The $\alpha\mu$ Search Algorithm for the Game of Bridge

Tristan Cazenave [1] and **Véronique Ventos** [2]

Abstract. $\alpha\mu$ is an anytime heuristic search algorithm for incomplete information games that assumes perfect information for the opponents. $\alpha\mu$ addresses the strategy fusion and non-locality problems encountered by Perfect Information Monte Carlo sampling. In this paper $\alpha\mu$ is applied to the game of Bridge.

1 Introduction

As superhuman level has been reached for Go starting from zero knowledge [18] and as it is also the case for other two player complete information games such as Chess and Shogi [17] some of the next challenges in games are imperfect information games such as Bridge or Poker. Multiplayer Poker has been solved very recently [1] while computer Bridge programs are still not superhuman.

The state of the art for computer Bridge is Perfect Information Monte Carlo sampling (PIMC). It is a popular algorithm for imperfect information games. It was first proposed by Levy [13] for Bridge, and used in the popular program GIB [10]. PIMC can be used in other trick-taking card games such as Skat [2, 12], Spades and Hearts [19]. The best Bridge and Skat programs use PIMC. Long analyzed the reasons why PIMC is successful in these games [15].

However PIMC plays sub-optimally due to two main problems: strategy fusion and non locality. We will illustrate these problems in the second section. Frank and Basin [6] have proposed a heuristic algorithm to solve Bridge endgames that addresses the problems of strategy fusion and non locality for late endgames. The algorithm we propose is an improvement over the algorithm of Frank and Basin since it solves exactly the endgames instead of heuristically and since it can also be used in any state even if the search does not have enough time to reach the terminal states. Ginsberg has proposed to use a lattice and binary decision diagrams to improve the approach of Frank and Basin for solving Bridge endgames [10]. He states that he was generally able to solve 32 cards endings, but that the running times were increasing by two orders of magnitude as each additional card was added. $\alpha\mu$ is also able to solve Bridge endings but it can also give a heuristic answer at any time and for any number of cards and adding cards or searching deeper does not increase as much the running time.

Furtak has proposed recursive Monte Carlo search for Skat [8] to improve on PIMC but the algorithm does not give exact results in the endgame and does not solve the non locality problem.

Other approaches to imperfect information games are Information Set Monte Carlo Tree Search [3], counterfactual regret minimization [21], and Exploitability Descent [14].

$\alpha\mu$ searches with partial orders. It is related to partial order bounding [16] and to searching game trees with vectors of integer values [4]. However our algorithm is different from these algorithms since it searches over vectors only composed of 0 and 1 and uses different backups for sets of vectors at Max and Min nodes as well as probabilities of winning.

The contributions of the paper are:

1. An anytime heuristic search algorithm that assumes Min players have perfect information and that improves on PIMC and previous related search algorithms.
2. An anytime solution to the strategy fusion problem of PIMC that solves the strategy fusion problem when given enough time.
3. An anytime solution to the non-locality problem of PIMC using Pareto fronts of vectors representing the outcomes for the different possible worlds. It also converges given enough time.
4. A search algorithm with Pareto fronts.
5. The description of the early and root cuts that speed up the search.
6. Adaptation of a transposition table to the algorithm so as to improve the search speed using iterative deepening.
7. Experimental results for the game of Bridge.

The paper is organized as follows: the second section deals with PIMC for computer Bridge and its associated defects. The third section defines vectors of outcomes and Pareto fronts. The fourth section deals with search with strategy fusion and non locality. The fifth section gives experimental results.

2 Perfect Information Monte Carlo Sampling

In this section we illustrate the problems of PIMC.

2.1 Double Dummy Solver

A very efficient Double Dummy Solver (DDS) has been written by Bo Haglund [11]. In our experiments we use it to evaluate double dummy hands. It makes use of partition search [9] among many other optimizations to improve the solving speed of the $\alpha\beta$.

2.2 Some Problems of PIMC

PIMC is the state of the art of computer Bridge, it is used for example in GIB [10] and in WBRIDGE5 [20] the current computer world champion. The PIMC algorithm is given in algorithm 1. In this algorithm S is the set of possible worlds and $allMoves$ is the set of moves to be evaluated. The play function plays a move in a possible world and returns the corresponding state. The doubleDummy function evaluates the state using a double dummy solver.

There are multiple problems with PIMC [15]. Here we will illustrates some problems for the declarer with a No Trump contract since our experiments use this restriction.

[1] LAMSADE Université Paris-Dauphine PSL CNRS, France, email: Tristan.Cazenave@dauphine.psl.eu
[2] NUKKAI, Paris, France, email: vventos@nukk.ai

Algorithm 1 The PIMC algorithm.

```
Function PIMC (allMoves, S)
  for move ∈ allMoves do
    score[move] ← 0
    for w ∈ S do
      s ← play (move, w)
      score[move] ← score[move]+ doubleDummy (s)
    end for
  end for
  return argmax_move(score[move])
```

♠ K J 10 7
♡ A K Q
♢ A K Q
♣ x x x

N

S

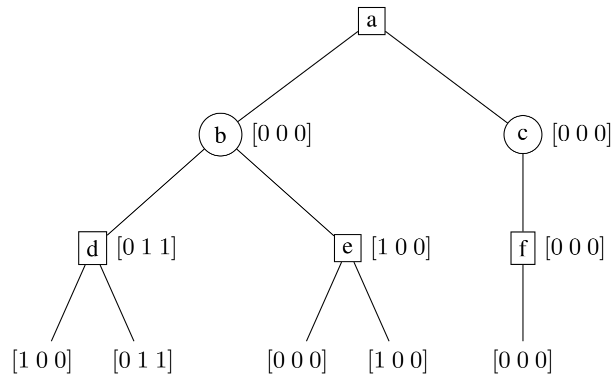
♠ A 9 8 6
♡ x x x
♢ x x x
♣ A K Q

In this hand from [10], PIMC finds that the declarer always makes all of the four tricks at Spades. This problem is known as strategy fusion [5]. The reason why PIMC misevaluates the hand is because it can play different cards in different worlds whereas it should play the same cards in all the worlds. Frank and Basin solve this problem with an algorithm they call Vector Minimaxing [6] that plays the same cards for the Max player in all the worlds. In Bridge terms the reason why PIMC fails is that the finesse of the Queen of Spades always works in all worlds given perfect information. This can be misleading for the bidding phase also since Flat Monte Carlo thinks it can make four tricks when it has only 50% chances of making them thus reducing the chances of making the contract to 50% instead of 100%. Another problem with this hand is that the best way of playing is to play other cards before trying the finesse since it can gain information on the repartition of the Spades. If such information is available it is best to try the finesse for the side that has the least Spades. This is known as Discovery Play. The last thing about this hand is that if South has decided to finesse the Queen of Spades at East, it should first play the Ace in case the Queen is single.

	♠ J 8 7 6 ♡ — ♢ — ♣ —	
♠ 9 4 ♡ — ♢ — ♣ —	N W E S	♠ K Q 5 ♡ — ♢ — ♣ —
	♠ A 10 3 2 ♡ — ♢ — ♣ —	

This hand from [5] illustrates the problem of non locality. In Bridge terms when the dummy plays the 6 East plays the King and the declarer plays the Ace the best play is to finesse the 7 of Spades which is better than to finesse the Jack. From an algorithmic point of view non locality can be explained using figure 1 from [7]. It illustrates non-locality when searching with strategy fusion for Max and perfect information for Min. As usual the Max nodes are squares and the Min nodes are circles. The leaves gives the result of the game in the three possible worlds. For example the move to the right from node d reaches a state labeled $[1\ 0\ 0]$ which means that the game is won in world 1 (hence the 1 in the first position), lost in world 2 (hence the 0 in the second position) and also lost in world 3 (hence the 0 in the third position). The vectors near the internal nodes give the values that are backed up by the strategy fusion for Max and perfect information for Min algorithm. We can see that each Max node is evaluated by choosing the move that gives the maximum average outcome. For example at node d there are two moves, the left one leads to $[1\ 0\ 0]$ and therefore has an average of $\frac{1}{3}$ whereas the right one leads to $[0\ 1\ 1]$ and has an average of $\frac{2}{3}$. So node d backs up $[0\ 1\ 1]$. However it is not globally optimal. If instead of choosing the right move at node d it chooses the left move it backs up $[1\ 0\ 0]$ and then the b node would have been evaluated better also with $[1\ 0\ 0]$. It illustrates that choosing the local optimum at node d prevents from finding the real optimum at node b. At Min nodes the algorithm chooses for each world the minimal outcome over all children since it can choose the move it prefers most in each different world.

Figure 1: Example of a tree with three worlds illustrating non-locality.



	♠ x x x ♡ — ♢ K x ♣ —	
♠ 6 2 ♡ — ♢ A x ♣ x	N W E S	♠ A Q J ♡ — ♢ x x ♣ —
	♠ K ♡ — ♢ x x x x ♣ —	

This hand illustrates that PIMC can drop the King of Spades as South when West plays the small Clubs because the King is useless in DDS as it is always taken by the Ace. In a real game the declarer can play a small Spades from the dummy and it is not clear East will play the Ace, leaving the possibility to win the trick with the King.

	♠ A K Q ♡ A K Q ♢ K Q J 10 ♣ A K J	
♠ — ♡ — ♢ — ♣ —	N W E S	♠ — ♡ — ♢ — ♣ —
	♠ x x x ♡ x x x ♢ x x x x ♣ x x x	

This hand comes from Fred Gitelman and illustrates reasoning on the cards played by the defense to infer the possible hands. The contract is 6 NT. If West starts with the Ace of Diamonds and then plays a small Clubs the declarer can infer that West does not have the Queen of Clubs and choose not to finesse the Queen. GIB for example does not see it and finesse the Queen.

	♠ K Q 9 8 7 6 ♡ — ♢ — ♣ —	
♠ — ♡ — ♢ — ♣ —	N W E S	♠ — ♡ — ♢ — ♣ —
	♠ A 10 5 ♡ — ♢ — ♣ —	

This hand illustrates that strong Bridge players know how to play so as to avoid rare negative events. Monte Carlo search does not see these events if the corresponding worlds are not generated by the sampling. The Spades are won 100% of the time for a human player. The only way to lose a trick at Spades is when the Jack is with the three remaining Spades which is rare. A human player will play the King to discover if it is the case and then play accordingly doing the right finesse.

3 Vectors of Outcomes and Pareto Fronts

In this section we define Vectors and Pareto fronts that are used by the algorithms in the next section.

3.1 Definitions for Vectors

Given n different possible worlds, a vector of size n keeps the status of the game for each possible world. A zero at index n means that the game is lost for world number n. A one means the game is won. Associated to the vector there is another vector of booleans indicating whether the world is possible in the current state. At the root of the search all worlds are possible but when an opponent makes a move, the move is usually only valid in some of the worlds and the valid worlds are reduced.

The maximum of two vectors is a vector that for each index contains the maximum of the two values at this index in the two vectors. Similarly for the minimum.

A vector is greater or equal to another vector if for all indices it contains a value greater or equal to the value contained at this index in the other vector and if the valid worlds are the same for the two vectors. A vector dominates another vector if it is greater or equal to the other vector.

The score of a vector is the average among all possible worlds of the values contained in the vector.

3.2 Pareto Front

A Pareto front is a set of vectors. It maintains the set of vectors that are not dominated by other vectors. When a new vector is a candidate for insertion in the front the first thing to verify is whether the candidate vector is dominated by a vector in the front. If it is the case the candidate vector is not inserted and the front stays the same. If the candidate vector is not dominated it is inserted in the front and all the vectors in the front that are dominated by the candidate vector are removed.

For example consider the Pareto front $\{[1\ 0\ 0], [0\ 1\ 1]\}$. If we add the vector $[0\ 0\ 1]$ to the front, then the front stays unchanged since $[0\ 0\ 1]$ is dominated by $[0\ 1\ 1]$. If we add the vector $[1\ 1\ 0]$ then the vector $[1\ 0\ 0]$ is removed from the front since it is dominated by $[1\ 1\ 0]$, and then $[1\ 1\ 0]$ is inserted in the front. The new front becomes $\{[1\ 1\ 0], [0\ 1\ 1]\}$.

It is useful to compare Pareto fronts. A Pareto front is greater or equal to another Pareto front if for each element of the second Pareto front there is an element in the first Pareto front which is greater or equal to the element of the second Pareto front.

4 Search with Strategy Fusion and Non-Locality

In this section we explain in details the search algorithm and its optimizations.

4.1 Maximizing the probability of winning

In Bridge the score of a board is calculated from the number of tricks required by the contract and the number of won tricks. For instance, let us consider the contract of 3NT used in our experiments where the minimum number of tricks required is 9. For 9 won tricks, the score is +400, for 10 won tricks the score is +430 since the bonus points for one overtrick is only +30 but if the declarer gets only 8 tricks the contract is defeated and the score is then -50. This threshold has an impact on the card play, where the first goal is to make the contract and then to try to obtain overtricks if it does not endanger the contract. In our experiments we maximize the probability of making the contract which is not optimal but reasonable.

4.2 Search with Strategy Fusion

Let assume that the defense know the cards of the declarer and that the declarer optimizes against all possible states that corresponds to his information. The score of a move for the declarer is the score of the vector that has the best score among the vectors in the Pareto front of the move. At a Max node the declarer computes after each move the union of the Pareto fronts of all the moves that have been tried so far. Min has knowledge of the declarer cards so in each world she takes the move that minimizes the result of Max. The code for Min and Max nodes is given in algorithm 2. $\alpha\mu$ is a generalization of PIMC since a search with a depth of one is PIMC.

The parameter M controls the number of Max moves, when $M = 0$ the algorithm reaches a leaf and each remaining possible world is evaluated with a double dummy search. The stop function is given in algorithm 3. It also stops the search if the contract is already won no matter what is played after. The parameter $state$ contains the current state where all the moves before have been played and which does not contain the hidden information. The parameter $Worlds$ contains the set of all possible worlds compatible with the moves already played. The Pareto front is first initialized with an empty set (line 5). If at a min node, the set of all possible moves in all possible worlds is calculated (lines 7-11). For each move, the move is played, the possible worlds updated and a recursive call is performed. The Pareto front resulting from the recursive call is then combined with the overall front (lines 12-17). We will explain later the min algorithm. Similar operations are performed for a Max node except that the combination with the overall front is then done with the max algorithm (lines 19-28). We explain the max algorithm in the next section.

Algorithm 2 The $\alpha\mu$ search algorithm without cuts and without transposition table.

1: Function $\alpha\mu$ $(state, M, Worlds)$
2:   **if** $stop(state, M, Worlds, result)$ **then**
3:     **return** $result$
4:   **end if**
5:   $front \leftarrow \emptyset$
6:   **if** Min node **then**
7:     $allMoves \leftarrow \emptyset$
8:     **for** $w \in Worlds$ **do**
9:       $l \leftarrow$ legalMoves (w)
10:       $allMoves = allMoves \cup l$
11:     **end for**
12:     **for** $move \in allMoves$ **do**
13:       $s \leftarrow$ play $(move, state)$
14:       $W_1 \leftarrow \{w \in Worlds : move \in w\}$
15:       $f \leftarrow \alpha\mu\,(s, M, W_1)$
16:       $front \leftarrow$ min$(front, f)$
17:     **end for**
18:   **else**
19:     **for** $w \in Worlds$ **do**
20:       $l \leftarrow$ legalMoves (w)
21:       $allMoves = allMoves \cup l$
22:     **end for**
23:     **for** $move \in allMoves$ **do**
24:       $s \leftarrow$ play $(move, state)$
25:       $W_1 \leftarrow \{w \in Worlds : move \in w\}$
26:       $f \leftarrow \alpha\mu\,(s, M-1, W_1)$
27:       $front \leftarrow$ max$(front, f)$
28:     **end for**
29:   **end if**
30:   **return** $front$

4.3 Max nodes

At Max nodes each possible move returns a Pareto front. The overall Pareto front is the union of all the Pareto fronts of the moves. The idea is to keep all the possible options for Max, i.e. Max has the choice between all the vectors of the overall Pareto front. In order to optimize computations and memory, vectors that are dominated by another vector in the same Pareto front are removed.

Algorithm 3 The function that stops search.

Function $stop$ $(state, M, Worlds, result)$
  if $declarerTricks(state) \geq contract$ **then**
    for $w \in Worlds$ **do**
      $result[w] \leftarrow 1$
    end for
    return $True$
  end if
  if $defenseTricks(state) > 13 - contract$ **then**
    for $w \in Worlds$ **do**
      $result[w] \leftarrow 0$
    end for
    return $True$
  end if
  if $M = 0$ **then**
    for $w \in Worlds$ **do**
      $result[w] \leftarrow$ doubleDummy (w)
    end for
    return $True$
  end if
  return $False$

4.4 Min nodes

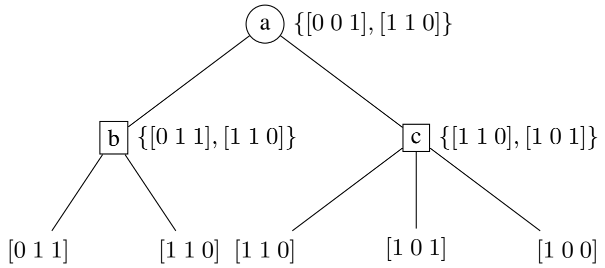
The Min players can choose different moves in different possible worlds. So they take the minimum outcome over all the possible moves for a possible world. So when they can choose between two vectors they take for each index the minimum between the two values at this index of the two vectors.

Now when Min moves lead to Pareto fronts, the Max player can choose any member of the Pareto front. For two possible moves of Min, the Max player can also choose any combination of a vector in the Pareto front of the first move and of a vector in the Pareto front of the second move. In order to build the Pareto front at a Min node we therefore have to compute all the combinations of the vectors in the Pareto fronts of all the Min moves. For each combination the minimum outcome is kept so as to produce a unique vector. Then this vector is inserted in the Pareto front of the Min node.

An example of the product of Pareto fronts is given in figure 2. We can see in the figure that the left move for Min at node a leads to a Max node b with two moves. The Pareto front of this Max node is the union of the two vectors at the leaves: $\{[0\ 1\ 1], [1\ 1\ 0]\}$. The right move for Min leads to a Max node c with three possible moves. When adding the vectors to the Pareto front of the Max node c, the algorithm sees that $[1\ 0\ 0]$ is dominated by $[1\ 0\ 1]$ and therefore does not add it to the Pareto front at node c. So the resulting Pareto front for the Max node c is $\{[1\ 1\ 0], [1\ 0\ 1]\}$. Now to compute the Pareto front for the root Min node we perform the product of the two reduced Pareto fronts of the children Max nodes and it gives: $\{[0\ 1\ 0], [0\ 0\ 1], [1\ 1\ 0], [1\ 0\ 0]\}$. We then reduce the Pareto front of the Min node and remove $[0\ 1\ 0]$ which is dominated by $[1\ 1\ 0]$ and also remove $[1\ 0\ 0]$ which is also dominated by $[1\ 1\ 0]$. Therefore the resulting Pareto front for the root Min node is $\{[0\ 0\ 1], [1\ 1\ 0]\}$.

We can also explain the behavior at Min nodes on the non-locality example of figure 1. The Pareto front at Max node d is $\{[1\ 0\ 0], [0\ 1\ 1]\}$. The Pareto front at Max node e is $\{[0\ 0\ 0], [1\ 0\ 0]\}$. It is reduced to $\{[1\ 0\ 0]\}$ since $[0\ 0\ 0]$ is dominated. Now at node b the product of the Pareto fronts at nodes d and e gives $\{[1\ 0\ 0], [0\ 0\ 0]\}$ which is also reduced to $\{[1\ 0\ 0]\}$. The Max player can now see that the b node is better than the c node, it was not the case for the strategy fusion algorithm without Pareto fronts.

Figure 2: Product of Pareto fronts at Min nodes.



The function to compute the minimum of two Pareto fronts is given in algorithm 4.

Algorithm 4 The function for joining two Pareto fronts at Min nodes.

Function min $(front, f)$
  $result \leftarrow \emptyset$
  for $vector \in front$ **do**
    for $v \in f$ **do**
      for $w \in 0..size(vector)$ **do**
        if $vector[w] < v[w]$ **then**
          $r[w] \leftarrow vector[w]$
        else
          $r[w] \leftarrow v[w]$
        end if
      end for
      remove the vectors from $result \leq r$
      if no vector from $result \geq r$ **then**
        $result \leftarrow result \cup r$
      end if
    end for
  end for
  return $result$

4.5 Skipping Min nodes

The search one depth deeper at a Min node will always give the same result as the Pareto front at that node since the Double Dummy Solver has already searched all worlds with an $\alpha\beta$ and that the Min player can choose the move for each world and therefore will have the same result as the $\alpha\beta$ for each world.

This is why we only keep the number M of Max moves to be played in the search. The search will never stop after a Min move since recursive calls at Min node do not decrease M. This is intended since the results of the search after a Min move are the same as before the Min move.

4.6 Iterative Deepening and Transposition Table

Iterative deepening starts with one Max move and increases the number of Max moves at every iteration. The number of Max moves is the number of Max nodes that have been traversed before reaching the current state. The results of previous searches for all the nodes searched are stored in a transposition table.

An entry in the transposition table contains the Pareto front of the previous search at this node and the best move found by the search.

When a search is finished at a node, the entry in the transposition table for this node is updated with the new Pareto front and the new best move.

4.7 Comparing Pareto Fronts

A Pareto front p_1 is smaller or equal to another Pareto front p if $p \cup p_1 = p$. When it is the case it is safe to ignore the move associated to p_1 since it adds no options to p. If it is true for the current front p_1 at a Min node it will also be true when searching more this Min node since p_1 can only be reduced to a smaller Pareto front by more search at a Min node.

An efficient way to compare p to p_1 is to ensure that each vector of p_1 is dominated by another vector in p. The corresponding algorithm is given in algorithm 5.

Algorithm 5 The function to test if a Pareto front is smaller than another one.

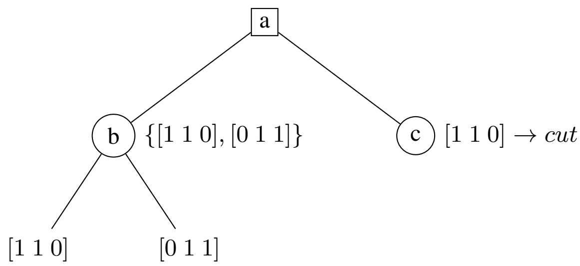
Function $\leq (front, f)$
  for $vector \in front$ **do**
    $oneGreaterOrEqual \leftarrow False$
    for $v \in f$ **do**
      if $vector <= v$ **then**
        $oneGreaterOrEqual \leftarrow True$
        $break$
      end if
    end for
    if $oneGreaterOrEqual = False$ **then**
      return $False$
    end if
  end for
  return $True$

4.8 Early Cut

If a Pareto front at a Min node is dominated by the Pareto front of the upper Max node it can safely be cut since the evaluation is optimistic for the Max player. The Max player cannot get a better evaluation by searching more under the Min node and it will always be cut whatever the search below the node returns since the search below will return a Pareto front smaller or equal to the current Pareto front. It comes from the observation that a world lost at a node is also lost at all nodes below.

Figure 3 gives an example of an early cut at a Min node. The root node a is a Max node, the first move played at a returned $\{[1\ 1\ 0], [0\ 1\ 1]\}$. The second move is then tried leading to node c and the initial Pareto front calculated with double dummy searches at node c is [1 1 0]. It is dominated by the Pareto front of node a so node c can be cut.

Figure 3: Example of an early cut at node c.



4.9 Root Cut

If a move at the root of $\alpha\mu$ for M Max moves gives the same probability of winning than the best move of the previous iteration of iterative deepening for $M-1$ Max moves, the search can be safely be stopped since it is not possible to find a better move. A deeper search will always return a worse probability than the previous search because of strategy fusion. Therefore if the probability is equal to the one of the best move of the previous shallower search the probability cannot be improved and a better move cannot be found so it is safe to cut.

4.10 $\alpha\mu$

$\alpha\mu$ with transposition table and cuts is a search algorithm using Pareto fronts as evaluations and bounds. The algorithm is given in algorithm 6.

The evaluation of a state at a leaf node is the double dummy evaluation for each possible world. An evaluation for a world is 0 if the game is lost for the Max player and 1 if the game is won for the Max player (lines 2-5).

The algorithm starts with getting the entry t of $state$ in the transposition table (line 6). The entry contains the last Pareto front found for this state and the best move found for this state, i.e. the move associated to the best average.

If the state is associated to a Min node, i.e. a Min player is to play, the algorithm starts to get the previously calculated Pareto front from the transposition table (line 8). Then it looks for an early cut (lines 9-11). If the node is not cut it computes the set of all possible moves over all the valid worlds (lines 12-16). It then moves the move of the transposition table in front of the possible moves (line 17). After that it tries all possible moves (line 18). For each possible move it computes the set W_1 of worlds still valid after the move and recursively calls $\alpha\mu$ (lines 19-21) . The parameters of the recursive call are s, the current state, M the number of Max moves to go which is unchanged since we just played a Min move, W_1 the set of valid worlds after $move$, and an empty set for $alpha$ to avoid deeper cuts. The front returned by the recursive call is then combined to the current front using the min function (line 22). When the search is finished it updates the transposition table and returns the $mini$ Pareto front (lines 24-25).

If the state is associated to a Max node it initializes the resulting front with an empty set (line 27). Then as in the Min nodes it computes the set of all possible moves and moves the transposition table move in front of all the possible moves (lines 28-32). Then it tries all the moves and for each move computes the new set W_1 of valid worlds and recursively calls $\alpha\mu$ with $M-1$ since a Max move has just been played and $front$ as $alpha$ since a cut can happen below when the move does not improve $front$ (lines 33-36). The resulting front f is combined with front with the max function (line 37). If the score of the best move ($\mu(front)$) is equal to the score of the best move of the previous search and the node is the root node then a Root cut is performed (lines 38-42). When the search is finished the transposition table is updated and $front$ is returned (lines 44-45).

The search with strategy fusion is always more difficult for the Max player than the double dummy search where the Max player can choose different moves in the different possible worlds for the same state. Therefore if a double dummy search returns a loss in a possible world, it is sure that the search with $\alpha\mu$ will also return a loss for this world.

If the search is performed until terminal nodes and all possible worlds are considered then $\alpha\mu$ solves the strategy fusion and the non locality problem for the game where the defense has perfect information.

If the search is stopped before terminal nodes and not all possible worlds are considered then $\alpha\mu$ is a heuristic search algorithm.

The algorithm is named $\alpha\mu$ since it maximizes the mean and uses an α bound.

Algorithm 6 The $\alpha\mu$ search algorithm with cuts and transposition table.

1: Function $\alpha\mu$ $(state, M, Worlds, \alpha)$
2:   **if** $stop(state, M, Worlds, result)$ **then**
3:     update the transposition table
4:     **return** $result$
5:   **end if**
6:   $t \leftarrow$ entry in the transposition table
7:   **if** Min node **then**
8:     $mini \leftarrow \emptyset$
9:     **if** $t.front \leq \alpha$ **then**
10:       **return** $mini$
11:     **end if**
12:     $allMoves \leftarrow \emptyset$
13:     **for** $w \in Worlds$ **do**
14:       $l \leftarrow$ legalMoves (w)
15:       $allMoves = allMoves \cup l$
16:     **end for**
17:     move $t.move$ in front of $allMoves$
18:     **for** $move \in allMoves$ **do**
19:       $s \leftarrow$ play $(move, state)$
20:       $W_1 \leftarrow \{w \in Worlds : move \in w\}$
21:       $f \leftarrow \alpha\mu\ (s, M, W_1, \emptyset)$
22:       $mini \leftarrow$ min$(mini, f)$
23:     **end for**
24:     update the transposition table
25:     **return** $mini$
26:   **else**
27:     $front \leftarrow \emptyset$
28:     **for** $w \in Worlds$ **do**
29:       $l \leftarrow$ legalMoves (w)
30:       $allMoves = allMoves \cup l$
31:     **end for**
32:     move $t.move$ in front of $allMoves$
33:     **for** $move \in allMoves$ **do**
34:       $s \leftarrow$ play $(move, state)$
35:       $W_1 \leftarrow \{w \in Worlds : move \in w\}$
36:       $f \leftarrow \alpha\mu\ (s, M-1, W_1, front)$
37:       $front \leftarrow$ max$(front, f)$
38:       **if** root node **then**
39:         **if** $\mu(front) = \mu$ of previous search **then**
40:           break
41:         **end if**
42:       **end if**
43:     **end for**
44:     update the transposition table
45:     **return** $front$
46:   **end if**

4.11 Equivalent Cards and Partitions

Before performing the Iterative Deepening search the program checks whether it is useful to perform a search. For example if there is only one possible move which often happens or if there are two equivalent moves. To detect that two moves of the same color are equivalent the program normalizes the state using the same idea as in Partition Search [9]. In the normalized state all the cards in a color have consecutive values. It is then easy to detect that two cards in the same hand are equivalent: they have consecutive values.

4.12 Generating Possible Worlds

Before performing the search the program generates the set of possible worlds. The principle of the generation is to randomly generate worlds and to retain those that satisfy a set of constraints. The constraints are constraints on the initial deal corresponding to the generated world. The reconstructed initial deal must comply with the constraints on the contract. The generated world also has to comply with the known sluffs of the other players.

5 Experimental Results

In our experiments we fix the bids so as to concentrate on the evaluation of the card play. We use the one no trump, pass, three no trump, pass, pass, pass bid for all experiments.

We use duplicate scoring. It means that the different evaluated programs will play the same hands against the same opponents. When $\alpha\mu$ is the declarer it will play against two PIMC as the defense. $\alpha\mu$ is a generalization of PIMC since $\alpha\mu$ at depth one is PIMC. So in order to compare $\alpha\mu$ as a declarer to PIMC as a declarer we compare $\alpha\mu$ as a declarer to $\alpha\mu$ with $M = 1$ as a declarer.

There are constraints on the hands due to the contract. Initial deals and possible worlds for PIMC and $\alpha\mu$ are generated according to the constraints. However when using no more constraints many initial deals are useless for evaluating the program since they are always won or always lost and that they do not discriminate between programs since all the programs have the same result. In order to alleviate this problem we only keep initial deals where PIMC has more than 30% and less than 70% winning rate, i.e. the undecided and balanced deals.

We first test $\alpha\mu$ at different depth versus PIMC with a fixed number of possible worlds. Table 1 gives the results for different runs of $\alpha\mu$ as the declarer with 20 worlds and 40 worlds against PIMC as the defense with 20 worlds. All results are computed playing the same 500 initial deals with the same seed for each deal. We see that looking two or three Max moves ahead can be beneficial. The number of discrepancies is the number of times $\alpha\mu$ chooses a different move than the move at depth one (i.e. the PIMC move). We note that it happens relatively rarely. PIMC is already a very strong player as a declarer so improving on it even slightly is difficult. For 52 cards and 20 worlds PIMC ($\alpha\mu$ with $M = 1$) scores 60.2% and $\alpha\mu$ with $M = 3$ scores 62.0%. For 52 cards and 40 worlds PIMC ($\alpha\mu$ with $M = 1$) scores 62.4% and $\alpha\mu$ with $M = 3$ scores 63.2%. For 36 cards and 20 worlds PIMC ($\alpha\mu$ with $M = 1$) scores 46.4% and $\alpha\mu$ with $M = 3$ scores 48.2%. We can conclude that $\alpha\mu$ improves on PIMC.

We now compare the times to play moves with and without Transposition Tables and cuts. Table 2 gives the average time per move of different configurations of $\alpha\mu$ playing entire games. TT means Transposition Table, R means Root Cut, E means Early Cut. We can observe that a Transposition Table associated to cuts improves the search time. For $M = 1$ the search time is 0.096 seconds. For $M = 3$ without transposition table and cuts the average search time per move is 18.678 seconds. When using a transposition table associated to early and root cuts it goes down to 1.228 seconds.

Table 1: Comparison of the scores of different configurations of $\alpha\mu$ on deals with 52 or 36 cards.

Cards	M	Worlds	Discrepancies	Score
52	1	20	0 / 13 000	60.2%
52	2	20	169 / 13 000	63.0%
52	3	20	276 / 13 000	62.0%
52	1	40	0 / 13 000	62.4%
52	2	40	213 / 13 000	62.4%
52	3	40	388 / 13 000	63.2%
36	1	20	0 / 13 000	46.4%
36	2	20	124 / 13 000	47.8%
36	3	20	190 / 13 000	48.2%

Table 2: Comparison of the average time per move of different configurations of $\alpha\mu$ on deals with 52 or 36 cards.

Cards	M	Worlds	TT	R	E	Time
52	1	20				0.096
52	2	20	n	n	n	1.306
52	2	20	y	y	n	0.389
52	2	20	y	n	y	0.436
52	2	20	y	y	y	0.363
52	3	20	n	n	n	18.678
52	3	20	y	y	n	4.089
52	3	20	y	n	y	1.907
52	3	20	y	y	y	1.228

6 Conclusion and Future Work

We presented the $\alpha\mu$ algorithm for Bridge card play. It assumes the opponents have perfect information. It enables to search a few moves ahead taking into account the strategy fusion and the non locality problems. To solve the non locality problem it uses Pareto fronts as evaluations of states and combines them in an original way at Min and Max nodes. To solve the strategy fusion problem it plays the same moves in all the valid worlds during search. Experimental results for the 3NT contract shows it improves on PIMC.

We also presented the use of a transposition table as well as the early and the root cut for $\alpha\mu$. When searching three Max moves ahead it enables the search to be fifteen times faster while returning the same move as the longer search without the optimizations.

In future work we expect to use partition Search with $\alpha\mu$. We also plan to take into account that the defense only has incomplete information and to take advantage of that for the declarer. We will also apply the algorithm to the defense play. It should also improve the algorithm to deal with real scores instead of only win/loss.

Acknowledgment

Thanks to Alexis Rimbaud for explaining me how to use the solver of Bo Haglund and to Bo Haglund for his Double Dummy Solver.

REFERENCES

[1] Noam Brown and Tuomas Sandholm, 'Superhuman AI for multiplayer poker', *Science*, **365**(6456), 885–890, (2019).

[2] Michael Buro, Jeffrey Richard Long, Timothy Furtak, and Nathan Sturtevant, 'Improving state evaluation, inference, and search in trick-based card games', in *Twenty-First International Joint Conference on Artificial Intelligence*, (2009).

[3] Peter I Cowling, Edward J Powley, and Daniel Whitehouse, 'Information set monte carlo tree search', *IEEE Transactions on Computational Intelligence and AI in Games*, **4**(2), 120–143, (2012).

[4] Pallab Dasgupta, Partha Pratim Chakrabarti, and S.C. DeSarkar, 'Searching game trees under a partial order', *Artificial Intelligence*, **82**(1-2), 237–257, (1996).

[5] Ian Frank and David Basin, 'Search in games with incomplete information: A case study using bridge card play', *Artificial Intelligence*, **100**(1-2), 87–123, (1998).

[6] Ian Frank and David Basin, 'A theoretical and empirical investigation of search in imperfect information games', *Theoretical Computer Science*, **252**(1-2), 217–256, (2001).

[7] Ian Frank, David A Basin, and Hitoshi Matsubara, 'Finding optimal strategies for imperfect information games', in *AAAI/IAAI*, pp. 500–507, (1998).

[8] Timothy Furtak and Michael Buro, 'Recursive monte carlo search for imperfect information games', in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pp. 1–8. IEEE, (2013).

[9] Matthew L. Ginsberg, 'Partition search', in *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1.*, pp. 228–233, (1996).

[10] Matthew L. Ginsberg, 'GIB: imperfect information in a computationally challenging game', *J. Artif. Intell. Res.*, **14**, 303–358, (2001).

[11] Bo Haglund. Search algorithms for a bridge double dummy solver, 2010.

[12] Sebastian Kupferschmid and Malte Helmert, 'A skat player based on monte-carlo simulation', in *International Conference on Computers and Games*, pp. 135–147. Springer, (2006).

[13] David NL Levy, 'The million pound bridge program', *Heuristic Programming in Artificial Intelligence The First Computer Olympiad*, 95–103, (1989).

[14] Edward Lockhart, Marc Lanctot, Julien Pérolat, Jean-Baptiste Lespiau, Dustin Morrill, Finbarr Timbers, and Karl Tuyls, 'Computing approximate equilibria in sequential adversarial games by exploitability descent', *arXiv preprint arXiv:1903.05614*, (2019).

[15] Jeffrey Richard Long, Nathan R Sturtevant, Michael Buro, and Timothy Furtak, 'Understanding the success of perfect information monte carlo sampling in game tree search', in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, (2010).

[16] Martin Müller, 'Partial order bounding: A new approach to evaluation in game tree search', *Artificial Intelligence*, **129**(1-2), 279–311, (2001).

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al., 'A general reinforcement learning algorithm that masters chess, shogi, and go through self-play', *Science*, **362**(6419), 1140–1144, (2018).

[18] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis, 'Mastering the game of go without human knowledge', *Nature*, **550**(7676), 354, (2017).

[19] Nathan R Sturtevant and Adam M White, 'Feature construction for reinforcement learning in hearts', in *International Conference on Computers and Games*, pp. 122–134. Springer, (2006).

[20] Véronique Ventos, Yves Costel, Olivier Teytaud, and Solène Thépaut Ventos, 'Boosting a bridge artificial intelligence', in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 1280–1287. IEEE, (2017).

[21] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione, 'Regret minimization in games with incomplete information', in *Advances in neural information processing systems*, pp. 1729–1736, (2008).