Artificial Intelligence for Games

Bruno Bouzy, Tristan Cazenave, Vincent Corruble, Olivier Teytaud

Artificial Intelligence for Games

Bruno Bouzy, Tristan Cazenave, Vincent Corruble, and Olivier Teytaud

Abstract This chapter presents the classical alpha-beta algorithm and several variants, Monte Carlo Tree Search which is at the origin of recent progresses in many games, techniques used in video games and puzzles, and retrograde analysis which performs perfect play in endgames.

1 Introduction

As widely repeated, games are the drosophilia of AI - an excellent testbed for techniques. Section 2, based on alpha-beta methods, presents powerful techniques for the fully observable case when situations can be approximately evaluated efficiently - e.g. in Chess, for which counting 10 points for a queen, 5 for a rook and so on provides a simple but effective approximation. Section 3 presents Monte Carlo methods, powerful in partially observable cases, and also, with the Tree Search adaptation, when situations are hard to evaluate. Section 4presents methods used for puzzles - these methods, including Monte Carlo variants and applications of A*, are also quite effective for industrial applications. Section 5 presents retrograde-analysis, which solves exactly various problems (and is also widely used in supply

Bruno Bouzy
LIPADE UFR de mathematiques et d'informatique Universite Rene Descartes 45, rue des Saints-Peres 75270 Paris Cedex 06, France

Tristan Cazenave
LAMSADE - Universit Paris-Dauphine Place du Marchal de Lattre de Tassigny 75775 Paris Cedex 16 , France

Vincent Corruble
Sorbonne Universit - LIP6 Bote courrier 169 Couloir 25-26, tage 4, Bureau 412 4 place Jussieu 75252 Paris Cedex 05, France

Olivier Teytaud
TAU, Inria, Bt 650, Rue Noetzlin, 91190 Gif-sur-Yvette, France

chain management), and Section 6 presents tools for video games and is also applicable in military applications.

2 Minimax, Alpha-Beta and enhancements

This section presents Minimax, Alpha-Beta and its enhancements. Alpha-Beta is a tree search algorithm used in two-player games [Schaeffer and van den Herik, 2002], [Allis, 1994]. Its development is strongly associated to computer Chess between 1950 [Shannon, 1950] and 1997 when Deep Blue beat the world Chess champion Gary Kasparov [Anantharaman et al., 1989], [Campbell et al., 2002]. Its origin is hard to determine exactly, because it is described by numerous papers [Campbell and Marsland, 1983], [Knuth and Moore, 1975], [Marsland, 1986]. Alpha-Beta is an enhancement of Minimax [von Neumann and Morgenstern, 1944] described in Section 2.1. Regarding Minimax, the specificity of Alpha-Beta is pruning important parts of the search tree without losing correctness (section 2.2). Practically, the efficiency of Alpha-Beta is dependent of various enhancements explained in the following sections: transposition tables (section 2.3), Iterative Deepening (ID) (section 2.4), MTD(f) (section 2.5), and many others (section 2.6). All these enhancements assume the search is performed with a fixed depth or horizon. Other algorithms descending from Minimax are worth mentioning (section 2.7).

2.1 Minimax

Minimax is a tree search algorithm for two-player zero-sum games with alternating moves [von Neumann and Morgenstern, 1944]. It assumes that a position can be evaluated by an evaluation function at a fixed depth. The friendly player intends to reach positions which maximize the evaluation, and the opponent intends to reach positions which minimize the evaluation. On friendly nodes (respectively adversarial nodes), the minimax value is the maximum (respectively minimum) of the minimax values of the child nodes. For a tree with branching factor b and depth d, Minimax visits b^d nodes.

2.2 Alpha-Beta

Alpha-Beta (AB) computes the minimax value of the root node by visiting less nodes than Minimax does. For this and for each node, AB computes an AB value, called v here, and uses two values, α and β, bounding the interval in which v is situated. The friendly player (respectively the opponent) aims at maximizing (respectively minimizing) v. α (respectively β) is the minimal (respectively maximal)

AB value that the friendly player (respectively the opponent) is sure to obtain, given the search performed so far. At any node, we have $\alpha \leq v \leq \beta$ and $\alpha < \beta$. At the beginning, the current node is the root node and we have $\alpha = -\infty$ and $\beta = +\infty$. The root node is a friendly node. On a friendly node, AB recursively calls AB on each child node with the two parameters α and β. The order in which the calls are performed is important and can be given by domain knowledge. Let r be the value returned by AB on a child node. If $r \geq \beta$, AB stops and returns β (by definition of β, r cannot be superior to β, and the maximal value AB is looking for is greater than r). This case is called a β cut because the following childs are not visited. If $\beta > r > \alpha$, AB improves its maximal value so far, and α is updated with r. If $r \leq \alpha$, AB continues by calling AB on the next child node. When all child nodes have been visited or cut, AB returns α. On an adversarial node, AB proceeds analogeously: if $r \leq \alpha$, AB stops and returns α. This is called an α cut. If $\alpha < r < \beta$, the opponent improves what he could get so far and β is updated with r. If $r \geq \beta$, AB continues. When all the child nodes have been visited, AB returns β.

If Alpha-Beta, launched with initial values α and β, returns value r, then the following results are garanteed. If $\alpha < r < \beta$ then r is the minimax value. If $\alpha = r$ then the minimax value is smaller than α. If $r = \beta$ then the minimax value is greater than β. The size of the memory used by Alpha-Beta is linear in d.

Alpha-Beta efficiency depends a lot on the order in which AB explores the child nodes of a given node. This order is often given by domain knowledge or search heuristics. Exploring the best node in first brings about many cuts and shortens the search. So as to know the minimax value of the root node of a tree with depth d and branching factor b, [Knuth and Moore, 1975] shows that AB explores a number of nodes greater than $2b^{d/2}$ approximately, and this number can be reached when the order of exploration is sufficiently good. T being the number of nodes explored by Minimax, this means that AB explores approximately $2\sqrt{T}$ nodes in good cases.

Practically, Alpha-Beta is used in its NegaMax version. Negamax does not explicitly distinguish friendly nodes and opponent nodes, but recursively calls - NegaMax with $-\beta$ and $-\alpha$ as parameters.

2.3 Transposition Table

In practice, AB can be enhanced with a transposition table (TT) which stores all the results of the searches starting from a given position. Its interest lies in the fact that two different nodes in the search tree could correspond to the same position. This situation happens very often. The simplest case is when two sequences of moves A, B, C and C, B, A lead to the the same position. With a TT, searching twice on the same position is avoided. After each search starting on a given node, AB stores the result in the TT, and each time a node is visited, AB looks into the TT whether there is an existing result corresponding to the position of the node.

To represent a game position with an index, the first idea is to map the position with a non negative integer inferior to $|E|$, the size of the game state space. $|E|$ can

be huge. For 9×9 Go, $|E| \simeq 3^{81} \simeq 10^{40} \simeq 2^{133}$ and this first idea is not realistic on current computers. The position must be mapped to a smaller number, risking to observe collisions: two different positions with the same index (type 1 collision). Zobrist designed a coding scheme enabling a program to use index of positions on current computers (a 32-bit or a 64-bit number) with an arbitrary small probability of collision.

For each couple (property, value) of the position, a random number (a 32-bit or a 64-bit number) is set up offline, once for all and beforehand. Online, the position which can be defined by the set of all its couples has a Zobrist value which equals the XOR of all its couple random numbers. When a move is played, the Zobrist value is incrementally updated by xoring the Zobrist value of the position with the random numbers of the couple (property, value) of the position that have been changed. Zobrist has shown that, for a search with a fixed number of nodes, a type 1 collision has a probability of happening that can be arbitrarily small provided that the number of bits of the random numbers is sufficient [Zobrist, 1990]. This mechanism is called Zobrist hashing.

In practice, the size of the TT is fixed, say 2^L (with $L = 20$ or $L = 30$). The index of the position is made up with the first L bits of the Zobrist value of the position. Collisions happen in the TT when two positions have the same index (type 2 collision), which is frequent. To avoid type 2 collision, the Zobrist value of the position is stored with its search result. When reading an entry in the TT, the tree search checks that the Zobrist value of the position equals the Zobrist value of the entry, and the search result contained in the entry is used. The first Chess programs used TT [Greenblatt et al., 1967]. Nowadays, Zobrist hashing is currently used for many games.

2.4 Iterative Deepening

AB is a depth-first search algorithm. If the optimal solution is short and situated below the second node of the root node, AB explores all the nodes situated before the first node before entering the second node. It may spend a useless time below the first node before finding the optimal solution below the second node. To prevent this problem, Iterative Deepening (ID) calls AB with a fixed depth 1, then 2, and so on and so forth, iteratively while computing time is available [Korf, 1985a]. ID is anytime. ID finds the shortest solution. ID can be used with a TT. A subsequent and deeper search can use the results of a previous and shallower search. Particularly, if the best move of a shallow search is stored in the TT, a subsequent and deeper search can search this move first to produce cuts [Slate and Atkin, 1977].

2.5 MTD(f)

Rather than launching AB with $\alpha = -\infty$ and $\beta = +\infty$, AB can be launched with any values provided that $\alpha < \beta$. Let v be the AB value of the root. It can be shown that iff $v \leq \alpha$, then AB returns α. Similarly, iff $v \geq \beta$, then AB returns β. Iff $\alpha < v < \beta$, then AB returns v. The minimal-window idea is to set up $\beta = \alpha + 1$. The corresponding search produces many cuts and its computing time is significantly smaller than the computing time of the search launched with $\alpha = -\infty$ and $\beta = +\infty$. If AB returns $\alpha + 1$, then $\alpha + 1$ is a lower bound of v: $v \geq \alpha + 1$. If AB returns α, then α is an upper bound of v: $v \leq \alpha$.

Memory Test Driver (MTD) names a class of algorithms [Plaat et al., 1996] using the minimal-window principle. MTD(f) is the simplest and the most efficient one. MTD(f) iteratively calls AB with $\alpha = \gamma$ and $\beta = \gamma + 1$. The initial value of γ can be a random value or the result of a previous MTD(f) search. At each iteration, if AB returns γ, then $v \leq \gamma$ and γ is decremented. Otherwise, $v \geq \gamma + 1$ and γ is incremented. After a finite number of iterations v is known and the best move read in the TT. At the expense of using a TT and ID, MTD(f) is a significant enhancement of AB, used in current Chess programs.

2.6 Other Alpha-Beta Enhancements

Other AB enhancements exist. First, *Principal Variation Search* (PVS) assumes that the nodes are already well ordered by the knowledge-based move generator. Consequently, PVS is designed to check this order [Pearl, 1980b], [Pearl, 1980a]. PVS calls PVS on the first child node with a minimal window so as to check that α cannot be surpassed. If this is the case, the computing time is low. Otherwise, a normal AB search is launched on this node, which costs a second search. Secondly, the null move heuristic [Donninger, 1993] launches a shallow search assuming the first move is a null move, which gives a first value to α at a low cost. Thirdly, the history heuristic [Schaeffer, 1989] assesses the moves in term of number of cuts, and stores the results in a table. The moves with a good assessment in the table are tried first later on. This heuristic assumes that the moves can be the same from one position to another. In Chess, a move can be identified by its kind of piece, its origin and its destination. Fourthly, *Quiescence search* [Beal, 1990] searches while the position is not quiet, i.e. at least one urgent move exists (for instance capturing a piece in Chess). [Rivest, 1988] studies the back-up formula. Finally, [Junghanns, 1998] is an overview of Alpha-Beta.

2.7 Best First Search

Other algorithms improve Minimax by exploring the best moves in first, the depth of the search not being fixed beforehand. *Proof Number Search* (PNS) [Allis et al., 1994] is useful in a AND-OR tree. PNS computes the proof number of a node: the number of nodes to explore under this node so as to prove its value. PNS explores in first the node with the lowest proof number. *Best-First Search* [Korf and Chickering, 1994] calls the evaluation function for all child nodes and explores the best node first. SSS* [Stockman, 1979] explores all nodes in parallel as A* would do it with a specific heuristic. B* [Berliner, 1979] uses an optimistic evaluation and a pessimistic one. B* searches so as to prove that the pessimistic value of the best node is better than the optimistic value of the second best node. [McAllester, 1988], [Schaeffer, 1990] define conspiracy nodes. A conspiracy node is a leaf node whose evaluation influences the Minimax value of the root. The conspiracy nodes are searched first.

3 Monte Carlo Search

A major change occurred recently in the game of Go. In 1998, Martin Mueller (amateur 6 Dan) could win against Many Faces of Go, the best program at that time, with an astronomic handicap of 29 stones[Müller, 2002]. In 2008, MoGo, from the French Monte Carlo Go school, won with a decent 9 stones handicap against Kim Myungwang, 8 Dan pro. Later on, programs won with reduced handicap, and then AlphaGo[Silver et al., 2017b], from Google Deepmind, won without handicap against the very best professional players. In [Silver et al., 2017a] this was reproduced for several games without using any human knowledge; and [Tian et al., 2018] releases an open source version also beating the best professional players.

These successes came from algorithmic improvements. The same Monte Carlo techniques were used in active learning[Rolet et al., 2009], in optimization of grammars[De Mesmay et al., 2009], in non-linear optimization[Rolet et al., 2009]. Simultaneously, related algorithmes were used in planning. In the case of AlphaGo, the Monte Carlo method was combined with deep networks (Chapter 12 of the present volume).

3.1 Monte Carlo Evaluation

The first use of simulated annealing for ranking a list of moves goes back to [Bruegmann, 1993]. The state of the art was the alpha-beta pruning; it works quite well for checkers of chess but it needs a decent and fast evaluation function. An evaluation function is a mapping from a board position to an evaluation of the value of this position for each of the players.

Typically, a human expert can write a good evaluation function for checkers or chess; whereas this does not exist in Go. [Bruegmann, 1993] proposed a work-around: randomly simulate many games, for approximating a winning probability (see Alg. 1); Monte Carlo Go was born.

Algorithm 1 Evaluation of a position p by the Monte Carlo method via n simulations.

Input: a position p, a number n of simulations.
for $i = \in \{1,\ldots,n\}$ **do**
  Let $p' = p$.
  while p' is not a final state **do**
    $c =$random move among legal moves at p'
    $p' = transition(p',c)$
  end while
  if p' is a win for black **then**
    $r_i = 1$
  else
    $r_i = 0$
  end if
end for
Return $\frac{1}{n}\sum_{i=1}^{n} r_i$ (estimated probability of gain for black).

While the Monte Carlo method is old (we usually trace it back to the Manhattan project, i.e. the project for the construction of the nuclear bomb in united stated during world war 2, but it has also been pointed out much early as an original method for approximating π), its use in games was then new.

3.2 Monte Carlo Tree Search

The technique was producing convincing results; it was further developed in [Bouzy and Cazenave, 2001; Bouzy and Helmstetter, 2003] and improved by combination with search and with knowledge [Bouzy, 2005; Cazenave and Helmstetter, 2005].

Nonetheless, the real "take off" of the performance will be the combinatiojn with an incremental tree building. The resulting algorithm, termed *Monte Carlo Tree Search* [Coulom, 2006; Chaslot et al., 2006] is presented in Alg. 2 The structure T is a set of situations, progressively augmented by adding, at each random simulation, the first situation in this simulation which had not yet been stored in it; this structure stores, for each node, a number of wins for black and a number of wins for white.

The default policy part does not have to be a pure default random - the early successes of MoGo were due to the use of a sophisticated default policy [Gelly et al., 2006]. Combinations with tactical solvers (solving local situations) have been tried without clear success.

This technique is currently applied in all strong programs in the game of Go, and in many games:

Algorithm 2 Evaluation of a position p by the Monte Carlo Tree Search technique with n simulations. Notations: ¬white = black, ¬black = white; $transition(p,c)$ is the situation in which we move when the player to play chooses move c in position p.

Input: p a position, n a number of simulations.
$T \leftarrow$ empty structure.
for $i = \in \{1,\ldots,n\}$ **do**
  Let $p' = p$, $q = \emptyset$, $game = \emptyset$.
  while p' is not a final state //Do a complete game **do**
    if p' is in T // If p' is in memory **then**
      $j =$ player to play in p' // Algorithm called "bandit"
      for each c legal move in p' **do**
        //Compute the score for each move as follows
        $p'' = transition(p',c)$
        $Score(c) = banditFormula(T(\neg j,p''),T(j,p''),T(j,p')+T(\neg j,p'))$
      end for
      $c =$ legal move in p' maximizing $Score(c)$
    else
      if $q = \emptyset$ // We have not yet found the state to be added
      then
        game← game + p'
      end if
      $c =$random move among legal moves in p' // default policy
    end if
    $p' = transition(p',c)$
  end while
  Add q in T // if $q \neq \emptyset$
  if p' is a win for black **then**
    $r_i = 1$
  else
    $r_i = 0$
  end if
  for p' in game **do**
    $T(r_i,p') = T(r_i,p') + 1$ // Increase $T(r_i,p')$
  end for
end for
Output: $\frac{1}{n}\sum_{i=1}^{n} r_i$.

- Monte Carlo (not MCTS): Scrabble world champion beaten by MC [Sheppard, 2002]
- General Game Playing (Cadiaplayer world champion [Finnsson and Björnsson, 2008a])
- Hex (MCTS world champion [Arneson et al., 2010])
- Havannah [Teytaud and Teytaud, 2009]
- Arimaa (game built specifically built for being hard for computers [Kozelek, 2009])
- Nogo [Chou et al., 2011]
- Fortress positions in chess (folklore claim)
- Hide and seek "Scotland Yard" [Nijssen and Winands, 2012]

- Chinese Checkers, Focus, Rolit, Blokus [Nijssen and Winands, 2013]
- Amazons, Breakthrough, M. Jack, Chinese Military Chess, real-time video games (Ms Pac-Man), Total war:Rome, Poker, Skat, Magic: The Gathering, Settlers of Catan, 7 wonders...

Monte Carlo is used in various puzzles: SameGame [Schadd et al., 2008]; Morpion Solitaire (state of the art by Nested-rollout MCTS [Rosin, 2011; Cazenave et al., 2016]); Samurai Sudoku [Finley, 2016]; and in applications far from games: operations research [Chang et al., 2005]; sometimes claimed to be an early variant of MCTS); Linear Temporal Logic problems, including car driving [Paxton et al., 2017]; traveling salesman problem with time windows (Nested Rollout method; [Cazenave and Teytaud, 2012]); unit commitment (an early combination of neural nets and MCTS, [Couetoux, 2013]); continuous uncertain industrial problems [Couetoux, 2013]; sailing [Kocsis and Szepesvari, 2006].

A particularly interesting point is the so-called "general game playing" (GGP[Pitrat, 1968]); in these competitions, the program has to read the rules (in a given format, usually "game description language"), and then play. The best GGP programs use MCTS [Finnsson and Björnsson, 2008b].

Algorithm 2 does not specify what is the "bandit" formula. A classical variant, though not the most widely used in the case of Go, is UCT (*Upper Confidence Tree* [Kocsis and Szepesvari, 2006]) as follows:

$$banditFormula(w,l,n) = w/(w+l) + \sqrt{K\log(n)/(w+l)} \quad (1)$$

(where K is an empirically chosen constant, n is the number of simulations at the considered situation, w the number of wins for the considered move, and l the number of losses; $n = w+l$ for games without draw). The first term $v/(v+d)$, called exploitation, is in favor of moves which have a high success rate; the second term is in favor of moves which are not much explored ($w+l$ is small) and is therefore called exploration term. The formula 1 is not properly defined for $v+d=0$; it is frequent to specify

$banditFormula(w,l,n) = F$ when $w+l=0$, for a given constant F [Gelly et al., 2006].

Different modifications of this formula have been proposed. The RAVE formula (*Rapid Action Value Estimates*), based on so-called AMAF (*All Moves As First*) values, is as follows:

$$banditFormula(w,l,w',l') = \alpha(w+l)w/l + (1-\alpha(w+l)w'/l'.$$

We use the same w and l as in UCT, and we also use for the bandit formula for a given situation:

- w' the number of wins in which the considered move c has been played by the player to play in the considered situation *before* being played by the other player, even if this was not played in the considered situation.

- l' the number of losses in which the considered move c has been played by the player to play in the considered situation *before* being played by the other player, even if this was not played in the considered situation.

$\alpha(.)$ is a function converging to 1, for example $\alpha(n) = n/(n+50)$:

- if $w+l$ is much larger than 1, we use these values and their ratio; whereas when w and l are too small for the ratio $w/(w+l)$ to have a meaning;
- then, as the number of simulations increases, we move to $w/(w+l)$ which is asymptotically better (less biased) than $w'/(w'+l')$.

A second important modification [Coulom, 2007; Lee et al., 2009; Chaslot et al., 2008] consists in using heuristics tuned on databases; a simple method for using a heuristic $h(p',c)$ (typically estimating the frequency at which a move c is played in situation p', given the configuration p' around move c, is:

$$banditFormula(w,l,p',c) = w/(w+l) + Kh(p',c)/(w+l)$$

for some empirically tuned constant K.

Strong results can be obtained by combining these different approaches [Lee et al., 2009].

After the wide success of deep neural networks for various tasks, in particular pattern recognition tasks, neural networks have been used for estimating $h(p',c)$ (such a network is called a critic network), with p' the entire board[Silver et al., 2016]. Deep neural networks were also used for generating the so-called default policy in Alg. 2 (such a network, actually playing moves, is called an actor network). The training can be done in different manners; a possibility is as follows:

- learn the actor network $move = random(board)$ by the reinforce method[Williams, 1992], i.e. by self-play (the network plays against itself and applies gradient updates to its weights);
- learn the critic network $h(p',c)$ by classical supervised learning on the situations met in self-play games;

Using this method, combined with MCTS, AlphaGo won a long uninterrupted series of games against the best professional players in the world [Silver et al., 2017b].

MCTS has the following advantages:

- scaling: the program becomes stronger if the computational power increases;
- very low need for human expertise; the algorithm presented in section 2 is independent of the game; even the heuristic $h(.,.)$ might be tuned automatically on databases, though human expertise can help.

Due to the nice scaling properties of MCTS, parallelization has been applied[Cazenave and Jouandeau, 2007; Gelly et al., 2008]; however, results, if numerically good in the sense that MCTS running on dozens of CPU does outperform the single CPU version, keep the same limitations; it looks like the performance against humans does not increase as much as suggested by the performance against the non-parallel

version of the code. The parallel code remains, at least when no special trick and no deep network is applied, unable to evaluate so-called "capturing races" (also known as "semeais"); the program tends to always believe that semeais are won with probability 50% whenever humans know clearly that it's a win for e.g. black with probability 100%.

An improved version of AlphaGo Zero named AlphaZero has been proposed [Silver et al., 2017a]. Apart from the game of Go it has been applied to Chess and Shogi. After a few hours of training it has been able to defeat the best computer Chess and Shogi players, Stockfish and Elmo. For the game of Go it has surpassed AlphaGo Zero [Silver et al., 2017b] and it is considered as a more generic version of AlphaGo Zero.

4 Puzzles

Puzzles are one player problems where we search for a sequence of moves that gives the solution of the problem. Algorithms can either optimize the number of moves or the score of the solution.

*4.1 A**

The A* algorithm (cf. chapter **??**) [Hart et al., 1968] enables to find solution with a minimal number of moves to various puzzles. Examples of such puzzles are the Rubik's Cube [Korf, 1997], the 9-puzzle or Sokoban [Junghanns and Schaeffer, 2001]. A* is also used in video games [Cazenave, 2006a; Bulitko et al., 2008; Sturtevant et al., 2009].

For each puzzle adressed by A*, an admissible heuristic has to de defined. This heuristic will be computed for every state of the search. An heuristic is admissible when it always gives a value smaller than the true number of moves required to reach the solution from the evaluated state.

4.1.1 The Manhattan Heuristic

The most widely used heuristic is the Manhattan heuristic. The principle of the Manhattan heuristic is to compute very rapidly the solution of a simplified problem and to use the cost of this solution as a lower bound of the real cost. It calculates for each piece the cost of moving it to its goal without taking into account the interactions with the other pieces. The heuristic is the sum of all these calculations for all the pieces. For example in the 9-puzzle the heuristic counts for each tile the number of moves to move it to its goal location as if there were no other tile. For the Rubik's cube, the same calculation if done for every cube, however each move at the Ru-

bik's cube moves eight cubes, so the sume has to be divided by eight in order to be admissible. For finding the optimal moves on maps of video games, the Manhattan heuristic calculates the distance as if there were no obstacle.

4.1.2 Tree Search

A* developes at each search step the state that has the lowest estimated cost. The cost of a path is the cost already used to reach the state plus the lower bound on the remaining cost to reach the goal. It ensures that when a solution is found it has a minimal cost (all other states that could be developed have a greater associated cost). In games, the cost of a state is often the number of moves required to reach the goal.

Algorithm 3 Search of a minimal cost solution with A*

```
Input: a position p.
Open ← {p}.
Closed ← {}.
g[p] = 0
h[p] = estimated cost from p
f[p] = g[p] + h[p]
while Open ≠ {} do
    pos = position in Open with the smallest f
    if pos is the goal then
        return the path to pos
    end if
    remove pos from Open
    add pos to Closed
    for c =legal move of pos do
        pos' = transition(pos, c)
        g' = g[pos]+cost of c
        if pos' is not in Closed then
            if pos' is not already on Open with g[pos'] ≤ g' then
                g[pos'] = g'
                h[pos'] =estimated cost from pos'
                f[pos'] = g[pos'] + h[pos']
                add pos' to Open
            end if
        end if
    end for
end while
return fail
```

4.2 Monte Carlo

The recent success of Monte Carlo methods for games has brought them as interesting candidates for solving puzzles. Monte Carlo are suited to puzzles lacking a good heuristic to guide the search. It is the case for puzzles such as Morpion Solitaire or SameGame. For these two games as well as for Sudoku and Kakuro, a nested Monte Carlo search has good results [Cazenave, 2009]. The principle of this algorithm is to play random games at the lowest level and to choose for upper levels the move that resulted in the best score of a playout of the underlying level (for example each move of a playout of level one is chosen after the score of a random game starting with the move). Moreover the methods described in the previous section *Monte Carlo Search* are general and can be applied to puzzles.

4.3 Further Reading

It is possible for some puzzles to us a depth-first version of A* named Iterative Deepening A* (IDA*) [Korf, 1985b] that enable to use A* with very few memory.

[Kendall et al., 2008] is a nice survey of NP-complete puzzles.

5 Retrograde Analysis

Retrograde analysis enable to precompute a solution for each element of a subset of the states of a game. It has enabled to optimally solve a few games. We first present its application to endgames of two player games, then to puzzles.

5.1 Endgame Tablebases

The principle underlying an endgame tablebase is to calculate the exact value of some endgame states. For example in Chess it is possible to calculate for every state containing five pieces or less its exact value. Ken Thompson calculated all values for six pieces endgames [Thompson, 1996].

Retrograde Analysis is the algorithm used to calculate the score of endgame states. The principle is to start enumerating all won states. Then for each won state, it undoes a White move and a Black move and verifies with a depth two search the status of the new state. It then finds new won states. It continues this process of finding new won states as long as it finds new won states.

Endgame Tablebases are used by Chess programs as they can play some endgames instantly and better than any human. They have changed Chess theory for certain

won states that human players thought to be draw (notably the King-Bishop-Bishop-King-Knight).

Retrograde analysis can also be used in other games. Chinook solved Checkers using some endgames tablebases and search algorithms [Schaeffer et al., 2007]. Another popular game completely solved by retrograde analysis is Awari, there exist a program that can play perfectly and instantly all the Awari states [Romein and Bal, 2003].

5.2 Pattern Databases

In order to improve A* or IDA* on a given problem, it is natural to try improving the admissible heuristic. Improving the admissible heuristic reduces to make it find greater values for an equivalent computing time.If the heuristic finds greater values, A* will develope less states in order to find the solution as it will cut some paths earlier.

A nice way to improve the admissible heuristic is to precompute the solution of numerous configurations of a problem more simple than the original one and to use the precomputed values as admissible heuristics. For example in the 16-puzzle the Manhattan heuristic consider each tile as independent of the other. If some interactions between tiles are taken into acount, the heuristic will be improved. All states containing some predefined tiles are solved taking into account the interactions between tiles. All configurations for the first eight tiles can be calculated, removing the other tiles and replaceing them with empty tiles [Culberson and Schaeffer, 1998]. A retrograde analysis algorithm close to the one used in Chess compute the minimal number of moves required to solve each configuration of the first eight tiles. Using this pattern database is fast as it consist in finding a precomputed number in a table with the index corresponding to the configuration of the first eight tiles of the state to evaluate.

Pattern databases can be used for other problems than the 16-puzzle. It is possible for example to precompute all the combinations of the eight corner cubes of the Rubik's cube. It is then possible to use it as an admissible heuristic [Korf, 1997]. The 16-puzzle and the Rubik's cube are examples among many problems that can be speeded up with pattern databases, enabling much faster solving than with the Manhattan heuristic alone (a thousand times faster for the 16-puzzle and required to solve the Rubik's cube)

In order to solve the Rubik's cube faster it is possible to combine pattern databases, for example by computing a database for the eight corner cubes and another one for six out of the twelve border cubes. The evaluation of a position is then the maximum of the two values found by the patterns databases. Moves at Rubik's cube move both corner and boarde cubes, so it is no possible to add the two heuristics. For other problems such as the 16-puzzle it is not the case: if two databases containing disjoint sets of tiles are available, the two values can be added and still

give an admissible result since no move of the first database is also a move in the other database (tile of a database are not in the other database) [Felner et al., 2004].

For some problems such as the four-peg Towers of Hanoi it is valuable to compress pattern databases so that they fit in memory. Compression stores only one value for a set of patterns (the minimum number of moves over all the patterns of the set) [Felner et al., 2007].

Precomputing to improve the admissible heuristic and accelerate search is not limited to puzzles. For the shortest path problem on a game map, it is possible to precompute the distance from a given point to all other points. The triangular inequality can then be used to compute an admissible heuristic between any two points [Cazenave, 2006a].

It is also possible to compute pattern databases for two-player games. For the game of Go, all living shapes that fit within a given rectangle can be precomputed and used to accelerate life and death search [Cazenave, 2003].

5.3 Further topics

This section has been devoted to fully observable games; there exist extension to non-observable applications[Cazenave, 2006b; Rolet et al., 2009]. In some games, the modeling of the opponent is critical (for example Poker[Maîtrepierre et al., 2008]).

6 AI in Video Games

Besides major AI contributions to the area of classical games, new types of games have emerged over the last three decades that have called for new developments in the field of AI. Video games first distinguished themselves from classical games by relying heavily on graphics and reflex action (instead of analysis and reasoning). Yet these new video games have given more and more importance to AI, not only to provide artificial opponents to human players, but also to animate the virtual characters that inhabit the complex virtual worlds of some of the more recent games, so as to make them credible and entertaining. A strong and active community has therefore developed in this area over the last years, involving both industry and academia, with specialized books, conferences supported by major academic societies (e.g., IEEE with *Computational Intelligence in Games*, AAAI with *Artificial Intelligence and Interactive Digital Entertainment*), and journals such as *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*.

Many AI researchers have turned to the area of video games as a rich problem area [Laird, 2002]. Indeed video games constitute excellent plateforms for experimental work in AI, and that is true of a wide range of game types, including historically FPS (first-person shooters), RTS (Real-Time Strategy), RPG (role-playing

games) or even adventure games, each one bringing its own research problems [Corruble and Ramalho, 2009]. One limitation for research oriented work in this area was for a long time the limited accessibility of open plateforms, commercial games being typically closed to open investigation by outside researchers. The situation has improved recently, with open plateforms becoming more available, often as a result of partnerships between academia and industry. Alternatively, a few projects have recently tackled AI game-playing with commercial version of games, e.g. simulating mouse clicks to communicate AI moves [Madeira and Corruble, 2009], up to the point where the game state is acquired by a video camera monitoring the screen as human players would (e.g. [**?**]). In parallel, more and more game platforms are being released in the framework of competitions or challenges to the research community. Lastly, we see in this section how the video game AI domain, beyond its role of experimental platform for testing and challenging AI techniques, also contributes its own research questions, which brings about the enrichment and renewal of the field of AI as a whole.

6.1 Transitioning from Classical Games to Video Games

It is possible for some game genres to consider, from an AI perspective, video games as extensions of classical games, while other game genres introduce fundamentally new problems. In the first category, one can place modern strategy games which, similarly to classical games, stage a conflict or competition between two or more sides, each representing an army, civilization or faction, in a context that can be historical, or imaginary. Typical examples are *Age of Empires* (Microsoft), the *Total War* series (Creative Assembly), that combines strategy and tactical combat at various historical periods, Sid Meier's *Civilisation*, or Paradox Interactive grand strategy games such as *Europa Universalis*, or *Victoria*. Innovations in this group are significant and go beyond visual immersion. Besides moving units on a map in a 2D or 3D environment as one can find in classical games and many *wargames*, players are challenged to manage an econmy (ressource collection, production, budget,...), diplomacy (alliances,...) or even research and innovation policies. These multiple levels of simulation, from the most tactical (involving moving units on a map) to the more strategic (with long term policies) and their complex interactions, bring about new levels of complexity, where the middle or long term impact of decisions is extremely difficult to predict. While these strategy games can be played in a turn-based fashion or real-time, from a complexity perspective, there main innovation in comparison with classical games is their high degree of parallelism: be it for a wargame or a grand strategy game, all units can potentially receive independent orders at any moment or game turn. As a result, the combined set of possible decisions or actions at a given point in time becomes hard to enumerate and even more to evaluate, as its size grows exponentially with the number of units. Thus, the traditional AI approach to games based on tree search usually becomes non practical.

This initial remark related to the complexity of modern games might go some way toward explaining a phenomenon that could seem surprising at first: AI in video games has until recently used relatively few results or techniques coming from AI research on classical games and more generally from academic AI. Yet the video game industry has been one of the first areas to adopt the notion of AI to the point of making it a commercial argument. The game industry, and players, refer to game AI mainly as the part of the game that manages the automated behavior of the player's opponents or of the non-player characters that populate the game environment. The issue of whether this game AI actually uses techniques coming from AI research is often, maybe justifiably, seen as secondary.

As we are about to see in the next section, a large proportion of video games use techniques that can seem rather basic from an AI research perspective, but that show real strengths from the point of view of game designers and still allowing for some degree of refinement. We will see also how more recent work coming both from academia and industry have initiated a move from what is described as traditional, scripted game AI toward one that is more advanced. Furthermore, this overview of game AI must go beyond these basic questions and address other important ones. A key topic for game AI is the notion of non-player characters (NPCs), these creatures that populate the game world, especially in adventure and role-playing games, but can be relevant also in other popular genres such as simulation, sports games, or first-person shooters (FPS). NPCs extend the notion of opponent (they can indeed represent allies or be neutral to the player character), and are related to the notion of actor, that must follow the instructions of a director) or of an autonomous agent that must act, react, or interact in a credible manner with the story and the player.With this area, much of the recent research on autonomous intelligent agents finds an application and a field of experimentation that is particularly exciting [Corruble and Ramalho, 2009], but we will see in the following that one has to enrich, redefine, and sometime even strengthen some of the goals of classical research on rational agents.

6.2 AI in the Game Industry - Scripting and Evolution

Modern video games should be seen as interactive media which borrow much from a movie culture where an author imagine a story and directs actors. They add to this the key dimension of interactivity : the evolution of the story is strongly impacted by the player's actions, who might as a result guide it towards a direction or another. This complex intertwining between the levels of story and individual actions is studied in a new research domain known as interactive narrative [Perlin, 2005; Natkin, 2004]. The roots of video games in a movie culture goes some way towards explaining the reservations held by some (now rare) game designers towards the notion of an Artificial Intelligence leading to autonomous agents : in their minds, NPCs are seen mainly as actors, they must behave by following indications from the game designer, and help in steering the story towards one of the paths they antici-

pated. In that vein, some of most scripted games are referred to as *roller-coasters*, they are designed precisely to make the player live a planned sequence of emotions where intense moments are followed by relaxing episodes and so on. *Roller-coaster* games are often opposed to *sandbox* games, where the player is the one building the story that emerges from its interactions with a rich game environment. In the first category, *Roller-coaster* games have most often what is called a scripted approach to AI. Some behaviors or action sequences are triggered when specific conditions on the game state (e.g. the position of the player character, or a specific point in the story) are met. NPCs designed this way lead to behaviors expected by the designer at the time they expect it and therefore are a suitable solution to many needs of the video game industry. Nonetheless, it has its own shortcomings; Development costs can be significant as all relevant game situations must be taken into account at the time of design, which in turns implies some level of constraint on the behaviors of the player... which has generate some level of player frustration which is not conducive to immersion and enjoyment. Furthermore, it tends to lead to a rather closed or less dynamic game environment. The same situation leads to the same behaviors. It can lead to some form of player boredom, and also decrease the replay value of the game. Another consequence is that the player can exploit this tendency to repeat the same actions and *game the game*: when predicting easily the game AI reaction to its own moves, the player acquires an advantage that quickly ruins the game challenge. At a more technical level, several approaches have been used to model the behaviors of NPCs. Finite state machines (FSM) have for long been the basic tool for the scripted AI approach. The limitation of FSM in terms of representation have lead to some improvements by using hierarchical finite state machines (HFSM) [Harel, 1987] that allow the modeling of generalized states and state transitions, hence factoring some aspects of similar states. Further, *behaviour trees* [Flórez-Puga et al., 2008] aim at maximizing the level of factoring between states and make more explicit the logic of transitions between behaviors - they have thus become the leading AI technique in the video game industry over the last decade. In order to implement scripted behavior, powerful script languages have been developed over the years, several of them motivated by the game industry. One of them is LUA [Ierusalimschy et al., 2007], used in successful games such as *World of Warcraft*). They have been well received by offering a good balance between speed of execution and a development that remains outside of the game engine (the computation intensive part that deals with graphics, etc.), which lets studios work on the game AI in the last months of game development, or to refine it after game release, or in some cases to let players contribute to its improvements via modding.

6.3 Research Directions : Adaptive AI and Planning

In order to overcome the limitations of scripted approaches to game AI listed above, some attempts have been made to combine the strong points of scripted AI, well regarded in the video game industry, and adaptive abilities as studied by AI re-

searchers, especially in the area of Machine Learning. AN important example is *Dynamic Scripting* [Spronck et al., 2006]. In its initial version, this approach aims at computing, based on experience, to associate a score to each script defined by hand. This learning stage allows then afterwards the selection of the script that is the most relevant to the current situation and player. The later versions of *Dynamic Scripting* introduced a reinforcement learning stage that modifies the scripts themselves. It is therefore a rel machine learning approach to game AI, though it leaves open the possibility of using a base of pre-defined scripts as input.

Mostly in academic circles, some researchers have proposed for some years to design a game AI with a more radically different approach, placing learning techniques at its heart, with a triple objective: avoid the somewhat frozen behaviors of scripted approaches, offer the possibility of an adaptive game AI (it adapts naturally to the player and its play style as its by playing that it develops or adapts its strategy), and from a practical, development points of view, offer an alternative to the classical approach that implies the programming and debugging of numerous scripts by several programmers (some game studios hire dozens of AI script programmers in their final production phase). Resistance against this type of learning approach in the game industry has several explanations: current learning techniques on complex problems tend to converge slowly, induced behaviors are difficult to predict and it is therefore difficult to offer a guarantee a priori about the acceptability of results by the designer and ultimately by the player. On the other hand, the game industry recognizes that adaptive methods based on learning are certainly promising, as expressed plainly in the enthusiastic preface of [Rabin, 2002] qualifying learning as the *Next Big Thing* for game AI, before addressing various aspects and projects of learning game AI in ten chapters of the book.

By nature, most video games can be modeled as an agent or a system of agents interacting with their environment. They often receive an evaluation of their actions, for example, through the evolution of a game score. They therefore appear open to AI approaches based on machine learning. They are however domains where state and action spaces can be much larger than more classical games, and constitute a good motivation to improve these methods, for example, using approaches for factoring underlying Markov Decision Processes [Degris et al., 2009], or using a hierarchical decomposition for learning on adapted representations [Madeira and Corruble, 2009], which lead to viable solutions respectively for FPS games or strategy games such as historical wargames. In this last case in particular, hierarchical decomposition of the decision and learning processes, and the automated adaptation of representations by abstraction mechanisms proved necessary due to the high-level of parallelism that makes most wargames analogous to actual multi-agent simulations [Corruble and Ramalho, 2009].

Besides these approaches using learning intensively, one should not neglect approaches using some form of planning. Research presented in [Degris et al., 2009] is an interesting example of work combining reinforcement learning with a model itself learned from experience. Other approaches using sophisticated planning techniques such as Hierarchical Task Networks (HTN) become an important research direction [Hoang et al., 2005] that inspire the game AI of several commercial

games such as *Total War* (Creative Assembly), *Airborne Assault* (Panther Games), or *Armed Assault* (Bohemia Interactive),

6.4 New Research Directions for Video Game AI

We have introduced in the previous pages the state of the art and some research directions for video game AI seen as an extension of AI research on classical games. The underlying hypothesis was that the aim is to design a game AI that plays better, that is to say whose performance level approaches, reaches, or even takes over the one of a human player. This objective, so obvious that it is often not stated, is seriously challenged by the video game domain.Indeed, though until recently for some game types, either classical (chess, go,...) or "modernes" (strategy games), the main challenge for the AI researcher is to develop a challenging opponent for the human player, it is not the case anymore for many games where machines can now easily beat the human players. The challenge for AI research is then somewhat different: the goal is not to reach human level anymore, it is to propose an opponent, or a game companion, with whom the human player will enjoy the confrontation. This touches on some complex issues related to the notion of enjoyment and entertainment, at the heart of gaming, but that science and technology has ignored until recently .

A significant amount of work, at the intersection of AI, psychology and social sciences, addresses the definition and measure of satisfaction and entertainment of players. Theories, coming from the area of aesthetics, literature and cinema on one hand, and experimental work, looking at player activity and physiological parameters (heart rate, etc.) to evaluate the level of interest, are two approaches that are sometimes combined. In turn, this can guide the design of the game AI so that the behavior produced lead to the satisfaction or enjoyment of the player. One specific example of such work is the issue of game balancing or more accurately of dynamic difficulty adjustment. How to proceed so that the game AI plays at the right level whatever its human opponent and at anytime along the player's personal evolution? An important component in that area is the *flow theory* from psychology [Csikszentmihalyi, 1975, 1990] that relates well-being and satisfaction with a good balance between competence level and task difficulty. In particular, [Andrade et al., 2006] has proposed an approach to dynamic difficulty adjustment using a method derived from traditional reinforcement learning so that the selected action is not necessarily the one with the highest utility, depending on the estimated difficulty perceived by the player. All while learning to play well, agents learn also to adapt their level of play (their *performance*) by selecting sub-optimal actions, because they are seen as better adapted to the level of their opponent, the human player.

Work on difficulty adjustment are made easier by the availability of objective measures, such as the game score. Other game dimensions are however more difficult to evaluate but have a strong impact on player perceptions and his/her immersion in a story. Character believability is a good example of this, especially for adventure and role-playing games as they usually assume complex interactions in-

cluding dialogues, negotiation, etc. between NPCs and PCs. AN entire research domain has developed around this question, which attracts interest beyond the game domain, including all areas of virtual characters. In this domain, one has to go beyond the goal of having NPCs behaving rationally with high level of performance. To be credible or believable, what is important is that they have a recognizable personality influencing their actions in the long term, and that they react emotionally in a credible manner to events in their environment and to interactions with other characters. Game NPCs have therefore become an important application area for *affective computing* [Rickel et al., 2002]. [Ochs et al., 2009] for example propose a computational model allowing the simulation of the dynamics of emotions and social relations taking into account the personality of NPCs.

The few research directions outlined above give an idea of the rich scene that game AI research has become over the last decade or so. This list is far from exhaustive. By moving from the simple role of opponent to the one of NPC, game AI has extended its domain, but it is now invoked for other areas of game development. Can it contribute more centrally to game design (by crafting stories for example)? What about stage direction, camera placement and so on? And a game music that is dependant on game state and player tastes? All these questions currently constitutes new frontiers both for AI research and for the game industry.

7 Conclusion

In this chapter we have presented somme classical algorithms for programming games: alpha-beta and its improvements for zero-sum two-player games, A* for puzzles. We have also presented more recent approaches such as Monte-Carlo algorithms and applications of AI to video games. As we have seen, AI in games includes many algorithms and raises many research questions which are relevant beyond the game domain. The most active research areas of the last years include Monte-Carlo methods and video-game AI.

References

Allis, L. (1994). *Searching for solutions in games and Artificial Intelligence*. PhD thesis, Vrije Universitat Amsterdam.

Allis, L., van der Meulen, M., and van den Herik, H. (1994). Proof-number search. *Artificial Intelligence*, 66:91–124.

Anantharaman, T., Campbell, M., and Hsu, F. (1989). Singular extensions: adding selectivity to brute force searching. *Artificial Intelligence*, 43(1):99–109.

Andrade, G., Ramalho, G., Gomes, A. S., and Corruble, V. (2006). Dynamic game balancing: An evaluation of user satisfaction. In *AAAI conference on Artificial Intelligence and Interactive Digital Entertainement*, pages 3–8.

Arneson, B., Hayward, R. B., and Henderson, P. (2010). Monte Carlo Tree Search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258.

Beal, D. (1990). A generalised quiescence search algorithm. *Artificial Intelligence*, 43:85–98.

Berliner, H. (1979). The B* Tree Search Algorithm: a best-first proof procedure. *Artificial Intelligence*, 12:23–40.

Bouzy, B. (2005). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Inf. Sci.*, 175(4):247–257.

Bouzy, B. and Cazenave, T. (2001). Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103.

Bouzy, B. and Helmstetter, B. (2003). Monte-Carlo Go developments. In *ACG*, volume 263 of *IFIP*, pages 159–174. Kluwer.

Bruegmann, B. (1993). Monte Carlo go. *Unpublished*.

Bulitko, V., Lustrek, M., Schaeffer, J., Bjornsson, Y., and Sigmundarson, S. (2008). Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research*, 32(1):419–452.

Campbell, M., Hoane, A., and Hsu, F.-H. (2002). Deep Blue. *Artificial Intelligence*, 134:57–83.

Campbell, M. and Marsland, T. (1983). A comparison of minimax Tree Search algorithms. *Artificial Intelligence*, 20:347–367.

Cazenave, T. (2003). Metarules to improve tactical Go knowledge. *Inf. Sci.*, 154(3-4):173–188.

Cazenave, T. (2006a). Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *CIG*, pages 27–33.

Cazenave, T. (2006b). A Phantom-Go program. In *Advances in Computer Games 2005*, volume 4250 of *Lecture Notes in Computer Science*, pages 120–125. Springer.

Cazenave, T. (2009). Nested Monte Carlo search. In *IJCAI 2009*, pages 456–461, Pasadena, USA.

Cazenave, T. and Helmstetter, B. (2005). Combining tactical search and Monte-Carlo in the game of go. *IEEE CIG 2005*, pages 171–175.

Cazenave, T. and Jouandeau, N. (2007). On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101.

Cazenave, T., Saffidine, A., Schofield, M. J., and Thielscher, M. (2016). Nested Monte Carlo search for two-player games. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 687–693.

Cazenave, T. and Teytaud, F. (2012). Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. *Learning and Intelligent Optimization*, pages 42–54.

Chang, H. S., Fu, M. C., Hu, J., and Marcus, S. I. (2005). An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139.

Chaslot, G., Saito, J., Bouzy, B., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2006). Monte-Carlo Strategies for Computer Go. In Schobbens, P.-Y., Vanhoof, W., and Schwanen, G., editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91.

Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., and Bouzy, B. (2008). Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357.

Chou, C.-W., Teytaud, O., and Yen, S.-J. (2011). Revisiting Monte Carlo Tree Search on a normal form game: Nogo. In *European Conference on the Applications of Evolutionary Computation*, pages 73–82. Springer.

Corruble, V. and Ramalho, G. (2009). *Jeux vidéo et Systèmes Multi-Agents*, pages 235–264. IC2 Series. Hermès Lavoisier. isbn: 978-2-7462-1785-0.

Couetoux, A. (2013). *Monte carlo Tree Search for continuous and stochastic sequential decision making problems*. PhD thesis, Université Paris Sud-Paris XI.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*.

Coulom, R. (2007). Computing "Elo ratings" of move patterns in the game of go. *ICGA Journal*, 4(30):198–208.

Csikszentmihalyi, M. (1975). *Beyond boredom and anxiety*. Jossey-Bass San Francisco.

Csikszentmihalyi, M. (1990). *Flow: The psychology of optimal experience*. Harper & Row New York.

Culberson, J. C. and Schaeffer, J. (1998). Pattern Databases. *Computational Intelligence*, 4(14):318–334.

De Mesmay, F., Rimmel, A., Voronenko, Y., and Püschel, M. (2009). Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *ICML*, Montréal Canada.

Degris, T., Sigaud, O., and Wuillemin, P. (2009). Apprentissage par renforcement factorisé pour le comportement de personnages non joueurs . *Revue d'Intelligence Artificielle*, 23(2):221–251.

Donninger, C. (1993). Null move and deep search: selective search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143.

Felner, A., Korf, R. E., and Hanan, S. (2004). Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)*, 22:279–318.

Felner, A., Korf, R. E., Meshulam, R., and Holte, R. C. (2007). Compressed pattern databases. *J. Artif. Intell. Res. (JAIR)*, 30:213–247.

Finley, L. (2016). Nested Monte Carlo Tree Search as applied to samurai sudoku.

Finnsson, H. and Björnsson, Y. (2008a). Simulation-based approach to general game playing. In *AAAI*, volume 8, pages 259–264.

Finnsson, H. and Björnsson, Y. (2008b). Simulation-based approach to general game playing. In *AAAI*, pages 259–264.

Flórez-Puga, G., Gómez-Martın, M., Dıaz-Agudo, B., and González-Calero, P. (2008). Dynamic Expansion of Behaviour Trees. In *AAAI conference on Artificial Intelligence and Interactive Digital Entertainement*, pages 36–41.

Gelly, S., Hoock, J. B., Rimmel, A., Teytaud, O., and Kalemkarian, Y. (2008). The parallelization of Monte-Carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203. To appear.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of uct with patterns in Monte Carlo go. Rapport de recherche INRIA RR-6062.

Greenblatt, R., Eastlake, D., and Crocker, S. (1967). The Greenblatt chess program. In *Fall Joint Computing Conference*, volume 31, pages 801–810, New york ACM.

Harel, D. (1987). Statecharts: A visual formalism for complex systems.

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107.

Hoang, H., Lee-Urban, S., and Muñoz-Avila, H. (2005). Hierarchical plan representations for encoding strategic game ai. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*.

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA. ACM.

Junghanns, A. (1998). Are there practical alternatives to Alpha-Beta? *ICCA Journal*, 21(1):14–32.

Junghanns, A. and Schaeffer, J. (2001). Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artif. Intell.*, 129(1-2):219–251.

Kendall, G., Parkes, A., and Spoerer, K. (2008). A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34.

Knuth, D. and Moore, R. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326.

Kocsis, L. and Szepesvari, C. (2006). Bandit-based Monte-Carlo planning. In *ECML'06*, pages 282–293.

Korf, R. (1985a). Depth-first Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109.

Korf, R. and Chickering, D. (1994). Best-first search. *Artificial Intelligence*, 84:299–337.

Korf, R. E. (1985b). Depth-first iterative-deepening: an optimal admissible Tree Search. *Artificial Intelligence*, 27(1):97–109.

Korf, R. E. (1997). Finding optimal solutions to rubik's cube using pattern databases. In *AAAI-97*, pages 700–705.

Kozelek, T. (2009). Methods of mcts and the game arimaa.

Laird, J. E. (2002). Research in human-level ai using computer games. *Commun. ACM*, 45(1):32–35.

Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., and Hong, T.-P. (2009). The computational intelligence of mogo revealed in taiwan's Computer Go tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*.

Madeira, C. and Corruble, V. (2009). Strada : une approche adaptative pour les jeux de stratégie modernes. *Revue d'Intelligence Artificielle*, 23(2):293–326.

Maîtrepierre, R., Mary, J., and Munos, R. (2008). Adaptative play in texas hold'em poker. In *European Conference on Artificial Intelligence - ECAI*.

Marsland, T. (1986). A review of game-tree pruning. *ICCA Journal*, 9(1):3–19.

McAllester, D. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, 35:287–310.

Müller, M. (2002). Computer go. *Artificial Intelligence*, 134(1-2):145–179.

Natkin, S. (2004). *Jeux vidéo et médias du XXIe siècle: quels modèles pour les nouveaux loisirs numériques?* Vuibert.

Nijssen, J. and Winands, M. H. (2013). Search policies in multi-player games1. *Icga Journal*, 36(1):3–21.

Nijssen, P. and Winands, M. H. (2012). Monte carlo Tree Search for the hide-and-seek game scotland yard. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):282–294.

Ochs, M., Sabouret, N., and Corruble, V. (2009). Simulation de la dynamique des emotions et des relations sociales de personnages virtuels. *Revue d'Intelligence Artificielle*, 23(2):327–358.

Paxton, C., Raman, V., Hager, G. D., and Kobilarov, M. (2017). Combining neural networks and Tree Search for task and motion planning in challenging environments. *arXiv preprint arXiv:1703.07887*.

Pearl, J. (1980a). Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14:113–138.

Pearl, J. (1980b). SCOUT: a simple game-searching algorithm with proven optimal properties. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 143–145.

Perlin, K. (2005). Toward interactive narrative. In *International Conference on Virtual Storytelling*, pages 135–147. Springer.

Pitrat, J. (1968). Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574.

Plaat, A., Schaeffer, J., Pils, W., and de Bruin, A. (1996). Best-first fixed depth minimax algorithms. *Artificial Intelligence*, 87:255–293.

Rabin, S. (2002). *AI game programming wisdom*. Charles River Media.

Rickel, J., Marsella, S., Gratch, J., Hill, R., Traum, D., and Swartout, W. (2002). Toward a new generation of virtual humans for interactive experiences. *IEEE Intelligent Systems*, pages 32–38.

Rivest, R. (1988). Game-Tree Searching by min-max approximation. *Artificial Intelligence*, 34(1):77–96.

Rolet, P., Sebag, M., and Teytaud, O. (2009). Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*.

Rolet, P., Sebag, M., and Teytaud, O. (2009). Optimal robust expensive optimization is tractable. In *Gecco 2009*, page 8 pages, Montréal Canada. ACM.

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

Rosin, C. D. (2011). Nested rollout policy adaptation for Monte Carlo Tree Search. In *Ijcai*, pages 649–654.

Schadd, M. P., Winands, M. H., Van Den Herik, H. J., Chaslot, G. M.-B., and Uiterwijk, J. W. (2008). Single-player Monte Carlo Tree Search. In *International Conference on Computers and Games*, pages 1–12. Springer.

Schaeffer, J. (1989). The history heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transcations on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212.

Schaeffer, J. (1990). Conspiracy Numbers. *Artificial Intelligence*, 43:67–84.

Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *Science*.

Schaeffer, J. and van den Herik, J. (2002). Games, Computers, and Artificial Intelligence. *Artificial Intelligence*, 134:1–7.

Shannon, C. (1950). Programming a computer to play Chess. *Philosoph. Magazine*, 41:256–275.

Sheppard, B. (2002). World-championship-caliber scrabble scrabble® is a registered trademark. all intellectual property rights in and to the game are owned in the usa by hasbro inc., in canada by hasbro canada corporation, and throughout the rest of the world by jw spear & sons limited of maidenhead, berkshire, england, a subsidiary of mattel inc. *Artificial Intelligence*, 134(1-2):241–275.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of Go with deep neural networks and Tree Search. *Nature*, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017a). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017b). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Slate, D. and Atkin, L. (1977). Chess 4.5 - the northwestern university chess program. In Frey, P., editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. (2006). Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3):217–248.

Stockman, G. (1979). A minimax algorithm better than Alpha-Beta ? *Artificial Intelligence*, 12:179–196.

Sturtevant, N. R., Felner, A., Barrer, M., Schaeffer, J., and Burch, N. (2009). Memory-based heuristics for explicit state spaces. In *IJCAI*, pages 609–614.

Teytaud, F. and Teytaud, O. (2009). Creating an upper-confidence-tree program for havannah. In *Advances in Computer Games*, pages 65–74. Springer.

Thompson, K. (1996). 6-piece endgames. *ICCA Journal*, 19(4):215–226.

Tian, Y., Jerry Ma*, Qucheng Gong*, Sengupta, S., Chen, Z., and Zitnick, C. L. (2018). Elf opengo. `https://github.com/pytorch/ELF`.

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Zobrist, A. (1990). A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73.