Bo Haglund, Soren Hein
DDS v2.8, 2014-11-18

Search Algorithms for a Bridge Double Dummy Solver

This description is intended for anyone interested in the inner workings of a bridge double dummy solver (DDS). It contains Bo's description from 2010 together with some updates by Soren. "I", "me", "my" in the text refers to Bo.

DDS algorithm descriptions already exist – see the reference list at the end. However, to my knowledge, no document exists that gives an in-depth description of all algorithms used

1. The basic search algorithm

The search is based on the zero window search [Pearl 1980]. Pseudo code for its application on DD solver search is given. Cards searched are described as "moves" in contrast to cards that are really played.

```
int  Search(posPoint, target, depth)
{
    if (depth==0) {
        tricks=Evaluate;
        return (tricks >= target ? TRUE : FALSE);
    }
    else
    {
        GenerateMoves;
        if  (player_side_to_move) {
            value=FALSE;   moveExists=TRUE;
            while (moveExists) {
                Make;
                value=Search(posPoint, target, depth-1);
                Undo;
                if  (value==TRUE) // Cutoff, current move recorded as "best move"
                    goto searchExit;
            }
        }  // Opponents to move
        else
        {
             value=TRUE;   moveExists=TRUE;
             while (moveExists) {
                 Make;
                 value=Search(posPoint, target, depth-1);
                 Undo;
                 if  (value==FALSE) // Cutoff, current move recorded as "best"
                     goto searchExit;
             }
        }
    }

    searchExit: return value;
}
```

The `Search` parameters are:

- **`posPoint`** - a pointer to a structure containing state information for the position (deal) to be searched, e.g. leading hand, hand-to-play, cards yet to play etc.
- **`target`** - the number of tricks the player must take.
- **`depth`** - the current search depth.

`Search` returns `TRUE` if the target is reached, otherwise `FALSE`.

When `Search` is called, **`depth`** is set to the number of cards left to play minus 4.

`GenerateMoves` generates a list of alternative moves (=cards) that can be played in the initial position whose state data is pointed to by **`posPoint`**. For cards that are equivalent (e.g. AK), only the card with highest rank is generated. Card equivalence is reanalyzed after each trick. So if the hand-to-play has AQ in a suit where K was played in a previous trick, then A and Q become equivalents.

If the side of the player has the move, `Search` tries to find a move that meets the target, i.e that evaluates to `TRUE`. If such a move is found, search returns `TRUE`, and saves the move as ”best”. If the other side has the move, `Search` tries to find a move that prevents meeting the target, i.e. that evaluates to `FALSE`. If such a move is found, search returns `FALSE`, and saves the move as ”best”.

Each move in the generated move list is handled by first calling `Make`, which generates a new move and removes the card from the position state information. `Search` is then recursively called with a position state that excludes the played card; **`depth`** has been decremented by one. For each new recursive call to `Search`, a card is removed from the position state information and **`depth`** is decremented. This goes on until **`depth`** equals 0, in which case only one trick remains. The outcome of this trick is calculated by `Evaluate`.

If the total number of tricks won by the side of the player reaches **`target`**, `Search` returns `TRUE`, otherwise `FALSE`. This result propagates upwards as `Search` returns for each level, `Undo` is called which reinstalls the searched card on this level. Finally, `Search` returns to the top level.

This basic search algorithm is not powerful enough to terminate the search of a typical 52 cards deal in a reasonable time. To accomplish this, a number of search algorithm enhancements are required, which will be described in the following chapters.

The described search algorithm only determines whether a predefined target can be reached. It does not say how many tricks that the side of the player can take. This is accomplished by repeated calls to `Search`:

```
g          = guessed number of tricks for side of the player
iniDepth   = number of cards to play minus 4
upperbound = 13;
lowerbound = 0;
do  {
    if  (g==lowerbound)
        tricks=g+1;
    else
        tricks=g;
    if  ((Search(posPoint, tricks, iniDepth)==FALSE)  {
        upperbound=tricks-1;
        g=upperbound;
    }
    else  {
        lowerbound=tricks;
        g=lowerbound;
    }
}
while (lowerbound < upperbound);
g=maximum tricks to be won by side of player.
```

2. Overview of the search algorithms used in the DD solver

The additional functions in the pseudo code for supporting the search speed enhancements are given in ***bold italics***.

```
int  Search(posPoint, target, depth) {
      if (no_move_yet_in_trick)  {
          TargetTooLowOrHigh;
          if (target_already_obtained)
              return TRUE;
          else if (target_can_no_longer_be_obtained)
              return FALSE;
          QuickTricks;
          LaterTricks;
          if  (cutoff_for_player_side)
             return TRUE;
          else if  (cutoff_for_opponent_side)
             return FALSE;
          RetrieveTTresult;
          if (transposition_table_entry_match) {
             if  (target_reached)
                return TRUE;
            else
                return FALSE;
          }
      }
```

```
    if (depth==0) {
        evalRes=Evaluate;
        if  (evalRes.tricks >= target)
           value=TRUE;
       else
           value=FALSE;
       return value;
    }
    else {
        GenerateMoves;
        MoveOrdering;
        if  (player_side_to_move) {
            value=FALSE;    moveExists=TRUE;
            while (moveExists) {
               Make;
               value=Search(posPoint, target, depth-1);
               Undo;
               if  (value==TRUE)  {
                   MergeMoveData;
                   goto searchExit;
               }
               MergeAllMovesData;
               moveExists=NextMove;
           }
       }          /* Opponents to move */
       else {
           value=TRUE;   moveExists=TRUE;
           while (moveExists) {
               Make;
               value=Search(posPoint, target, depth-1);
               Undo;
               if  (value==FALSE)  {
                   MergeMoveData;
                   goto searchExit;
               }
               MergeAllMovesData;
               moveExists=NextMove;
           }
       }
    }
    searchExit:
    AddNewTTentry;
    return value;
}
```

TargetTooLowOrHigh checks the target value against the number of tricks currently won by the player's side against the number of tricks left to play. It is executed at the beginning of each trick, before any card has been played.

- If the number of currently won tricks by the player's side equals or exceeds target, `Search` returns `TRUE`.
  If number of currently won tricks by player's side plus tricks left to play is less than target `Search` returns `FALSE`.
- Since possible winning cards for the remaining tricks are irrelevant, no winning cards are backed up at cutoff termination.

TargetTooLowOrHigh search enhancement is described e.g. in [Chang].

QuickTricks determines whether the side to move can take one or more sure tricks. For example, if the hand to move has an Ace in an NT contract, at least one sure trick can be taken.

It is executed at the beginning of each trick, before any card has been played. A simple quick trick is also executed after the leading card of the trick is played. Assuming that the sure tricks are won by the side to move, then the conditions for search cutoff in *TargetTooLowOrHigh* are again tested to produce further search cutoffs.

When *QuickTricks* win by rank, they are backed up at cutoff termination. The detailed conditions for determination of sure tricks are described in Chapter 3.

The idea of *QuickTricks* is described e.g. in [Chang].

LaterTricks determines whether the opponents of the side to move can take one or more tricks at their turn or later in the play. It is also executed at the beginning of each trick and uses similar criteria for search cutoff as *Quicktricks*.

When *QuickTricks* win by rank, they are backed up at cutoff termination. For a detailed description, see Chapter 4.

RetrieveTTresult scans the set of positions in the transposition table to see if there is a match against the current position.

It is executed at the beginning of each trick, before any card has been played. After detection of a transposition table entry match, the winning ranks necessary in the remaining cards are backed up. For details see Chapter 6.

Evaluate returns **`evalResult`** which updates the position state information; it contains:

- **`evalResult.tricks`**, the number of tricks won by the side of the player, and
- **`evalResult.winRank`** which includes the card in the last trick that won by rank.

For example, if the last trick includes the spades A, Q, 9 and 3, **`evalResult.winRank`** returns the spade Ace. But if the last trick was won without a win by rank as for spade 5 (leading and winning card), heart A, heart Q, heart 5, no winning rank is returned.

Keeping record of cards that win by ranks and subsequently using this information to ignore ranks for other cards is discussed in the Partition Search concept invented by Matthew Ginsberg and described in his paper [Ginsberg].

MoveOrdering. The alternative cards created by *MoveGenerate* are sorted, with the cards most likely to terminate the search fastest being sorted first in the move list. The allocation of card weights is described in detail in Chapter 5.

MergeMoveData. At move search cutoff, *MergeMoveData* collects the union of the backed-up accumulated winning ranks and the rank of the made move, assuming it did win by rank. The state data of the position is updated with the collected information.

MergeAllMovesData collects the union of the backed-up accumulated winning ranks, the previously accumulated winning ranks of the alternative moves generated at this depth, and the rank of the made move, assuming it did win by rank. When all alternative moves have been searched without a cutoff, the state data of the position is updated with the collected information.

The information from *MergeMoveData* and *MergeAllMovesData* is later stored in the transposition table and determines which ranks that are essential when *RetrieveTTresult* scans the set of positions in the transposition table. A match of ranks with the current position is only needed for winning ranks. See Chapter 6.

AddNewTTentry adds the evaluated position as a new entry in the transposition table. See Chapter 6.

NextMove filters out all ”small” cards except one per hand/suit combination. A ”small” card is a backed up card that is shown to never win by rank. The rest of the ”small” card moves for the hand/suit combination are never searched, leading to a smaller search tree. This search enhancement was suggested by Hans Kuijf [Kuijf].

3. The Quick Tricks cutoff algorithm

The number of tricks that can immediately be taken by the side to play the leading card of the trick consists of:

a) The number of tricks that can be taken by the hand-to-play, and
b) The number of tricks that can be taken by the partner of the hand-to-play.

At return by `QuickTricks`, the position state information is updated with the winning ranks found.

Of course, in order to add b), there must be an entry from the hand-to-play to the partner’s hand.

For each ”**s**” (suit) the following is calculated:

If the hand-to-play is the only hand having cards in **s**, and the opponents have no trumps (when **s** is not trumps), the number of quick tricks for **s** is the suit length of the hand-to-play.

If the opponents have no trumps, a check is made to see if quick tricks equal to the maximum of the trumps length for leading hand and the partner causes a search cutoff.

If the hand-to-play has a card in a suit where the partner has a winning rank, and partner is the only hand having cards in **s**:
The number of quick tricks for s is the suit length of partner.

Else:
If the winning rank is in hand-to-play, and the opponents cannot ruff, the number of quick tricks is incremented by one. Further, if the second best rank is also in hand-to-play, and the opponents cannot still ruff, the quick tricks is again incremented by one.

Else:
If the winning rank is in partner and partner has winning rank as entry, the same applies for the partner as for the hand-to-play described above.

If it is a trump contract, the first suit to be investigated is the trump suit. Then if there are trump suit quick tricks for the side to play, those are cashed and quick tricks incremented accordingly.

When the other suits are investigated for quick tricks, only the remaining opponent trump cards need to be considered.

The quick tricks are then summarized from each suit, and the total calculated.

A simple `QuickTricks` algorithm is also executed after the leading card of the trick has been played:

A quick trick is gained either if the hand-to-play or the partner can win the current trick with the card having the highest rank of the suit played, or if hand-to-play or the partner can win the trick by ruffing.

The idea to also execute `QuickTricks` after the leading card has been played was given by Hans Kuijf [Kuijf].

4. The LaterTricks cutoff algorithm

Check for search cutoff if the opponents to the trick leading hand have at least a sure trick later.

If not trump contract:

1) The opponents have at least a sure trick if for all suits where the trick leading hand has a card,

the side of the leading hand does not have the highest rank.
More than one sure trick can be taken by the opponents if they possess the winning rank for more than one suit, or

2) Assume that all suits where the side of the trick leading hand has the winning rank give maximum possible number of tricks, i.e. that the sure trick number is the sum of the maximum lengths of these suits.
If this still cannot cause a cutoff for the trick leading side, allocate one sure trick for the opponents' side.

If trump contract:

Quick tricks for the opponents of the leading hand are added when the opponents have one or more winning trumps. This idea was given by Pedja Stanojevic [Stanojevic].

1) If the opponent side have all the trumps, the number of sure tricks is the maximum suit length, or

2) If the opponent side has the highest trump, they have 1 sure trick. If they also have the second-highest trump, they have 2 sure tricks, or

3) If the opponent side has the second highest trump plus at least one trump more behind the hand with the highest trump, the opponents' side has 1 sure trick.

5. The Move Ordering algorithm

The weight of a card in the move list is affected by the suit and the rank of the card and by the other cards in the same trick. The weights of the cards in the move list are used to sort them, with the cards having the highest weight being sorted first in the list.

If the hand-to-play is the trick-leading hand or is void in the suit played by leading hand, the card with the highest weight for each present suit will get a high additional bonus weight. After list resorting, those cards will occupy the first positions in the move list.

Two "best moves" are maintained for each searched depth, one for an alpha-beta cutoff and one at a Transposition Table entry match. At an alpha-beta cutoff, the move causing the cutoff overwrites the present "best move" for the current depth. When a Transposition Table entry is created, the current best move is stored in that entry if:

- The target is met and the leading hand belongs to the player's side,
- Or the target is not met and the leading hand belongs to the other side.
- Otherwise the best move is not stored in the Transposition Table entry.

At a Transposition Table entry match, its stored best move will be best move for the current search depth.

By ”card move” in the following pseudo code is meant the card by the hand-to-play that is getting a weight in the move list. The ”card rank” is a value in the range 2 (deuce) to 14 (Ace).

For the determination of the weight, it is calculated whether or not the current card move wins the current trick for the side of the hand-to-play, assuming that both sides play optimum cards.

The following pseudo-code contains empirical weights that are used to obtain move orderings that tend to put optimum move early in the list of moves. These may or may not be the exact weights and algorithms used in the current DDS version, but they give an idea of the important factors; the code is significantly more complex. One aim is to move the likely candidates to the top of the list, and another aim is to have good mixture of moves (i.e. not all cards from the same suit first) in case the heuristic is not good for a particular set-up.

If the hand-to-play is void in the trick lead suit, the suit selected for the discard gets a bonus:

```
suitAdd = ((suit length) * 64)/36;
```

If the suit length is 2, and the hand-to-play has the next highest rank of the suit, the bonus is reduced by 2.

Hand-to-play is trick-leading hand

The contribution of the suit to the weight is:

```
suitWeightDelta = suitBonus – ((countLH+countRH) * 32)/15
```

`suitBonus` has the initial value 0, changed if conditions below apply:

If it is a trump contract, and the suit is not trump, then there is a `suitBonus` change of –10 if

- LHO is void and LHO has trump card(s), or
- RHO is void and RHO has trump card(s).

If RHO has either the highest rank of the suit played by hand-to-play or the next highest rank, then there is a `suitBonus` change of –18.

If it is a trump contract, the suit is not trump, the own hand has a singleton, the own hand has at least one trump, partner has the highest rank in the suit and at least a suit length of 2, then there is a `suitBonus` change of +16. Suits are thus favoured where the opponents have as few move alternatives as possible.

`countLH = (suit length of LHO) * 4`, if LHO is not void in the suit,
`countLH = (depth + 4)`, if LHO is void in the suit
`countRH = (suit length of RHO) * 4`, if RHO is not void in the suit,
`countRH = (depth + 4)`, if RHO is void in the suit

```
if (trick winning card move) {
    if (one of the opponents has a singleton highest rank in the suit)
        weight = suitWeightDelta + 40 – (rank of card move)
    else if (hand-to-play has highest rank in suit)  {
        if (partner has second highest rank in suit)
            weight = suitWeightDelta + 50 – (rank of card move)
        else if  (the card move is the card with highest rank in the suit)
            weight = suitWeightDelta + 31
        else
            weight = suitWeightDelta + 19 – (rank of card move)
    }
    else if  (partner  has highest rank in suit)  {
        if (hand-to-play has second highest rank in suit)
            weight = suitWeightDelta + 50 – (rank of card move)
        else
            weight = suitWeightDelta + 35 – (rank of card move)
    }
    else if  (hand-to-play has second highest rank together with
              equivalent card(s) in suit)
        weight = suitWeightDelta + 40
    else
        weight = suitWeightDelta + 30 – (rank of card move)
    if  (the card move  is ”best move” as obtained at alpha-beta cutoff)
        weight = weight + 52;
    if  (the card move is ”best move” as obtained from the Transposition Table)
        weight = weight + 11;
}
else {     /* Not a trick winning move */
    if  (either LHO or RHO has singleton in suit which has highest rank)
        weight = suitWeightDelta + 29 – (rank of card move)
    else if (hand-to-play has highest rank in suit)  {
        if (partner has second highest rank in suit)
            weight = suitWeightDelta + 44 – (rank of card move)
        else if  (the card move is the card with highest rank in the suit)
            weight = suitWeightDelta + 25
        else
            weight = suitWeightDelta + 13 – (rank of card move)
    }
    else if  (partner  has highest rank in suit)  {
        if (hand-to-play has second highest rank in suit)
            weight = suitWeightDelta + 44 – (rank of card move)
        else
            weight = suitWeightDelta + 29 – (rank of card move)
    }
    else if  (hand-to-play has second highest rank together with
              equivalent card(s) in suit)
        weight = suitWeightDelta + 29
    else
        weight = suitWeightDelta + 13 – (rank of card move)
    if  (the card move  is ”best move” as obtained at alpha-beta cutoff)
        weight = weight + 20;
    if  (the card move  is ”best move” as obtained from the Transposition Table)
        weight = weight + 9;
}
```

Hand-to-play is left hand opponent (LHO) to leading hand

```
if (trick winning card move) {
    if  (hand-to-play void in the suit played by the leading hand)  {
        if  (trump contract and trump is equal to card move suit)
            weight = 30 - (rank of card move) + suitAdd
        else
            weight = 60 - (rank of card move) + suitAdd
    }
    else if (lowest card for partner to leading hand is higher than LHO played card)
        weight = 45 - (rank of card move)
    else if (RHO has a card in the leading suit higher than the leading card
             but lower than the highest rank of the leading hand)
        weight = 60 - (rank of card move)
    else if (LHO played card is higher than card played by the leading hand) {
        if (played card by LHO is lower than any card for RHO in the same suit)
            weight = 75 - (rank of card move)
        else if (LHO's card by LHO beats any card in that suit for the leading hand)
            weight = 70 - (rank of card move)
        else  {
            if  (LHO move card has at least one equivalent card) {
                weight = 60 - (rank of card move)
            else
                weight = 45 - (rank of card move)
        }
    }
    else if (RHO is not void in the suit played by the leading hand) {
        if  (LHO move card has at least one equivalent card)
            weight = 50 - (rank of card move)
        else
            weight = 45 - (rank of card move)
    }
    else
        weight = 45 - (rank of card move)
}
else {     // card move is not trick winning
    if  (hand-to-play void in the suit played by the leading hand)  {
        if  (trump contract and trump is equal to card move suit)
            weight = 15 - (rank of card move) + suitAdd
        else
            weight = - (rank of card move) + suitAdd
    }
    else if (lowest card for leader's partner or for RHO in the suit played is
             higher than played card for LHO)
        weight = - (rank of card move)
    else if (LHO played card is higher than card played by the leading hand) {
        if  (LHO move card has at least one equivalent card)
            weight = 20 - (rank of card move)
         else
             weight = 10 - (rank of card move)
     }
     else
         weight = - (rank of card move)
}
```

Hand-to-play is partner to trick leading hand

```
if (trick winning card move) {
    if  (hand-to-play void in the suit played by the leading hand)  {
        if (card played by the leading hand is highest so far) {
            if (card by hand-to-play is trump and the suit played by the
                leading hand is not trump)
                weight = 30 - (rank of card move) + suitAdd
            else
                weight = 60 - (rank of card move) + suitAdd
        }
        else if (hand-to-play is on top by ruffing)
            weight = 70 - (rank of card move) + suitAdd
        else if (hand-to-play discards a trump but still loses)
            weight = 15 - (rank of card move) + suitAdd
        else
            weight = 30 - (rank of card move) + suitAdd
    }
    else
        weight = 60 - (rank of card move)
}
else {               // card move is not trick winning
    if  (hand-to-play void in the suit played by the leading hand)  {
        if (hand-to-play is on top by ruffing)
            weight = 40 - (rank of card move) + suitAdd
        else if (hand-to-play underruffs */
            weight = -15 - (rank of card move) + suitAdd
        else
            weight = - (rank of card move) + suitAdd
    }
    else {
         if (the card by hand-to-play is highest so far) {
             if (rank of played card is second highest in the suit)
                weight = 25
             else if (hand-to-play card has at least one equivalent card)
                 weight = 20 - (rank of card move)
             else
                 weight = 10 - (rank of card move)
         }
         else
             weight = -10 - (rank of card move)
    }
}
```

Hand-to-play is right hand opponent (RHO) to leading hand

```
if  (hand-to-play is void in leading suit)  {
    if  (LHO has current highest rank of the trick)  {
       if  (card move ruffs)
          weight = 14- (rank of card move) + suitAdd
       else
          weight = 30- (rank of card move) + suitAdd
     }
    else if  (hand-to-play ruffs and wins)
         weight = 30- (rank of card move) + suitAdd
    else if  (card move suit is trump, but not winning)
        weight = - (rank of card move)
    else
        weight = 14- (rank of card move) + suitAdd
}
else if  (LHO has current winning move)  {
    if  (RHO ruffs LHO's winner)
        weight = 24 - (rank of card move)
    else
        weight = 30- (rank of card move)
}
else if  (card move superior to present winning move not by LHO)  {
     weight = 30- (rank of card move)
else  {
    if  (card move ruffs but still losing)
        weight = - (rank of card move)
    else
        weight = 14- (rank of card move)
}
```

6. Storage and retrieval of position state data in the Transposition Table

Positions stored in the Transposition Table always consist of completed tricks. Positions stored start at depth=4, then 8,12, and so on. The information stored is information on won cards, the suit lengths of the hands, the hand to play the leading card in the position and upper and lower bounds for the number of future tricks to be taken by the side of the player.

Starting from issue 1.1.8, each ”winning cards node” contain all winning cards for one suit after an idea by Joël Bradmetz. This new solution is faster.

6.1 Transposition Table storing winning card ranks

For the outcome of played tricks, only card ranks that are winning due to their ranks matter. Assume that the last two tricks of a deal without trumps looks like the following:

Trick 12: Leading hand North plays heart A, East, South and West follow with hearts Q, 9 and 7 respectively.
Trick 13: North then leads spade A, the other hands plays diamonds J, 8,3 in that order.

In trick 12, heart A wins by rank. In trick 13, spade A wins but not by rank.

The sequence of cards could have been the following without changing the outcome:
Trick 12: heart A, heart x, heart x, heart x
Trick 13: spade x, diamond x, diamond x, diamond x
where x is any rank below lowest winning rank.

The cards that win by rank are recorded during the search and backed up similarly to the search value. If a card wins by rank and there are equivalent cards, e.g. only spade A is searched from a sequence of AKQ, then also the other cards K and Q must be recorded as having won by rank.

The cards winning by rank are stored in the Transposition Table as relative ranks, however any rank larger than the lowest winning rank in the suit is also stored as ”winning ranks”. Using relative ranks rather than absolute ranks considerably increases the number of positions that match this Transposition Table entry.

As an example, assume that there are only 4 cards left in a suit, A, Q, 9, 7 where each hand has one card in the suit. Then any combination of ranks, e.g. 8, 6, 3, 2 that preserves the relative order of ranks between hands will cause a match.

In the state position information absolute ranks are used; it is only in the Transposition Table where the ranks are stored as relatives.

6.2 Backing up the winning ranks

At the search termination, either at the last trick or at a cutoff, the cards that have won by rank are backed up in the search tree together with the search value. As this information propagates upwards, it is aggregated with backed up information from other tree branches. At a search cutoff, `MergeMoveData` merges the information (∪ is a union):

```
(winning ranks of all suits for current depth) =
  (winning ranks of all suits for depth - 1)  ∪
  (possible winning rank for the current move causing the cutoff)
```

For each new move not causing cutoff, *MergeAllMovesData* merges:

```
(winning ranks of all suits for current depth) =
   (winning ranks of all suits for current depth)  ∪
   (winning ranks of all suits for depth - 1)  ∪
   (possible winning rank for the current move)
```

6.3 Implementing the Transposition Table

The Transposition Table (TT) can be implemented in a number of ways. Indeed it is encapsulated as a C++ object, so it can be modified without further implications on DDS.

The basic functional requirement is that it must be possible to store nodes that each cover several

actual play positions, and it must be possible to query the TT with a specific play position. This is the purpose of keeping track of winning ranks.

At the time when a node is stored, the winning ranks for that node are known. At the time when the TT is queried, the position contains only actual cards, and indeed it could be that there are several nodes in the TT matching that position. One match will then be more detailed (contain more winning ranks) than the other.

A position can in principle be characterized and indexed in any way, but it seems practical to index first on the suit distribution and then on the actual suit cards.

For example, a starting position may consist of

- North holding 3=5=3=2 (3 spades, 5 hearts, 3 diamonds and 2 clubs),
- East holding 4=4=3=2,
- South holding 1=2=4=6, and
- West holding the remaining 5=2=3=3.

We already know that each player starts with 13 cards (or whatever number of tricks is still left to play at the time), so for each player we can leave out the number of one suit, say clubs. We can encode this rather loosely with 4 bits per suit, so 12 bits per player. This yields 48 bits for the whole hand. It would also be possible to enumerate the actual distributions more carefully, but there are a lot of them. Of course a direct 48-bit or even 32-bit index is impractically large.

In terms of precise cards, the absolute holdings must be converted into relative ones. So if the players collective still hold the KQT9542, this would be considered the same as AKQJT98, relatively speaking. Otherwise we don't get nearly enough matches in the TT.

Furthermore, winning ranks must be taken into account. If only the top 3 ranks matter in this particular example, then that suit would be considered to hold AKQxxxx.

With this general information, we now describe the data structure that is currently used in DDS.

Storage

In earlier DDS version, the suit distribution was stored in a binary tree with a 48-bit key. This caused a certain amount of hopping around in memory. The current structure consists of (a) a hash from 48 bits to 8 bits, followed by (b) a flat list that is searched linearly. For most hands the number of hash collisions is small. Effectively we trade some storage space for speed.

Specific holdings in a suit are characterized by (a) a bit vector of the (relative) ranks, with 0..3 representing the players North..West, so 2 bits per (relative) rank, and (b) a mask bit vector. If only the top 3 ranks matter, then only the top 6 bits of the mask vector are non-zero.

Once the exact suit distribution has been found, there is another flat list with a fixed number of elements (currently 125) corresponding to specific hands. If we run out of space, we overwrite

the list cyclically from the beginning. This too was earlier a tree, leading to the same trade-off between space and speed.

Retrieval

The suit distribution is always exact, never approximate, so the look-up works as above.

The hand look-up could proceed player by player or suit by suit. It makes sense for each comparison to have a good number of bits (close to 32) in order to cut down on the number of comparisons. A whole suit can be encoded with 2 bits (player number) times 13 cards for a total of 26 bits. However, we might be unlucky and start with a suit with very few cards.

In the current implementation of DDS, we create a 32-bit vector corresponding to the top 4 cards ("Ace", "King", "Queen" and "Jack", relatively speaking) in each of the four suits. This is more symmetrical and more selective. If that matches, we compare the next four ranks, then the next four, and finally the deuce.

When we are looking up a specific holding, we first make these 32-bit vectors for specific holdings. Then for each position in the TT (for the given distribution), we read out the stored suit vectors and mask vectors. We apply the mask vectors to the given holdings and compare with the stored winning ranks.

If everything matches, we check the stored bounds to see whether the node causes a cut-off or not. The stored upper and lower value bounds are checked against the number of tricks won so far by the player's side and the target value. The following conditions are then checked, assuming that it is the North/South side that is the player's side:

If the sum of the stored lower value bound and the number of tricks won so far for the player's side is equal or larger than target, then target can be reached for the player's side in the current position. Search on this depth is terminated and `TRUE` is returned.

If the sum of the stored upper value bound and the number of tricks won so far for the player's side is less than target, then reaching target can be prevented by the opponents to the player's side in the current position. Search on this depth is terminated and `FALSE` is returned.

If instead it is East/West that is the player's side, the following conditions apply:

If the sum of number of tricks remaining and the number of tricks won so far for the player's side minus the upper value bound is equal or larger than target, then target can be reached for the player's side in the current position. Search on this depth is terminated and `TRUE` is returned.

If the sum of number of tricks remaining and the number of tricks won so far for the player's side minus the lower value bound is less than target, then reaching target can be prevented by the opponents to the player's side in the current position. Search on this depth is terminated and `FALSE` is returned.

For all other cases, the search continues for the current depth.

The usage of upper and lower value bounds in transposition tables is described in [Chang] and [Kupferschmid, Helmert].

When the value of the current position is known and it is the end of a trick (except the last), position state information is collected for storage in the Transposition Table. The ranks of the backed-up winning cards are converted from absolute to relative.

For each suit, it is determined which winning rank that is lowest. The relative ranks then stored in the new Transposition Table entry are all ranks above and including the lowest rank, filling out any ”holes” in the ranks that might have been present.

References

James Dow Allen:
Source code for a simple DDS.
http://freepages.genealogy.rootsweb.com/~jamesdow/Tech/dbldum.htm

Matthias Brill:
DDS algorithms description (in German) and DDS source code.
http://linux.softpedia.com/get/Science-and-Engineering/Artificial-Intelligence/cddsolve-20055.shtml

Ming-Sheng Chang:
DDS algorithms description.
http://cs.nyu.edu/web/Research/TechReports/TR1996-725/TR1996-725.ps.gz

Ed Colley:
DDS source code and DDS executable.
http://freefinesse.sourceforge.net/

Matthew L. Ginsberg:
DDS algorithms description.
http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume14/ginsberg01a.pdf

Dan Hirschberg:
DDS algorithms description and DDS executable (MS DOS, cannot run in XP?)
http://www.ics.uci.edu/~dan/bridge/index.html

Alexey Slovesnov:
DDS source code and DDS executable.

Judea Pearl: Asymptotic properties of minimax trees and game search precedures.
Artificial Intelligence 14(2):113-138. [Pearl 1980]

Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie de Bruin: Exploiting graph properties of game trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 234-239, 1996 [Plaat et al.]

Hans Kuijf, personal communication.

Pedja Stanojevic, personal communication.

Sebastian Kupferschmid, Malte Helmert: A Skat Player Based on Monte Carlo Simulation.

Joël Bradmetz, personal communication.
http://jibe-bridge.perso.cegetel.net/