Computational Intelligence, Volume 12, Number 1, 1996

A PLANNING APPROACH TO DECLARER PLAY IN CONTRACT BRIDGE*

S. J. J. Smith

Computer Science Department, University of Maryland, College Park, MD 20742, USA, (301)-405-2717, sjsmith@cs.umd.edu

D. S. Nau

Computer Science Department, Institute for Systems Research, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA, (301)-405-2684, nau@cs.umd.edu

T. A. Throop

Great Game Products, 8804 Chalon Drive, Bethesda, MD 20817, USA, (301)-365-3297, bridgebaron@mcimail.com

Although game-tree search works well in perfect-information games, it is less suitable for imperfect-information games such as contract bridge. The lack of knowledge about the opponents' possible moves gives the game tree a very large branching factor, making it impossible to search a significant portion of this tree in a reasonable amount of time.

This paper describes our approach for overcoming this problem. We represent information about bridge in a task network that is extended to represent multi-agency and uncertainty. Our game-playing procedure uses this task network to generate game trees in which the set of alternative choices is determined not by the set of possible actions, but by the set of available tactical and strategic schemes.

We have tested this approach on declarer play in the game of bridge, in an implementation called *Tignum 2*. On 5000 randomly generated notrump deals, Tignum 2 beat the strongest commercially available program by 1394 to 1302, with 2304 ties. These results are statistically significant at the $\alpha = 0.05$ level. Tignum 2 searched an average of only 8745.6 moves per deal in an average time of only 27.5 seconds per deal on a Sun SPARCstation 10. Further enhancements to Tignum 2 are currently underway.

Key words: game, game tree, game-playing, planning, uncertainty, imperfect information, pruning, bridge.

1. INTRODUCTION

Although game-tree search works well in perfect-information games (such as chess (Levy and Newborn, 1982; Berliner *et al.*, 1990), checkers (Samuel, 1967; Schaeffer *et al.*, 1992), and othello (Lee and Mahajan, 1990)), it does not always work as well in other games. One example is the game of bridge. Bridge is an imperfect-information

*This material is based on work supported in part by an AT&T Ph.D. scholarship to Stephen J. J. Smith, Maryland Industrial Partnerships (MIPS) grant 501.15, Great Game Products, and the National Science Foundation under Grants NSF EEC 94-02384 and IRI-9306580. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, AT&T, Maryland Industrial Partnerships, or Great Game Products.

game, in which no player has complete knowledge about the state of the world, the possible actions, and their effects. Thus the branching factor of the game tree is very large. Since the bridge deal must be played in just a few minutes, a full game-tree search will not search a significant portion of this tree within the time available.

To address this problem, some researchers have tried making assumptions about the placement of the opponents' cards based on information from the bidding and prior play, and then searching the game trees resulting from these assumptions. However, such approaches have several limitations, as described in Section 2.

In this paper, we describe a different approach to this problem, based on the observation that bridge is a game of planning. For addressing various card-playing situations, the bridge literature describes a number of *tactical schemes*, or short-term card-playing tactics, such as finessing and ruffing; the bridge literature also describes a number of *strategic schemes*, or long-term card-playing tactics, such as crossruffing. It appears that there is a small number of such schemes for each bridge deal, and that each of them can be expressed relatively simply. To play bridge, many humans use these schemes to create plans. They then follow those plans for some number of tricks, replanning when appropriate.

We have taken advantage of the planning nature of bridge, by adapting and extending some ideas from task-network planning. To represent the tactical and strategic schemes of card-playing in bridge, we use *multi-agent methods*—structures similar to the "action schemas" or "methods" used in hierarchical single-agent planning systems such as Nonlin (Tate, 1976; Tate, 1977), NOAH (Sacerdoti, 1974; Sacerdoti, 1975; Sacerdoti, 1977), O-Plan (Currie and Tate, 1985), and SIPE (Wilkins, 1984), but modified to represent multi-agency and uncertainty.

To generate game trees, we use a procedure similar to task decomposition. The methods that perform our tasks correspond to the various tactical and strategic schemes for playing the game of bridge. We then build up a game tree whose branches represent moves that are generated by these methods. This approach produces a game tree in which the number of branches from each state is determined not by the number of actions that an agent can perform, but instead by the number of different tactical and strategic schemes that the agent can employ. If at each node of the tree, the number of applicable schemes is smaller than the number of possible actions, this will result in a smaller branching factor, and a much smaller search tree.

To test this approach, we have developed an implementation called *Tignum 2*.[1] On 5000 randomly generated notrump deals, Tignum 2 beat the strongest commercially available program by 1394 to 1302, with 2304 ties. These results are statistically significant at the $\alpha = 0.05$ level.

2. RELATED WORK

Stanier (1975) did some of the first work on bridge; while his bidding program was primitive, his ideas about play still offer insight today. Quinlan (1979) wrote a knowledge-based system for reasoning about high cards, but it never developed into an algorithm for play. Berlin (1985) proposed an approach to play of the hand at

[1] Tignum 2 is a follow-up to a prototype program called *Tignum* that was described in (Smith and Nau, 1993). Tignum demonstrated that we could generate small game trees and still do correct play within the limited set of situations that its methods addressed, but its structure did not allow us to add methods to generalize its capabilities. Tignum 2 overcomes these limitations.

bridge that is similar to ours; sadly, he never had a chance to develop the approach (his paper was published posthumously). Some of the work on bridge has focused on bidding (Lindelof, 1983; Gamback *et al.*, 1990; Gamback *et al.*, 1993).

There are no really good computer programs for card-playing in bridge, especially in comparison to the success of computer programs for chess, checkers, and othello; most computer bridge programs can be beaten by a reasonably advanced novice. Sterling and Nygate (1990) wrote a rule-based program for recognizing and executing squeeze plays, but squeeze opportunities in bridge are rare. Recently, Frank and others (1992) have proposed a proof-planning approach, but thus far, they have only described the results of applying this approach to planning the play of a single suit. Khemani (1994) has investigated a case-based planning approach to notrump declarer play, but hasn't described the speed and skill of the program in actual competition. The approaches used in current commercial programs are based almost exclusively on domain-specific techniques, as described below.

One approach is to make assumptions about the placement of the opponents' cards based on information from the bidding and prior play, and then search the game tree resulting from these assumptions. This approach was taken in *Alpha Bridge* program (Lopatin, 1992), with a 20-ply (5-trick) search. However, this approach didn't work very well: at the 1992 Computer Olympiad, Alpha Bridge placed last.

Better-quality play can be achieved by generating several random hypotheses for what hands the opponents might have, and doing a full game-tree search for each hypothesized hand, as is done in Great Game Products' *Bridge Baron* program. However, this approach is feasible only late in the game, after most of the tricks have been played, because otherwise the game tree is too large to search to any significant depth within the time available.

Some work has been done on extending game-tree search to address uncertainty, including Horacek's work on chess (Horacek, 1990), and Ballard's work on backgammon (Ballard, 1983). However, these works do not address the kind of uncertainty that we discussed in the introduction, and thus it does not appear to us that these approaches would be sufficient to accomplish our objectives.

Wilkins (1980; 1982) uses "knowledge sources" to generate and analyze chess moves for both the player and the opponent. These knowledge sources have a similar intent to the multi-agent methods that we describe in this paper—but there are two significant differences. First, because chess is a perfect-information game, Wilkins's work does not address uncertainty and incomplete information, which must be addressed for bridge play. Second, Wilkins's work was directed at specific kinds of chess problems, rather than the problem of playing entire games of chess; in contrast, we have developed a program for playing entire deals of bridge.

Our work on hierarchical planning draws on (Tate, 1976; Tate, 1977; Sacerdoti, 1974; Sacerdoti, 1975; Sacerdoti, 1977). In addition, some of our definitions were motivated by (Erol *et al.*, 1993; Erol *et al.*, 1995).

3. PROBLEM CHARACTERISTICS

Bridge is a card game that is played by four players called North, East, South, and West, using a standard 52-card playing deck. North and South are partners against East and West. A Bridge game consists of a series of *deals*. In each deal, the cards are distributed evenly among the four players; thus, each player has a *hand* of thirteen cards. Then, there is an "auction" followed by thirteen "tricks" of play.

The purpose of the *auction* is to decide who gets to declare what the trump suit is, and how many tricks this *declarer* needs to take. The auction consists of a number of *calls* which are made by the players one at a time, starting with the dealer and progressing clockwise. A call is either a bid to take a certain number of tricks with a certain "trump" suit, or a Pass, Double, or Redouble.

Unless the auction began with four Passes, the play begins. The *contract* is the last bid that was made, plus any doubling or redoubling. The *trump* suit is the suit of the last bid that was made (or no suit if the last bid was a No Trump bid.) The *declarer* is the player who made the last bid (not call) in the auction—unless his partner made the first bid of the same suit for that partnership in the auction, in which case this partner becomes declarer. The player on declarer's left plays a card, the *opening lead.* Then *dummy*, or declarer's partner, exposes his hand for the declarer to see and play and for the *defenders*, or opponents, to see.

After everyone plays a card, the *trick* is over. The winner of a trick leads to the next trick. A trick is won by the player who plays the highest card in the trump suit, if any; otherwise, the trick is won by the player who plays the highest card in the same suit as the card that was *led* (played first in a trick.) Each player must play a card in the led suit whenever possible. (If the led suit was not trump, but a player plays a trump card, it is called a *ruff.*) After all thirteen tricks have been played, scoring occurs, based on whether declarer took all the tricks contracted for or not, how many extra tricks or how much shortfall there was, and whether the contract was doubled or redoubled.

In this paper, we consider the problem of declarer play at bridge. Our player controls two agents, declarer and dummy. Two other players control two other agents, the defenders. The auction is over and the contract has been fixed. The opening lead has been made and the dummy is visible. The hands held by the two agents controlled by our player are in full view of our agent at all times; the other two hands are not, hence the imperfect information.

Bridge has the following characteristics that are necessary for our approach:

1. Only one player may move at at time.
2. In general, no player has perfect information about the current state S. However, each player has enough information to determine whose turn it is to move.
3. A player may control more than one agent in the game (as in bridge, in which the declarer controls two hands rather than one). If a player is in control of the agent A whose turn it is to move, then the player knows what moves A can make.
4. If a player is not in control of the agent A whose turn it is to move, then the player does not necessarily know what moves A can make. However, in this case the player does know the set of possible moves A *might* be able to make; that is, the player knows a finite set of moves M such that every move that A can make is a member of M.

Our approach is applicable to any domain with these characteristics. Modifications of our approach may be possible if some of these characteristics are missing.

4. PROBLEM REPRESENTATION

Abstractly, we will consider the current state S (or any other state) to be a collection of *ground atoms* (that is, completely instantiated predicates) of some function-free first-order language $\mathcal{L}$ that is generated by finitely many constant symbols and predicate symbols. We do not care whether this is how S would actually be represented in an implementation of a game-playing program.

Among other things, S will contain information about who the players are, and whose turn it is to move. To represent this information, we will consider S to include a ground atom Agent(x) for each player x, and a ground atom Turn(y) for the player y whose turn it is to move. For example, in the game of bridge, S would include the ground atoms Agent(North), Agent(South), Agent(East), and Agent(West). If it were South's turn to move, then S would include the ground atom Turn(South).

We will be considering S from the point of view of a particular player $\mathcal{P}$ (who may be a person or a computer system). One or more of the players will be under $\mathcal{P}$'s control; these players are called the *controlled* agents (or sometimes "*our*" agents). The other players are the *uncontrolled* agents, or our *opponents*. For each controlled agent x, we will consider S to include a ground atom Control(x). For example, in bridge, suppose $\mathcal{P}$ is South. Then if South is the declarer, S will contain the atoms Control(South) and Control(North).

Because $\mathcal{P}$ is playing an imperfect-information game, $\mathcal{P}$ will be certain about some the ground atoms of S, and uncertain about others. To represent the information about which $\mathcal{P}$ is certain, we use a set of ground literals I_S called $\mathcal{P}$'s *state information set* (we will write I rather than I_S when the context is clear). Each positive literal in I_S represents something that $\mathcal{P}$ knows to be true about S, and each negative literal in I_S represents something that $\mathcal{P}$ knows to be false about S. Because we require that $\mathcal{P}$ knows whose turn it is to move, this means that I_S will include a ground atom Turn(y) for the agent y whose turn it is to move, and ground atoms ¬ Turn(z) for each of the other agents. For example, in bridge, suppose that $\mathcal{P}$ is South, South is declarer, it is South's turn to move, and South has the 6♣ but not the 7♣. Then I_S would contain the following literals (among others):

Control(North),	¬ Control(East),	Control(South),	¬ Control(West),
¬ Turn(North),	¬ Turn(East),	Turn(South),	¬ Turn(West),
Has(South,♣,6),	¬ Has(South,♣,7)		

Unless South somehow finds out whether West has the 7♣, I_S would contain neither Has(West,♣,7) nor ¬ Has(West,♣,7).

In practice, $\mathcal{P}$ will know I but not S. Given a state information set I, a state S is *consistent* with I if every literal in I is true in S. I^* is the set of all states consistent with I. $\mathcal{P}$ might have reason to believe that some states in I^* are more likely than others. For example, in bridge, information from the bidding or from prior play often gives clues to the location of key cards. To represent this, we define $\mathcal{P}$'s *belief function* to be a probability function $p : I^* \rightarrow [0,1]$, where $[0,1] = \{x : 0 \leq x \leq 1\}$.

To represent the possible actions of the players, we use operators somewhat similar to those used in STRIPS (Fikes and Nilsson, 1971). More specifically, if $X_1, X_2, \ldots, X_n$ are variable symbols, then a *primitive operator* $O(X_1; X_2, \ldots, X_n)$ is a triple (Pre(O),Add(O),Del(O)), where Pre(O), Add(O), and Del(O) are as follows:[2]

1. Pre(O), the *precondition formula*, is a formula in $\mathcal{L}$ whose variables are all from $\{X_1, \ldots, X_n\}$. Pre(O) must always begin with "Agent(X_1) ∧ Turn(X_1) ∧ …".

[2]The semicolon separates X_1 from the rest of the arguments because X_1 is the agent who uses the operator when it is X_1's turn to move.

2. Add(O) and Del(O) are both finite sets of atoms (possibly non-ground) whose variables are all from $\{X_1, \ldots, X_n\}$. Add(O) is called the *add list* of O, and Del(O) is called the *delete* list of O.

For example, in bridge, one operator might be PlayCard($P; S, R$), where the variable P represents the player (North, East, South or West), S represents the suit played ($\clubsuit$, $\diamondsuit$, $\heartsuit$, or $\spadesuit$), and R represents the rank (2, 3, ..., 9, T, J, Q, K, or A). Pre(PlayCard) would contain conditions to ensure that player P has the card of suit S and rank R. Add(PlayCard) and Del(PlayCard) would contain atoms to express the playing of the card, the removal of the card from the player's hand, and possibly any trick which may be won by the play of the card.

We define applicability in the usual way: if $O(a_1; a_2, a_3, \ldots, a_n)$ is an instantiation of a primitive operator O, then $O(a_1; a_2, a_3, \ldots, a_n)$ is *applicable* in a state S if Pre($O(a_1; a_2, a_3, \ldots, a_n)$) is true in S. If $O(a_1; a_2, a_3, \ldots, a_n)$ is applicable in some state $S_a \in I^*$, and if Control(a_1) holds, then we require that the instantiation be applicable in **all** states $S \in I^*$. This will guarantee that, as required, if $\mathcal{P}$ is in control of the agent a_1 whose turn it is to move, then $\mathcal{P}$ will have enough additional information to determine which moves a_1 can make. In bridge, for example, this means that if $\mathcal{P}$ has control of South, and it is South's turn, then $\mathcal{P}$ knows what cards South can play.

We let $\mathcal{S}$ be the set of all states, and $\mathcal{I}$ be the set of all state information sets.

An *objective function* is a partial function $f : \mathcal{S} \to [0, 1]$. Intuitively, $f(S)$ expresses the perceived benefit to $\mathcal{P}$ of the state S; where $f(S)$ is undefined, this means that S's perceived benefit is not known. In bridge, for states representing the end of the hand, f might give the score for the participant's side, based on the number of tricks taken. For other states, f might well be undefined.

Game-playing programs for perfect-information games make use of a *static evaluation function*, which is a total function $e : \mathcal{S} \to [0, 1]$ such that if S is a state and $f(S)$ is defined, then $e(S) = f(S)$. In imperfect-information games, it is difficult to use $e(S)$ directly, because instead of knowing the state S, all $\mathcal{P}$ will know is the state information set I. Thus, we will instead use a *distributed evaluation function* $e^*(I) = \sum_{S \in I^*} p(S)e(S)$.

Intuitively, $e^*(I)$ expresses the estimated benefit to $\mathcal{P}$ of the set of states consistent with the state information set I. Our game-playing procedure will use e^* only when it is unable to proceed past a state.[3]

A *game* is a pair $G = (\mathcal{L}, \mathcal{O})$ where $\mathcal{L}$ is the planning language and $\mathcal{O}$ is a finite set of operators. Given a game G, a *problem* in G is a quadruple $P = (I_{S_1}, p, f, e)$, where I_{S_1} is the state information about an *initial* state S_1, p is a belief function, f is an objective function and e is a static evaluation function. For example, if G is the game of bridge, then the problem P would be a particular hand, from a particular player's point of view. All the information required to compute e^* is expressed in e and p, thus we need not include e^* in our definition of P.

[3]However, we can imagine that in time-sensitive situations, one might want to modify our procedure so that it sometimes uses e^* on nodes that it can proceed past, just as chess-playing computer programs use a static evaluation function rather than searching to the end of the game.

5. MULTI-AGENCY IN TASK NETWORKS

Given a game $G = (\mathcal{L}, \mathcal{O})$ and a problem $P = (I_{S_1}, p, f, e)$ in G, the set of all plausible initial states for P is the set $I^*_{S_1}$ of all states consistent with I_{S_1}. The set of all states that might plausibly occur as a result of the various players' moves is the set $\mathcal{T}_P$ of all states that can be reached via sequences of legal moves from states in $I{*}_{S_1}$. In general, $\mathcal{T}_P$ is a proper subset of $\mathcal{S}$, but $\mathcal{T}_P$ is usually quite large.

Much of the difficulty of game-playing is due to the large number of states in $\mathcal{T}_P$ that must be examined and discarded. In order to avoid generating and examining every state in $\mathcal{T}_P$ (as would be done by a brute-force search procedure), we will attempt to generate only those states that appear to fit into coherent tactical and strategic schemes such as finessing, ruffing, and crossruffing.

Our approach is an adaptation of hierarchical task networks (HTNs). We use two kinds of *multi-agent methods* to build a task network: operator methods and decomposable methods. An *operator method* is a triple $M = (T, P, E)$, where

1. T is a *task*. This may either be the expression 'NIL' or a syntactic expression of the form $N(X_1; X_2, X_3, \ldots, X_n)$ where N is a symbol called the *name* of the task, and each X_i is a variable of $\mathcal{L}$. Note that n may be 0, in which case the task has no variables. If $n > 1$, then a semicolon separates X_1 from the rest of the arguments because X_1 is the agent whose turn it is to move when the method is used. If, when an operator method M is used, X_1 is an opponent, then M must associate both a critic and a weighting function with the branch that it creates in the game tree, as described in Section 6.3.
2. P is the *precondition formula* of M. This may be any formula in $\mathcal{L}$ that is sufficient to ensure the truth of the precondition formula of the operator instantiation $O(t_1, t_2, \ldots, t_m)$ described below.
3. E is a syntactic expression $O(t_1, t_2, \ldots, t_m)$, where O is an operator in $\mathcal{O}$, and $t_1, t_2, \ldots, t_m$ are terms of $\mathcal{L}$.

A *decomposable method* is a triple $M = (T, P, E)$, where

1. T is a *task*. This may either be the expression 'NIL' or a syntactic expression of the form $N(X_1; X_2, X_3, \ldots, X_n)$ where N is a symbol called the *name* of the task, and each X_i is a variable of $\mathcal{L}$.
   Note that n may be 0, in which case the task has no variables. If $n > 1$, then a semicolon separates X_1 from the rest of the arguments because X_1 is the agent whose turn it is to move when the method is used.
2. P, the *precondition formula*, is any formula in $\mathcal{L}$ such that if $n > 0$ above, then P is sufficient to ensure the truth of the atom Agent(X_1).
3. E is a (possibly empty) tuple of tasks

$$(T_1(t_{1,1}, t_{1,2}, \ldots, t_{1,m_1}),\ T_2(t_{2,1}, t_{2,2}, \ldots, t_{2,m_2}), \cdots,\ T_k(t_{k,1}, t_{k,2}, \ldots, t_{k,m_k})).$$

   E, called the *expansion list*, lists the subtasks that make up this method.

For declarer and dummy, networks of methods are used to represent tactical schemes such as finessing and setting up suits. For defenders, networks of methods represent tactical schemes such as causing the finesse to fail or to succeed.

For example, consider Figure 1, in which the task Finesse$(P; S)$ represents the tactical scheme of finessing. In finessing, we try to win a trick with a high card by

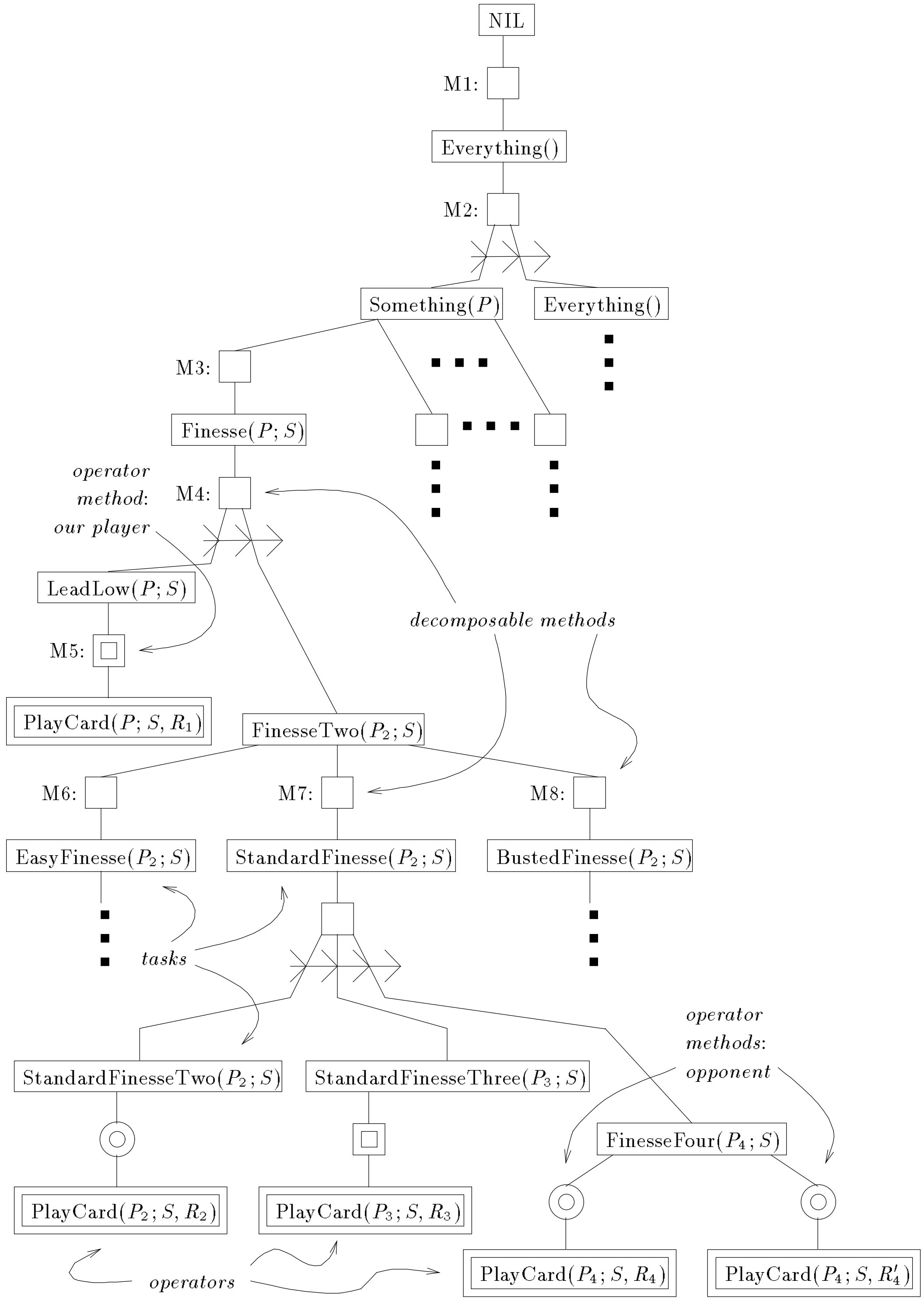

FIGURE 1. The part of the network of multi-agent methods for the game of bridge that addresses finessing.

TABLE 1. Some of the methods and operators used in Figure 1.

Decomposable methods:

T:	NIL
P:	NIL
E:	Everything()
T:	Everything()
P:	Agent(P)$\wedge$Turn(P)
E:	Something(P) then Everything()
T:	Something(P)
P:	Agent(P)$\wedge$OnLead(P)$\wedge$ (MACRO:TrumpDrawn(P)) $\wedge$ (MACRO:FinesseInSuit(P; S))
E:	Finesse(P; S)
T:	Finesse(P; S)
P:	Agent(P)$\wedge$Turn(P)$\wedge$Next(P, P_2)
E:	LeadLow(P; S) then FinesseTwo(P; S)
T:	FinesseTwo(P_2; S)
P:	Agent(P_2)$\wedge$Turn(P_2)$\wedge$ (MACRO:Singleton(P_2; S, R)) $\wedge$ (MACRO:KeyFinesseCard(S, R))
E:	EasyFinesse(P_2; S)
T:	FinesseTwo(P_2; S)
P:	Agent(P_2)$\wedge$Turn(P_2)$\wedge$ (MACRO:VoidInSuit(P_2; S))
E:	BustedFinesse(P_2; S)
T:	FinesseTwo(P_2; S)
P:	Agent(P_2)$\wedge$Turn(P_2)$\wedge$ Has(P_2; S, R) $\wedge$ ($R \neq R_2$) $\wedge$ (MACRO:KeyFinesseCard(S, R_2))
E:	StandardFinesse(P_2; S)

Operator method:

T:	LeadLow(P; S)
P:	Agent(P)$\wedge$ (MACRO:Playable(P; S, R)) $\wedge$ (MACRO:LowestInSuit(P; S, R))
E:	PlayCard(P; S, R)

Operator:

Name:	PlayCard(P; S, R)
Preconditions:	Agent(P)$\wedge$ (MACRO:Playable(P; S, R))
Add list:	Played(P; S, R)$\wedge$ (MACRO:TurnChangeAdds(P; S, R))
Delete list:	Has(P; S, R)$\wedge$ (MACRO:TurnChangeDels(P; S, R))

playing after the opponent that has a higher card; see Figure 3 and Figure 4 for examples of a decision tree for a finesse. The task StandardFinesse(P_2; S) is the core of the tactical scheme of a finesse. The task FinesseFour(P_4; S) represents the defenders causing the finesse to fail or to succeed. "Two" in FinesseTwo(P_2; S) and StandardFinesseTwo(P_2; S) refers to the second card played to the trick; similarly "Three" and "Four". For the meaning of BustedFinesse(P_2; S) and EasyFinesse(P_2; S), see Section 6.2.

Some of the methods used in Figure 1 are shown in Table 1. The syntax

$$(\text{MACRO:MacroName}(X_1; X_2, X_3, \ldots, X_n))$$

means that MacroName($X_1; X_2, \ldots, X_n$) is not an atom of $\mathcal{L}$, but expands into a second-order formula of $\mathcal{L}$. For example, (MACRO:Playable($P; S, R$)) expands into

$$\text{Has}(P; S, R) \wedge \text{Turn}(P) \wedge (\text{OnLead}(P) \vee \text{SuitLed}(S) \vee (\forall R_1)(\neg \text{Has}(P; S, R_1))).$$

We present a detailed example using these methods in Section 6.2.

If an instantiation of a method (say $M(a_1; a_2, a_3, \ldots, a_n)$) is applicable in some state $S_a \in I^*$, and if Control(a_1) holds, then we require that the instantiation be applicable in **all** states $S \in I^*$. This will guarantee that if $\mathcal{P}$ is in control of the agent a_1 whose turn it is to move, then $\mathcal{P}$ will have enough additional information to determine which methods are applicable. In bridge, for example, this means that if $\mathcal{P}$ has control of South, and it is South's turn, then $\mathcal{P}$ knows what strategic and tactical schemes South can employ.

Let $\mathcal{M}$ be a set of operator methods and decomposable methods. Then $\mathcal{M}$ is a *network of multi-agent methods*. Figure 1 shows how methods link tasks together, making a "network".

In contrast to the task networks used in most task-network planners, our task networks are totally ordered; that is, the expansion lists in all the decomposable methods are in the order in which tasks must be completed. For example, nowhere do we allow a single decomposable method to generate two tasks such as "Set Up (Spade) Suit" and "Get a (Heart) Ruff" without specifying the order of the tasks.

We chose this total ordering because of the difficulty of reasoning with imperfect information. It is difficult enough to reason about the probable locations of the opponents' cards. If our task networks were partially ordered, then in many planning situations we wouldn't know what cards the opponents had already played. This would make reasoning about the probable locations of the opponents' cards nearly impossible; this reasoning is a more serious problem than the problems with uninstantiated variables that occur in perfect-information domains.

Some of the tasks used for bridge in our implementation are listed in Section 7.

6. GAME-PLAYING PROCEDURE

Our game-playing procedure constructs a decision tree, then evaluates the decision tree to produce a plan for playing some or all of the game. It then executes this plan either until the plan runs out, or until some opponent does something that the plan did not anticipate (at which point the procedure will re-plan). The details of the procedure are described in this section.

6.1. Constructing a Decision Tree

Given a network of multi-agent methods and a state information set I, our game-playing procedure constructs a *decision tree* rooted at I.[4] A decision tree resembles a game tree.[5] It contains two kinds of non-leaf nodes: *decision nodes*, representing

[4]We use the term "decision tree" as it is used in the decision theory literature (French, 1986; Feldman and Yakimovsky, 1974; Feldman and Sproull, 1977), to represent a structure similar to a game tree. We are **not** employing decision trees (also, and less ambiguously, referred to as *comparison trees* (Knuth, 1973)) as they are defined in the sorting literature (Cormen et al., 1990, p. 173). We apologize for the confusion, but it is inescapable.

[5]In the decision theory literature, what we call external-agent nodes are usually called *chance nodes*, because decision theorists usually assume that the external agent is random. What we call leaf nodes are usually called *consequence nodes*, because they represent the results of the paths taken to reach them.

the situations in which it is $\mathcal{P}$'s turn to move, and *external-agent nodes*, representing situations in which it is some external agent's turn to move. The tree's leaves are nodes where the procedure has no methods to apply, either because the game has ended, or because the methods simply don't tell the procedure what to do.

Each node of the decision tree T will contain a state information set and a sequence of 0 or more tasks to be solved. Our procedure for creating T is as follows:

> Let the root of T be a node containing the state information set I and no tasks. Do the following steps repeatedly:
>
> 1. Pick a leaf node u of T such that u is not the end of the game and we have never tried to expand u before. If no such node exists, then exit, returning T. Otherwise, let I_u be the state information set in u, and $\mathcal{U} = (U_1, U_2, \ldots, U_n)$ be the sequence of tasks in u.
> 2. For each instantiated method $\overline{M}$ that is applicable to u, let v be the node produced by applying $\overline{M}$ to u. Install v into T as a child of u. (Below, we define the terms 'instantiated method,' 'applicable,' and 'produced.')

Using the network shown in Figure 2, the procedure might generate a piece of a decision tree for finesse such as that shown in Figure 3. We will give an example of an operation of this procedure in Section 6.2.

An *instantiated method* is any ground instance $\overline{M}$ of some method M. Let u be a node whose state information set is I_u and whose task sequence is $\mathcal{U} = (U_1, U_2, \ldots, U_n)$. Then an instantiated method $\overline{M} = (\overline{T}, \overline{P}, \overline{E})$ is *applicable* to u if:

1. Either $\mathcal{U}$ is empty and $\overline{T}$ is NIL, or $\overline{T}$ matches U_1.
2. Some state S consistent with I_u satisfies $\overline{P}$, i.e., some state $S \in I_u^*$ satisfies $\overline{P}$.

If $\mathcal{P}$ is in control of the agent a_u whose turn it is to move at node u, and if one state S consistent with I_u satisfies $\overline{P}$, then all states S' consistent with I_u (i.e., all $S' \in I_u^*$) satisfy $\overline{P}$. We made this property a requirement of our multi-agent methods in Section 5. Because all states S' consistent with I_u satisfy $\overline{P}$, it follows that $\overline{P} \subseteq I_u$.

If $\mathcal{P}$ is **not** in control of the agent a_u whose turn it is to move at node u, then it is possible for one state S_1 consistent with I_u to satisfy $\overline{P}$, while some other state S_2 consistent with I_u does **not** satisfy $\overline{P}$. In this case, our procedure will need to assume that $\overline{P}$ holds, so that it can investigate what happens when some opponent makes a move using the instantiated method $\overline{M}$. Other instantiated methods $\overline{M}_1, \overline{M}_2, \ldots, \overline{M}_m$ will investigate what happens in states where $\overline{P}$ does not hold.

For example, in Figure 3, before investigating the move Play(North; ♠, 3), our procedure would need to make the assumption that North holds the 3♠. The procedure would investigate the other moves for West under different assumptions (say, that North holds only the Q♠, or that North holds no spades.)

If $\overline{M}$ is applicable to u, then applying $\overline{M}$ to u *produces* the node v whose state information set I_v and task sequence $\mathcal{V}$ are as follows:

- If $M = (T, P, E)$ is a decomposable method, then $I_v = I_u \cup \overline{P}$. Intuitively, I_v is I_u with all the conditions in $\overline{P}$ assumed. If $M = (T, P, E)$ is an operator method, then $I_v = [(I_u \cup \overline{P}) - \mathrm{Del}(\overline{E})] \cup \mathrm{Add}(\overline{E})$. Intuitively, I_v is I_u, with all the conditions in $\overline{P}$ assumed, and then all the conditions in $\mathrm{Del}(\overline{E})$ deleted, and all the conditions in $\mathrm{Add}(\overline{E})$ added.

Our players: East declarer,
West dummy
Opponents: Defenders,
South and North
Contract: East—3NT.
On lead: West at trick 3.

East:	West:	Out:
♠KJ74	♠A2	♠QT98653

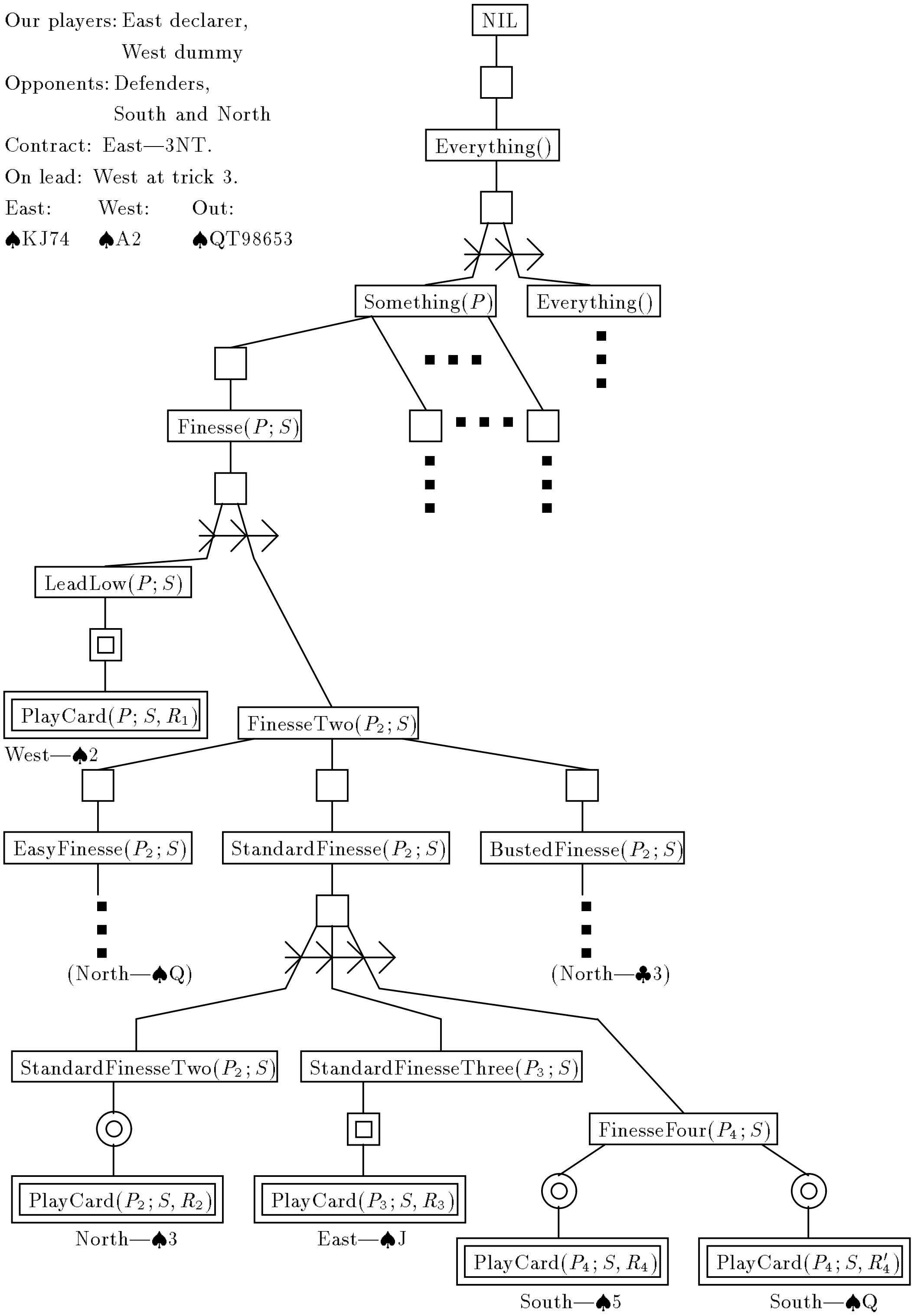

FIGURE 2. Instantiation of the finessing part of the network.

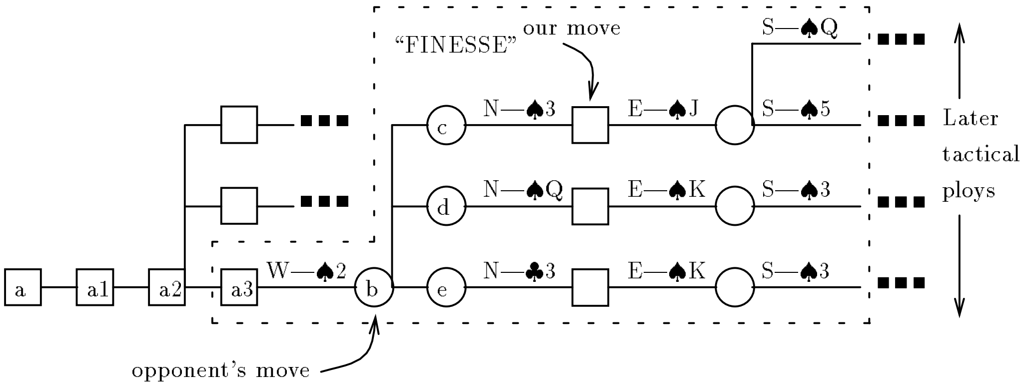

Figure 3. Decision tree generated on our example. For the sake of clarity, some nodes have been omitted.

- If $M = (T, P, E)$ is a decomposable method and $E = (V_1, V_2, \ldots, V_m)$, then $\mathcal{V} = (V_1, V_2, \ldots, V_m, U_2, U_3, \ldots, U_n)$. Intuitively, this corresponds to saying that the tasks $V_1, V_2, \ldots, V_m$ need to be solved *before* attempting to solve the tasks $U_2, U_3, \ldots, U_n$. If M is an operator method, then $\mathcal{V} = (U_2, U_3, \ldots, U_n)$.

Each solution tree in the decision tree represents an alternative plan. However, as seen in Figure 3, branches in our decision tree correspond to possible schemes, **not** to cards in the various hands. All opponents' moves are completely instantiated in our decision tree. We will see in Section 6.3 how a completely instantiated card can represent a possible scheme, and also what to do if the opponent makes a move that does not appear in our decision tree.

6.2. Example of Decision-Tree Construction

As an example, consider the situation in which "our" players (the players of which $\mathcal{P}$ is in control) are East, the declarer, and West, the dummy. The opponents, North and South, are the defenders. The contract is 3NT by East. West is on lead at trick 3 East has the ♠ KJ74, West has the ♠ A2, and the ♠ QT98653 are in North's and South's hands, although we do not know who holds which spades.

Initially, the decision tree consists of the root node a, containing the state information set I_a and the task sequence $\mathcal{V}_a$. I_a contains the following literals:

¬ Control(North),	Control(East),	¬ Control(South),	Control(West),
¬ Turn(North),	¬ Turn(East),	¬ Turn(South),	Turn(West),
Has(East,♠,K),	Has(East,♠,J)	Has(East,♠,7)	Has(East,♠,4)
Has(West,♠,A),	Has(West,♠,2)	¬ Has(West,♠,Q)	¬ Has(East,♠,Q)
¬ Has(West,♠,T)	¬ Has(East,♠,T)	*(and various other literals)*	

$\mathcal{V}_a = ()$, the empty sequence. Refer to Table 1, Figure 1, Figure 2, and Figure 3.

The only instantiated method applicable to a is M1 in Figure 1. Thus, node $a1$ becomes the child of a. $\mathcal{V}_{a1}$ = (Everything()). $I_{a1} = I_a$, because M1 has no preconditions.

The only instantiated method applicable to $a1$ is M2 in Figure 1, with the instantiation P = West. Thus, node $a2$ becomes the child of $a1$. $\mathcal{V}_{a2}$ = (Something(West), Everything()). $I_{a2} = I_{a1}$, because M2's only preconditions, as seen in Table 1, are

Agent(P)$\wedge$Turn(P). Indeed, if it is our player's turn to move at a node u, then for any child v of u produced by a decomposable method, $I_v = I_u$, because $I_v = I_u \cup \overline{P}$ by the definition of "produces"; and $\overline{P} \subseteq I_u$, as we saw when we considered applicability of $\overline{P}$ to u in Section 6.1.

The instantiated method applicable to $a2$ which we will consider is M3 in Figure 1, with the instantiations P = West, S = ♠. Thus, node $a3$ becomes the child of $a2$. $\mathcal{V}_{a3}$ = (Finesse(West; ♠), Everything()). $I_{a3} = I_{a2}$. Other instantiated methods are applicable to $a2$, such as the method for cashing all the high cards in East's and West's hands. These methods are indicated by the dots in Figures 1–3.

The only instantiated method applicable to $a3$ is M4 in Figure 1, with the instantiations P = West, P_2 = North, S = ♠. Thus, node $a4$ becomes the child of $a3$. $\mathcal{V}_{a4}$ = (LeadLow(West; ♠), FinesseTwo(North; ♠), Everything()). $I_{a4} = I_{a3}$.

The only instantiated method applicable to $a4$ is M5 Figure 1, with the instantiations P = West, S = ♠, R = 2. Thus, node b becomes the child of $a4$. $\mathcal{V}_b$ = (FinesseTwo(North; ♠), Everything()).

$$\begin{aligned} I_b &= [(I_{a4} \cup \overline{P}) - \mathrm{Del}(\overline{E})] \cup \mathrm{Add}(\overline{E}) \\ &= [I_{a4} - \mathrm{Del}(\overline{E})] \cup \mathrm{Add}(\overline{E}) \\ &= [I_{a4} - \{\mathrm{Has(West,}\spadesuit, 2), \mathrm{Turn(West)}\}] \cup \{\mathrm{Played(West,}\spadesuit, 2), \mathrm{Turn(North)}\}. \end{aligned}$$

EasyFinesse(P_2; S) refers to a finesse in which the first opponent plays the card that we are trying to trap. In the current situation, if North plays the Q♠, East can play the K♠, and now the J♠ is a card that is sure to win a trick. BustedFinesse(P_2; S) refers to a finesse in which the first opponent plays a card in a suit other than the suit led, in which case it is clear that the first opponent does not have the card that we are trying to trap. In the current situation, if North plays the 3♣, West would have to play the K♠, because South must have the Q♠ and the finesse must fail.

Three instantiated methods are applicable to b: M6, M7, and M8. They are decomposable methods whose tasks are FinesseTwo(P_2; S) with the instantiations P_2 = North, S = ♠:

- M6's expansion list is (EasyFinesse(P_2; S)). M6 also has the instantiation R = Q. For M6, node d becomes b's child. $\mathcal{V}_d$ = (EasyFinesse(North; ♠), Everything()). M6's preconditions include (MACRO:HasSingletonInSuit(P_2; S, R)) $\wedge$ (MACRO: KeyFinesseCard(S, R)), and thus $I_d = I_b \cup$ {Has(North, ♠, Q), ¬Has(North, ♠, 3), ¬Has(North,♠, 5), ..., ¬Has(North,♠, T)}.
- M7's expansion list is (BustedFinesse(P_2; S)). For M7, node e becomes b's child. $\mathcal{V}_e$ = (BustedFinesse(North; ♠), Everything()). M7's preconditions include (MACRO: VoidInSuit(P_2; S)), and thus $I_e = I_b \cup$ {¬Has(North,♠, 3), ¬Has(North,♠, 5), ..., ¬Has(North,♠, Q)}.
- M8's expansion list is (StandardFinesse(P_2; S)). M8 also has the instantiations R =3, R_2 =Q. For M8, node c becomes b's child. $\mathcal{V}_c$ = (StandardFinesse(North; ♠), Everything()). M8's preconditions include Has(P_2; S, R), and thus $I_c = I_b \cup$ {Has(North,♠, 3)}.

The rest of the decision tree in Figure 3 is generated in a similar manner.

6.3. Decision-Tree Evaluation and Plan Execution

Given a decision tree T, $\mathcal{P}$ will want to evaluate this tree by assigning a utility value to each node of the tree. As we generate the decision tree T (as described in

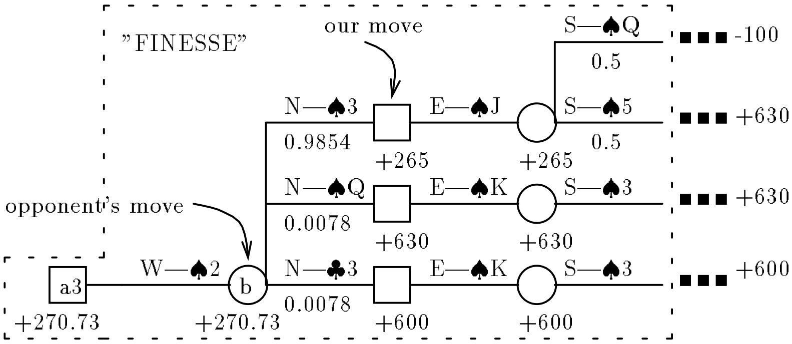

Figure 4. Evaluation of the decision tree generated on our example. For the sake of clarity, many nodes have been omitted.

the previous section), it is possible to evaluate it at the same time. However, for the sake of clarity, this section describes the evaluation of T as if the entire tree T had already been generated. Refer to Figure 4.

In perfect-information games, the usual approach is to use the minimax procedure, which computes the maximum at nodes where it is $\mathcal{P}$'s move, and the minimum at nodes where it is the opponent's move. In the decision theory literature, this procedure is referred to as the Wald maximin-return decision criterion. This decision criterion is less appropriate for imperfect-information games: because we do not know what moves the opponent is capable of making, it makes less sense to assume that the opponent will always make the move that is worst for us. Thus, a more complicated criterion which considers the belief function is to be preferred, such as the weighted-average-of-utilities criterion outlined below.

Let u be an external-agent node whose children are $u_1, \ldots, u_n$. For each u_i, let I_i be the state information set contained in u_i. Suppose we have already computed a utility value $v_i \in [0, 1]$ for each u_i. Then we define a *external-agent criterion* to be an algorithm C that returns a utility value $v = C(u, u_1, \ldots, u_n)$ for the node u.[6]

Many external-agent criteria can be used, taking ideas from the pure decision criteria (such as Hurwicz's optimism-pessimism index, Savage's minimax regret, and Wald's maximin return). Some will make use of the belief function p, others will not. In bridge, we generally use an external-agent criterion that gives a weighted average of the utility values v_i resulting from the best move the opponents could make in all the states consistent with the state information set I. The weights are computed using p by functions associated with each operator method that yields opponents' moves, as described in Section 5. A weight represents the probability that the opponent makes a given move, and the functions compute the weights on the basis of information from the prior bidding and play.

Given a decision tree, a external-agent criterion for each uncontrolled agent, an objective function, and a belief function, we *evaluate* the decision tree as follows:

1. The utility value of a leaf node u is the value of $e^*(I)$, where I is the state

[6]This definition of external-agent criterion is somewhat different from the usual definition of decision criterion in decision theory (French, 1986, p. 28), which essentially defines decision criteria on a two-level structure of decision nodes and chance nodes, without the belief function p. However, we believe that our definition, while in theory is not always as powerful, is in practice strong enough to implement most decision criteria we would want in most domains.

information set associated with u. Recall that if f, the objective function, is defined at a state S, then $e(S) = f(S)$. Thus, if we have reached a the end of the game, then the objective function is used, as desired.
2. The utility value of an external-agent node u is the value $C(u, u_1, u_2, \ldots, u_n)$, where $u_1, u_2, \ldots, u_n$ are the children of u.
3. The utility value of a decision node u is the maximum of the utility values of its children.

Although this evaluation may be computed recursively as defined, there may also be more efficient computations (for example, if $C(u, u_1, u_2, \ldots, u_n)$ were the minimum of the utility values of $u_1, u_2, \ldots, u_n$, then we could use alpha-beta pruning as a more efficient computation of minimax).

Once the decision tree is solved, a plan (a *policy* in the decision theory literature) has been created; $\mathcal{P}$ will, at the state information set associated with any decision node, simply choose the method that leads to the node with highest utility value.

This plan can be thought of as a contingency plan. After each of our moves, the tree includes nodes that match most of the opponent's possible responses; the subtree below each such node represents the plan we will use to respond to the opponent's move. Each node where it is the opponent's move contains a "planned card" (the card we think the opponent is most likely to play), as well as a critic function (as mentioned in Section 5). The purpose of this critic function is to tell us if certain other cards match by virtue of being "equivalent" to the planned card. For example, the critic associated with the method for StandardFinesseTwo(P_2; S) in Figure 1 and Figure 2 would accept any of 3♠, 5♠, 6♠, 8♠, 9♠, or T♠ as equivalent to 3♠.

$\mathcal{P}$ follows the plan as far as possible. If the plan takes $\mathcal{P}$ to the end of the game, then the problem is solved. If the plan should terminate before the end of the game—which may occur either because an external agent performs an action which is not present in the plan, or because the plan has reached a previously unexpanded node—then $\mathcal{P}$ simply re-plans, starting at the node where the plan ends.

7. IMPLEMENTATION AND TESTING

To test our approach, we have done a full implementation of a program to perform declarer play at bridge, called *Tignum 2*. For now, Tignum 2 has concentrated on playing notrump contracts, which comprise about 28.6% of all bridge contracts.[7]

The tasks used in Tignum 2 are shown in Tables 2 and 3. *Second hand* refers to the player who plays the second card to a given trick; similarly *third hand* and *fourth hand*. A *sequence* is a set of cards, such as the K♠ and Q♠, such that one of the cards must eventually win a trick; if the K♠ loses to the opponents' A♠, the Q♠ is the highest remaining spade and can then win a trick. A *marked finesse* is a finesse that is known to be successful, for example, when fourth hand is known to be void in the suit. A *winner* is a card, such as the Ace, that is sure to win a trick. To *follow suit* is to play a card in the same suit as the card led to the current trick. To *cross* is lead to a winner in partner's hand, so that partner is on lead. To *cash* is to play a winner for the purposes of winning a trick.

[7]Our 95% confidence interval for the frequency of notrump contracts bid by the Bridge Baron is [27.9%, 29.3%].

TABLE 2. Tasks used in Tignum 2, part one.

Task name	**Task description**
Everything	Play the whole hand
Something	Use a particular strategic or tactical scheme
CashOut	Cash all the winners in declarer's and dummy's hands
CashHand	Cash all the winners in a particular hand
CashWinner	Cash a winner in declarer's hand in a particular suit
CashDummyWinner	Cash a winner in dummy
CashFollowed	Cash a winner in declarer's hand; have the opponents follow suit
CashDummyFollowed	Cash a winner in dummy; have the opponents follow suit
ConsiderCross	Think about crossing from declarer to dummy, or vice versa
SimpleCross	Cross from declarer to dummy, or vice versa
LeadLow	Lead a low card
Finesse	Take a finesse
StandardFinesse	Play in a standard finesse
StandardFinesseThree	Play third hand in a standard finesse
MarkedStandardFinesse	Play in a marked finesse
PlaySetCard	Play a card in a sequence
PlaySetFollowed	Play a card in a sequence; have the opponents follow suit
ConsiderLength	Think about setting up a low card to win a trick
LengthWinner	Set up a low card to win a trick
WaitForControl	Wait to get back on lead before continuing
Follow	Follow to a trick that the opponents led
FollowVoid	Follow to a trick when void in the suit led
FollowSuit	Follow to a trick when holding cards in the suit led
ConsiderHard	Think about setting up a low card by losing some tricks
HardLengthWinner	Set up low card by losing some tricks in the suit
HardCash	Cash a winner before losing some tricks
HardWinner	Cash a winner in declarer's hand before losing some tricks
HardDummyWinner	Cash a winner in dummy before losing some tricks
HardFollowed	Cash a winner before losing some tricks; have the opponents follow suit
HardDummyFollowed	Cash a winner in dummy before losing some tricks; have the opponents follow suit
HardLose	Lose a trick to set up a low card
LoseTrick	Lose a trick
LoseFollowed	Lose a trick; have the opponents follow suit
HardDuck	Intentionally lose a trick to preserve high cards for crossing
RuffIn	Ruff a trick when void in the suit led
DiscardLow	Discard a low card when void in the suit led
DiscardDuringCash	Discard a low card when void in the suit led while cashing out
FreeFinesse	Play a card in second hand that might win a trick
ThirdHandLow	Play a low card when third hand
ThirdHandHigh	Play a high card when third hand
ThirdHandEquiv	Play a card in third hand equivalent to a particular card
ThirdHandCover	Play a card in third hand that covers the card that an opponent played in second hand
ThirdHandRuff	Play a trump card in third hand when void in the suit led
ThirteenthTrick	Play the thirteenth trick

TABLE 3. Tasks used in Tignum 2, part two.

Task name	**Task description**
DrawTrump	Play cards in the trump suit to remove cards in the trump suit from the opponents' hands
ConsiderRuff	Think about setting up a low card opposite a void to get a ruff
SetUpRuff	Set up a low card opposite a void to get a ruff
RuffLow	Ruff with a low card
RuffRuffed	Ruff a trick, forcing the opponents to ruff with a higher card
RuffOut	Cash all the winners and take all the ruffs in declarer's and dummy's hands
CashTrumps	Cash all the winners in the trump suit
ForceThemRuff	Force the opponents to ruff by cashing winners
CashRuffed	Cash a winner in declarer's hand, forcing the opponents to ruff
CashDummyRuffed	Cash a winner in dummy's hand, forcing the opponents to ruff
FinesseTwo	Opponents: play in second hand during a finesse
StandardFinesseTwo	Opponents: play in second hand during a standard finesse
EasyFinesse	Opponents: play a card being finessed against
BustedFinesse	Opponents: show void in suit during a finesse
FinesseFour	Opponents: play in fourth hand during a finesse
FreeFinesseThree	Opponents: play in third hand during a free finesse
MarkedFinesseTwo	Opponents: play in second hand during a marked finesse
MarkedFinesseFour	Opponents: play in fourth hand during a marked finesse
ThemNotWinner	Opponents: play a card that does not win the trick
ThemAny	Opponents: play any card
ThemLead	Opponents: lead to a trick
ThemVoid	Opponents: follow to a trick when void in the suit led
ThemSuit	Opponents: follow to a trick when holding cards in the suit led
ThemWin	Opponents: win a trick
ThemAnyLow	Opponents: play any low card
ThemSplit	Opponents: follow to a trick, possibly void in the suit led, possibly not
ThemSplitLess	Opponents: follow to a trick, possibly void in the suit led, possibly holding cards that do not beat a particular card
ThemSuitLess	Opponents: follow to a trick when holding cards in the suit led that do not beat a particular card
ThemRuff	Opponents: ruff a trick
ThemOverruff	Opponents: ruff a trick with a higher card
ThemFinesse	Opponents: take a finesse against declarer
Stop	Stop planning

We wanted to test the play of Tignum 2 against the Bridge Baron (BB), a commercially available bridge program. Like most such programs, BB is primarily rule-based (without rule chaining). It is probably safe to say that the Bridge Baron is the best program in the world for declarer play at contract bridge.[8]

[8]The Bridge Baron (which was formerly known as the Micro-Bridge Companion) won the 1990 and 1991 Computer Olympiads, lost the 1992 Computer Olympiad on a tiebreaker, won the 1993 Computer Bridge world championship, and won the 1994 and 1995 Computer Bridge competitions sponsored by the American Contract Bridge League (ACBL). In their review of seven commercially available programs (Manley, 1993), the ACBL rated the Bridge Baron to be the best of the seven, and the skill of the Bridge Baron to be the best of the five that do declarer play without "peeking" at the opponents' cards.

Deal:

North
♠ KJ852
♡ T
◇ 43
♣ QJ952

West
♠ AT73
♡ KJ532
◇ T
♣ A63

East
♠ Q964
♡ Q8
◇ AK62
♣ 874

South
♠ —
♡ A9764
◇ QJ9875
♣ KT

Bidding:

W	N	E	S
			Pass
1♡	Pass	1NT	Pass
2♣	Pass	2NT	Pass
Pass	Pass		

Contract: East—2NT
Lead: South—Q ◇
Vulnerable: All

Figure 5. One of the deals on which the Tignum 2 team beat the BB team.

Table 4. Results of competition between two teams described in the text.

Result	**Deals**	**Wins**	**Losses**	**Ties**	**Margin of victory**
Tignum 2 over BB	5000	1394	1302	2304	92

The best method of comparing bridge competitors is *duplicate bridge*, which eliminates the possibility of any competitor gaining a gross advantage simply by the luck of the deal. In duplicate bridge, each deal is played twice. The first time (played at the "first table"), two members of Team A sit North and South, and two members of Team B sit East and West. The second time (played at the "second table"), two other members of Team A sit East and West, and two other members of Team B sit North and South. (No one team member plays the same deal twice, to ensure that no one has foreknowledge of the unknowns in the deal.)

In order to compare the declarer play of Tignum 2 against the declarer play of the Baron, we formed the following two teams:

- the *BB* team: BB for declarer play, and BB for bidding and defender play.
- the *Tignum 2* team: Tignum 2 for declarer play, and BB for bidding and defender play (because Tignum 2 does not do bidding and defender play).

8. TEST RESULTS

One method of scoring duplicate bridge is *Swiss teams board-a-match scoring*. Whichever team gets the higher number of total points wins the board; if the teams have the same number of total points, the board is tied, and each team wins 1/2 a board. For example, on the deal in Figure 5 and Figure 6, the Tignum 2 team scores +120 at one table and the Bridge Baron team scores -100 at the other table; thus, Tignum 2 wins the board.

We held a duplicate bridge competition based on Swiss teams board-a-match scoring on 5000 randomly generated notrump deals between the BB team and the Tignum 2 team. For now, Tignum 2 is better on notrump deals than it is on suit deals, because we have not yet encoded enough bridge knowledge for Tignum 2 to play all suit deals well.

The results of this competition in are shown in Table 4. On 5000 notrump deals, the declarer play of Tignum 2 was 92 boards better than that of the strongest

BB declarer vs. BB defense				Tignum 2 declarer vs. BB defense				
W	N	E	S	W	N	E	S	
T♢	4♢	K♢*	→Q♢	T♢	4♢	K♢*	→Q♢	Get in control
2♡	T♡	→Q♡	A♡*	2♡	T♡	→Q♡	A♡*	Start trying to set up hearts; unblock
3♣	3♢	A♢*	→J♢	3♣	3♢	A♢*	→J♢	Get in control
A♠*	2♠	→4♠	6♡	J♡*	2♠	→8♡	9♡	Try to set up hearts; realize failure
→K♡*	2♣	8♡	4♡	→A♠*	5♠	4♠	7♢	Try to set up spades
→J♡*	9♣	4♣	7♡	→T♠	K♠*	6♠	5♢	Take marked spade finesse
→5♡	5♣	6♠	9♡*	A♣*	→Q♣	4♣	T♣	Get in control
6♣	8♠	2♢	→9♢*	→K♡*	5♣	7♣	4♡	Cash heart winner
3♠	5♠	6♢	→8♢*	→3♠	8♠	9♠*	8♢	Take marked spade finesse
7♠	J♣	7♣	→7♢*	7♠	J♠	→Q♠*	6♡	Cash last spade winner
T♠	Q♣	8♣	→5♢*	3♡	2♣	→2♢	9♢*	Defense gets the last three tricks
A♣*	J♠	9♠	→K♣	6♣	9♣	8♣	→K♣*	
→3♡*	K♠	Q♠	T♣	5♡	J♣	6♢	→7♡*	

2 NT—Down 1
100 to North/South

2 NT—Made
120 to East/West

Tignum 2 team wins board

FIGURE 6. Play of the deal from Figure 5 showing the Tignum 2 team beating the BB team. An arrow (→) indicates a card led to a trick. An asterisk (*) indicates a card that won a trick.

commercially available program. These results are statistically significant at the $\alpha = 0.05$ level. We had never run Tignum 2 on any of these deals before this test, so these results are free from any training-set biases in favor of Tignum 2.

Tignum 2 searched an average of 8745.6 moves per deal on these 5000 notrump deals. Note that in each deal, Tignum 2 must play 26 cards. Tignum 2 never searched more than 583638 moves in a single deal. (In contrast, in the worst case, a brute-force game-tree search would search $\binom{25}{13}(13!)^4/13 = 6.01 \times 10^{44}$ moves.) These small search trees demonstrate the effectiveness of Tignum 2's pruning. Tignum 2's declarer play on each deal averaged 27.5 seconds on a Sun SPARCstation 10.

In the next section, we will look at one of the deals on which Tignum 2 demonstrated the power of its planning ability. We will look at the BB declarer's play, and then the Tignum 2 declarer's play in some depth.

8.1. Example Deal

Figure 5 shows one of the deals on which Tignum 2 demonstrated the power of its planning ability. As seen in Figure 6, Tignum 2 first tried to set up tricks in hearts. After discovering that South had too many hearts, Tignum 2 considered alternative plans and concluded that its best chance of making the contract was to execute a spade finesse. After succeeding, it quickly cashed enough tricks to make the contract.

In the next two subsections, we will see how the bidding and play on this deal by the BB team and the Tignum 2 team took place. In the first subsection, we discuss play at the "first table", and in the second subsection, play at the "second table".

The First Table. At the first table, the Tignum 2 team sat North and South, and the BB team sat East and West. The bidding proceeded as shown in Figure 5—because Tignum 2 does not bid, all four players used the BB bidding routines. East became declarer at a contract of 2NT. Because the BB team won the contract, BB performed the declarer play at this table. Because the Tignum 2 team was on defense, and because Tignum 2 does not perform defender play, BB also performed the defensive play at this table.

The play proceeded, and East, the BB declarer, won the first trick with the A♢. At the second trick, East was on lead. On the basis of its ad-hoc rules, it decided to lead the Q♡, the correct play. South, a defender, won the trick with the A♡, and at trick 3, led back the J♢, setting up four diamond tricks if it ever got back into the lead. After winning trick 3 with the A♢, East, the BB declarer, led the 4♠ at trick 4, and South showed a spade void by playing the 6♡, a card in another suit. West, the BB dummy, played the A♠, winning the trick. At this point, the BB declarer was guaranteed to make the contract with correct play, unless North had all four missing diamonds (which was very unlikely, based on South's leads at trick 1 and trick 3.) At trick 5, West played the K♡, knowing it would win the trick because the A♡ had already been played, and North, a defender, showed a heart void by playing the 2♣.

At this point, on lead at trick 6, the BB declarer was still almost certain to make the contract with correct play. However, West played the J♡, making South's 9♡ a winner. As the cards lay, the BB declarer could still make the contract, but because of its ad-hoc rules, it decided at trick 7 to lead the 5♡, a mistake. After winning the 9♡, South quickly cashed four diamond tricks, for a total of six tricks for the defense. Because the BB declarer had only taken five tricks, and because at the eleventh trick there were only two tricks remaining, the BB declarer could only take a total of seven tricks—but the BB declarer had contracted for eight tricks. Thus, the BB declarer fell one trick short, for a score of -100 to the Bridge Baron team.

The Second Table. Because all four players at the second table were again using the deterministic BB bidding routines, the bidding was identical to that at the first table. East was again declarer at a contract of 2NT. Because the Tignum 2 team won the contract, Tignum 2 performed the declarer play at this table and BB performed the defensive play at this table.

South, a BB defender, made the opening lead of the Q♢. It was then dummy's turn to play, and because Tignum 2 was declarer, Tignum 2 played both declarer's cards and dummy's cards. West, the Tignum 2 dummy, had only one card in the diamond suit, so it was forced to play the 10♢. North, a BB defender, played the 4♢. East, the Tignum 2 declarer, played the K♢. Because there were no alternatives that Tignum 2 considered worth investigating at trick 1 (that is, Tignum 2 believed that it had only one sensible choice), Tignum 2 stopped planning after trick 1, and planned for trick 2 when it was time to play to trick 2. The reasoning that Tignum 2 used to decide on the play of the K♢ is shown in Figure 7.

East was on lead at trick 2. Tignum 2 now did extensive planning to decide among three alternatives: trying to set up hearts; cashing its A♢, for fear that it wouldn't be able to get back to East's hand to do so later; or cashing its high cards (starting with the A♢.)

After generating and evaluating its game tree, Tignum 2 decided to try to set up hearts, and played the Q♡ appropriately. Tignum 2 preferred the Q♡ to the 8♡ on the basis of its bridge knowledge; playing the 8♡ would have made it harder to play the rest of the hearts, because Tignum 2 would have to win the next trick with East's Q♡ and then get back to West to lead the next high heart.

South won trick 2 with the A♡, and at trick 3, led back the J♢, setting up four diamond tricks if it ever got back into the lead.

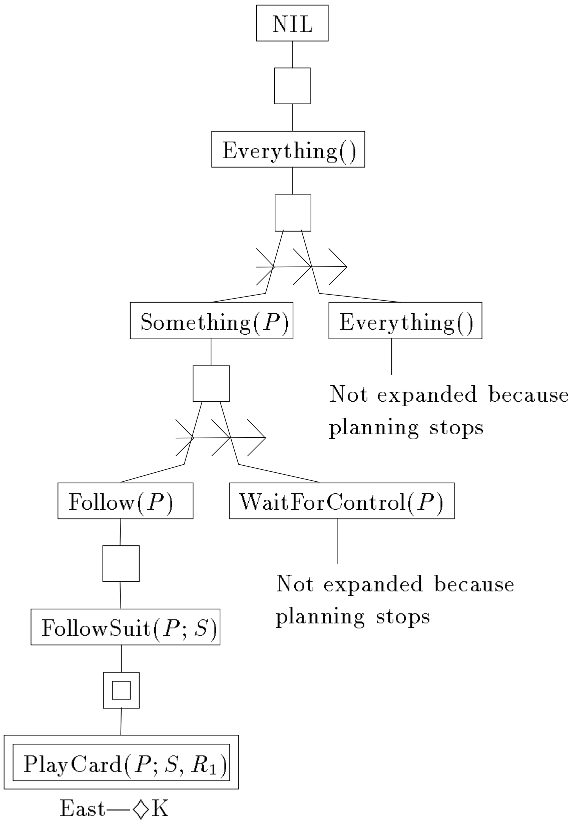

FIGURE 7. Reasoning that Tignum 2 used to decide on the play of the K♢ at trick 1.

Deal:

North
♠ KJ852
♡ —
♢ —
♣ QJ952

West
♠ AT73
♡ KJ53
♢ —
♣ A6

East
♠ Q964
♡ 8
♢ 62
♣ 874

South
♠ —
♡ 9764
♢ 9875
♣ KT

Bidding:

W	N	E	S
			Pass
1♡	Pass	1NT	Pass
2♣	Pass	2NT	Pass
Pass	Pass		

Contract: East—2NT
Vulnerable: All
Declarer: 2 tricks
Defense: 1 trick
On lead: East at trick 4

FIGURE 8. Situation at trick 4 at the second table.

The lead of the J♢ did not meet the criteria of Tignum 2's critic, because Tignum 2 expected a club lead. Thus, after winning trick 3 with the A♢, East re-planned, deciding among three alternatives at trick 4: trying to set up hearts; trying to set up spades; or cashing its high cards (starting with the A♢.)

The situation was now as shown in Figure 8. An annotated version of part of the game tree that Tignum 2 generated and searched is shown in Figure 9.

After generating and evaluating this game tree, Tignum 2 decided to try to set up hearts, and played the 8♡ appropriately; this was the first Tignum 2 declarer play at the second table that differed from BB's declarer play at the first table. South played the 9♡, and West played the J♡ as planned, knowing it would win the trick because the A♡ and Q♡ had already been played and West itself held the K♡. Now North finished the trick, showing a heart void by playing the 2♠.

The rest of the play proceeded as shown in Figure 6. The Tignum 2 declarer took eight tricks, making its contract, for a score of +120 to the Tignum 2 team.

9. CONCLUSION

In this paper, we have described an approach to playing imperfect-information games. By using techniques adapted from task-network planning, our approach re-

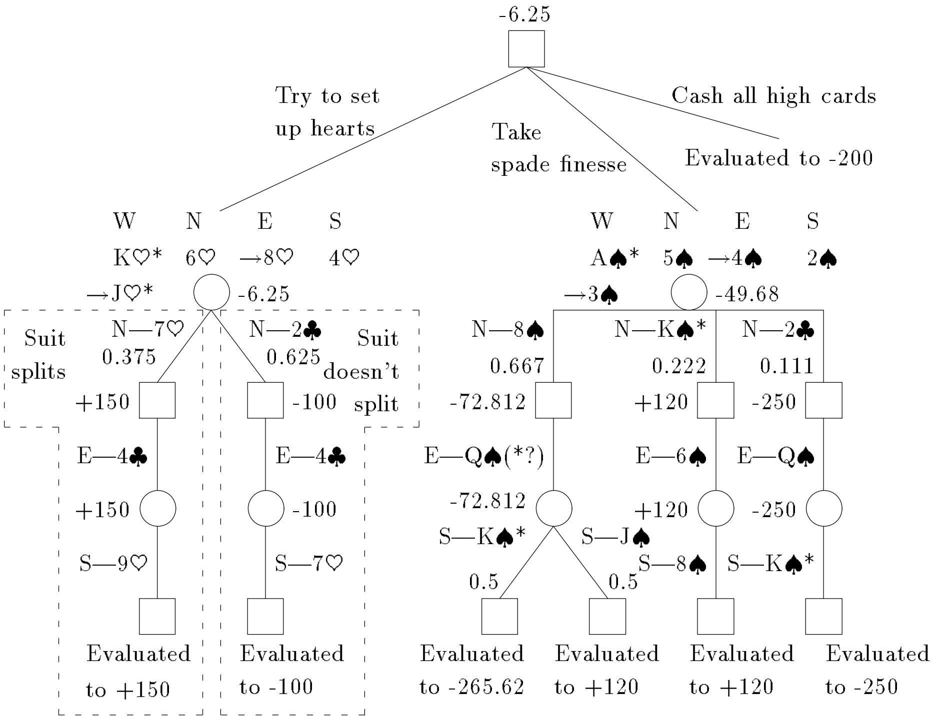

Figure 9. Annotated version of part of the game tree that Tignum 2 investigated at trick 4 for the hand shown in Figure 5.

duces the large branching factor that results from uncertainty in such games. It does this by producing game trees in which the number of branches from each state is determined not by the number of actions that an agent can perform, but instead by the number of different tactical and strategic schemes that the agent can employ. By doing a modified game-tree search on this game tree, one can produce a plan that can be executed for multiple moves in the game.

Our approach appears to be particularly suited to bridge, because bridge is an imperfect-information game that is characterized by a high degree of planning during card play. Thus, to test our approach, we created an implementation, called *Tignum 2*, that uses these techniques to do card-playing for declarer in the game of bridge. The declarer play of this implementation on 5000 notrump deals was statistically significantly better than the declarer play of the strongest commercially available program.

Tignum 2 "consciously" handles cashing out, ruffing out, crossing, finesses, free finesses, automatic finesses, marked finesses, sequence winners, length winners, winners that depend on splits, opponents on lead, opponents finessing against declarer and dummy, dangerous opponents, ducking, hold-up plays, discarding worthless cards, drawing trump, ruffing, and setting up ruffs; i.e., it has tasks and methods to address these tactical and strategic schemes. Endplays and even squeezes are occasional emergent behavior; i.e., while Tignum 2 does not consciously handle endplays and squeezes, occasionally other tasks and methods will combine to produce them. For Tignum 2 to handle endplays and squeezes consciously, we would just have to add tasks and methods; we have yet to address these strategies because they are rare.

Anyone wishing to apply our approach to other domains should consider the following factors that led us to choose and shape our approach:

- Bridge has a natural element of hierarchical planning. Humans use hierarchies of schemes to create plans to play bridge deals. The bridge literature describes many such schemes. Hierarchical planning gives each play a context; without such a context, one might search through many methods at each play.
- Because our approach avoids examining all possible moves for all agents, it is related to the idea of forward pruning. The primary difference from previous approaches to forward pruning is that previous approaches used heuristic techniques to prune "unpromising" nodes from the game tree, whereas our approach simply avoids generating nodes that do not fit into a tactical and strategic scheme for any player. Although forward pruning has not worked very well in games such as chess (Biermann, 1978; Truscott, 1981), our recent study of forward pruning (Smith and Nau, 1994) suggests that forward pruning works best in situations where there is a high correlation among the minimax values of sibling nodes. Part of our motivation for the development of Tignum 2 is that we believe that bridge has this correlation.
- Although our approach is based on ideas from hierarchical task-network planning, it differs from most other task-network planners in that it generates totally ordered plans. This seemed to be the best solution to imperfect information, which causes problems more severe than the uninstantiated variables that occur in perfect-information domains.

We hope that the approach described in this paper will be useful in a variety of imperfect-information domains, possibly including defensive play in bridge. We intend to investigate this issue in future work.

REFERENCES

BALLARD, B. W. 1983. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence* 21:327–350.

BERLIN, D. L. 1985. SPAN: integrating problem solving tactics. *Proc. 9th International Joint Conference on Artificial Intelligence*, 1047–1051.

BERLINER, H. J.; GOETSCH, G.; CAMPBELL, M. S.; and EBELING, C. 1990. Measuring the performance potential of chess programs. *Artificial Intelligence* 43:7–20.

BIERMANN, A. W. 1978. Theoretical issues related to computer game playing programs. *Personal Computing*, September 1978:86–88.

CORMEN, T. H.; LEISERSON, C. E.; and RIVEST, R. L. 1990. *Introduction to Algorithms.* MIT Press/McGraw Hill.

CURRIE, K. and TATE, A. 1985. O-Plan—control in the open planner architecture. BCS Expert Systems Conference, Cambridge University Press, UK.

EROL, K.; NAU, D. S.; and HENDLER, J. 1993. Toward a general framework for hierarchical task-network planning. In *AAAI Spring Symposium.*

EROL, K.; NAU, D. S.; and SUBRAHMANIAN, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, to appear.

FELDMAN, J. A. and YAKIMOVSKY, Y. 1974. Decision theory and artificial intelligence i. A semantics-based region analyzer. *Artificial Intelligence* 5:349–371.

FELDMAN, J. A. and SPROULL, R. F. 1977. Decision theory and artificial intelligence ii: The hungry monkey. *Cognitive Science* 1:158–192.

Fikes, R. E. and Nilsson, N. J. 1971. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Frank, I.; Basin, D.; and Bundy, A. 1992. An adaptation of proof-planning to declarer play in bridge. In *European Conference on Artificial Intelligence*.

French, S. 1986. *Decision Theory: An Introduction to the Mathematics of Rationality.* Wiley: New York.

Gamback, B.; Rayner, M.; and Pell, B. 1990. An architecture for a sophisticated mechanical bridge player. In Beal, D. F. and Levy, D.N.L., editors, *Heuristic Programming in Artificial Intelligence—The Second Computer Olympiad.* Ellis Horwood: Chichester, UK.

Gamback, B.; Rayner, M.; and Pell, B. 1993. Pragmatic reasoning in bridge. Tech. Report 299, Computer Laboratory, University of Cambridge.

Horacek, H. 1990. Reasoning with uncertainty in computer chess. *Artificial Intelligence* 43:37–56.

Khemani, D. 1994. Planning with thematic actions. In *Proc. Second Internat. Conf. AI Planning Systems*, Kristian Hammond, ed. AAAI Press: Menlo Park, CA.

Knuth, D. E. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley Publ. Co.

Lee, K.-F. and Mahajan, S. 1990. The development of a world class othello program. *Artificial Intelligence* 43:21–36.

Levy, D. and Newborn, M. 1982. *All About Chess and Computers.* Computer Science Press.

Lindelof, E. 1983. COBRA: the computer-designed bidding system. Victor Gollancz Ltd: London.

Lopatin, A. 1992. Two combinatorial problems in programming bridge game. *Computer Olympiad*, unpublished.

Manley, B. 1993. Software 'judges' rate bridge-playing products. *The Bulletin* (published monthly by the American Contract Bridge League), Volume 59, Number 11, November 1993:51–54.

Quinlan, J. R. 1979. A knowledge-based system for locating missing high cards in bridge. *Proc. 6th International Joint Conf. Artificial Intelligence*, pp. 705–707.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In Allen, J.; Hendler, J.; and Tate, A., editors, *Readings in Planning.* Morgan Kaufman, 1990. 162–170. Originally appeared in *Proc. 4th International Joint Conf. Artificial Intelligence*, pp. 206-214.

Sacerdoti, E. D. 1977. *A Structure for Plans and Behavior.* American Elsevier Publishing Company.

Samuel, A. L. 1967. Some studies in machine learning using the game of checkers. ii–recent progress. *IBM Journal of Research and Development* 2:601–617.

Schaeffer, J.; Culberson, J.; Treloar, N.; Knight, B.; Lu, P.; and Szafron, D. 1992. A world championship caliber checkers program. *Artificial Intelligence* 53:273–290.

Smith, S. J. J. and Nau, D. S. 1993. Strategic planning for imperfect-information games. In *Games: Planning and Learning, Papers from the 1993 Fall Symposium*, Technical report FS9302, AAAI Press, Menlo Park, CA.

Smith, S. J. J. and Nau, D. S. 1994. An analysis of forward pruning. In *Proc. 12th National Conference on Artificial Intelligence*, pp. 1386–1391.

Stanier, A. 1975. Bribip: a bridge bidding program. In *Proc. 4th International Joint Conf. Artificial Intelligence.*

Sterling, L. and Nygate, Y. 1990. Python: an expert squeezer. *Journal of Logic Programming* 8:21–39.

Tate, A. 1976. Project planning using a hierarchic non-linear planner. Tech. Report 25, Department of Artificial Intelligence, University of Edinburgh.

Tate, A. 1977. Generating project networks. In *Proc. 5th International Joint Conf. Artificial Intelligence.*

Truscott, T. R. 1981. Techniques used in minimax game-playing programs. Master's thesis, Duke University, Durham, NC.

WILKINS, D. E. 1980. Using patterns and plans in chess. *Artificial Intelligence* 14:165–203.

WILKINS, D. E. 1982. Using knowledge to control tree searching. *Artificial Intelligence* 18:1–51.

WILKINS, D. E. 1984. Domain independent planning: representation and plan generation. *Artificial Intelligence* 22:269–301.