TECHNICAL RESEARCH REPORT

Strategic Planning for Imperfect-Information Games

by S.J.J. Smith and D.S. Nau

T.R. 93-56

The Institute for Systems Research is supported by the
National Science Foundation Engineering Research Center Program (NSFD CD 8803012),
the University of Maryland, Harvard University, and Industry

Strategic Planning for Imperfect-Information Games *

Stephen J. J. Smith
Computer Science Department
University of Maryland
College Park, MD 20740
sjsmith@cs.umd.edu

Dana S. Nau
Institute for Advanced Computer Studies,
Computer Science Department,
and Institute for Systems Research
University of Maryland
College Park, MD 20740
nau@cs.umd.edu

June 30, 1993

Abstract

Although game-tree search works well in perfect-information games, there are problems in trying to use it for imperfect-information games such as bridge. The lack of knowledge about the opponents' possible moves gives the game tree a very large branching factor, making the tree so immense that game-tree searching is infeasible.

In this paper, we describe our approach for overcoming this problem. We develop a model of imperfect-information games, and describe how to represent information about the game using a modified version of a task network that is extended to represent multi-agency and uncertainty. We present a game-playing procedure that uses this approach 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.

In our tests of this approach on the game of bridge, we found that it generated trees having a much smaller branching factor than would have been generated by conventional game-tree search techniques. Thus, even in the worst case, the game tree contained only about 1300 nodes, as opposed to the approximately 6.01×10^{44} nodes that would have been produced by a brute-force game tree search in the worst case. Furthermore, our approach successfully solved typical bridge problems that matched situations in its knowledge base. These preliminary tests suggest that our approach has the potential to yield bridge-playing programs much better than existing ones—and thus we have begun to build a full implementation.

Address correspondence to Dana S. Nau, Computer Science Dept., University of Maryland, College Park, MD 20742.

*This 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 NSF grants IRI-8907890 and NSFD CDR-88003012.

1 Introduction

Although game-tree search works well in perfect-information games (such as chess [2, 16], checkers [20], and othello [15]), it does not always work so well in other games. For example, the game of bridge is an imperfect-information game, in which no player has complete knowledge about the state of the world, the possible actions, and their effects. As a consequence, the branching factor of the game tree is very large, making the tree so immense that game-tree searching is completely infeasible. Thus, a different approach is needed.

In this paper, we describe an approach to this problem, based on the observation that bridge is a game of planning. The bridge literature describes a number of tactical and strategic schemes for dealing with various card-playing situations. It appears that there is a small number of such schemes for each bridge hand, 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 instances of *multi-agent methods*—structures similar to the task decompositions used in hierarchical single-agent planning systems such as Nonlin [25, 26], NOAH [19], and MOLGEN [24], but modified to represent multi-agency and uncertainty. To generate game trees, we use a procedure similar to task-network decomposition.

This approach produces 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. For example, a prototype implementation of our approach produced game trees of no more than about 1300 nodes in the worst case—a significant reduction from the approximately 6.01×10^{44} nodes that would have been produced by a brute-force game tree search in the worst case.

Since our approach avoids generating all possible moves for all agents, it is in essence a type of forward pruning. Although forward pruning has not worked very well in games such as chess [3, 27], our study of forward pruning [23] suggests that forward pruning works best in situations where there is a high correlation among the minimax values of sibling nodes. We believe that bridge has this characteristic. This encourages us to believe that our approach may work well in the game of bridge—and a preliminary study of our prototype implementation suggests that our approach can do well on at least some bridge hands. To test this idea further, we are developing a full implementation of the approach.

This paper is organized as follows. Section 1.1 discusses related work. Section 2 presents our formalism for representing certain kinds of imperfect-information games, and the uncertainty that can occur in them. Section 3 presents our structure for representing game-playing knowledge using a task-network formalism that is modified to represent multi-agency. Section 4 describes an abstract game-playing procedure based on this structure. Section 5 describes our preliminary tests of an implementation of this procedure in the game of bridge. Section 6 contains concluding remarks.

1.1 Related Work

The strongest work on bridge has focused on bidding [11, 12]. However, there are no really good computer programs for card-playing in bridge—most of them can be beaten by a reasonably advanced novice. The approaches used in current programs are based almost exclusively on domain-specific techniques. Some programs (such as Great Game Products' *Bridge Baron* program) do decision-making by repeatedly generating random hypotheses for what hands the opponents might have, and doing a full game tree-search for each hypothesized hand—but this approach is feasible only late in the game, after most of the tricks have been played. Recently, Frank and others [9] have proposed a proof-planning approach, but thus far, this approach has only been used for planning the play of a single suit.

Some work has been done on extending game-tree search to deal with uncertainty, including Horacek's work on chess [13], and Ballard's work on backgammon [1]. However, these works do not deal with the kind of uncertainty that we discussed in Section 1, and thus it does not appear to us that these approaches would be sufficient to accomplish our objectives.

Wilkins [28, 29] uses "knowledge sources" to generate chess moves for both the player and the opponent, and then chooses the move to play by investigating the results. In their intent, these knowledge sources are similar to the multi-agent methods that we describe in Section 3. However, since chess is a perfect-information game, Wilkins' work does not deal with uncertainty and incomplete information.

Our work on hierarchical planning draws on Tate's [25, 26] which in turn draws on Sacerdoti's [17, 18]. In addition, some of our definitions were motivated by those in [5, 6].

2 A Model of Imperfect-Information Games

In this section, we describe a formalism for expressing imperfect-information games having the following characteristics:

1. Only one player may move at at time.
2. Each player need not have complete 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; i.e., the player knows a finite set of moves M such that every move that A can make is a member of M.

Our motivating example isthe game of bridge. The specific problem we wish to address is how the declarer should play the game after the bidding is over (i.e., once trump suit has been chosen and one of the declarer's opponents has made an initial lead). There are four hands around the table; our player will play two hands (the declarer and the dummy), and two external agents (the defenders) will play the other two hands. The two hands played by our agent are in full view of our agent at all times; the two other hands are not, hence the incomplete information.

In order to talk about the current state S (or any other state) in an abstract manner, we will consider it to be a collection of *ground atoms* (i.e., 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 or not 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).

Since $\mathcal{P}$ 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. Since 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 $\neg$ 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 atoms (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 $[0, 1]$ is the set of reals falling between 0 and 1.

To represent the possible actions of the players, we use STRIPS-style operators. More specifically, if $X_0, X_1, X_2, \ldots, X_n$ are variable symbols, then we define a *primitive operator* $O(X_0; X_1, X_2, \ldots, X_n)$ to be the following triple:[1]

1. Pre(O) is a finite set of literals, called the *precondition list* of O, whose variables are all from the set $\{X_0, X_1, X_2, \ldots, X_n\}$. Pre($O$) must always include the atoms Agent(X_0) and Turn(X_0).

2. Add(O) and Del(O) are both finite sets of atoms (possibly non-ground) whose variables are all from the set $\{X_0, X_1, X_2, \ldots, X_n\}$. 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 (♣, ♢, ♡, or ♠), 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, that is, a primitive operator O is *applicable* in a state S if every literal in Pre(O) is in S. If an instantiation of a primitive operator (say $O(a_0; a_1, a_2, \ldots, a_n)$) is applicable in some state $S_a \in I^*$, and if Control(a_0) 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_0 whose turn it is to move, then $\mathcal{P}$ will have enough additional information to determine which moves a_0 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 define $\mathcal{S}$ to be the set of all states. We define $\mathcal{I}$ to be the set of all state information sets.

An *objective function* is a partial function $f : \mathcal{S} \rightarrow [0, 1]$. 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 $e : \mathcal{S} \rightarrow [0, 1]$ 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, our game-playing procedure 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.[2]

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, an *problem* in G is a quadruple $P = (I_{S_0}, p, f, e)$, where I_{S_0} is the state information about an *initial* state S_0, p is a belief function, f is an objective function and e is a static evaluation function. For example,

[1] The semicolon separates X_0 from the rest of the arguments because X_0 is the agent who uses the operator when it is X_0's turn to move.

[2] 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.

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 Networks of Multi-Agent Methods

We define a *primitive operator method*, or just *operator method*, M, to be a triple (Task(M), Pre(M), Operator(M)), where

1. Task(M) is a *task*. This may either be the expression 'NIL' or a syntactic expression of the form $M(X_0; X_1, X_2, \ldots, X_n)$ where each X_i is a variable of $\mathcal{L}$.

2. Pre(M) is a finite set of literals, called the *precondition* list of M. Pre(M) must always include all of the literals in the precondition list of the operator $O(t_0, t_1, \ldots, t_m)$, where O is as described below.

3. Operator(M) is a syntactic expression $O(t_0, t_1, \ldots, t_m)$. O is an operator in $\mathcal{O}$, and $t_0, t_1, \ldots, t_m$ are terms of $\mathcal{L}$.

For example, in bridge, the playing of a single card would probably have its own primitive operator method M. Task(M) might be PlayCardMethod($P; S, R$). Operator(M) would be the syntactic expression PlayCard($P; S, R$), where PlayCard($P; S, R$) is the primitive operator described in Section 2. Pre(M) would be identical to the precondition list for PlayCard($P; S, R$).

A *decomposable method* M is a triple (Task(M), Pre(M), Expansion(M)), where

1. Task(M) is again either the expression 'NIL' or a syntactic expression of the form $M(X_0; X_1, X_2, \ldots, X_n)$, where each X_i is a variable of $\mathcal{L}$.

2. Pre(M) is a finite set of literals, called the *precondition* list of M. Pre(M) must always include the atom Agent(X_0).

3. Expansion(M) is a (possibly empty) tuple of tasks $(T_1(t_{1,1}, t_{1,2}, \ldots, t_{1,m_1}), T_2(t_{2,1}, t_{2,2}, \ldots, t_{2,m_2}), \ldots, T_k(t_{k,1}, t_{k,2}, \ldots, t_{k,m_k}))$.

For example, Figure 1 shows part of the network of multi-agent methods that we use for finessing in bridge. In this figure, each box represents a decomposable method.

If an instantiation of a method (say $M(a_0; a_1, a_2, \ldots, a_n)$) is applicable in some state $S_a \in I^*$, and if Control(a_0) 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_0 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* (since the Expansions of methods serve to link methods, the term "network" is appropriate.)

4 Game-Playing Procedure

Our game-playing procedure constructs a decision tree, then evaluates the decision tree to produce a plan for how to play 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.

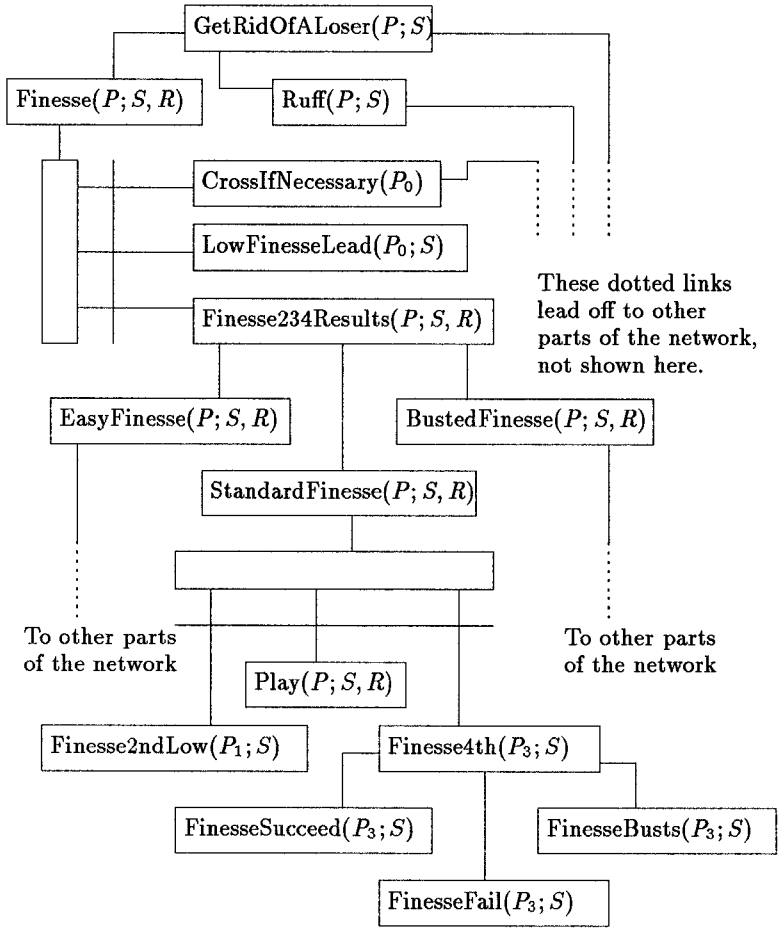

Figure 1: The part of the network of multi-agent methods for the game of bridge that deals with finessing.

4.1 Constructing a Decision Tree

Given a network of multi-agent methods and a state information set I, our game-playing procedure uses these methods to construct a *decision tree* rooted at I.[3] A decision tree resembles a game tree.[4] It contains two kinds of non-leaf nodes: *decision nodes*, representing 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 leaf nodes are nodes at which the procedure does not have any methods to apply, either because the game has ended, or because the methods simply don't tell the procedure what to do.

[3] We use the term "decision tree" as it is used in the decision theory literature [10, 7, 8], to represent a structure similar to a game tree. We are not employing decision trees (also, and less ambiguously, referred to as *comparison trees* [14]) as they are defined in the sorting literature ([4], p. 173.) We apologize for the confusion, but it is inescapable.

[4] 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.

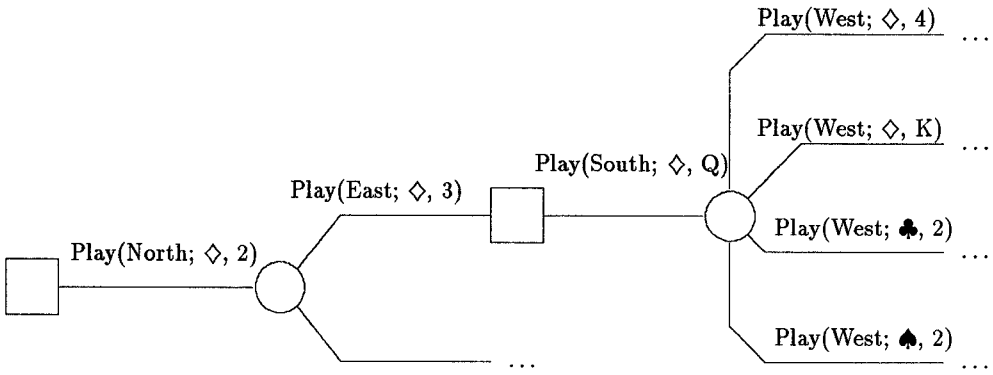

Figure 2: A piece of the decision tree that might be generated by the finessing part of the network of multi-agent methods for the game of bridge.

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 node of T be a node containing the state information set I and no tasks.

loop:

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.')

repeat

As an example of the operation of this procedure, Figure 2 shows a piece of a decision tree that the procedure might generate for finessing, using the network showin in Figure 1.

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 $\overline{M}$ is *applicable* to u if the following conditions hold:

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

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 Pre($\overline{M}$), then all states S' consistent with I_u (i.e., all $S' \in I_u^*$) satisfy Pre($\overline{M}$). We made this property a requirement of our multi-agent methods in Section 3.

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 Pre($\overline{M}$), while some other state S_2 consistent with I_u does **not** satisfy Pre($\overline{M}$). In this case, our procedure will need to make the assumption that Pre($\overline{M}$) 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 Pre($\overline{M}$) does not hold.

For example, in Figure 2, before investigating the move Play(West; ♢, 4), our procedure would need to make the assumption that West holds the 4♢. The procedure would investigate the other moves for West under different assumptions (say, that West holds the K♢, or that West holds no cards in the ♢ suit.)

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 is a decomposable method, then $I_v = I_u \cup \text{Pre}(\overline{M})$. Intuitively, I_v is I_u with all the conditions in $\text{Pre}(\overline{M})$ assumed.

  If M is an operator method, then

  $$I_v = I_u \cup \text{Pre}(\overline{M}) \cup \text{Add}(\text{Operator}(\overline{M})) - \text{Del}(\text{Operator}(\overline{M})).$$

  Intuitively, I_v is I_u, with all the conditions in $\text{Pre}(\overline{M})$ assumed, and then all the conditions in $\text{Add}(\overline{M})$ added, and all the conditions in $\text{Del}(\overline{M})$ deleted.

- If M is a decomposable method and $\text{Expansion}(M) = (V_1, V_2, \ldots, V_m)$, then $\mathcal{V} = (V_1, \ldots, V_m, U_2, U_3, \ldots, U_n)$. If M is an operator method, then $\mathcal{V} = (U_2, U_3, \ldots, U_n)$. 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 $U_2, U_3, \ldots, U_n$.

4.2 Evaluator 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 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.

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: since 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 the nodes $u_1, u_2, \ldots, u_n$. 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 $v = C(u, u_1, u_2, \ldots, u_n)$ for the node u.[5]

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, known to computer scientists as minimax). Some will make use of the belief function p, others will not. In bridge, we would generally use an external-agent criterion that would give a weighted average, using p, 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.

Given a decision tree, a external-agent criterion for each uncontrolled agent, an objective function, and a belief function, we define an *evaluation* of 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 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 maximium 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).

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.

[5] This definition of external-agent criterion is somewhat different from the usual definition of decision criterion in decision theory (e.g. [10], 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.

Contract: South - 5 ♠
Lead: West - 6 ♢
Deal:

	North ♠ 432 ♡ 5432 ♢ 5432 ♣ 32	
West ♠ T98 ♡ Q96 ♢ T86 ♣ A987	South ♠ AKQJ65 ♡ AK ♢ AK ♣ 654	East ♠ 7 ♡ JT87 ♢ QJ97 ♣ KQJT

Play:

W	N	E	S
6♢	2♢	J♢	**A♢**
7♣	2♣	**J♣**	4♣
T♢	3♢	7♢	**K♢**
8♣	3♣	**T♣**	5♣
8♢	4♢	9♢	**5♠**
9♣	**2♠**	Q♣	6♣
9♠	3♠	7♠	**A♠**
8♠	4♠	K♣	**K♠**
T♠	5♢	Q♢	**Q♠**
A♣	2♡	8♡	**J♠**
6♡	3♡	7♡	**6♠**
9♡	4♡	T♡	**A♡**
Q♡	5♡	J♡	**K♡**

Figure 3: One of the hands that Tignum correctly solved. Ranks in boldface represent cards that won tricks.

$\mathcal{P}$ 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.

5 Implementation and Preliminary Tests in Bridge

To test our approach, we implemented and tested a small prototype, called *Tignum*. Since Tignum is a prototype and its knowledge base was both too small and too inaccurate, we did not carry out extensive tests. However, as described in this section, we performed some preliminary tests of Tignum on a suite of bridge hands.

We developed our suite of bridge hands with several criteria in mind. Some of them were developed in order to present various "worst case" situations to Tignum, some were developed to represent "normal" bridge hands, and some were developed to represent situations that cause problems to previously developed bridge programs.

Our test of Tignum had the following results:

- On our suite of hands, the largest search tree generated by Tignum contained 1301 nodes. Since we selected our suite of hands to include hands that we thought would generate "worst-case" sized search trees, we believe that Tignum will not generate search trees significantly larger than this. This represents a large improvement over the 6.01×10^{44} nodes that would be generated by a brute-force game-tree search.
- On hands that presented situations to Tignum that could be solved by schemes in Tignum's limited knowledge base, Tignum came up with the correct plans. An example is shown in Figure 3, where Tignum successfully sets up and takes a club ruff in dummy.

These results were encouraging enough that we decided to proceed with the full implementation of our approach, called *Tignum II*, which we are still developing.

6 Conclusion

In this paper, we have described an approach to playing imperfect-information games. By using techniques adapted from task-network planning, our approach reduces the large branching factor that results from uncertainty in the game. 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, since bridge is an imperfect-information game that is characterized by a high degree of planning during card play. Thus, to test our approach, we have implemented a prototype system that uses these techniques to do card-playing in the game of bridge. In testing this prototype, we have found that it produces game trees small enough that it can search them all the way to the end of the game—and by doing so, it can successfully solve typical bridge problems that matched situations in its knowledge base.

These preliminary tests suggest that this approach has the potential to yield bridge-playing programs much better than existing ones—and thus we have begun a full implementation of our approach. We hope that our work will be useful in a variety of imperfect-information games.

References

[1] B. Ballard. "The *-Minimax Search Procedure for Trees Containing Chance Nodes." *Artificial Intelligence* 21 (1983): 327–350.

[2] H. J. Berliner, G. Goetsch, M. S. Campbell, and C. Ebeling. Measuring the performance potential of chess programs. *Artificial Intelligence* **43**:7–20, 1990.

[3] A. W. Biermann. Theoretical issues related to computer game playing programs. *Personal Computing*, September 1978:86-88.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* Cambridge, Massachusetts: MIT Press, 1990.

[5] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. 1992. Submitted for publication.

[6] K. Erol, D. S. Nau, and J. Hendler. Toward a general framework for hierarchical task-network planning. In *AAAI Spring Symposium*, April 1993.

[7] J. A. Feldman and Y. Yakimovsky. "Decision Theory and Artificial Intelligence I. A Semantics-Based Region Analyzer." *Artificial Intelligence* 5 (1974): 349–371.

[8] J. A. Feldman and R. F. Sproull. "Decision Theory and Artificial Intelligence II: The Hungry Monkey." *Cognitive Science* 1 (1977): 158–192.

[9] I. Frank, D. Basin, and A. Bundy. "An Adaptation of Proof-Planning to Declarer Play in Bridge." In *European Conference on Artificial Intelligence*, 1992.

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

[11] B. Gambäck, M. Rayner, and B. Pell. "An Architecture for a Sophisticated Mechanical Bridge Player", in: D. F. Beal and D.N.L. Levy(eds.), *Heuristic Programming in Artificial Intelligence—The Second Computer Olympiad.* Chinchester, England: Ellis Horwood, 1990.

[12] B. Gambäck, M. Rayner, and B. Pell. "Pragmatic Reasoning in Bridge". Cambridge: Technical Report No. 299, Computer Laboratory, University of Cambridge [1993].

[13] H. Horacek. "Reasoning with Uncertainty in Computer Chess." *Artificial Intelligence* 43 (1990): 37-56.

[14] D. Knuth. *The Art of Computer Programming*, vol. 3: *Sorting and Searching.* Reading, Massachusetts: Addison Wesley, 1973.

[15] K.-F. Lee and S. Mahajan. The Development of a World Class Othello Program. *Artificial Intelligence* **43**:21–36, 1990.

[16] D. Levy and M. Newborn. *All About Chess and Computers.* Computer Science Press, Rockville, MD, 1982.

[17] E. D. Sacerdoti. "Planning in a Hierarchy of Abstraction Spaces." *Artifical Intelligence* 5 (1974): 115–135.

[18] E. D. Sacerdoti. "The Non-Linear Nature of Plans." *IJCAI-75*: pp. 206–214.

[19] E. D. Sacerdoti. *A Structure for Plans and Behavior.* New York: American Elsevier Publishing Company, 1977.

[20] A. L. Samuel. Some studies in machine learning using the game of checkers. II - Recent progress. *IBM Journal of Research and Development*, **2**: 601-617, 1967.

[21] S. J. J. Smith. "Game-Playing with Uncertainty: Reasoning Techniques." M. S. thesis, University of Maryland at College Park, 1991.

[22] S. J. J. Smith, D. S. Nau, and T. Throop. "A Hierarchical Approach to Strategic Planning with Non-Cooperating Agents under Conditions of Uncertainty." In *Proceedings of the First International Conference on AI Planning Systems*, pages 299–300, June 1992.

[23] S. J. J. Smith and D. S. Nau. Toward an analysis of forward pruning. 1993. Submitted for publication.

[24] M. Stefik. Planning with Constraints (MOLGEN: Part 1). *Artificial Intelligence* **16**:111–140, 1981.

[25] A. Tate. *Project Planning Using a Hierarchic Non-Linear Planner.* Edinburgh: Research report no. 25, Department of Artificial Intelligence, University of Edinburgh [1976].

[26] A. Tate. "Generating Project Networks." *IJCAI-77*: 888–893.

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

[28] D. Wilkins. "Using Patterns and Plans in Chess." *Artificial Intelligence* 14 (1980): 165–203.

[29] D. Wilkins. "Using Knowledge to Control Tree Searching." *Artificial Intelligence* 18 (1982): 1–51.