Kévin Gomez Pinto & Jonathan Da Silva Computers Science Department

Artificial Intelligence Report

Graphs and Game 2011

Abstract

In this report, we will see how to implement an artificial intelligence for the Jungle game, using advanced algorithms like Minimax or Alpha-Beta. We will also try to merge the computational power of these algorithms with the “human knowledge” we have of this game.

Résumé

Dans ce rapport, nous verrons comment implémenter une intelligence artificielle pour le jeu Jungle, en utilisant des algorithmes avancés tels que Minimax ou encore Alpha-Beta. Nous essaierons aussi de combiner la puissance de calcul de ces algorithmes avec les « connaissances humaines » que nous avons de ce jeu.

Summary

Introduction

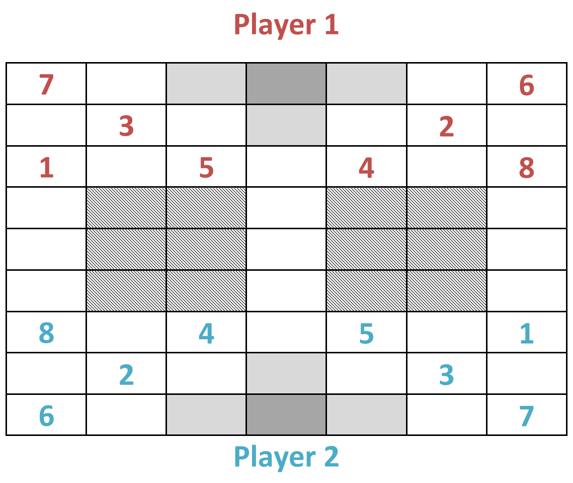
Introduction to the Jungle game

The Jungle – which is also called Dou Shou Qi – is a traditional strategy game. It is played with eight pieces for each of the two players. There are two main winning cases: the first is by moving a piece onto a special square (the den) on the opponent's side, the second is by eating all the opponent's pieces.

Each piece represents an animal and has a number which symbolizes his strength. Here is the piece's list, ordered by strength:

1. Rat
2. Cat
3. Dog
4. Wolf
5. Leopard
6. Tiger
7. Lion
8. Elephant

Obviously, a piece cannot be eaten by another piece with a strictly smaller strength. However, there is an exception with the rat which can eat the elephant (but the elephant cannot eat the rat). At the beginning of a game, the board looks like this:



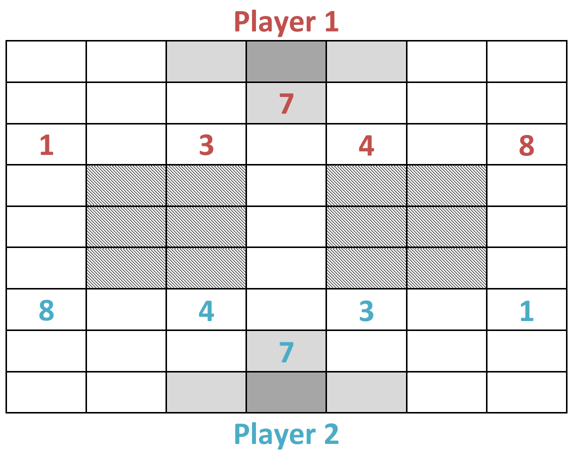
The squares in dark grey are the two dens, the squares in light grey are traps and the hatched ones are lakes.

During his turn, the player **must** move. He can move one square horizontally or vertically (not diagonally) except if the destination square is part of a lake. Once again, there are some pieces which are special regarding this rule. Indeed, the rat can go into the water and both the lion and the tiger can jump over it. It is important to notice that the rat cannot eat the elephant directly from a water square and that the lion and the tiger cannot jump over the lake if there already is a rat onto it.

The contest

A contest is organized between the best artificial intelligences (AIs, for short) made by the students. It is the pretext for us to play with the Jungle API and to implement simple – and not so simple – artificial intelligences.

Because of the lack of time for the tournament and because of the relatively humble number of computers at our disposal, the rules have changed a little. The board's size is the same, but each player has now only five pieces which are placed as displayed below:



Moreover, each AI has a computation time equals to ten minutes per game. After this period, it is a "t*ime control"* win for the opponent.

Two naïve AIs

A first try: the naïve AI

In order to discover the Jungle's API and as an introduction to the world of artificial intelligences implementation, we started by developing a very naive AI which had to follow this strategy:

"I look at all the moves I can play and play a winning move if I can, else I discard all moves that make me lose, and I play at random from the remaining moves unless all moves make me lose and I simply play at random." [1]

We will see later in this report that it is certainly not the best strategy we can find, but at least it allowed us to familiarize ourselves with the game and the way it handle things like the board and the pieces.

We also implemented another AI – based on the previous – which tries to eat an opponent's piece when it is possible. Thereafter, we will call it the "*naive eater*" AI.

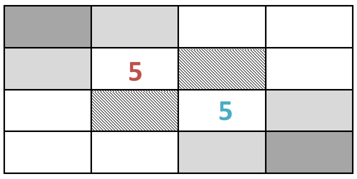
Experimental results analysis

For each couple of AIs, we ran **25 games** in **"real" conditions** (except for the board's size and the number of pieces).

Throughout the experiments, we also used two AIs which were given to us:

- "*StraightForward*": it chooses the nearest piece from the opponent's den, and it makes it charge straight to the den ;
- "*Random*": it plays random moves.

4x4 board with 1 piece per player



4x4 board 1

[1] From the first week's practical subject

Player 2 / Player 1	Random	StraightForward	Naive	NaiveEater
Random	12 / 13	0 / 25	2 / 23	1 / 24
StraightForward	10 / 15	0 / 25	0 / 25	0 / 25
Naive	18 / 7	11/ 14	3 / 22	2 / 23
NaiveEater	19 / 6	12 / 13	4 / 21	3 / 22

Experimental results 1

If we look at the random AI fighting against itself, there is pretty much nothing to say: the chance does the work for us and the winning rate reflects that.

But it seems to be different when we look at the three others AIs. We clearly see that the player two statistically wins more game than the player one. We could distinguish three main cases:

1. Two naïve AI play against each other: most of the time, the game will be won by the second player.
2. A *naïve* AI plays against the *StraightForward* AI: if the *StraightForward* AI is the first player, it will lose. Otherwise the outcome of the game is not clear.
3. A *Random* AI plays against another AI: the R*andom* AI is likely to lose (except if the second player is the *StraightForward* AI, but we did not manage to find out).

It appeared that on a very small board, it is easy to obtain good results, even with a really naïve and opportunistic AI. Indeed, by playing random moves and choosing a winning one when it is possible, our two naïve AIs obtained quite good results as they were often able to eat the opponent's piece or to reach his den.

6x5 board with 3 pieces per player

For these simulations, we chose a bigger board and we added two pieces.

Player 2 / Player 1	Random	StraightForward	Naïve	NaiveEater
Random	9 / 16	0 / 25	1 / 24	2 / 23
StraightForward	25 / 0	25 / 0	24 / 1	20 / 5
Naive	20 / 5	0 / 25	12 / 13	6 / 19
NaiveEater	25 / 0	5 / 20	18 / 7	14 / 11

Experimental results 2

This time, we see that playing random moves is definitely not a suitable strategy: the *Random* AI and the *Naïve* AI are really having a bad time. Despite this, the improved *Naïve* AI (a.k.a. *NaiveEater*) is able to win a few games against the *StraightForward* AI, thanks to its ability to eat an opponent's piece when it is possible.

9x7 board with 5 pieces per player

Player 2 / Player 1	Random	StraightForward	Naive	NaiveEater
Random	17 / 8	0 / 25	1 / 12 (Time Loss)	1 / 21 (Time Loss)
StraightForward	25 / 0	25 / 0	25 / 0	14 / 11
Naive	11 (Time Loss) / 3	0 / 25	4 (Time Loss) / 6	3 / 10 (Time loss)
NaiveEater	23 (Time Loss) / 0	10 / 15	5 (Time Loss) / 3	3 (Time Loss) / 8

Experimental results 3

With the board that will be used for the contest, it appears that the *Naïve Eater* AI is able to challenge the *StraightForward* AI. Ten victories out of eleven were obtained by eating all the opponent's pieces.

However, if we look at the games between two naïve AIs, or a naïve AI and the *Random* AI, we can say that it is not possible for them to play on this kind of board. Playing random moves on a board as big as this one make the games last forever.

The Minimax algorithm

Building a perfect AI with Minimax

The next step is the implementation of a perfect player. To achieve this, we used the Minimax algorithm.

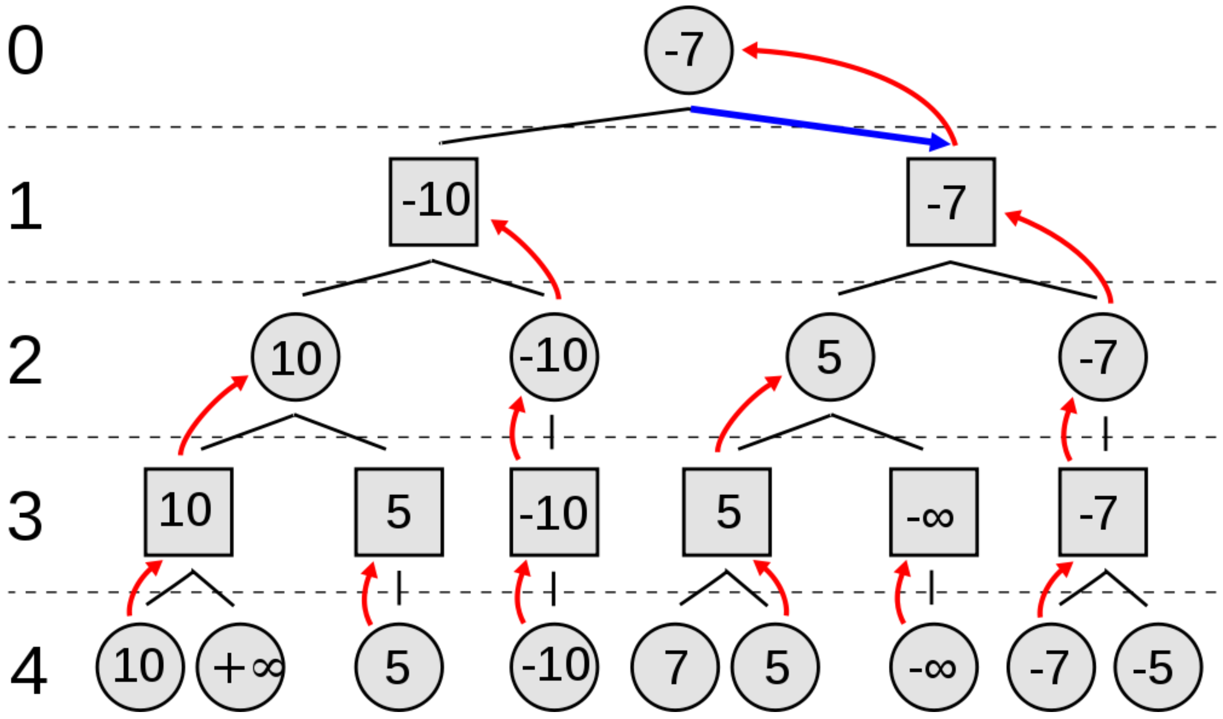
Basically, the idea consists in – given a starting game state – considering all the possible outcomes until the game ends. It means that I have to look at all my possible moves and all the answers from my opponent and so on until I reach an endgame. I also have to deduce the payoff of the current move depending on whose turn it is to play. I want to maximize my outcome whereas my opponent will try to minimize it. Results will be propagated upward so that I will be able to find the optimal move to play.

As we just saw, each move considered creates a new game state which is represented as a node in a search tree. For each game state (or node) we then consider each one of the opponent's possible moves. We iterate this process until an endgame is reached.

By exploring the whole game tree, the algorithm can tell if one of the players has a winning strategy. If it is the case, no matter what his opponent will do, he will not lose. In our case, there must be a winning strategy for one of the players because we are in a zero-sum game with no possible draw.

Note that at Jungle, the tree has a finite height as every game is bounded in length by a special rule which prevents a configuration to occur more than a specific number of times. Typically, this value is set to one, and if a player moves a piece to reach a configuration already witnessed during the game, this player will lose (this type of defeat is called "TooManyOccurences" in the implementation of the Jungle we use).

A game tree example



Game tree example 1

Experimental results analysis

In addition to the *BruteForce AI* we implemented a threaded version of the Minimax algorithm. When it is its turn to play, he explores the different possible moves in separated threads. It allows us to divide the exploration time and to play on bigger boards than the non-threaded version. To summarize, if for its first move there are N possible moves, there will be N exploration threads.

4x4 board with 1 piece per player

Player 2 / Player 1	Random	StraightForward	Naive	NaiveEater	Bruteforce	ThreadedBruteforce
Random	9 / 16	1 / 24	0 / 25	0 / 25	0 / 25	0 / 25
StraightForward	15 / 10	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25
Naive	17 / 8	13 / 12	5 / 20	1 / 24	0 / 25	0 / 25
NaiveEater	16 / 9	12 / 13	8 / 17	6 / 19	0 / 25	0 / 25
BruteForce	10 / 15	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25
ThreadedBruteforce	9 / 16	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25

Experimental results 4

This time, we can see that the perfect player really plays well. In fact it won all its games except against the *Random AI*. Nevertheless, it is important to notice that on this board and when both of the players play perfects moves, the second player always wins.

We can deduce that there is a winning strategy for the second player. It means that - on this board and – if he only plays "*perfect moves*", he will win no matter what his opponent will do. Against another *BruteForce AI* or against the *StraightForward AI*, the second player always wins by *TooManyOccurrences*, whereas against a *Naïve* one he wins by eating all its opponent pieces.

We also collected some statistics about the number of visited nodes and the average degree of a tree node:

- A *Naïve AI* visits on average 150 nodes per game;
- A *Bruteforce AI* visits on average 1,200 nodes per game against the *StraightForward*, 600 against another *BruteForce* and 11,000 against a *Random* or a *Naïve AI*.
  The average exploration depth is equals to 8 against the *StraightForward* (15 is the maximum), 7 against another *BruteForce* (13 is the maximum) and 18 against a *Random* or a *Naïve AI* (35 is the maximum).

The average degree is approximately equals to 2.

6x5 board with 3 pieces per player

We ran simulations on this board, but it appeared that both the *BruteForce AI* and the *ThreadedBruteForce AI* were not able to explore the whole game tree and loss by time control.

Time improvement: Minimax cut-off

In order to deal with regular board with our *BruteForce AI*, we added a cut-off in the game tree exploration algorithm. It means that "*we follow the optimal strategy as long as the position is not too deep otherwise, if we reach a position that is not an endgame after c moves, we use a heuristic function to evaluate it, and continue as in the optimal strategy.*"[2]

The role of the heuristic function is to give an "artificial" payoff to a given state of the game. It can do that by counting the pieces which are still alive, by looking at the distance between us and the den, etc. The goal is really to give a score to a game state considering our pieces and their positions, but also our opponent's pieces.

N.B: BruteForce AI and ThreadedBruteForce AI always use the same heuristic function and stop the exploration at the same depth.

A first heuristic function

The first heuristic function that we made was really simple: it returned the total of my remaining pieces minus the total of my opponent's pieces. It was like a "domination score" and it helped the *BruteForce AI* to mimic the behaviour of the *Naïve AI* when the exploration was interrupted.

Experimental results analysis

With this improvement, the two BruteForce were able to run on a 6x5 board.

[2] From the third week's practical subject

6x5 board with 3 pieces per player (max. depth = 2)

Player 2 / Player 1	Random	StraightForward	Naïve	NaiveEater	Bruteforce	ThreadedBruteforce
Random	15 / 10	0 / 25	0 / 25	0 / 25	**0 / 25**	**0 / 25**
StraightForward	25 / 0	25 / 0	24 / 1	22 / 3	**25 / 0**	**0 / 25**
Naïve	18 / 7	0 / 25	14 / 11	5 / 20	**24 / 1**	**1 / 24**
NaiveEater	22 / 3	5 / 20	18 / 7	10 / 15	**23 / 2**	**22 / 3**
BruteForce	**25 / 0**	**0 / 25**	**23/ 2**	**24 / 1**	**25 / 0**	**25 / 0**
ThreadedBruteForce	**25 / 0**	**25 / 0**	**24 / 1**	**24 / 1**	**25 / 0**	**25 / 0**

Experimental results 5

The results are comparable to the results of the 4x4 board without cut-off except for two types of matches:

- Between a *BruteForce AI* and a *Random AI*: there was no clear winner whereas now, the BruteForce always wins;
- Between the *BruteForce AI* and a *StraightForward AI*: the *BruteForce* should have the upper hand (and it is confirmed with the *ThreadedBruteforce*) but the StraightForward wins.
  As the BruteForce and the ThreadedBruteforce should have exactly the same behaviour, we can deduce that there is a bug in the implementation of one of these two AIs.

This time again, we collected some statistics about the number of visited nodes and the average degree of a tree node:

- A *Naïve AI* visits on average 2,750 nodes per game and the average degree is approximately equals to 7;
- A *Bruteforce AI* visits on average 5,200 nodes per game against the *StraightForward*, 5,083 against another *BruteForce* and 30,000 against a *Random* or a *Naïve AI*.
  The average degree is equals to 7 against the *StraightForward* (10 is the max. and 6 the min.), 9 against another *BruteForce* (10 is the max. and 9 the min.) and 8 against a *Random* or a *Naïve AI* (12 is the max. and 2 the min.).

6x5 board with 3 pieces per player (max. depth = 3)

We obtained similar results than with a maximum depth of two in terms of victories except for the *BruteForce AI* which lost by time control a few games against the *Naïve* and the *NaiveEater AI*. It shows that the Minimax algorithm isn't efficient enough to handle this board with a decent exploration depth. We succeeded in getting around this by using threads, but we couldn't increase the exploration depth or play on the 9x5 board.

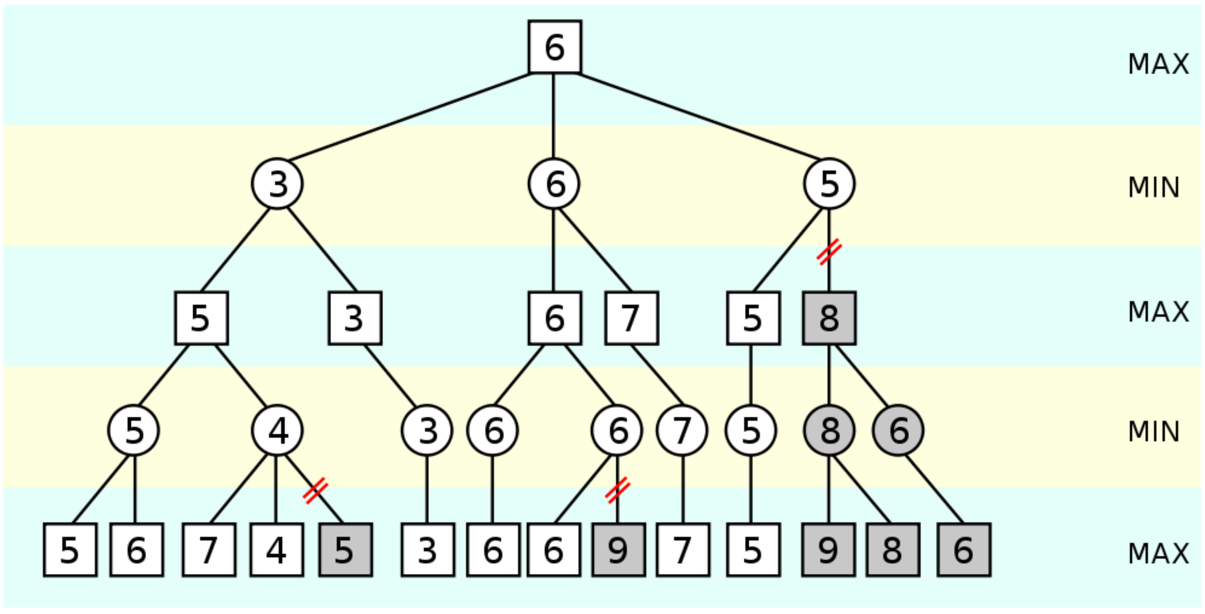
Our statistics reported that the average visited nodes rose 40,000 against a *Naïve AI* and 78,600 against the *Straightforward AI*.

Alpha-Beta pruning

The algorithm

The Alpha-Beta algorithm is an improvement of the Minimax algorithm. Thanks to this algorithm, we no longer need to explore the whole game tree as it can prune and eliminate the branches that will not be interesting. It is important to precise that this algorithm will return **exactly** the same results than Minimax but it will visit fewer nodes. That means that with the same search depth, it will be faster than Minimax. Therefore we should be able to increase the search depth. In fact, if the nodes are visited in an optimal order, the optimization reduces the effective depth to slightly more than half.

A game tree example



Game tree example 2

Experimental results analysis

Instead of the perfect moves ordering, we implemented a random move ordering. As expected, we obtained the same results as before, but the total of visited nodes was divided by approximately 1.6.

Improving the AI

Building a better heuristic

Our current heuristic function is very simple: it counts our pieces and our opponent's ones and it makes the difference.

A simple heuristic

We implemented a new version that was able to consider the strength of each piece. If I have two pieces left, let say the elephant and the mouse, my "pieces score" will be $1 * 8 + 1 * 1 = 9$. In addition to the "pieces score", this heuristic also considered the distance between my pieces and my opponent's den. The "distance score" was computed using the minimal distance and a mathematical function to give a score according to the distance. The more the piece is close to the den, the higher the score.

Because of the lakes and the special abilities of some pieces, the distances can be tricky to compute. In order not to spend too much time in useless computations, we hardcoded two tables of distances: one for the tiger and the lion, and the other for the pieces without special abilities. For the rat, we used the Manhattan distance.

Using the two tables below, we can access in constant time the distance separating a piece from the den.

Distances table for "jumping pieces"

Player 2 (Yellow)

Row \ Column	0	1	2	3	4	5	6
0	11	10	9		9	10	11
1	10	9	8	7	8	9	10
2	9	5	4	6	4	5	9
3	6	▨	▨	5	▨	▨	6
4	5	▨	▨	4	▨	▨	5
5	4	▨	▨	3	▨	▨	4
6	5	4	3	2	3	4	5
7	4	3	2	1	2	3	4
8	3	2	1		1	2	3

Player 1 (Red)

Distances table for "standard pieces"

Player 2 (Yellow)

Row \ Column	0	1	2	3	4	5	6
0	11	10	9		9	10	11
1	10	9	8	7	8	9	10
2	9	8	7	6	7	8	9
3	8	▨	▨	5	▨	▨	8
4	7	▨	▨	4	▨	▨	7
5	6	▨	▨	3	▨	▨	6
6	5	4	3	2	3	4	5
7	4	3	2	1	2	3	4
8	3	2	1		1	2	3

Player 1 (Red)

An improved heuristic

We tried to improve the precedent heuristic by tweaking a little the way we computed the "pieces scores". We noticed that the elephant is a very strong piece which cannot be eaten by a piece but the mouse. Therefore, we decided to increase its score according to the following rules.

- Bonus for the elephant:
  - If my elephant is not alive: return 0
  - If my opponent's mouse is not alive: return 100
  - Else: return 50
- Bonus for the mouse:
  - If my mouse is not alive or if my opponent's elephant is not alive: return 0
  - Else: return 50

We also added a "danger score" which decreases our global score if our pieces are next to an opponent's pieces. It means that if my tiger (7) is next to an elephant (8), the global score will be decreased by 8. But if my tiger is next to a mouse (1), the global score will remain unchanged.

Each time, the new heuristic beats the precedent one.

Look ahead

The idea is that it is cheap to look at the few positions obtained by the next plies and look up which one is an endgame. For example, if a ply leads to a win or a loss, we can prune directly. This technic allows us to save time by avoiding the exploration of nine unsuccessful branches if the 10th is an endgame.

Delay tactic

This strategy consists in distinguishing win (or loss) that happen in a near future – less than two moves - in opposition to a win that happens in three or more moves. It allows us to win as soon as possible while trying to delay our loss if we are in a losing position.

Hardcoded opening moves

As the firsts moves are very difficult to play – due to the large number of possibilities – we tried to find the three or four firsts moves that will be the most efficient and make our AI play them no matter what the opponent plays.

After a few tests, it appears that all the opening tables we tried were easily defeated by another AI, without opening table.

Iterative deepening

The efficiency of Alpha-Beta pruning relies on the order in which the positions are explored. The idea behind iterative deepening is to first explore depth one, then order the moves according to their grades. Then explore depth two, in the order found during the previous exploration, and so on.

Iterative deepening repeats some of its work since for each exploration it has to start back at depth 1. But the gains that it provides by correctly ordering the nodes outweigh the cost of the repetition. It is also useful since it allows us to refine the solution gradually. That means that when we are exploring depth *d*, we already have a good solution for depth *d-1*. What we could do with our AI, is keep track of the time remaining to return a solution. When we are almost out of time, we stop iterative deepening and return the last best solution.

But due to the lack of time, we didn't implement this idea.

Team work and contest

As a “team leader”, I had to introduce the different AIs that were already implemented so that we could work and improve them. I also had to allocate the different tasks to make between our team’s members and coordinate their work. Some of us tried to improve our heuristic functions while some others were running tests to determine which of our AIs was the best, while the others were adapting the code to make it run on the contest’s computers.

The AI we sent to the contest fought pretty well as it never lost a match except by time control. We can guess that the Alpha-Beta algorithm was implemented successfully (or at least without giant bug) and that the heuristic function is quite good. It would have been interesting to implement the “iterative deepening” variant of the Alpha-Beta. Moreover, it will be necessary, in order to improve the AI, to profile our code and detect which parts of it make it slow. As the heuristic function is called a lot of times before we can play a move, we suspect it to be the root of our troubles.

In the end, we finished in the third place (*ex-aeqo*) of the contest.

Conclusion

Programming artificial intelligences can be complex but definitely is very instructive. Implementing both advanced algorithms and putting them together with human knowledge – heuristic function – is something very interesting. We saw in the report how to improve existing algorithms and we learnt how to adapt them to the game we play.

Appendix

Webography

- http://chessprogramming.wikispaces.com/ - repository of information about programming computers to play chess.
- http://ai-contest.com/ – Google AI Challenge web site.
- http://jeuxstrategie.free.fr/Xoudouqi_complet.php – Opening moves examples.
- http://www.sifflez.org/misc/tronbot/index.html – Article written by a participant of the Google AI Challenge.
- http://en.wikipedia.org/wiki/Minimax
- http://en.wikipedia.org/wiki/Alpha-beta_pruning – Explanations of the Alpha-Beta algorithm
- http://chess.verhelst.org/1997/03/10/search/ – By M. Verhelst in 2004 – Blog article about AI technics applied to chess.