LINGI2261: Artificial Intelligence
Assignment 3: Project: Adversarial Search

Jean-Baptiste Mairy, Cyrille Dejemeppe, Yves Deville
October 2013

Guidelines

- This assignment is due on **Wednesday 27 November 2013 at 6pm**.
- 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.
- *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 Quoridor 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.

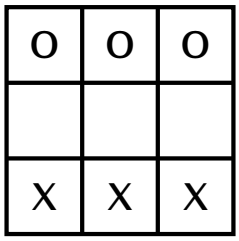
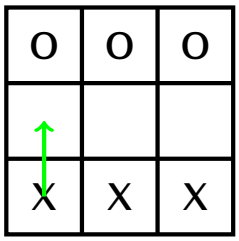


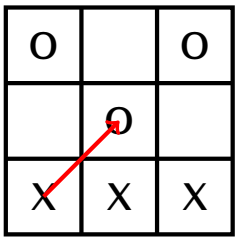
Figure 1: Initial state of the Scipion game.

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

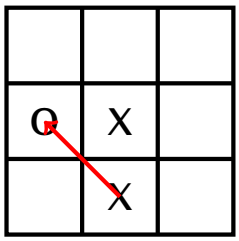


(b) Capturing an opponent

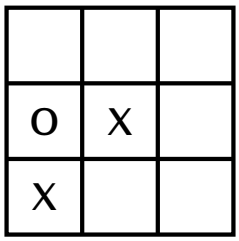
Figure 2: The two moves allowed in a Scipion game.

There are three ways to win a game:

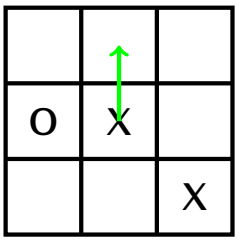
1. Capture all the chips of your opponent (as shown in Figure 3a).
2. Block your opponent in such a way he is unable to perform a move (as shown in Figure 3b).
3. Having one of your chip on the last row at the opposite of your starting row (as shown in Figure 3c).



(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.

We will use the MiniMax algorithm using the heuristic function

$$3X_2 + X_1 + X_C - (3O_2 + O_1 + O_C)$$

where X_2 = number of unblocked **X** chips in the second row,
X_1 = number of unblocked **X** chips in the starting row,
X_C = 1 if there is a **X** chip in the central tile; 0 otherwise,
O_2 = number of unblocked **O** chips in the second row,
O_1 = number of unblocked **O** chips in the starting row,
O_C = 1 if there is a **O** chip in the central tile; 0 otherwise

The initial state is the board configuration shown in Figure 1, with **X** to play. A chip is unblocked if it can be moved. The central tile is the only tile at the center of the board horizontally and vertically.

Questions

1. Draw the game tree for a depth of 2, i.e. one turn for each player. 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. Evaluate the value of the leaves using the heuristic function (on the figure you draw in question 1, you don't have to implement it!).
3. Using the MiniMax algorithm, find the value of the other nodes (no implementation needed, do it by hand).
4. Circle the move the root player should do (i.e. circle the move on the first level of the game tree that the minimax algorithm would choose).

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.

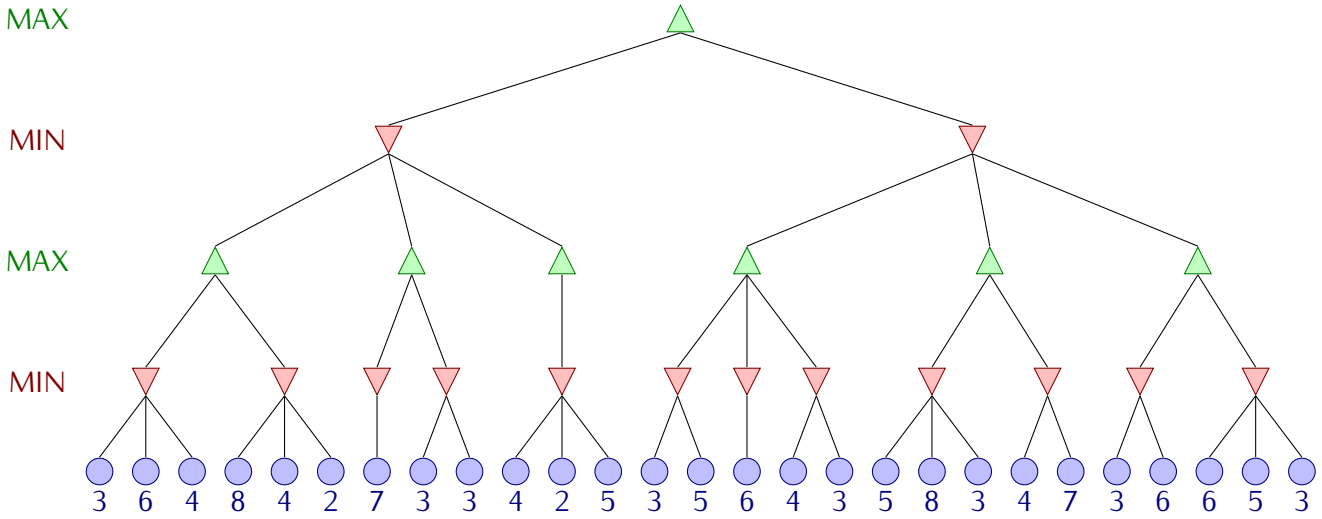


Figure 4: MiniMax

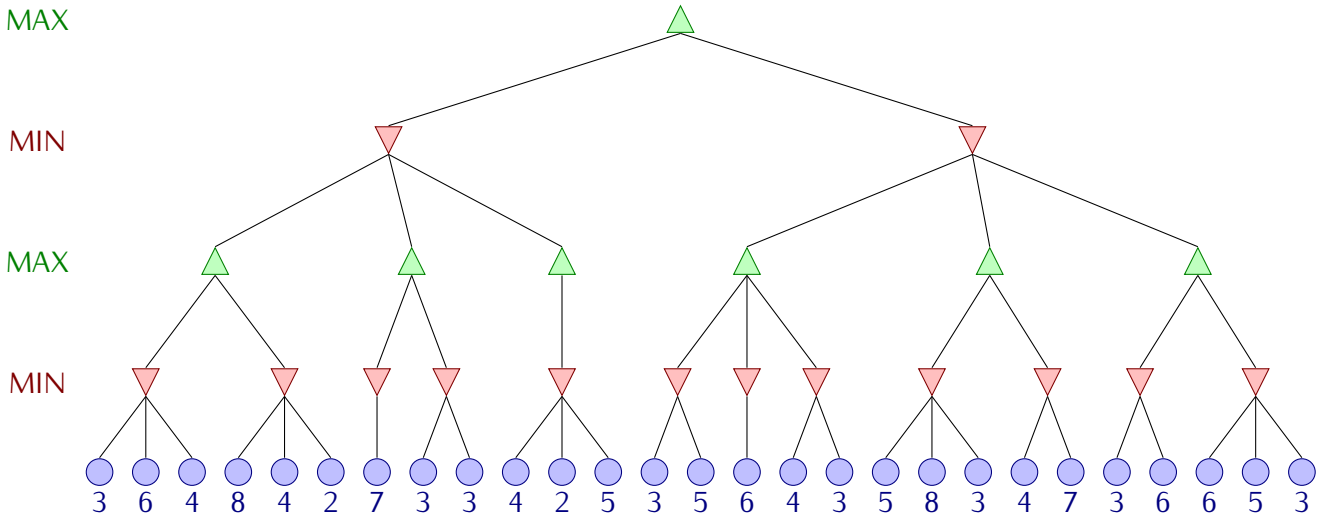


Figure 5: Alpha-Beta, left-to-right expansion

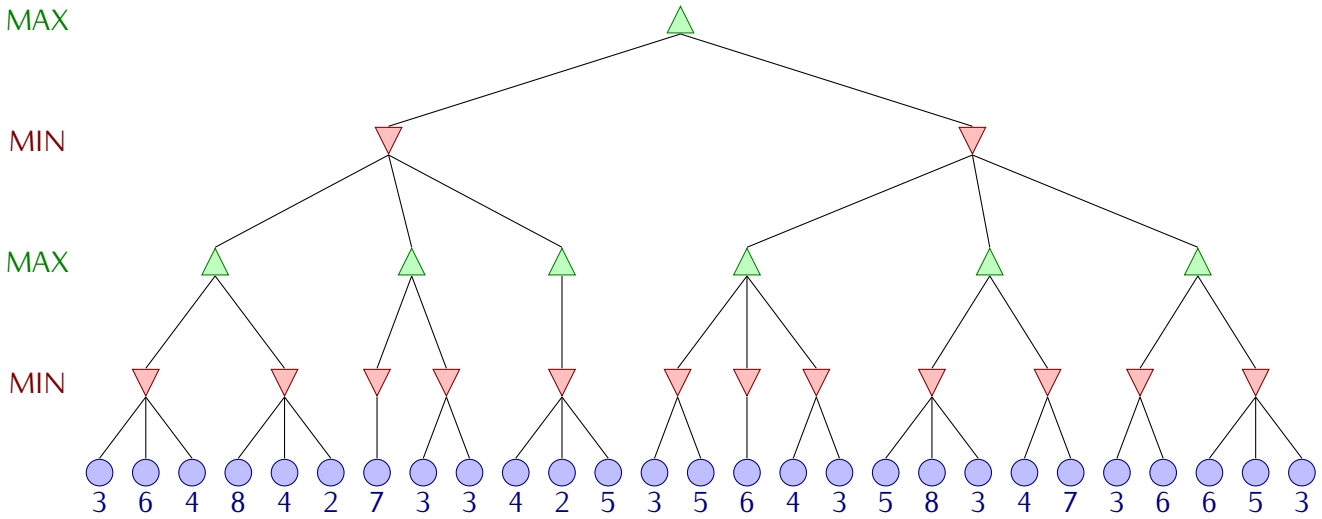


Figure 6: Alpha-Beta, right-to-left expansion

3 Quoridor (34 pts)

You have to imagine and implement an AI player to play the Quoridor game. A copy of the rules should be available on the iCampus course website.

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_legal_pawn_moves` in that order (i.e. for now we don't consider a player can place a wall on the board).
2. In order to avoid to explore the whole tree, we will limit the search up to a depth of 2. Of course, we have to ensure the search will also stop if the game is over. This can be done using the `Board.is_finished` method. Implement this in the `cutoff` method.
3. The evaluation function you will use is a very simple one. This function shall return 1 if the agent is in an advantageous situation, -1 if his opponent has the advantage and 0 if no player has an advantage on the other. To determine whether an agent has the advantage, use `Board.get_score`. It returns the difference between the length of the shortest path from the pawn of the player to his respective goal and the length of the shortest path of its opponent to its respective goal (in case of tie, it returns the difference between the number of walls remaining for the agent and the number of walls remaining for the opponent). If the output of `Board.get_score` is positive, return 1; if it is negative return -1 and if it is null, return 0.
   Be careful that the evaluation function must always be evaluated for the player that has to play a move at the root of the search tree (i.e. even if you are at an opponent node on the search tree).

3.2 Comparison of two evaluation functions (3 pts)

Let us now observe how the basic agent behaves when playing against itself.

Questions

4. Launch two instances of your basic agent and make them play against each other. Use the replay options to have a look at the two first moves of both payers. What do you observe?
5. Replace the basic evaluation function such that it returns directly the result of `Board.get_score` instead of -1, 0 or 1. Launch again two instances of your agent and make them play against each other. Does their respective behaviour radically change? What do you observe?
6. Explain why the respective behaviours of the instances of your agent changes when considering the two different evaluation function. To help you and illustrate your answer, draw and compare the search trees of depth 2 for the second move of the second player for both evaluation functions. Perform the MiniMax algorithm (not the alpha-beta) on these two trees, i.e. put a value to each node. Circle the move the root player should do.

Now you will begin to implement your final agent. In the next sections, the questions are there to help you to create a very smart agent. We highly recommend you answer these questions before coding you final agent since they will help you in this process.

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 length of the shortest path from the pawn of the agent to its respective goal and the length of the shortest path of its opponent to its respective goal (in case of tie, it returns the difference between the number of walls remaining for the agent and the number of walls remaining for the opponent). 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 (8 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 and a lower bound on the branching factor for a search tree on the Quoridor game. Justify your answer.
11. How does the branching factor evolve after a pawn has moved? How does the branching factor evolve after a wall has been placed? Explain.
12. From random games (at least 100), compute the average number of possible actions at each step of the game (end the game after 100 steps). A step represents the turn of one player; i.e. if both player play one move each, two steps have been performed. Plot the results in a graph. What do you observe?
13. Are there special moves in the game after which your branching factor changes radically? What are these moves? Explain.
14. Are all these successors necessary to be exhaustive (think about symmetry)? Why? If not, how will you consider only the necessary states?
15. If the number of successors is still too large, can you think of states that might be ignored, at the expense of loosing completeness?
16. Describe your successors function.

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, and so shall you). 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. Remember that the contest limits the amount of time your agent can explore the search tree.

Questions

17. The `cutoff` method receives an argument called `depth`. Explain precisely what is called the *depth* in the `minimax.py` implementation.
18. Explain why it might be useful (for the Quoridor contest) to cut off the search for another reason than the depth.
19. Describe your cut-off function.

3.6 Contest (10 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, as well as using CPU on other machines are prohibited. Your agent should run on a single core machine. 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., `quoridor.py`, `game.py`, `minimax.py`, etc.). Be careful that you can modify `minimax.py` and `quoridor.py` but not `game.py`.
  To test if your agent will run correctly and pass our automatic routines, launch a game directly in the a3 folder of your SVN repository.
- Your agent should be implemented in a file named `super_player.py`. It will be started with the following command:

  ```
  python3 super_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 TB'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 `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 Quoridor rules for 2 players we described in the Quoridor Instructions file provided on the iCampus course page. Each agent will get a time credit of 20 minutes per match.

There are 30 groups, thus hopefully 30 different agents. These will be divided into 6 pools of 5 agents. All agents inside one pool will play twice (once playing as first player and once playing as second player) against all other agents in the same pool. For each match, the winner gets 1 point. 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. To evaluate the speed of a

player, we divide the total time he played during his matches and we divide it by the number of actions he performed (i.e. the speed of a player is the average time he takes to play a move). 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-four playoff. 4 games form a match (twice playing as first player and twice playing as second player). For each match, the winner is the agent winning more games than his opponent. If both agents win 2 matches then the fastest player will be selected. The selected agents will compete in matches as shown in Figure 7. The player labels displayed represent the pool from which the agent comes (i.e. pool A, B, C, D, E or F) 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 7. 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.

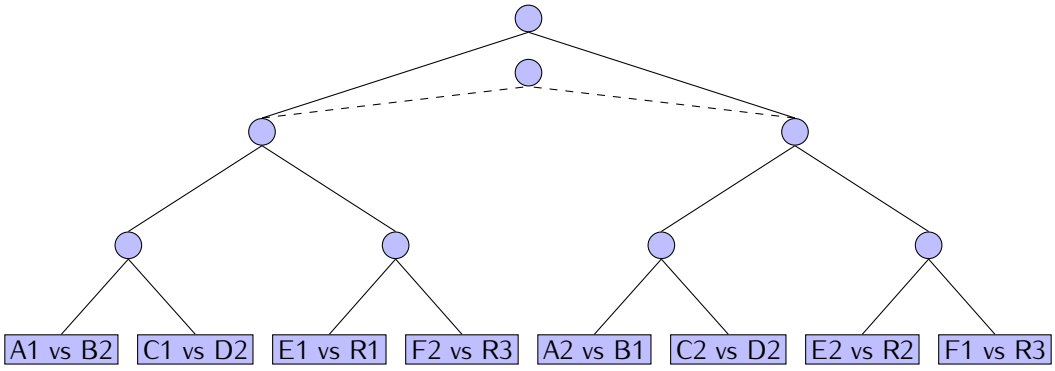


Figure 7: Playoffs

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/quoridor2013`. 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`