CHAPTER 23

Planning in the Game of Bridge

23.1 Introduction

This chapter describes how an adaptation of HTN planning is used in Bridge Baron, a computer program for the game of bridge. The use of HTN planning in Bridge Baron contributed to Bridge Baron's winning the 1997 world championship of computer bridge.

23.2 Overview of Bridge

Bridge is a game played by four players, using a standard deck of 52 playing cards, divided into four suits (spades ♠, hearts ♡, diamonds ◇, and clubs ♣), each containing 13 cards. The players, who are normally referred to as North, South, East, and West, play as two opposing teams, with North and South playing as partners against East and West. A bridge deal consists of two phases, bidding and play.

1. *Bidding.* Whichever player was designated as dealer for the deal deals the cards, distributing them equally among the four players. Each player holds his or her cards so that no other player can see them.

   After the cards are dealt, the players make *bids* for the privilege of determining which suit (if any) is the *trump* suit (i.e., cards of this suit will win over cards not of this suit). Nominally, each bid consists of two things: a proposed trump suit or a bid of "notrump" to propose that no suit should be trump, and how many *tricks* (see below) the bidder promises to take. However, various bidding conventions have been developed in which these bids are also used to convey information to the bidder's partner about how strong the bidder's hand is.

   The bidding proceeds until no player is willing to make a higher bid. At that point, the highest bid becomes the contract for the hand. In the highest

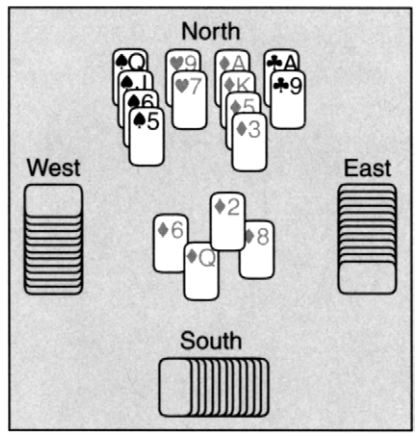

Figure 23.1 The basic unit of play is the trick, in which each player places a card face up in the middle of the table. In this example, West leads the 6 of diamonds, North (dummy) plays the 2 of diamonds, East plays the 8 of diamonds, and South (declarer) plays the queen of diamonds. The latter takes the trick.

bidder's team, the player who bid this suit first becomes the *declarer*, and the declarer's partner becomes the *dummy*. The other two players become the *defenders*.

2. *Play*. The first time that it is the dummy's turn to play a card (see below), the dummy lays her or his cards on the table, face up so that everyone can see them. During the card play, the declarer is responsible for playing both the declarer's cards and the dummy's cards.

   The basic unit of card play is the *trick*, in which each player in turn plays a card by placing it face up on the table as shown in Figure 23.1. Whenever possible, the players must *follow suit*: they must play cards of the same suit as the card that was *led*, i.e., played by the first player. The trick is taken by the highest card in the suit led, unless some player plays a card in the trump suit, in which case the highest trump card takes the trick. The first trick is led by the player to the left of the declarer, and the winner of each trick plays the lead card for the next trick.

   The card play proceeds, one trick at a time, until no player has any cards left. At that point, the bridge hand is scored according to how many tricks each team took and whether the declarer's team took as many tricks as promised during the bidding.

In most bridge hands, the declarer spends some time at the beginning of the game planning how to play his or her cards and the dummy's cards. Because the declarer cannot be certain of which cards are in each opponent's hand and how each opponent will choose to play those cards, the plan needs to contain contingencies

for various possible card plays by the opponents. Thus, rather than being a single linear sequence of moves, the plan is more like a tree structure (see Section 23.3).

The plan is normally a combination of various stratagems for trying to win tricks. There are a number of well-known stratagems, which have names like ruffing, cross-ruffing, finessing, cashing out, discovery plays, and so forth (Section 23.4 gives an example). The ability of a bridge player depends partly on how skillfully he or she can plan and execute these stratagems. This is especially true for the declarer, who is responsible for playing both his or her cards and the dummy's cards.

23.3 Game-Tree Search in Bridge

Game-tree search is a technique for analyzing an adversarial game in order to try to determine who can win the game and what moves the players should make in order to win. Game-tree search is one of the oldest topics in AI. The original ideas were developed by Shannon [472] in 1950 and independently by Turing in 1951, in the context of the game of chess. Their ideas still form the basis for the techniques used today.

Computer programs based on game-tree search techniques are now as good as or better than humans in several popular games of strategy, such as chess [274] and checkers [465]. However, as described in this section, there are some difficulties in finding a good way to use game-tree search techniques in the game of bridge, and even the best bridge programs still play significantly more poorly than the best human bridge players.

A *game tree* is a tree whose nodes represent states of the game and whose branches represent the moves made by the agents who are playing the game. Game trees can be used to model any game that satisfies the following restrictions: The game is played by two players, whom we will call *Max* and *Min*, who make moves sequentially rather than simultaneously; at each turn a player can choose from only a finite number of possible moves; the game is guaranteed to end within some finite number of moves; the game is zero-sum (i.e., an outcome that is good for one player is equally bad for the other player); and at each point in the game, each player knows the entire state of the game and knows what moves the other player is able to make.

In a game tree, the leaf nodes represent the possible outcomes of the game and have numeric "payoff values" that represent the outcome for one of the players. (Because the game is a zero-sum game, the outcome for the other player is taken to be the negative of the payoff value.) Values are computed for the other nodes of the tree using the well-known *minimax formula*, which is defined recursively as follows:

$$m(u) = \begin{cases} \text{the payoff value for } u & \text{if } u \text{ is a leaf node} \\ \max\{m(v) | v \text{ is a child of } u\} & \text{if it is Max's move at } u \\ \min\{m(v) | v \text{ is a child of } u\} & \text{if it is Min's move at } u \end{cases}$$

The minimax computation is basically a brute-force search. If implemented as shown in the formula, it would examine every node in the game tree. In practical implementations of minimax game-tree searching, a number of techniques are used to improve the efficiency of this computation: putting a bound on the depth of the search, using alpha-beta pruning, doing transposition-table lookup, and so forth. However, even with enhancements such as these, minimax computations often involve examining huge numbers of nodes in the game tree. For example, in the match between Deep Blue and Kasparov in 1997, Deep Blue examined roughly 60 billion nodes per move [274]. In contrast, humans examine at most a few dozen board positions before deciding on their next moves [70].

In adapting the game-tree model for use in bridge, it is not too hard to accomodate four players rather than two because the players play as two opposing teams. However, it is more difficult to find a good way to accomodate the fact that bridge is a *partial-information* game. Since bridge players don't know what cards are in the other players' hands (except, after the opening lead, what cards are in the dummy's hand), each player has only partial knowledge of the state of the world, the possible actions, and their effects. If we were to construct a game tree that included all of the moves a player *might* be able to make, the size of this tree would vary from one deal of the cards to another—but it would include about 5.6×10^{44} leaf nodes in the worst case ([493], p. 226), and about 2.3×10^{24} leaf nodes in the average case ([369], p. 8). Because a bridge hand is normally played in just a few minutes, there is not enough time to search enough of this tree to make good decisions.

One way to make the game tree smaller is to generate many random hypotheses for how the cards might be distributed among the other players' hands, generate and search the game trees corresponding to each of the hypotheses, and average the results to determine the best move. This Monte Carlo approach removes the necessity of representing uncertainty about the players' cards within the game tree itself, thereby reducing the size of the game tree by as much as a multiplicative factor of 5.2×10^6. However, to reduce the number of nodes from 5.6×10^{44} down to about 10^{38} still leaves a prohibitively large game tree, unless other ways can be found to reduce the size of the tree even further.

Another way to reduce the size of the game tree is to take the value that was computed for one node of a game tree and use it as the value of other nodes that are sufficiently similar to the first one. For example, if a player could play the ♠6 or the ♠5, then these plays are basically equivalent, so the same value could be used for both of them. This scheme was originally developed for use in the game of Sprouts [22], but it is used in several computer bridge programs (e.g., [230]).

A third approach, which is the one described in this chapter, is based on the observation that bridge is a game of planning. The bridge literature describes a number of stratagems (finessing, ruffing, cross-ruffing, and so forth) that people combine into strategic plans for how to play their bridge hands. It is possible to take advantage of the planning nature of bridge, by adapting and extending some ideas from total-order HTN planning (see Chapter 11). For declarer play in bridge, a modified version of total-order HTN planning can be used to generate a game tree whose branching factor depends on the number of different stratagems that a

Table 23.1 Game-tree size produced in bridge by a full game-tree search and by the planning algorithm used for declarer play in Bridge Baron.

	Brute-force search	*Bridge Baron*
Worst case	About 5.6×10^{44} leaf nodes	About 305,000 leaf nodes
Average case	About 2.3×10^{24} leaf nodes	About 26,000 leaf nodes

player might pursue, rather than the number of different possible ways to play the cards [495]. Because the number of sensible stratagems is usually much less than the number of possible card plays, this approach generates game trees that are small enough to be searched completely, as shown in Table 23.1.

23.4 Adapting HTN Planning for Bridge

Tignum 2 is an algorithm for declarer play in bridge, based on a modified version of total-order HTN planning. To represent the various stratagems of card playing in bridge, Tignum 2 uses structures similar to totally ordered methods, but modified to represent uncertainty and multiple players.

- Although Tignum 2 cannot be certain about which cards have been dealt to each of the opponents, it can calculate the probabilities associated with the locations of those cards—and as part of the current state, Tignum 2 includes "belief functions" that represent those probabilities.
- Some methods refer to actions performed by the opponents. These methods make assumptions about the cards in the opponents' hands, and Tignum 2's authors wrote a large enough set of methods that most of the likely states of the world are each covered by at least one method.

To generate game trees, Tignum 2 uses a procedure similar to Forward-decomposition (see Chapter 11), but adapted to build up a game tree rather than a plan. The branches of the game tree represent moves generated by the methods. Tignum 2 applies all methods applicable to a given state of the world to produce new states of the world and continues recursively until there are no applicable methods that have not already been applied to the appropriate state of the world.

For example, Figure 23.2 shows a bridge hand and a portion of the task network that Tignum 2 would generate for this hand. (Note that it refers to actions performed by each of the players in the game.) This portion of the task network is generated by the HTN methods for *finessing*, a stratagem in which the declarer tries to win a trick with a high card by playing it after an opponent who has a higher card. West leads the ♠2. If North (a defender) has the ♠Q but does not play it, then East (dummy) will

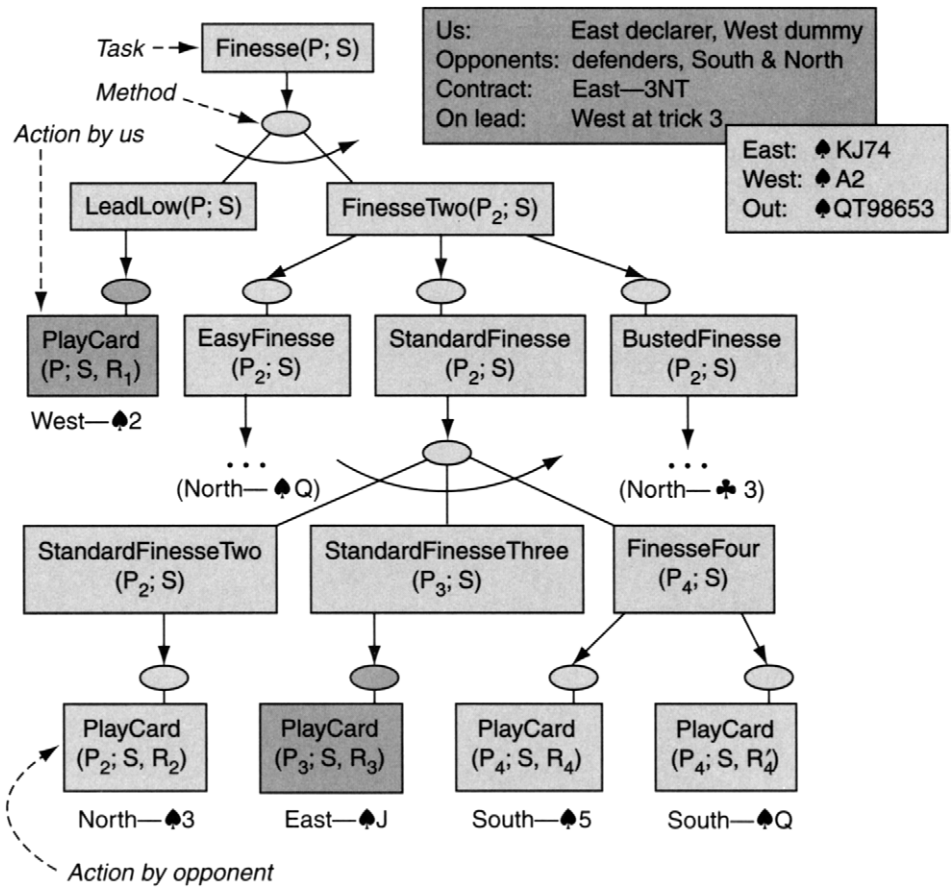

Figure 23.2 A bridge hand, and a portion of the task network that Tignum 2 would generate for finessing in this bridge hand.

be able to win a trick with the ♠J because East plays after North. (North wouldn't play the ♠Q if she or he had any alternative because then East would win the trick with the ♠K and would win a later trick with the ♠J.) However, if South (the other defender) has the ♠Q, South will play it after East plays the ♠J, and East will not win the trick.

In a task network generated by Tignum 2, the order in which the actions will occur is determined by the total-ordering constraints. By listing the actions in the order they will occur, the task network can be "serialized" into a game tree. As an example, Figure 23.3 shows the process of serializing the task network in Figure 23.2. Figure 23.4 shows a game tree that includes both the serialization produced in Figure 23.3 and the sequence of the actions generated by another stratagem called *cashing out*, in which the declarer simply plays all of the high cards that are guaranteed to win tricks.

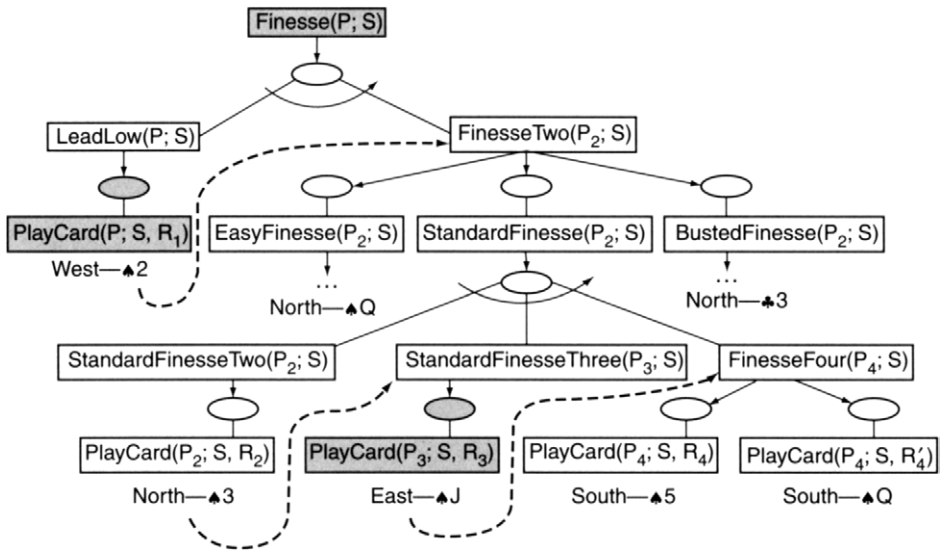

Figure 23.3 Serializing the task network for finessing.

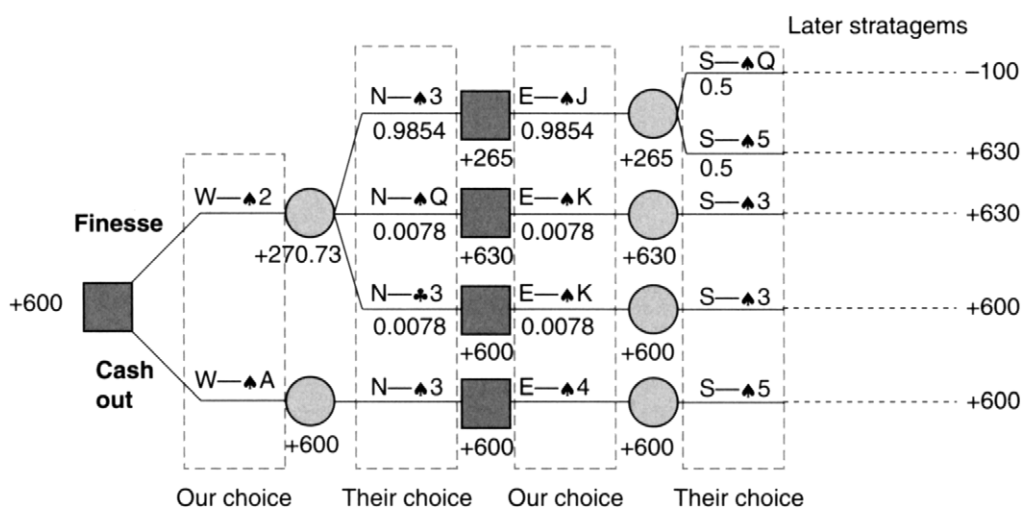

Figure 23.4 The game tree produced from the task network for finessing and a task network for cashing out.

For game trees such as this one, the number of branches at each node is not the number of moves that a player can make (as in a conventional game tree) but instead is the number of moves that correspond to the stratagems that a player might use. As shown in Table 23.1, such trees are small enough that Tignum 2 can search them all the way to the end to predict the likely results of the various sequences of cards that the players might play.

For each branch of the game tree, Tignum 2 uses its belief functions to estimate the probability that the branch will occur. For example, because the probability is 0.9854 that North holds at least one "low" spade—i.e., at least one spade other than the ♠Q—and because North is sure to play a low spade if North has one, the belief function generates a probability of 0.9854 for North's play of a low spade. North's other two possible plays are much less likely and receive much lower probabilities. Tignum 2 uses these probabilities to calculate values similar to minimax values.

To evaluate the game tree at nodes where it is an opponent's turn to play a card, Tignum 2 takes a weighted average of the node's children. For example, at the node where West has just played the 2♠, the minimax value is $265 \times .9854 + .0078 \times 630 + .0078 \times 600 = 270.73$.

To evaluate the game tree at nodes where it is the declarer's turn to play a card, Tignum 2 chooses the play that results in the highest score. For example, in Figure 23.4, West chooses to play the ♠A that resulted from the cashing out method, which results in a score of +600, rather than the ♠2 that resulted from the finesse method, which results in a score of +270.73.

23.5 Implementation and Results

In 1997, an implementation of the Tignum 2 procedure for declarer play was incorporated into Bridge Baron, an existing computer program for the game of bridge. This version of the Tignum 2 code contained 91 HTN task names and at least 400 HTN methods.

In comparison with other computer bridge programs, Bridge Baron was already a strong player: it had already won four international computer bridge championships. However, experimental studies showed that the Tignum 2 code significantly improved Bridge Baron's declarer play. With the help of the Tignum 2 code, Bridge Baron went on to win the 1997 world championship of computer bridge [495], and this was reported in several major newspapers [116, 513]. As of this writing, the Tignum 2 code (with subsequent enhancements over the years) has been used in six commercially released versions of Bridge Baron: versions 8 through 13.