Building a Fast Double-Dummy Bridge Solver

Ming-Sheng Chang – PhD student
Department of Computer Science
Courant Institute of Mathematics Sciences
New York University *
changm@cs.nyu.edu

August 1, 1996

Abstract

Compared to other games, particularly chess, the research in computer bridge is immature, and the best bridge-playing programs are mediocre. In this paper we address the problem of designing a fast double-dummy bridge game (i.e., a simplified bridge game with perfect information) solver. Although th size of the game tree we generated for searching the best line of play is huge (about on the order of $13! \cdot 2^{39} \approx 10^{21}$, even if we assume the average branching factor for players to follow suit is just 2), we show that , through varieties of searching techniques and some efficient moves ordering and pruning heuristics, most double-dummy bridge hands can be solved within a reasonable amount of time. In this paper we first give a brief introduction to computer bridge and previous work on the card-playing phase of bridge. Next, we describe the top-level architecture of our double-dummy solver (dds), followed by a number of implementation techniques we employed in our dds. Finally we present experimental results, draw our conclusion and describe some future work toward automating card-playing in real bridge.

*This research is supported in part by NSF grant #IRI-9300446

1 Introduction

Computer bridge is an interesting and challenging topic for AI researchers. From the perspective of AI, bridge is a game of playing with imperfect information in both bidding and card-playing stages. During the course of bidding, with only 13 out of 52 cards known and previous bidding history, a decent program might need the abilities to represent the knowledge of bidding methods and conventions, gather the clues from both his partner and opponents, encode and decode the information he or his partner needs in current bid, disguise his hand from his opponents whenever possible, evaluate all possible final contracts and plan to reach a contract to his team's most advantage. When it comes to the course of card-playing, with half of the cards still unseen initially, the program should evaluate and select the best plan out of a set of possible lines of play, extract useful information from the cards just been played, and possess the ability of revising its plan in the light of new information. Unfortunately, all these tasks are difficult even for average human players and seem hard for computers to model and handle well.

In our research to date, we have skipped the bidding phase of computer bridge and have assumed perfect information in the card-playing phase. Thus, in this paper we address the problem of solving double-dummy bridge; Under the assumption of perfect information (or double-dummy) and optimal play (all players play in a mini-max way), our double-dummy solver should be able to suggest a card for the current player to make the maximal number of tricks for his team. Note that double-dummy bridge can be considered as a two-person (the declarer and the defender) adversary game of perfect information — a category of games where the alpha-beta search techniques have been successfully applied to.

Before we go into the details of our dds implementation, let us give a brief review of previous suggested approaches to the card-playing phase of computer bridge.

- By adapting the techniques of automated mathematical theorem proving to declarative card-playing in bridge, Frank developed a system call FINESSE. In his system, the basic elements that make up plans are *methods* instead of cards. Each method corresponds to a common human player's tactic (e.g.

finesse, cash, duck) and is expressed as a structure with slots specifying the preconditions and postconditions of its associated tactic. To construct a plan, FINESSE tries to combine methods by reasoning about their specification. In the domain of declarative planning for a single suit, they claim that FINESSE can suggest the proper lines of play for many example hands bridge texts and support its decisions with probabilistic and qualitative information [1, 2].

- Gamback proposed the idea of using AI planning methods to construct a global plan through merging sub-plans that are obtained for each individual suit by brute-force [3].

- Ginsberg [4] and Levy [5] make similar proposals for card-playing. Suppose we have a fast double-dummy bridge games solver. Then, in real life bridge (with only one dummy — NORTH), a reasonable line of play may be constructed by selecting the plan that works best on average over a number of hands that we generate according to the bidding history and the opening lead.

Although double-dummy bridge is not real bridge, and the approach to producing a competent plan by "averaging over" the plans for a number of double-dummy hands is ambiguous and questionable, we consider the task of designing a fast double-dummy solver as important to the research of computer bridge for the following reasons:

- A fast dds might not only help us in the phase of card-playing but also in the phase of bidding — when evaluating different contracts we might reach, a fast dds tells us the reachability of a contract for any guess of real cards distribution.

- Due to its huge search space, it is challenging and worth our effort to try reducing the search space to a reasonable level for today's computing power.

- A fast dds can itself be a valuable tool to justify some common sense among bridge players through statistical simulation.

- Double-dummy bridge is itself a simplified version of real bridge. If we couldn't solve it within reasonable amount of time, then how could we expect an expert-level card-playing program?

- The best way to justify Levy's or Ginsberg's proposed mechanism of card-playing in computer bridge is to implement and test it.

2 Top-level architecture

Before we present the techniques employed in our dds, let us take a look at a primitive implementation of its top-level structure, which is basically Scout [7] search. Our algorithm is described using C++.

```
ddsearch(struct state *sp, int goal){
   if (goal<= 0) return 1;
   if (goal>tricks_left) return 0;
   if (tricks_left==1) return LastTrick(sp);
   GenerateMoves(sp);
   while ((card=NextMove(sp)) != EMPTY) {
      PlayCard(sp, card);
      result = sp->switch_team? !ddsearch(sp+1,...):
          ddsearch(sp+1,...);
      UnPlayCard();
      if (result) return 1;
   }
   return 0;
}
```

Figure 1: A primitive implementation of double-dummy search

Given parameter *sp* which points to a structure maintaining current state information and parameter *goal* which is a number of tricks, function *ddsearch* returns 1 if the team of current player could make at least *goal* tricks, otherwise, it returns 0. Basically, it works as follows: When *goal* is some trivial value like 0 or a number greater than total tricks left, it returns 1 or 0. When only 4 cards are left ($tricks\_left == 1$), function $LastTrick$ returns 1 if the team of current player could win the last trick, else it returns 0. If none of the above condition holds, we call $GenerateMoves$ to generate a list of legal moves M for current game state. Each time $NextMove$ is invoked, it retrieves the next available move from M. We call $PlayCard$ to make the selected move and update current game state. After a card is played, field *switch_team* in current state will be set to 1 if the team of next player is different from the team of current player, it's set to 0 otherwise. We then call *ddsearch* recursively with the next game state and the updated goal as parameters. Variable *result* indicates whether playing the selected *card* can make at least *goal* tricks or not. If *result* is 1, then we skip all the remaining moves and return 1, otherwise, we unplay the selected *card* and try the next move.

The code in figure 2, basically Scout search, computes the maximal trick for the team of current player.

```
for(low=0,high=total_tricks+1;low+1<high;){
     goal=(low+high)/2;
     if (ddsearch(sp,goal)) low=goal; else high=goal;
}
MaximalTricks=low;
```

Figure 2: A primitive top-level architecture of dds

In the code of figure 2, the condition $low \leq MaximalTricks < high$ always holds and the interval between *low* and *high* decreases in half after each iteration. Thus, $MaximalTricks$ is equal to *low* when the loop is finished. In 13-trick bridge game, solving games with $MaximalTricks$ equal to 0 or 7 requires calling function *ddsearch* 3 times. Otherwise, it will be necessary to execute *ddsearch* four times.

We will present a number of further improvements to this primitive algorithm in subsequent sections. But before we end this section, let us look at a simple but effective improvement — checking the existence of quick trick in function *ddsearch*. The main idea is: we could skip any further search and return 1 if *goal* equals 1 and the side to play would win at least one trick; Or similarly, we could return 0 if *goal* equals $tricks\_left$ and the side of opponent could win at least one trick. In our experiment conducted on 10-trick bridge games, the use of quick trick checking reduces over 50 percent of expanded nodes (or the number of times function *ddsearch* is called) on average.

3 Implementation of hash table

In computer chess, the hash table has been widely used and has proved helpful in pruning expanded nodes [6]. From the codes in previous section, we see that re-visits of some nodes that have been searched before (in current or previous call of *ddsearch*) are common. Thus, using a hash table to store the lower and higher bounds of the maximal trick for nodes frequently visited or nodes with larger depth (or $tricks\_left$) in the search tree may be promising in reducing the size of search tree.[1]

Let us describe some relevant issues before we go into details of implementation. For any two nodes (or game states) with the same current player in a search tree, what are the conditions that these two nodes would have the same value of maximal trick? It is easily seen that the following conditions suffice:

- the same cards have been played in the two nodes, and
- the trick in progress has the same leading suit and the same current winning card.

We have two methods of performing hash table checking to satisfy the above conditions:

1. The hash table checking is only performed at the beginning of a new trick.
2. The hash table checking is performed at each time *ddsearch* is called, but some extra information about the current trick (say, leading suit and current winning card) should also be stored and checked.

We adopt the first approach in our experiments since it is slightly better in terms of the number of expanded nodes and total spent cpu time. In the second method, the extra hash table entries we store for checking lower and higher bounds usually provide us with information similar to their parent nodes, which are already stored when the current trick is started. But these extra information are hardly used (if they could potentially cause some cut-offs, their corresponding parent nodes may already be stored and serve us the same purpose) and might take up the space for

[1]Special thanks to Thomas Anantharaman for his valuable suggestions on using hash table

more useful nodes.

We implement the hash table as follows. Our hash table contains 2^{20} entries; each entry has 8 bytes. For each bridge game, we split all 52 cards into two disjoint sets: set A contains the top 5 cards in each suit; set B contains the bottom 8. We also distinguish a third set C, which has the top 5 cards still remaining in each suit in B. Let $Bits$ be a function that map a set of cards into a sequence of bits, with each bit corresponds to each individual card, to show which cards in this set have already been played in current state. For each bit, value 1 means 'card is played' and 0 means the opposite. Thus, $Bits(A) = Bits(B) = Bits(C) = 0$ holds initially. We compute the value of $Bits(A) Xor Bits(C)$ as our hashkey, which is the index to hash table. Here, we borrow the idea of using xor operation from computer chess to make the hashkey more evenly distributed over the hash table. For the purpose of identifying game states, we store $Bits(B)$ and the current player in each entry of the hash table. Note that the combination of our hashkey and $Bits(B)$ uniquely identify all cards that have been played so far.

We rotate $Bits(C)$ 5 bits to the left before performing xor operation to obtain the hashkey. This bits rotation has the effect of making hashkey more evenly distributed over the hash table on our experiments.

After entering function $ddsearch$, there are 3 possible results of hash table checking:

- The corresponding hash table entry is empty. The only action to take for hash table is to store the new lower and higher bound at the end of the search.
- The current game state was visited before (hash hit). We check if the stored lower bound is not less than $goal$ or the stored higher bound is less than $goal$. If it is true, then we can return 1 or 0 without further search, otherwise, we continue the search and update the stored lower or higher bounds.
- The stored game state is different from the current one (hash collision). After finishing the search, we replace the stored state with current state only if the

field *tricks_left* of the stored state is not greater than that of the current state or a certain of number of hash collisions has occurred in this hash entry.

To further improve hits/collisions rate, we also try 2,4,8 and 16 way rehashing. For illustration, we take 4 way rehashing as an example. In our 2^{20} hash table entries, every 4 consecutive entries is grouped together. To check hash table, we select the 18 most significant bits out of the 20 bit hashkey to access the corresponding 4-entry group and check the 4 entries in succession. Similarly, when inserting the search result of current state, we look for the entry with the lowest depth (and highest number of hash collisions to solve tie) in the corresponding group if all 4 entries are occupied. In our experiment, 8 way rehashing outperforms others with its smallest spent cpu time. Though 16 way rehashing has slightly better *hits/collisions* ratio and smaller number of expanded nodes than those of 8 way rehashing, its associated overhead in hash table checking and inserting wore them away.

4 Lower bound estimation and Single suit analysis

When human bridge players plan their best line of play for the whole hand, they often first consider play and the number of winners and losers in each individual suit, and then proceed to form a global plan for the whole hand. Although there have been two research groups [3, 2] claiming to start writing a bridge card-playing program using this idea, up to now, no results are published yet. Moreover, the idea of using the results of single suit analysis in double-dummy search seems not yet been proposed before. In this section we first present an example hand, and then describe how we apply the analysis results to our dds and implement the single-suit analyzer.

An Example Hand[8]

3NT	♠ A7	West leads ♠Q
	♡ 864	
	♢ KQJT3	
	♣ 732	
♠ QJT53	N	♠ 964
♡ QT92	W E	♡ J75
♢ 92	S	♢ A87
♣ JT		♣ KQ64
	♠ K82	
	♡ AK3	
	♢ 654	
	♣ A985	

We assume no trump contract for simplicity and compute the following for each individual suit and possible leading player.

sure tricks the maximal tricks that the side of leading player can win in a row without discards or losing control to the opponent.

long-suit tricks after cashing sure tricks, long-suit trick is the number of tricks that the side of leading player might win while the partner of the winning player discards.

controls information about who gains control after cashing sure tricks or long-suit tricks if any.

In the example hand above, North/South have 2, 2 and 1 sure tricks for suits spade, heart, and club, respectively. Suppose cards ♢4,2,T, and A were played before and it is now South's turn, then N/S would have 2 sure tricks and 2 long-suit tricks in diamond suit, and the control shifts to the hand of North after cashing sure tricks or long-suit tricks.

With the information above from single-suit analysis, we can estimate a lower bound of maximal tricks for the team of current player in current game state. This helps us in two places of our previous code:

- Instead of using 0 as the initial lower bound value in figure 2, we may use a larger value of lower bound to reduce the times of executing function *ddsearch* from four (for most 13-trick bridge games) to a smaller number.

- Before searching further in function *ddsearch*, we can perform lower bound estimation to see if our estimate is already not less than *goal* when starting a trick or playing right after the leading card in current trick. Because the strength of cards normally fall on certain side, the time we choose to perform lower bound estimation makes the stronger side can have a large lower bound estimation earlier (which might result in earlier cut-offs) in case they don't start the current trick.

Since a sharp lower bound estimation is time-consuming even if we are given detailed results of single-suit analysis, our estimator is pretty simple: the sum of sure tricks in each individual suit plus the maximal long-suit tricks. For example, after the opening lead of West in our example hand, our estimator shows that S/N can win at least 5 tricks (5 sure tricks from suits spade, heart, and club plus 0 long-suit trick).

It is now time to state how we implement our single-suit analyzer. In terms of structure, it is similar to function *ddsearch*, with only slight differences in the way we identify and evaluate the leaf nodes. To save the time needed for executing single-suit analysis, we precompute and save onto a table the analysis results of all possible relative ranks distributions and leading players for all lengths of suit not greater than 9. For suit of length 9, we need 2^{18} different values to index all possible relative ranks distributions, where each relative rank is represented by 2 bits denoting the player who owns it. If we use 2 bytes to store the analysis result, then the memory space needed in suit of length 9 for storing all possible ranks distributions(2^{18} cases) and leading players(4 cases) is 2^{21} or 2 Mega bytes. The total memory space for suits of length not greater than 9 is less than 3 Mega bytes.

5 Moves generation: ordering and pruning

Moves generation heuristic plays a central role in game tree searching. In this section we assume no trump contract for simplicity and describe how we perform moves ordering and pruning on our dds.

Here is a list of moves pruning heuristics we may apply to generate moves in a certain suit X for current player P:

1. For a set of legal cards in suit X with consecutive relative ranks among all cards not yet been played, only one of them should be considered since they all have the same effect.
2. All spot cards (i.e., cards with low relative ranks) can be approximately considered as the same move and only one of them is selected since small cards seldom make a trick during the play.
3. If even the highest rank of legal cards of player P is still lower than the lowest rank of another player, then we try only the smallest card for player P.
4. Among all cards that are impossible to win the current winning card, only the one with smallest rank is selected.

All these heuristics except the first one are approximate and we adopt some of them on our dds. Since any double-dummy bridge hand is only a guess of the real cards distribution, it makes sense to pursue only near-optimal solution.

After applying some moves pruning heuristics to generate a list of cards L in certain suit, we divide L into three categories:

winner These are cards that can win the current trick.

loser These are cards that cannot win the current trick.

unknown These are cards that may win or lose, depending on how later players in the current trick respond.

The task of ordering these 3 categories is complicated and dynamic by nature. Some information of current state required for efficient ordering are:

- the relative playing order of current player in current trick
- player who drew the current winning card (potential winner)
- entries information of all suits (provided by single-suit analysis)

Here are two examples. Let South be the last player of current trick, then he may prefer trying *loser* earlier than *winner* if North is the potential winner, or in the opposite order if his opponent is the potential winner. Another example comes from our previous example hand. After West played ♠Q, North have two moves: ♠A(*winner*) and ♠7(*loser*). Since most of the sure tricks of S/N are in South's hand, North might prefer playing *loser* first to reserve his only entry. Note that it happens to be the correct play for this hand. Our current implementation of these orderings is still primitive, and works toward building a table summarizing these highly domain-dependent ordering heuristics are just begun.

There are situations the current player P can select his move from a set of suits. We present our suits ordering heuristics on two different cases.

1. When player P starts a trick, our criteria for ordering suits are
   - the more sure tricks and long-suit tricks the side of player P has in a suit, the higher its priority
   - the more sure tricks and long-suit tricks the side of opponent has in a suit, the lower its priority
2. When player P discards, our criteria for ordering suits are
   - the more sure tricks and long-suit tricks the side of player P has in a suit, the lower its priority
   - the more sure tricks and long-suit tricks the side of opponent has in a suit, the higher its priority
   - the longer of the combined length the side of player P has in a suit, the higher its priority

After sorting all suits, we need to decide the total ordering of cards selected from each suit. Instead of trying cards suit by suit, we try only the best card from each suit in the first few moves and sort them by the ordering of suits, then we try all the rest cards, with the ordering of suit dominates the ordering of ranks. This can prevent us from selecting the right suit as the last move in case the ordering of suits is bad.

6 Table lookup techniques

Information that is often used in search and does not take too much space to store should be stored in a table. These information may be static (i.e., independent of the bridge hands we are solving) or dynamic. If static, information can be precomputed and kept in a file. Dynamic information can be stored when first computed, then retrieved when needed again. Although table lookup techniques don't help in reducing the expanded nodes, they do give the program a constant speedup. Here are some information suitable for table lookup. The last two are dynamic, others are static.

absolute and relative ranks conversion table Given 13 bits representing the current state of a certain suit and 4 bits representing an absolute (or relative) rank, it returns the corresponding relative (or absolute) rank.

single-suit analysis results The results of single-suit analysis are helpful in estimating lower bounds and generating moves. As already mentioned, we precompute and store the analysis results of all possible relative ranks combinations for those with length of suit not greater than 9.

bits mask for generating hash keys Once we have decided which cards are used for setting the hashkey and their corresponding bit positions in hashkey, we can calculate a bits mask for updating the hashkey once the suit and absolute rank of the card to be played are known.

moves table Given the current state of a suit X (a 13-bit representation), we can store the available moves in suit X for current player after applying the moves pruning heuristics onto a table for future lookup. Our experiment conducted on 10 randomly generated 13-trick bridge games shows that over 98 percent of these moves information are obtained through table lookup.

7 Experiments

To see how much the ideas we presented in previous sections can help in our dds, we test them using 10 randomly generated no trump 12-trick bridge games on Sparc-10 workstation and measure the following:

- average of expanded nodes (both internal and leaf nodes) — N.
- average of leaf nodes (or times function $LastTrick$ is called) — L.
- average spent cpu time in sec — C.

The codes we tested are based on combinations of the following methods:

Q quick trick checking

H hash table with 8 way rehashing

H1 hash table without using rehashing techniques

L lower bound estimation

M codes with suits ordering

M1 codes with arbitrary suits ordering

T table lookup technique

Here is a table summarizing some of our experimental results.

Methods	N	L	C
M1	2.51E7	1.39E6	400
Q+H1+M1	2.8E6	1.2E5	216
Q+H1+M1+T	2.8E6	1.2E5	54.7
Q+H1+M+T	3.91E5	1.02E4	9.57
Q+H1+L+M+T	2.32E5	3.18E3	7.17
Q+H+L+M+T	1.54E5	1.21E3	5.15

To sum up, for the randomly generated 10 no-trump contract hands, our current version of dds can averagely solve 12-trick games in 5 sec and 13-trick games in 15 sec on Sparc-10.

8 Conclusion

Although th size of the search tree is potentially huge, in this paper we show that the task of solving 13-trick double-dummy bridge games can be done within reasonable time using accessible computing power (on Sparc-10). Except the idea of using single-suit analysis results, techniques we employed in our implementation are all currently available. Below we summarize some future directions toward automating card-playing in real bridge.

- implement a lower bound estimator for suit contracts
- implement a better dynamic orderings of cards in the classes *winner, loser,* and *unknown* as we mentioned in the section about moves generation
- incorporate to our system the ability of dealing with uncertainty in real bridge
- test Levy's [5] or Ginsberg's [4] proposed card-playing mechanism in real bridge

References

[1] Ian Frank. An adaptation of proof-planning techniques to declarer play in the game of bridge. Master's thesis, Depart. of AI, Edinburgh, 1991.

[2] Ian Frank, D. Basin, and A. Bundy. An adaptation of proof-planning to declarer play in bridge. In *Proc. of the European Conf. on AI*, 1992.

[3] B. Gamback, M. Rayner, and B. Pell. Pragmatic reasoning in bridge. Technical Report 299, Univ. of Cambridge, Computer Lab., 1993.

[4] Matt. Ginsberg. How computer will play bridge. Newsgroup rec.games.bridge, 1995.

[5] N.L. Levy and D.F. Beal, editors. *Heuristic Programming in Artificial Intelligence - The First Computer Olympiad.* Ellis Horwood Ltd. Chichester, 1989.

[6] T.A. Marsland. A review of game-tree pruning. In *ICCA Journal*, pages 3–19, March 1988.

[7] J. Pearl. Asymptotic properties of minimax trees and game searching procedures. *Artificial Intelligence*, 14:113–138, 1980.

[8] D. Roth. *The Expert Improver*. Collins Willow, 1992.