Trick Costs for $\alpha\mu$ and New Relatives

Samuel Bounan[1], **Stefan Edelkamp**[2]

[1] École normale supérieure de Lyon, Auvergne-Rhône-Alpes, France
[2] Computer Science Department, Faculty of Electrical Engineering, Czech Technical University in Prague
samuel.bounan@ens-lyon.fr, edelkste@fel.cvut.cz

Abstract

In this paper we present a player for incomplete information card games with tricks scored by points or eyes. We factorize existing algorithms into a template that captures the main ingredients of such game trees. We then analyze three different algorithms, and the impact of the information given during the algorithm on the decisions it makes. We extend these algorithms to work with cost instead of winning vectors, and illustrate their effectiveness in finding good cards. We develop a new algorithm that tries to respect known information.

Introduction

Many perfect-information board games such as Checkers [24], Oware [2], Connect4 [1], Awari [23], and Nine-Men-Morris [14] have been solved, or, as in Chess, Shogi or Go, computer AIs clearly outperform humans [27]. Therefore, research attention has shifted to incomplete information games. The partially observable board game Stratego has recently been analyzed with *DeepNash*, a model-free multiagent reinforcement learning algorithm [22]. It achieved an all-time top-3 rank on the Gravon games platform, competing with human expert players.

Card games remain an objective of research for decision making with imperfect information. After some variants of Poker have been solved or played to a satisfying degree [3, 21], trick-taking games such as Skat [5, 17], Hearts [29] and Bridge [6, 13, 18] have been identified as current AI challenges. One obstacle is that, given the large number of tricks and degree of uncertainty, a direct application of reinforcement learning as in Go [26] is less obvious.

Most recently, some card game AIs are beginning to challenge human supremacy. Notably world-class caliber play in Bridge [7, 6], Spades [8], and Skat [10, 11]. In these cutting-edge players, domain-dependent information is provided, such as winning probabilities extracted from human expert games [9]. We have implemented an efficient framework for general card games. General game playing has a long tradition: there have been several insightful international competitions using GDL or GDL-II (for incomplete information) [25] as the input language. While the players achieved a remarkable playing strength [12], the generation of moves is slow. Even faster frameworks like Google DeepMind's *OpenSpiel* are less efficient for card games than our framework with its concise card encodings.

Another problem is that for multi-player teams most general game playing algorithms are not yet competitive. We focus on trick-taking stage of card games. While we have implemented bidding and dog putting strategies for all the card games, they are not the subject of this work, as they are often based on domain-dependent conventions.

The paper is organized as follows. First we will define the building blocks for the formalization of the problem. Our notation is not taken from any specific work, but helps to design a naive algorithm. Section refers to the construction of the driver template algorithm. Then, we discuss the efficiency of three algorithms. The first one, called Perfect Information Monte-Carlo, PIMC, is well-known [15]. The second one, $\alpha\mu$ is newer [7] and shown effective in Bridge. The third one is a new. In all three, we present the algorithm principles in a template algorithm thus adapt them to our model. Also, $\alpha\mu$ was only used with Boolean score vectors and with two players. We extend it to scores in $\mathbb{R}$ and to several players. Finally, we present some results we obtained in our implementation. Our code generalizes ideas from [10] to implement, while in a restricted setting with simplified bidding. The contribution of this paper are as follows. We provide an efficient framework. calculate the effect of truncating tree search with subsequent random playouts. quantify the information gain of $\alpha\mu$ wrt PIMC. successfully apply $\alpha\mu$ algorithm to support more players in a team and to general trick costs; propose a novel general incomplete-information algorithm as a compromise of different nodes in the backup.

Preliminaries

The *set of cards* $\mathcal{C}$ is dealt to p players at the beginning of the game. A *world* is any possible deal of the cards among the players, $\mathcal{W}$ the set of worlds, and for a world $w \in \mathcal{W}$, w_i denotes the hand of player i, $i \in \{1, \ldots, p\}$. Worlds change during a game. We call *state* the history of cards already played, and $\mathcal{S}$ the set of possible states.

In a state s, the *next player* to play a card is determined by a function *turn*$(s) \in \{1, \ldots p\}$ and the set of *legal cards* that can be played in s with a hand w_i by *legal*$(w_i, s) \subseteq \mathcal{C}$. In some states the game ends, this is determined by a Boolean

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

function *over*(s), and in these cases each player receives an amount of points given by a function *score*$(i, s) \in \mathbb{R}$. The goal of each player is to maximize its final score, and we assume rationality of everyone and common knowledge of rationality. In general in a state $s \in \mathcal{S}$ the world is neither perfectly known nor completely unknown and with a hand w_i player $i \in \{0, \ldots, p\}$ assumes a particular distribution on the set of worlds $D_{i,w_i,s}$.

To construct an artificial player, we (only) need to know in a state s, in a position i and with a hand h, what is the best card to play. In extension to articles that formalize tree searches, as in [4], [20] we chose the following definition.

If we define $\sigma(i, h, s) \in \mathbb{R}$ as the best score that can be obtained, with argσ to denote a probabilistic function that returns a card corresponding to the maximum score, uniformly chosen in $\mathcal{U}$ among cards that satisfy this property, we have

$$\sigma(i, h, s) = \max_{c \in \mathit{legal}(h,s)} \mathbb{E}_{w \hookrightarrow D(i,h,s)} \mathbb{E}_{f \hookrightarrow \mathcal{U}(\mathcal{F})} \mathit{score}(i, f)$$

where $\mathcal{F}$ is the set of possible optimal final states. With $s_1 = s$ and $s_{k+1} = s_k \cup \arg\sigma(\mathit{turn}(s_k), w_{\mathit{turn}(s_k)} \setminus s_k, s_k)$ for $k > 1$ we have

$$\mathcal{F} = \{f \in \mathcal{S} \mid \exists k \in \mathbb{N} \quad \mathbb{P}[s_k = f] \neq 0 \text{ and } \mathit{over}(f)\}.$$

Every object can be easily derived from the rules, except the knowledge distribution $D_{i,h,s}$. We will construct O the set of possible worlds with a non-zero probability for D.

Let w be a world in $\mathcal{W}$ with state s and hand h. We have the following constraints on w: a) $w_i = h$; and b) in every state before s, the players played optimally (because of rationality). Let n be chosen, so that s is an n-tuple. Let s_k be the card played in s at step $k < n$. From the second constraint, we know that in the state $s_{|k} = (s_1, \ldots s_k)$ player *turn*$(s_{|k})$ played a card s_{k+1} so that $\mathbb{P}[s_k = \arg\sigma(\mathit{turn}(s_{|k}), w_{\mathit{turn}(s_{|k})} \setminus s_{|k}, s_{|k})] \neq 0$. We have

$$\begin{aligned} O \quad = \quad & \{w \in \mathcal{W}; w_i = h \text{ and } \forall k \in [n] \\ & \quad \mathbb{P}[s_{k+1} = \arg\sigma(\mathit{turn}(s_{|k}), w_{s_{|k}} \setminus s_{|k}, s_{|k}] \neq 0.\} \end{aligned}$$

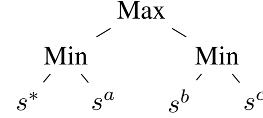
Because no additional knowledge can be used to infer a probability among this set of possible worlds O we have $D_{i,h,s} = \mathcal{U}(O)$. There are two problems with this definition of D. First, in practice players may differ from the perfect rationality, and restricting O to the worlds where everyone played optimally can quickly become a strong bad bias. Second, computing O is computationally costly (it needs to compute σ several times). To simplify $D_{i,h,s}$ we assume it follows an uniform distribution over the set of *legal* worlds P, that can be computed in constant time as $D_{i,h,s} = \mathcal{U}(P)$ with $w \in P$ iff $w_i = h$ and for all $k < n$ we have $s_{k+1} \in \mathit{legal}(w_{\mathit{turn}(s_{|k})}, s_{|k})$.

From a card played by a player i, as player j can often derive a set of cards that player i does not have (initially, player j's hand, maybe enlarged by certainties through the bidding process). We can represent the knowledge of a player j with sets $\neg K_i^j$ corresponding to the cards that player j knows that player i must not have. This ease the exploration and the search tree can be pruned based on the knowledge. When a card c has been played by player i, we can update $\neg K_i^j$ efficiently with Algorithm 1. With this reduction of D we study the general structure of the algorithm we want to design.

Algorithm 1: *update*$(\neg K_i^j, c, s)$

for $c' \in \mathcal{C} \setminus \neg K_i^j$ **do**
  if $c \notin \mathit{legal}(\{c, c'\}, s)$ **then**
    $\neg K_i^j \leftarrow \neg K_i^j \cup \{c'\}$

Figure 1: Depth-1 minimax tree.



Minimax Truncated with Random Playouts

Most algorithms include a fast open-card solver to compute the game value. There are different approaches but they all use the minimax search tree as their basis. Note that due to the tricks taken and the teams the max or min modes may not alternate on each path. Finding the optimal result of a minimax tree can be computationally expensive, so an intuitive idea is to limit its depth by estimating the score of a node at the maximal depth by random playouts of the end of the game. In our case, the tree is not that deep, and we have more information on the knowledge of the players, starting with the knowledge that each leaf of the tree is at a constant depth. It is, therefore, possible to obtain a bound on the error made by approximating the optimal score with these playouts, in particular by taking the average of the score obtained by a number of random playouts.

Suppose we have a perfect alternating minimax tree T. Given the outcomes at each leaf of T, we know that some of them cannot be the optimal minimax score. Let s^* denote the optimal score. In Figure 1, we know that $s^a \geq s^*$ because *Min* would otherwise play for s^a. We also know that one of s^b and s^c is lower than s^* because *Max* chooses the left subtree in the optimal strategy. So we know that if we order the scores of the leaves $s_1 \leq s_2 \leq s_3 \leq s_4$, we have $s^* \in \{s_2, s_3\}$. More generally, we can compute the number of playouts g (resp. l) that have a score higher (resp. lower) than s^*. We will estimate the exact number of playouts greater and less than s^* that are needed to be certain that s^* is optimal for the minimax strategy. Thus, our bounds on the minimax score will be tight. Note that this is also the minimum number of leaves that an algorithm needs to explore to compute s^*. We study alternating minimax trees. For consistency, we denote by depth of a tree T the number (*Max*,*Min*) nodes to a leaf. To estimate l and g we additionally assume that the number of actions at a given depth d is constant, so that a_d denotes the number of possible actions at the root of a tree of depth d for *Max*, and b_d for *Min*.

First, we compute the number of playouts that have a score greater than s^*. We note $s(T)$, the optimal minimax score of a tree T. We have $s(T_{\mathit{init}}) = s^*$. To compute the number of playouts with a score greater than s^* we only need the assumption that $s(T_{\mathit{init}}) \geq s^*$. Indeed, if we know that $s(T) \geq s^*$, it means that *Max* chooses a card a such that for all possible choices b of *Min*, the score of the sub-

tree of T after playing a and b has a score greater than s^*. Otherwise *Min* could choose an action that leads to a score that is strictly lower than s^*. It follows that $s(T) < s^*$, a contradion. Thus, we have $\exists a\, \forall b \quad s(T_{a,b}) \geq s^*$.

We use the assumptions $s(T_{a,b}) \geq s^*$ to setup a recursion. The terminal case is when T has one node, in which case we have one playout that we know has a score greater than s^*. With the above assertion we can deduce that the number of playouts in a tree that have a score greater than s^* depends only on the number of actions *Min* that can made at each node of the tree. Here, we have made the assumption that in a tree with depth d, the number of possible actions for *Min* is a fixed number b_d. Thus the number of playouts that achieve a score greater than s^* only depends on the depth of the tree. We can, therefore, define it as a sequence g_d. The number g_d represents the number of playouts that have a score greater than the optimal minimax score for a tree of depth d. Following what we established earlier, we have $g_0 = 1$ and $g_d = b_d\, g_{d-1}$ for all $d > 0$. Finally, we get $g_d = \prod_{i=1}^{d} b_i$. Symmetrically, we can compute the number of playouts that achieve a score lower than s^* as $l_d = \prod_{i=1}^{d} a_i$. Assuming T is a perfect alternating minimax tree of depth d, the optimal score can be achieved by a playout that is neither among the $l_d - 1$ lowest playout scores nor in the $g_d - 1$ higher playout scores. So the total number of playouts is $l_d\, g_d$.

For a bound on the actual score we need to model the distribution of the playout scores. We assume that the number of points that can be won in T_d is S_d (we assume here that the minimum score that can be obtained is 0 but we can easily adjust the result with S_d in $s_{max}(T_d) - s_{min}(T_d)$). We will, therefore, model the distribution of score with a binomial law of parameters $n = S_d$, p unknown. We approximate this law by $\mathcal{N}(\mu, \sigma^2)$ with $\mu = S_d p$ and $\sigma = \sqrt{S_d p(1-p)}$. In this setting we can compute bounds on s^* directly. In fact, a lower bound on s^* is given by the value of the $l_d/(l_d\, g_d)$ quantile, as we showed above. We have $Q(l_d/(l_d\, g_d)) \leq s^* \leq Q((l_d\, g_d - g_d)/(l_d\, g_d))$ with Q being the quantile function of the normal distribution $Q(p) = \mu + \sigma\sqrt{2}\, f^{-1}(2p-1)$ and f being shorthand for *erf*. By estimating s^* as the midpoint between the two quantiles (μ in our model) we can estimate the error ϵ_d. For simplicity we assume that $g_d = l_d = n_d \geq 2$ (symm. *Min* / *Max*). Thus, we have $l_d/(l_d\, g_d) = 1/n_d$, so that the error $is \epsilon_d =$

$$\frac{Q\left(\frac{n_d-1}{n_d}\right) - Q\left(\frac{1}{n_d}\right)}{2} = \frac{\sigma\sqrt{2}\left(f^{-1}(1-\frac{2}{n_d}) - f^{-1}(\frac{2}{n_d}-1)\right)}{2}$$

$$= \frac{\sigma}{\sqrt{2}}\left(2 \cdot f^{-1}\big(1 - \frac{2}{n_d}\big)\right)$$ (symmetry of f^{-1}). We set

$\sigma = \sqrt{S_d p(p-1)} \leq \frac{\sqrt{S_d}}{2}$, so that $\epsilon_d \leq \sqrt{\frac{S_d}{2}}\, f^{-1}\big(1 - \frac{2}{n_d}\big)$.

If we approximate s^* with the mean of the score obtained by a number m of random playouts $\hat{\mu}$, we have, using the Bienaymé-Tchebychev inequality for the normal approximation $P(|\hat{\mu} - \mu| \geq \epsilon_m) \leq \frac{\sigma^2}{m\epsilon_m^2} = \frac{S_d}{4m\epsilon_m^2}$. Adding the estimation error, we know that by choosing $m \geq \frac{S_d}{4\delta\epsilon_m^2}$ we have $P(|\hat{\mu} - s^*| \geq \epsilon_m + \epsilon_d) \leq \delta$.

For Belote we set $n_d = 2^d d!$ and $S_d = 10d$. if we want a 20%-approximation of the score, valid with a 95% chance, we can estimate the tree after the 5th trick, with $m = 8$. If we want to have a 15%-approximation valid with a 95% chance, we can cut after the 6th trick with $m = 3$.

Algorithm 2: *naive*

```
function search(i, h, s)
    res ← {s}
    max ← ⊥
    for c ∈ legal(h, s) do
        j ← turn((s, c))
        score' ← ⊥
        for w ∈ P do
            F ← search(j, w_j, (s, c))
            for x ∈ F do
                score' += score(i, x)/|F × P|
        if score > max then
            res ← {c}
            max ← score'
        else if score' = max then
            res ← res ∪ {c}
    return res

function chooseCard(h, s)
    for c ∈ legal(h, s) do
        F ← search(id, h, s)
        S[c] ← μ_{f∈F}(score(id, f))
    return arg max S
```

Template Algorithm

Following $\sigma(i, a, s)$ we design a first naive Algorithm 2 to solve our problem, which branches on every maximum and expectation of σ. The problem with this algorithm is its time complexity. Let $T(d)$ denote the time complexity of the algorithm if d cards are still to be played before the game is over. We have $T(d) = |legal(h, s)| \cdot |P| \cdot T(d-1)$. If we assume (for the sake of clarity) the number of legal cards to be fixed during the game, and that the number of possible worlds to be constant, we have $T(d) = K^d$, where K is a constant. Because of this time complexity the naive algorithm is not practical. The common methods to solve this problem use the same set of worlds for all nodes in the search tree. This was not the case in our naive algorithm, where each node branches to its possible set of worlds P. Let us assume a set of worlds W for whole tree. We will give each node a value representing the optimal scores in state s for the worlds in W. The type of value depends on the specification of the algorithm, it will be some kind of array indexed by $w \in W$ with the optimal scores of each w. It should contain the scores of each player, as it will be the only value passed through the tree.

With this tree, it is often easy to cut some unused nodes using deep pruning, as in the $\alpha\beta$ pruning algorithm. This pruning is essential in the minimax tree. As shown in [16] we can hope to reach a complexity close to $K^{d/2}$. This is adapted to two player minimax games, with perfect informa-

Algorithm 3: tree

function *tree*$(s, \alpha, W, parent)$
  $parent[myteam] \leftarrow myid$
  $r \leftarrow init_{myid}(s, W)$
  for $c \in \bigcup_{w \in W} legal(s, w_{myid})$ **do**
    $\alpha[myteam] \leftarrow \max_{myid}(r, \alpha[myteam])$
    if $\exists t \neq myteam : \alpha[t] <_{parent[t]} r$ **then**
      return $\bot_t$
    $W_c \leftarrow \{w \in W \mid c \in legal(s, w_{myid})\}$
    $v \leftarrow tree(\alpha, (s, c), W_c, parent)$
    if $v = \bot_t$ **then**
      if $t \neq team$ **then**
        return $\bot_t$
    else
      $r \leftarrow \max_{myid}(r, v)$
  return r

function *chooseCard*(s, h)
  $W \leftarrow genWorlds()$
  for every team t **do**
    $parent[t] \leftarrow \emptyset$
    $\alpha[t] \leftarrow \bot_t$
  return $\arg criterion_{c \in legal(s,h)}(tree((s, c), \alpha, W, parent))$

tion. It has been extended to other frameworks, for example with [19] to partially ordered values with a cache. We will propose here a new generalization, not very sophisticated, but flexible to different specifications.

Imperfect Information Tree Search

Algorithm 3 captures the structure of a MiniMax tree search algorithm with vector α and set of worlds W. We have

- $<_i$ partial order on the pruning preferences of player i;
- $init_i(s, W)$, initialize the result for a player i, which has to be a lower bound on the actual value (depending on the worlds in W) of the node s for $<_i$
- $\max_i$ calculates the return value based on the values of the children. It is actually a fusion of the values of the children rather than a maximum over $<_i$.
- *genWorlds*() generates the worlds over which to compute the values, uses the distribution D of the players.
- *criterion*$_i$ a total order representing the preferences of player i for the final choice of the card to play

The complexity of this algorithm is hard to study, the partial order causing pruning is difficult to approach and it makes the model deviating from the ones developed in [16]. In practice, however, we can see a significant improvement of time complexity with this pruning. Now that we have a generic algorithm for solving the problem, with an a priori reasonable time complexity, we will compare different specifications of this algorithm.

We will explore different specifications of the template we presented. Each one is specified for two teams. The score of a final state is the score made by the *Max* (declarer) team. We tried to adapt it to multiple teams following the work of [28], but it is a long time work. This could probably be done in a future work. The first one is a direct extension of the usual $\alpha\beta$ pruning algorithm for perfect information games. The second one is the one proposed in [7]. The last one is a novel contribution. In each case we will present the specifications of the functions we have previously invoked. We will not describe the function $init_i$ in detail, because we only found a trivial lower bound over the value of a node, based on the score already made by a team in our trick-taking card games.

Perfect-Information Monte Carlo The idea of the Perfect-Information Monte Carlo (PIMC) algorithm is to generate random worlds according to D, and then to compute a score in each world *as if it were in perfect information* using the $\alpha\beta$ algorithm. The generation of the worlds follows directly from D. We generate a set of N worlds. Over the worlds in W, the *value* of a node is the score of each world in this setting of perfect information for everyone: $\mathbb{R}^N$. This hypothesis of perfect information in these worlds is not consistent with the settings. To illustrate it, this algorithm fits a game where everyone shows their hand, and plays openly after one card is played. Thus the generation of the worlds tries to capture the initial unknown, and then in each world, we play with perfect information. Let us talk about pruning. For the *Max* team a value a is greater than b if the score of each world $w \in W$ is greater in a than in b: $\forall k \in [N] a_k \geq b_k$ It is symmetric for *Min* nodes: $\forall k \in [N] a_k \leq b_k$ In this framework, we can derive the other function. The function $max_i(a, b)$ returns the maximum/minimum, according to the team of i, of each score of W. In fact with perfect information the players can choose the "objective" best card in each world. To make the final choice for the card to play, we set the criterion to prefer the best mean of the score, according to the team. The details are presented in Algorithm 4 (left).

$\alpha\mu$ Algorithm The problem with PIMC is that it assumes perfect information for everyone. It leads to several imperfections and difficulties in decision-making as shown in [13]. An example of a situation where PIMC fails is shown in Fig. 2 (left). South's expected score over these two tricks is 0.5. In fact, South has a 50% chance of discarding the useless ace on the ♣A and keeping the useful ace. In this case South scores 1, otherwise 0. However PIMC would generate several worlds, some with North having the ♠2, some with North having the ♡2, and in each one South would score 1. In fact, if South knows the North's remaining card (this is the case in the PIMC algorithm, where perfect information is given), it is easy to keep the right ace, and to win the last trick. Thus PIMC would average the score over the world sample and predict score 1 for South, which is a mistake.

The idea with the $\alpha\mu$ algorithm presented in [7] is to include the imperfect information of the player who has to choose a card in the reasoning. The player at the root of the tree doesn't know which world is the good one, but the others do and play with perfect information. The algorithm fits a game where one player has to play a card, but everyone else sees his/her hand. Although it is not perfect in terms of the information given, it is an improvement over PIMC.

Algorithm 4: PIMC (left) and $\alpha\mu$ (right)

```
function genWorlds
    W ← ∅
    for k ∈ [N] do
        w ↪ D
        W ← W ∪ {w}
    return W

function max_i(a, b)
    for k ∈ [N] do
        if Max team then
            r_k ← max(a_i, b_i)
        else
            r_k ← min(a_i, b_i)
    return r

function criterion_i(a, b)
  s_a ← Σ_{k=1}^{n} a_k
  s_b ← Σ_{k=1}^{n} b_k
    if Max team then
        return s_a > s_b
    else
        return s_a < s_b
```

```
function max_i(X, Y)
    R ← ∅
    if root then
        R ← X ∪ Y
    else
        for (x, y) ∈ X × Y do
            r ∈ ℝ^N
            for k ∈ [N] do
                if Max team then
                    r_k ← max(x_i, y_i)
                else
                    r_k ← min(x_i, y_i)
            R ← R ∪ {r}
    for r_1 ≠ r_2 ∈ R do
        if ∀k ∈ [N] r_1k ≤ r_2k then
            if root is Max then
                remove r_1 from R
            else
                remove r_2 from R
    return R

function criterion_i(X, Y)
    if Max team then
        s_X ← max_{x∈X} Σ_{k=1}^{n} x_k
        s_Y ← max_{y∈Y} Σ_{k=1}^{n} y_k
        return s_X > s_Y
    else
        s_X ← max_{x∈X} Σ_{k=1}^{n} x_k
        s_Y ← max_{y∈Y} Σ_{k=1}^{n} y_k
        return s_X < s_Y
```

We study it using information theory. Suppose C cards are remaining and player i knows the location of $|C| - k$ of those C cards, k being unknown. We will model D as the uniform distribution on the unknown cards over all players. These unknown cards can be dealt p^k different times. Thus, for $w \in \mathcal{W}$ we have $D(w) = 1/p^k$ (with p the number of players) if w matches the known cards, and 0, otherwise.

Knowing that we are playing in a world w^* (which is coherent with the known cards of i) gives the information $-\log(D(w^*)) = k \log p$. Note that this is also the KL-divergence between the certain distribution where $w*$ has a probability of 1 and D. We next measure the total information $I(n)$ given to the players when n cards are to be played. Consider that n cards are unknown at this state, and all the other $C - n$ cards are known. For a solution given by PIMC we compute the information gain contained in this solution. In Formula () every call of σ adds some information to this gain: the perfect information given to the node of the state. We have

$$I_{pimc}(n) = n \log p + \sum_{k=0}^{n-1} I(k) = n \log p + \sum_{k=0}^{n-1} (n-k)k \log p$$
$$= \log p(n + n(n^2-1)/6) \geq \log p(n^3/6)$$

Now consider a solution of $\alpha\mu$. Perfect information is

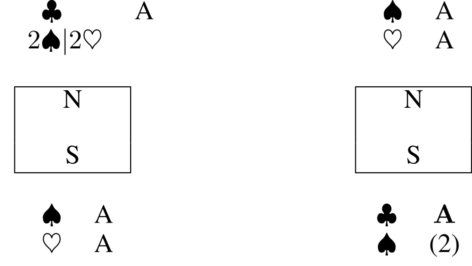


Figure 2: In this game we follow the rules of Bridge except we only play with two enemy players. Trump is clubs. (left) North plays ♣A and South is to play. South does not know if the second card of North is ♠2 or ♡2, it is a $50\% - 50\%$ situation. (right) Trump is clubs. North does not know if the second card of South is ♠2 or ♡2. South has to play.

only given to the non-root player. So, assuming we start with the root. we have

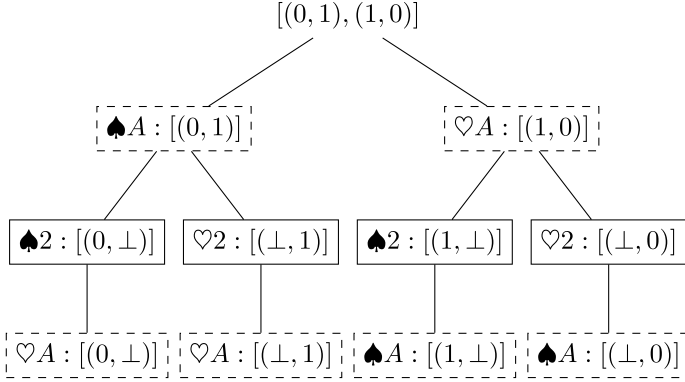
$$\begin{aligned} I_{\alpha\mu}(n) &= 0 + \sum_{k=0}^{n-1} (n-k)\delta_{p_{depth=k},root}(k \log p) \\ &= \log p\left(\sum_{k=0}^{n}(n-k)k - \sum_{i=0}^{n/p}(n-ip)ip\right) \\ &= \log p\left(\frac{n(n^2-1)}{6} - p^2\frac{\frac{n}{p}((\frac{n}{p})^2-1)}{6}\right) \\ &= \log p\left(\frac{n^3}{6}\right)\left(1 - \frac{1}{p} + \frac{p-1}{n^2}\right). \end{aligned}$$

Since the expression $\frac{p-1}{n^2}$ is small when we are not at the end of the game (in Bridge in middle game with 7 tricks left, we have $p = 4$, $n = 4 \cdot 7$), the information saved with $\alpha\mu$ is about a factor $(1 - 1/p)$. We noticed that with $\alpha\mu$ we achieve a reduction of the information gain by a factor $(1 - 1/p)$. This is not surprising, since we have removed the information gift of one player.

Let us introduce the functions used in $\alpha\mu$. This algorithm solves the problem of Fig. 2 (left). In Fig. 3 we see that the uncertainty about the outcome of the game is preserved. The value computed, $[(0,1),(1,0)]$ represents the fact that one strategy leads to a nonzero score in one world, and another strategy leads to a nonzero score in the other world.

In $\alpha\mu$ *generateWorlds* is the same as in PIMC; $\max_{root}$ computes the union of the values of the children, and removes the elements that are dominated (i.e., if the scores associated with one strategy are all better than another, it removes the latter); max_i is the union of the product of the strategies of the children. One element is computed by taking an element from all the children (we construct one possible strategy for root), and by taking in each world the best score for i, as in PIMC (if root follows the constructed strategy, then the score would be the one computed, with perfect information of i). Again, the set is simplified by removing the dominated elements for $root$; $criterion$ selects the node

Figure 3: Application of Fusion algorithm to the problem raised in Fig. 2 (left). The scores are the ones of South, viewed as a Max node. Max nodes are circled by continuous line, and Min nodes by dashed lines. Here world 1 is the world where North has ♠2 (the real world), and world 2 is the one where she has ♡2 (the fake world but South doesn't know it).



containing the strategy with the best expected score over the different worlds in W.

Fusion Algorithm

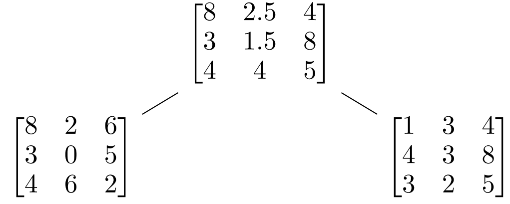
Even with $\alpha\mu$ we are giving away too much information to the players other than root. This can lead to bugs like the one shown in Fig. 2 (right). If South plays ♠2, s/he will score 1. If s/he plays ♣A s/he has 50% chance of scoring 2 and 50% chance of scoring 1. So ♣A is the better choice. $\alpha\mu$ supposes perfect information of North. Thus if South plays ♣A she will score 1, and if s/he plays ♠2 s/he will also score 1. Both cards are equal, and $\alpha\mu$ could make the mistake of choosing ♠2.

To solve this problem, we need to incorporate the imperfect information of all the players into our algorithm. This is the goal of the fusion algorithm. We choose to fix the value of a node to be the expected score of each world if everyone plays optimally according to their knowledge. At a node with two children, player i will choose the value of the child that has the best expectation, according to his/her team, over the worlds that are possible according to his/her knowledge. If the two children have the same expected value, player i has 50% chance of playing each one or them. So the value of the node should be a mixture of both (the middle of the score segment).

For example, if a Max node has two children $[2, 5, 3]$ and $[3, 7, 0]$, its value would be $[2.5, 6, 1.5]$ The problem with this simple idea is that different players have different knowledge, and different possible worlds. So each set of possible worlds P is different for each player. Moreover for a player, his/her set of possible worlds depends on its hand, that is unknown. We can't take one set W for the whole tree and assume it represents all the players, as we did before.

We compute the values over subsets of W that correspond to the different P possible for a player. For example, if both *Max* and *Min* can have two hands, there are four possible

Figure 4: Fusion of two values by a Max node. The row correspond to the different hands possible for Min, and the columns the one for Max (m_{ij} represents the score of the world where Min has hand i and Max hand j).



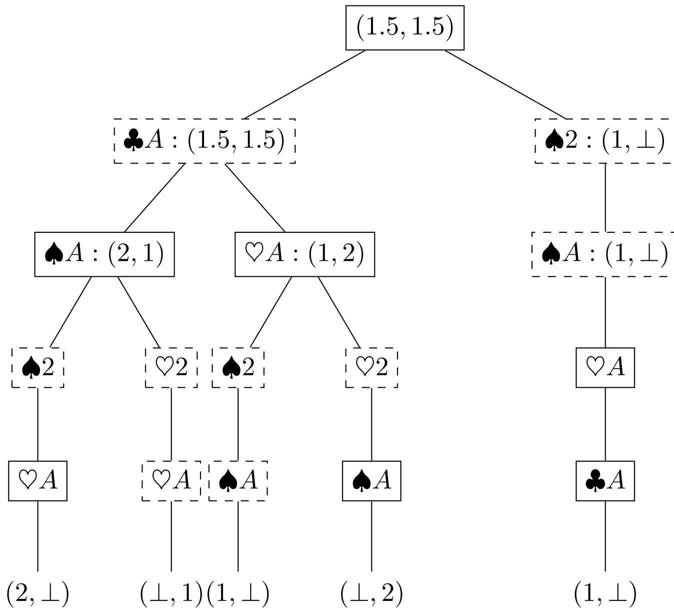
worlds $\{Max_1Min_1, Max_1Min_2, Max_2Min_1, Max_2Min_2\}$. Suppose *Max* is the root, and you want to merge two values at a *Min* node: $[0, 2, 3, 1]$ and $[2, 0, 2, 3]$. If *Min* has hand 1, the worlds she will consider are the first and third (you do not know the hand of *Max*. So the first value is better because it has an expected score of 1.5 instead of 2. But if *Min* has hand 2 the two values have the same expected score over worlds 2 and 4, which is 1.5. Thus *Min* has a 50% chance of playing each value and we mix the scores of these worlds. We come up with the value $[0, 1, 3, 2]$. A more detailed example is shown in Fig. 4. This algorithm solves the issue raised in Fig 2 as shown in Fig. 5. At the root, South has the choice of two values, over the actual world 1.5 and 1. Because $1.5 > 1$ s/he will then play ♣A.

In practice, taking any set W won't work to have this system of subsets of possible worlds. To have a correct number of worlds for each hand of each player, we need to have similarities between the worlds of W. To do this, we use a small set of initial worlds and we create the set W by taking all the possible permutations of the hands with this initial set. So in Alg. 5 *generateWorlds* samples some worlds and then returns all the permutations of those worlds. max_i follows the scheme explained before. *criterion*$_i$ selects the value with the best average score over the possible worlds of i. We prune if all the scores of one value are worse than another one.

Algorithm 5: Fusion

function $max_i(a, b)$
  for h possible hand of i **do**
    $score_a \leftarrow \sum_{w \in P_h} a_w$
    $score_b \leftarrow \sum_{w \in P_h} b_w$
    if $score_a < score_b$ **then**
      for $w \in P_h$ **do**
        if Max team **then** $r_w \leftarrow b_w$ *e*lse $r_w \leftarrow a_w$
    else
      for $w \in P_h$ **do**
        if Max team **then** $r_k \leftarrow a_w$ **else** $r_k \leftarrow b_w$
  return r

Figure 5: Application of the Fusion algorithm to the problem raised in Fig. 2 (right). The scores are the ones of South, viewed as a Max node. Here world 1 is the world where South has ♠2 (the real world), and world 2 is the one where she has ♡2 (the fake world but North doesn't know it).



Implementation

For the implementation, we have adapted the above formalization to the code. This allows us to keep some flexibility in the rules used later. We put the organization of the card game into one file. This file is for the general game, and when rule-dependent functions are needed, we call them in the specific rule file associated with them.

A simple modification in the Makefile selects which game is played and which rule file is read when these rule-dependent functions are used. It uses a game class that stores the information about the game being played: the contract, the cards already played, who has to play, the scores of the teams, what the current trick consists of, and who is the leader of that trick. We have a *player* class, which contains the information that a player has (his hand, what he knows about the other hands, etc). The communication with the player is realized in the methods of this class. The selection of the card to play is implemented by a function pointer, which is an attribute of the player class. Several functions have been created to select a card. We define this attribute of the player class to define its strategy.

We have collected all these different strategies in a directory. These are mainly the strategies described above. Since they often follow the same pattern, as we have explained, we used the algorithm of Figure 3. We then used an abstract class for each tree strategy. This abstract class is then specified into precise classes corresponding to each strategy that follows the template. In each class, the specification-dependent functions, such as $\max_i$, are implemented as attributes. To invoke the function that selects a card, we call the template with the class of the selected specification.

To encode the knowledge, the player class has an attribute *have_not*, which is an array containing the cards that this player knows that the other players do not have. This knowledge is used to generate the worlds used in the strategy functions, so that the generated worlds respect what the player knows about the others.

As for the basic types, we used bit vectors (mainly a set of unsigned integers) to represent the set of cards. For example, in Belote with its 32 cards, we used one unsigned integer. If a bit in a representing vector is set to true, then the corresponding card in the deck is contained in the set.

You can start the games by choosing which hands are played, which strategies are used by each player, and how many games are played. We ran several experiments with this structure.

Experiments

We tested the approach in two different games: Belote and Bridge. While Bridge is well-studied, our core interest is in (La) Belote, a French national card game for four players. For the player's strategy we implemented the three algorithms we discussed and a random card strategy, to have some kind of naive player. We then compared pairs of algorithms. In both cases there are two teams, so we assigned one strategy to one team and the other to the second team. We played matches of 100 different deal (same across the different experiments). With each deal, we play once as it is, and then swap the team's positions, so that it is fair. We measure the score made by the team of the first player to play. In each experiment we finally collect the 200 scores across these games. We add the 100 scores made by one strategy, and the 100 scores made by the other, and deduce the percentage of the total amount of points per game that on strategy wins over the other.

In each algorithm there are some hyperparameters to fix: n_{sample} is the number of worlds used in the algorithm. *depth-leaf* in $\alpha\mu$ and the fusion algorithm is the depth at which we stopped the specification to end the tree with PIMC (this is done for complexity issues, it is also done in [7]. *depth-rd* is the depth at which pimc is stopped to end the tree with the an average of the random playouts.

For Belote, we derived a 10% approximation of the score, that was valid with 95% probability with these random playouts. For Bridge, we imposed a low value of *depth-rd* for matters of complexity, and the bounds on the approximation made were not consistent. We tried to keep the same hyperparameters for all the algorithms, but Fusion needed more worlds than the others. All algorithms are better than random. We can also see that the $\alpha\mu$ algorithm is slightly better than the others. Let us note that all these algorithms are designed to play against smart players so the differences of the scores against random are not significant (PIMC being better than $\alpha\mu$ in beating random in Bridge is not relevant). Fusion algorithm is slightly worse than the others. However, pruning in the Fusion algorithm needs to be improved.

In Belote vs. another AI PIMC ($n_{sample} = 10$, *depth-rd* $=$ 25) we found a 9.7% gain in the max. score on average.

	rd	pimc	$\alpha\mu$	fusion
rd	-1.8	-19.3	-22.1	-18.5
pimc	19.3	-3.3	-0.2	1.5
$\alpha\mu$	22.1	0.2	2.4	0.8
fusion	18.5	-1.5	-0.8	3.5

Figure 6: Belote; team 0 in rows, team 1 in columns. Pimc: $n_{sample} = 10$. $\alpha\mu$: $n_{sample} = 10$, *depth-leaf* $= 5$, *depth-rd* $= 20$. Fusion: $n_{sample} = 24$, *depth-leaf* $= 5$, *depth-rd* $= 20$.

	rd	pimc	$\alpha\mu$	fusion
rd	1.5	-25.2	-21.8	-21.6
pimc	25.2	-0.7	-1.8	5.0
$\alpha\mu$	21.8	1.8	-2.1	2.7
fusion	21.6	-5.0	-2.7	0.5

Figure 7: Bridge; team 0 in rows, team 1 in columns. Pimc: $n_{sample} = 10$. $\alpha\mu$: n_{sample}, *depth-leaf* $= 5$, *depth-rd* $= 9$, Fusion: $n_{sample} = 24$, *depth-leaf* $= 5$, *depth-rd* $= 20$.

Conclusion

We presented an efficient framework for imperfect information card games and found a factorization of the search in a template that captures the main ingredients of the search. We then analyzed three different algorithms and derived the impact of the information given during the algorithm on the decisions it makes. Finally, we designed a new algorithm that tries to respect known information more. We also estimated the effects of truncating the search tree with random rollouts, and the information gain obtained by $\alpha\mu$. To continue this work, we need to explore more deeply the possibilities for a better pruning of the tree. Last, but not least we will aim at card games with *talon* that do not yet fit the setting.

Acknowledgments

This research was partly funded by AFOSR project Flexible and Resilient Auton. Systems (FRAS) and by the Czech Science Foundation, grant number 22-30043S.

References

[1] L. V. Allis. A knowledge-based approach to connect-four. the game is solved: White wins. Master's thesis, Vrije Univeriteit, The Netherlands, 1998.

[2] X. Blanvillain. Oware is strongly solved. In *Computers and Games*. Springer, 2022.

[3] M. Bowling, N. Burch, M. Johanson, and O. Tammelin. Heads-up limit hold'em poker is solved. *Commun. ACM*, 60(11):81–88, 2017.

[4] A. Brennan, S. Kharroubi, A. O'Hagan, and J. Chilcott. Calculating Partial Expected Value of Perfect Information via Monte Carlo Sampling Algorithms. *Medical Decision Making*, 27(4):448–470, July 2007.

[5] M. Buro, J. R. Long, T. Furtak, and N. R. Sturtevant. Improving state evaluation, inference, and search in trick-based card games. In *IJCAI*, pages 1407–1413, 2009.

[6] T. Cazenave and V. Ventos. The $\alpha\mu$ search algorithm for the game of bridge. *CoRR*, abs/1911.07960, 2019.

[7] T. Cazenave and V. Ventos. The $\alpha\mu$ Search Algorithm for the Game of Bridge. In *Monte Carlo Search*, Communications in Computer and Information Science, pages 1–16, Cham, 2021. Springer.

[8] G. Cohensius, R. Meir, N. Oved, and R. Stern. Bidding in spades. In *ECAI*, pages 387–394, 2020.

[9] S. Edelkamp. Challenging Human Supremacy in Skat. In *Twelfth Annual Symposium on Combinatorial Search*, July 2019.

[10] S. Edelkamp. Knowledge-Based Paranoia Search. In *2021 IEEE Conference on Games (CoG)*, pages 1–8, Aug. 2021. ISSN: 2325-4289.

[11] S. Edelkamp. Improving computer play in Skat with hope cards. 2023. To appear.

[12] H. Finnsson. *Cadia-player: A general game playing agent*. PhD thesis, 2007.

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

[14] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, ETH Zürich, 1995.

[15] M. Ginsberg. Step toward an expert-level Bridge-playing program. In *IJCAI*, pages 584–589, 1999.

[16] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, Dec. 1975.

[17] S. Kupferschmid. Entwicklung eines Double-Dummy Skat Solvers mit einer Anwendung für verdeckte Skatspiele. Master's thesis, Albert-Ludwigs-Universität Freiburg, 2003.

[18] J. Li, B. Zanuttini, T. Cazenave, and V. Ventos. Generalisation of alpha-beta search for AND-OR graphs with partially ordered values. In *IJCAI*, pages 4769–4775. ijcai.org, 2022.

[19] J. Li, B. Zanuttini, T. Cazenave, and V. Ventos. Generalisation of alpha-beta search for AND-OR graphs with partially ordered values. Research Report, GREYC CNRS UMR 6072, Universite de Caen, May 2022.

[20] J. R. Long, N. R. Sturtevant, M. Buro, and T. Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[21] M. Moravcík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. H. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017.

[22] J. Perolat, B. D. Vylder, D. Hennes, E. Tarassov, F. Strub, V. de Boer, P. Muller, J. T. Connor, N. Burch, T. Anthony, S. McAleer, R. Elie, S. H. Cen, Z. Wang, A. Gruslys, A. Malysheva, M. Khan, S. Ozair, F. Timbers, T. Pohlen, T. Eccles, M. Rowland, M. Lanctot, J.-B. Lespiau, B. Piot, S. Omidshafiei, E. Lockhart, L. Sifre, N. Beauguerlange, R. Munos, D. Silver, S. Singh, D. Hassabis, and K. Tuyls. Mastering the game of stratego with model-free multiagent reinforcement learning. *Science*, 378(6623):990–996, dec 2022.

[23] J. W. Romein and H. E. Bal. Solving awari with parallel retrograde analysis. *Computer*, 36(10):26–33, 2003.

[24] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Solving checkers. In *IJCAI*, pages 292–297, 2005.

[25] S. Schiffel and M. Thielscher. Reasoning about general games described in gdl-ii. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 846–851. AAAI Press, 2011.

[26] D. Silver and A. H. et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484, 2016.

[27] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. Technical Report 1712.018, arxiv, 2017.

[28] N. R. Sturtevant and R. E. Korf. On pruning techniques for multi-player games. *AAAI/IAAI*, 49:201–207, 2000.

[29] N. R. Sturtevant and A. M. White. Feature construction for reinforcement learning in hearts. In *Computers and Games*, pages 122–134. Springer, 2006.