LINGI2261: Artificial Intelligence
Assignment 3: Project: Adversarial Search

Jean-Baptiste Mairy, Cyrille Dejemeppe, Yves Deville
October 2012

Guidelines

- This assignment is due on **Wednesday 14 November 2012 at 18h00**.
- This project accounts for 40% of the final grade for the practicals.
- *No delay* will be tolerated.
- Not making a *running implementation* in *Python 3* able to solve (some instances of) the problem is equivalent to fail. Writing some lines of code is easy but writing a correct program is much more difficult. (Remember B. Lecharlier's courses.)
- *Document* your source code (at least the difficult or more technical parts of your programs). Python docstrings for important classes, methods and functions are also welcome.
- Indicate clearly in your report if you have *bugs* or problems in your program. Do not let the instructor discover it.
- Copying code from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated. The consequences of *plagiarism* is *0/20 for all assignments*.
- Answers to all the questions must be delivered at the INGI *secretary* (paper version). Put your names and your group number on it. Remember, the more concise the answers, the better.
- Source code shall be submitted via the *SVN* tool. Only programs submitted via this procedure will be corrected. No report or program sent by email will be corrected.
- Respect carefully the *specifications* given for your program (arguments, input/output format, etc.) as we will use fully-automated tests. When we provide a way to test your program, it *must* pass this test.

Deliverables

- The answers to all the questions in paper format **at the secretary** (do not forget to put your group number on the front page).
- The following files inside the `milestone3` directory of your SVN repository:
  - `basic_player.py`: the basic Alpha-Beta agent from section 3.1,
  - `super_player.py`: your super-tough Sarena AI agent for the contest.
- A **mid-project meeting** will be organized approximately two weeks before the deadline. Each group will get to see one of the teaching assistants to discuss the progress so far. You have to register for a slot on the iCampus site by editing the wiki page. The meeting is not evaluated, but still mandatory (read: we will remove points if you do not come). You should at least have done sections 1, 2, 3.1 and 3.2. Come with the results.

1 Scipion (2 pt)

Let us play Scipion. The game is played on a 3×3 grid, one player playing with 3 **X** chips and the second one with 3 **O** chips. Each player can only move his own chips. The initial state of the game is displayed in Figure 1.

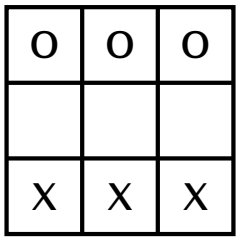
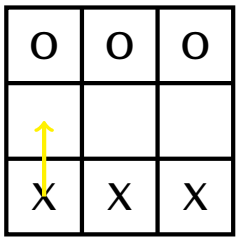


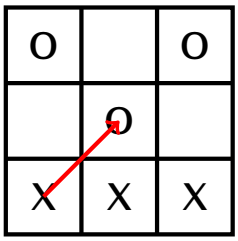
Figure 1: Initial state of the Scipion game.

At each turn, a player has to move one of his own chips. The **X** moves first, then players alternate moves. There are only two moves allowed in Scipion:

1. Go forward (up for **X**, down for **O**) if the destination square is free (as shown in Figure 2a).
2. Capture an oponent chip in a forward diagonal (as shown in Figure 2b).



(a) Moving forward



(b) Capturing an opponent

Figure 2: The two moves allowed in a Scipion game.

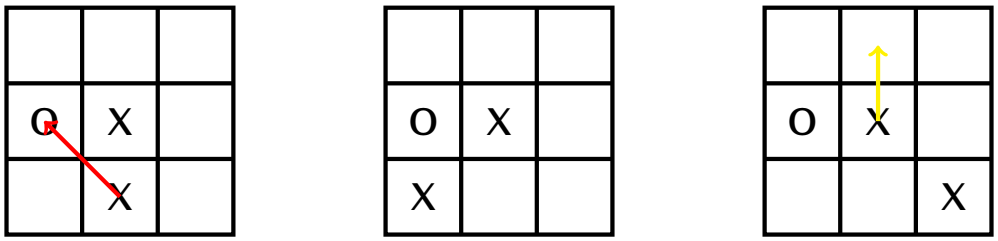
There are three ways to win a game:

1. Capture all the chips of your opponent (as shown in Figure 3a).
2. Block your opponent in such a way he is unable to perform a move (as shown in Figure 3b).
3. Having one of your chip on the last row at the opposite of your starting row (as shown in Figure 3c).

We will use the MiniMax algorithm using the heuristic function

$$2X_2 + X_1 - (2O_2 + O_1)$$

where X_2 = number of unblocked **X** chips in the second row,
X_1 = number of unblocked **X** chips in the starting row,
O_2 = number of unblocked **O** chips in the second row,
O_1 = number of unblocked **O** chips in the starting row



(a) Capturing all the chips of your opponent

(b) Blocking your opponent

(c) Reach the last row of the board

Figure 3: The three ways to win a game.

The initial state is the board configuration shown in Figure 1, with **X** to play. A chip is unblocked if it can be moved.

Questions

1. Draw the game tree for a depth of 2, i.e. one turn for each player.
2. Evaluate the value of the leaves using the heuristic function (on the figure you draw in 1, you don't have to implement it!).
3. Using the MiniMax algorithm, find the value of the other nodes.
4. Circle the move the root player should do.

In the game tree, if a node has expanded symmetrical states, you should only consider one of them. For instance, the first level has only two nodes.

2 Alpha-Beta search (4 pt)

Questions

1. Perform the MiniMax algorithm on the tree in Figure 4, i.e. put a value to each node. Circle the move the root player should do.
2. Perform the Alpha-Beta algorithm on the tree in Figure 5. At each non terminal node, put the successive values of α and β. Cross out the arcs reaching non visited nodes. Assume a left-to-right node expansion.
3. Do the same, assuming a right-to-left node expansion instead (Figure 6).
4. Can the nodes be ordered in such a way that Alpha-Beta pruning can cut off more branches (in a left-to-right node expansion)? If no, explain why; if yes, give the new ordering and the resulting new pruning.

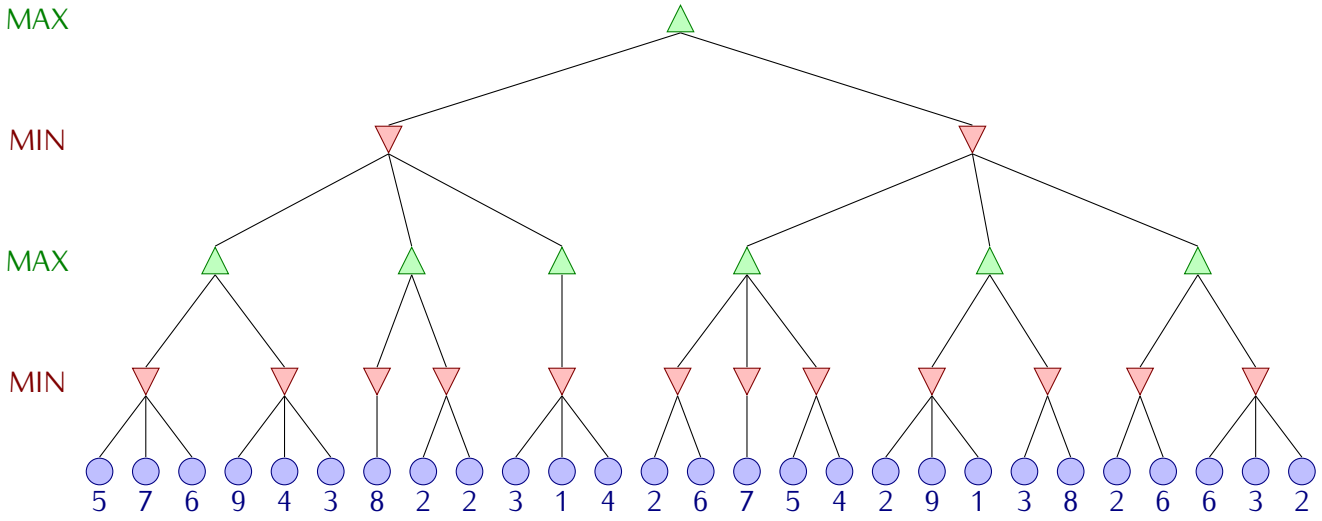


Figure 4: MiniMax

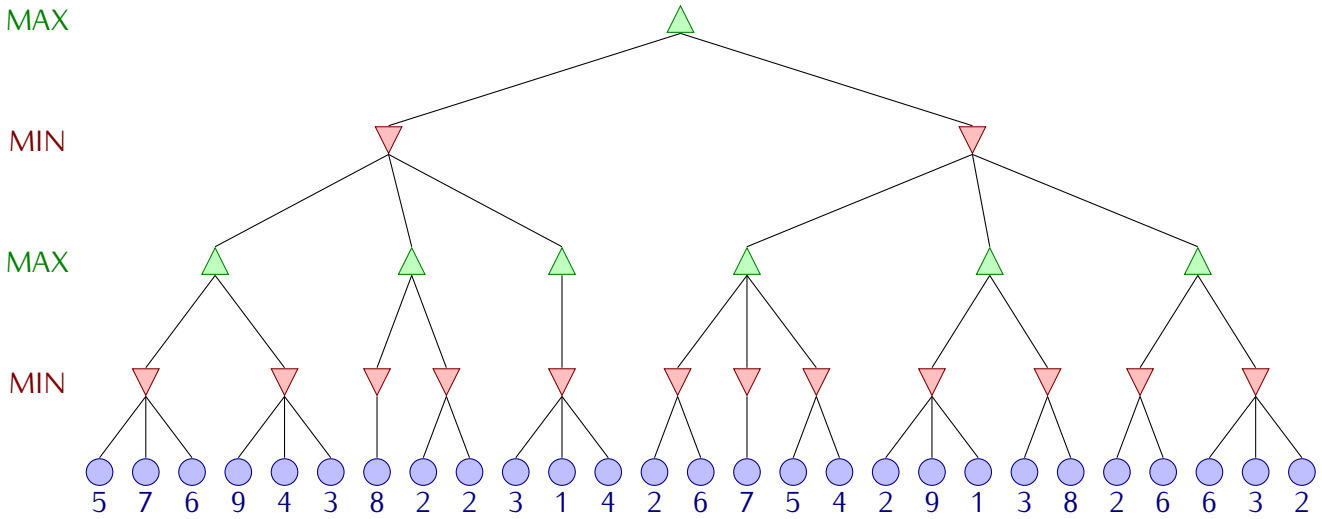


Figure 5: Alpha-Beta, left-to-right expansion

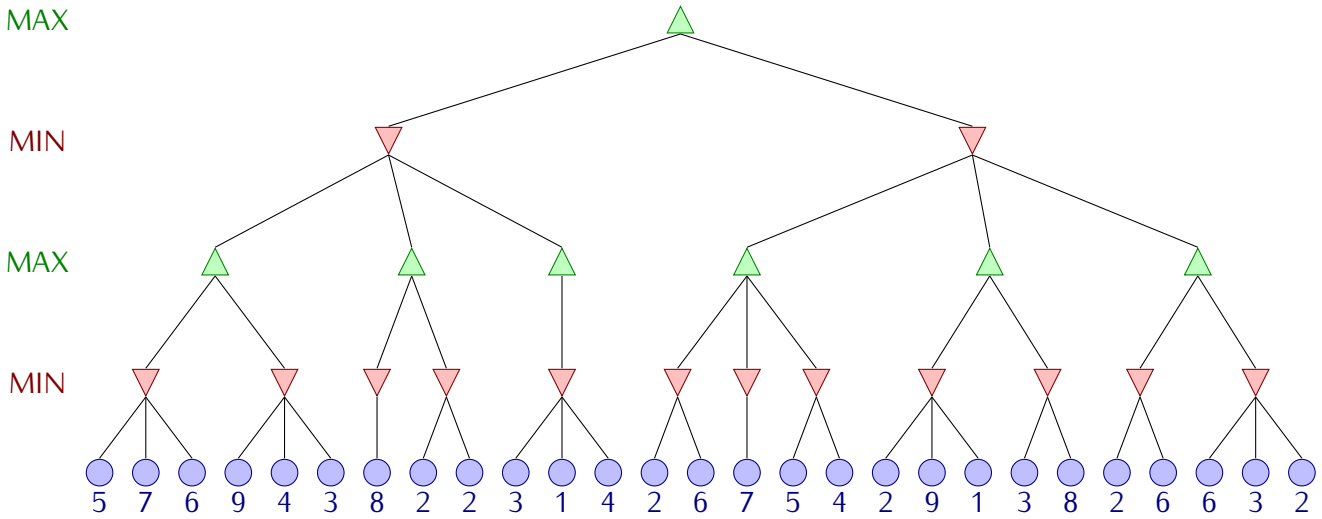


Figure 6: Alpha-Beta, right-to-left expansion

3 Sarena (34 pts)

You have to imagine and implement an AI player to play the Sarena game. You should have received a copy of the rules at the same time you received the assignment 2 statement.

3.1 A basic Alpha-Beta agent (3 pts)

Before you begin coding the most intelligent player ever, let us start with a very basic player using pure Alpha-Beta. Copy the file `player.py` to `basic_player.py` and fill in the gaps by answering to the following questions.

Note: you are required to follow exactly these steps. Do not imagine another evaluation function for example. This is for later sections.

Questions

1. Implement the `successors` method *without modifying* `minimax.py`. You shall return all the successors given by `Board.get_actions` in that order.
2. We want to explore the whole tree. The cut-off function thus only has to check if the given state is a terminal state. This can be done using the `Board.is_finished` method. Implement this in the `cutoff` method.
3. As we explore the whole tree, the evaluation function is in fact the utility function. This function shall return 1 if the agent is winning, -1 if he is loosing and 0 if it is a draw game. You can use `Board.get_score` to determine the winner, but beware that the agent is not always playing yellow.

3.2 Comparison of MiniMax and Alpha-Beta (3 pts)

Let us now compare the performances of the MiniMax and Alpha-Beta algorithms. To do so, we will use the tiny board shown in Figure 7.

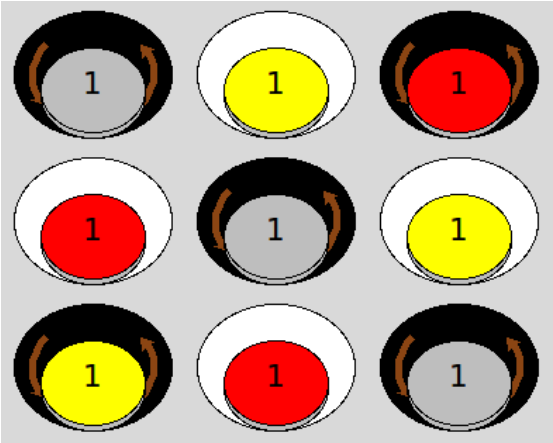


Figure 7: The mini board used in section 3.2.

The board can be loaded with the following Python code:

Loading the tiny board

```
from sarena import *
board = Board(load_percepts("mini_board.dmp"))
```

It can be loaded alternatively by using `game.py` with the option `-board==mini_board.dmp`.

We assume that the yellow player, i.e., player 1, has to make a move. Use the successor and evaluation functions of the basic agent from section 3.1.

Questions

4. What action will be played when using the MiniMax algorithm and when using the Alpha-Beta algorithm? Is there a difference between both results?
5. For both algorithms, give the time taken and the number of nodes that are visited when searching the tree. Does this meet your expectations?
6. For both algorithms, report the number of nodes that are visited at each depth level of the tree.[a] What do you observe? Explain.

[a] You can do that inside the cut-off function which is called at each visited node.

3.3 Evaluation function (6 pts)

When the size of the search tree is huge, a good evaluation function is a key element for a successful agent. This function should at least influence your agent to win the game. A very simple example is given by the `get_score()` method of the `Board` class. It returns the difference between the number of chips in towers belonging to the agent and those belonging to the adversary (in case of tie, it returns the difference between the number of chips of the agent color in towers belonging to the agent and the number of chips of the adversary color in towers belonging to the adversary). But we ask you to make a better one.

Questions

7. Describe your evaluation function.
8. In Alpha-Beta, the evaluation function is used to evaluate leaf nodes (when the cut-off occurs). As seen in previous questions, the pruning of Alpha-Beta depends on the order of the successors. Explain how your evaluation function could be used to (we hope) obtain more pruning with Alpha-Beta. Are there any drawbacks to your approach?
9. Make an agent using the successor function of the basic player (section 3.1), using your new evaluation function and cutting the tree at its root to use the evaluation function on its direct successors (you can achieve this by making `cutoff` always return `True`). Let this agent play against another similar agent using `Board.get_score` as evaluation function. Try out multiple matches and vary who plays first. How well does your evaluation function fare?

3.4 Successors function (6 pts)

The successors of a given board can be obtained by applying all the moves returned by the `get_actions(player)` method. However, there might be too many of them, making your agent awfully slow. Another crucial point is thus designing a good way to filter out irrelevant successors.

Questions

10. Give an upper bound on the number of successors for one state.
11. From random games (at least 100), compute the average number of possible actions at each step of the game. Plot the results in a graph. What do you observe?
12. Are all these successors necessary to be exhaustive (think about symmetry)? Why? If not, how will you consider only the necessary states?
13. If the number of successors is still too large, can you think of states that might be ignored, at the expense of loosing completeness?
14. Describe your successors function.
15. On average, how deep can you explore the tree made by your successors function starting from random initial states (as generated by the constructor of the `Board` class) in less than 30 seconds? Use an Alpha-Beta agent with the basic evaluation function and increase progressively the cut-off depth.

3.5 Cut-off function (4 pts)

Exploring the whole tree of possibilities returns the optimal solution, but takes a lot of time (the sun will probably die before). We need to stop at some point of the tree and use an evaluation function to evaluate that state. The decisions to cut the tree is taken by the cut-off function.

Questions

16. The `cutoff` method receives an argument called `depth`. Explain precisely what is called the *depth* in the `minimax.py` implementation. Illustrate on an example (draw a search tree and indicate the depths).
17. Explain why it might be useful (for the Sarena contest) to cut off the search for another reason than the depth.
18. Describe your cut-off function.

3.6 Contest (12 pts)

Now that you have put all the blocks together, it is time to assess the intelligence of your player. Your agent will fight the agents of the other groups in a pool-like competition. You are free to tune and optimize your agent as much as you want, or even rewrite it entirely. For this part, you are not restricted to Alpha-Beta if you wish to try out something crazy. We do emphasize on two important things:

1. *Technical requirements* must be carefully respected. The competition will be launched at night in a completely automatic fashion. If your agent does not behave as required, it will be eliminated.
2. Your agent should not only be smart, but also *fair-play*. Launching processes to reduce the CPU time available to other agents, using CPU on other machines are prohibited. Any agent that does not behave fairly will be eliminated. If you are in doubt with how fair is a geeky idea, ask us by mail. Your agent must only be working during the calls to `play`.

You should also write down the key ideas and techniques of your agent in your report. As always, try to be concise and go straight to the point.

Technical requirements

- All your files should be put directly inside the `milestone3` subdirectory of your group's directory on the SVN repository (no further subdirectories).
- All the files required for your agent to work shall be in this directory, including provided files (e.g., `sarena.py`) if needed.
- Your agent should be implemented in a file named `agent.py`. It will be started with the following command:

  ```
  python3 player.py -p $port
  ```

Your program should start an XMLRPC server listening to at least `localhost` on port `$port`. The easiest way to comply is to use the provided `player_main` function.

- Your agent must use the CPU only during the calls to the `play` method. It is forbidden to compute whatever stuff when it should be the other agent to play.
- You can assume your agent will be run on a single-processor single-core machine. Do not waste time parallelizing your algorithms.
- Your agent cannot use a too large amount of memory. We do not quantify the *too large*, but you need to leave some space for the other processes. We do not want to look for a super computer with GB's of memory just to make your agent running.
- Your agent will receive a time credit for the whole game. The time taken in the and `play` method will be subtracted from this credit. If the credit falls below 0, your agent will loose the game. The time credit is passed by as argument of both methods.

Competition rules

The contest will be played with the official Sarena rules for 2 players (no variant). Each agent will get a time credit of 20 minutes per match.

There are 38 groups, thus hopefully 38 different agents. These will be divided into 6 pools of 6/7 agents. All agents inside one pool will play twice against all other agents in the same pool. For each match, the winner gets 3 points. A draw game gives 1 point to the two opponents. In each pool, the two agents having the highest number of points will be selected. In case of ties, the slowest agent will be eliminated (we sum the time the agent took to return each action). As we need 16 agents for the final phase of the competition, the four fastest agents coming in third position in the pools will also be selected.

The selected 16 agents will participate to a best-of-three playoff. For each match, the winner is the agent winning two out of the three matches. A draw game is won by the faster agent. The selected agents will compete in matches as shown in Figure 8. The player labels displayed represent the pool from which the agent comes and its position in this pool. The `R` letter represents the agents which came third in the pool phase but were still selected. For each match, the winner goes to the next level, again as shown in Figure 8. A third place playoff will also be played to determine the third place between the losers of the semi-final.

The trace of each match will be stored and the most exciting or representative ones will be replayed during the debriefing sessions.

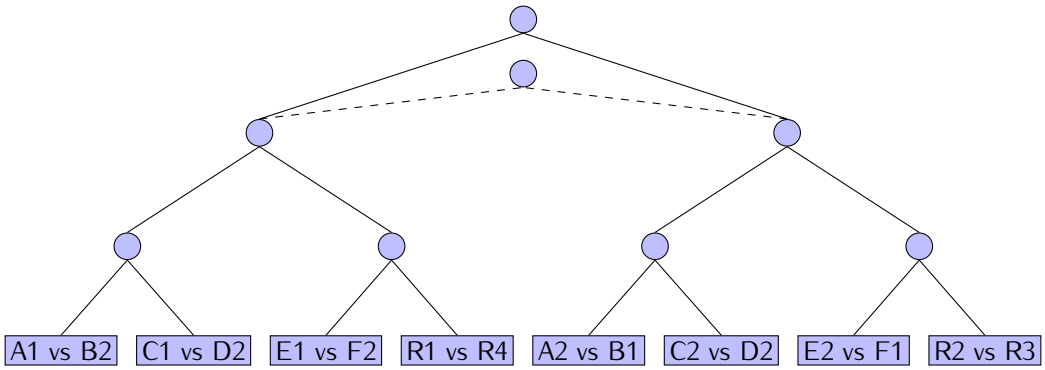


Figure 8: Playoffs

Evaluation

The mark you will get for this part will be based on

- the quality of your agent,
- the quality of your documentation,
- the originality of your approach,
- the design methodology (i.e., how did you chose your parameters?).

The position of your agent in the contest will give you bonus points.

Licensing

The provided classes are licensed under the General Public License[1] version 2. As such, your agents shall also have a GPL license. The working agents will then be put together and published on the website `http://becool.info.ucl.ac.be/aigames/sarena2012`. If you want, you may also give your agent a nice name.

We hope your agents will play exciting games and that they will outsmart the humans. We hope you will have these small amounts of luck needed to make good thoughts into great ideas.

[1] `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`