases leads the program to correct preferences independent of the relative weights of the advisors. However, when there are multiple differences between two choices, then tradeoffs

arise and a choice of relative weights for advisors can have a significant impact on the playing performance of the program. Two examples of such tradeoffs are (a) whether a player should sacrifice some material to gain mobility in a particular position and (b) whether to prefer a highly mobile piece which can only reach half the squares on the board or a less mobile piece which can eventually reach all the board squares.

In the current version of METAGAMER, we have set the weights of all the advisors to be equal, at one point each. This has yielded sufficient performance in games in which pieces tend to differ on only a few dimensions (which is the case in chess and checkers, for example), and in games in which the METAGAMER does not encounter significant tradeoffs during search (which often is true due to its limited search depth). However this uniform weighting has obvious limitations, which are reflected for example in METAGAMER's under-estimation of the value of pieces which can promote into other pieces (see section 3). Section 6 discusses future research directions on improving the choice of weights for advisors in Metagame-playing.

3. EXAMPLES OF MATERIAL ANALYSIS

One important aspect of METAGAMER's game analysis, which was discussed in Section 2.4.4, is concerned with determining relative values for each type of piece in a given game. This type of analysis is called *material analysis*, and the resulting values are called *material values* or *static piece values*. This section demonstrates METAGAMER's material analysis when applied to chess and checkers. In both cases, METAGAMER took as input only the rules of the games. In conducting this analysis, METAGAMER used the material advisors (Section 2.4.4) all with equal weight of one point each.

3.1. Checkers

Table 1 lists material values determined by METAGAMER for the game of checkers, given only the encoding of the rules as presented in Figure 3. In the table, *K* stands for *king*, and *M* stands for *man*. For compactness, advisors which do not apply to a game (and thus have value of 0 for all pieces) are not listed in material analysis tables for that game.

METAGAMER concludes that a king is worth almost two men. According to expert knowledge,⁶ this slightly underestimates the value of a man. Strong checkers players value a king as worth about 1.5 men, although the actual value can vary depending on position. The reason that men are undervalued here is that METAGAMER does not yet consider the static value of a piece based on its possibility to promote into other pieces (see Section 2.4.4). When actually playing a game, METAGAMER does consider this, using the dynamic **promote-distance** advisor.

3.2. Chess

Table 2 lists material values determined by METAGAMER for the game of chess, given only an encoding of the rules similar to that for checkers (Pell 1993b). In the table, the names of the pieces are just the first letters of the standard piece names,

⁶I am thankful to Nick Flann for serving as a checkers expert.

Material Analysis: checkers		
Advisor	Piece	
	K	M
max-static-mob	4	2
max-eventual-mob	6.94	3.72
avg-static-mob	3.06	1.53
avg-eventual-mob	5.19	2.64
eradicate	1	1
victims	2	2
stalemate	1	1
Total	23.2	13.9

TABLE 1. Material value analysis for checkers.

except that N refers to a knight.

Material Analysis: chess						
Advisor	Piece					
	B	K	N	P	Q	R
max-static	13	8	8	1	27	14
max-eventual	12	12.9	14.8	1.99	23.5	20.2
avg-static	8.75	6.56	5.25	0.875	22.8	14
avg-eventual	10.9	9.65	11.8	1.75	22.4	20.2
eradicate	0	1	0	0	0	0
victims	6	6	6	6	6	6
stalemate	1	1	1	1	1	1
Total	51.7	45.1	46.9	12.6	103	75.5

TABLE 2. Material value analysis for chess.

As discussed for checkers above, pawns are here undervalued because METAGAMER does not consider their potential to promote into queens, rooks, bishops, or knights. According to its present analysis, a pawn has increasingly less eventual-mobility as it gets closer to the promotion rank. Beyond this, the relative value of the pieces is surprisingly close to the values used in conventional chess programs (queen=9, rook=5, bishop=3.25, knight=3, and pawn=1) (Botvinnik 1970; Abramson 1990), given that the analysis was so simplistic.

4. SUMMARY OF RESULTS

An evaluation of the game-playing performance of METAGAMER was carried out by Pell (1993b) across a set of existing and generated games.

4.1. Known Games

Performance in the games of chess and checkers demonstrated that the knowledge encoded in METAGAMER endows it with a modest level of competence against highly specialized programs, which was still impressive given that METAGAMER plays these games, in effect, from first-principles.

4.1.1. Checkers. The performance of METAGAMER in checkers was assessed by playing it against Chinook (Schaeffer *et al.* 1991). Chinook is the world's strongest computer checkers player, and the second strongest checkers player in general. As it is a highly optimized and specialized program, it is not surprising that METAGAMER always loses to it (at checkers, of course!). However, to get a baseline for METAGAMER's performance relative to other possible programs when playing against Chinook,⁷ we have evaluated the programs when given various handicaps (number of men taken from Chinook at the start of the game).

The primary result from the checkers experiments was that METAGAMER is around even to Chinook, when given a handicap of one man. This is compared to a deep-searching greedy material program which requires a handicap of 4 men, and a random player, which requires a handicap of 8. In fact, in the 1-man handicap positions, METAGAMER generally achieves what is technically a winning position, but it is unable to win against Chinook's defensive strategy of hiding in the double-corner.

On observation of METAGAMER's play of checkers, it was interesting to see that METAGAMER "re-discovered" the checkers strategy of not moving its back men until late in the game. It turned out that this strategy emerged from the `promote-distance` advisor, operating defensively instead of in its "intended" offensive function. In effect, METAGAMER realized from more general principles that by moving its back men, it made the promotion square more accessible to the opponent, thus increasing the opponent's value, and decreasing its own.

4.1.2. Chess. In chess, METAGAMER played against GnuChess, a very strong publicly available chess program.⁸ GnuChess is vastly superior to METAGAMER (at chess, of course!), unless it is handicapped severely in time and moderately in material. The overall result of the experiments was that METAGAMER is around even to GnuChess on its easiest level,⁹ when given a handicap of one knight. For comparison, a version of METAGAMER with only a standard hand-encoded material evaluation function (queen=9, rook=5, bishop=3.25, knight=3, and pawn=1) played against METAGAMER with all its advisors and against the version of GnuChess used above. The

⁷In our experiments, Chinook played on its easiest level and it played without access to its opening book or endgame database. However, it is unlikely that the experimental results would have been much altered had it been using these components, since the opening book is not relevant in handicap games and Chinook is already much stronger than METAGAMER in the endgame.

⁸GnuChess was the winner of the C Language division of the 1992 Uniform Platform Chess Competition.

⁹In the experiments, GnuChess played on level 1 with depth 1. This means it searches 1-ply in general but can still search deeply in quiescence search. METAGAMER played with one minute per move, and occasionally searched into the second-ply.

result was that the material program lost every game at knight's handicap against GnuChess, and lost every game at even material against METAGAMER with all its advisors. This showed that METAGAMER's performance was not due to its search abilities, but rather to the knowledge in its evaluation function.

On observation of METAGAMER's play of chess, we have seen the program develop its pieces quickly, place them on active central squares, put pressure on enemy pieces, make favorable exchanges while avoiding bad ones, and restrict the freedom of its opponent. In all, it is clear that METAGAMER's knowledge gives it a reasonable *positional* sense and enables it to achieve longer-term strategic goals while searching only one or two-ply deep. This is actually quite impressive, given that none of the knowledge encoded in METAGAMER's advisors or static analyzer makes reference to any properties specific to the game of chess—METAGAMER worked out its own set of material values for each of the pieces (see Section 3), and its own concept of the value of each piece on each square. On the other hand, the most obvious immediate limitation of METAGAMER revealed in these games is a weakness in *tactics* caused in part by an inability to search more deeply within the time constraints, in part by a lack of quiescence search, and also by the reliance on full-width tree-search. These are all important areas for future work.

4.2. New Games

Pell (1993b) carried out an experiment in the form of a *Metagame tournament*. In the experiment, several versions of METAGAMER with different settings of weights for their advisors played against each other and against baseline¹⁰ players on a set of generated games which were *unknown* to the human designer in advance of the competition. The rules were provided directly to the programs, and they played the games without further human intervention.

The procedure used in the experiment was as follows. We began by choosing settings of the *generator* which would further the goals of the experiment. We then decided on the layout of the tournament, in terms of the number of players, the number of generated games, the number of contests per game against each opponent, the maximum game-length before a draw is declared, and the time-constraints under which a given contest was played. Then we chose a set of players to play whatever games were later generated. Once the entire format of the tournament was specified, all of these details were fixed thereafter. That is, we did not change the details of any programs or their parameters, alter any generated games in any way, or experiment with different time-limits: we ceased to play a role in the process. The games were then generated and the rules fed directly to the programs, which then played the tournament without human intervention. It should be noted here that none of the players modified themselves, either, in the course of the tournament (except for building analysis tables when presented with the games). That is, no experience-based learning took place.

Each of the programs played every other program in a match of 20 contests (10 as white, 10 as black) on each of the 5 generated games. With 15 pairings, 5 generated games, and 20 contests per game, there were thus a total of 1500 contests. Each contest was declared a draw if 200 moves were played without a goal achieved, and programs were given 30 seconds (0.5 minutes) to make each move. Thus the

¹⁰The baseline players consisted of a random player and a player that conducted a 2-ply search using a random evaluation function at non-terminal positions.

maximum time for running the entire tournament (if every game was a draw) was $1500 \times 200 \times 0.5 = 150,000$ minutes, or 2500 hours of cpu-time.¹¹

Two results of the experiment were particularly significant. First, the version of METAGAMER which made use of the most knowledge clearly outperformed all opponents in terms of total score on the tournament. This was true despite the added evaluation cost incurred when using more knowledge. This result is evidence that the knowledge implemented in METAGAMER provides it with competitive strength across the entire class of games, at least relative to more limited versions of itself and to a set of baseline programs. Second, despite the strong overall performance of the most knowledge-intensive version of METAGAMER in the tournament, no single version of METAGAMER (as defined by weight settings) performed the best on each individual game. This suggests that an experience-based learning program, which could customize the weights for its advisors based on experience in competition on a particular game, might have a demonstrable competitive advantage in a Metagame-tournament.

5. RELATED WORK

This section compares METAGAMER to other work on general game-playing programs and on automatic methods for determining feature values in games. There have been many efforts to develop general game-playing programs which have been tested on more than one game (Epstein 1989; Williams 1972; Tadepalli 1989; Collins *et al.* 1991; Callan, Fawcett, & Rissland 1991; Gherrity 1993). However, none of these programs have been evaluated within the context of Metagame, which poses several unique challenges. First, the human designer cannot influence the representation of specific games (as they are input directly to the metagamers). Second, there is no existing body of knowledge or game records for newly generated games, which poses difficulties for approaches which rely on learning concepts or weights from textbooks or large amounts of existing games. Finally, there are no existing experts against which programs can train, which suggests that stronger programs will be based on self-play (Tesauro 1994) or more active forms of rule analysis (such as that performed by METAGAMER).

Despite its simplicity, METAGAMER's analysis produced useful piece values for a wide variety of games, which agree qualitatively with the assessment of experts on some of these games. This appears to be the first instance of a game-playing program automatically deriving material values based on active analysis when given only the rules of different games. It also appears to be the first instance of a program capable of deriving useful piece values for games unknown to the developer of the program (Pell 1993b). The remainder of this section compares METAGAMER to previous work with respect to determination of feature values.

5.0.1. Expected Outcome and Self-Play. Abramson (1990) developed a technique for determining feature values based on predicting the *expected-outcome* of a position in which particular features (not only piece values) were present. The expected-outcome of a position is the fraction of games a player expects to win from a position if

¹¹The experiments were divided among 50 machines (DEC-Station 2500's) running in parallel, giving a maximum total time of 50 hours per machine. In practice the time was on average 25 hours per machine since at least half the games ended in a player winning.

the rest of the game after that position were played randomly. He suggested that this method was an *indirect* means of measuring the mobility afforded by certain pieces. The method is statistical, computationally intensive, and requires playing out many thousands of games. Similar considerations apply to work on self-play (Tesauro 1994; Epstein 1994). On the other hand, the analysis performed by METAGAMER is a *direct* means of determining piece values, which follows from the application of general principles to the rules of a game. It took METAGAMER under one minute to derive piece values for each of the games discussed in section 3, and it conducted the analysis without playing out even a single contest.

5.0.2. Automatic Feature Generation. There has recently been much progress in developing programs which generate features automatically from the rules of games (de Grey 1985; Callan & Utgoff 1991; Fawcett & Utgoff 1992). When applied to chess such programs produce features which count the number of chess pieces of each type, and when applied to Othello they produce features which measure different aspects of positions which are correlated with mobility. The methods operate on any problems encoded in an extended logical representation, and are more general than the methods currently used by METAGAMER. However, these methods do not generate the *weights* of these features, and instead serve as input to systems which may learn their weights from experience or through observation of expert problem-solving. While METAGAMER's analysis is specialized to the class of symmetric chess-like games, and thus less general than these other methods, it produces piece values which are immediately useful, even for a program which does not perform any learning.

5.0.3. Evaluation Function Learning. There has been much work on learning feature values by experience or by observation of strong players (e.g., (Samuels 1967; Lee & Mahajan 1988; Levinson & Snyder 1991; Callan, Fawcett, & Rissland 1991; Tunstall-Pedoe 1991)). These are all examples of passive analysis (Pell 1993b), and would not seem likely to produce a strong program in a Metagame-tournament until later rounds, after which the program would have had significant experience with stronger players.

5.0.4. Hoyle. HOYLE (Epstein 1989) is a program, similar in spirit to METAGAMER, in which general knowledge is encapsulated using the metaphor of *advisors*. HOYLE has an advisor responsible for guiding the program into positions in which it has high mobility. However, HOYLE does not analyze the rules of the games it plays, and instead uses the naive notion of immediate-mobility as the number of moves available to a player in a particular position. The power of material values is that they abstract away from the mobility a piece has in a particular position, and characterize the potential options and goals which are furthered by the existence of each type of piece, whether or not these options are realized in a particular position. As HOYLE does not perform any analysis of the rules or construct analysis tables as does METAGAMER, it is unable to benefit from this important source of power.

6. LIMITATIONS AND FUTURE WORK

This paper has demonstrated that SCL-Metagame can be practically addressed and that doing so necessitates increased understanding and implementation of many aspects of the process of game-analysis. However this work is just the start, and

virtually everything that was done here opens an interesting area of research, with the advantage that increased understanding should translate directly into competitive advantage. In particular, two main categories of future work are (a) improving performance of METAGAMER and SCL-Metagame-playing programs and (b) applying the idea of Metagame to other classes of games and problems in general.

6.1. Improved SCL-Metagame-playing programs

Three areas for future work are apparent from the performance of METAGAMER as observed in Section 4. These areas are learning weights for advisors, deriving new advisors automatically from the class definition, and more advanced search techniques.

6.1.1. Learning Weights. One way to summarize the discussion on weights in Section 2.5 is with the observation that the subfeature advisors are in effect feature-weight generators, which serve to modularize more general knowledge. When applied to a specific game, the advisors yield values for game-specific features as are used in current game-playing programs. But since a small number of general advisors generate a large number of specific values, it is clear that the game-specific values produced by the advisors are much more constrained than the corresponding values in a game-specific program.

Given the constraints and modularity imposed by these advisors, the meta-level weight learning issue is still important. For example, it is still the case that for different games, some sets of weights may lead to performance which is dramatically stronger than other sets of weights. While experiments discussed in Section 4.2 have demonstrated that even weighing all advisors equally results in competitive performance across a variety of games, these experiments also indicate that programs that can modify their general weights based on analysis or experience with a particular game will have a marked competitive advantage over those which do not. The issue of finding weights for advisors used by METAGAMER or similar programs is thus an important area for research.

One approach to this problem is to apply temporal-difference learning (Sutton 1984) and self-play (Tesauro 1994) to this problem. It would be interesting to investigate whether the “knowledge-free” approach which was so successful in learning backgammon (Tesauro 1994) and moderately successful in learning the game of Go (Schraudolph, Dayan, & Sejnowski 1994) also transfers to these different games, or whether it depends for its success on properties specific to certain types of games.

Another approach would be to use a genetic approach with competition among a population of players with different weights. One interesting recent approach to weight learning (Angeline & Pollack 1993) has a population of programs compete repeatedly in a knockout tournament, in which winners advance to the next round and losers also play against each other in consolation matches. This appears to be a more efficient method to extract information and compare programs than the format used in the Metagame-tournament summarized in Section 4.2, as the losing programs get weeded out of the competition early so that more effort can be focussed on the stronger programs.

6.1.2. Deriving New Advisors. While METAGAMER performs its own analysis of the rules of each game it plays, the analysis methods and the knowledge which draws on them were both the products of a human analysis of the whole class of games. The result is that the analysis methods implemented in METAGAMER are still very

simplistic and are only linked indirectly to the semantics of the class of games. As a consequence, for a wide variety of generated games METAGAMER's analysis offers no useful guidance. For example, all static analysis at present is based on the assumption of an otherwise empty board. Pieces which move only by hopping over other pieces are determined to have no mobility by this measure. In short, while the advisors have been well-motivated by the extensive human analysis by Pell (1993b), it would be desirable and competitively advantageous to have a more systematic method of deriving new general advisors directly from the semantics of the class definition.

One approach to this problem would be to extend the techniques we have used for efficiency specialization (Pell 1993a) (see Section 2 for a summary of these techniques) to derive useful subgoals and invariant properties for a given game by abstract interpretation. It might also be possible to apply *knowledge-based feature construction* (Callan 1993; Fawcett & Utgoff 1992) directly to this problem, as it is designed to extract functional features from a logical encoding of problem definitions.

6.1.3. Advanced Search Techniques. Section 4.1 noted that while METAGAMER's analysis of each game seemed to give it a basic understanding of strategy, it is noticeably weak in tactics. This weakness is caused in part by an inability to search more deeply within the time constraints, by a lack of quiescence search, and also by the reliance on full-width tree-search. Almost any improvement should produce a markedly better player. One way to address this problem might be through a tactics analyzer as developed by Berliner (Berliner 1974) or a knowledge-based planner as developed by Wilkins (Wilkins 1982). These would need to be extended to games beyond just chess.

Another approach would be to apply some of the recent developments in *rational game-tree search* (Russell & Wefald 1992; Baum & Smith 1993; Good 1968) to SCL-Metagame. These methods appear to be powerful and general tools, but within the traditional approach to games it has been nearly impossible to demonstrate their advantage against programs which have already been carefully engineered by humans to play a single game well. It would be exciting to apply these techniques to this new context, in which programs are required to perform game-specific optimizations without human assistance.

6.2. Extending Metagame To Other Classes of Games

This paper has discussed a program to play Metagame in the class of symmetric chess-like games. While this class of games has been the main testbed for AI research on games, it would be interesting and useful to apply the idea of Metagame to other classes of games which have also been important targets of AI research.

6.2.1. Poker and Trick-Taking Card Games. For example, a class of *trick-taking card games* would include *bridge* (Gambäck, Rayner, & Pell 1991; 1993; Frank, Basin, & Bundy 1992; Smith & Nau 1993), and *hearts* (Mostow 1983; Ellman 1988), games which feature chance, incomplete information, and communication as strategic elements. Another interesting class of games might include an infinite variety of *poker-like games*, which were the subject of a decade-long research project by Findler, Sicherman, & McCall (1983). Poker is in fact a particularly appropriate basis for Metagame-playing because a basic part of poker-playing by humans involves inventing modifications of the standard rules (e.g., declaring wild-cards and inverting the goal) and developing strategies to exploit novel rule sets, both of which are key

components of Metagame-playing.

6.2.2. Applying Metagame to Heuristic Search. While the two extensions above involve formalizing new classes of games, it would also be interesting to apply the paradigm of Metagame to evaluate heuristic search algorithms on a known problem. In the traditional evaluation of search algorithms, a researcher compares two different algorithms on a few known problems (a typical example being the eight-puzzle (Nilsson 1980, pp. 18–20)), and measures performance in terms of number of nodes generated and time to solution. One potential problem with this is that one search algorithm may outperform the other not because it is generally a better algorithm, but merely because it has a favorable interaction with the particular heuristic. Within the paradigm of Metagame, one approach might be to develop a generator of heuristic functions for a particular problem, and compare the performance of candidate algorithms on that problem across a large set of generated heuristic functions (fed directly to the programs). As the details of the functions would be unknown to human researchers in advance of the comparison, this method of evaluation would provide a degree of experimental control not present in the traditional paradigm.

7. CONCLUSION

This paper introduced METAGAMER, the first program designed within the paradigm of Meta-Game Playing (Metagame). The strategic analysis performed by METAGAMER relates a set of general knowledge sources to the details of each particular game it plays. These general knowledge sources represent a generalization of knowledge typically hand-coded in special-purpose game-playing programs. Although METAGAMER does not learn from experience, the values resulting from its analysis are qualitatively similar to values used by experts on known games, and are sufficient to produce competitive performance the first time METAGAMER actually plays each new game.

By identifying such general strategic game-playing knowledge and presenting techniques to represent and apply that knowledge (by means of game-independent features and subfeatures), this paper paves the way for a wide variety of future developments in increasingly general game-playing systems.

One additional contribution which resulted from the work in this paper is the development of a general platform which supports research in Metagame. This platform contains a formal class specification for chess-like games, the symmetric chess-like game generator, interpreters, baseline players, efficiency optimization routines, a generic search engine, a library of game definitions, and most of the code implementing METAGAMER. This platform has been made publicly available, under the name *Metagame Workbench*.¹²

The reception to this workbench, and to the idea of Metagame, has been favorable. It has been used as part of several courses on computer game-playing, and

¹²The Metagame Workbench has been placed on the world chess ftp server and can be retrieved from:

`ftp ics.onenet.net:pub/chess/projects/metagame/`

It has also been stored on the Prime Time AI CD-Rom, and can be retrieved from:

`ftp.cs.cmu.edu:user/ai/software/games/metagame/`

as the basis for several research projects. The results on existing and generated games, presented in this paper, establish METAGAMER as a competent competitor for SCL-Metagame. At present, a number of challenger Metagame-playing programs are under development by independent researchers in the games and learning communities. This raises the possibility of a large-scale Metagame-tournament in the near future, which is an exciting prospect.

ACKNOWLEDGEMENTS

Thanks to Jonathan Schaeffer for permitting me to use Chinook, and to Stuart Cracraft for making GnuChess available. Thanks also to John Allen, Graham Farr, Lise Getoor, Othar Hansson, Yaakov Kerner, Robert Levinson, Pandu Nayak, Manny Rayner, and the anonymous reviewers for useful comments on drafts of this work. Parts of this work have been supported by the British Marshall Scholarship, the American Friends of Cambridge University, Trinity College (Cambridge), the Cambridge University Computer Laboratory, and the Research Institute for Advanced Computer Science (RIACS).

REFERENCES

- AAAI. 1991. *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, Cambridge, Mass.: AAAI/MIT Press.
- ABRAMSON, B. 1990. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12(2).
- ANGELINE, P., AND POLLACK, J. 1993. Competitive environments evolve better solutions for complex tasks. In FORREST, S., *Editor.*, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann.
- BAUM, E. B., AND SMITH, W. D. 1993. Best play for imperfect players and game tree search. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*. AAAI Press.
- BERLINER, H. 1974. *Chess as Problem Solving: The Developments of a Tactics Analyzer*. Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh.
- BOTVINNIK, M. M. 1970. *Computers, chess and long-range planning*. Springer-Verlag New York, Inc.
- BRAMER, M. A., *Editor*. 1983. *Computer Game-Playing: theory and practice*. Ellis-Horwood Publishers.
- CALLAN, J. P., AND UTGOFF, P. 1991. Constructive induction on domain knowledge. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)* (1991), 614-619.
- CALLAN, J. P.; FAWCETT, T. E.; AND RISSLAND, E. L. 1991. Adaptive case-based reasoning. In *IJCAI-91* (1991).
- CALLAN, J. P. 1993. *Knowledge-Based Feature Generation for Inductive Learning*. Ph.D. Dissertation, Department of Computer and Information Science, University of Massachusetts.
- CHURCH, R. M., AND CHURCH, K. W. 1979. Plans, goals, and search strategies for the selection of a move in chess. In FREY, P. W., *Editor.*, *Chess Skill in Man and Machine*. Springer-Verlag.
- COLLINS, G.; BIRNBAUM, L.; KRULWICH, B.; AND FREED, M. 1991. Plan debugging in an intentional system. In *IJCAI-91* (1991).
- DE GREY, A. 1985. Towards a versatile self-learning board game program. Unpublished Final Project, Tripos in Computer Science, University of Cambridge.

- ELLMAN, T. 1988. Approximate theory formation: An explanation-based approach. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-88)*, 570-574. Cambridge, Mass.: AAAI.
- EPSTEIN, S. 1989. The Intelligent Novice - Learning to Play Better. In LEVY, D., AND BEAL, D., *Editors.*, *Heuristic Programming in Artificial Intelligence - The First Computer Olympiad*. Ellis Horwood.
- EPSTEIN, S. 1994. Toward an ideal trainer. *Machine Learning* 15(3).
- FAWCETT, T. E., AND UTGOFF, P. E. 1992. Automatic feature generation for problem solving systems. In SLEEMAN, D., AND EDWARDS, P., *Editors.*, *Proceedings of the Ninth International Workshop on Machine Learning*.
- FINDLER, N. V.; SICHERMAN, G. L.; AND MCCALL, B. 1983. A multi-strategy gaming environment. In Bramer (1983). chapter 17.
- FLANN, N. S., AND DIETTERICH, T. G. 1989. A study of explanation-based methods for inductive learning. *Machine Learning* 4(2):187-226.
- FRANK, I.; BASIN, D.; AND BUNDY, A. 1992. An adaptation of proof-planning to declarer play in bridge. In *Proceedings of the European Conference on Artificial Intelligence*, 72-76. Longer Version available from Edinburgh as DAI Research Paper No. 575.
- GAMBÄCK, B.; RAYNER, M.; AND PELL, B. 1991. An Architecture for a Sophisticated Mechanical Bridge Player. In LEVY, D., AND BEAL, D., *Editors.*, *Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad*. Ellis Horwood.
- GAMBÄCK, B.; RAYNER, M.; AND PELL, B. 1993. Pragmatic Reasoning in Bridge. Technical Report No. 299, University of Cambridge Computer Laboratory.
- GHERRITY, M. 1993. *A Game-Learning Machine*. Ph.D. Dissertation, Computer Science Department, University of California, San Diego.
- GOOD, I. J. 1968. A five-year plan for automatic chess. In DALE, E., AND MICHIE, D., *Editors.*, *Machine Intelligence 2*. Oliver and Boyd. 89-118.
- HUNVALD, H. 1972. *Chess: Quotations from the Masters*. Mount Vernon, New York: Peter Pauper Press.
- IJCAI. 1991. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Los Altos, CA: Morgan Kaufman Publishers.
- KIERULF, A.; CHEN, K.; AND NIEVERGELT, J. 1990. Smart Game Board and Go explorer: A case study in software and knowledge engineering. *Communications of the ACM* 33(2).
- LEE, K.-F., AND MAHAJAN, S. 1988. A pattern classification approach to evaluation function learning. *Artificial Intelligence* 36(1):1-25.
- LEVINSON, R. A., AND SNYDER, R. 1991. Adaptive, pattern-oriented chess. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)* (1991), 601-605.
- LEVINSON, R. A., AND SNYDER, R. 1993. Distance: Towards the unification of chess knowledge. *Int'l Computer Chess Association Journal* 16(3):123-136.
- LEVY, D., AND NEWBORN, M. 1991. *How Computers Play Chess*. W.H. Freeman and Company.
- MOSTOW, J. 1983. A problem-solver for making advice operational. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-83)*, 279-283. Cambridge, Mass.: AAAI.
- NILSSON, N. J. 1980. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Publishing Co.
- PELL, B. 1992a. Metagame: A New Challenge for Games and Learning. In VAN DEN HERIK, H., AND ALLIS, L., *Editors.*, *Heuristic Programming in Artificial Intelligence 3 - The Third Computer Olympiad*. Ellis Horwood. Also appears as University of Cambridge Computer Laboratory Technical Report No. 276.
- PELL, B. 1992b. Metagame in Symmetric, Chess-Like Games. In VAN DEN HERIK, H., AND ALLIS, L., *Editors.*, *Heuristic Programming in Artificial Intelligence 3 - The Third Computer Olympiad*. Ellis Horwood. Also appears as University of Cambridge Computer Laboratory Technical Report No. 277.

- PELL, B. 1993a. Logic Programming for General Game Playing. In *Proceedings of the ML93 Workshop on Knowledge Compilation and Speedup Learning*. Amherst, Mass.: Machine Learning Conference. Also appears as University of Cambridge Computer Laboratory Technical Report No. 302.
- PELL, B. 1993b. *Strategy Generation and Evaluation for Meta-Game Playing*. Ph.D. Dissertation, Computer Laboratory, University of Cambridge. Also appears as University of Cambridge Computer Laboratory Technical Report No. 315.
- RUSSELL, S., AND WEFALD, E. 1992. *Do the Right Thing*. MIT Press.
- SAMUELS, A. L. 1967. Some studies in machine learning using the game of Checkers. II. *IBM Journal* 11:601-617.
- SCHAEFFER, J.; CULBERSON, J.; TRELOAR, N.; KNIGHT, B.; LU, P.; AND SZAFRON, D. 1991. Reviving the game of checkers. In LEVY, D., AND BEAL, D., *Editors.*, *Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad*. Ellis Horwood.
- SCHRAUDOLPH, N. N.; DAYAN, P.; AND SEJNOWSKI, T. J. 1994. Temporal difference learning of position evaluation in the game of go. In COWAN, J.; TESAURO, G.; AND ALSPECTOR, J., *Editors.*, *Advances in Neural Information Processing 6*. San Francisco: Morgan Kaufmann.
- SMITH, S. J., AND NAU, D. S. 1993. Strategic planning for imperfect-information games. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*, 84-91. AAAI Press.
- SUTTON, R. S. 1984. *Temporal Credit Assignment in Reinforcement Learning*. Ph.D. Dissertation, Department of Computer and Information Science, University of Massachusetts.
- TADEPALLI, P. 1989. Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 694-700. Los Altos, CA: IJCAI.
- TESAURO, G. 1994. TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation* 6(2):215-219.
- TUNSTALL-PEDOE, W. 1991. Genetic Algorithms Optimizing Evaluation Functions. *ICCA-Journal* 14(3):119-128.
- WELLMAN, M. P., AND DOYLE, J. 1991. Preferential semantics for goals. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)* (1991), 698-703.
- WILKINS, D. E. 1982. Using knowledge to control tree search. *Artificial Intelligence* 18:1-51.
- WILLIAMS, T. G. 1972. Some studies in game playing with a digital computer. In SIKLOSSY, L., AND SIMON, H., *Editors.*, *Representation and Meaning*. Prentice-Hall.